CodeBase 6.0™
User's Guide

The C Engine For Database Management
Clipper Compatible
dBASE Compatible
FoxPro Compatible

Sequiter Software Inc.

Contents

Introduction 5
How To Use This Manual 6
1 Database Concepts.....7
CodeBase File Format 8
2 C Programming 11
Memory Corruption 11
Structure Parameters.....13
3 DataBase Access.....17
Getting Started 17
CodeBase Components 17
An Example CodeBase Program.....19
Initializing The CODE4 Structure 21
Code fragment from SHOWDATA.C.....21
Opening Data Files 21
Closing Data Files.....22
Moving Between Records 22
Accessing Fields.....23
Retrieving A Field's Contents.....25
Creating Data Files 26
Adding New Records.....30
Assigning Field Values.....31
Removing Records 32
Data File Information 35
Advanced Topics 37
4 Indexing.....39
Indexes & Tags.....39
Index Expressions.....40
Creating An Index File 41
Maintaining Index Files 45
Opening Index Files.....45
Referencing Tags.....46
Selecting Tags.....47
Tag Filters 47
Unique Keys 50
Seeking 52
Group Files.....57
Bypassing Group Files.....57
Reindexing.....60
Advanced Topics 61
5 Queries And Relations 63
Relations.....63
Complex Relations.....67
Relation Types.....68
Creating Relations 71
Setting The Relation Type.....74
Moving Through The Composite Data File.....74
Queries And Sorting.....75
Accessing The Query Set.....78
Queries On Multi-Layered Relation Sets 79

Lookups On Relations....80
Iterating Through The Relations....82
6 Query Optimization....83
What is Query Optimization....83
When is Query Optimiztion used....84
How To Use Query Optimization....86
7 Date Functions....87
Date Pictures....87
Date Formats....87
code fragment from DATE1.C....89
code fragment from DATE1.C....90
code fragment from DATE1.C....90
Testing For Invalid Dates....90
Other Date Functions....90
code fragment from DATE1.C....91
8 Linked Lists....93
The Linked List....93
CodeBase Linked Lists....94
Dynamic Allocation....97
Stacks And Queues....99
9 Memory Optimizations....101
Using Memory Optimizations....101
Memory Requirements....103
When To Use Memory Optimization....104
10 Multi-User Applications....107
Locking....107
Creating Multi-User Applications....108
Common Multi-User Tasks....110
Multi-User Optimizations....113
Avoiding Deadlock....116
Exclusive Access....117
Read Only Mode....118
Code fragment from MULTI.C....118
Lock Attempts....118
Automatic Record Locking....119
Enforced Locking....119
11 Performance Tips....121
Memory Optimization....121
Memory Requirements....121
Observing the performance improvement....121
Functions that can reduce performance....121
Locking....121
Appending....122
Time Consuming Functions....122
Queries and Relations....122
General Tips....123
12 Transaction Processing....125
Transactions....125
Logging in the Stand-Alone Case....128
Logging in the Client-Server case....130
Locking....130
Index....133

Introduction

CodeBase is a high performance database engine for C, C++, Delphi and Visual Basic programmers.

CodeBase is compatible with dBASE IV, FoxPro and Clipper file formats. CodeBase allows you to create, access or change their data, index or memo files. Applications written in CodeBase can seamlessly share files with concurrently running dBASE IV, FoxPro and Clipper applications.

CodeBase has four different interfaces depending on programming language. CodeBase supports the C++, C, Visual Basic languages and Pascal under Delphi. In addition, these interfaces can be used under a variety of operating systems, including DOS, Windows 3.1/95/NT or OS/2 and UNIX, depending on the language.

CodeBase is scalable to your needs, which means that it can be used in a stand-alone, multi-user or client-server configuration.

CodeBase has many useful features including a full set of query and relation functions. These functions use Query Optimization to greatly reduce the time it takes to perform queries and generate reports. CodeBase also has logging and transaction capabilities, which allow you to control the integrity of a database easily.

This *User's Guide* discusses the CodeBase C API (Application Program Interface). The CodeBase C API allows the programmer to write applications using CodeBase in C. It is necessary to have a C compiler and some knowledge of the C programming language.

Refer to the *Getting Started* booklet for information on installing CodeBase, running the example programs and contacting Sequiter Software.

How To Use This Manual

This manual was designed for both novice and veteran database programmers. The *CodeBase 6.0 Getting Started* booklet details how to compile and link an example CodeBase application. This booklet is recommend reading for all users of CodeBase. The "Database Concepts" chapter of this guide discusses database terminology. Even if you are an expert database programmer, you should at least skim this chapter because some of the terms, such as tags and indexes, may be used differently than you expect. The remaining chapters deal with the basics of writing database applications with the CodeBase library. For your convenience, all example programs illustrated in this manual can be found on disk.

Manual Conventions

This manual uses several conventions to distinguish between the various language constructs. These conventions are listed below:

CodeBase classes, functions, structures, constants, and defines are always shown highlighted as follows:

e.g. **d4create , CODE4, r4eof, E4DEBUG**

dBASE expressions are always contained by two double quotes (" ").

e.g. " 'Hello There ' + FIRST_NAME"

You should note that the containing quotes are part of the C/C++ syntax rather than part of the dBASE expression.

dBASE functions are displayed in upper case and are denoted by a trailing set of parenthesis.

e.g. UPPER() , STR(), DELETED()

C++ language functions and constructs are shown in bold typeface.

e.g. **main , for, scanf, cout, memcpy.**

C++ data types are shown bolded and encapsulated in parenthesis.

e.g. **(int), (char *),(long)**

1 Database Concepts

If you are not familiar with the concept of a *database*, simply think of it as a collection of information which has been organized in a logical manner. The most common example of a database is a telephone directory. It contains the names, phone numbers, and addresses of thousands of people. Each listing in the phone book corresponds to one *record*, and each piece of information in the record corresponds to one *field*. The phone book excerpt below illustrates this concept.

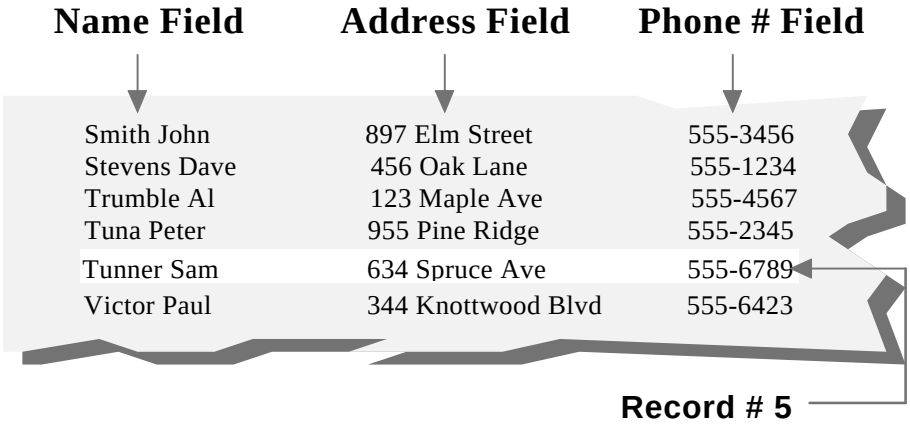


Figure 1.1 Phone Book Example

As shown in this example, each *data file* (also known as a table) is a collection of one or more fields (also known as a tuple). Each of these fields has a set of characteristics that determine the size and type of data to be stored. Collectively, these field descriptions make up the structure of your data file.

CodeBase File Format

CodeBase uses the industry standard .DBF data files. This standard allows for several types of fixed width fields and one type of variable length field.

Fields

The **.DBF** standard uses four attributes to describe each field. These attributes are Name, Type, Length, and Decimal. They are listed below. For detailed information on field attributes, please refer to the **FIELD4INFO** structure as described in the CodeBase *Reference Guide* under **d4create**.

- **Name**: This simply refers to the name that will be used to identify the field. Each field name can be a maximum of 10 characters, and each field name must be unique within a data file and must consist of alphanumeric or underscore characters. The first letter of the name must be a letter.
- **Type**: The type of the field determines what kind of information should be stored in the field. There are eight different types that can be specified for any given field. They are Character, Numeric, Floating Point, Date, Logical, Memo, Binary and General.
- **Length**: This attribute refers to the number of characters or digits that can be stored in the field.
- **Decimal**: This attribute applies only to Numeric and Floating Point fields. It specifies the number of digits after the decimal point.

Records

A record consists of one instance of every field, and has a unique *record number* and a *deletion flag* associated with it. The record number indicates the physical position in the data file while the deletion flag is used during the deletion process. The deletion flag is set to true (non-zero) to indicate that the record should be removed from the data file when the data file is packed.

Tags

A *tag* determines the ordering in which the records in a data file are presented. The tag ordering does not affect the physical ordering of the records in the data file, only the order in which they can be accessed. The information that this ordering is based upon is called the *index key*.

In the phone book example, the index key is based on the Name field and the actual index keys for records 1 to 3 are "Smith John", "Stevens Dave", and "Trumble Al". If you wanted to display the phone information sorted by the phone number, you would create a tag based on the phone number field.

Indexes

An *index* is a file containing the sorted index keys for one or more tags. There are several formats of index files which CodeBase supports.

- "**.MDX**" (dBASE IV)
- "**.NTX**" (Clipper)
- "**.CDX**" (FoxPro)

The "**.CDX**" and "**.MDX**" allow you to have multiple tags in each index file, and to have *production indexes*. A production index is an index file that is opened automatically when the associated data file is opened. The "**.NTX**" format limits you to one tag per index file, and does not allow for production indexes. CodeBase does, however, provide this functionality through the use of group files. Refer to the "Indexing" chapter for more details on index file formats.

Filters

Filters are used to obtain a subset of the available data file records. The subset is based on true/false conditions calculated from data file field information. Only records that pass through the filter have entries in the tag. This allows for fast access to a data subset. Refer to the "Indexing" chapter for details on using filters.

Relations

A relation is a connection between two or more data files. A relation determines how records in related data files can be found from a record in the current data file. Refer to the "Queries And Relations" chapter for details on relations.

2 C Programming

The C programming languages are perhaps the most popular and flexible languages currently in use. While it is not the purpose of this manual to teach you how to program in C, this chapter deals with some of the common difficulties encountered by C programmers when first using the CodeBase library.

Although CodeBase is straight forward and easy to use, it does require a basic understanding of the C language, including the use of structures and pointers. The following sections illustrate some common problems and their remedies.

Memory Corruption

One of the main features of C is the flexibility in using memory. Unfortunately this flexibility also lets you change memory that you shouldn't, causing memory corruption. Memory corruption, due to programming oversights, causes the most difficult bugs in C programs.

The sizeof Operator

Many situations that cause memory corruption can be avoided by using the C language's **sizeof** operator. One of the biggest culprits of memory corruption occurs when data is written beyond the end of a string character array. The following code fragment illustrates this error:

INCORRECT

```
char temp[8];
strcpy(temp, "This string is longer than 8");
```

This sort of problem can be avoided by using the **sizeof** operator and a string copy function that copies a maximum number of characters. The CodeBase function **u4ncpy** is an excellent choice because it also ensures that the destination string is null terminated under all circumstances.

CORRECT

```
char temp[8];
u4ncpy(temp, "This string is longer than 8", sizeof(temp));
```

The **sizeof** operator may also be used with functions like **memset**.

```
char temp[8];
memset (temp, 'X', sizeof(temp));
```

Memory Scoping

Another common mistake is dealing with memory scoping. In the next example, the programmer wants to return a string that has been filled with data.

INCORRECT

```
#include "d4all.hpp"

char *SetString( void )
{
    char buf[ 15 ];
    u4ncpy( buf, "HELLO WORLD!", sizeof(buf) ) ;
    return buf ;
}

void main( void )
{
    char *Message;
    Message = SetString( );

    ... more stuff ...
}
```

This program may appear to work at first, but the memory for the variable *buf* is automatically deallocated after the *SetString* function is exited. Since the memory to which *Message* points is free, it can be reused by the system at any time. Consequently, using the information pointed to by *Message* is an error.

To avoid this situation, several correct methods can be used. The first uses static memory. Static memory is not deallocated while the program is running, and the same memory is used every time the function is called.

CORRECT

```
#include "d4all.hpp"

char *SetString( void )
{
    static char buf[ 15 ] ;
    u4ncpy( buf, "HELLO WORLD!", sizeof(buf)) ;
    return buf ;
}

void main( void )
{
    char *Message;
    Message = SetString();

    ... more stuff ...
}
```

The next example accomplishes the same result by allocating the memory in the calling function by declaring the variable *message* in **main**.

CORRECT

```
#include "d4all.hpp"

void SetString( char *buf, int size)
{
    u4ncpy( buf, "HELLO WORLD!", size);
}

void main( void )
{
    char Messa ge[ 15 ];
    SetString( Message, sizeof( Message ) );

    ... more stuff ...
}
```

Finally you can dynamically allocate the memory. Remember to deallocate it when you are done.

CORRECT

```
char *SetString( void )
{
    char *buf ;

    buf = (char *) u4alloc( 15 * sizeof(char)) ;;
    if (buf)
    {
        u4ncpy( buf, "HELLO WORLD!", sizeof(buf)) ;
        return buf ;
    }
}
void main( void )
{
    char *Message;
     Message = SetString( );

    ... more stuff . . .

    u4free( Message ) ;
}
```

Structure Parameters

Many of the common problems encountered by novice C users deal with the passing and returning of structure parameters.

Structure passing

Consider this example. The programmer wishes to pass a **CODE4** structure to a function that initializes several of its flags.

INCORRECT

```
#include "d4all.h"

void InitFlags(CODE4 codeBase)
{
    /*set some flags */
    codeBase.errOpen = 0 ;
    codeBase.safety = 0 ;
}

void main( void )
{
    CODE4 cb;

    code4init( &cb );
    /* code4init sets cb.errOpen and cb.safety to 1*/

    InitFlags( cb ) ;
    /*cb.errOpen and cb.safety are still set to 1 !!*/
}
```

When you pass a variable (in this case a structure) as a parameter, the function actually receives a copy of that variable. Therefore any changes which are made to the structure members in the function, are lost when the function returns.

This causes several problems. First of all, memory is wasted by storing a copy of the structure members. Any changes that are made to the structure members are lost and more importantly, many CodeBase structures (such as the **CODE4** structure) maintain several sets of linked lists that can be corrupted by accessing them through a copy.

The correct way to pass a structure to a function, is to pass it in as a pointer. This is illustrated in the corrected version:

CORRECT

```
#include "d4all.h"

void InitFlags(CODE4 *codeBase)
{
    /*set some flags */
    codeBase->errOpen = 0 ;
    codeBase->safety = 0 ;
}

void main( void )
{
    CODE4 cb ;

    code4init ;
    /* code4init sets cb.errOpen and cb.safety to 1*/

    InitFlags( &cb ) ;
    /* cb.errOpen and cb.safety are now set to 0 */
}
```

Pointer Initialization

The next example demonstrates another common difficulty: pointer initialization.

INCORRECT

```
#include "d4all.h"

CODE4 codeBase ;

void OpenDatafile( DATA4 *Datafile )
{
    Datafile = d4open( &codeBase, "TEST.DBF") ;
    /* a data file is opened a nd Datafile is assigned a value */

}

void main( void )
{
    DATA4 *data = 0 ;
    OpenDatafile( data ) ;
    d4top( data ) ;
    /* Error: data still equals 0 */
    code4close( &codeBase ) ;
}
```

The problem with this program is similar to the problem in the previous example. As in example 1, a copy of the parameter (in this case a pointer) is passed to the function. The reason this causes problems is that we are changing a copy of the pointer, not the actual pointer.

When *data* is passed the *OpenDatafile* function, it contains an address of 0. A copy of this pointer is created and is used inside the function. When the **d4open** is called, it assigns a new address to the copy. When the function is exited, the copy is discarded and back in main *data* still has a value of 0.

Two examples in which the above problem is corrected are listed below.

In the first example, a pointer to variable *data* is passed to *OpenDatafile*. Consequently, *OpenDatafile* is able to modify the contents of variable *data* which avoids the problem of modifying a copy.

CORRECT

```
#include "d4all.h"

CODE4 codeBase ;

void OpenDatafile( DATA4 **Datafile )
{
    *Datafile = d4open( *codeBase, "TEST.DBF" ) ;
    /* a data file is opened and Datafile is assigned a value */
}

void main ( void )
{
    DATA4 *data = 0 ;

    code4init( &codeBase ) ;
    OpenDatafile( &data ) ;
    /* data now has a valid address */

    d4top( data ) ;
    code4close( &codeBase ) ;
}
```

In this next example, the pointer is passed as a return value, which is then assigned directly to variable *data*. This also effectively works around the problem of modifying a copy.

```
#include "d4all.h"

CODE4 codeBase ;

DATA4 *OpenDatafile( void )
{
    DATA4 *Datafile = 0 ;
    Datafile = d4open( &codeBase, "TEST.DBF" ) ;
    /* a data file is opened and Datafile is assigned a value */

    return (Datafile) ;
}

void main( void )
{
    DATA4 *data ;

    code4init( &codeBase ) ;
    data = OpenDatafile( ) ;
    /* data now has a valid address */

    d4top( data ) ;
    code4close( &codeBase ) ;
}
```

You should note that you have to take these special precautions only when you are modifying the actual address that the pointer contains. The following code fragment is perfectly acceptable because the pointer is not being modified, just what it points to:

```
void InitFlags( CODE4 *codeBase )
{
    /* set some flags */
    codeBase->errCreate = 0 ;
    codeBase->safety = 0 ;
}
```

3 DataBase Access

Now that you have been introduced to the concepts of database management, you're ready to start programming some simple database programs using C. This chapter take you through the details of manipulating the data files by explaining how to create and open data files, store and retrieve data, and how records are added and deleted.

Getting Started

To make the task of learning CodeBase easier, all the examples in this manual only use the standard C input and output functions. These programs will run with any ANSI C/C++ compiler. Since the application programming interface (API) of CodeBase is the same for all environments, you can simply paste the example CodeBase functions into your application.

To run any of the tutorial programs, compile the program and link it to the CodeBase library according to the instructions in the *Getting Started* Booklet.

CodeBase Components

In addition to the elements of standard C programs, every CodeBase program will contain CodeBase functions and structures.

CodeBase Functions

All data manipulation is performed by CodeBase functions that are identified by the format of their names. These names are lowercase starting with one to six letters followed by the number four. The leading letters indicate the module of the function while the remaining portion of the name describes its purpose. For example, functions that manipulate data files start with **d4,** while all the date functions start with **date4.**

CodeBase Structures

CodeBase structures are used for storing information and for referencing objects such as data files and fields. CodeBase structures follow the same naming conventions as CodeBase functions except that they are all upper case.

CodeBase Constants

In addition to CodeBase structures and functions, you can also use CodeBase constants. Most constants are integers which are used for return values and error codes. These constants have names which usually start with **r4,** or **e4**. Some examples include **r4success**, **r4locked**, **e4memory**, and **e4open**. These constants are listed in more detail in "Appendix A: Error Codes" and "Appendix B: Return Codes" of the CodeBase *Reference Guide*.

Overview

The next section introduces some the of the important CodeBase structures and examines how they are related to data files.

The **CODE4** Structure

Foremost of the CodeBase structures is the **CODE4**. This structure contains settings and information used by most of the other CodeBase functions.

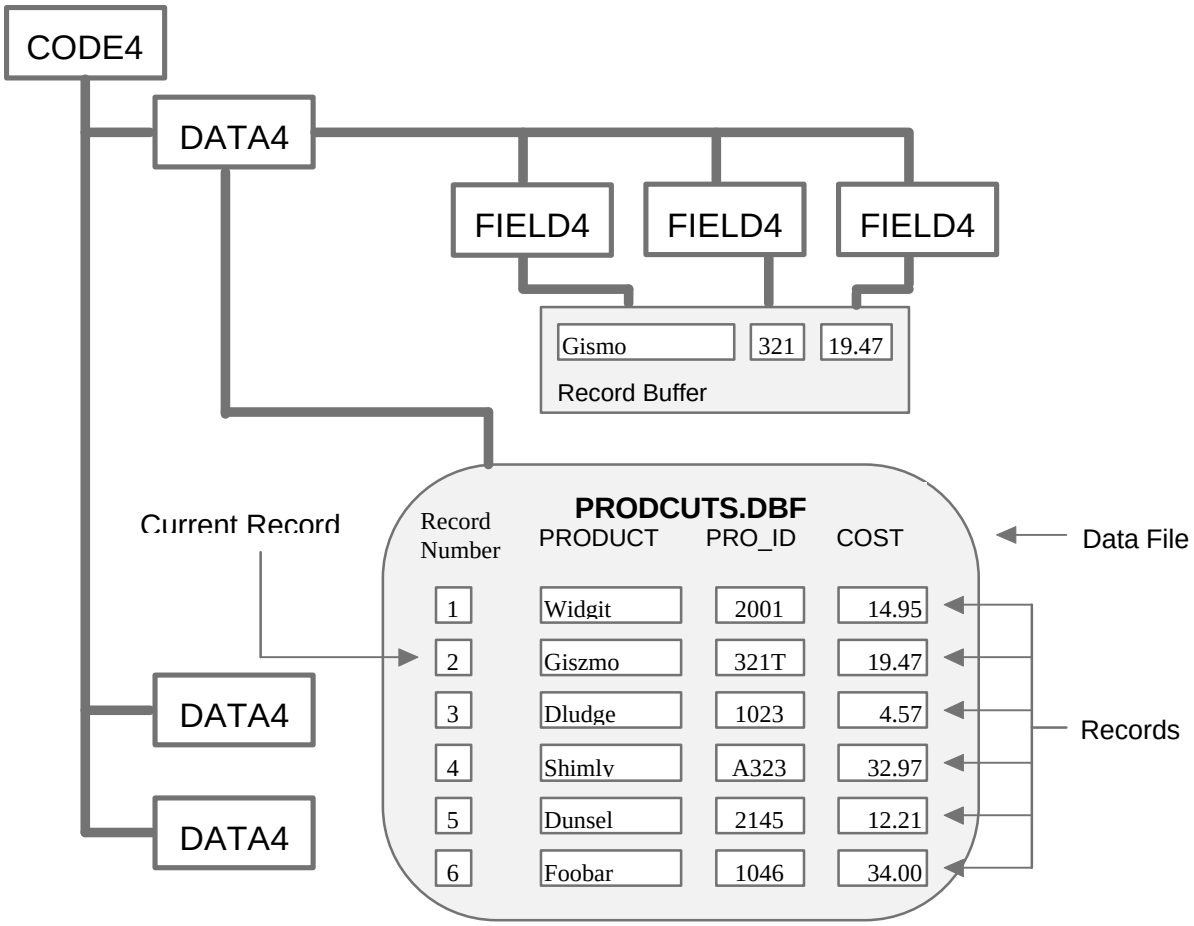


Figure 3.1 CodeBase Structures

Note

Most applications only require one **CODE4** structure. If you are using more than one **CODE4** structure, you will most likely be wasting memory resources. Exceptions to this rule are discussed later in this chapter.

The **DATA4** structure

The **DATA4** structure is used to reference a particular data file. When a data file is opened, the information for the **DATA4** structure is allocated and stored in the **CODE4** structure. When you call a function that operates on a data file, you specify which data file you want by passing the function a pointer to its **DATA4** structure. (For convenience, the terms "pointer to its **DATA4** structure" and "**DATA4** pointer" will be used interchangeably.) Note that more than one **DATA4** structure can reference the same open data file. However, since the information for the structure is stored within the **CODE4** structure, each new structure constructed with **code4data** contains the same information, including the same current record and optimization settings.

The record buffer

Each data file has a memory area associated with it called the *record buffer,* which is used to store one record from the data file. The record that

is stored in the record buffer is called the *current record*.

Changes made to the record buffer using the field functions are automatically written to the data file before any new current record is read.

The **FIELD4** structure

A **FIELD4** structure is used to reference a particular field in the record buffer. When the data file is opened, a **FIELD4** structure is automatically allocated for each of its fields. When you call any function that uses a specific field of the record buffer, you specify which field by passing a **FIELD4** pointer.

An Example CodeBase Program

At this point, you are ready to write your first useful program that uses the CodeBase library. This program will display all of the records in any data file whose name you provide as a command line parameter.

PROGRAM SHOWDATA.C

```
#include "D4ALL.h"

int main(int argc,char *argv[])
{
    CODE4    codeBase;
    DATA4    *dataFile = NULL;
    FIELD4   *field = NULL;

    int      rc,j,numFields;

    if(argc != 2)
    {
        printf(" USAGE: SHOWDATA <FILENAME.DBF> \n");
        exit(0);
    }

    code4init(&codeBase);

    dataFile = d4open(&c odeBase,argv[1]);
    error4exitTest(&codeBase);

    numFields = d4numFields(dataFile);
    for(rc = d4top(dataFile);rc == r4success
                        ;rc = d4skip(dataFile, 1L))
    {
        for(j = 1;j <= numFields;j ++)
        {
            field = d4fieldJ(dataFile,j);
            printf("%s ", f4memoStr(field));
        }
        printf("\n");
    }

    d4close(dataFile);
    code4initUndo(&codeBase);
    return 0;
}
```

Explanation :

The next section contains a brief explanation of the SHOWDATA.C program, which is followed by a more detailed explanation.

Include Files

This program, like almost every program you write using the CodeBase library includes the "D4ALL.H" include file. This file contains the definitions and includes the type definitions for the CodeBase library. You will need to include this file in any module which uses CodeBase functions or structures.

Strucutures and Variables

In addition to some variables of standard types, this program uses three CodeBase data structures: **CODE4**, **DATA4** and **FIELD4**.

```
CODE4    codeBase;
DATA4    *dataFile = NULL;
FIELD4   *field = NULL;

int      rc,j,numFields;
```

You should be aware that variable *codeBase* is an actual instance of a structure, while *dataFile* and *field* are pointers to structures.

Note

It is a good idea (although not required) to initially set any CodeBase structure pointers to null. Since many CodeBase functions check for null pointers, you will be notified if you try to use an uninitialized pointer.

The number of arguments is checked and the program is terminated if an incorrect number was provided.

```
if(argc != 2)
{
    printf(" USAGE: SHOWDATA <FILENAME.DBF> \n");
    exit(0);
}
```

The **CODE4** structure is initialized by a call to **code4init**.

```
code4init(&codeBase);
```

Next the **d4open** function is used to open the data file specified by the command line argument. The **error4exitTest** function is called to terminate the program if an error has occurred.

```
dataFile = d4open( &codeBase, argv[1] ) ;
error4exitTest( &codeBase ) ;
```

The number of fields in the data file is obtained by a call to **d4numFields**.

```
numFields = d4numFields( dataFile ) ;
```

Next, a **for** loop is used to load each record into the record buffer. This **for** loop consists of the following operations: a call to **d4top** to load the first record, a call to **d4skip** to load the next record, and a check to see if **d4top** or **d4skip** have reached the end of the file.

```
for(rc = d4top(dataFile);rc == r4success ;rc = d4skip(dataFile, 1L))
{
```

Inside this **for** loop, a second **for** loop is used to iterate through the fields by incrementing a counter from one to the total number of fields in the data file.

```
    for( int j = 1; j <= numFields; j++ )
    {
```

This counter is used in conjunction with the **d4fieldJ** function to obtain a pointer to a field's **FIELD4** structure. A pointer to the contents of that field is returned from a call to **f4memoStr**.

```
    field = d4fieldJ(dataFile,j);
    printf("%s ", f4memoStr(field));
```

```
    }
}
```

The data file is closed by the **d4close** function and the program is exited.

```
    d4close(dataFile);
    code4initUndo(&codeBase);
    return 0;
}
```

Initializing The CODE4 Structure

Because the **CODE4** structure contains random data when the program starts, the **CODE4** structure must be initialized before any CodeBase functions can be used. This is accomplished by passing a pointer to the structure to the **code4init** function.

Code fragment from SHOWDATA.C

```
code4init(&codeBase);
```

When the **CODE4** structure is initialized, all of the flags and member variables are set to their default values. The **code4init** function should normally only be called once in the application.

After **code4init** has been called, you can then change the value of the **CODE4** structure's flags.

Opening Data Files

Only after a **CODE4** structure is initialized, can data files be opened with **d4open**.

Code fragment from SHOWDATA.C

```
dataFile = d4open( &codeBase, argv[1]) ;
error4exitTest( &codeBase ) ;
```

d4open accepts a string containing the path and file name of a data file. If the specified file name does not contain an extension, the default extension of "**.DBF**" is used. If no path is specified, the current directory is searched.

Obtaining a **DATA4** pointer

If the file is located, it is opened and the address to an internal **DATA4** structure is returned. This address is used as a reference to the data file.

If the file does not exist, or if CodeBase detects any other error, an error message is displayed and a null pointer is returned.

Note

Almost every function that starts with **d4** takes a **DATA4** pointer as a parameter. This pointer specifies which data file the function should operate on.

Checking for errors

error4exitTest checks for any such errors and terminates the program if one has occurred. Alternatively, if you did not want to exit the program, you can check to see if *dataFile* is equal to null.

After the data file has been successfully opened, the current record number is set to '-1' to indicate that a record has not yet been read into the record buffer.

Closing Data Files

Before your application completes, it is important that any open data files are closed. This ensures that any changes to the data file are updated correctly.

There are two functions that you can use. They are **d4close** and **code4close**. The **d4close** function closes the data file whose **DATA4** pointer was provided as a parameter. The function also closes any index files (see the "Indexing" chapter) or memo files associated with the data file.

Code fragment from SHOWDATA.C

```
d4close(dataFile);
```

The function **code4close** closes all open data, index, and memo files.

Moving Between Records

CodeBase provides ten functions for moving between records in the data file: **d4top**, **d4bottom**, **d4skip**, **d4go, d4seek, d4seekDouble, d4seekN, d4seekNext, d4seekNextDouble** and **d4seekNextN**. These functions change the current record by loading a new record into the record buffer.

- The **d4top** function sets the current record to the first record of the data file. The **d4bottom** performs a similar function for the last record.
- The **d4skip** function moves a specified number of records above or below the current record. This function must have a current record in order to function correctly.
- The **d4go** function loads the record buffer with the record whose record number was provided as a parameter.
- The **d4seek, d4seekDouble and d4seekN** functions locate the first record in a sorted order that matches a search key. These functions are described in detail in the "Indexing" chapter.
- The **d4seekNext, d4seekNextDouble** and **d4seekNextN** functions are similar to the **d4seek** functions except that they locate the next record in a sorted order that matches a search key. These functions are described in detail in the "Indexing" chapter.

Scanning Through The Records

A common task in database programming is to display a list of records. To accomplish this, the SHOWDATA.C program loads each record into the record buffer so its contents can be displayed.

This is precisely what the following **for** loop does.

Code fragment from SHOWDATA.C

```
for( int rc = d4top(dataFile); rc == r4success ; rc =
                                   d4skip( dataFile, 1L ))
{
```

d4top

In the initialization section of the **for** loop, **d4top** is called. This function causes the first record in the data file to be loaded into the record buffer. If the function is successful, it returns a value of **r4success.**

d4skip

At the end of each iteration through the **for** loop, **d4skip** is called. **d4skip** allows you to skip either forwards or backwards through the data file. The numeric argument is used to specify how many records are skipped. If the number is positive, you move towards the end of the data file; if it is negative you move back towards the top. In this example the value is '1L' (the 'L' ensures that this is a (long) value), which indicates that the next record should be loaded. The **d4skip** function also returns a value of **r4success** when it executes successfully.

The **for** loop is terminated when *rc* is no longer equal to **r4success**. This happens when either **d4skip** or **d4top** encounters an error or the end of the data file.

Accessing Fields

Before any information can be stored to or retrieved from a field, you must first obtain a pointer to its **FIELD4** structure. There are two ways to accomplish this: the first involves knowing the field's position in the data file, and the second requires that you already know the field's name.

PROGRAM CUSTLIST.C

This program lists the customer information contained in the DATA1.DBF data file.

```
#include "D4ALL.h"

int main( void )
{
    CODE4    codeBase;
    DATA4    *dataFile = 0;
    FIELD4   *fName           = 0,
              *lName          = 0,
              *address        = 0,
              *age            = 0,
              *birthDate      = 0,
              *married        = 0,
              *amount         = 0,
              *comment        = 0;

    int      rc,j,ageValue;
    double   amountValue;
    char     nameStr[25];
    char     addressStr[20];
    char     dateStr[9];
    char     marriedStr[2];
    const char     *commentStr;

    code4init(&codeBase);

    dataFile = d4open(&codeBase,"DATA1.DBF");
    error4exitTest(&codeBase);

    fName        = d4field(dataFile,"F_NAME");
    lName        = d4field(dataFile,"L_NAME");
    address  = d4field(dataFile,"ADDRESS");
    age      = d4field(dataFile,"AGE");
    birthDate    = d4field(dataFile,"BIRTH_DATE");
    married  = d4field(dataFile,"MARRIED");
    amount       = d4field(dataFile,"AMOUNT");
```

```
    comment  = d4field(dataFile,"COMMENT");

    for(rc = d4top(dataFile);rc == r4success;rc = d4skip(dataFile, 1L))
    {
        u4ncpy(nameStr,f4str(fName),sizeof(nameStr));
        u4ncpy(addressStr,f4str(address),sizeof(addressStr));

        ageValue = f4int(age);
        amountValue = f4double(amount);

        u4ncpy(dateStr,f4s tr(birthDate),sizeof(dateStr));
        u4ncpy(marriedStr,f4str(married),sizeof(marriedStr));

        commentStr = f4memoStr(comment);

        printf("------------------------------\n");
        printf("Name     : %20s\n",nameStr);
        printf("Address  : %15s\n",addressStr);
        printf("Age : %3d   Married : %1s\n",ageValue,marriedStr);
        printf("Comment: %s\n",commentStr);
        printf("Purchased this year:$%5.2lf \n",amountValue);
        printf("\n");
    }
    d4close(dataFile);
    code4initUndo( &codeBase ) ;
    return 0;
}
```

Referencing By Field Number

Each field in the data file has a unique *field number*. Field numbers range from one to the total number of fields in the data file. This value denotes the field's physical order in the record. The **d4fieldJ** function uses a field's field number to return a pointer to the field's **FIELD4** structure.

Iterating through the fields

Iterating through the fields in a record is a common use of this method of field referencing. Before you can do this, **d4numFields** is called to determine the number of fields in the data file.

Code fragment from SHOWDATA.C

```
   numFields = d4numFields( dataFile ) ;
```

A **for** loop is then used to iterate through each field to generically obtain a **FIELD4** pointer.

```
for( int j = 1; j <= numFields; j++ )
{
    field = d4fieldJ( dataFile,  j ) ;
```

Referencing By Field Name

If you know ahead of time what the names of the fields are, you can obtain a field's **FIELD4** pointer using its name. This is the method used in the CUSTLIST.C program.

The **d4field** function returns a pointer to a **FIELD4** structure of the field whose name is specified. This is demonstrated in the following section of code.

Code fragment from CUSTLIST.C

```
FIELD4    *fName, *lName, *address, *age, *birthDate,
       *married, *amount, *comment;

... data file is opened ....

fName       = d4field(dataFile,"F_NAME");
lName       = d4field(dataFile,"L_NAME");
address  = d4field(dataFile,"ADDRESS");
age      = d4field(dataFile,"AGE");
birthDate  = d4field(dataFile,"BIRTH_DATE");
married  = d4field(dataFile,"MARRIED");
amount      = d4field(dataFile,"AMOUNT");
comment  = d4field(dataFile,"COMMENT");
```

In this code fragment, a **FIELD4** pointer is obtained and stored in separate variables for each field of the record.

When the data file is closed, all **FIELD4** pointers for the data file become invalid.

Retrieving A Field's Contents

The C language has many different data types such as **(int), (char), (double),** and **(long)**. As a result, the CodeBase provides a number of functions that retrieve the contents of strings and fields and automatically perform any necessary conversions to the C data type. All the field functions accept a **FIELD4** pointer that specifies which field is to be operated upon.

Retrieving field contents as Strings

If you need to access the field's contents in the form of a null terminated string, you can use **f4str** or **f4memoStr**. They both return **(char *)** pointers to an internal buffer which contains a copy of the field's contents.

Note

Each time a **f4str** or **f4memoStr** function is called, the internal CodeBase buffer is overlaid with data from the new field. Consequently, if the field's value needs to be saved, it is necessary for the application to make a copy of the field's contents.

The **f4str** function can be used on any type of field except Memo fields. **f4memoStr** works on all types of fields including Memo fields. The reason for having two almost identical functions is that Memo fields require special handling. If you know a field is not a Memo field, it is appropriate to use the **f4str** function instead.

To enable the program to work on any type of field, the SHOWDATA.C program uses **f4memoStr** function.

Code fragment from SHOWDATA.C

```
printf("%s ",f4memoStr(field));
```

The CUSTLIST.C program uses **f4str** on non-Memo fields.

Code fragment from CUSTLIST.C

```
u4ncpy(nameStr,f4str(fName),sizeof(nameStr));
u4ncpy(addressStr,f4str(address),sizeof(addressStr));
    .
    .
    .
u4ncpy(dateStr,f4str(birthDate),sizeof(dateStr));
```

Using **f4str** and

Character fields are returned by **f4str** and **f4memoStr** as left justified strings

f4memoStr
On Character fields

padded out with blanks to the full length of the field.

On Date fields

When these functions are used on Date fields, they return the date in the form of an eight character string. Please refer to "Date Functions" chapter for more information.

On Numeric fields

The contents of Numeric fields are returned as right justified strings padded out to the full length of the field. If the field has any decimal places, the string will contain exactly that many decimal places.

On Logical fields

When used on Logical fields, these functions simply return a string containing "Y", "N", "y", "n", "T", "F", "t", or "f" (depending on the value stored on disk in the Logical field).

Retrieving Numerical Data

In addition to returning a field's contents as a string, CodeBase can also convert its contents to numerical data.

f4double

f4double returns the contents of the field as a **(double)** value. If **f4double** is unable to convert the field's contents, a value of **0** is returned.

f4int and **f4long**

The other two numeric operators, **f4int** and **f4long** convert the field's contents to **(int)** or **(long)** values. Any digits to the right of the decimal place (if any are stored in the field) are truncated.

Creating Data Files

Not only does CodeBase allow you to access existing files, you can also create new files. This is accomplished by a single function call.

The NEWLIST.C program creates a new data file if one of the same name does not already exist. It then appends several new records and assigns values to the fields in each record.

PROGRAM NEWLIST.C

```
/* NEWLIST.C */

#include "D4ALL.h"

CODE4    codeBase;
DATA4    *dataFile = 0;
FIELD4   *fName, *lName, *address, *age, *birthDate,
           *married, *amount, *comment;

FIELD4INFO  fieldInfo [] =
{
    {"F_NAME",r4str,10,0},
    {"L_NAME",r4str,10,0},
    {"ADDRESS",r4str,15,0},
    {"AGE",r4num,2,0},
    {"BIRTH_DATE",r4date,8,0},
    {"MARRIED",r4log,1,0},
    {"AMOUNT",r4num,7,2},
    {"COMMENT",r4memo,10,0},
    {0,0,0,0},
};

void  OpenDataFile( void )
{
    dataFile = d4open( &codeBase, "DATA1.DBF" );
    if(dataFile == NULL)
        dataFile = d4create( &codeBase, "DATA1.DBF", fieldInfo, 0 );

    fName = d4field( dataFile, "F_NAME" );
    lName = d4field( dataFile, "L_NAME" );
    address = d4field( dataFile, "ADDRESS" );
    age = d4field( dataFile, "AGE" );
    birthDate = d4field( dataFile, "BIRTH_DATE" );
```

```
    married = d4field( dataFile, "MARRIED" );
    amount = d4field( dataFile, "AMOUNT" );
    comment = d4field( dataFile, "COMMENT" );
}

void PrintRecords( void )
{
    int      rc,j,ageValue;
    double   amountValue;
    char     nameStr[25];
    char     addressStr[20];
    char     dateStr[9];
    char     marriedStr[2];
    const char     *commentStr;

    for(rc = d4top(dataFile);rc == r4success;rc = d4skip(dataFile, 1L))
    {
        u4ncpy( nameStr, f4str(fName), sizeof(nameStr) );
        u4ncpy( addressStr, f4str(address), sizeo f(addressStr) );

        ageValue = f4int( age );
        amountValue = f4double( amount );

        u4ncpy( dateStr, f4str(birthDate), sizeof(dateStr) );
        u4ncpy( marriedStr, f4str(married), sizeof(marriedStr) );

        commentStr = f4memoStr( comment );

        printf("-----------------------------\n");
        printf("Name     : %20s\n", nameStr);
        printf("Address  : %15s\n", addressStr);
        printf("Age : %3d   Married : %1s\n", ageValue, marriedStr);
        printf("Comment: %s\n", commentStr);
        printf("Purchased this year : $%5.2lf \n", amountValue);
        printf( "\n" );
    }
}

void AddNewRecord( char *fNameStr,
                   char*lNameStr,
                   char *addressStr,
                   int ageValue,
                   int marriedValue,
                   double amountValue,
                   char *commentStr )
{
    d4appendStart(dataFile,0);

    f4assign(fName,fNameStr);
    f4assign(lName,lNameStr);
    f4assign(address,addressStr);
    f4assignInt(age,ageValue);

    if(married Value)
        f4assign(married,"T");
    else
        f4assign(married,"F");

    f4assignDouble(amount,amountValue);
    f4memoAssign(comment,commentStr);

    d4append(dataFile);
}

int main( void )
{
    code4init(&codeBase);
    codeBase.errOpen = 0;
    codeBase.safety = 0;

    OpenDataFile();

    PrintRecords();

    AddNewRecord("Sarah", "Webber", "132-43 St.", 32, 1, 147.99, "New Customer");

    AddNewRecord("John", "Albridge", "1232-76 Ave.", 12, 0, 98.99, 0);

    PrintRecords();

    code4close(&codeBase);
```

```
    code4initUndo(&codeBase);
    return 0;
}
```

FIELD4INFO Structures

The **FIELD4INFO** structure is an integral component of the data file creation process. This structure contains the information which defines a field.

The **FIELD4INFO** structure contains a member for each of the field's four attributes. These members are described below:

- **(char *) name** This is a pointer to a null terminated string containing the name of the field. This name must be unique to the data file and any characters after the first ten are ignored. Valid field names can only contain letters, numbers, and/or underscores. The first letter of the name must be a letter.
- **(char) type** This member contains an abbreviation for the field type. CodeBase supports Character, Date, Floating Point, Logical, Memo, Numeric, Binary, and General field types. Valid abbreviations are either the character constants 'C', 'D', 'F', 'L', 'M', 'N', 'B' and 'G'. In addition you can also use the equivalent CodeBase constants: **r4str, r4date, r4float, r4log, r4memo, r4num**, **r4bin** and **r4gen**, respectively.
- **(int) len** This integer value determines the length of the field.
- **(int) dec** This member determines the number of decimal places in Numeric or Floating Point fields.

Refer to **d4create** in the CodeBase *Reference Guide* for more detailed information on field attributes.

FIELD4INFO arrays

Before a data file can be created, an array of **FIELD4INFO** structures must be defined. Each element of this array describes a single field.

Note

Each member of the last element in a **FIELD4INFO** array must be set to null. This indicates that there are no more elements in the array. Failure to provide an element filled with null's will result in errors when the data file is created.

An example **FIELD4INFO** array is defined below:

Code fragment from NEWLIST.C

```
FIELD4INFO  fieldInfo [] =
{
    {"F_NAME",r4str,10,0},
    {"L_NAME",r4str,10,0},
    {"ADDRESS",r4str,15,0},
    {"AGE",r4num,2,0},
    {"BIRTH_DATE",r4date,8,0},
    {"MARRIED",r4log,1,0},
    {"AMOUNT",r4num,7,2},
    {"COMMENT",r4memo,10,0},
    {0,0,0,0},
} ;
```

Creating The Data File

The actual creation of the data file is performed by the **d4create** function. This function uses the information in a **FIELD4INFO** array as the field specifications for a new data file.

The second parameter of the **d4create** function is a pointer to a null terminated string containing a file name. This file name can contain any extension, but if no extension is provided, the default ".DBF" extension is used. Like the **d4open** function, the current directory is assumed if no path is provided as part of the file name.

If the data file is successfully created, a pointer to its **DATA4** structure is returned. If for any reason the data file could not be created, a null pointer is returned instead..

The fourth parameter to **d4create** is used for creating production index files. Please refer to the "Indexing" chapter for details.

The **CODE4.safety** flag

The **CODE4.safety** flag determines whether any existing file is replaced when CodeBase attempts to create a file. If the file exists and the **CODE4.safety** flag is set to its default value of true (an integer value of 1), the new file is not created; otherwise when this flag is set to false (an integer value of 0), the old file is replaced by the newly created file. If the new file is not created because the file already exists and **CODE4.errCreate** is true, an error message is generated.

Note

As with all **CODE4** members, the **CODE4.safety flag** can only be changed after **code4init** has been called.

Opening Or Creating

It is often the case that you want to open a data file if it exists or create it if it does not. The following section of code demonstrates how this can be accomplished:

Code fragment from NEWLIST.C

```
CODE4 codeBase ;

code4init( &codeBase ) ;
codeBase.errOpen = 0 ;
codeBase.safety = 0 ;

. . .

dataFile = d4open( &codeBase, "DATA1.DBF" ) ;
if( dataFile == NULL )
    dataFile == d4create( &codeBase,"DATA1.DBF", fieldInfo, 0);
```

The **CODE4.errOpen** flag

If the data file does not already exist, **d4open** will normally generate an error message when it fails to open the file. This error message can be suppressed by changing the **CODE4.errOpen** flag. When this flag is set to true (its default value), an error message is generated when CodeBase is unable to open a file. If this flag is set to false (zero), CodeBase does not display this error message. It does however still return a null **DATA4** pointer, which can be tested. This is the recommended method for opening a data file if it exists and creating it if it does not.

Adding New Records

There are two methods that can be used to add new records to the data file. They both append a new record to the end of the data file.

Appending A Blank Record

If all you need to do is add a blank record to the bottom of the data file, **d4appendBlank** should be used.

```
d4appendBlank( dataFile ) ;
```

This function adds a new record to the data file, blanks out all of its fields, and makes the new record the current record.

The Append Sequence

The second method involves three steps. This method is more efficient and should be used when you intend to immediately assign values to the new records fields. Using this method results in less disk writes and therefore improves performance.

starting the append sequence

The first step is performed by the **d4appendStart** function. This function sets the current record number to zero to let CodeBase know that a new record is about to be appended. Additionally, **d4appendStart** temporarily disables automatic flushing for the current record, so that if changes need to be aborted, the record is not flushed to disk.

Code fragment from NEWLIST.C

```
d4appendStart( dataFile, 0 ) ;
```

The second step involves assigning values to the fields. After the **d4appendStart** function is called, the record buffer contains a copy of the previous current record. If no changes are made to the record buffer, a copy of this record is appended. If you prefer an empty record buffer, a call to **d4blank** will clear it.

```
d4appendStart( dataFile, 0) ;
d4blank( dataFile ) ;
```

Assigning values to the fields will be discussed in the next section.

Appending the record

The final step is accomplished by **d4append**. This function causes a new record to be physically added to the end of the data file and its key values to be added to any open index tags.

Code fragment from NEWLIST.C

```
d4append( dataFile ) ;
```

Assigning Field Values

Just as there are several CodeBase functions for retrieving information from fields, there are corresponding functions for storing information in fields.

CODE4.lockEnforce

In multi-user configurations, only one application should edit a record at a time. One way to ensure that only one application can modify a record is to explicitly lock the record before it is changed. To ensure that a record is locked before it is changed, set **CODE4.lockEnforce** to true (non-zero). When **CODE4.lockEnforce** is true (non-zero) and an attempt is made to modify an unlocked record using a field function or **d4blank**, **d4changed**, **d4delete** or **d4recall**, an **e4lock** error is generated and the modification is aborted.

An alternative method of ensuring that only one application modifies a record is to deny all other applications write access to the data file. In this case explicit locking is not required, even when **CODE4.lockEnforce** is true, since only one outside applications can write to the data file. Write access can be denied to other applications by setting the **CODE4.accessMode** to **OPEN4DENY_WRITE** or **OPEN4DENY_RW** before the data file is opened.

Refer to the "Multi-User Applications" chapter of this guide for more information.

Copying Strings To Fields

f4assign and **f4memoAssign** are two functions that copy strings to fields. The **f4assign** function works an any type of field except Memo fields, while **f4memoAssign** works on all types.

Copying strings to Character fields

If the length of the assigned string is less than the size of the field, the field's contents are left justified, and the remaining spaces are filled with blanks. If the string is larger than the field, its contents are truncated to fit. When an assignment is done with a memo field, the length is automatically adjusted to accommodate any value.

Code fragment NEWLIST.C

```
f4assign(fName,fNameStr);

f4memoAssign(comment,commentStr);
```

Copying strings to Numeric fields

f4assign and **f4memoAssign** can also store strings into Numeric fields, but generally this is not a good idea as it is very easy to format the information incorrectly. A correctly formatted number is right justified, zero-padded right to the number of decimals in the field, and space-padded left for any unused units.

Note

The functions **f4assign** and **f4memoAssign** store the information exactly as it appears in the string. This can cause problems when they are used to store non-numeric data into Numeric type fields. It is recommended that the functions **f4assignInt**, **f4assignDouble** or **f4assignLong** be used to store data in Numeric fields.

Copying strings to Date fields

Any strings that are copied to Date type fields should be exactly eight characters long and should contain a date in standard format (CCYYMMDD). For details please refer to the "Date Functions" chapter.

Copying strings to Logical fields

The only strings that should be copied to Logical fields are "T", "F","t", "f","Y", "N", "y" or "n".

Code fragment from NEWLIST.C

```
    if( marriedValue )
        f4assign( marrie d, "T" );
    else
        f4assign( married, "F" );
```

Storing Numeric Data

Instead of worrying about the formatting of strings containing numerical data, you can let CodeBase perform the formatting for you. **f4assignDouble, f4assignInt** and **f4assignLong** automatically convert and format numerical data.

Assigning **(double)** values to a field

You can assign a **(double)** value to a field using **f4assignDouble**. This function can automatically format the numerical value according to the field's attributes. For example, if the field has a length of six and has two decimal places, and the **(double)** value of 34.12345 assigned to it, the field will contain " 34.12" after **f4assignDouble** is called. If the **f4assignDouble** function is used on a Character field, it assumes the field has no decimals and right justifies its contents.

Code fragment from NEWLIST.C

```
f4assignDouble( amount, amountValue ) ;
```

Assigning integer values to a field

f4assignInt and **f4assignLong** are used to copy and format integer and long values to fields. Right justification is used, and if the field is of type Numeric, any decimal places are filled with zeros.

Code fragment from NEWLIST.C

```
f4assignLong( age, ageValue ) ;
```

Removing Records

CodeBase provides a two level method for removing records. You can delete a record by simply changing the status of its deletion flag to true. The records that are flagged for deletion can then be physically removed by "packing" the data file.

PROGRAM DELETION.C

The DELETION.C program demonstrates the effects of deleting, recalling, and packing records.

```
/* DELETION.C */

#include "D4ALL.H"

CODE4       codeBase;
DATA4       *dataFile = 0;
FIELD4INFO  fieldInfo[]=
{
    {"DUMMY",'C',10,0},
    {"MEMO",'M',10,0},
    {0,0,0,0},
};

void printDeleteStatus(int status,long recNo)
{
    if(status)
        printf("Record %5ld - DELETED\n",recNo);
    else
        printf("Record %5ld - NOT DELETED\n",recNo);
}

void printRecords(DATA4 *dataFile)
{
    int  rc,status;
    long recNo;

    printf("\n");
```

```
    rc = d4top(dataFile);
    while(rc != r4eof)
    {
        recNo = d4recNo(dataFile);
        status = d4deleted(dataFile);
        printDeleteStatus(status,recNo);
        rc = d4skip(dataFile,1L);
    }
}

void main( void )
{
    int count;

    code4init(&codeBase);
    dataFile = d4create(&codeBase,"TUTOR5",fieldInfo,0);
    error4exitTest(&codeBase);

    for(count = 0;count < 5;count ++)
        d4appendBlank(dataFile);

    printRecords(dataFile);

    d4go(dataFile,3L);
    d4delete(dataFile);
    d4go(dataFile,1L);
    d4delete(dataFile);
    d4go(dataFile,4L);
    d4delete(dataFile);
    printRecords(dataFile);

    d4go(dataFile,3L);
    d4recall(dataFile);
    printRecords(dataFile);

    d4pack(dataFile);
    d4memoCompress(dataFile);
    printRecords(dataFile);

    code4close(&codeBase);
    code4initUndo(&codeBase);

}
```

Explanation: This program creates the data file (or re-creates, if it exists) and then appends five records. The program then displays the deletion status of all the records:

DELETION.C output part 1

```
Record #   1 - NOT DELETED
Record #   2 - NOT DELETED
Record #   3 - NOT DELETED
Record #   4 - NOT DELETED
Record #   5 - NOT DELETED
```

Record numbers one and three are then marked for deletion and the deletion status of the records are again displayed.

DELETION.C output part 2

```
Record #   1 - DELETED
Record #   2 - NOT DELETED
Record #   3 - DELETED
Record #   4 - DELETED
Record #   5 - NOT DELETED
```

Record number three is recalled (ie. the deletion mark is removed) and the deletion status of the records are displayed.

DELETION.C output part 3

```
Record #   1 - DELETED
Record #   2 - NOT DELETED
Record #   3 - NOT DELETED
Record #   4 - DELETED
Record #   5 - NOT DELETED
```

Finally the data file is packed. Displaying the record status shows that record one was removed from the data file. Record two now occupies the space of record one.

DELETION.C output part 4

```
Record #   1 - NOT DELETED
Record #   2 - NOT DELETED
Record #   3 - NOT DELETED
```

Note

Since the record numbers are contiguous and always start at one, the record number for a specific record may have changed after the data file has been packed

Determining The Deletion Status

Each record has its own deletion flag. The status of the deletion flag of the current record can be checked using **d4deleted**.

Code fragment from DELETION.C

```
status = d4deleted( dataFile ) ;
```

d4deleted returns a true (non zero) value when the current record has been marked for deletion.

Marking A Record For Deletion

d4delete is used to set the current record's deletion status to true. The next time the data file is packed, the record is physically removed from the file and any reference to it in any open index file is also removed.

Code fragment from DELETION.C

```
d4delete( dataFile ) ;
```

Note

When a record is marked for deletion, it is not physically removed from the data file. In fact you can still access the relate4cord normally. The main importance of the deletion flag is that the marked record is physically removed when the data file is packed. The deletion flag may be used in tag filter expressions. Refer to the "Indexing" chapter for details.

Recalling Records

If the data file has not yet been packed, any deleted records can be recalled back to non-deleted status. This is accomplished by a call to **d4recall**.

```
d4recall( dataFile ) ;
```

Packing The Data File

Physically removing records from the data file is called packing. If your data file is quite large, this process can be quite time consuming. That is why records are first marked for deletion and then physically removed. The cumulative time necessary for removing individual records would be substantially greater than removing the many "deleted" records at one time. Packing the data file is performed by **d4pack**.

Note

Consider opening the file exclusively before packing to keep others from accessing the data file while packing is occurring. Otherwise, the data may appear as corrupted to other users for the duration of the pack.

Compressing The Memo File

When **d4pack** removes a record from the data file, it does not remove the corresponding memo entry (assuming that there is at least one Memo field). This results in unreferenced memo entries, which cause no difficulties in an application except that they simply waste disk space. To remove any wasted memo file space, the memo file can be compressed by calling **d4memoCompress**. Again, since compressing a memo file can consume a large amount of time, it may be appropriate to compress a data file sometime other than when the data file is packed

Note

Wasted memo file space is mainly caused by packing records that have memo entries without doing a memo compress. If you are using Clipper compatibility, wasted memo space can also occur when memo entries are increased beyond 504 bytes. In either case, memo compressing reduces disk space usage.

Data File Information

CodeBase contains a set of functions that provide information about the data files, their records and fields.

This program displays information about a data file whose name is provided as a command line argument. It displays the data file's alias, record count, record width, and the attributes of its fields.

PROGRAM DATAINFO.C

```
/* DATAINFO.C*/

#include "D4ALL.h"

int main(int argc,char *argv[])
{
    CODE4    codeBase;
    DATA4    *dataFile;
    FIELD4   *fieldRef;
    int      j,numFields;
    int      len,dec;
    int      recWidth;
    const char     *name;
    char     type;
    const char     *alias;
    long     recCount;

    if(argc != 2)
    {
        printf(" USAGE: FLDINFO <FILENAME.DBF> \n");
        exit(0);
    }

    code4init(&codeBase);

    dataFile = d4open(&codeBase,argv[1]);
    error4exitTest(&codeBase);

    recCount = d4recCount(dataFile);
    numFields = d4numFields(dataFile);
    recWidth = d4recWidth(dataFile);
    alias = d4alias(dataFile);

    printf("+--------------------------------+\n");
    printf("ï Data File: %12s        ï\n",argv[1]);
    printf("ï Alias    : %12s         ï\n",alias);
    printf("ï                                ï\n");
    printf("ï Number of Records: %7ld      ï\n",recCount);
    printf("ï Length of Record : %7d      ï\n",recWidth);
    printf("ï Number of Fields : %7d      ï\n",numFields);
    printf("ï                                ï\n");
    printf("ï Field Information  :           ï\n");
    printf("ï--------------------------------ï\n");
    printf("ï Name       ï type ï len  ï dec ï\n");
    printf("ï------------+------+---- --+-----ï\n");
```

```
    for(j = 1;j < d4numFields(dataFile);j ++)
    {
        fieldRef = d4fieldJ(dataFile,j);
        name = f4name(fieldRef);
        type = f4type(fieldRef);
        len = f4len(fieldRef);
        dec = f4decimals(fieldRef);

        printf("ï %10s ï   %c  ï %4d ï %4d ï\n", name, type, len, dec);
    }

    printf("+--------------------------------+\n");

    d4close(dataFile);
    code4initUndo(&codeBase);
    return 0;
}
```

The data file alias

Each data file that you have opened has a character string label called the *alias*. The alias is mainly used for qualifying field names in dBASE expressions or looking up a **DATA4** pointer using **code4data**.

d4alias returns a pointer to a string containing the data file's alias. This character pointer is valid as long as the data file is open.

Code fragment for DATAINFO.C

```
const char *alias;

alias = d4alias(dataFile);
```

Although you can directly modify the string returned from **d4alias**, it is highly recommended that you change the alias by calling **d4aliasSet**. This prevents memory corruption that may be caused by writing beyond the end of the string.

Finding the number of records

An important data file statistic is the record count. You can obtain this value using **d4recCount**. This tells you how many records are in the data file. This function does not take into account the deleted status of any records, nor any filter condition of a open tag.

Code fragment for DATAINFO.C

```
long recCount;

recCount = d4recCount( dataFile );
```

Determining the record width

Another useful function is **d4recWidth**. This function returns the length of the record buffer. This value is the total length of all the fields plus one byte for the deletion flag. This function may be used to save copies of the record buffer.

Code fragment for DATAINFO.C

```
long  recWidth

recWidth = d4recWidth( dataFile );
```

Field Information

If you have access to a field's **FIELD4** pointer, you can retrieve the field attributes.

Field name

f4name returns a pointer to the field's name. This pointer is valid as long as the data file is open.

Code fragments from DATAINFO.C

```
const char *name;

name = f4name( fieldRef );
```

WARNING

Do not modify the string returned by **f4name.** Doing so corrupts CodeBase.

Field type

f4type returns the field type:

```
char type;

type = f4type( fieldRef );
```

Field length and decimals

The length of the field and the number of decimals are returned by **f4len** (**f4memolen**) and **f4decimals** respectively.

```
int len, dec;

len = f4len( fieldRef );
dec = f4decimals( fieldRef);
```

Advanced Topics

This section explains the use of multiple **CODE4** structures in an application, and copying data file structures.

Multiple CODE4 Structures

Normally, you have one **CODE4** structure in your application. However, sometimes your application needs to access data files only in a few places for a short time. In this case, it can be appropriate to have more than one **CODE4** structure.

Uninitializing the **CODE4** structure

If you have a single **CODE4** structure that is initialized at the beginning of the application, you can tie up memory that you are not using. To avoid this, you can initialize the **CODE4** structure before you start using it and uninitialize it after. This is done through calls to **code4init** and **code4initUndo**.

code4init initializes CodeBase internal memory and sets all of the **CODE4** member variables to their default values.

code4initUndo closes all data, memo, and index files. It returns any allocated memory back to the CodeBase memory management pool or to the operating system.

The following is an example program that uses two **CODE4** structures.

```
void Function( void )
{
    CODE4   codeBase;
    DATA4   *dataFile;

    d4init(&codeBase);

    dataFile = d4open(&cod eBase,"DATAFILE");

    ...  perform CodeBase operations ...

    code4initUndo(&codeBase);

}

void OtherFunction( void )
```

```
{
    CODE4   codeBase;
    DATA4   *dataFile;

    d4init(&codeBase);

    ...  Do some other CodeBase operations ...

    cod4initUndo(&codeBase);
}

void main( void )
{
    Function();

    OtherFunction();

    ... etc ...
}
```

Copying Data File Structures

You may occasionally require a new data file that has an identical structure as an existing data file.

To satisfy this need CodeBase provides the function **d4fieldInfo,** which returns a copy of the **FIELD4INFO** array of an existing file. Pass this array to the **d4create** function to create a new data file with the identical structure.

Note

The **FIELD4INFO** array that is returned by **d4fieldInfo** is allocated dynamically and should therefore be deallocated after you are finished using it. The **u4free** function can be used for this purpose.

The **FIELD4INFO** array, returned by **d4fieldInfo**, contains pointers to actual field names; not copies. If you attempt to modify the field names, you cause functions such as **d4field** to malfunction.

PROGRAM COPYDATA.C

The COPYDATA.C program takes two file names as arguments. It uses the structure from the data file specified by the first argument to create an empty data file with the name specified by the second argument.

```
#include "d4all.h"

int main(int argc,char *argv[])
{
    CODE4      codeBase;
    DATA4      *dataFile,*dataCopy;
    FIELD4INFO *fieldInfo;

    if(argc != 3)
    {
        printf("USAGE: COPYDATA <FROM FILE> <TO FILE>" ) ;
        exit(1);
    }

    code4init(&codeBase);
    codeBase.safety = 0;

    dataFile = d4open(&codeBase,argv[1]);
    error4exitTest(&codeBase);

    fieldInfo = d4fieldInfo(dataFile);

    dataCopy = d4create(&codeBase ,argv[2],fieldInfo,0);

    u4free(fieldInfo);

    code4close(&codeBase);
    code4initUndo(&codeBase);
    exit(1) ;
}
```

4 Indexing

The purpose of a database is to organize information in a useful manner. So far you have seen how the information is divided into fields and records. Now it is time to order the records in purposeful ways. One approach is to physically sort records in your data files. Unfortunately, maintaining data file sorts involves continually shuffling large amounts of data. In addition, it is only possible to maintain a single sort order using this method.

A preferable method is to leave the records in the data file in their original order and store the sorted orderings in a separate file called an index file. When you create an index file, you are effectively sorting the data file. Index files are efficiently maintained and you can have an unlimited number of sorted orderings continually available.

Indexes & Tags

Each index file can contain one or more sorted orderings. These sorted orderings are identified by a *tag*. That is, each index file tag corresponds to a single sorted ordering. CodeBase supports three types of index files. Their attributes and differences are described below:

File Format	Compatibility	Number of Tags	Production Indexes	Group Files	Index Filtering	Descending Ordering
.MDX	dBASE IV	1 - 47	yes	no	yes	yes
.CDX	FoxPro 2.x FoxPro 3.0	1 - 47	yes	no	yes	yes
.NTX	Clipper 87 Clipper 5.x	1	Only with group files	yes	no	Special Case

Figure 4.1 Index File Formats

The difference between indexes and tags are illustrated in Figure 4.2.

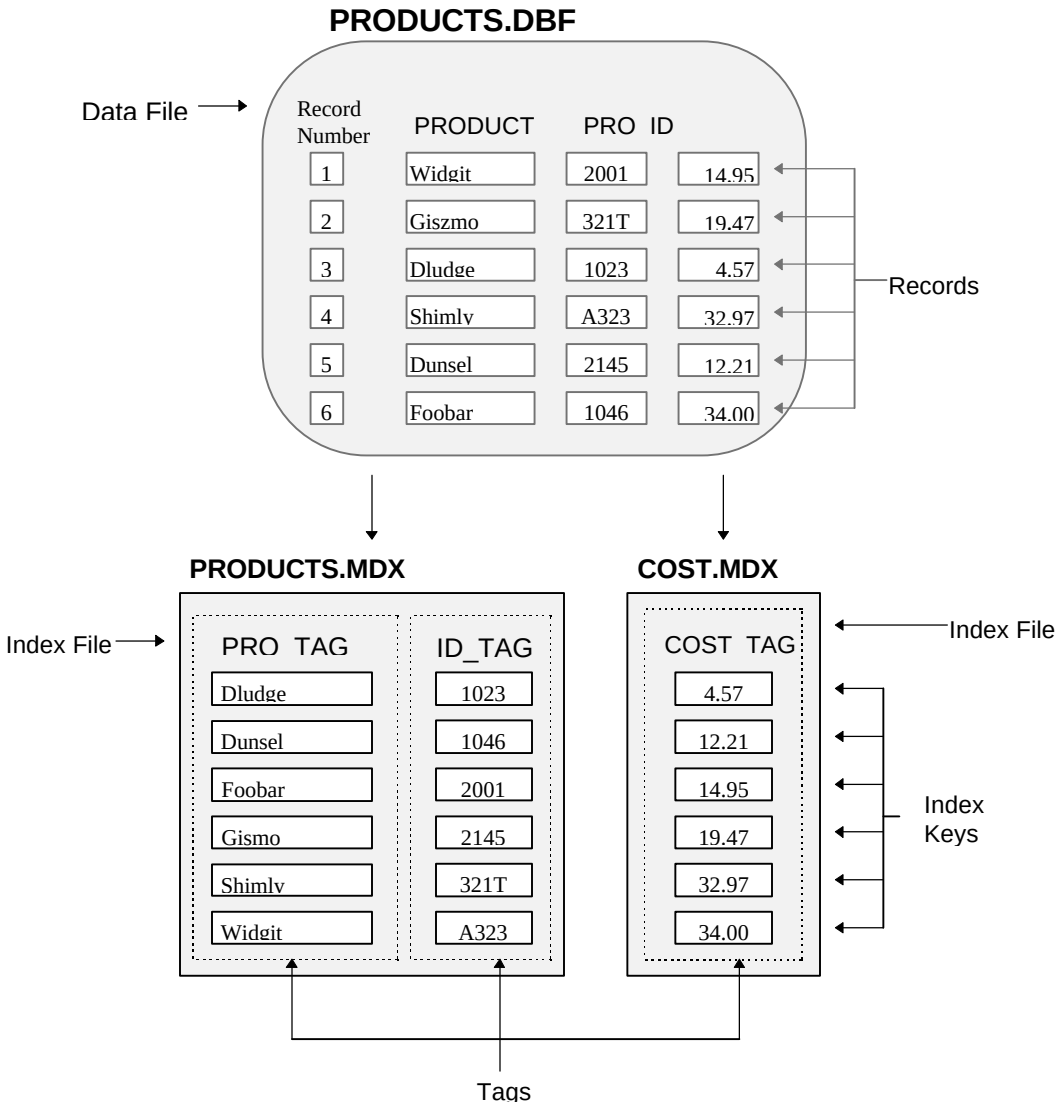


Figure 4.2 Index Files

Index Expressions

An *index expression* is a *dBASE expression* that is used to determine the *index key* for each record. An index expression must evaluate to a value of one of the following types: Numeric, Character, or Date types. When you are using **.CDX** (FoxPro) indexes, you can also index on Logical expressions. Refer to "Appendix C: dBASE Expressions" in the CodeBase *Reference Guide* for details on dBASE expressions.

The most commonly used index expression is a field name. For example, the index expression for the PRO_TAG tag in Figure 4.2 is "PRODUCT". When this expression is evaluated for record number 6, the value of the field PRODUCT is returned; for record number 6, this value is "Foobar ". To put it simply, tag PRO_TAG is ordered by the contents in field PRODUCT.

Creating An Index File

Other common index expressions involve generating tags based on two or more fields. This is known as a *compound index key*. For example, if we assume that field PRO_ID is also a character field, we can base a tag ordering with the following index expression: "PRODUCT + PRO_ID". This produces an index key consisting of the PRODUCT field concatenated with the PRO_ID field. For record number 6, the resulting index key is "Foobar 1046".

In addition, most *dBASE functions* are also permitted. See "Appendix C: dBASE Expressions" in the CodeBase *Reference Guide* for a list of dBASE functions supported. If we wanted to base a tag on the PRODUCT field, but remain case insensitive, we can use the dBASE function UPPER() to convert the index key to upper case. In this case the index expression would be "UPPER(PRODUCT)".

PROGRAM NEWLIST2.C

Creating an index file is similar to creating a data file; in fact you can even create both at the same time. There are two types of index files that you can create: *production indexes* and *non-production indexes*.

The following is the program listing for NEWLIST2.C. This modified version of the NEWLIST.C program, creates an index file along with the data file.

```
/* NEWLIST2.C */
#include "D4ALL.h"

CODE4    codeBase;
DATA4    *dataFile;
FIELD4   *fName, *lName, *address, *age, *birthDate, *married, *amount, *comment;
TAG4     *nameTag, *ageTag, *amountTag;

FIELD4INFO  fieldInfo [] =
{
    {"F_NAME", r4str, 10, 0},
    {"L_NAME", r4str, 10, 0},
    {"ADDRESS", r4str, 15, 0},
    {"AGE", r4num, 2, 0},
    {"BIRTH_DATE", r4date, 8, 0},
    {"MARRIED", r4log, 1, 0},
    {"AMOUNT", r4num, 7, 2},
    {"COMMENT", r4memo, 10, 0},
    {0, 0, 0, 0},
};

TAG4INFO  tagInfo[] =
{
    {"NAME_TAG", "F_NAME + L_NAME", 0, 0, 0},
    {"AGE_TAG", "AGE", 0, 0, 0},
    {"AMNT_TAG", "AMOUNT", 0,  0, 0},
    {0, 0, 0, 0, 0},
};

void  CreateDataFile( void )
{
    dataFile = d4create(&codeBase, "DATA1.DBF", fieldInfo, tagInfo);
    fName = d4field(dataFile, "F_NAME");
    lName = d4field(dataFile, "L_NAME");
    address = d4field(dataFile, "ADDRESS");
    age = d4field(dataFile, "AGE");
    birthDate = d4field(dataFile, "BIRTH_DATE");
    married = d4field(dataFile, "MARRIED");
    amount = d4field(dataFile, "AMOUNT");
    comment = d4field(dataFile, "COMMENT");
```

```
    nameTag = d4ta g(dataFile, "NAME_TAG");
    ageTag = d4tag(dataFile, "AGE_TAG");
    amountTag = d4tag(dataFile, "AMNT_TAG");
}

void PrintRecords( void )
{
    int      rc, j, ageValue;
    double   amountValue;
    char     fNameStr[15], lNameStr[15];
    char     addressStr[20];
    char     dateStr[9];
    char     marriedStr[2];
    const char     *commentStr;

    for(rc = d4top(dataFile); rc == r4success; rc = d4skip(dataFile, 1L))
    {
        f4ncpy(fName, fNameStr, sizeof(fNameStr));
        f4ncpy(lName, lNameStr, sizeof(lNameStr));
        f4ncpy(address, addressStr, sizeof(addressStr));
        ageValue = f4int(age);
        amountValue = f4double(amount);
        f4ncpy(birthDate, dateStr, sizeof(dateStr));
        f4ncpy(married, marriedStr, sizeof(marriedStr));
        commentStr = f4memoStr(comment);

        printf("------------------------------\n");
        printf("Name     : %10s %10s\n", fNameStr, lNameStr);
        printf("Address  : %15s\n", addressStr);
        printf("Age : %3d   Married : %1s\n", ageValue, marriedStr);
        printf("Comment: %s\n", commentStr);
        printf("Amount purchased this year: "              " $%5.2lf\n\n", amountValue);
    }
}

void AddNewRecord(char *fNameStr,
                  char *lNameStr,
                  char *addressStr,
                  int ageValue,
                  int marriedValue,
                  double amountValue,
                  char *commentStr)
{
    d4appendStart(dataFile, 0);

    f4assign(fName, fNameStr);
    f4assign(lName, lNameStr);
    f4assign(addre ss, addressStr);
    f4assignInt(age, ageValue);

    if(marriedValue)
        f4assign(married, "T");
    else
        f4assign(married, "F");

    f4assignDouble(amount, amountValue);
    f4memoAssign(comment, commentStr);

    d4append(dataFile);
}

int main( void )
{
    code4init(&codeBase);
    codeBase.safety = 0;

    CreateDataFile();

    AddNewRecord("Sarah", "Webber", "132-43 St.", 32, 1, 147.99, "New Customer");
    AddNewRecord("John", "Albridge", "1232-76 Ave.", 12, 0, 9  8.99, 0);
    PrintRecords();

    d4tagSelect(dataFile, nameTag);
    PrintRecords();

    d4tagSelect(dataFile, ageTag);
    PrintRecords();
```

```
    d4tagSelect(dataFile, amountTag);
    PrintRecords();

    code4close(&codeBase);
    code4initUndo(&codeBase);
    return 0;
}
```

Production Indexes

A production index is an index that is opened automatically when its associated data file is opened. A data file can have only one production index and this index has the same name as the data file, but with a different extension.

Non-Production Indexes

All other index files are non-production indexes. A data file can have an unlimited number of non-production indexes. To use these indexes, they must explicitly be opened after the data file has been opened.

The TAG4INFO Structure

Just as the attributes of a field in the data file are defined by a **FIELD4INFO** structure, the attributes of a tag in an index file are specified by a **TAG4INFO** structure. This structure contains the following five members:

The **TAG4INFO** members

The first two members must be defined for every **TAG4INFO** structure. The last three members are used to specify optional properties of the tag which will be discussed later in this chapter.

- **(char *) name** This is a pointer to a character array containing the name of the tag. The name may be composed of letters, numbers and underscores. Only letters and underscores are permitted as the first character of the name. This member cannot be null except to indicate that there are no more tags.

  When using FoxPro or **.MDX** formats, the name must be unique to the data file and have a length of ten characters or less excluding the extension. If you are using the **.NTX** index format, then this name can include a tag name with a path. In this case, the tag name within the path is limited to eight characters or less, excluding the extension.

- **(char *) expression** This is a **(char)** pointer to the tag's index expression. This expression determines the sorting order of the tag. (see the "Index Expression" section of this chapter for details) This member cannot be null.

- **(char *)filter** This is a pointer to the tag's filter expression.

- **(int) unique** This integer determines how duplicate keys are treated. See the "Unique Keys" section of this chapter for more details.

- **(int) descending** This integer determines if the index should be generated in descending order. Set this member to **0** to specify ascending order, or **r4descending** if descending order is desired.

Before the index file can be created, an array of **TAG4INFO** structures must be defined. Each element of this array denotes a single tag.

The **TAG4INFO** array

Before the index file can be created, an array of **TAG4INFO** structures must be defined. Each element of this array denotes a single tag.

Note

Each member of the last element in a **TAG4INFO** array must be set to null. This indicates that there are no more elements in the array. Failure to provide an element filled with null's will result in errors when the index file is created.

An example **TAG4INFO** array is defined below.

Code fragment from NEWLIST2.C

```
TAG4INFO  tagInfo[] =
{
    {"NAME_TAG", "F_NAME + L_NAME", 0, 0, 0},
    {"AGE_TAG", "AGE", 0, 0, 0},
    {"AMNT_TAG", "AMOUNT", 0, 0, 0},
    {0, 0, 0, 0, 0},
};
```

Using d4create To Create Index Files

The most common method for creating an index file is to create it when the data file is created. In this case the index will be a production index. If you recall, the **d4create** accepts four parameters, only three of which were used in the last chapter. The fourth parameter is a pointer to a **TAG4INFO** array which is used for creating a production index file. This is illustrated in the following code segment:

Code fragment from NEWLIST2.C

```
dataFile = d4create( codeBase, "DATA1.DBF", fieldInfo, tagInfo);
```

Using i4create To Create Index Files

The other method for creating index files involves the use of the **i4create** function. In addition to a **TAG4INFO** array, this function also accepts a string containing a file name. An index file is created with this file name. If no extension is specified, an appropriate extension for the file compatibility used (**.MDX**, **.CDX**, or **.NTX**) is provided.

For example, the program NEWLIST2.C could be modified to use **i4create** instead.

```
Index4 *index = NULL;
dataFile = d4create(&codeBase, "DATA1.DBF", fieldInfo, NULL);
index = i4create(dataFile, "DATAINDX", tagInfo);
```

i4create returns a pointer to the index file's **INDEX4** structure. The pointer is used by many low level index functions.

Creating Production Indexes with **i4create**

Normally**, i4create** does not produce production indexes, even if the file name that you provide is the same as the data file's. **i4create** can be used to create production indexes in the following manner. The data file must be opened exclusively before **i4create** is called. This can be achieved by setting the **code4accessMode** to **OPEN4DENY_RW** before the data file is opened. After the data file is opened or created exclusively, then **i4create** is used to create a production index by passing null as the file name parameter. This is illustrated as follows:

```
INDEX4 *index ;

codeBase.accessMode = OPEN4DENY_RW;

dataFile = d4create( codeBase, "DATA1.DBF", fieldInfo ) ;

index = i4create(dataFile, NULL, tagInfo) ;
```

Maintaining Index Files

CodeBase automatically updates all open index files for the data file. When a record is added, modified, or deleted, the appropriate tag entries for all of the open index tags are modified automatically. In general application programming, there is no need to manually add, delete, or modify the tag entries. CodeBase handles all of the key manipulation in the background, letting you concentrate on application programming.

Code fragment

```
CODE4 codeBase ;
DATA4 *dataFile;
FIELD4 *nameField;
int rc ;

code4init(&codeBase);
dataFile = d4open(&codeBase, "DATAFILE");
nameField = d4field(dataFile,"NAME");

rc = d4top(dataFile);
f4assign( nameField, "ROBERT WILKHAM") ;

rc = d4skip(dataFile, 1L) ;

/* the changes are flushed to disk.  If "NAME" is a                 */
/* key field, the index tag entry is automatically                  */
/* updated.  A seek for "ROBERT WILKHAM" would succeed      */
```

Opening Index Files

Opening index files can be accomplished by one of two functions. If the data file has a production index, **d4open** automatically opens it. Other index files may be opened using function **i4open.**

```
CODE4   codeBase;
DATA4   *dataFile;
INDEX4  *index;

code4init(&codeBase);
dataFile = d4open(&codeBase, "DATAFILE");

index = i4open(dataFile, "INDEXFILE");
```

Disabling the automatic opening of production indexes

There may be occasions when you prefer to leave the production index file closed. You can disable the automatic opening of production index files by setting flag **CODE4.autoOpen** to false (zero). The production index can then be opened like any other index file:

```
CODE4   codeBase;
DATA4   *dataFile;
INDEX4  *index, *index2;

code4init(&codeBase);
codeBase.autoOpen = 0;
dataFile = d4open(&codeBase, "DATAFILE");

index = i4open(dataFile, "INDEXFILE");
index2 = i4open(dataFile, "DATAFILE");
```

Referencing Tags

Before a tag is used, you must first obtain a pointer to its **TAG4** structure. When calling tag functions, this pointer specifies which tag the function operates on. There are six functions that can be used to retrieve a tag's **TAG4** pointer: **d4tag**, **d4tagDefault**, **d4tagNext**, **d4tagPrev, d4tagSelected** and **t4open**.

PROGRAM SHOWDAT2.C

The program SHOWDAT2.C is a modified version of SHOWDATA.C. This version displays the records of the data file in natural order and according to any tags in its production index.

```
/* SHOWDAT2.C */
#include "D4ALL.h"

CODE4    codeBase;
DATA4    *dataFile = NULL;
FIELD4   *fieldRef = NULL;
TAG4     *tag = NULL;

int      rc,j;
const char     *fieldContents;

void printRecords( void )
{
    for(rc = d4top(dataFile); rc == r4success; rc = d4skip  (dataFile, 1L))
    {
        for(j = 1;j <= d4numFields(dataFile);j ++)
        {
            fieldRef = d4fieldJ(dataFile,j);
            fieldContents = f4memoStr(fieldRef);
            printf("%s ",fieldContents);
        }
        printf("\n");
    }
}

int main(int argc,char *argv[])
{
    if(argc != 2)
    {
        printf("USAGE: SHOWDAT2 <FILENAME.DBF> \n");
        exit(0);
    }

    code4init(&codeBase);

    dataFile = d4open(&codeBase,argv[1]);
    error4exitTest(&codeBase);

    printf("Data File %s in Natural Order\n",argv[1]);
    printRecords();
    for(tag = d4tagNext(dataFile,NULL);tag !=NULL; tag = d4tagNext(dataFile,tag))
    {
        printf("\nPress ENTER to continue:");
        getchar();

        printf("\nData File %s sorted by Tag %s\n",argv[1],t4alias(tag));
        d4tagSelect(dataFile,tag);
        printRecords();
    }

    d4close(dataFile);
    code4initUndo(&codeBase);
    return 0;
}
```

Referencing By Name

If you know the tag's name, you can retrieve its **TAG4** pointer using **d4tag**. This function looks through all the open index files of the specified data file for a tag matching the name. This method is used by program 'NEWLIST2.C'.

Code Fragment from NEWLIST2.C

```
nameTag = d4tag( dataFile, "NAME_TAG" ) ;
ageTag = d4tag( dataFile, "AGE_TAG" ) ;
amountTag = d4tag( dataFile, "AMNT_TAG" ) ;
```

Obtaining the Default Tag

The default tag is the currently selected tag. If there are no tags selected, the default tag is the first tag in the first opened index file. You can obtain a pointer to the default tag's **TAG4** structure by calling function **d4tagDefault**.

```
TAG4 *tag;

dataFile = d4open(&codeBase, "DATAFILE");
tag = d4tagDefault(dataFile);
```

Iterating Through The Tags

The functions **d4tagNext** and **d4tagPrev** are used for iterating through the tags.

d4tagNext

The function **d4tagNext** returns the next tag after the specified tag. The first tag of the first index is returned by passing null as the tag parameter. If the last tag in the index file is passed into the tag parameter, the first tag in the next index file is returned.

Code fragment from SHOWDAT2.C

```
for( tag = d4tagNext( dataFile, NULL ); tag != NULL;
                                    tag = d4tagNext( dataFile, tag ))
{
    . . .
}
```

d4tagPrev

The function **d4tagPrev** works like **d4tagNext** except that it starts at the last tag of the last index file and works its way toward the first tag of the first index file. If null is passed as the tag parameter to **d4tagPrev**, the last tag of the last index is returned.

Selecting Tags

Initially a data file does not have a selected tag and is therefore in *natural order*. Natural order is the order in which the records were added to the data file. When you want to use a particular sort ordering, select a tag by calling **d4tagSelect**.

Code fragment from NEWLIST2.C

```
d4tagSelect( dataFile, nameTag ) ;
PrintRecords( ) ;

d4tagSelect( dataFile, ageTag ) ;
PrintRecords( ) ;

d4tagSelect( dataFile, amountTag ) ;
PrintRecords( ) ;
```

Selecting Natural Order

If you want to return to natural order simply pass null to **d4tagSelect** instead of a **TAG4** pointer.

The Effects Of Selecting a Tag

Once a tag is selected, the behaviour of **d4top, d4bottom**, and **d4skip** changes. Functions **d4top** and **d4bottom** then set the current record to the first and last data file entries respectively, using the tag's sort ordering. Similarly, **d4skip** skips using the tag ordering instead of the natural ordering.

Tag Filters

A tag filter is used to obtain a subset of the available data file records. The subset is based on true/false conditions calculated from a dBASE expression. Only records that pass through the filter have entries in the tag. A tag filter is created when the tag is created and cannot be changed later without destroying and recreating the tag.

PROGRAM SHOWLST2.C

In program 'SHOWLST2.C', when the index file is created, several filters are present.

```
#include "D4ALL.h"

CODE4    codeBase;
DATA4    *dataFile = 0;
FIELD4 *fName,
        *lName,
        *address,
        *age,
        *birthDate,
        *married,
        *amount,
        *comment;
TAG4   *nameTag,
        *ageTag,
        *amountTag,
        *addressTag,
        *birthdateTag;

TAG4INFO  tagInfo[] =
{
    {"NAME","L_NAME+F_NAME",".NOT. DELETED()",0,0},
    {"ADDRESS","ADDRESS",0,0,0},
    {"AGE_TAG","AGE","AGE >= 18",0,0},
    {"DATE_TAG","BIRTH_DATE",0,0,0},
    {"AMNT_TAG","AMOUNT",0,0,0},
    {0,0,0,0,0},
};

void  OpenDataFile( void )
{
    codeBase.autoOpen = 0;                /*prevents production index from being
                                            opened when data file is opened*/
    codeBase.safety = 0;
    codeBase.accessMode = OPEN4DENY_RW;   /*open data file exclusively so new
                                                    production index can be created*/

    dataFile = d4open(&codeBase,"DATA1.DBF");

    i4create(dataFile,NULL,tagInfo);

    fName = d4field(dataFile,"F_NAME");
    lName = d4field(dataFile,"L_NAME");
    address = d4field(dataFile,"ADDRESS");
    age = d4field(dataFile,"AGE");
    birthDate = d4field(dataFile,"BIRTH_DATE");
    married = d4field(dataFile,"MARRIED");
    amount = d4field(dataFile,"AMOUNT");
    comment = d4f ield(dataFile,"COMMENT");

    nameTag = d4tag(dataFile,"NAME");
    addressTag = d4tag(dataFile,"ADDRESS");
    ageTag = d4tag(dataFile,"AGE_TAG");
    birthdateTag = d4tag(dataFile,"DATE_TAG");
    amountTag = d4tag(dataFile,"AMNT_TAG");
}

void PrintRecords( void )
{
    int      rc,j,ageValue;
    double   amountValue;
    char     fNameStr[15],lNameStr[15];
    char     addressStr[20];
    char     dateStr[9];
    char     marriedStr[2];
    const char     *commentStr;

    for(rc = d4top(dataFile);rc == r4success;rc = d4skip(dataFile, 1L))
    {
        f4ncpy(fName,fNameStr,sizeof(fNameStr));
        f4ncpy(lName,lNameStr,sizeof(lNameStr));
        f4ncpy(address,addressStr,sizeof(addressStr));
        ageValue = f4int(age);
        amountValue = f4double(amount);
        f4ncpy(birthDate,dateStr,sizeof(dateStr));
        f4ncpy(married,marriedStr,sizeof(marriedStr));
        commentStr = f4memoStr(comment);
```

```
        printf("---------------------------\n");
        printf("Name     : %10s %10s\n",fNameStr,lNameStr);
        printf("Address  : %15s\n",addressStr);
        printf("Age : %3d   Married : %1s\n",ageValue,marriedStr);
        printf("Comment: %s\n",commentStr);
        printf("Amount purchased : $%5.2lf \n",amountValue);
        printf("\n");
    }
}

int main( void )
{
    code4init(&codeBase);
    OpenDataFile();

    d4tagSelect(dataFile,nameTag);
    PrintRecords();

    d4tagSelect(dataFile,ageTag);
    PrintRecords();

    d4tagSelec t(dataFile,amountTag);
    PrintRecords();

    code4close(&codeBase);
    code4initUndo(&codeBase);
    return 0;
}
```

Filter Expressions

A *filter expression* is a dBASE expression that returns a Logical result and is used as a tag filter. The expression is evaluated for each record as its tag entry is updated. If the filter expression evaluates to true, an entry for that record is included in the tag. If it evaluates to false, that record's tag entry is omitted from the tag.

Creating A Tag Filter

The tag filter is specified, when the index is initially created, through the **TAG4INFO.filter** structure member**.** If this member is null, then no records are filtered. The following code segment shows several tags with filters:

Code Fragment from SHOWDAT2.C

```
TAG4INFO  tagInfo[] =
{
    {"NAME", "L_NAME+F_NAME",  ".NOT. DELETED()", 0, 0},
    {"ADDRESS", "ADDRESS", 0, 0, 0},
    {"AGE_TAG", "AGE",  "AGE >= 18", 0, 0},
    {"DATE_TAG", "BIRTH_DATE", 0, 0, 0},
    {"AMNT_TAG", "AMOUNT", 0, 0, 0},
    {0, 0, 0, 0, 0},
} ;
```

The first tag has a filter expression of ".NOT. DELETED()". This filters out any records that have been marked for deletion. The third tag's filter expression is "AGE >= 18". This excludes any record which has an AGE field value of less than eighteen.

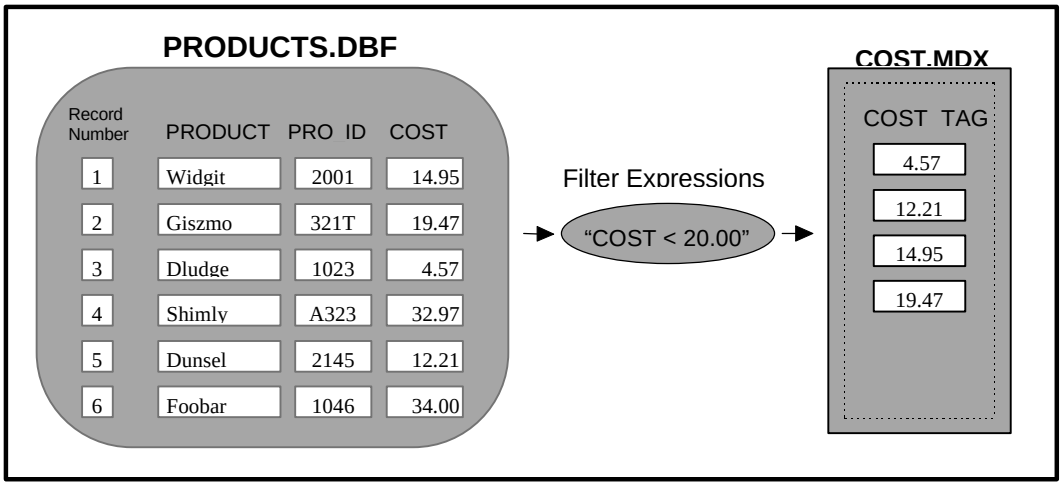


Figure 4.3 Filter Expressions

Unique Keys

The fourth member of the **TAG4INFO** structure, **(int) unique**, determines how duplicate keys are handled. This member can have four possible values: **0**, **r4uniqueContinue, e4unique** and **r4unique.**

It is often desirable to ensure that no two index keys are the same. To meet this requirement, you can specify that a tag must only contain unique keys. As a result, when a new key is to be added to the tag, a search is first made to see if that key already exists in the tag. If it does exist, the new entry is not added. CodeBase provides three methods of handling the addition of non-unique keys: **r4uniqueContinue, e4unique** and **r4unique.**

The descriptions of the values for the **TAG4INFO.unique** member are as follows:

- **0** Duplicate keys are allowed.
- **r4uniqueContinue** Any duplicate keys are discarded. However, the record continues to be added to the data file. In this case, there may not be a tag entry for a particular record.
- **e4unique** An error message is generated if a duplicate key is encountered.
- **r4unique** The data file record is not added or changed. Regardless, no error message is generated. Instead a value of **r4unique** is returned.

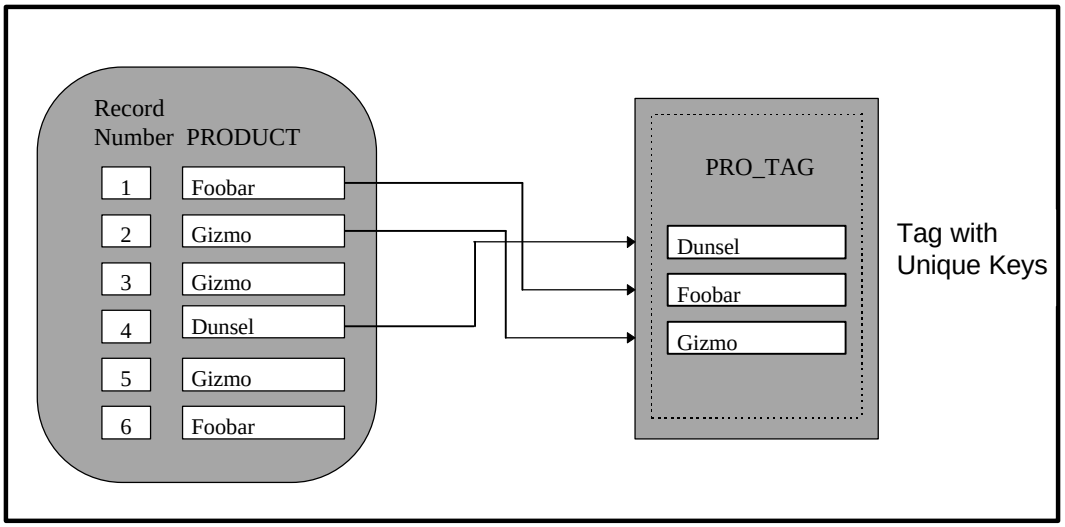


Figure 4.4 Unique Keys

Here is an example **TAG4INFO** array that uses unique keys:

```
TAG4INFO  tagInfo[ ] =
{
    {"NAME", "L_NAME + F_NAME", 0,  r4unique, 0},
    {"AGE_TAG", "AGE", "AGE >= 18", 0, 0},
    {"AMNT_TAG", "AMOUNT", 0,  r4uniqueContinue, 0},
    {0, 0, 0, 0, 0},
};
```

WARNING

Although a specific unique code for a tag is specified when the tag is created, only a true/false flag, which indicates whether a tag is unique or non-unique, is actually stored on disk. No information on how to respond to a duplicate key for unique key tags is saved.

As a result, when a tag is re-opened, the value contained in **CODE4.errDefaultUnique** is used to set the tag's unique code. If the tag was designed to be **r4unique** or **e4unique**, then the default value of **r4uniqueContinue** must be changed before any records are modified or appended.

This change can be accomplished in two ways. The first involves setting the **CODE4.errDefaultUnique** to another value, in which case all indexes opened subsequently will have this value for their unique code. This setting has no effect on non-unique tags.

The second method is to set the unique code for one or more tags individually after the index file has been opened. Pass the appropriate unique code to the function **t4uniqueSet**. This function only has an effect on unique tags.

Note

t4uniqueSet can only be used to change the setting of a unique tag. A CodeBase error is generated when an attempt is made to set a unique tag to a non-unique tag or a non-unique tag to a unique tag.

Seeking

One of the most useful features of a database is the ability to find a record by providing a *search key*. This process is referred to as *seeking*. When a seek is performed, the search key, which is usually a string, is compared against the index keys in the selected tag. When a match occurs, the corresponding record is loaded in the record buffer.

The SEEKER.C program demonstrates seeks on the various types of tags. When it performs the seeks, it displays the record that was found and the return value that was returned from the seek.

PROGRAM SEEKER.C

```
#include "D4ALL.h"

CODE4    codeBase;
DATA4    *dataFile = 0;
FIELD4   *fName,*lName,*address,*age,*birthDate,*married,*amount,*comment;
TAG4     *nameTag,*ageTag,*amountTag,*addressTag,*birthdateTag;

void  OpenDataFile( void )
{
    dataFile = d4open(&codeBase,"DATA1.DBF");

    fName = d4field(dataFile,"F_NAME");
    lName = d4field(dataFile,"L_NAME");
    address = d4field(dataFile,"ADDRESS");
    age = d4field(dataFile,"AGE");
    birthDate = d4field(dataFile,"BIRTH_DATE");
    married = d4field(dataFile,"MARRIED");
    amount = d4field(dataFile,"AMOUNT");
    comment = d4field(dataFile,"COMMENT");

    nameTag = d4tag(dataFile,"NAME_TAG");
    addressTag = d4tag(dataFile,"ADDR_TAG");
    ageTag = d4tag(dataFile,"AGE_TAG");
    birthdateTag = d4tag(dataFile,"DATE_TAG");
    amountTag = d4tag(dataFile,"AMNT_TAG");
}

void seekStatus(int rc,char *status)
{
    switch(rc)
    {
        case r4success:
            strcpy(status,"r4success");
            break;

        case r4eof:
            strcpy(status,"r4eof");
            break;

        case r4after:
            strcpy(status,"r4after");
            break;

        default:
            strcpy(status,"other");
            break;
    }
}

void printRecord(int rc)
{
    int      ageValue;
    double   amountValue;
    char     fNameStr[15],lNameStr[15];
    char     addressStr[20];
    char     dateSt r[9];
    char     marriedStr[2];
    const char     *commentStr;
    char     status[15];

    f4ncpy(fName,fNameStr,sizeof(fNameStr));
    f4ncpy(lName,lNameStr,sizeof(lNameStr));
    f4ncpy(address,addressStr,sizeof(addressStr));
```

```
    ageValue = f4int(age);
    amountValue = f4double(amount);
    f4ncpy(birthDate,dateStr,sizeof(dateStr));
    f4ncpy(married,marriedStr,sizeof(marriedStr));
    commentStr = f4memoStr(comment);

    seekStatus(rc,status);

    printf("Seek status : %s\ n",status);
    printf("--------------------------------\n");
    printf("Name     : %10s %10s\n",fNameStr,lNameStr);
    printf("Address  : %15s\n",addressStr);
    printf("Age:%3d BirthDate:%8s Married : %1s\n",ageValue,dateStr,marriedStr);
    printf("Comment: %s\n",commentStr);
    printf("Amount purchased : $%5.2lf \n",amountValue);
    printf("\n");
}

int main( void )
{
    int rc;

    code4init(&codeBase);

    OpenDataFile();

    d4tagSelect(dataFile,addressTag);

    rc = d4seek(dataFile,"123 - 45 Ave   ");
    printRecord(rc);

    rc = d4seek(dataFile,"12");
    printRecord(rc);

    rc = d4seek(dataFile,"12            ");
    printRecord(rc);

    d4tagSelect(dataFile,birthdateTag);

    rc = d4seek(dataFile,"19500101");
    printRecord(rc);

    d4tagSelect(dataFile,amountTag);

    rc = d4seekDouble(dataFile,47.23);
    printRecord(rc);

    rc = d4seek(dataFile," 47.23");
    printRecord(rc);

/*The following code finds all the occurrences of John Albridge in the tag*/

    d4tagSelect( dataFile,nameTag) ;
    rc = d4seekNext( dataFile,"Albridge  John      ");
    for (rc; rc == r4success; rc = d4seekNext( dataFile,"Albridge  John      "))
        printRecord( rc );

    code4close(&codeBase);
    code4initUndo(&codeBase);
    return 0;
}
```

Performing Seeks

There are six functions in the CodeBase library that perform seeking: **d4seek** and **d4seekNext** may seek for character data, **d4seekDouble** and **d4seekNextDouble** may seek for numeric data and **d4seekN** and **d4seekNextN** may seek for binary data.

d4seek with character data

d4seek accepts a **(char *)** string as the search key and performs a seek using the default tag, which is the selected tag if one is selected. If there is no selected tag, the default tag is the first tag of the first opened index. **d4seek** can perform seeks using Character, Numeric or Date tags.

Code fragment from SEEKER.C

```
d4tagSelect(dataFile,addressTag);

rc = d4seek(dataFile,"123 - 45 Ave   ");
printRecord(rc);
```

d4seek on Date tags

When performing seeks using Date tags, **d4seek** must be passed an eight character string containing a date in the standard "CCYYMMDD" format. See "Date Functions" chapter for details on using the date functions.

Code fragment from SEEKER.C

```
d4tagSelect(dataFile,birthdateTag);

rc = d4seek(dataFile,"19500101");
printRecord(rc);
```

d4seek on Numeric tags

When **d4seek** uses a Numeric tag, and the search key is a character pointer, the character array is converted into **(double)** value before searching. As a result, you do not have to worry about formatting the character parameter to match the key value.

Code fragment from SEEKER.C

```
d4tagSelect(dataFile,amountTag);

rc = d4seek(dataFile," 47.23");
printRecord(rc);
```

d4seekDouble on Numeric Tags

d4seekDouble accepts a **(double)** value as the search key and this function should only be used on numeric keys. All numeric tags, regardless of the number of decimals, store their key entries as **(double)** values. When seeking on numeric tags, it is more efficient to seek with a **(double)** value than a character representation.

Code fragment from SEEKER.C

```
d4tagSelect(dataFile,amountTag);

rc = d4seekDouble(dataFile,47.23);
printRecord(rc);
```

d4seekN on Binary Data

Use the **int d4seekN(DATA4 *file, char *key, int len)** function to seek on character tags that contain binary data. This seek function can be used to seek without regard for nulls, since *len* specifies the length of the search key.

Code fragment

```
static char key[12] = "\0\0000111\010";

d4tagSelect(dataFile, charTag);
rc = d4seekN(dataFile, key, sizeof(key));
```

d4seekNext, d4seekNextDouble and **d4seekNextN**

The **d4seekNext**, **d4seekNextDouble** and **d4seekNextN** functions only differ from their **d4seek** counterparts in that the searches begin at the current position in the tag. This provides the capability of performing an incremental search through the data base. Consequently, all index keys matching the search key can be found by repeated calls to **d4seekNext** functions. On the other hand, **d4seek** functions always begin their searches from the first logical record as determined by the selected tag and therefore, they only locate the first index key in the tag that matches the search key.

All three **d4seekNext** functions work in the following manner. If the index key at the current position in the tag does not match the search key, **d4seekNext** calls **d4seek** to find the first occurance of the search key. Conversely, if the index key at the current position does match the search key, **d4seekNext** tries to find the next occurrence of the search key. Similarly, **d4seekNextDouble** and **d4seekNextN** call **d4seekDouble** and **d4seekNextN** respectively to find the first occurrence of the search key.

```
d4tagSelect( dataFile,nameTag) ;
rc = d4seekNext( dataFile,"Albridge  John      ");
for (rc; rc == r4success;
            rc = d4seekNext(dataFile,"Albridge  John      "))
    printRecord( rc );
```

d4seek, d4seekDouble, d4seekN, **d4seekNext**, **d4seekNextDouble** and **d4seekNextN** return an integer value indicating the success or failure in locating the key value in the selected tag.

Exact Matches

CodeBase gives you the ability to perform both exact and partial matches on tags. An exact match occurs when the index key in the tag is the same as the search key. When searching using a character tag, the criteria that all **d4seek** and **d4seekNext** functions use for determining an exact match is as follows:

- The characters in the search string must be the same as corresponding characters in the index key.
- The comparison is case sensitive. Therefore a search key of "aaaa" does not match an index key of "AAAA".

  To do a case insensitive search, one must change the indexing strategy, not the search method. Create a tag with the index expression "UPPER(fieldName)" and then a search for "AAAA" will be case insensitive. Therefore, both "Aaaa" and "aAaA" will be found.
- The strings are only compared up to end of the shortest value. Therefore a search key of "abcd" exactly matches an index key of "abcdefg" because only four characters are compared.

Note

You can force CodeBase to compare additional characters by padding the search key out to the full length of the index key. Therefore a search key of "abcd " does not exactly match "abcdefg"

When the **d4seek** or **d4seekNext** functions find an exact match, a value of **r4success** is returned.

Partial Matches

When a seek is performed, the tag is positioned to the place in the tag where the search key should be located. If this position lies between two index keys (ie. the tag was not exactly found), then a partial match has occurred. The record whose index key is directly after the "correct" position is loaded into the record buffer. When this happens, a value of **r4after** is returned by the **d4seek** functions and **r4entry** or **r4after** is returned by the **d4seekNext** functions. Consider the following three seeks.

Code fragment from SEEKER.C

```
d4tagSelect(dataFile,addressTag);

rc = d4seek(dataFile,"123 - 45 Ave   ");
printRecord(rc);

rc = d4seek(dataFile,"12");
printRecord(rc);

rc = d4seek(dataFile,"12             ");
printRecord(rc);
```

Assuming that there is a record in the data file containing "123 - 45 Ave" in the address field: The first time **d4seek** is called, it returns **r4success** because its search key exactly matched and was the same length as the index key. The second call will also return **r4success** because it only compares the first two characters. The third call however, returns **r4after** because it could not find an exact match. All three, however, result in the "123 - 76 Ave." record being loaded into the record buffer.

possible returns from **d4seekNext** functions

When **d4seekNext** is called and the index key at the current position in the tag does not match that of the search key, **d4seek** is called to find the first instance of the search key. Should this seek fail, **r4after** is returned and tag is positioned as discussed above.

On the other hand, if **d4seekNext** is called and index key at the current position does match the search key, **d4seekNext** searches for the next occurrence of the search key. If this seek fails, **r4entry** is returned and the tag is positioned in the same manner as **r4after**.

d4seekNextDouble and **d4seekNextN** function in the same manner as **d4seekNext**.

No Match

If there are no index keys in the tag or if the search key's position is after the last index key, then the **d4seek** and **d4seekNext** functions return **r4eof**.

Seeking On Compound Keys

Performing seeks on compound index keys is slightly more difficult. Before a seek can be performed on a tag with a compound index expression, a search key must be created in a similar manner. For example, if the index expression for the tag is "L_NAME + F_NAME", an example is as follows:

```
d4tagSelect( dataFile,nameTag) ;

rc = d4seek( dataFile,"Krammer   David     ") ;
```

The first part of the search key, "Krammer ", matches the format of field L_NAME and is padded out with blanks to the length of that field. The second part, "David ", matches field F_NAME.

WARNING

The various parts of the search key must be padded out to the full length of their respective fields. Failure to do so may cause an incorrect seek.

A similar strategy is used when seeking on compound index keys built with fields of different types. If the index expression is "PRODUCT + STR(COST, 7, 2)" where 'PRODUCT' is a Character field of length eight and Cost is a Numeric field, then a valid search key is "Dunsel 1234.21". Since the STR() function converts Numeric values to Character values, the search string is constructed with the first eight characters matching the PRODUCT field and the "1234.21" matching the return value of STR().

WARNING

You should note the difference between the dBASE "+" and "-" operators when they are applied to Character values. The "+" includes any trailing blanks when the Character values are concatenated. For example if the index expression is "L_NAME + F_NAME", the search key should be "Krammer David ". The "-" removes any trailing blanks from the first string. The second string is concatenated and the previously removed blanks are added to the end of the concatenated string. As a result, if the index expression is "L_NAME - F_NAME", the search key should be "KrammerDavid ".

Group Files

In Clipper, the .**NTX** indexes must be manually opened each and every time the data file is opened. This causes programming time to be increased, as well as having a good chance of forgetting to open an index file. dBASE IV and FoxPro use compound index files (more than one tag in a file) and production index files (automatically opens when data file opens) to avoid these problems.

CodeBase has introduced group files in order to compensate for this limitation of the .**NTX** file format. A group file allows you to use the same function calls when using .**NTX** index files as you would when using .**CDX** and .**MDX** index files.

This is accomplished by emulating production indexes and multiple tags per index file. The same database that was shown in Figure 4.2 is depicted in Figure 4.5 using .**NTX** index files and group files.

Creating Group Files

A group file is automatically created when using **i4create** or **d4create** when using **.NTX** index compatibility. In addition, a group file can be created for existing index files using a text editor. Simply create a file with a file name extension of ".CGP". Then you enter the names of the index files to be grouped together. Enter one index file name per line.

GROUP FILE PRODUCTS.CGP

For example, here is the actual contents of group file 'PRODUCTS.CGP' from Figure 4.5:

```
PRO_TAG
P_NUM_TAG
```

Bypassing Group Files

Because group files are unique to CodeBase, index files generated with Clipper do not have group files. Under these conditions, it may be more convenient for you to bypass the group files and access the .**NTX** group files directly instead of creating your own group files.

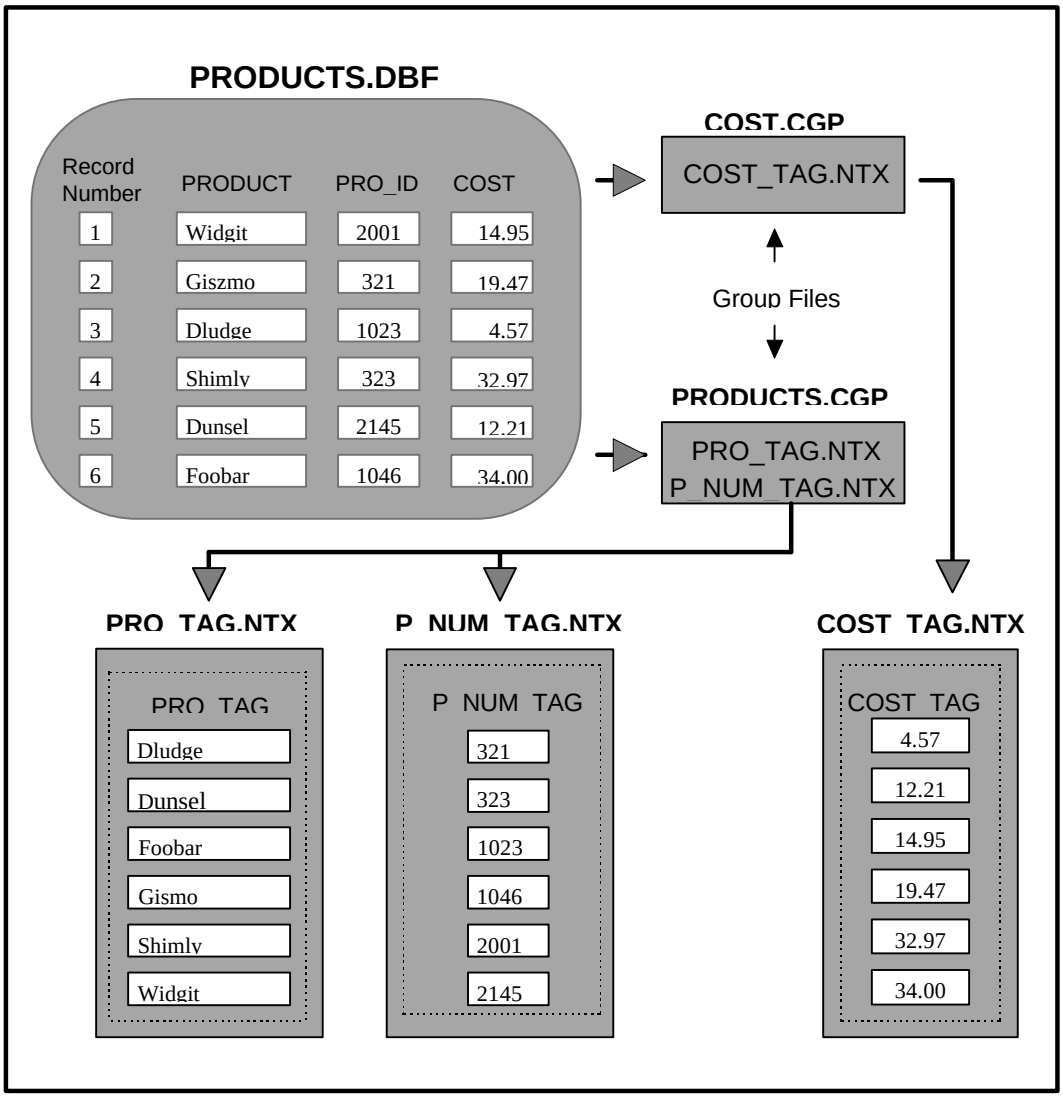


Figure 4.5 Group Files

Disabling The Auto Open

When using **.NTX** index compatibility, CodeBase automatically tries to open a group file when a data file is opened. If you have tried to open a data file that does not have a group file, you will encounter the following error:

```
ERROR #  -60
UNABLE TO OPEN  DATA1.CGP
PRESS ANY KEY TO CONTINUE
```

This results from CodeBase being unable to locate a group file. This behavior can be disabled by setting the **CODE4.autoOpen** flag to false (zero).

Creating Index Files

You can create **.NTX** files without a corresponding group file by passing a null file name to **i4create**. As documented earlier, the tag names also become file names.

PROGRAM NOGROUP1.C This program demonstrates how .**NTX** index files can be created without group files.

```
#include "D4ALL.h" /*THIS PROGRAM ONLY WORKS UNDER CLIPPER*/

CODE4  codeBase;
DATA4  *dataFile = 0;
TAG4   *nameTag,*addressTag,*ageTag,*dateTag;

TAG4INFO tagInfo[] =
{
    {"NAME","L_NAME+F_NAME",0,0,0},
    {"ADDRESS","ADDRESS",0,0,0},
    {"AGE_TAG","AGE",0,0,0},
    {"DATE","BIRTH_DATE",0,0,0},
    {0,0,0,0,0}
} ;

void  main ( void )
{
    code4init(&codeBase);
    codeBase.autoOpen = 0;
    codeBase.safety = 0;
    codeBase.accessMode = OPEN4DENY_RW;

    dataFile = d4open(&codeBas e,"DATA1.DBF");

    i4create( dataFile, 0, tagInfo ) ;

    nameTag = d4tag( dataFile, "NAME" ) ;
    addressTag = d4tag( dataFile, "ADDRESS" ) ;
    ageTag = d4tag( dataFile, "AGE_TAG" ) ;
    dateTag = d4tag(dataFile, "DATE" );

    code4close(&codeBase);
    code4initUndo(&codeBase);
}
```

Opening Index Files

A single .**NTX** index file can also be opened without using group files.

Note

The following example program contains code segments that are specific to CodeBase libraries built with .**NTX** index file compatibility.

PROGRAM NOGROUP2.C This program demonstrates how .**NTX** index files can be opened without using a group file.

```
#include "D4ALL.h"  /*THIS PROGRAM ONLY WORKS UNDER CLIPPER*/

CODE4   codeBase;
DATA4   *dataFile = 0;
FIELD4  *fName,*lName,*address,*age,*birthDate,*married,*amount,*comment;
TAG4    *nameTag,*addressTag,*ageTag,*dateTag;

void printRecords( void )
{
    .
    .
    .

}

 main ( void )
{
    code4init(&codeBase);
    codeBase.autoOpen = 0;
    codeBase.safety = 0;

    dataFile = d4open(&codeBase,"DATA1.DBF");

    nameTag = t4open(dataFile,NULL,"NAME");
    addressTag = t4open(dataFile,NULL,"ADDRESS");
    ageTag = t4open(dataFile,NULL,"AGE_TAG");
```

```
    dateTag = t4open(dataFile,NULL,"DATE");

    d4tagSelect(dataFile,nameTag);
    printRecords();

    code4close(&codeBase);
    code4initUndo(&codeBase);
}
```

The function that provides this functionality is **t4open**. It opens the specified file and returns the **TAG4** pointer of its single tag.

Code fragment from NOGROUP2.C

```
nameTag = t4open(dataFile,NULL,"NAME");
addressTag = t4open(dataFile,NULL,"ADDRESS");
ageTag = t4open(dataFile,NULL,"AGE_TAG");
dateTag = t4open(dataFile,NULL,"DATE");
```

The second parameter of the **t4open** function is an **INDEX4** pointer. This is mainly for internal CodeBase use, so you should set this parameter to null.

Alternatively, **i4open** may be used to open an **.NTX** index file. When doing so, the **.NTX** file name extension must be provided.

Reindexing

Index files become out of date if they remain closed when their data file is modified. Alternatively, a computer could be turned off in the middle of an update. When an index is out of date, the data file may contain records that do not have corresponding tag entries, or the index file may have tag entries that specify the wrong record.

To rectify this problem, it is sometimes necessary to reindex the index files. You could accomplish this in two ways. First you could delete the index file from your system and then recreate the index, or you could open an existing index file and call one of the reindex functions.

Reindexing All Index Files

If you suspect one or all of your index files may be out of date, you can use the **d4reindex** function. This function reindexes any index files that are associated with that data file and are currently open. This includes both production and non-production index files.

```
d4reindex( dataFile );
```

Reindexing A Single Index

If you have several index files but only one is out of date, you can reindex just that index file by using the **i4reindex** function. This function reindexes the index file corresponding to the **INDEX4** pointer you pass to it. All tags associated with the **INDEX4** structure are automatically updated.

```
INDEX4 *indexFile ;

indexFile = d4index( dataFile, "INDEX" ) ;
i4reindex( indexFile ) ;
```

You can obtain the index's **INDEX4** pointer either from the **i4open** or **i4create** functions, or by using **d4index** with the index file's name.

Advanced Topics

The next section contains details on copying the index file structures.

Copying Index File Structures

You may occasionally require a new index file that has an identical structure as an existing index file. This situation usually arises when you are making a copy of a data file.

As a parallel function to **d4fieldInfo,** CodeBase provides the function **i4tagInfo**, which returns the **TAG4INFO** array of an existing index file. You can pass this array to **d4create** or **i4create** to create a new index file.

WARNING

The **TAG4INFO** returned by **i4tagInfo** is allocated dynamically and should therefore be deallocated after you are finished using it. The **u4free** function may be called to free up the memory for this purpose.

In addition, the **TAG4INFO** array contains pointers to the actual tag names, index expression strings and filter expression string; not copies. Do not modify these members or else you will corrupt CodeBase.

PROGRAM COPYDAT2.C

Program COPYDAT2.C takes two file names as arguments. It uses the structure from the data file specified by the first argument to create an empty data file with the name specified by the second argument. A duplicate index file is made if one exists.

```
#include "D4ALL.h"

void main(int argc,char *argv[])
{
    CODE4      codeBase;
    DATA4      *dataFile,*dataCopy;
    INDEX4     *index;
    FIELD4INFO *fieldInfo;
    TAG4INFO   *tagInfo;

    if(argc != 3)
    {
        printf("USAGE:COPYDATA <FROMFILE> <TOFILE>");
        exit(1);
    }

    code4init(&codeBase);
    codeBase.safety = 0;
    dataFile = d4open(&codeBase,argv[1]);
    error4exitTest(&codeBase);

    /* obtain the INDEX4 pointer of the production index, if one exists */

    index = d4index(dataFile,argv[1]);
    if(index) tagInfo = i4tagInfo(index);

    fieldInfo = d4fieldInfo(dataFile);
    d4create(&codeBase,argv[2],fieldInfo,tagInfo);

    u4free(fieldInfo);
    u4free(tagInfo);

    code4close(&codeBase);
    code4initUndo(&codeBase) ;
}
```

5 Queries And Relations

One of the most powerful features of CodeBase is its relation and query capabilities. By using relations, you can turn a collection of separate data files into a fully relational database. Relations are used for two tasks: lookups and queries.

Lookups are used to automatically locate records between related data files. Essentially, this makes several related data files act like a single large data file.

Queries retrieve groupings of records from the database that meet conditions that you can specify at run-time. Queries use CodeBase's Query Optimization which greatly reduces the time it takes to perform queries.

Relations

In its simplest form, a relation is a connection between two data files that specifies how the records from the first data file are used to locate one or more records from the second.

Master & Slave Data File

The data file which controls the others is called the master. The controlled data files are called slaves. A master can have any number of slaves which in turn can be masters of other slaves.

Master Expression & Slave Tag

To create a relation, you usually provide two pieces of information. The first is the *master expression*, and the second is a tag based on the slave data file called the *slave tag*.

The purpose of the master expression is to generate an index key called the *lookup key*. The master expression is evaluated for each record in the master data file and the resulting value is that record's lookup key. A record in the slave is then found by performing a seek using the lookup key on the slave tag.

The most common type of master expressions are ones which only contain the name of a field from the master data file. The slave tag ordering is usually based on an identical field in the slave data file. Figure 5.1 shows a relation of this type.

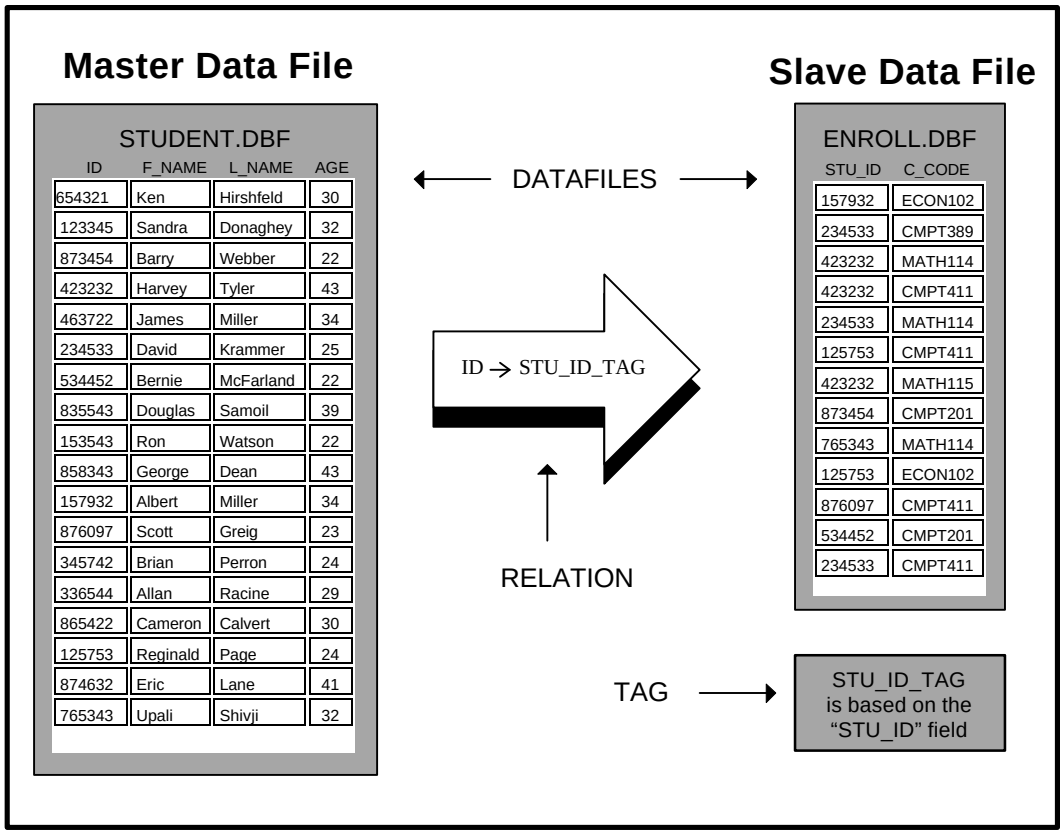


Figure 5.1 Relation on two data files

In this example, the master expression is "ID" and the slave tag is the tag STU_ID_TAG. The index expression for the slave tag is "STU_ID". If, for example, the current record for the master data file is set to record number 4, the master expression would then evaluate to "423232". The lookup on the slave data is then performed, locating record number 3.

Composite Records

Related records from the master and slaves data files are collectively referred to as a *composite record*. The composite record consists of the fields from the master along with the fields from the slave data files. The following is a composite record from the previous relation example:

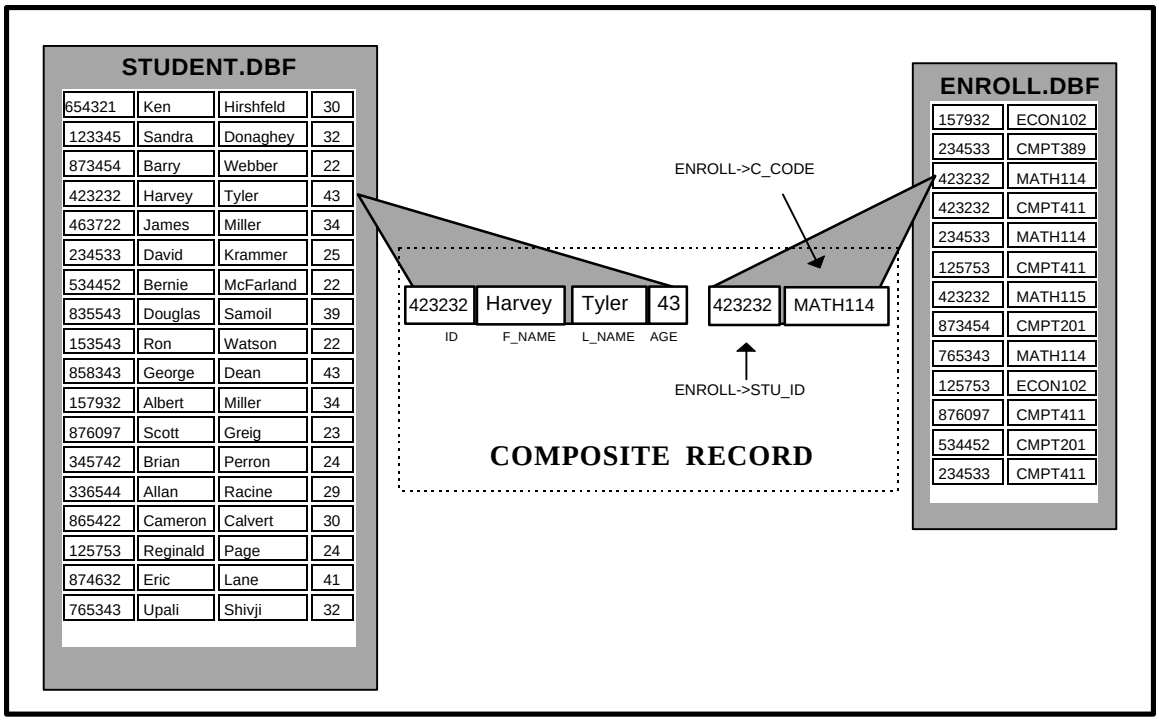


Figure 5.2 A Composite Record

When using fields from the composite record in a dBASE expression, you must precede any field names from slave data files with the field name qualifier. The field name qualifier is the name of the data file followed by the "->" characters. This avoids any potential naming conflicts caused by data files with field names in common.

For example, " ENROLL->STU_ID = 876987" is a dBASE expression containing fields from the slave data file. This expression could be used as part of a query (discussed later in this chapter).

Composite Data Files

The composite data file is the set of composite records that are produced by the relation. Figure 5.3 lists the composite data file for the example relation.

The composite data file does not physically exist on disk. It is merely a way of viewing the information in the data files of the relation.

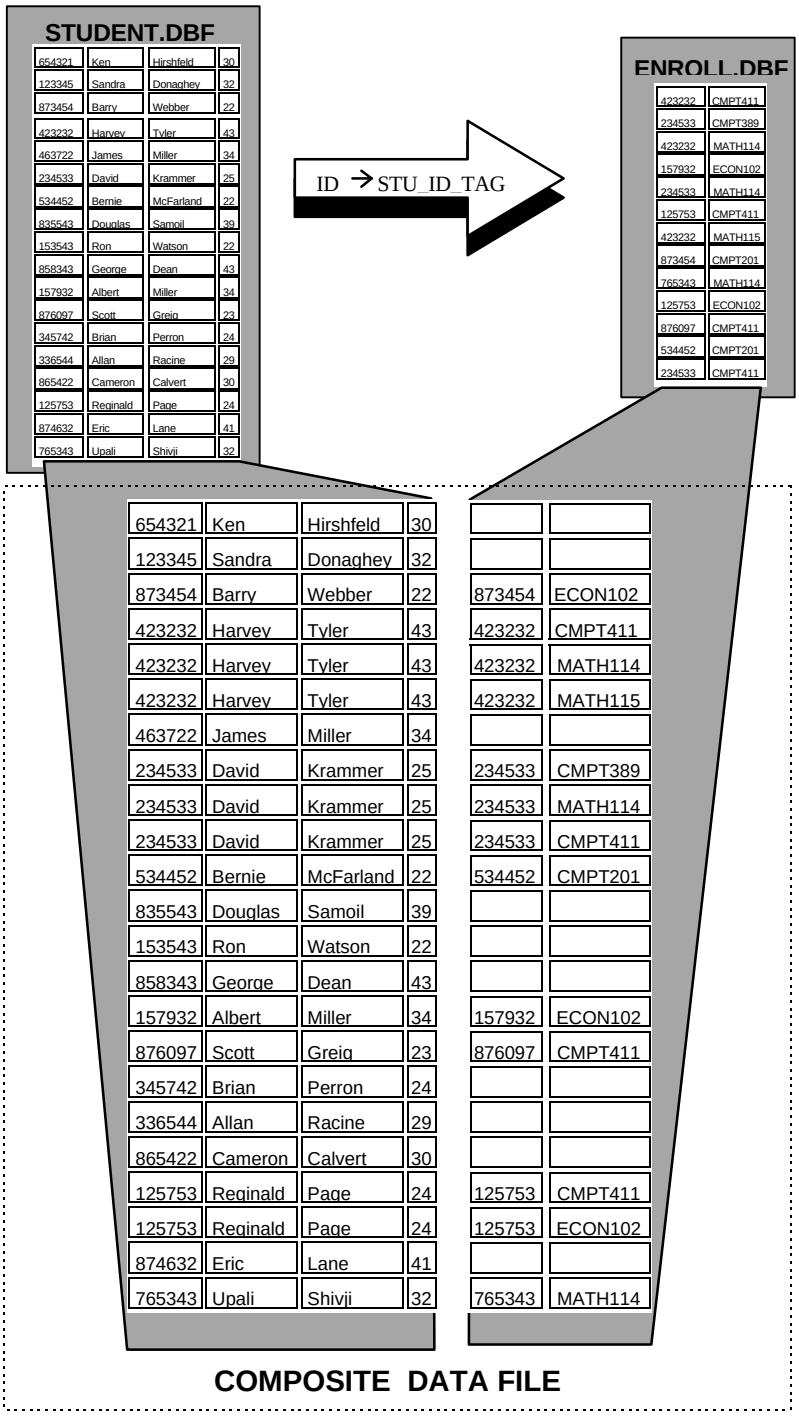
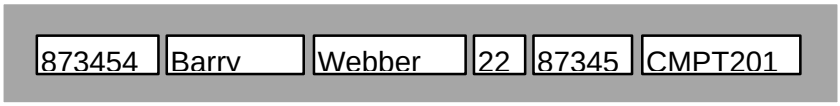


Figure 5.3 The Composite Data File

For example, when the relation is applied to record number 3 of the master data file the following composite record is generated:

Composite Data File Record # 3



By default, if the relation does not find a match in the slave data file, the slave's fields in the composite record are left as blanks. For example record number 1 in the master data file does not have a corresponding slave record, so the generated composite record is:

Composite Data File Record # 1

Complex Relations

Although having a relation between two data files is quite useful, it is often necessary to set relations between one master and several slaves, or to set relations between the slaves and other data files.

Single Master, Multiple Slave Relations

Relations with one master and several slaves are very similar to the single master, single slave relations. In this case there is a master expression and slave tag for each slave, and the composite record consists of fields from all the data files. CodeBase permits any number of slaves for a single master. The only limitations are the resources of your system.

Multi-Layered Relations

There are many situations that require a master with a relation to a slave, which in turn has a relation to other data files. CodeBase supports an unlimited number of these multi-layered relations and will automatically perform lookups in any associated slave data files.

The Top Master

A master data file is a master only in the context of a specific relation. The data file can also be slave in a different relation. If the data file is not a slave in any other relation, it is called a *top master*.

The Relation Set

A relation set consists of a top master and all other connected relations. There must be exactly one top master in a relation set.

Figure 5.4 describes a new relation between the ENROLL.DBF data file and the new data file COURSE.DBF. It shows the entire relation set and points out the various relations between the data files.

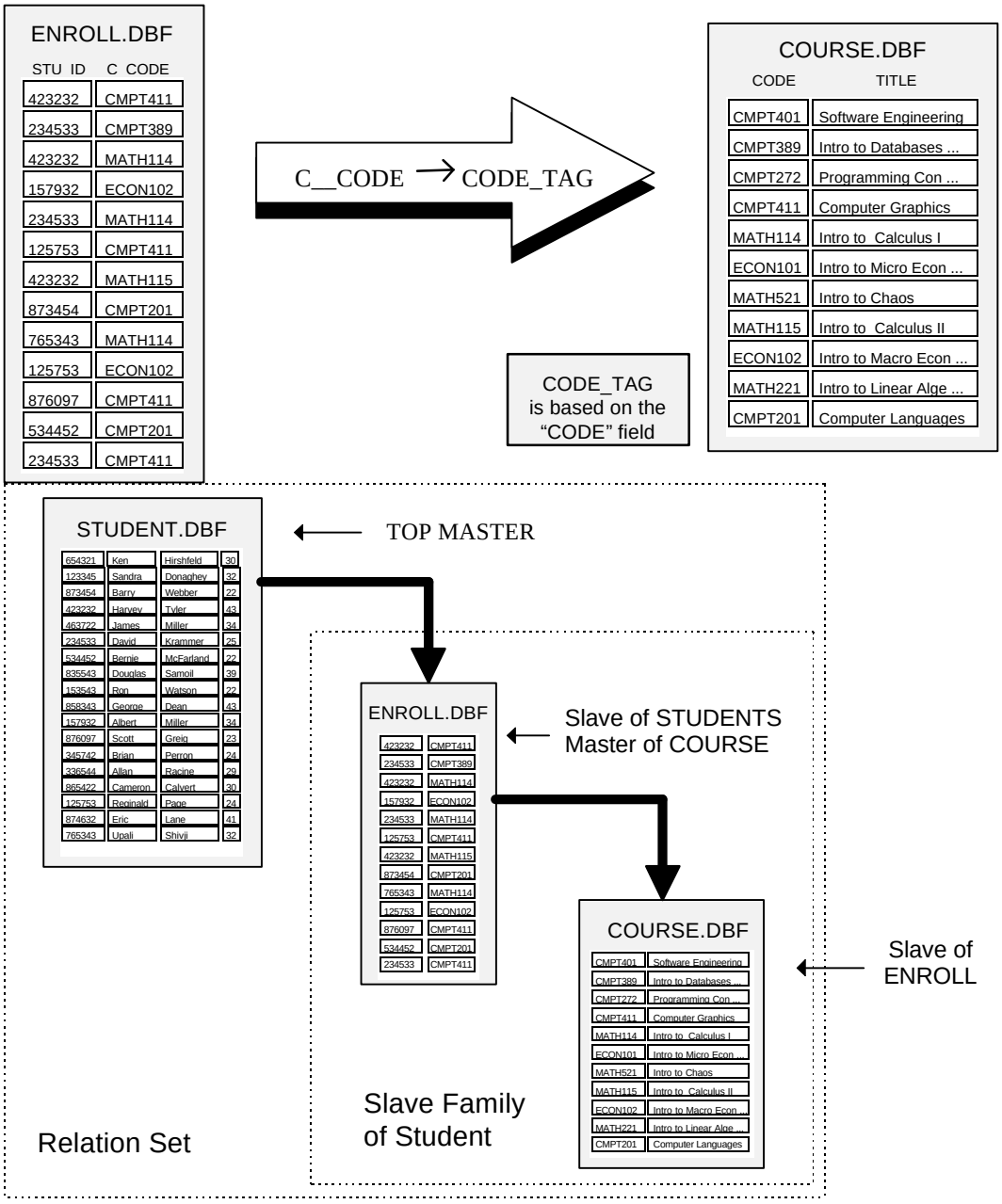


Figure 5.4 A Relation Set

Relation Types

There are three basic types of relations. They are *exact match*, *scan* and *approximate match* relations. The type of relation determines what record or records are located during a lookup.

Exact Match Relations

An *exact match* relation permits only a single match between the master and slave data files. Both one to one and many to one relations are exact match relations. In a one to one relation, there is only one record in the master that matches a single record in the slave. In a many to one relation, there are one or more records in the master that match a single slave record.

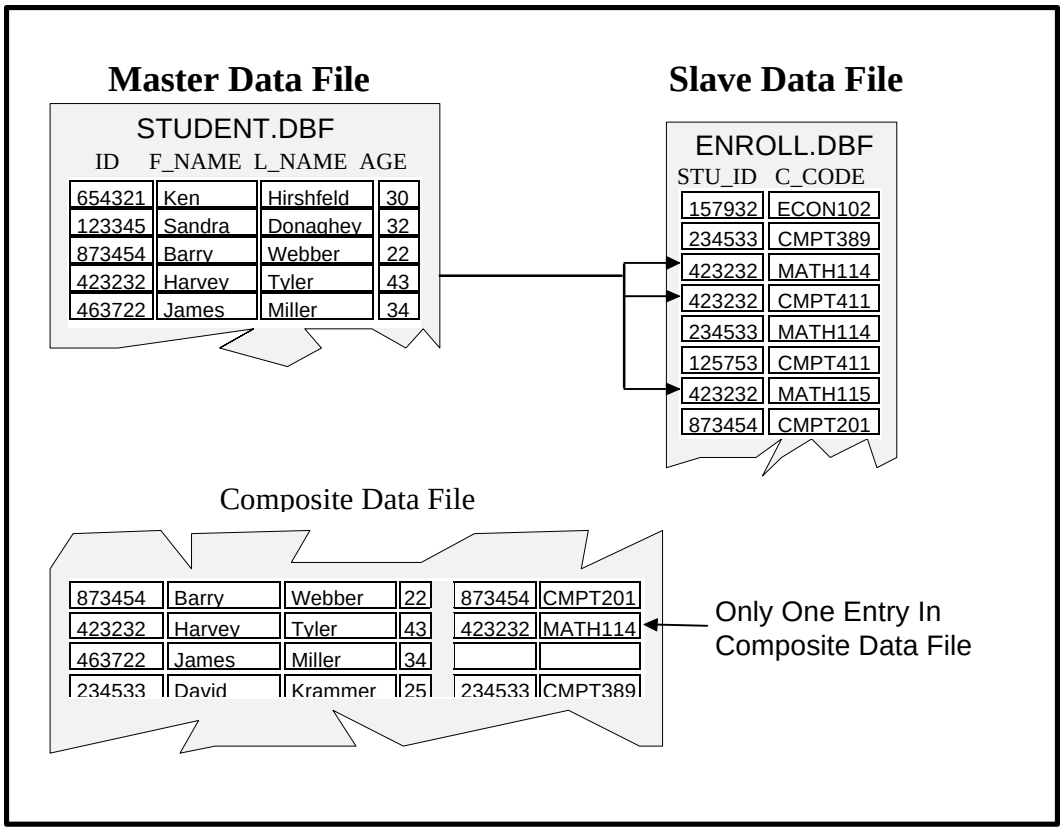


Figure 5.5 An Exact Match Relation

If there are multiple slave records that match a single master record, then a composite record is only generated for the first slave record, as illustrated above. There are three records in the slave data file that match record number 4 of the master. Since this is an exact relation, a composite record is made from only the first matching record of the slave data file.

Scan Relations

In a *scan* relation, if there are multiple slave records for a master record, there is one composite record in the extended data file for each of the matching slave records. Both one to many and many to many relations are scan relations. In a one to many relation, each record in the master may have multiple matching slave records. A many to many relation is the same as one to many except that different master records may match the same slave record.

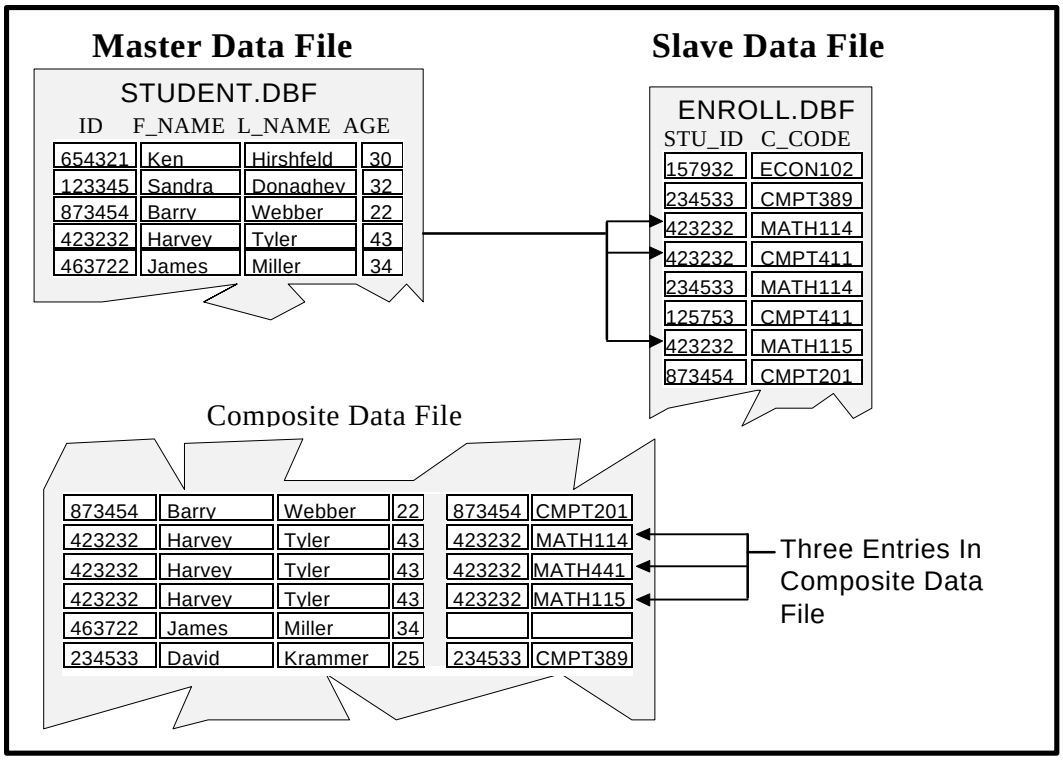


Figure 5.6 A Scan Relation

Approximate Match Relations

The final type of relation is the approximate match relation. This is similar to the exact match relation in that it permits only one match for a master record. The only difference is the way it behaves when an exact match is not found in the slave data file. If the match fails, the slave record whose index key appears next in the slave tag is used instead. Approximate match relations are generally quite rare and are usually used only when a range of records are represented by a single high value.

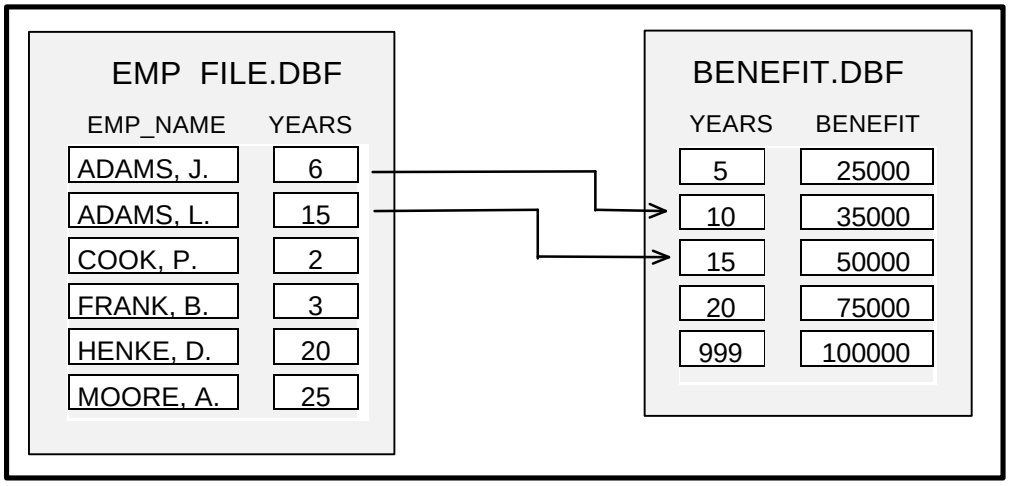


Figure 5.7 Approximate Match Relations

An approximate match relation is shown in Figure 5.7. In this case, the employees' retirement benefits are determined by the number of years that they put into the company. Instead of making an entry for each possible year served, the BENEFIT.DBF only lists the upper limit for each pay out level. The first pay out level is from zero to five years, the second is six to ten, et cetera until the maximum entry of twenty-one years and above pays out 100000.

Creating Relations

Relations are created during the execution of your application. Before you can create a relation, you must have opened all the data files and any index files that are used in the relation set.

PROGRAM RELATE1.C

Program RELATE1.C sets up a scan type relation between the STUDENT.DBF and ENROLL.DBF data files that were illustrated in Figure 5.1. This program lists all of the records in composite data file.

```
/* RELATE1.C */
#include "D4ALL.h"

CODE4   codeBase;
DATA4   *student = NULL,*enrollment = NULL;
RELATE4 *master = NULL,*slave = NULL;
TAG4    *idTag,*nameTag;

void openDataFiles()
{
    code4init(&codeBase);

    student = d4open(&codeBase,"STUDENT");
    enrollment = d4open(&codeBase,"ENROLL");

    nameTag = d4tag(student,"NAME");
    idTag = d4tag(enrollment,"STU_ID_TAG ");

    error4exitTest(&codeBase);
}

void setRelation()
{
    master = relate4init(student);
    if(master == NULL) exit(1);

    slave = relate4createSlave(master,enrollment,"ID",idTag);

    relate4type(slave,relate4scan);

    relate4top(master);
}

void printRecord()
{
    RELATE4 *relation;
    DATA4   *data;
    FIELD4  *field;
    int     j;

    for(relation = master; relation != NULL; relate4next(&relation))
    {
        data = relate4data(relation);

        for(j = 1; j <= d4numFields(data); j++)
            printf("%s ",f4memoStr(d4fieldJ(data,j)));
    }
    printf("\n");
}

void listRecords( void )
{
    int rc;

    for(rc = relate4top(master); rc != r4eof; rc = relate4skip(master,1L))
        printRecord();
```

```
    printf("\n");
    code4unlock(&codeBase);

}

void main( void )
{
    openDataFiles();

    setRelation();

    listRecords();

    relate4free(master,0);

    code4close(&codeBase);
    code4initUndo(&codeBase);
}
```

The RELATE4 Structure

The **RELATE4** structure contains the control information about a single data file in a relation. Every data file that is in the relation set must have its own **RELATE4** structure.

A data file's **RELATE4** structure contains information such as what data file (if any) is the master of this data file and what type of relation exists between the data file and its master.

In a single relation set, a data file can only have one **RELATE4** structure associated with it.

Specifying The Top Master

The first step for creating a relation entails specifying which data file will be the top master of the relation set. This step is accomplished by a call to **relate4init**, which creates and returns a pointer to the **RELATE4** structure for that data file.

Code fragment from RELATE1.C

```
master = relate4init(student);
```

At this point, you have a complete relation set consisting of one data file -- the top master. Why would you want a relation set with just a single data file in it? A single data file relation set may be used when a query involves only the one data file.

Adding Slave Data Files

Adding slave data files to a relation is accomplished by a call to **relate4createSlave**. To do this, there are four pieces of information that you must specify.

The master's **RELATE4** pointer

First, the master data file's **RELATE4** structure must be provided. This can either be the **RELATE4** pointer returned from **relate4init** or from a previous call to **relate4createSlave**.

The slave's **DATA4** pointer

The second piece of information is a pointer to the slave's **DATA4** structure.

WARNING

A data file should only be used once in a relation set. The result of using a data file more than once in a relation set is undefined and unpredictable.

The master expression

The master expression is the third piece of information needed when adding a slave. This dBASE expression is evaluated using the current record of the master data file to produce a lookup key.

A **TAG4** pointer from the slave data file

The final piece of information is a **TAG4** pointer to one of the slave's tags. During a lookup, a search for the lookup key (generated by the master expression) is made using this tag.

After the slave has been added to the relation set **relate4createSlave** returns a pointer to the slave's **RELATE4** structure. You should save this pointer in a variable if you plan on changing the default relation type or on adding slaves to the slave data file.

Code fragment from RELATE1.C

```
slave = relate4createSlave(master, enrollment, "ID", idTag);
```

In the above code fragment, *master* is the **RELATE4** pointer returned from the call to **relate4init** and *enrollment* is the **DATA4** pointer of the slave data file. The master expression is "ID". As a result, the lookup key is the contents of the ID field from the master data file. The lookups are performed using a tag from the slave data file. This tag is identified by *idTag*.

Creating Slaves Without Tags

CodeBase allows an alternative method of performing lookups that does not use tags from the slave data file. Instead of having the master expression evaluate to a lookup key, it evaluates to a record number. This record number specifies the physical record number of the slave record retrieved from the slave data file.

To use this method, simply pass null for the **TAG4** parameter of **relate4createSlave**. This method is mainly useful on static unchanging slave data files that are never packed. This method has the advantage of being faster and more efficient than performing seeks on a tag. The following is a example of this method.

```
slave = relate4createSlave(master, enrollment, "RECNO()", null);
```

Since the master expression is "RECNO()", it returns the record number of master data file's current record. As a result, record **n** in the master must match record **n** in the slave.

If the master expression contains a reference to a non existent record number (<= 0 or > **d4recCount**), CodeBase generates a -70 error ("Reading File", see "Appendix A: Error Codes" in the *Reference Guide*).

Setting The Relation Type

The default type of relation is an exact match relation. You can change a relation's type by calling **relate4type**. This function takes a **RELATE4** pointer and an integer specifying the type of the relation. The valid integer values are specified by the following defined CodeBase constants:

- **relate4exact** (Default) This specifies an exact match relation.
- **relate4approx** This specifies an approximate match relation.
- **relate4scan** This specifies a scan relation.

Since the master data file may have multiple slaves each using a different type of relation, it is the slave's **RELATE4** pointer that is passed to the **relate4type** function. In RELATE1.C the **RELATE4** pointer *slave* is passed to the **relate4type** function.

Code fragment from RELATE1.C

```
relate4type(slave, relate4scan);
```

Setting The Error Action

Sometimes a record in the slave data file cannot be located from a master data file record. This occurs when you are using an exact type relation and there is no match for the lookup key, or when the relation is using direct record lookup and the generated record number does not exist in the slave data file. Under either of these circumstances, there are several ways in which CodeBase can react. **relate4errorAction** specifies which method is used and it accepts a pointer to the slave's **RELATE4** structure and an integer that specifies the type error action. The valid error action codes are as follows:

- **relate4blank** (Default) This means that when a slave record cannot be located, it becomes a blank record.
- **relate4skipRec** This code means that the entire composite record is skipped as if it did not exist.
- **relate4terminate** This means that a CodeBase error is generated and the CodeBase relate function, usually **relate4skip**, returns an error code.

Moving Through The Composite Data File

CodeBase provides three functions for moving through the records in the composite data file. They are **relate4top**, **relate4bottom** and **relate4skip**. These functions are used mainly for obtaining composite records when a query is performed.

Listing The Composite Data File

When used together, these three functions allow you iterate through all of the composite data file. The following code segment demonstrates this process:

Code fragment from RELATE1.C

```
void listRecords( void )
{
    int rc;
    for(rc = relate4top(master); rc != r4eof;
                                  rc = relate4skip(master, 1L) )
        printRecord( ) ;

    printf("\n");
}
```

Finding the first record in the Composite data file

relate4top sets the current record of the master data file to the first record in the composite data file. If there are slaves in the masters slave family, lookups are automatically performed. If there are no composite records in the composite data file, **r4eof** is returned.

Skipping through records

For each iteration of the **for** loop, **relate4skip** is used to move to the next record in the composite data file. Just as in the **d4skip** function, **relate4skip's** second parameter specifies the number of records to be skipped. When there are no more records in the composite data file, **r4eof** is returned.

Note

Before **relate4skip** is used, a call to **relate4top** or **relate4bottom** must be made.

Skipping backwards through the composite data file is also permitted. Since skipping backwards requires extra internal overhead, you must either call **relate4bottom** or **relate4skipEnable** to initialize the backwards skipping abilities.

An example of this is given in the following code segment:

```
void listRecords( void )
{
    int rc;

    for(rc = relate4bottom(master); rc != bof;
                                  rc = relate4skip(master, -1L))
        printRecord();
}
```

Queries And Sorting

Once a relation has been set up, you can then perform queries upon the composite data file. A *query* allows you to retrieve selected records from the composite data file by specifying a query expression. Essentially the query specifies a subset of the composite data file, which is called the *query set*, by filtering out any composite records that do not meet the search criterion.

In addition to performing queries, you can also specify the order in which the composite records are presented.

When a query is specified, CodeBase uses its Query Optimization, whenever possible, to minimize the number of necessary disk accesses. This greatly improves performance when retrieving composite records from the query set.

PROGRAM RELATE2.C

Program RELATE2.C demonstrates how queries can be performed on a single data file.

```
/* RELATE2.C */

#include "D4ALL.h"

CODE4   codeBase;
DATA4   *student = NULL;

void openDataFiles()
{
    student = d4open(&codeBase,"STUDENT");
    error4exitTest(&codeBase);
}

void printRecord(DATA4 *dataFile)
{
    int j;

    for(j=1;j<=d4numFields(dataFile);j++)
        printf("%s ",f4memoStr(d4fieldJ(dataFile,j)));

    printf("\n");
}

void query(DATA4 *dataFile,char *expr,char *order)
{
    RELATE4 *relation = NULL;
    int     rc;

    relation = relate4init(dataFile);
    if(relation == NULL) exit(1);

    relate4querySet(relation,expr);
    relate4sortSet(relation,order);

    for(rc = relate4top(relation);rc != r4eof;rc = relate4skip(relation,1L))
        printRecord(dataFile);

    printf("\n");

    code4unlock(&codeBase);
    relate4free(relation,0);
}

void main( void )
{
    int     rc,numFields,j;

    code4init(&codeBase);
    openDataFiles();

    query(student,"AGE > 30","");

    query(student,"UPPER(L_NAME) = 'MILLER'","L_NAME + F_NAME");

    code4close(&codeBase);
    code4initUndo(&codeBase);
}
```

The Query Expression

To generate a query, all you need to provide is a dBASE expression, called the *query expression*, which evaluates to a Logical value. This expression is used to determine whether a composite record should be included in the query. If the query expression evaluates to .TRUE. then the record is kept, otherwise it is discarded from the query.

Setting the query expression

The query expression is specified for a relation set by the **relate4querySet** function. This function takes a pointer to the top master's **RELATE4** structure and specifies the query expression for the entire relation set.

The query expression can reference fields from different files in the relation

set. When a query is made, the default data base used is the master. This means that the field will automatically be associated with the master data base. To refer fields of different data files use the file's alias followed by an arrow (->) and the field name.

For example, suppose a relation consists of a master data file called FATHER.DBF and a slave data file called SON.DBF. Assume that each file has a field called NAME.

For these data files, the following queries are equivalent:

```
relate4querySet( relation, "FATHER->NAME = 'Ben Nyland'") ;
relate4querySet( relation, "NAME = 'Ben Nyland'") ;
```

These queries compare a field called NAME in FATHER.DBF with the string "Ben Nyland". The file name FATHER does not need to be specified in the query because the FATHER.DBF file is a master.

In order to check for the composite record that has Ben Nyland and his son Eric Nyland, the query can be specified by either of the following:

```
relate4querySet( relation, "FATHER->NAME = 'Ben Nyland'.AND. SON->NAME = 'Eric Nyland'");
relate4querySet( relation, "NAME = 'Ben Nyland' .AND. SON->NAME = 'Eric Nyland'");
```

The query expression can be changed at any time, although **relate4top** or **relate4bottom** must then be called before **relate4skip** can be called.

The Sort Expression

The *sort expression* of a relation is very similar to an index expression. They are both dBASE expressions, and are both used to determine the sort ordering.

The difference is that the sort expression determines the sort order of the query set and you are allowed to use fields from any data file in the relation set.

Setting the sort expression

The sort expression is specified for a relation set by **relate4sortSet**. This function takes a pointer to the top master's **RELATE4** structure and specifies the sort ordering for the entire query set.

The sort expression can contain field names from any data file in the relation set. It is required that any fields, except those from the top master, are qualified by the data file's alias.

```
relate4sortSet( relation, "L_NAME + F_NAME + CLASSES->C_CODE");
```

Like the query expression, the sort expression can be changed at any time, although **relate4top** or **relate4bottom** must then be called before **relate4skip** can be called.

Performing Queries on Relation Sets

Queries can be performed on relation sets of any size, including those consisting of a single data file. The following figures illustrate two queries. They are on the composite data file shown in Figure 5.3.

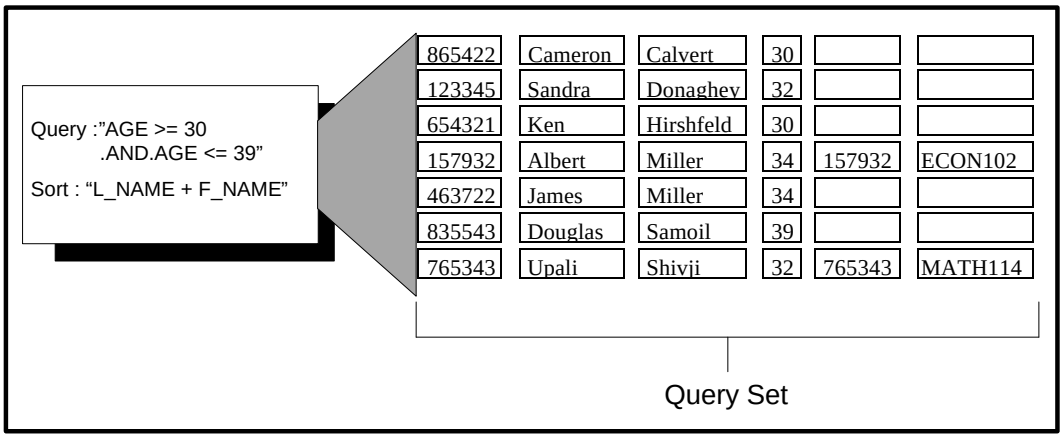


Figure 5.8 Query Example One

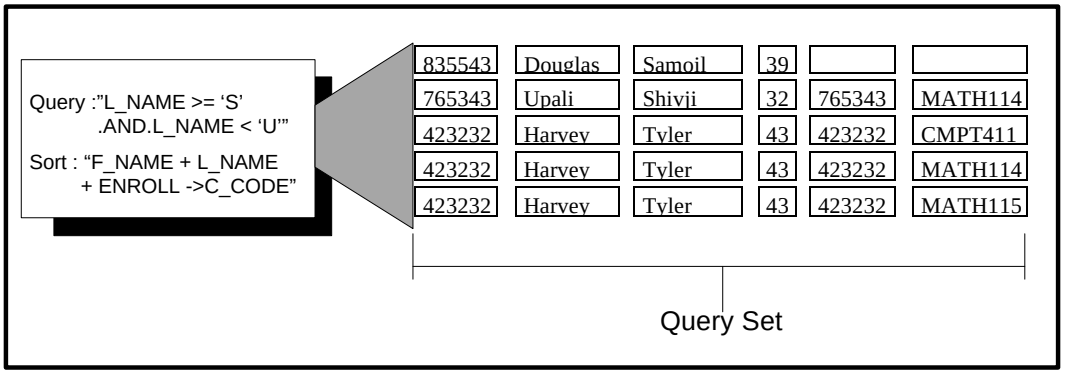


Figure 5.9 Query Example Two

Accessing The Query Set

The query set is accessed by **relate4top** , **relate4bottom** and **relate4skip**. When a query has been specified, these three functions then access a query set instead of the entire composite data file.

Generating The Query Set

The query set is initially formed by the first call to either **relate4top** or **relate4bottom**. These functions analyze the query expression for the most efficient way of performing the query. They then locate all of records that are in the query set. If a sort expression has been specified, the composite records are sorted. And finally, lookups are performed on the data files in the relation to locate the first or last composite record in the query set.

WARNING

When complex relations and sort expressions are used on a relation set consisting of large data files, **relate4top** and **relate4bottom** can take considerable time to execute.

As long as the query and sort expressions are not changed, further calls to **relate4top** and **relate4bottom** do not cause the query set to be regenerated.

Regenerating the Query Set

You can force CodeBase to completely regenerate the query set by calling **relate4changed**. This forces CodeBase to discard any buffered information and regenerate the query set based on the current state of the relation set's data files. After **relate4changed** is called, **relate4top** or **relate4bottom** must be called before any further calls to **relate4skip** may be made.

Queries On Multi-Layered Relation Sets

The steps involved in performing queries on multi-layered relation sets are the same as on a relation set containing a single data file. You create your relations, set the types of the relations, specify query and sort expressions and retrieve the composite records from the query set.

PROGRAM RELATE3.C

Program RELATE3.C creates a multi-layered relation between the STUDENT, ENROLL and CLASSES data files and then performs queries on it.

```
/* RELATE3.C */

#include "D4ALL.H"

CODE4   codeBase;
DATA4   *student,*enrollment,*classes;
FIELD4  *studentId,*firstName,*lastName,*age,*classCode,*classTitle;
TAG4    *codeTag,*idTag;
RELATE4 *classRel,*studentRel,*enrollRel;

void openDataFiles()
{
    code4init(&codeBase);

    student = d4open(&codeBase,"STUDENT");
    enrollment = d4open(&codeBase,"ENROLL");
    classes = d4open(&codeBase,"CLASSES");

    studentId = d4field(student,"ID");
    firstName = d4field(student,"F_NAME");
    lastName  = d4field(student,"L_NAME");
    age = d4field(student,"AGE");
    classCode = d4field(classes,"CO DE");
    classTitle = d4field(classes,"TITLE");

    idTag = d4tag(student,"ID_TAG");
    codeTag = d4tag(enrollment,"C_CODE_TAG");

    error4exitTest(&codeBase);
}

void printStudents()
{
    printf("\n        %s ",f4str(firstName));
    printf("%s ",f4str(lastName));
    printf("%s ",f4str(studentId));
    printf("%s ",f4str(age));
}

void setRelation( void )
{
    classRel = relate4init(classes);

    enrollRel = relate4createSlave(classRel,enrollment,"CO  DE",codeTag);

    studentRel = relate4createSlave(enrollRel,student,"STU_ID",idTag);
}

void printStudentList(char *expr,long direction)
{
    int     rc,endValue;

    relate4querySet(classRel,expr);
    relate4sortSet(classRel,"STUDENT->L_NAME + STUDENT->F_NAME");

    relate4type(enrollRel,relate4scan);
```

```
    if(direction > 0)
    {
        rc = relate4top(classRel);
        endValue = r4eof;
    }
    else
    {
        rc = relate4bottom(classRel);
        endValue = r4bof;
    }

    printf("\n%s",f4str(classCode));
    printf("  %s\n",f4str(classTitle));

    for(;rc != endValue;rc = relate4skip(classRel,direction))
        printStudents();
}

void main( void )
{
    openDataFiles();

    setRelation();

    printStudentList("CODE = 'MATH114 '",1L);

    printStudentList("CODE = 'CMPT411 '",-1L);

    relate4free(classRel,0);

    code4close(&codeBase);
    code4initUndo(&codeBase);
}
```

Lookups On Relations

The method illustrated above for retrieving composite records from the composite data file is well suited for performing queries and generating reports, but has unnecessary overhead if all you want to do is perform lookups to slave data files. As a result CodeBase provides an alternative method for performing lookups that is independent of the query set and sort orderings.

PROGRAM RELATE4.C

Program RELATE4.C sets up an exact match type relation between the STUDENT.DBF and ENROLL.DBF that were illustrated in Figure 5.1. This program performs some seeks and lookups.

```
/* RELATE4.C */

#include "D4ALL.h"

#ifdef __TURBOC__
    extern unsigned _stklen = 10000;
#endif

CODE4   codeBase;
DATA4   *student = NULL,*enrolment = NULL;
FIELD4  *id,*fName,*lName,*age,*cCode;
RELATE4 *master = NULL,*slave = NULL; TAG4   *idTag,*nameTag;

void openDataFiles()
{
    coder4init(&codeBase);

    student = d4open(&codeBase,"STUDENT");
    enrolment = d4open(&codeBase,"ENRO LL");

    id = d4field(student,"ID");
    fName = d4field(student,"F_NAME");
    lName = d4field(student,"L_NAME");
    age = d4field(student,"AGE");
    cCode = d4field(enrolment,"C_CODE");

    nameTag = d4tag(student,"NAME");
    idTag = d4tag(enrolment,"STU_ID_TAG");

    error4exitTest(&codeBase);
```

```
}

void setRelation()
{
    master = relate4init(student);
    slave = relate4createSlave(master,enrolment,"ID",idTag);
}

void seek(DATA4 *dataFile,TAG4  *tag,RELATE4 *relation,char *key)
{
    TAG4 *oldTag;

    oldTag = d4tagSelected(dataFile);
    d4tagSelect(dataFile,tag);

    d4seek(dataFile,key);
    relate4doAll(relation);

    d4tagSelect(dataFile,oldTag);
}

void printRecord()
{
    printf("%15s ",f4str(fName));
    printf("%15s ",f4str(lName));
    printf("%6s ",f4str(id));
    printf("%2s ",f4str(age));
    printf("%7s\n",f4str(cCode));
}

void main( void )
{
    openDataFiles();

    setRelation();

    seek(student,nameTag,master,"Tyler           Harvey             ");
    printRecord();

    seek(student,nameTag,master,"Miller          Albert             ");
    printRecord();

    code4unlock(&codeBase);
    relate4free(master,0);

    code4close(&codeBase);
    code4initUndo(&codeBase);
}
```

Performing A Lookup

Performing lookups is quite simple. First you locate the record in the master data file that you wish to perform the lookup from, using the normal **d4top**, **d4bottom**, **d4go**, **d4seek, d4seekDouble, d4seekN, d4seekNext, d4seekNextDouble, d4seekNextN** and **d4skip** functions. The lookup is then accomplished by a call to **relate4doAll**. This function is passed a **RELATE4** pointer of a master data file and it performs lookups on the slave data files of that master. In addition, it travels down the relation set performing lookups on the slave's slaves.

The following function performs a seek on a data file and then performs the lookups on it's slaves.

Code fragment from RELATE4.C

```
void seek( DATA4 *dataFile, TAG4  *tag, RELATE4 *relation,
                                                        char *key)
{
    TAG4 *oldTag;

    oldTag = d4tagSelect( dataF ile ) ;
    d4tagSelect( dataFile, tag ) ;

    d4seek( dataFile,key ) ;
    relate4doAll( relation ) ;

    d4tagSelect( dataFile, oldTag ) ;
}
```

relate4doOne

A similar function to **relate4doAll** is **relate4doOne**. This function accepts the **RELATE4** pointer to a slave and performs a lookup on that slave.

Note

Any query and sort expressions are ignored by functions **relate4doAll** and **relate4doOne**. Consequently, these functions provide somewhat independent functionality. Using them in conjunction with relate functions such as **relate4top, relate4bottom,** and **relate4skip** is not particularly useful.

Iterating Through The Relations

As an aid in writing generic reusable code, CodeBase provides the function **relate4next**, which allows you to iterate through the relations in a relation set.

This function takes the address of a **RELATE4** pointer and changes it to a pointer to the next **RELATE4** structure in the relation set.

WARNING

The **relate4next** function changes the value of the pointer that is passed in as a parameter. Consequently, you should make a copy of this pointer if you need it later.

The following code segment displays all of the records from all of the relations in the relation set.

Code Fragment from RELATE1.C

```
void printRecord()
{
    RELATE4 *relation;
    DATA4   *data;
    FIELD4  *field;
    int     j;

    for(relation = master; relation  != NULL; relate4next(&relation))
    {
        data = relation->data;

        for(j = 1; j <= d4numFields(data); j++)
            printf("%s ",f4memoStr(d4fieldJ(data,j)));
    }
    printf("\n");
}
```

6 Query Optimization

What is Query Optimization

Query Optimization is an application invisible method of returning query results at high speed. The CodeBase Relation/Query module uses Query Optimization to analyze the query condition and return the matching set of records with a minimum of disk accesses.

This is accomplished by comparing the query expression to the tag sort orderings of the top master tag. If part of the query matches the tag expression, the tag itself is used to filter out records that do not match the expression.

This results in lightning quick performance, even on the largest data files, since very few records are actually physically read. For example, if there were an index built on PRODUCT and the query was:

PRODUCT = 'GIZMO'

The Query Optimization would automatically use the index file to quickly seek to the appropriate record and, using the PRODUCT tag, and skip to retrieve all the records where PRODUCT is equal to 'GIZMO'. Once this is no longer the case, Query Optimization ignores the remaining records.

It makes little difference whether the data file has 500 records or 500,000 records, since the same number of disk reads are performed in each case.

Complex expressions

Query Optimization can also work on more complex expressions involving the .AND. and .OR. operators. In these cases, Query Optimization breaks down the whole expression into sub-expressions that can be optimized. For example, the two sub-expressions in the query:

L_NAME = "SMITH" .AND. F_NAME = "JOE"

could both be optimized if there were two tags, one based on the expression L_NAME, the other on F_NAME. CodeBase could still partially optimize the query if there was a tag on either of the two sub-expressions. Even a partially optimized query can lead to very fast performance.

When is Query Optimiztion used

To start with, the Query Optimization capabilities are built into the CodeBase relate/query module, the report module and into CodeReporter. Your applications can use the Query Optimization provided a few simple conditions are met.

Requirements of Query Optimization

To ensure that the Query Optimization is used to return your queries, the following steps must be taken:

1. Insure that the Master data file's index file(s) are open. In general, the more tags open on the master data files, the greater the chance that Query Optimization can take effect.
2. Specify a query expression that contains in whole or in part, a top master tag key compared to a constant.
3. The query expression is set using **relate4querySet**. Query optimization is only effective when the query expression involves the top master data file.

The following tables illustrate some of the situations where Query Optimization is used automatically, and where it cannot be used. The information in the tables is based on a data file that has the following fields and tags:

```
FIELD4INFO fieldInfo [] =
{
    {"L_NAME", r4str, 10, 0},
    {"F_NAME", r4str, 10, 0},
    {"COMPANY", r4str, 10, 0},
    {"AGE", r4num, 2, 0},
    {"DT", r4date, 8, 0},
    {"AMOUNT", r4num, 7, 2},
    {"ADDRESS", r4str, 15, 0},
    {"COMMENT", r4memo, 10, 0},
    {"PRODUCT", r4str, 10, 0},
    {"ID", r4str, 5, 0},
    {"PHONE", r4str, 8, 0},
    {0,0,0,0},
};

TAG4INFO tagInfo [] =
{
    {"L_NAME_TAG", "L_NAME", 0,0,0},
    {"COMPANY_TAG", "UPPER(COMPANY)", 0,0,0},
    {"AGE_TAG", "AGE", 0,0,0},
    {"DT_TAG", "DT", 0,0,0},
    {"AMNT _TAG", "AMOUNT", ".NOT.DELETED()",0,0},
    {"ADDR_TAG","ADDRESS", "DELETED().AND..NOT.DELETED(), 0,0},
    {"COMMENT_TAG", "COMMENT", 0,0, r4descending},
    {"PROD_TAG", "PRODUCT", 0, r4uniqueContinue, 0},
    {"PRO_ID_TAG ", "PRODUCT+ID", 0,0,0},
    {"PHONE_TAG", "PHONE", 0, r4unique, 0},
    {0,0,0,0,0},
};
```

Query optimization is automatically used in the following situations.

Query Expression	Explanation
L_NAME = "SMITH"	L_NAME is a tag expression which is compared to a constant.
L_NAME >= "S" .AND. L_NAME <= "T"	Each sub expression compares a tag expression against a constant, and so the entire query can use Query Optimization.
L_NAME = "SMITH" .AND. F_NAME = "JOE"	Only the first sub expression can use Query Optimization because there is no tag based on F_NAME.
UPPER(COMPANY) = "IBM"	Again, this is a tag expression compared to a constant. Even though the expression contains a function (UPPER), Query Optimization is still used.
UPPER(COMPANY) = "IBM" .AND. L_NAME="SMITH"	Both parts of the expression use Query Optimization, since both sides compare a tag expression to a constant value.
AGE > 50	Even though the tag is Numeric, the expression can still be optimized.
AGE > 25*2	25*2 is still a constant even though it is a complicated constant.
DTOS(DT) >= "19930101"	This would return all records where the date value in the DT field is greater than or equal to January 1,1993. If the tag expression was DT instead of DTOS(DT), then a query expression such as DT>=CTOD("01/01/93") would also use Query Optimization. (You have to be careful when using CTOD because the format of its parameter depends on the data picture set in **code4dateFormat**.)
COMMENT = "SALE"	In this case the tag was set using **r4descending**. Query Optimization can be used because CodeBase is indifferent to the ordering of the tag.
PHONE = "555 6031"	Query Optimization can utilize unique tags as long the tag has not been specified with **r4uniqueContinue. r4unique** prevents duplicate records from being added to the data base. Therefore, **r4unique** does not act as a filter for the tag, unlike **r4uniqueContinue**. Note that when the index is opened, the unique code is automatically set to the default value of **CODE4.errDefaultUnique**, which is **r4uniqueContinue.** Use **t4uniqueSet** to set unique code to **r4unique** before making a query. See the "Unique Keys" section of the "Indexing" chapter of this guide for more information.

The following expressions cannot use the Query Optimization. Queries on these expressions are not optimized and take a longer time to execute.

Query Expression	Explanation
F_NAME = "JOE"	There is no tag based on F_NAME.
COMPANY = "IBM"	There is no tag based on COMPANY. The tag is UPPER(COMPANY). Again, the tag expression must match the expression being compared to the constant exactly.
L_NAME = F_NAME	This expression cannot be optimized because F_NAME is not a constant.
AMOUNT = 1000.00	Query Optimization will not be used because the tag contains a filter.
ADDRESS = "104 ELM ST"	Query Optimization will not be used because the tag contains a filter, even though the filter is useless. Tags with filters are useless, regardless of the filter result.
PRODUCT ="GIZMO"	Query Optimization will not be used because the tag specifies **r4uniqueContinue**. **r4uniqueContinue** acts like a filter, allowing duplicate records in the database but not in the index.
PRODUCT + ID ="GIZMO 13445"	Query Optimization will not be used because the tag consists of two

	fields.

How To Use Query Optimization

Query Optimization is automatically enabled whenever a relation operation occurs. CodeBase uses Query Optimization on the top master data file transparently when retrieving records from the composite data file.

Query Optimization, as described above, operates only on the top master data file and only when a tag sort ordering corresponds to a portion of the query expression. All you need to do is ensure that the top master data file has an appropriate index file open.

How to tell if Query Optimization can be used

It is possible to determine whether Query Optimization can be used by calling the function **relate4optimizeable.** This function returns true (non-zero) when the relation is able to use the Query Optimization and it returns false (zero) when it can not. If there is insufficient memory, the Query Optimization will not be invoked, even if **relate4optimizeable** returns true (non-zero).

Call **relate4optimizeable** before functions that may use Query Optimization. If false (zero) is returned, it may be possible to create a new index file for those queries that can not use Query Optimization. In the above scenario, the expression "F_NAME = 'JOE' ", is not able to use Query Optimization. If a new index were built on "F_NAME", the execution time of the query would be improved.

Memory Requirements of Query Optimization

The activation of the Query Optimization also depends on the compiler.

When a 16 bit compiler is used, the Query Optimization will be able to handle 500,000 records. If the data base has more than half a million records, the query will be made without using Query Optimization.

It is recommended that 32 bit compiler be used when using data bases that have more than 500,000 records. Query Optimization will be able to handle 32 billion records when a 32 bit compiler is used.

7 Date Functions

The date functions allow you to convert between a variety of date formats and to perform date arithmetic. The date functions also allow you to retrieve the various components of a date, such as the day of the week, in numeric form.

Date Pictures

The format of a date string is represented by a date picture string. A date picture is a string containing several special formatting characters and other characters. The special formatting characters are:

- **C** Century. A 'C' represents the first digit of the century. If two 'C's appear together, then both digits of the century are represented. Additional 'C' s are not used as formatting characters.
- **Y** Year. A 'Y' represents the first digit of the year. If two 'Y's appear together, they represent both digits of the year. Additional 'Y' s are not used as formatting characters.
- **M** Month. One or two 'M's represent the first or both digits of the month. If there are more than two consecutive 'M's, a character representation of the month is returned.
- **D** Day. One or two 'D' s represent the first or both digits of the day of the month. Additional 'D' s are not used as formatting characters.
- **Other Characters** Any character which is not mentioned above is placed in the date string when it is formatted.

For example, if the date August 4 1994 is converted to a date string using the date picture "MM/DD/CCYY", the resulting date string is "08/04/1994". If the same date is converted with the date picture "MMMMMMMM DD, YY" the resulting date string is "August 04, 94".

Date Formats

Dates can be stored in many formats. The two used by CodeBase are the standard format and the Julian day format.

PROGRAM DATE1.C

Program DATE1.C uses several of the date functions to calculate the number of days until Christmas and until your next birthday.

```
/* DATE1.C */

#include "D4ALL.h"

int  validDate(char *date)
{
     long rc;

     rc = date4long(date);

     if(rc < 1)
          return 0;
     else
          return 1;
}

void howLongUntil(int month,int day,char *title)
{
     char todayStandard[9],today[25],date[9],*dow;
     int  year,days;
     long julianToday,julianDate;
```

```
    memset(todayStandard,NULL,sizeof(todayStandard));
    memset(today,NULL,sizeof(today));
    memset(date,NULL,sizeof(date));

    date4today(todayStandard);
    date4format(todayStandard,today,"MMM DD/CCYY");

    printf("\nToday's date is %s\n",today);

    /*convert todayStandard to julian day format*/

    julianToday = date4long(todayStandard);

    year = date4year(todayStandard);
    sprintf(date,"%4d%2d%2d",year,month,day);

    julianDate = date4long(date);

    /* date has be en passed this year */

    if(julianDate < julianToday)
    {
        year ++;
        sprintf(date,"%4d%2d%2d",year,month,day);
        julianDate = date4long(date);
    }

    /* calculate the number of days to the date */

    days = julianDate - julianToday;

    printf("There are %d days until %s",days,title);

    dow = date4cdow(date);

    printf("(which is a %s this year)\n",dow);
}

void main( void )
{
    char birthdate[80],standard[9];

    howLongUntil(12,25,"Christmas ");

    do
    {
        printf("\nPlease enter your birthdate");
        printf(" in \"DEC 20/1993\" format:");
        gets(birthdate);
        date4init(standard,birthdate,"MMM DD/CCYY");
    }
    while(!validDate(standard));
        howLongUntil(date4month(standard),date4day(standard), "your next birthday");
}
```

Standard Format

Dates are stored in the data file in "CCYYMMDD" format. This is known as the standard format. The **f4str** function returns a date from a Date field in this standard format. **f4assign** performs the opposite operation by storing a standard format date string to the data file.

Note

If **f4assign** is passed a date string in anything other than standard format, indexing and seeking will not be performed correctly.

Julian Day Format

Another important date format is the Julian day format, which is stored as **(long).** A Julian day is defined as the number of days since Jan. 1, 4713 BC. Having the date accessible in this format lets you perform date arithmetic. Two dates can be subtracted to find the number of days separating them, or an integer number of days can be added to a date.

Converting Between Formats

Since dates are only stored in the data file in standard format, conversion functions have been provided for converting between standard and julian day format, as well as converting between standard and any other format.

The interrelationships between the data file, date formats and date functions are illustrated by Figure 7.1.

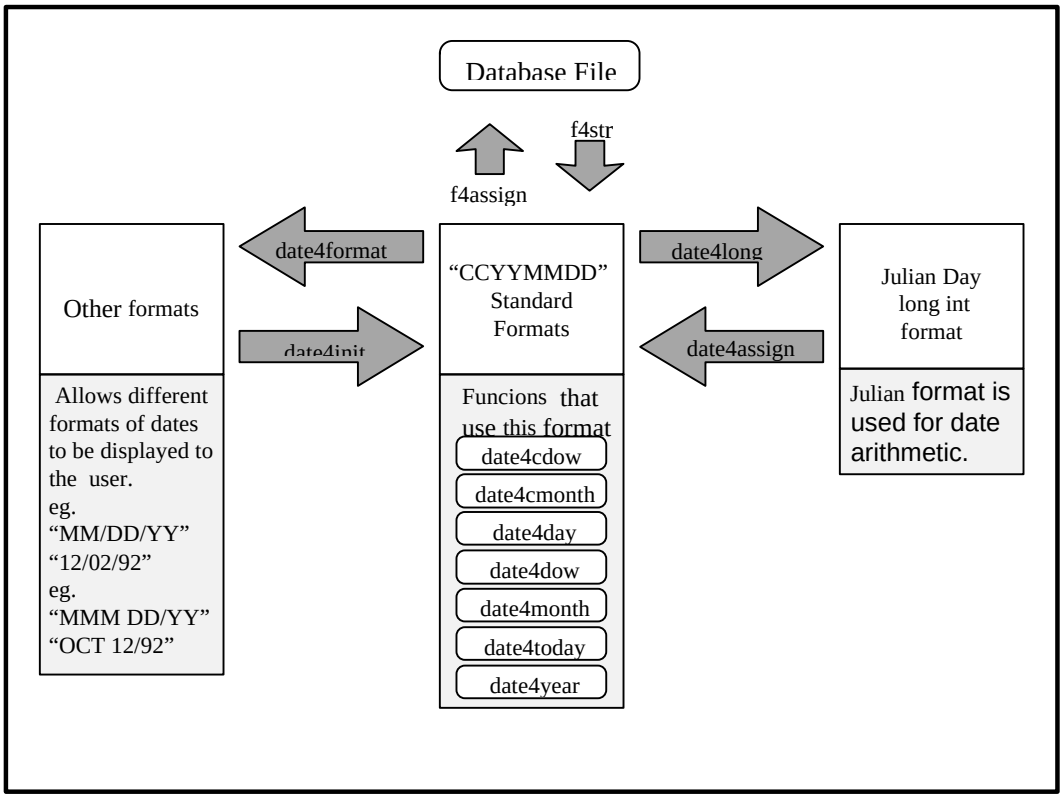


Figure 7.1 Date Conversions

As mentioned above, dates are stored and retrieved from the data file using the functions **f4assign** and **f4str**.

Date strings in standard format can be converted to Julian day format by using the **date4long** function. The inverse of this function is **date4assign**, which converts a Julian date back into a standard date string.

code fragment from DATE1.C

```
julianToday = date4long(todayStandard);
```

To convert from a standard date string to another format, the **date4format** function is used. This function uses a provided date picture to format the date string.

code fragment from DATE1.C

```
date4format(todayStandard, today, "MMM DD/CCYY");
```

To convert from a date string in any other format to standard format, the **date4init** function is used. This time the date picture string is used to specify what format the original string is in. If any part of the date is missing, the missing portion is filled in using the date January 1, 1980.

code fragment from DATE1.C

```
date4init(standard, birthdate, "MMM DD/CCYY");
```

Testing For Invalid Dates

You can test for invalid dates using the **date4long** function. If the date string, in standard format, is not a valid date, the date4long function will return '**(long) -1**'. If it is passed a blank string, the function returns zero. The following example function tests for valid dates:

Code fragment from DATE1.C

```
int  validDate( char *date )
{
    long rc;

    rc = date4long( date);

    if( rc < 1)
        return 0;
    else
        return 1;
}
```

Other Date Functions

There are several other date functions that perform various date related tasks:

Days and Months as Characters

There are two functions that return the day or month portion of the date string as a character strings. They are:

date4cdow The day of the week in character form is returned.

date4cmonth The day of the month in character form is returned.

Code fragment from DATE1.C

```
dow = date4cdow(date);
```

Days, Months and Years as Integers

There are four functions that return a portion of the date string as integer values:

- **date4day** The day of the month is returned as an integer value from 1 to 31.
- **date4dow** The day of the week is returned as an integer value from 1 to 7.
- **date4month** The month of the year is returned as an integer value from 1 to 12.
- **date4year**. Returns the century/year portion of the date as an integer value.

code fragment from DATE1.C

```
howLongUntil(date4month(standard), date4day(standard),
                                          "your next birthday");
```

Miscellaneous Functions

CodeBase also provides the following miscellaneous date functions:

- **date4isLeap** Returns true (non-zero) if the date falls within a leap year and false (zero) otherwise.
- **date4timeNow** Returns a string containing the current system time formatted as "HH:MM:SS".
- **date4today** Initializes a date string with today's date.

8 Linked Lists

The C language has excellent support for arrays. You can create arrays of predefined types such as integers or doubles, or you can create arrays of your own types and structures. Although arrays are highly useful constructs, they have some major limitations: you cannot insert new elements between existing elements, you cannot remove existing elements, and you have to allocate all of the array's memory at once (either at compile time or run time). Although there are ways around the limitations of arrays, these methods are generally inefficient and involve shuffling large amounts of memory.

The Linked List

A more elegant way of providing this functionality is to use linked lists. A linked list is an group of connected elements called *nodes*. Each node contains data and pointers to other nodes.

The list can be constructed in the following manner: the pointer of the first node points to the second node, and the pointer of the second node point to the third node and so on. The pointer of the last node contains null (in this manual, a pointer which contains null is called a null pointer).

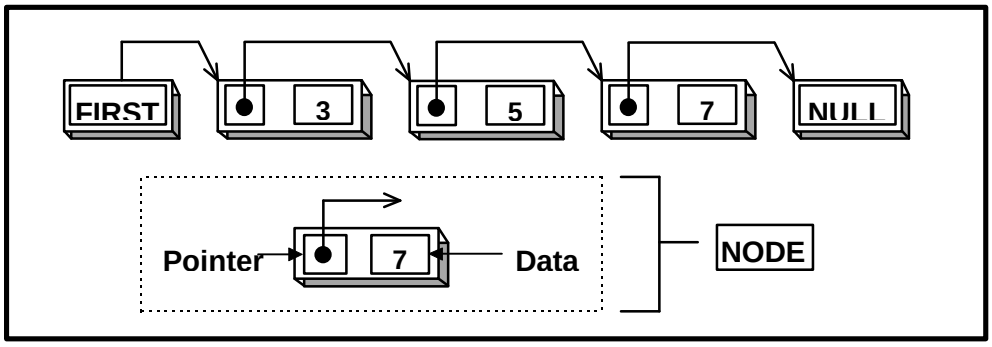


Figure 8.1 A Linked List

The advantages of linked lists over arrays are that the nodes can be located anywhere in memory, new nodes can be added at any time, and nodes can be added and deleted from the list by simply changing the value of a few pointers.

Double Linked Lists

The linked lists used by CodeBase are *double linked lists*. A double linked list is similar to the type of linked list described above except that each node has two pointers instead of one. The first pointer points to the next node, while the second points to the previous node. The following is a standard double linked list.

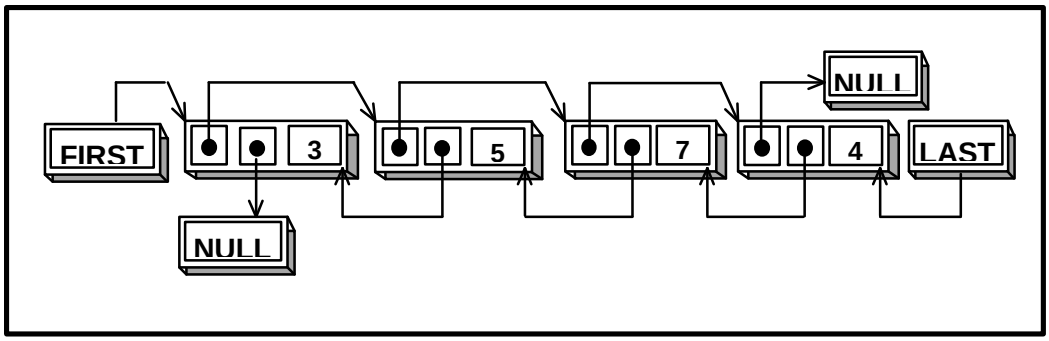


Figure 8.2 Double Linked List

This allows you to traverse the linked list from both directions starting from pointers to the first and last nodes. In addition, it allows you to remove a node without traversing the list.

CodeBase Linked Lists

To implement a linked list using CodeBase, there are two requirements. The first requirment is a **LIST4** structure and the second is a user defined structure for the nodes. Since the linked list functions are independent of the CodeBase data base management, a **CODE4** structure does not need to be created.

PROGRAM LIST1.C

The program "LIST1.C" demonstrates how to traverse, add and remove nodes from a linked list.

```
#include "D4ALL.H"

#ifdef __TURBOC__
    extern unsigned _stklen = 10000;
#endif

typedef struct
{
    LINK4   link;
    int     age;
}AGES;

void printList(LIST4 *);

void main( void )
{
    LIST4   ageList;
    AGES    firstAge,middleAge,lastAge;

    memset(&ageList,0,sizeof(ageList));

    firstAge.age = 3;
    middleAge.age = 5;
    lastAge.age = 7;

    l4add(&ageList,&middleAge);
    l4addBefore(&ageList,&middleAge,&firstAge);
    l4addAfter(&ageList,&middleAge,&lastAge);

    printList(&ageList);

    l4remove(&ageList,(void *) &middleAge);

    printList(&ageList);

    l4pop(&ageList);

    printList(&ageList);
}

void printList(LIST4 *list)
{
    AGES    *agePtr;

    printf("\nThe re are %d links\n",l4numNodes(list));
```

```
    agePtr = (AGES *) l4first(list);
    while(agePtr != NULL)
    {
        printf("%d\n",agePtr->age);
        agePtr = (AGES *) l4next(list,agePtr);
    }

    agePtr = (AGES *) list->selected;
}
```

The LIST4 structure

The **LIST4** structure contains the control information for the linked list. It has a counter containing the number of nodes in the linked list and a pointer to the selected node.

Initializing a **LIST4** structure

Each linked list that your application contains will have its own **LIST4** structure. Before you can use a linked list, the **LIST4** structure must be initialized by setting its contents to zeros. This is accomplished by the following segment of code:

Code Segment From "LIST1.C"

```
LIST4 ageList ;

memset(&age, 0, sizeof(ageList));
```

The Node structure

The node of a CodeBase linked list is a data structure that you define. This may be structure, as long as the first member is a **LINK4** structure.

Example Node Structure

```
typedef struct
{
    LINK4   link;
    int     age;
} AGES ;
```

The **LINK4** Structure

The **LINK4** structure contains pointers to the next and previous nodes in the list. The **LINK4** structure also maintains pointers to the first and last nodes of the list. These pointers are maintained by CodeBase functions so there is no need for you to access them.

Adding Nodes to Linked Lists

There are three functions that add new nodes to CodeBase linked lists. They are **l4add**, **l4addBefore** and **l4addAfter**.

The **l4add** function adds the new node to the end of the list. This function requires two parameters: first, a pointer to the linked list's **LIST4** structure and second, a pointer to the node to be added.

Note

Pointers to node structures are passed to CodeBase linked list functions as void pointers. Any time you see parameters that are of type (void *), you can assume that CodeBase expects a pointer to a node structure.

Here is a code fragment that assigns some values to the example node structures and adds one of them to the linked list.

Code Segment from "LIST1.C"

```
    AGES    firstAge, middleAge, lastAge;
    .
    .
    .
    /* assign values to the nodes */

    firstAge.age = 3;
    middleAge.age = 5;
    lastAge.age = 7;

    /* add a node to the linked list */
    l4add( &ageList, &middleAge ) ;
```

Inserting Nodes

l4addBefore inserts a node into the linked list directly before the given node. **l4addAfter** inserts a node directly after the given node.

In the example program LIST1.C, the new nodes are added before and after a node in the linked list.

Code Segment From "LIST1.C"

```
    l4addBefore( &ageList, &middl eAge, &firstAge) ;
    l4addAfter( &ageList, &middleAge, &lastAge ) ;
```

At this point, the *ageList* linked list would look as follows:

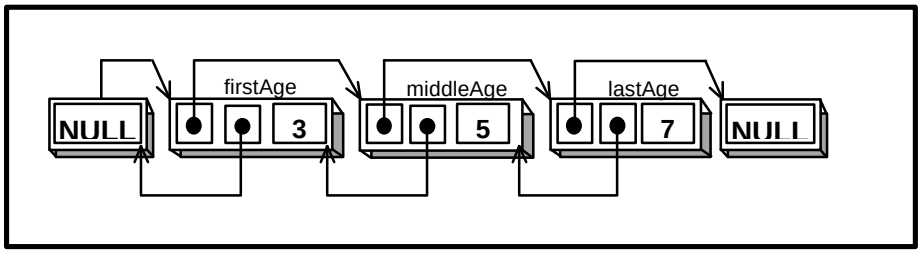


Figure 8.3 Linked List after several additions

Traversing The Linked List

Once items have been added to the linked list, you can then traverse the list from either direction.

Retrieving the First and Last Nodes

The linked list module lets you retrieve the first and last nodes in the list by calling **l4first** and **l4last**, respectively. Both of these functions will return a null pointer if there are no nodes in the linked list.

Obtaining the Next Node

l4next returns a pointer to the next node in the linked list. For example, if you passed **l4next** a pointer to *middleAge*, it would return a pointer to *lastAge*. You can also use **l4next** to obtain a pointer to the first node by passing it a null pointer. You can combine these functions to iterate through the list. This is illustrated by the following code fragment that prints out all the nodes in a linked list:

Code Segment From "LIST1.C"

```
agePtr = (AGES *) l4first(list);
while (agePtr != NULL)
{
    printf("%d\n", agePtr->age);
    agePtr = (AGES *) l4next(list, agePtr);
}
```

The first node is located by **l4first**. To avoid compiler errors, the return value from **l4first** is cast as **(AGES *).**

Note

l4first, **l4last**, **l4next**, **l4prev**, and **l4pop** all return **(void *)** pointers that actually point to node. To avoid compiler errors you must cast the return values of these functions as node pointers.

Next, a **while** loop is entered and the contents of the node are printed out. Before the loop terminates, the next node of the linked list is located by a call to **l4next**. After this call, *agePtr* is either a pointer to the next node, or null if there are no more items in the list.

Traversing The List

If you want to move through the list from the end to the beginning, you would

in Reverse Order

use **l4last** and **l4prev**. **l4prev** behaves the same as **l4next** except it returns a pointer to the node before the specified node.

The Node Count

If you need to know how many nodes there are in a linked list, **l4numNodes** may be used. This returns the number of nodes currently in the list. Since this value is automatically updated each time the list is modified, it is always current.

Removing Items From A Linked List

There are two functions that remove items from a linked list. The first is **l4pop**. This function removes the last node from the linked list. **l4pop** returns a pointer to the node being removed, so that you can free the memory associated with the node. If you want to remove a specific node from anywhere in the linked list, **l4remove** is used.

Dynamic Allocation

The most powerful feature of linked lists is the ability to dynamically allocate nodes (i.e. allocate nodes at run-time). This section will show how you can dynamically add and remove nodes from linked lists.

PROGRAM LIST2.C

The "LIST2.C" is a modification of "LIST1.C". In this program the nodes are dynamically allocated and freed.

```
#include "D4ALL.H"

#ifdef __TURBOC__
    extern unsigned _stklen = 10000;
#endif

typedef struct
{
    LINK4   link;
    int     age;
} AGES;

void printList(LIST4 *);
AGES *addAge(LIST4 *,int);
void removeAge(LIST4 *,AGES *);
void freeAges(LIST4 *);

void main( void )
{
    LIST4   ageList;
    AGES    *agePtr;

    memset(&ageList,0,sizeof(ageList));

    addAge(&ageList,3);
    agePtr = addAge(&ageList,5);
    addAge(&ageList,7);
    addAge(&ageList,6);
    addAge(&ageList,2);

    ageList.selected = (LINK4 *)agePtr;

    printList(&ageList);

    removeAge(&ageList,agePtr);

    printList(&ageList);

    freeAges(&ageList);

    printList(&ageList);
}

void printList(LIST4 *list)
{
    AGES    *agePtr;

    printf("\nThere are %d links\n",l4numNodes(list));
    if (list->selected != NULL )
        printf("The selected node contains the age %d\n",((AGES *)(list->selected))->age);
```

```
    agePtr = (AGES *) l4first(list) ;
    while(agePtr != NULL)
    {
        printf("%d\n",agePtr->age);
        agePtr = (AGES *) l4next(list,agePtr);
    }
}

AGES *addAge(LIST4 *list,int age)
{
    AGES* agePtr;

    agePtr = (AGES *) u4alloc(sizeof(AGES));

    agePtr->age = age;
    l4add(list,agePtr);

    return(agePtr);
}

void removeAge(LIST4 *list,AGES *agePtr)
{
    l4remove(list,agePtr);
    u4free(agePtr);
}

void freeAges(LIST4 *list)
{
    AGES *agePtr;

    while(agePtr = (AGES * ) l4pop(list))
        u4free(agePtr);
}
```

Adding Nodes

The process of dynamically adding a node to a linked list involves two steps. First you must allocate the memory for the node and then add that node to the linked list using a **LIST4** member function.

Allocating Memory

Allocating memory can be performed using the C operator **new**. This operator allocates global memory for the object and calls its constructor. Since this operator is part of the C language, it is extremely portable between all operating systems. Alternatively, the CodeBase utility function **u4alloc** may be used to perform portable memory allocation.

The "LIST2.C" the *addAge* function is uses **u4alloc** to dynamically allocate a node structure and then adds the node to the linked list.

Code Segment From "LIST2.C"

```
AGES *addAge(LIST4 *list,int age)
{
    AGES* agePtr;

    agePtr = (AGES *) u4alloc(sizeof(AGES));

    agePtr->age = age;
    l4add(list,agePtr);

    return(agePtr);
}
```

Removing Nodes

When you dynamically allocate nodes in a linked list, it is important that you deallocate them when you are finished with them. If you do not, the memory is not returned to the system until your application terminates.

Freeing Memory

Deallocation is performed with the C **delete** operator. You provide it a pointer to memory that was previously allocated with the **new** operator, and it is freed and returned to the system. If the memory was allocated with CodeBase function **u4alloc**, **u4free** should be called to free the memory to the system.

Code Segment From "LIST2.C"

In "LIST2.C" the *removeAge* member function removes a node from the linked list and then frees its memory.

```
void removeAge(LIST4 *list,AGES *agePtr)
{
    l4remove(list,agePtr);
    u4free(agePtr);
}
```

Cleaning Up The List

It is always good programming practice to free up any memory that is allocated by your application. When you are finished using your linked list, you should deallocate all of its nodes.This can be accomplished efficiently using the **l4pop** function in a **while** loop.

The *freeAges* function in "LIST2.C" removes and deallocates all of the nodes from the linked list.

Code Segment From "LIST2.C"

```
void freeAges(LIST4 *list)
{
    AGES *agePtr;

    while(agePtr = (AGES *) l4pop(list))
        u4free(agePtr);
}
```

Stacks And Queues

In addition to the standard type of linked list functions, you can also use the linked list functions to implement *stacks* and *queues*.

Stacks

A stack is a list of nodes that can only be added or removed from one end. A deck of playing cards can be an example of a stack. Cards are added by placing them on top of the deck (traditionally called *pushing* a node onto the stack), and removed by taking the top card from the deck (traditionally called *popping* an node from the stack). CodeBase provides two functions, **l4add** and **l4pop,** that provide these behaviours.

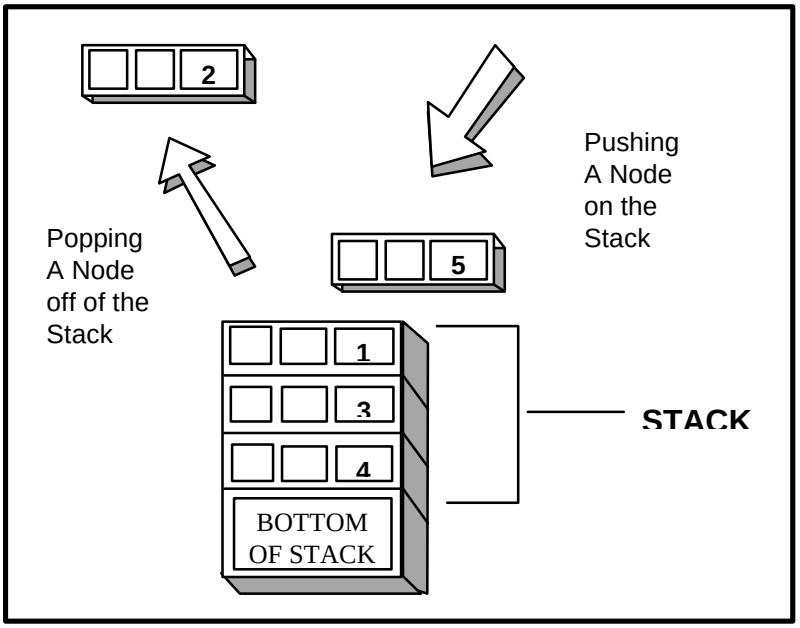


Figure 8.4 Popping and Pushing node on a Stack

Queues

A queue is similar to stack except that nodes are added at one end and removed from the other. A line at the movie box office is an example of a queue. People line up at the back and leave from the front once they have purchased their tickets.

Queues can also be easily implemented using CodeBase linked lists. To remove a node from the queue, **l4pop** is used. To add a node to the list **l4first** and **l4addBefore** are used.

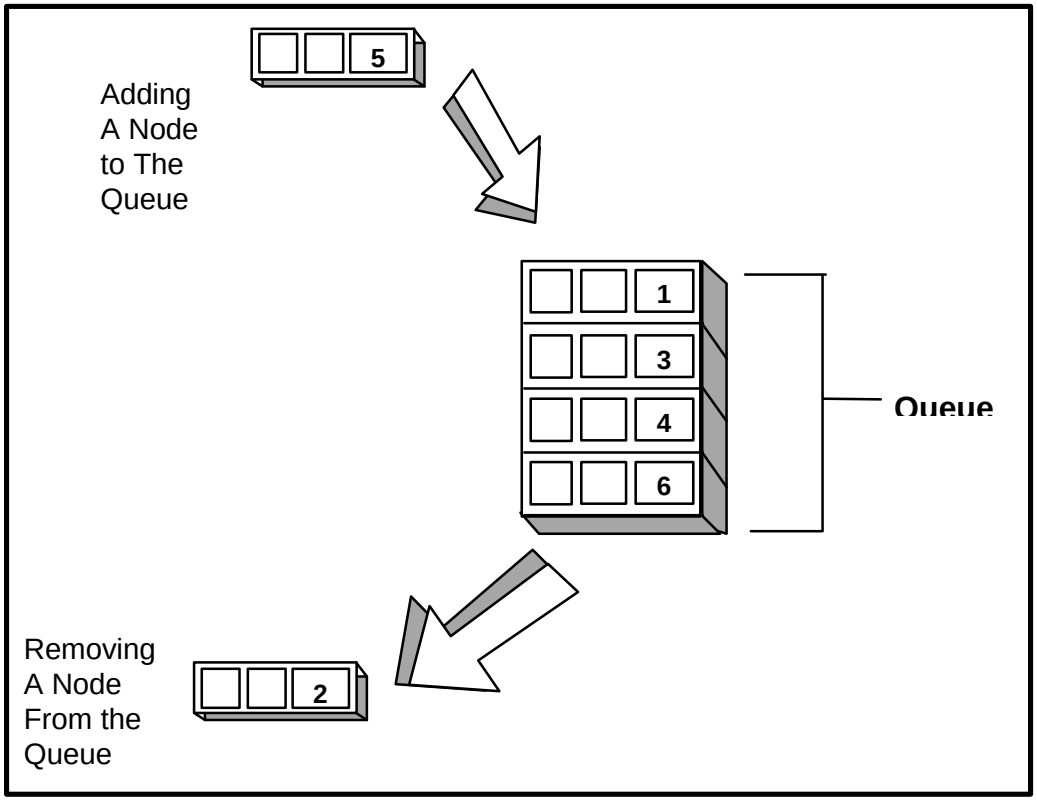


Figure 8.5 Adding and Removing Items from a Queue.

9 Memory Optimizations

Memory optimization is accomplished by buffering portions of data, index and memo files in memory. This improves performance because physically accessing the disk takes longer than accessing memory. In addition, reading and writing several records at once is considerably quicker than reading and writing individual records separately.

Memory optimization is more effective in some operating environments than others. This is because some operating systems already do some memory optimizations at the operating system level and some hard disks do memory optimizations at the hardware level.

Regardless, when memory is available, proper use of CodeBase memory optimizations will always improve performance. This is because CodeBase memory optimizations are designed specifically to optimize database operations. In addition, in network applications, they can reduce requests to the network server.

Using Memory Optimizations

Using memory optimization is quite easy. All you have to do is turn it on. CodeBase has defaults for determining which files should be optimized and whether the files should be optimized for reading and/or writing. If you wish, you can override the defaults for any particular file(s).

Specifying Files To Optimize

Any file that you can open with a CodeBase function can be optimized. This includes data files, index files and memo files, along with files that are opened with **file4open**. Each file opened by CodeBase has two flags associated with it. The first specifies whether the file is optimized. When a file is optimized, the second flag specifies whether the file is optimized for both reading and writing or just reading.

There are two ways of setting a file's optimization flags. You can use the **CODE4** default settings when the file is opened or you can explicitly set them.

Changing The Default Settings

When the file is opened, its optimization flags are set from the **CODE4.optimize** and **CODE4.optimizeWrite** flag settings. Valid settings for **CODE4.optimize** are:

- **OPT4EXCLUSIVE** (Default) Read optimize files when the files are opened exclusively, when the **S4OFF_MULTI** compilation switch is defined or when the DOS read-only attribute is set. Otherwise do not read-optimize the file.
- **OPT4OFF** Do not read optimize the file.
- **OPT4ALL** Same as **OPT4EXCLUSIVE** except that shared files are also optimized.

Valid settings for **CODE4.optimizeWrite** are:

- **OPT4EXCLUSIVE** (Default) Write optimize files when they are opened exclusively or when the **S4OFF_MULTI** compilation switch is defined. Otherwise do not write optimize.
- **OPT4OFF** Do not write optimize.
- **OPT4ALL** Same as **OPT4EXCLUSIVE** except that shared files which are locked are also write optimized.

Note

If read optimization is disabled, then write optimization is also disabled.

The following code segment opens three data files and changes their optimization flag settings. When memory optimization is activated, the first data file is read optimized, the second is read/write optimized, and the third has no optimization at all.

```
CODE4  codeBase;
DATA4  *d1,*d2,*d3;

code4init(&codeBase);

codeBase.optimize = OPT4ALL;
codeBase.optimizeWrite = OPT4OFF;
d1 = d4open(&codeBase,"DATA1");

codeBase.optimizeWrite = OPT4ALL
d2 = d4open(&codeBase,"DATA2");

codeBase.optimize = OPT4OFF
codeBase.optimizeWrite = OPT4OFF;
d3 = d4open(&codeBase,"DATA3");
```

Overriding The Default Settings

A file's optimization flags can be changed at any time by calling one of the following functions: **d4optimize, file4optimize, d4optimizeWrite,** or **file4optimizeWrite**. Both the **file4optimize** and **file4optimizeWrite** functions change the optimization flags of a single file while the **d4optimize** and **d4optimizeWrite** change the optimization flags of a data file and all of its open index and memo files. Following is the above example converted to use the **d4optimize** and **d4optimizeWrite** functions.

```
CODE4  codeBase;
DATA4  *d1,*d2,*d3;

code4init(&codeBase);

d1 = d4open(&codeBase, "DATA1");
d4optimize(d1,OPT4ALL);
d4optimizeWrite(d1,OPT4OFF);

d2 = d4open(&codeBase, "DATA2");
d4optimize(d2,OPT4ALL);
d4optimizeWrite(d2,OPT4ALL);

d3 = d4open(&codeBase, "DATA3");
d4optimize(d3,OPT4OFF);
d4optimizeWrite(d3,OPT4OFF);
```

Activating The Optimizations

When the **code4optStart** function is called, the files whose optimization flags are set are memory optimized. Memory optimization is disabled by **code4optSuspend**.

```
code4optStart( &codeBase ) ;
 . . .

code4optSuspend( &codeBase ) ;
```

Refreshing The Buffers

You can also force a refresh of the optimization buffers. **file4refresh** causes any buffered portion of a file to be discarded. The **d4refresh** performs the same function for a data file and its index and memo files. Finally, the **d4refreshRecord** rereads the current record from the disk.

Memory Requirements

When using memory optimization, you can limit the amount of memory which CodeBase uses for this purpose by setting **CODE4.memStartMax**. The natural question is what is an appropriate maximum? In general, the best maximum is the amount of memory which is likely to be available. For more information please refer to the **CODE4** section of the *Reference Guide*.

Since most applications run under a number of hardware environments where varying amounts of memory are available, the application should assume that all extra available memory was used by CodeBase memory optimization. Therefore, the application should allocate its own memory before calling **code4optStart**, or after calling **code4optSuspend**. If you must allocate memory when the optimization is active, use **u4allocFree**. If this function fails to allocate memory, it will free memory from the CodeBase memory buffers and try again. These techniques make CodeBase memory optimization a lower priority.

If you expect very little memory to be available for CodeBase memory optimization, you should probably just not use it. In this case, you might want to build the CodeBase library using the conditional compilation switch **S4OFF_OPTIMIZE** in order to reduce executable size.

When To Use Memory Optimization

Using memory optimization is not especially difficult. The most difficult part about memory optimization is knowing when to use it.

Single User

The single user case is the most straight forward. Essentially, a single user application can safely memory read and write optimize all files. If an application explicitly flushes to disk by calling CodeBase flushing functions, write optimizations are ineffective. In all other single user cases write optimization is useful for improving performance.

When writing single user applications with memory optimization, it is a good idea to set **CODE4.accessMode** to **OPEN4DENY_RW**. This way CodeBase performs memory optimizations by default.

Multi-User

Whether or not to use memory optimization in a multi-user or multi-tasking environment is a more difficult decision. This is because memory optimization interferes with the multi-user sharing of information. On the other hand, the speed improvements resulting from the use of memory optimization can be even more dramatic because accessing a network server can be slower than accessing a local drive. In addition, memory optimization causes the server to be used less which can improve response time for other users.

Using memory read optimization in a network environment means that previously read database information is buffered in memory. The next time the information needs to be read, it can be quickly fetched from local memory. The disadvantage of this is that if the information has been changed by another application, the most recent piece of disk information may not be retrieved. At worst, one half of a read record could be an up-to-date version and the other half an older version. If read optimization is being used on memo files being updated by another user, it is theoretically possible to be returned an old or garbled memo entry. If the file is locked, using read optimization is completely safe because it cannot be updated by another user. Consequently, it is necessary to be careful to only use read optimization under appropriate circumstances.

Using memory write-optimization on shared files is potentially even more hazardous than using read-optimization. This is because information is written to disk only when the memory buffers are full. Consequently, from the perspective of any user reading the changed information, the information can appear as corrupt. If other applications are using index files which appear corrupt, they can generate errors. It is not quite as bad as it might sound because CodeBase is programmed with extensive error checking and reacts appropriately to most apparent corruption.

WARNING

Allowing one application to write optimize an index file, while another application may use the same index file, can lead to unpredictable results in the application reading the index file.

Note

CodeBase only allows write optimization when the entire data file is locked or when it is opened exclusively. This restriction is necessary in order to guarantee that index and memo files are not corrupt after they have been flushed.

Client-Server

In the client-server configuaration, the **S4CLIENT** switch automatically defines **S4OFF_OPTIMIZE** as the default setting. The **CODE4** members **CODE4.memStartMax**, **CODE4.optimize** and **CODE4.optimizeWrite** are automatically set according to the corresponding fields in the server configuration file. Please refer to the CodeBase server reference manual for more details.

For more examples of memory optimization, refer to the "Optimization" section of the "Multi-user Applications" chapter.

10 Multi-User Applications

A multi-user application can take many forms. For example, several people could be entering data, over a local area network, into the same data file at the same time. Another possibility is one person entering data while several others look at the data file - all using terminals attached to a powerful computer running UNIX. A third case is a single user accessing data in a multi-tasking or multi-threading environment, such as Microsoft Windows, OS/2, or UNIX.

When using CodeBase, you have many options in how you design the multi-user aspects of your application. For example, you can lock data areas before you read them or you can read them without locking. You can choose memory optimizations to improve performance or you can write/read directly to/from disk in order to ensure information is current. The exact options you choose depend on the requirements of the application and hardware resources available.

Locking

At the heart of multi-user applications is locking. Locking is a way in which multi-user database applications communicate with each other. When an application locks some data, it is telling other applications "you cannot modify this data". When data is locked, other applications can still read the data. However, no other application can lock or modify the data.

Operations that require locking

Locking is automatically performed by CodeBase before data is written to disk. It is necessary to lock data before it is written, to keep two applications from updating the same data at the same time. This avoids the corruption due to several applications updating the same index or memo file at the same time.

When a field is to be modified, the record should be locked to prevent multiple users from changing the same record at the same time. This principle is enforced when **CODE4.lockEnforce** is true (non-zero) and the field is modified with a field function or the following data functions: **d4blank**, **d4changed**, **d4delete** or **d4recall**.

Sometimes it is appropriate for an application to lock data before reading it. This is done to keep other applications from updating the data while the lock is present.

Supported Locking Protocols

CodeBase supports a variety of locking protocols that allow applications to be multi-user compatible with applications built using other products. When you build a CodeBase library that uses a specific type of index file compatibility, you also automatically get the multi-user compatibility. The supported locking protocols are as follows:

- **FoxPro** (This is the default locking protocol). This locking protocol is used when the CodeBase library is built with the **S4FOX** conditional compile switch. This provides multi-user compatibility with FoxPro 2.x and FoxPro 3.0. For more information on conditional compile switches please refer to the *Getting Started* booklet.
- **dBASE IV** An application is multi-user compatible with dBASE IV 1.5 when it is built using the **S4MDX** conditional compile switch.
- **Clipper** Clipper 5.2 multi-user compatibility is provided by building the CodeBase library with the **S4CLIPPER** conditional compile switch.

The product versions listed above were the versions which were multi-user compatible with CodeBase when this document was written. CodeBase may be updated to support additional product versions. Check the "README.TXT" file on your CodeBase disk to determine exactly which versions are currently supported.

Types Of Locking

CodeBase performs several types of locking, although the only locks that you need to be concerned with are record and file locks. The types of locks are listed as follows:

- **Record Locking** When a record is locked, that data file record cannot be updated by other applications. This is the lowest level of locking (ie. you cannot lock fields).
- **Data File Locking** When a data file is locked, no records in the data file may be updated by other applications. In addition, a data file lock means that no other application may append records while the data file lock is in place.
- **Index and Memo Locking** CodeBase often locks and unlocks index and memo files when they are updated. However, you do not need to be concerned about this since it is automatic.

Creating Multi-User Applications

Creating well behaved multi-user applications, that is applications that do not lock portions of files for long lengths of time, is relatively easy using CodeBase. If you use the default **CODE4** flag settings, there are only a few things you must consider when writing multi-user applications.

Recommended CODE4 Flag Settings

The following **CODE4** flag settings are often appropriate when you are writing multi-user applications. Unless otherwise specified, the discussions in the following sections assume that these settings are being used. For details on the effects of changing these settings, please refer to last sections of this chapter.

- **CODE4.accessMode = OPEN4DENY_NONE** (default) Files are opened in non-exclusive mode. This means that other applications can share the files with your application and have read and write access.
- **CODE4.readOnly = 0** (default) Opens files in read/write mode. This allows you to read and write records to and from the data file.

- **CODE4.readLock = 0** (default) There is no automatic record locking when a record is read.
- **CODE4.lockAttempts = WAIT4EVER** (default) If a lock fails, CodeBase will keep retrying the lock until it succeeds.
- **CODE4.lockEnforce = 1** An error is generated if an attempt is made to modify an unlocked record with a field function or **d4blank**, **d4changed**, **d4delete** or **d4recall**. This member variable must be set explicitly set to true (non-zero), since the default setting is false (zero).

Automatic Record Locking

Since locking and unlocking are time consuming operations (in the same order of magnitude as a write to disk), CodeBase functions that write a record to disk lock that record without unlocking it afterwards. This prevents redundant unlocking calls and allows you the option of leaving the record locked.

The only functions that modify records and perform automatic locking are: **d4append, d4appendBlank, d4flush, d4flushRecord** and **d4write**.

Normally, you do not want to leave the record locked after these operations. To unlock the record, a call can be made to **d4unlock**:

```
d4append(dataFile) ;
d4unlock(dataFile) ;
```

Automatic Data File Locking

In addition to functions that automatically lock a data file record, there are CodeBase functions that automatically lock the entire data file. These are **d4memoCompress, d4pack, d4reindex, d4zap** and **i4reindex**. It is strongly recommended that not only that the data file be locked, but that the file be opened exclusively before performing these operations. Please refer to the section on **CODE4.accessMode** in the CodeBase *Reference Guide*.

These functions leave the data file locked after they finish executing. As a result, if the data file was opened non-exclusively, these functions should be immediately followed by a call to **d4unlock**.

Automatic Unlocking Of Records

As a rule, CodeBase functions that move from an old record to a new record automatically remove any locks on the old record according to **code4unlockAuto**. These functions are **d4bottom, d4go, d4goEof, d4positionSet, d4seek, d4seekDouble, d4seekN, d4seekNext, d4seekNextDouble, d4seekNextN, d4skip,** and **d4top**. Refer to **code4unlockAuto** in the *Reference Guide* for more information on how the automatic unlocking works.

The only exception to this rule is when the new record is already locked. In that case, no unlocking is performed.

Common Multi-User Tasks

If you follow the advice about calling **d4unlock** after calling functions that automatically lock records and data files and are aware of functions that perform automatic unlocking, you should have little difficulty when writing multi-user applications.

To help you write multi-user applications, examples of common tasks performed by multi-user applications are provided below.

PROGRAM MULTI.C

Application "MULTI.C", which is perhaps simplistic, illustrates some basic operations which are present in almost any multi-user application: adding a record, modifying a record, finding a record, and reporting. It handles its multi-user aspects in a manner which is appropriate for many applications.

"MULTI.C" assumes that a data file "NAMES.DBF" is present along with a production index file containing a tag named "NAME". Data file "NAMES.DBF" also contains a field named "NAME".

```
/*MULTI.C   A multi-user data application. */

#include "d4all.h"

CODE4 cb ;
FIELD4 *fieldName ;
DATA4 *dataNames ;
TAG4 *tagName ;

void addRecord(),findRecord(), modifyRecord(), listData() ;

void main()
{
    code4init( &cb ) ;
    cb.accessMode = OPEN4DENY_NONE ; /* allows access to all applications */
    cb.readOnly = 0 ; /* Set to FALSE */
    cb.readLock = 0 ; /* Set to FALSE */
    cb.lockAttempts = WAIT4EVER ; /* Retry forever. */
    cb.lockEnforce = 1 ; /*must explicitly lock record before modifying*/

    dataNames = d4open( &cb, "NAMES" ) ;
    error4exitTest( &cb ) ;

    fieldName = d4field( dataNames, "NAME" ) ;

    tagName = d4tag( dataNames, "NAME" ) ;

    d4top( dataNames ) ;

    for(;;)
    {
        error4set( &cb, 0 ) ;

        printf( "\n\nRecord #: %ld   Name: %s\n",d4recNo( dataNames ),f4str(fieldName)) ;

        printf( "Enter Command ('a','f' ,'l','m' or 'x')\n" ) ;

        int command =  getchar() ;
    switch( command )
        {
            case 'a':
                addRecord() ;
                break ;

            case 'f':
                findRecord() ;
                break ;

            case 'l':
                listData() ;
                break ;

            case 'm':
                modifyRecord() ;
                break ;
```

```
            case 'x':
                code4close( &cb ) ;
                code4initUndo( &cb )  ;
                exit(0) ;
        }
    }
}

void addRecord()
{
    char buf[100] ;

    printf( "Enter New Record\n" ) ;
    scanf( "%s", buf ) ;

    d4appendStart( dataNames, 0 ) ;
    f4assign( fieldName, buf ) ;
    d4append( dataNames ) ;

    d4unlock( dataNames ) ;
}

void findRecord()
{
    char buf[100] ;

    printf( "Enter Name to Find\n" ) ;
    scanf( "%s", buf ) ;

    d4tagSelect( dataNames, tagName ) ;
    d4seek( dataNames, buf ) ;
}

void modifyRecord()
{
    char buf[100] ;
    int oldLockAttempts, rc ;

    oldLockAttempts = cb.lockAttempts ;
    cb.lockAttempts = 1 /*Only make one lock attempt*/

    rc = d4lock(dataNames, d4recNo(dataNames)) ;
    if(rc == r4locked)
        printf("Record locked. Unable to Edit\n")
    else
    {
        printf("Enter Replacement Record\n") ;
        scanf("%s", buf) ;
        f4assign(fieldName, buf);
        d4flush(dataNames) ;
        d4unlock(dataNames) ;
    }
    cb.lockAttempts = oldLockAttempts ;
}

void listData()
{
    code4optStart( &cb ) ;
    d4optimize( dataNa mes, OPT4ALL ) ;

    d4tagSelect( dataNames, tagName ) ;

    for( d4top(dataNames); ! d4eof(dataNames); d4skip(dataNames,1) )
        printf( "%ld %s\n", d4recNo(dataNames), f4str(fieldName) ) ;

    d4optimize( dataNames, OPT4OFF ) ;
    code4optSuspend( &cb ) ;
}
```

Opening Files

It is especially important to check for any file open errors in multi-user applications because there is a chance that the file might be opened exclusively by another application.

Code fragment from MULTI.C

```
dataNames = d4open( &cb, "NAMES" ) ;
error4exitTest( &cb ) ;
```

Control Loops

Most data file editing software has some kind of loop where the end user can enter various editing and possibly reporting commands. The MULTI.C application is no exception. It allows you to 'add', 'find', 'list', 'modify' or 'exit'.

The start of the loop sets the CodeBase error code to zero. An error can occur if the user tries to modify a record before any have been added.

```
d4top( dataNames ) ;

for(;;)
{
    error4set( &cb, 0 ) ;

    printf( "\n\nRecord #: %ld   Name: %s\n",d4recNo( dataNames ),f4str(fieldName) ) ;

    printf( "Enter Command ('a','f','l','m' or 'x')\n" ) ;

    int command =  getchar() ;
switch( command )
    {
        . . .
    }
}
```

Adding Records

In a single user situation, new records may simply be appended with a call to **d4append**. The multi-user situation is exactly the same, except that after appending the record **d4unlock** should be called (as in the *addRecord* function above). This call is necessary because the newly appended record remains locked after **d4append** completes.

```
void addRecord()
{
    char buf[100] ;

    printf( "Enter New Record\n" ) ;
    scanf( "%s", buf ) ;

    d4appendStart( dataNames, 0 ) ;
    f4assign( fieldName, buf ) ;
    d4append( dataNames ) ;

    d4unlock( dataNames )  ;
}
```

Finding Records

The *findRecord* function is significant, from a multi-user perspective, more for what it does not do rather than what it does. Specifically, there is no multi-user logic necessary in this function. When **CODE4.readLock** is false (zero), the data file functions do not perform any automatic locking as the data file is being read. Accordingly, you can search using index files and read data file records without having any extra multi-user logic present.

Code fragment from MULTI.C

```
void findRecord()
{
    char buf[100] ;

    printf( "Enter Name to Find\n" ) ;
    scanf( "%s", buf ) ;

    d4tagSelect( dataNames, tagName ) ;
    d4seek( dataNames, buf ) ;
}
```

Modifying Records

The field functions are used to assign values to the fields, and when appropriate, the changes are flushed to disk. Explicit flushing can be accomplished by calling **d4flush.**

After flushing a modified record in a multi-user situation, an application should call **d4unlock.** When **d4flush** is called, the changed record gets written to disk. In order to accomplish this, **d4flush** locks the record and leaves it locked. Consequently, the call to **d4unlock** is suggested once the modified record is no longer required, in order to give other users an opportunity to modify the record.

The call to **d4flush** is not strictly necessary. This is because **d4unlock** is smart enough to recognize when the record buffer has changed. In this case, **d4unlock** will lock the record, flush the record, and then unlock the record. However, if the **S4OFF_MULTI** conditional compilation switch is used, it is necessary to call **d4flush** in order to immediately flush the record. This is because **d4unlock** does nothing when compiled with **S4OFF_MULTI** defined.

Code fragment from MULTI.C

```
void modifyRecord()
{
    char buf[100] ;
    int oldLockAttem pts, rc ;

    oldLockAttempts = cb.lockAttempts ;
    cb.lockAttempts = 1 /*Only make one lock attempt*/

    rc = d4lock(dataNames, d4recNo(dataNames)) ;
    if(rc == r4locked)
        printf("Record locked. Unable to Edit\n")
    else
    {
        printf("Enter Replacement Record\n") ;
        scanf("%s", buf) ;

        f4assign(fieldName, buf);

        d4flush(dataNames) ;
        d4unlock(dataNames) ;
    }
    cb.lockAttempts = oldLockAttempts ;
}
```

Locking the Record before modifying

In the MULTI.C example, the record must be explicitly locked before it is modified by a field function, since the **CODE4.lockEnforce** member is set to true (non-zero). This serves to ensure that only one application can edit a record at a time. The example sets the **CODE4.lockAttempts** to **(int) 1**, to cause **d4lock** to immediately return **r4locked** if the record is already locked by another user, in which case the *modifyRecord* function is aborted. Adhering to this locking procedure will prevent the following undesirable scenerio from occurring.

Consider the case where two users of the application decide to modify the same record at the same time. When this happens, one application may write out its changes to disk before the second. The second application then writes to the disk, overwriting the first application's changes. Neither application knows that this has happened. User two makes changes based on an out of date version of the record, and user one loses all changes.

Multi-User Optimizations

This next section deals with using the memory optimization functions in a multi-user application. The optimizations can be used for both appending

large quantities of records and for efficiently generating lists of records.

Listing Records

Function *listData* from the program MULTI.C is an example of reporting. Notice that memory optimization is turned on at the start of the function and turned off at the end of the function. This speeds up the report. In addition, when a network is being used it minimizes requests to the server.

In a multi-user application, the use of memory optimization is recommended when reporting but is strongly discouraged when editing. Refer to the "Memory Optimizations" chapter for additional information.

Notice that there are no unlocking function calls. As explained under "Finding Records" section (above), when a record is read using the data file functions no automatic locking takes place. Consequently there is no need for any unlocking.

Code fragment from MULTI.C

```
void listData()
{
    code4optStart( &cb ) ;
    d4optimize( dataNames, OPT4ALL ) ;

    d4tagSelect( dataNames, tagName ) ;

    for( d4top(dataNames); ! d4eof(dataNames); d4skip(dataNames,1) )
        printf( "%ld %s\n", d4recNo(dataNames), f4str(fieldName) ) ;

    d4optimize( dataNames, OPT4OFF ) ;
    code4optSuspend( &cb ) ;
}
```

Repeated Appending

PROGRAM APPEND.C

The next example multi-user application demonstrates how to append records from one data file to another.

The program APPEND.C demonstrates how to append records from one data file to another. It assumes that data files "TO_DBF.DBF" and "FROM_DBF.DBF" are both present and that they both have a field named "INFO".

```
#include "d4all.h"

CODE4 cb ;
DATA4 *dataFrom, *dataTo ;
FIELD4 *infoFrom, *infoTo ;

void main()
{
    int rc1, rc2, rc ;

    code4init( &cb ) ;

    cb.optimize = OPT4ALL ;
    cb.optimizeWrite = OPT4ALL ;

    dataFrom = d4open( &cb, "FROM_DBF.DBF" ) ;
    dataTo = d4open( &cb, "TO_DBF.DBF" ) ;
    error4exitTest( &cb ) ;

    code4optStart( &cb ) ;

    infoFrom =  d4field( dataFrom, "INFO" ) ;
    infoTo =  d4field( dataTo, "INFO" ) ;

    cb.lockAttempts = 1 ;
    rc1 = d4lockFile( dataFrom ) ;
    rc2 = d4lockFile( dataTo ) ;
    if( rc1 != 0 || rc2 != 0 )
    {
        printf( "Locking Failed\n" ) ;
        exit(0) ;
```

```
    }

    for( rc = d4top(dataFrom); rc == 0; rc = d4skip(dataFrom,1) )
    {
        d4appendStart( dataTo, 0 ) ;
        f4assignField( infoTo, infoFrom ) ;
        d4append( dataTo ) ;
    }

    d4unlock( dataFrom ) ;
    d4unlock( dataTo ) ;

    code4close( &cb ) ;
    code4initUndo( &cb ) ;
}
```

In this application, both memory read optimization and memory write optimization are used. In order to do this, the **CODE4.optimize** and **CODE4.optimizeWrite** defaults are changed before the files are opened. Please refer to the "Memory Optimization" chapter for details.

Code fragment from APPEND.C

```
    cb.optimize = OPT4ALL ;
    cb.optimizeWrite = OPT4ALL ;

    dataFrom = d4open( &cb, "FROM_DBF.DBF" ) ;
    dataTo = d4open( &cb, "TO_DBF.DBF" ) ;
    error4exitTest( &cb ) ;

    code4optStart( &cb ) ;
```

Note that **code4optStart** is called *after* the files are opened. This is because a call to **code4optStart** may use up the available memory. In this case, the calls to **d4open** could be slowed down as optimization was repeatedly suspended and re-invoked inside **d4open**. Internally, **d4open** calls **u4allocFree** when it allocates memory. When memory is not available, **u4allocFree** temporarily suspends memory optimization in an attempt to allocate the requested memory.

This example illustrates the only situation in which write optimization should be considered in a multi-user application. When writing to a file with write optimizations enabled, the file should be locked.

With the file locked and write optimization enabled, the repeated appending goes considerably faster. The trade off is that if any other application attempts to read the records being appended, the other application could, at worst, generate CodeBase errors or could read garbage information. To avoid this, you can open the files exclusively or not use write optimization.

In this specific case, the read optimization is a very good idea. This is because there is no chance that information being returned could be out of date because the file is locked. As a result of the read optimization, the speed improvements in this example are considerable.

This application skips sequentially through data file "FROM_DBF" using memory optimization. Consequently, it is important to use function **d4skip** rather than **d4go** because function **d4skip** detects the sequential reading and does special performance optimizations.

Since the data files are locked going into the loop, they stay locked throughout the entire loop with no automatic unlocking occurring.

Code fragment from APPEND.C

```
    for(rc = d4top(dataFrom); rc == 0; rc = d4skip(dataFrom,1))
    {
        d4appendStart( dataTo, 0 ) ;
        f4assignField( infoTo, infoFrom ) ;
        d4append( dataTo ) ;
    }
```

Once the records have been appended, the files are unlocked and closed. When data file "TO_DBF.DBF" is unlocked its changes are automatically flushed to disk.

Code fragment from APPEND.C

```
    d4unlock( dataFrom ) ;
    d4unlock( dataTo ) ;

    code4close( &cb ) ;
}
```

The calls to **d4unlock** are not strictly necessary in this example because **code4close** does flushing and unlocking automatically. Alternatively, **code4unlock** could have been called to unlock both files at once.

Avoiding Deadlock

When locking multiple data files at once, you need to be careful to avoid deadlock. Deadlock happens when one application waits for a second application to unlock something and the second application waits for the first application to unlock something else. Since they are both waiting for each other, they both wait forever. This program avoids deadlock by changing **CODE4.lockAttempts** from the default of unlimited retries to one try. If the lock attempt fails, the program exits.

Code fragment from APPEND.C

```
    cb.lockAttempts = 1 ;
    rc1 = d4lockFile( dataFrom ) ;
    rc2 = d4lockFile( dataTo ) ;
    if( rc1 != 0 || rc2 != 0 )
    {
        printf( "Locking Failed\n" ) ;
        exit(0) ;
    }
```

Another way to avoid deadlock is to be careful to always lock data files in the same order. If one application locks data file "FROM_DBF" before "TO_DBF" using unlimited retries, then all applications using unlimited retries should do the same.

If applications always lock data files in the same order, deadlock is not possible. This is because once the first application succeeds in locking the first data file, other applications will wait for the first application to finish its locking, do what it needs to do and then unlock all of the data files. The best way to ensure you are always locking data files in the same order is to lock and unlock all data files together.

Group Locks

Sometimes it is desirable to lock many items as a group. In this case, either all items are locked or no items are locked. This functionality will help minimize the chance of deadlock. CodeBase supplies functions that will put an item on a queue so that the whole queue can be locked as a group. The functions **d4lockAdd**, **d4lockAddAppend**, **d4lockAddFile**, **d4lockAddAll** and **relate4lockAdd** put their respective items on a queue to be locked. These functions do NOT lock any items. The items are locked by a call to **code4lock**. If any item in the queue fails to be locked, the successful locks are removed, **code4lock** returns **r4locked** and the queue remains intact for a future lock attempt by a subsequent call to **code4lock**. When **code4lock** succeeds, all the locks are in place and the queue is emptied. Refer to **code4lock** in the *Reference Guide* for more details.

Exclusive Access

The simplest type of multi-user applications are those that open all of their files in exclusive access mode. If any other application tries to open a file which has been opened exclusively by another user, the application gets a file open error.

The big disadvantage of this method is that no other applications can use files that you are using. This is not a recommended method of creating multi-user applications.

The main use of opening files exclusively is when you are performing packing, zapping, indexing or reindexing. This prevents other applications from generating errors or garbage output if they use the file while one of these activities is occurring.

Opening Files Exclusively

Whether a file is opened exclusively is determined by the current status of the **CODE4.accessMode** flag. This flag specifies what access OTHER users have to the current file. **CODE4.accessMode** has three possible values:

- **OPEN4DENY_NONE** Open the database files in shared mode. Other users have read and write access. This is the default value.
- **OPEN4DENY_WRITE** Open the database files in shared mode. The application may read and write to the files, but other users may only open the file in read only mode and may not change the files.
- **OPEN4DENY_RW** Open the database files exclusively. Other users may not open the files.

This flag can be changed any time after the **CODE4** structure has been initialized with **code4init**. Changes to the flag only affect those files that are opened thereafter. Therefore if the access mode must be altered for an open file, the file must be closed and then reopened.

```
cb.accessMode = OPEN4DENY_RW;
dataFile = d4open( &cb, "DATAFILE" ) ; /* open exclusively*/
```

If a file cannot be opened exclusively due to the fact that some other application already has it open, the file open fails. When this happens, and the **CODE4.errOpen** flag is set to true , an error message is generated.

Read Only Mode

If your application reads information from a file and never modifies it, you can open the file in read only mode. This mode must be used when you only have read access to a file. This is a common situation on a network where a database is shared among many users, but only a few users have the authority to make changes to it.

Opening Files In Read Only Mode

Whether a file is opened in read only mode is determined by the current status of the **CODE4.readOnly** flag. If the flag is set to its default value of false (zero), files are opened in read / write mode. If this flag is set to true (non zero), files are opened in read only mode.

Code fragment from MULTI.C

```
cb.readOnly = 0; /* set to FALSE */
```

Lock Attempts

When a lock is performed on a file, the possibility exists that some other application has already locked that file or a portion of the file. When this occurs, what CodeBase does next is determined by the status of the **CODE4.lockAttempts** flag.

The default value of **CODE4.lockAttempts** is **WAIT4EVER**. This indicates that CodeBase will keep trying to establish a lock until it succeeds. The advantage of this method is that it simplifies programming because the application can assume that all lock attempts succeed. Unfortunately, this can increase the chances of deadlock bugs being present in an application. In addition, if applications attempt locking for long periods of time, users can be left waiting. One potential solution is to use the **S4LOCK_HOOK** conditional compilation switch and put the number of retries directly under the control of the end user. Refer to the CodeBase *Reference Guide* "Conditional Compilation Switches" chapter for more information on this technique.

Code fragment from MULTI.C

```
cb.lockAttempts = WAIT4EVER; // Retry forever
```

If **CODE4.lockAttempts** is set to any value greater than one, CodeBase keeps trying the lock at approximately one second intervals until either a lock is established or the number of attempts equals **CODE4.lockAttempts**. If the lock fails, a value of **r4locked** is returned from the function attempting the lock.

If you wish to perform a special operation if the lock fails, such as displaying a message to the user, you should set **CODE4.lockAttempts** to one and test for return values. Valid settings for CODE4.lockAttempts are WAIT4EVER (-1) or any value greater than or equal to 1. Any other value is undefined.

Automatic Record Locking

When **CODE4.readLock** is true, several CodeBase functions automatically lock records before reading them. This ensures that no other application can modify a record that your application has in its record buffer.

Unfortunately, locking data file records when they are only being read can create unnecessary locking contention. Simply put, this would increase the chance of a record being locked when an end user just wants to look at it. Consequently, the default of no automatic read locking is appropriate for most multi-user applications.

The following list of functions perform record locking when the **CODE4.readLock** flag is set to true: **d4bottom, d4go, d4position, d4seek, d4seekDouble, d4seekN, d4seekNext, d4seekNextDouble, d4seekNextN, d4skip,** and **d4top**.

Enforced Locking

Locking a record before it is modified is very important in the multi-user configuration, since it ensures that only one application can edit a record at a time. The **CODE4** member variable **CODE4.lockEnforce** can be used to ensure that an application has explicitly locked a record before it is modified with a field function or the following data file functions: **d4blank**, **d4changed**, **d4delete** or **d4recall**. When **CODE4.lockEnforce** is set to true (non-zero), an **e4lock** error is generated when a attempt is made to modify an unlocked record.

An alternative method of ensuring that only one appliction can modify a record at a time is to deny all other applications write access to a data file. Write access can be denied to other applications by setting **CODE4.accessMode** to **OPEN4DENY_WRITE** or **OPEN4DENY_RW** before opening the file. Thus, it is unnecessary to explicitly lock the record before editing it, even when **CODE4.lockEnforce** is true (non-zero), since no other application can write to the data file. This method suffers from the same disadvantage as discussed in "Exclusive Access" section of this chapter. The restricted access to the files prevents the creation of practical multi-user applications.

11 Performance Tips

Programmers are always looking for ways to improve the performance of their applications. This chapter provides the programmer with tips on how to achieve the highest performance when building applications with CodeBase.

Memory Optimization

The prudent use memory optimization can enhance the performance of an application. For details on when to use memory optimization refer the "Memory Optimizations" chapter of this guide. Write optimization is most useful when making multiple updates on files, such as appending many records at once. In general, when memory optimization is used, **d4skip** is faster than **d4go**.

Memory Requirements

The **CODE4** member variable **CODE4.memStartMax** specifies the maximum amount of memory allocated for memory optimization. Theoretically, the more memory that CodeBase can use for memory optimization the greater the improvement in performance. In practise, if **CODE4.memStartMax** is too large, CodeBase may not be able to allocate all of the requested memory causing the memory optimization to work less efficiently. In some operating systems all requested memory is allocated; but as virtual memory, which is counter-productive to memory optimization. On the other hand, if **CODE4.memStartMax** too small, the potential benefits of memory optimization will not warrant the overhead.

Observing the performance improvement

The improvement in performance can be observed by using **code4optAll**. **code4optAll** ensures that memory optimization is fully implemented by locking and fully read and write optimizing all opened data, index and memo files. A comparison of the performance can then be made between the application that employs memory optimization with one that does not.

Functions that can reduce performance

Memory optimization acts by buffering portions of files in memory and thus decreases the number of disk accesses and improves performance. Therefore, frequent calls to CodeBase functions that read data from disk will negate the effects of read optimization. For this reason, functions such as **d4refresh**, **d4refreshRecord** and **file4refresh** should not be called repeatedly while memory optimization is invoked. Frequent flushing also reduces the benefits of write optimization.

Locking

Locking can take as long a disk access, so repeated record locks are inefficient. It is more efficient to lock an entire file when multiple updates are necessary. The best way to lock a file that requires many modifications is to use **d4lockAddAll** with **code4lock,** or call **d4lockAll.**

A **CODE4** setting that can influence performance is **CODE4.readLock**. When **CODE4.readLock** is set to true (non-zero), each record is locked before it is read, which can reduce performance. It is useful to set **CODE4.readLock** to true when the records are to be modified.

Another important **CODE4** variable that can affect performance in a network environment is **CODE4.lockDelay**, which determines how long to wait between lock attempts. Consider this variable carefully because if the value is too small, it can cause an increase in network traffic and possibly a decrease network performance.

Appending

Whenever possible, it is more efficient to append many records at once rather than one at a time. When appending many records, it is sometimes faster to close the index files first, append the records and then reindex. The files must be locked before the records may be appended and the most efficient method is to call **d4lockAddAll** with **code4lock,** or **d4lockAll,** as discussed above. Batch appending can also take advantage of write optimization, which will improve performance.

Time Consuming Functions

Certain CodeBase functions are inherently time consuming and this aspect should be considered when incorporating them into an application. Avoid calling time consuming functions repeatedly, since this can reduce performance. Some functions are meant to be used for debugging purposes and should not be used in the final user application.

The following functions reindex index files, which is a time consuming procedure, so they should be used with care. **d4reindex** and **i4reindex** can be called to explicitly reindex files. Both **d4pack** and **d4zap** automatically reindex the associated index files when they are called.

d4pack and **d4zap** are time consuming functions even when there are no tags to reindex. These functions physically remove records from a data file and then reorganize the remaining records, which is a time consuming process.

d4check is a function that determines whether an index file has been corrupted, which is useful for debugging. This function can take as long as reindexing and therefore application performance can be hindered.

Queries and Relations

CodeBase uses Query Optimization to greatly increase the performance of queries in relation sets. To ensure that the Query Optimization can be used by CodeBase, the query expression must have a corresponding tag expression. See the "Query Optimization" chapter in this guide for more details on how to ensure that the query optimization will be invoked.

Another important issue to consider when manipulating relations is how to specify the sort order of the query set. The most efficient manner will depend on the size of the database. There are three ways in which a sorted order may be specified for the query set.

1. The sorted order can be determined by the selected tag in the master data file.
2. If there is no selected tag for the master data file, then the natural order of query set is used.
3. The function **relate4sortSet** can be used to specify the sorted order.

If there is a tag that specifies the same sort order as the **relate4sortSet** expression, then use the tag sorted order when the query set is almost the same size as the database. If the query set is almost the same size as the database, then **relate4sortSet** is inefficient when compared with the tag sorted order or the natural order. Use **relate4sortSet** when the query set is small compared to the size of the database. This will result in better performance when compared with the selected tag ordering.

If there is no tag that specifies a desired sort order, then use **relate4sortSet** to sort the query set. Using **relate4sortSet** will be faster than creating a new tag to specify the sort order, regardless of the size of the query set.

General Tips

The following section discusses miscellaneous issues that may influence the performance of an application.

- Be sure to use the field functions to manipulate fields for the best results. Avoid using the expression module to perform calculations on fields, otherwise the application will execute at a slower rate.
- It is recommended that all necessary tags be constructed when the index file is created with either **d4create** or **i4create**. This method is more efficient than adding new tags by calling **i4tagAdd** later in the application.
- The memory functions specialize in the repeated allocation and freeing of fixed length memory. As a result, it is better than some operating systems at memory management. Therefore, it is advantageous to use this module whenever possible.
- File access is quicker when the file is opened exclusively, since no future record or file locks will be necessary.
- The **CODE4** member variables that determine how many memory blocks are allocated should be defined at the beginning of the program and should not be changed during the program. This allows CodeBase to share memory pools efficiently. Set the following **CODE4** settings at the beginning of the application.

  CODE4.memStartData
  CODE4.memStartIndex
  CODE4.memStartBlock
  CODE4.memStartLock
  CODE4.memStartMax
  CODE4.memStartTag

12 Transaction Processing

At times it is necessary to have a group of actions executed as a unit. In this case, either all actions are completed or none of the actions are completed.

To illustrate this, consider the everyday example of transferring funds from one bank account to another. There are two steps to this process: first one account has the money debited and then the second account has the sum credited. A failure could occur after the debit but before the credit, in which case the money would be removed from the first account but the second account remains unchanged, resulting in lost funds. In this type of situation, one would like an all or nothing response, where either both the debit and credit are completed or nothing happens and the accounts remain unchanged.

Transactions

A transaction is one way of treating a group of actions as a whole. The scope of a transaction is delimited by a specified "start" and "end" between which the actions are treated as a unit. A transaction has the all or nothing property, so that either all the actions within the scope of a transaction are completed or none are completed. When a transaction has successfully completed all the operations within its scope, it is "committed". A transaction can be aborted and all the changes that were made up until the failure can be reversed. This constitutes a "rollback". After the transaction has terminated, the changes are permanent and can not be reversed.

PROGRAM TRANSFER.C

The following program shows how a simple bank transfer can be accomplished using a CodeBase transaction.

```
/*TRANSFER.C*/

#include "d4all.hpp"

extern unsigned _stklen = 10000 ; /*for all Borland Compilers*/

CODE4 codeBase ;
DATA4 *datafile ;

FIELD4 *acctNo, *balance ;
TAG4 *acctTag, *balTag ;

void OpenLogFile( void )          /*Opening or creating a log file is  */
                         /*only required for STAND-ALONE mode.*/
{
    int rc ;

                                        /*NULL is passed as the log file name */
                                        /*so the default "C4.LOG" is used as  */
                                        /*the log file name.                  */
    rc = code4logOpen( &codeBase, 0, "user1") ;
    if ( rc == r4noOp en )
    {
        codeBase.errorCode = 0 ;
        rc = code4logCreate( &codeBase, 0, "user1" ) ;
    }
}

void OpenDataFile( void )
{
    datafile = d4open( &codeBase, "BANK.DBF" ) ;

    acctNo = d4field( datafile, "ACCT_NO" ) ;
    balance = d4field( datafile, "BALANCE" ) ;
```

```
    acctTag = d4tag( datafile, "ACCT_TAG" ) ;
    balTag = d4tag( datafile, "BAL_TAG" ) ;
}

int Credit( long toAcct, double amt )
{
    int rc ;
    double newBal ;

    rc = d4tagSelect( datafile, acctTag ) ;
    rc = d4seekDouble( datafile, toAcct ) ;

    if (rc != r4success ) return rc;

    newBal = f4double( balance ) + amt ;
    d4lockFile( datafile ) ;
    f4assignDouble( balance, newBal ) ;

    return r4success;
}

int Debit( long fromAcct, double amt )
{
    return Credit( fromAcct, -amt ) ;
}

void Transfer( long fromAcct, long toAcct, double amt )
{
    int rc, rc1, rc2 ;

    code4tranStart( &codeBase ) ;

    rc1 = Debit( fromAcct, amt ) ;
    rc2 = Credit( toAcct, amt );

    if (rc1 == r4success && rc2 == r4success)
        rc = code4tranCommit( &codeBase ) ;
    else
        rc = code4tranRollback( &codeBase ) ;
}

void PrintRecords( void )
{
    int rc ;

    for( rc = d4top( datafile ); rc == r4success; rc = d4skip( datafile, 1L ))
    {
        printf("-----------------------------------\n") ;
        printf("Account Number: %ld\n", f4long( acctNo )) ;
        printf("Balance       : %f\n", f4double( balance )) ;
    }
    printf("=========================================\n") ;
}

void main( void )
{
    code4init( &codeBase ) ;

    codeBase.errOpen = 0 ;
    codeBase.safety = 0 ;
    codeBase.lockAttempts = 5 ;
    codeBase.lockEnforce = 1 ;

    OpenLogFile() ;

    OpenDataFile() ;

    PrintRecords() ;

    /* The account number 56789 doesn't exist in the database,*/
    /* the transfer is aborted and database is not changed    */

    Transfer( 12345, 56789, 200.00 ) ;
    PrintRecords() ;

    /* Both accounts exist so the transfer is completed       */
    /* and the database is updated                            */

    Transfer( 12345, 55555, 150.50 ) ;
    PrintRecords() ;
```

```
    code4initUndo( &codeBase ) ;
}
```

In the TRANSFER.C program, a transfer takes place when there is a debit from one account followed by a credit to another, as shown by the following function calls.

Code fragment for TRANSFER.C

```
rc1 = Debit( fromAcct, amt ) ;
rc2 = Credit( toAcct, amt );
```

These two operations must both be successful in order for the transfer to succeed. Therefore, these two actions must be contained within a transaction where either both the debit and the credit succeed or neither will be completed and the database will not be changed. In CodeBase, a transaction is delimited by the functions **code4tranStart** and **code4tranCommit**, which initiate and terminate the transaction respectively.

```
code4tranStart( &codeBase ) ;

...   /*code to be treated as a unit*/

code4tranCommit( &codeBase) ;
```

Events may occur that cause failure during a transaction. For example, if there is a power failure or the hardware crashes during a transaction, one would want to be able to reverse any changes that had occurred before the failure. Programmers must anticipate and test for possible failure points within a transaction. For instance, the example program TRANSFER.C checks to see if both specified accounts exist in the database and a failure occurs if either account did not exist. When the program encounters a failure of this sort, it calls **code4tranRollback** to abort the transaction and restore the database to its original state.

If the transfer succeeds, the transaction is completed by **code4tranCommit**, which commits the changes to the database. After the transaction has been committed, **code4tranRollback** can NOT be used restore the database to the state that existed before the transaction was started. An error is generated if **code4tranRollback** is called after **code4tranCommit**.

Code fragment from TRANSFER.C

```
if (rc1 == r4success && rc2 == r4success)
        code4tranCommit( &codeBase) ;
    else
        code4tranRollback( &codeBase) ;
```

How a Transaction works

CodeBase has implemented transactions in the following manner. When a transaction is initiated with **code4tranStart**, all the changes to the database are recorded in a log file. The log file stores both the old value and the new value, as well as who made the change. At every step during a transaction, each change is recorded in the log file and then the database is modified. After the transaction has been committed using **code4tranCommit,** the changes to the database are permanent. During a rollback, **code4tranRollback** uses the original values from the log file to reverse all the changes made to the database.

Logging in the Stand-Alone Case

All modifications that are made while an application is running, which are outside the scope of a transaction, can be automatically recorded in the log file. Automatic logging is useful when a back up copy of all changes is required. There are times when logging is not necessary; for example, when copying data files. In the stand-alone configuration, the logging can be turned off to save time and disk space. The following discussion applies only to logging in the stand-alone configuration.

CODE4.log

The **CODE4** member variable **CODE4.log** is used to specify whether changes to the data files are automatically recorded in a log file. Changing this setting only affects the logging status of the files that are opened subsequently. The files that were opened before the setting was changed retain the logging status that was set when the file was opened. **CODE4.log** has three possible values:

- **LOG4ALWAYS** The automatic logging takes place for all the data files for as long as the application is running. The logging can NOT be turned off for the current data file by calling **d4log**.
- **LOG4ON** The automatic logging takes place for all the data files for as long as the application is running. In this case, **d4log** can be used to turn the logging off and on for the current data file. CodeBase uses this value as the default.
- **LOG4TRANS** Only the changes made during a transaction are automatically recorded in the log file. Logging may be turned on and off for the current data file by calling **d4log**.

d4log

When a file is opened with **CODE4.log** set to either **LOG4ON** or **LOG4TRANS, d4log** can be used to turn the logging off and on for the current data file. **d4log** only has an effect on logging while the data file is open and it has no effect on the logging that is done during a transaction.

Pass false (zero) to **d4log,** to turn the logging off and pass it true (non-zero) to turn the logging back on. **d4log** also returns the previous logging setting. If -1 is passed to **d4log** the current logging status is returned. The possible return values are as follows:

- **r4logOn** This return value means that logging is turned on for the data file.
- **r4logOff** This return value means that either the logging is turned off

for the data file or that logging in general is not enabled.

- **<0** An error has occurred.

turning logging off

```
                /*Logging of changes takes place        */
                /*for all data files opened subsequently*/

codeBase.log = LOG4ON ;

datafile = d4open( &codeBase, "DATA.DBF" ) ;

rc = d4log( datafile, 0 ) ;
                          /*logging is turned off for this */
                          /*particular data file and the previous*/
                          /*logging status is returned  */
rc = d4log( datafile, -1 ) ;
                          /*the current logging status is returned*/
```

Log files

In the stand-alone configuration, a log file must exist or be explicitly created before any automatic logging or transaction logging can take place. If a log file does not exist, **code4tranStart** will generate an error.

The log file may be manually opened or created by using **code4logOpen** and **code4logCreate** respectively. **code4logOpen** and **code4logCreate** both take the **CODE4** pointer, the name of the log file and user identification as parameters. If null is passed as the file name, then the default file name "C4.LOG" is used. **code4logOpen** and **code4logCreate** must be called before **d4open** or **d4create**.

Generally, one would like to open a log file if it exists, otherwise create a new log file. This concept is the same as discussed in the "Opening or Creating" section of the "Database Access" chapter in this guide.

Code fragment from TRANSFER.C

```
void OpenLogFile( void )
{
    int rc ;

    rc = code4logOpen( &codeBase, 0, "user1") ;
    if ( rc == r4noOpen )
    {
        codeBase.errorCode = 0 ;
        rc = code4logCreate( &codeBase, 0, "user1" ) ;
    }
}
```

If **code4logOpen** or **code4logCreate** have not been explicitly called, **d4open, d4create** and **code4tranStart** automatically try to open a log file by calling **code4logOpen**. **code4logOpenOff** can be used to instruct **d4open**, **d4create** and **code4tranStart** not to automatically open the log file. When **code4logOpenOff** is used, no transactions can start unless the log file is opened explicitly by the application.

the log file is not automatically opened

```
rc = code4logOpenOff( &codeBase ) ;
datafile = d4open( codeBase, "DATA.DBF" ) ;
```

Logging in the Client-Server case

In the client-server configuration, the automatic logging and transaction logging are handled by the server. Calling the functions **code4logCreate** **code4logOpen** and **code4logOpenOff** in a client application will have no effect, since the log file is controlled by the server. These functions will always return **r4success** when called by a client application.

Locking

Every time a data file is being modified one must consider which locking procedure is appropriate. Locking is necessary step in preserving the integrity of the database. CodeBase has both automatic and explicit locking features, as discussed in the "Multi-user Applications" chapter in this guide.

Automatic locking during a transaction

While a transaction is in progress, any automatic locking that is required during a transaction is performed, but CodeBase acts as though the **code4unlockAuto** is set to **LOCK4OFF,** so no automatic unlocking will occur.

Automatic locking and unlocking can occur when a transaction is committed or rolled back. In both cases, record buffer flushing may be required. The locking and unlocking procedure follows that of **d4flush**. If a new lock is needed, everything is unlocked according to **code4unlockAuto** and then the lock is placed. Otherwise, nothing is unlocked, when no new locks are required. It may be necessary to explicitly unlock files after the transaction is committed or rolled back depending on the **code4unlockAuto** setting.

Automatic locking vs. Explicit locking

Whether using the automatic or explicit locking one must decide on a value for the **CODE4** member variable **CODE4.lockAttempts**. This member has a default value of **WAIT4EVER**, which may result in deadlock in a multi-user environment. To prevent deadlock, set **CODE4.lockAttempts** to a reasonable finite number.

The example program TRANSFER.C sets **CODE4.lockAttempts** to a finite value and takes advantage of the automatic locking properties of CodeBase. When **d4seek** is called during the transaction, the current record may need to be flushed before the new record can be loaded into the recorded buffer. Normally, if a new lock is required, everything is unlocked according to **code4unlockAuto** and then the lock is placed, but the seek takes place during a transaction so no unlocking is performed.

Code fragment from TRANSFER.C

```
codeBase.lockAttempts = 5 ;
```

When an application is running in a multi-user environment, it is strongly recommended that records be locked before they are modified to prevent more than one user access to the record at the same time. In this case, it is prudent to set **CODE4.lockEnforce** to true (non-zero) to ensure that a record is explicitly locked before it is modified. If it is known before hand that many changes will be made, use **d4lockAll,** or use **d4lockAddAll** followed by **code4lock** to explicitly lock the files.

If the file is explicitly locked, it must be explicitly unlocked by calling **code4unlock** or **d4unlock** after the transaction is committed or rolled back. If an attempt is made to unlock files during a transaction, an error is generated.

Whether locking is done explicitly or implicitly, a lock may not succeed because another application has that particular file or record locked, in which case **r4locked** is returned. To circumvent this, build the CodeBase library with the conditional compilation switch **S4LOCK_HOOK** defined. This enables CodeBase to automatically execute the function **code4lockHook** when a lock fails. **code4lockHook** is a function that is defined by the application programmer, so that it can allow the end user to retry the lock. Refer to the "Conditional Compilation Switches" chapter of the CodeBase *Reference Guide* for more information on **S4LOCK_HOOK** and **code4lockHook**.

Index

—.—

.CDX. *See* Index Files
.CGP. *See* Group Files
.DBF. *See* Data Files
.MDX. *See* Index Files
.NTX. *See* Index Files

—A—

Appending Records. *See* Records
Automatic Opening
  index files. *See* Index Files, production indexes

—B—

Buffering. *See* Memory Optimizations

—C—

CCYYMMDD. *See* Date Operations
Character Fields
  assigning values to, 31
  creating, 28
  retrieving contents, 25
Clipper
  index files, 39
  locking protocol, 109
CODE4, 13, 17, 21
  initializing, 21
  multiple instances of, 37
  multi-user settings, 110
  uninitializing, 37
CODE4 Members
  accessMode, 31, 106, 110, 111, 119, 121
  autoOpen, 45, 58
  errDefaultUnique, 51, 85
  errOpen, 29, 120
  lockAttempts, 111, 115, 118, 120, 132
  lockDelay, 124
  lockEnforce, 31, 109, 111, 115, 121, 133
  log, 130
  memStartMax, 105, 107, 123, 125
  optimize, 104, 107, 117
  optimizeWrite, 104, 107, 117
  readLock, 110, 111, 114, 121, 123
  readOnly, 110, 111, 120
  safety, 29
code4lockHook, 133
CodeBase
  constants, 17
  functions, 17
  structures, 17, 18
CodeBase Functions
  code4close, 22, 118
  code4init, 21, 29, 37
  code4initUndo, 37
  code4lock, 119, 123, 124, 133
  code4logCreate, 131, 132
  code4logOpen, 131, 132
  code4logOpenOff, 131, 132
  code4optStart, 105, 117
  code4optSuspend, 105
  code4tranCommit, 129, 130
  code4tranRollback, 129, 130
  code4tranStart, 129, 130, 131
  code4unlockAuto, 111, 132
Composite Data Files. *See* Relations
  moving between composite records, 74
Composite Record. *See* Relations
Constants
  CodeBase constants, 17
Converting Date Formats. *See* Date Operations
Current Record. *See* Data Files

—D—

Data File Functions
  d4alias, 36
  d4aliasSet, 36
  d4append, 30, 111, 114
  d4appendBlank, 30, 111
  d4appendStart, 30
  d4bottom, 22, 47, 81, 111, 121
  d4close, 21, 22
  d4create, 28, 29, 38, 44, 57, 61, 125, 131
  d4delete, 31, 34, 109, 111, 121
  d4deleted, 34
  d4flush, 111, 115
  d4go, 22
  d4lockAdd, 119
  d4lockAddAll, 119, 123, 124, 133
  d4lockAddAppend, 119
  d4lockAddFile, 119
  d4lockAll, 123, 124, 133
  d4log, 130
  d4memoCompress, 35, 111
  d4numFields, 20, 24
  d4open, 14, 20, 21, 29, 45, 117, 131
  d4optimize, 105
  d4optimizeWrite, 105
  d4pack, 111
  d4position, 121

d4recall, 31, 34, 109, 111
d4recCount, 36, 73
d4recWidth, 36
d4refresh, 105, 123
d4refreshRecord, 105, 123
d4reindex, 60, 111, 124
d4seek, 22, 53, 54, 55, 56, 81, 111, 121, 132
d4seekDouble, 22, 53, 54, 81, 111, 121
d4seekN, 22, 53, 54, 81, 111, 121
d4seekNext, 22, 53, 54, 55, 56, 81, 111, 121
d4seekNextDouble, 22, 53, 54, 56
d4seekNextN, 22, 53, 54, 56, 81, 111, 121
d4skip, 20, 22, 23, 47, 81, 118, 123
d4tagSelect, 47
d4top, 20, 22, 23, 47, 81, 111, 121
d4unlock, 111, 112, 114, 115, 118, 133
d4write, 111
d4zap, 111, 124
Data Files, 7
alias, 36
alias, changing, 36
buffering. *See* Memory Optimizations
closing, 22
composite. *See* Relations
compressing memo files, 35
creating, 26, 28, 29
current record, 19
deletion flag. *See* Deletion Flag
exclusive access. *See* Exclusive Access
field structure, 19, 28, 38
index files, creating with, 44
locking. *See* Locking
master. *See* Relations
moving between records, 22
natural order. *See* Natural Order
opening, 21, 29
packing, 34
queries. *See* Queries
read only mode. *See* Read Only Mode
record buffer, 19
record count, 36
record number. *See* Record Number
relations. *See* Relations
slaves. *See* Relations
top master. *See* Relations
DATA4, 18, 20, 29, 36, 72, 73
obtaining the pointer to, 18, 21, 29
Databases. *See* Data Files
Date Fields
assigning values, 32, 88
creating, 28
retrieving contents, 26, 88
Date Formats. *See* Date Operations
Date Functions
date4assign, 89
date4cdow, 90
date4cmonth, 90
date4day, 91
date4dow, 91
date4init, 90
date4isleap, 91
date4long, 89
date4month, 91
date4timeNow, 91
date4today, 91
date4year, 91
Date Operation
leap year, 91
Date Operations, 87
assigning values to date fields, 32, 88
CCYYMMDD, 88
converting between date formats, 89
date formats, 87
date pictures, 87
day of the month, 90, 91
day of the week, 90, 91
julian day format, 88
retrieving contents from date fields, 26, 88
retrieving the current day, 91
retrieving the current time, 91
standard format, 88
testing for valid dates, 90
years, 91
Date Pictures. *See* Date Operations
dBASE Expressions. *See* Reference Guide
filter expression, 49, 50
index expression, 40
master expression, 63, 73
query expression, 76
sort expression, 77
dBASE Functions, 41
DELETED(), 49
STR(), 57
UPPER(), 41, 85
dBASE IV
descending order tags, 43
index files, 39
index filters, 48
locking protocol, 109
dBASE operators
'+', '-', 57
Deadlock. *See* Multi-User
Deleting records. *See* Records
Deletion Flag, 8, 32
Descending Order. *See* Tags
Dynamic Allocation
nodes. *See* Linked Lists

—E—

e4unique, 50
Error Functions
error4exitTest, 20, 21
Exact Matches
seeks. *See* Seeking
Exclusive Access
opening files exclusively, 119, 121

—F—

Field Attributes
  decimal, 8, 28
  length, 8, 28
  name, 8, 28
  type, 8, 28
Field Functions
  f4assign, 31, 88
  f4assignDouble, 32
  f4assignInt, 32
  f4assingLong, 32
  f4decimals, 37
  f4double, 26
  f4int, 26
  f4len, 37
  f4long, 26
  f4memoAssign, 31
  f4memoStr, 25
  f4name, 36
  f4str, 25, 88
  f4type, 37
Field Types
  character. *See* Character Fields
  Date. *See* Date Fields
  determining, 37
  Floating Point. *See* Numeric Fields
  Logical. *See* Logical Fields
  Memo. *See* Memo Fields
  Numeric. *See* Numeric Fields
FIELD4, 19, 24, 36
FIELD4INFO
  dec member, 28
  len member, 28
  name member, 28
  obtaining a copy of, 38
  type member, 28
Fields
  accessing, 23
  assigning contents, 31
  character fields. *See* Character Fields
  creating, 28
  date fields. *See* Date Fields
  decimal, 37
  field attributes. *See* Field Attributes
  field number, 24
  field type, 37
  floating point fields. *See* Numeric Fields
  generic access, 24, 25
  length, 37
  name, 37
  numeric fields. *See* Numeric Fields
  numeric values, retrieving, 26
  qualifying, 36, 65, 77
  referencing, 23
  referencing by field number, 24
  referencing by name, 24
  retrieving contents, 25
File Functions
  file4open, 103
  file4optimize, 105
  file4optimizeWrite, 105
  file4refresh, 105, 123
Files
  buffering. *See* Memory Optimizations
  locking. *See* Locking
  opening exclusively, 110, 119, 121
  overwriting, 29
Filters
  queries. *See* Queries
  tag filters. *See* Tags
FoxPro
  descending order tags, 43
  index files, 39
  index filters, 48
  locking protocol, 109

—G—

Group Files, 57
  bypassing, 57
  creating, 57

—I—

Index Files, 39
  .CDX, 9, 39, 40, 44, 57
  .MDX, 9, 39, 43, 44, 57
  .NTX, 9, 39, 43, 44, 57, 58, 59, 60
  buffering. *See* Memory Optimizations
  creating, 41, 44
  descending order. *See* Tags
  exclusive access. *See* Exclusive Access
  filter expression, 49, 50
  index expression, 40
  index key, 40
  locking. *See* Locking
  non-production indexes, 41, 43
  opening, 45, 59
  production indexes, 9, 41, 43, 44
  read only mode. *See* Read Only Mode
  reindexing, 60
  reindexing a single file, 60
  reindexing all files, 60
  search key. *See* Seeking
  seeking. *See* Seeking
  tags. *See* Tags
  unique keys, 43, 50, 51, 85
INDEX4, 44, 60
Index4 Functions
  i4create, 44, 57, 60, 61
  i4open, 45, 59, 60
  i4reindex, 111
  i4tagInfo, 61
Indexes. *See* Index Files

—J—

Julian Day Format. *See* Date Operations

—K—

Keys
index key. *See* Index Files
lookup key. *See* Relations
search key. *See* Seeking

—L—

LINK4, 95
Linked List Functions
l4add, 95, 99
l4addAfter, 96
l4addBefore, 96
l4first, 96
l4last, 96
l4next, 96
l4numNodes, 97
l4pop, 97
l4prev, 97
l4remove, 97
Linked Lists
adding nodes, 95, 98
deallocating nodes, 99
double linked list, 93
dynamic allocation, 97
inserting nodes, 96
next node, 96
node structure, 93, 95
nodes, 93
number of nodes, 95, 97
previous node, 97
queues, 100
removing nodes, 97, 99
selected node, 95
stacks, 99
traversing, 96
LIST4, 95
Locking
automatic data file locking, 111
automatic record locking, 111, 121
automatic record unlocking, 111, 132
Clipper locking protocol, 109
data file locking, 110
dBASE IV locking protocol, 109
efficient locking, 123
enforced, 109, 111, 115, 121
explicit, 31, 121, 132, 133
FoxPro locking protocol, 109
group locks, 119
index and memo file locking, 110
lock attempts, 120, 124
protocols, 109
record locking, 110
unlocking, 111, 132
Locking Protocols. *See* Locking
Log Files, 130, 131
LOG4ALWAYS, 130
LOG4ON, 130
LOG4TRANS, 130
Logging
automatic, 130
changing the status, 32, 130
Client-Server configuration, 130, 132
current status, 119, 120
Stand-Alone configuration, 130
transaction, 127, 130
Logical Fields
assigning values to, 32
creating, 28
retrieving contents, 25
Lookups. *See* Relations

—M—

Master Data File. *See* Relations
Memo Fields
assigning values to, 31
creating, 28
Memo File
buffering. *See* Memory Optimizations
compressing, 35
exclusive access. *See* Exclusive Access
locking. *See* Locking
Memory
buffering. *See* Memory Optimizations
dynamic allocation, 12, 37, 38, 61, 97, 98
memory corruption, 11
static, 12
Memory Optimizations, 103
activating, 105
buffering, 103, 123
memory requirements, 105, 123
multi-user mode, 106, 116
read optimizations, 104, 106
refreshing the buffers, 105
removing all, 106
single user mode, 106
specifying files, 103, 105
suspending, 105
using, 103
write optimizations, 104, 106
Multi-User
appending records, 114, 116
applications, 109
applications, creating, 110
common tasks, 112
deadlock, 118
exclusive access. *See* Exclusive Access
finding records, 114
listing records, 116
lock attempts. *See* Locking
locking protocols. *See* Locking

memory optimizations. *See* Memory Optimizations
modifying records, 115
opening files, 113
optimizations, 116
read only mode. *See* Read Only Mode

—N—

Natural Order
selecting, 47
Node Structures. *See* Linked Lists
Nodes. *See* Linked Lists
Numeric Fields
assigning values, 31, 32
creating, 28
retrieving contents, 26

—O—

Optimizations. *See* Memory Optimizations
using Query Optimization, 86

—P—

Packing. *See* Data Files
Partial Matches
relations. *See* Relations, approximate match
seeks. *See* Seeking
Performance Issues
efficient appending, 124
efficient locking, 123
efficiently sorting a query set, 78
functions that reduce performance, 123
general tips, 125
memory optimization, 123
time consuming functions, 124
Production Indexes. *See* Index Files, production indexes

—Q—

Queries, 63, 75
creating, 76
generating, 76, 78
query expression, 76
query set, 75, 76, 78
sort expression, 77
sorting, 77
using Query Optimization, 83
Query Expression. *See* Queries
Query Optimization, 83
how to use, 86
queries, 84
relate4optimizeable, 86
relations, 83
requirements, 84
Query Set. *See* Queries
Queues
adding nodes, 95
removing nodes, 99

—R—

r4after, 55
r4bin, 28
r4date, 28
r4descending, 43
r4eof, 56
r4float, 28
r4gen, 28
r4locked, 120
r4log, 28
r4logOff, 130, 131
r4logOn, 130
r4memo, 28
r4num, 28
r4str, 28
r4success, 17, 23, 55, 56
r4unique, 50
r4uniqueContinue, 50
Read Only Mode
opening files in, 110, 111, 119, 120
Read Optimizations. *See* Memory Optimizations
Record Buffer. *See* Data Files
Record Buffering. *See* Memory Optimizations
Record Count, 36
Record Number, 8
Records, 8, 74
adding, 30
appending, 30
buffering. *See* Memory Optimizations
deleting, 32, 34
deletion status, 34
listing records, 19
physically removing, 34
recalling, 34
removing, 34
undeleting, 34
width, 36
Reference
passing by, 13
Reindexing. *See* Index Files
RELATE4, 72
relate4approx, 74
relate4blank, 74
relate4exact, 74
relate4scan, 74
relate4skipRec, 74
relate4terminate, 74
Relation Functions
relate4bottom, 74
relate4changed, 79
relate4createSlave, 72
relate4doAll, 81, 82
relate4doOne, 82
relate4errorAction, 74

relate4init, 72
relate4lockAdd, 119
relate4skip, 74
relate4skipEnable, 75
relate4sortSet, 76, 77
relate4top, 74
relate4type, 74
Relation Set. *See* Relations
Relation Types. *See* Relations
Relations, 63
approximate match relation, 70
complex, 67
composite data file, 65
composite record, 64
creating, 71
error action, 74
exact match relation, 68
lookup key, 63
lookups, 80, 81
many to many relation, 69
many to one relation, 68
master, 63, 72
master expression, 63, 73
multi-layered, 67
one to many relation, 69
one to one relation, 68
relation set, 67, 72
scan relation, 69
skipping, 75
skipping, backwards, 75
slave tag, 63, 73
slaves, adding, 72
slaves, creating without tags, 73
top master, 67, 72
top master, specifying, 72
types, 68
using Query Optimization, 83
Removing
removing records. *See* Records
Rereading, 105

—S—

Seeking
character tags, 53
compound keys, 56
date tags, 54
exact matches, 55
incremental, 54
no matches, 56
partial matches, 55
performing seeks, 53
search key, 52
soft seek. *See* Seeking, partial matches
Single User
memory optimizations, 106
sizeof, 11
Slave Data Files. *See* Relations
Soft Seek. *See* Seeking, partial matches
Sorting
queries. *See* Queries
records. *See* Indexes
Stacks
implementing with linked lists, 99
popping nodes, 99
pushing nodes, 99
Standard Format. *See* Date Operations
Strings
copying to fields, 31
retrieving from fields, 25
Structures
CodeBase structures, 17
passing as parameters, 13
Switches
S4CLIPPER, 110
S4FOX, 110
S4LOCK_HOOK, 120, 133
S4MDX, 110
S4OFF_MULTI, 104, 115
S4OFF_OPTIMIZE, 106

—T—

Tables, 7
Tag Functions
t4open, 60
t4uniqueSet, 85
TAG4, 41, 46, 47, 48, 52, 59, 60, 71, 73, 79, 80, 81, 112, 127
TAG4INFO, 43
arrays, 43, 44
descending member, 43
expression member, 43
filter member, 43
name member, 43
unique member, 43
Tags
creating, 43
currently selected, 47
default tag, 47
deleting, 45
descending order, 43
effects of, 47
filters, 43, 48
filters, creating, 49
index expression, 40, 43
index expression, compound, 41
index key, 40, 55
name, 43
natural order. *See* Natural Order
referencing, 46
referencing by name, 46
search key. *See* Seeking
seeking. *See* Seeking
selecting, 47
slave tag. *See* Relations
Top Master. *See* Relations
Transactions, 127

client-server configuration, 130
committing, 127, 129
implementation, 130
log files, 131
logging status, 130
rollback, 127, 129
stand-alone configuration, 132
starting, 127, 129
Tuples, 7

—U—

u4alloc, 98, 99
u4allocFree, 105
u4free, 99
Unlocking. *See* Locking

—W—

Write Optimizations. *See* Memory Optimizations