CodeBase 6.0™ Reference Guide

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

Sequiter Software Inc.

Contents

Introduction

This is the *CodeBase 6 C API Reference Guide*. Its purpose is to provide a way to quickly lookup information on CodeBase.

First time CodeBase 6 users should consult the *Getting Started* Book and the *User's Guide*. The *Getting Started* book explains how to install CodeBase 6, how to run the examples in the *Reference Guide* and the *User's Guide* and other important issues. The *User's Guide* explains the purpose of the library, CodeBase 6 concepts and provides examples.

The *Reference Guide* systematically documents the entire CodeBase library in alphabetical order. It comprehensively covers all of the information you need to know about any aspect of the library.

Covered in this *Reference Guide* are topics including conditional compilation switches and the various function modules. There are also a number of appendices providing information on error messages, return codes and dBASE expressions.

In summary, the best way to use CodeBase 6 is to start with the *Getting Started* book and the *User's Guide* and then refer to the *Reference Guide*, in order to obtain more comprehensive information on any particular function or conditional compilation switch.

2 CodeBase

Conditional Compilation Switches

C/C++ supports conditional compilation by use of the following language construct:

```
#ifdef SWITCH_NAME

    // Some code

#else

    // Some other code

#endif
```

CodeBase takes advantage of conditional compilation to help support different compilers and configurations. Following are the conditional compilation switches which can be used when compiling CodeBase source code and applications. The switches are located in the 'd4all.h' header file.

If any of the switches listed in this chapter are used when compiling the CodeBase library, then they must also be used when compiling the application. The switches have been broken into several categories: compiler, file format, screen output, operating environment, client-server configuration, general configuration and spoken language.

Note

As documented in the User's Guide, we suggest defining the appropriate switches at the top of include "d4all.h". Be sure to consult the Getting Started section and try example "d4exampl.c" after switching to or building a new library. This example has special logic which verifies that the switches used when the library was built are compatible with the switches used when building this example.

Compiler Switches

CodeBase has conditional compilation switches for a number of specific compiler products. Usually, the compiler has a predefined switch that CodeBase uses. In this case, you do not have to explicitly set any switch. If you do, the name of the switch is documented in compiler support file 'COMPILER.TXT'. Refer to the Getting Started documentation in the User's Guide.

File Format Switches

CodeBase supports several different file formats. Note that these switches change the multi-user protocol as well. This is because CodeBase is multi-user compatible with dBASE IV, FoxPro and Clipper. Only one of the following file format switches may be defined at a time.

Switch Name	Description
S4CLIPPER	This switch adds support for Clipper file formats. Specifically, the **.NTX** index files and the Clipper memo files are supported.

	When the **S4CLIPPER** switch is defined, all of the CodeBase functions can be used. In addition, the functions **t4open** and **t4close** can be used. Refer to the section on Clipper support in the User's Guide.
S4FOX	This switch adds support for FoxPro file formats including Visual FoxPro 3.0 which supports code pages and collation sequences. This includes **.CDX** compound index files, compact **.IDX** index files, and **.FPT** memo files. Because compound and compact index file formats are almost identical and they can easily be distinguished from one another, both are supported at the same time. When using the **S4FOX** switch, a single program could open up both **.CDX** and "compact" **.IDX** index files at the same time. However, the default file name extension is **.CDX**. If you wish to use an **.IDX** index file, it is necessary to explicitly specify the **.IDX** file name extension. See Also: FoxPro Configuration Switches
S4MDX	This switch adds support for dBASE IV file formats. Specifically, the **.MDX** index files and the dBASE IV memo files are supported.

FoxPro Configuration Switches

These switches provide support for the code pages and collation sequences that are used by Visual FoxPro 3.0. These switches can only be defined if the **S4FOX** switch has also been defined.

The following switches provide support for code pages and and one or both may be defined at a time.

Switch Name	Description
S4CODEPAGE_437	U.S. MS-DOS code page number 437.
S4CODEPAGE_1251	Windows ANSI code page number 1252.

The following switches provide support for international languages and either one or both may be defined.

Switch Name	Description
S4GENERAL	Adds support for the general collation sequence. This sort ordering supports English, French, German, Modern Spanish, Portuguese, and other Western European languages
S4MACHINE	This switch is automatically defined when **S4FOX** is defined. This sort ordering supports the older FoxPro collation sequences.

Screen Output Switches

CodeBase can be configured to work in almost any operating environment, since it is a database management library. By default, when CodeBase needs to display output, it sends error messages to **stderr** and other output to **stdout**.

In order to configure CodeBase to output using a different output method from one of the methods described here, you need to modify the error message reporting. Refer to conditional compilation switch **E4HOOK**. In addition, if the report module is used, you need to refer to the report module documentation.

Define one of these switches when non-standard output is required.

Only one of these switches can be defined at once.

Switch Name	Description
S4CODE_SCREENS	Define this switch when Sequiter's DOS screen management library CodeScreens is being used.
S4CONSOLE	**S4CONSOLE** should be defined when you want to run an application from the console or send output to the console.

Operating System Switches

The standard version of CodeBase supports DOS, OS/2 and Microsoft Windows.

If you are using Unix or a different operating environment, you need to purchase the portability version of CodeBase. The portability version comes with additional documentation and software which is useful in order to get CodeBase working under other operating systems.

There are two sets of switches in this catagory. The first set of switches define the type of library that will be used, static or dynamically linked. The second set of switches defines the type of operating system being used. Only one switch from each set may be defined at once.

Switch Name	Description
S4STATIC	This switch must be defined when using a CodeBase static library.
S4DLL	This switch must be defined when using a CodeBase dynamic link library.

Switch Name	Description
S4DOS	Use this switch when compiling under DOS.
S4OS2	This switch is used when compiling under OS/2.

S4WIN16	Use this switch when compiling CodeBase under WIN16.
S4WIN32	Use this switch when compiling CodeBase under WIN32.

Client-Server Configuration Switches

These switches determine whether the target being built is a client or a non-client. Only one of these switches can be defined.

Switch Name	Description
S4CLIENT	When the **S4CLIENT** switch is defined, the application is a client in a client-server configuration. Therefore, when this switch is defined, the **S4STAND_ALONE** switch can NOT be defined.
S4STAND_ALONE	When the **S4STAND_ALONE** switch is defined, the application is NOT a client in a client-server configuration and the **S4CLIENT** switch can NOT be defined.

If **S4CLIENT** is defined then the following switches are used to determine the default communication protocol. Only one of these switches can be defined.

Under Windows, the communication protocol can be dynamically set by calling the **code4connect** function with the desired protocol and thus override default protocol. Under DOS, the communication protocol used is always the default protocol specified by one of these switches.

Switch Name	Description
S4SPX	If **S4SPX** is defined, then the default communication protocol is for Novell SPX.
S4WINSOCK	If **S4WINSOCK** is defined, the default communication protocol is for Window Sockets.

General Configuration Switches

Any number of the following switches can be defined at the same time.

Switch Name	Description
S4CB51	When this switch is defined, then the 5.1 API can be used in CodeBase 6.0. The new locking protocol that is used in CodeBase 6.0 will not available when this switch is defined. The following

	CodeBase 6.0 functions will not be available when this switch is defined: **d4seekNext**, **d4seekNextDouble**, **d4seekNextN**.
S4LOCK_HOOK	This switch modifies the CodeBase locking functions so that they call function **code4lockHook** when a lock fails. It is up to the programmer to define function **code4lockHook**. **code4lockHook** is designed to allow the end user the opportunity to retry the lock. The function is found in the file 'C4HOOK.C' and is described by the following**:** **Usage:** int code4lockHook (CODE4 *cb, const char *fileName, const char *userId, const char *networkId, long item, int numAttempts) **Parameters:** cb — A reference to the **CODE4** structure. fileName — This is the name of the file for which the lock contention occurred. userId — This is the name of the user who holds the contentious lock. Use null if the *userId* is unknown. networkId — This is the network identification of the user who hold the contentious lock. Use null if the *networkId* is unknown. item — This is the item for which the lock is contentious. The value of *item* can be one of the following: > 0 This the record number of the locked record. 0 This indicates that the append bytes are locked. - 1 This indicates that the file is locked. - 2 This indicates that item locked cannot be identified. numAttempts — This is the number of times the lock has been attempted. **Returns:** 0 This return indicates to the top level CodeBase function to try again. r4locked This return indicates to the top level CodeBase function to return **r4locked**. < 0 This return indicates to the top level CodeBase

	function to return a negative value. Generally, if this is done, it is also appropriate for the **code4lockHook** to call **error4set** to set a CodeBase error. **NOTE:** In general, calling CodeBase functions from within the **code4lockHook** function is discouraged since it can cause conflicts within the internal structure settings. For example, calling **d4go,** to change the current record in the data file for which the lock has failed, will cause internal consistency problems when returning to the function that called the hook function. However, it is acceptable to call **error4set** from **code4lockHook,** when applicable.
S4MAX	This switch adds code to limit the amount of memory functions **u4alloc**, **u4allocErr** and **u4allocFree** will allocate in total. When this switch is used, it is necessary to also use the **E4MISC** switch. CodeBase puts the current amount of memory allocated in the global variable **long mem4allocated**. The maximum amount of memory is placed in the global variable **long mem4maxMemory**. This is useful for simulating conditions when little memory is available. By default, **mem4maxMemory** is 16K. When using switch **S4MAX**, CodeBase assumes that **mem4maxMemory** is less than the actual maximum available memory.
S4SAFE	Setting this switch causes CodeBase to immediately update file lengths after information has been written to file. This avoids delayed "out of disk space" error messages when memory write-optimization is being used.
S4TIMEOUT_HOOK	When this switch is defined, CodeBase calls **code4timeoutHook** whenever a CodeBase client is waiting to receive a message from the server and the **CODE4.timeout** has expired. It is up to the programmer to define the function **code4timeoutHook**. This function is designed so that the programmer may specify what action should take place when a time out occurs. This function is especially useful when used in conjunction with **code4lockHook** in order to determine whether CodeBase is delaying because of a lock or is waiting for a communication reply. The function is provided in the file 'C4HOOK.C'. The description of **code4timeoutHook** is as follows: **Usage:** int code4timeoutHook(CODE4 *cb, int numAttempts, long numHundredths) **Parameters:** cb This is a reference to the **CODE4** structure. numAttempts This the number of times the hook function has

	been called for this particular time out.
numHundredths	This is the total elapsed time that the network has been delaying, in hundredths of seconds.
Returns:	A return of zero indicates that the application should continue to wait for a response. Any other value will be passed up to the calling functions.

Error Switches

The following switches, when defined within the CodeBase library, determine what actions occur when an error is generated. Any number of the following switches can be defined at the same time.

Switch Name	Description
E4ANALYZE	This switch is used to perform structure analysis on internal CodeBase structures at runtime. This switch is especially useful during the development stage of applications. It should not be used during the final application compiles, since it adds significant code to CodeBase and decreases performance. When this switch is defined, it automatically defines **E4LINK** and **E4PARM_LOW**. **E4PARM_LOW** checks the parameters in low level CodeBase functions.
E4DEBUG	This switch automatically defines the following error switches: **E4PARM_HIGH, E4ANALYZE, E4MISC** and **E4STOP_CRITICAL**

E4HOOK	This switch modifies the CodeBase error functions so that they call function **error4hook** to display any error messages. It is up to the programmer to modify function **error4hook**. The prototype of **error4hook** as found in 'd4data.h' and is as follows: **Usage:** void error4hook(CODE4 *codebase, int errCode1, long errCode2, const char *desc1, const char *desc2, const char *desc3) **Description:** The purpose of the this function is to allow the programmer to easily alter or turn off error reporting. When CodeBase is built with the conditional compilation switch **E4HOOK**, **error4hook** is called by **error4** and **error4describe** to display messages. By default, **error4hook** does nothing. Consequently, to turn off error reporting, just define **E4HOOK** and do nothing else. To alter error reporting, alter **error4hook** to display error messages as desired. Function **error4hook** is defined statically in 'C4HOOK.C'. **Parameters:** The parameters to **error4hook** correspond to parameters passed to **error4** and **error4describe.** If **error4hook** is displaying error messages for **error4** then the parameters *desc2* and *desc3* are null. The parameter *desc1* may also be null depending on whether the error code has any auxiliary information associated with it. Call **error4text** to get the error string associated with either *errCode1* or *errCode2*. Note that if **E4OFF_STRING** or **E4OFF** is defined, the **error4text** function will return the string "Invalid or unknown Error Code". **See Also:** **error4, error4describe**, **E4OFF, E4OFF_STRING, error4text**
E4LINK	This switch provides link-list error checking in the CodeBase library. Whenever an item is added or removed from a CodeBase linked list the list is "run" to verify its internal integrity. Since CodeBase linked lists are all circular, this is easy to do. This is most useful when debugging applications which make use of the provided link-list functions. Compiling with this switch may increase the application execution time by a factor of 2 or more.

E4MISC	Using this switch adds extra code for CodeBase self diagnostics and additional error checking. Following are the effects of using the **E4MISC** switch: • Whenever a memory item is allocated, the returned pointer is saved in a list. When the pointer is subsequently freed, it is looked up in the list to verify that it was previously allocated. In addition, an extra 22 bytes are allocated as follows: FFFFFFFFFFLL, Allocated Memory, FFFFFFFFFF The 'F' represents a special fill character, and the 'L' is the number of allocated bytes. When the allocated memory is subsequently deallocated, a check is made to ensure that the fill characters have not changed. This is useful because C programmers sometimes modify more memory than they allocate. • Function **mem4freeCheck(int maxAlloc)** is also defined. This function, whose prototype is defined in 'd4declar.h', returns the number of memory allocations for which there have been no subsequent deallocation. If this number is greater than the value of parameter *maxAlloc*, a severe error is generated.
E4OFF	This switch disables all CodeBase error messages. This performs the same function as runtime flag **CODE4.errOff**. This switch does not affect the functionality of an application, however, only error numbers are available in an application. Using this switch results in a smaller executable size, since the text for the error messages is not included. Error hooks can still be used and this switch also does not effect the actual generation of CodeBase errors. When **E4OFF** is defined then **E4OFF_STRING** is implicitly defined. Defining this switch will cause **error4text** to return the string "Invalid or Unknown Error Code" whenever it is called.
E4OFF_STRING	This switch removes all the extended error strings from the CodeBase application. Consequently, CodeBase will display a unique error number and the basic error description associated with the error. Defining this switch will decrease the size of the application, since the long static strings will not be included in the executable. This default implementation under DOS is to have the library compiled with this switch. Defining this switch will cause **error4text** to return the string "Invalid or Unknown Error Code" whenever it is called.
E4PARM_HIGH	This switch causes CodeBase to include parameter checks for high-level functions. In general the library should be compiled with this switch defined. This switch should also be defined in end-user applications, since it will prevent applications from crashing when bad parameters are passed to CodeBase functions. The default setting for this switch is to be turned on.

E4PAUSE	Compiling with this switch causes CodeBase programs to prompt the user to "press a key to continue" (when **S4CONSOLE** is defined) whenever an error is encountered. This switch only applies to **S4CONSOLE** applications. The default setting for this switch is to be turned on.
E4STOP	This switch causes CodeBase to halt the execution whenever any CodeBase error is encountered. Note that using an error hook will override this switch.
E4STOP_CRITICAL	This switch causes CodeBase programs to halt whenever any critical error is encountered. Critical errors include the following: • Out of memory. • Bad input parameter. • Corrupt memory detected. • Corrupt structures detected. • Unexpected internal results (e.g. corrupt index) Using an error hook will override this switch. The default setting for this switch is to be turned on.

Library Reducing Switches

The following switches, when defined within the CodeBase library, reduce the size of the library by removing some its functionality. As a result, applications linking into a reduced library will be smaller and in many cases faster than the standard library. When these switches are defined, many CodeBase functions will be unsupported. For example, attempting to append a new record to a data file with **S4OFF_WRITE** defined is unsupported and will generate an error. Listed below are the switches, and the functions they effect.

Any number of the following switches can be defined at the same time.

Switch Name	Description
S4OFF_INDEX	This switch is used to build a library that does not use indexes. All code related to indexes is removed from the library. This dramatically reduces the size of the CodeBase library. The following functions are disabled and always return 'success' when used with the **S4OFF_INDEX** switch: **d4reindex, d4tagSync. i4fileName, t4alias, t4expr, t4filter** return null when this switch is defined. The following functions return un-initialized objects: **d4index, d4tagSelect, d4tagDefault.** When **S4OFF_INDEX** is defined, the following functions are unsupported and always generate an **e4notIndex** error if called: **d4freeBlocks, d4seek, i4close, i4create, i4open, i4reindex, i4tag, t4open.**

	This switch can only be defined when the application has been compiled with the **S4STAND_ALONE** switch defined.
S4OFF_MEMO	References to memo files are taken from the library. This reduces the executable size when memo files are not needed. The following functions are disabled and always return 'success' when used with the **S4OFF_MEMO** switch: **f4memoFree.** When the **memo field** functions are used to access memo fields (as opposed to any other 'generic' type of field) , the following functions are unsupported and always generate an **e4notMemo** error if used. **f4memoAssign, f4memoNcpy, f4memoLen, f4memoPtr, f4memoStr, d4memoCompress.** This switch can only be defined when the application has been compiled with the **S4STAND_ALONE** switch defined.
S4OFF_MULTI	This switch compiles a single-user version of the CodeBase library. No file locking is performed or allowed when this switch is used. If an application is to be used in a single-user environment, this switch results in smaller code sizes and much faster execution times. This switch should also be used when using a compiler that does not perform file locking. This switch should not be used when an application is running in a multi-tasking environment and more than one instance may be running. The following functions do nothing but return 'success' when **S4OFF_MULTI** is defined (Note: an application will be slightly smaller if they are not called): **d4lock, d4lockAll, d4lockAppend, d4lockFile, d4lockAdd, d4lockAddAll, d4lockAddAppend, d4lockAddFile, d4unlock, file4lock, file4refresh, file4unlock, relate4lockAdd, code4lock, code4unlock.** This switch can only be defined when the application has been compiled with the **S4STAND_ALONE** switch defined.
S4OFF_OPTIMIZE	This switch disables CodeBase memory optimization. This results in slower execution time in both single and multi-user applications. It can be used to reduce CodeBase memory requirements and code size. When this switch is used, the following functions do nothing but return 'success' (Note: an application will be slightly smaller if they are not called) : **code4optAll, code4optStart, code4optSuspend, d4optimize, d4optimizeWrite, file4optimize, file4optimizeWrite, file4refresh.** When the switch **S4CLIENT** is defined, then this switch is automatically defined as the default setting.
S4OFF_REPORT	This switch can be used to remove all CodeReporter reporting functions and functionality from the CodeBase library (i.e. functions which are documented in the CodeReporter reference manual). Using this switch will reduce the size of the library and resultant executables, even for programs which do not otherwise call the CodeReporter reporting functions.

S4OFF_TRAN	Use this switch to remove transactional capabilities from the CodeBase library. This means that the transactional functions can not be called by the client, but the CodeBase database server will retain transactional file capabilities and compatabilities. For the stand-alone version of CodeBase, compiling with this switch removes the compatability with other CodeBase applications that have transactional capabilities.
S4OFF_WRITE	This switch is used to create read-only applications. All code that performs disk writes is excluded from the CodeBase library. The following functions do nothing but return 'success' when **S4OFF_WRITE** is defined: **code4flushFiles, d4flush, d4optimizeWrite.** The following functions are completely unsupported, and always generate an **e4notWrite** error. **d4append**, **d4appendBlank, d4appendStart, d4changed, d4create, d4deleteRec, d4memoCompress, d4pack, d4recall, d4reindex, d4write, d4zap, f4assign, f4assignDouble, f4assignLong, i4create, i4reindex, i4tagAdd.** This switch can only be defined when the application has been compiled with the **S4STAND_ALONE** switch defined.

Spoken Language Switches

Following are the switches for "people" language support. There are two supported sort orderings for English and German.

Switch Name	Description
S4ANSI	This switch is defined to make CodeBase use the ANSI sort ordering instead of the default ASCII ordering. The ANSI sort ordering makes CodeBase use the same ordering as the Microsoft Windows comparison functions. Unfortunately, this makes CodeBase incompatible with the dBASE, FoxPro and Clipper sort orderings.
S4DICTIONARY	For the German version of CodeBase (if **S4GERMAN** is defined), this switch determines whether the sort order is by the default (phone book sort ordering), or dictionary ordering (if **S4DICTIONARY** is defined).
S4FINNISH	This switch is defined to get Finnish language support.
S4FRENCH	This switch is defined to get French language support.
S4GERMAN	This switch is defined to get German language support.
S4NORWEGIAN	This switch is defined to get Norwegian language support.
S4SCANDINAVIAN	This switch is defined to get Scandinavian language support.
S4SWEDISH	This switch is defined to get Swedish language support.

CodeBase Members and Functions

CODE4 Structure members

accessMode	errOpen	memExpandBlock	memStartData
autoOpen	errRelate	memExpandData	memStartIndex
codePage	errSkip	memExpandIndex	memStartLock
collatingSequence	errTagName	memExpandLock	memStartMax
createTemp	fileFlush	memExpandTag	memStartTag
errCreate	hInst	memSizeBlock	optimize
errDefaultUnique	hWnd	memSizeBuffer	optimizeWrite
errExpr	lockAttempts	memSizeMemo	readLock
errFieldName	lockAttemptsSingle	memSizeMemoExpr	readOnly
errGo	lockDelay	memSizeSortBuffer	safety
errOff	lockEnforce	memSizeSortPool	singleOpen
errorCode	log	memStartBlock	

CodeBase Functions

code4calcCreate	code4indexExtension	code4logCreate	code4tranCommit
code4calcReset	code4init	code4logFileName	code4tranRollback
code4close	code4initUndo	code4logOpen	code4tranStart
code4connect	code4lock	code4logOpenOff	code4tranStatus
code4data	code4lockClear	code4optAll	code4unlock
code4dateFormat	code4lockFileName	code4optStart	code4unlockAuto
code4dateFormatSet	code4lockItem	code4optSuspend	code4unlockAutoSet
code4exit	code4lockNetworkId	code4timeout	
code4flush	code4lockUserId	code4timeoutSet	

CodeBase uses the **CODE4** structure to maintain settings and error codes that apply to most CodeBase functions. Using a structure for these settings instead of global variables makes it technically feasible to use CodeBase as a dynamic link library. Generally, only one **CODE4** structure is used in any one application.

WARNING

It is generally recommended that only one **CODE4** structure be constructed per application. If more than one server connection is necessary within an application, multiple **CODE4** structures may be used. However, modules, which function on more than one database (CodeReporter, CodeControls, expression, relation etc.) may not be used to integrate databases found on separate **CODE4**-linked servers or separate **CODE4** structures.

Before calling any other CodeBase function, it is critical to remember to call **code4init**. **code4init** allocates memory for the structure, sets its default values, and returns a pointer to the structure. Pointers to this structure can then be passed to other functions such as **d4open**.

The **CODE4** structure contains flags that other functions use to determine how to react to different situations. For instance, there are flags that other functions use to determine the current locking method, memory usage and error handling.

Any documented **CODE4** member value can be changed at any time by the application program. However, the change only influences CodeBase's future behavior. For example, if the **CODE4.accessMode** flag is changed, then only subsequently opened files are opened differently.

The following is the list of true/false flags which are documented in this section:

CODE4 Member Flag	Question Answered
(int) autoOpen	Are production index files automatically opened with their data files?
(int) createTemp	Are created files automatically deleted once they are closed?
(int) errCreate	Is an error message generated when a file cannot be created?
(int) errExpr	Is an error message generated when an expression cannot be understood ?
(int) errFieldName	Is an error message generated when an invalid field name is encountered?
(int) errGo	Is an error message generated when an attempt is made to go to an invalid record?
(int) errOff	Should all error messages be disabled?
(int) errOpen	Is an error message generated when a file cannot be opened?
(int) errRelate	Should relation functions generate an error message if a record in the slave data file cannot be located?
(int) errSkip	Is an error message generated when attempting to skip from a non-existent record (eg. after **d4pack**).
(int) errTagName	Should attempts to construct a **TAG4** structure with an invalid tag name generate an error message?
(int) fileFlush	Should a hard flush of the file be done when a write occurs?
(int) lockEnforce	Must the record be locked before modifications to the record buffer are made?
(int) optimize	Should files automatically be optimized when opened and/or created?
(int) optimizeWrite	Should files be optimized when writing?
(int) readLock	Should records automatically be locked before they are read?
(int) readOnly	Should files be opened in read only mode ?
(int) safety	Should file creation functions fail if the file already exists?
(int) singleOpen	May a single data file be opened more than once by the same application?

Except for **CODE4.createTemp, CODE4.fileFlush, CODE4.lockEnforce, CODE4.readOnly** , **CODE4.readLock** and **CODE4.errOff** the above flags are all initialized to true (non-zero).

In addition to the **CODE4** members, there are CodeBase functions that perform various tasks. For example, there are functions that initialized the **CODE4** structure, lock files, and in the client-server configuration, the functions also perform all operations necessary to connect to and interface with the server.

```
/*ex0.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( void )
{
    CODE4 settings ;
    DATA4 *dataFile ;

     code4init( &settings ) ;
    settings.autoOpen = 0 ;
     settings.memSizeBuffer   = 0x2000 ; /*8192*/
    settings.errDefaultUnique = r4unique ;

    dataFile =d4open( &settings, "INFO" ) ;

    /*this is equivalent to calling code4exitTest( )*/

    if( settings.errorCode < 0 )  code4exit( &settings ) ;
    /*... */
    code4initUndo( &settings ) ;
}
```

CODE4 Structure Variables
CODE4.accessMode

Usage: int CODE4.accessMode = OPEN4DENY_NONE

Description: This member variable determines the file access mode for all files that are either opened or created. Access to these files in **OPEN4DENY_RW** mode (see below) is quicker, since the file is used exclusively, thus making record and file locks unnecessary. The performance increase is even greater when optimizations are enabled.

It is recommended that **CODE4.accessMode** be set to **OPEN4DENY_RW** whenever creating, packing, zapping, compressing, or reindexing files. Failure to do so can result in errors being generated by other applications accessing the files while the given process is executing.

CODE4.accessMode is set to **OPEN4DENY_NONE** by default.

The possible values for this member variable are:

OPEN4DENY_NONE	Open the data files in shared mode. Other users have read and write access.
OPEN4DENY_WRITE	Open the data 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 data files exclusively. Other users may not open the files.

CODE4.accessMode specifies what OTHER users will be able to do with the file. **CODE4.readOnly** specifies what the CURRENT user will be able to do with the file.

When a file is opened (which also occurs upon creation), its read/write permission attributes are obtained from

CODE4.accessMode. If this member is altered, the change will only affect files that are opened afterwards. Thus, to modify the read/write permission of an open file, it must first be closed and then reopened.

Client-Server: In the client-server configuration, the **CODE4.accessMode** determines the client access to the file, but **CODE4.accessMode** does not necessarily reflect how the file is physically opened. Refer to the openMode setting of the server configuration file for more information. The access to files can also be specified by the openMode setting in a catalog file.

See Also: **CODE4.optimize, code4optStart, CODE4.readOnly,** catalog files in the "Security" chapter of the users guide

```
/*ex1.c*/
#include "d4all.h"

#ifdef __TURBOC__
   extern unsigned _stklen = 10000 ;  // for all Borland compilers
#endif

void main( )
{
   CODE4 codeBase ;
   DATA4 *newDataFile ;
   FIELD4INFO fieldInfo [] =
   {
         {"NAME", 'C', 20, 0},
         {"AGE", 'N', 3, 0},
         {"BIRTHDATE", 'D', 8, 0},
         {0, 0, 0, 0},
   }

   code4init( &codeBase ) ;
   codeBase.accessMode = OPEN4DENY_RW ;/*prevents other applications from having
                              read and write access to any files opened
                     subsequently*/
   codeBase.safety = 0 ;  /* Ensure the create overwrites any existing file*/
   newDataFile = d4create( &codeBase, "NEWDBF", fieldInfo, 0 );

   d4close( newDataFile ) ;

                                          /* open in shared mode*/
   codeBase.accessMode = OPEN4DENY_NONE ;
   newDataFile = d4open( &codeBase, "NEWDBF" ) ;

   /* ... some other code ...  */

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

CODE4.autoOpen

Usage: int CODE4.autoOpen = 1

Description: When **CODE4.autoOpen** is true (non-zero) a production index file is automatically opened at the same time the data file is opened. When the **S4CLIPPER** compilation switch is set, an attempt is made to open a corresponding .**CGP** (group) file when a data file is opened. If there is a .**CGP** file of the same name as the data file, CodeBase assumes that this file contains a list of tags to be opened.

Set **CODE4.autoOpen** to false (zero) to specify that index files should not be automatically opened.

The default setting is true (non-zero).

Client-Server: Setting the **CODE4.autoOpen** to false (zero) will not prevent the server from automatically opening production index files, but it will prevent the client from having access to the production index, thus saving memory.

See Also: Group Files in User's Guide

```
/*ex2.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    DATA4 *info, *data;
     INDEX4 *infoIndex;

     code4init( &cb ) ;
     cb.autoOpen = 0 ;              /* Do not automatically open production index file*/
     cb.errOpen = 0 ;
     info = d4open( &cb, "INFO" ) ;

     infoIndex = i4open( info, "INFO" ) ;

    if( cb.errorCode < 0 )
       printf( "Production index file is not opened") ;

                               /* DATA.DBF has a production index.  Open it*/
    cb.autoOpen = 1 ;
    data = d4open( &cb, "DATA" ) ;

    /*  Some other code */

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

CODE4.codePage

Usage: int CODE4.codePage = cp0

Description: CodeBase 6.0 supports the use of code pages used in Visual FoxPro 3.0. A code page is a set of characters specific to a language of a hardware platform. Accented characters are not represented by the same ASCII values across platforms and code pages. In addition, some characters available in one code page are not available in another.

This member controls which code page to use when creating FoxPro database files (.DBF) and index files (.CDX). This member will only be used when creating a new database file. An attempt to open a database file that is using a different code page will fail if support for that code page has not been added to the CodeBase library using conditional compilation switches.

Note

This **CODE4** member will only have an effect when the CodeBase library has been built with the **S4FOX** switch defined.

CODE4.codePage may be assigned one of the following values:

cp1252 This value supports a Windows ANSI code page used in Visual FoxPro 3.0.

cp437 This value supports a U.S. MS-DOS code page used in Visual FoxPro 3.0.

cp0 This value supports FoxPro 2.x file formats which do not use a code page. This is the default setting.

See Also: **S4FOX, CODE4.collatingSequence**

CODE4.collatingSequence

Usage: int CODE4.collatingSequence = sort4machine

Description: CodeBase 6.0 supports the use of collation sequences used in Visual FoxPro 3.0. A collation sequence controls the sorting of character fields in indexing and sorting operations. The addition of collation sequences allows for the correct sorting of international languages. Note: not all collation sequences are available with all code pages.

This member controls which collating sequence to use when creating FoxPro index files (.CDX). This member will only be used when creating a new index file. An attempt to open an index file that is using a different collating sequence will fail if support for that collation sequence has not been added to the CodeBase library using conditional compilation switches.

This **CODE4** member will only have an effect when the CodeBase library has been built with the **S4FOX** switch defined.

CODE4.collatingSequence may be assigned one of the following values:

sort4machine This value supports the collation sequence used by FoxPro 2.x file formats. This is the default value.

sort4general This value supports the collation sequence for English, French, German, Modern Spanish, Portuguese and other Western European languages used by Visual FoxPro 3.0.

See Also: **S4FOX, CODE4.codePage**

```
/*ex2a.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers */

FIELD4INFO fields[] =
{
    { "NAME_FLD", 'C', 20, 0 },
    { "AGE_FLD", 'N', 3, 0 },
    { 0,0,0,0 }
};

void main( void )
{
    CODE4 cb ;
    DATA4  *data

     code4init( &cb ) ;
     cb.codePage = cp1252 ;
     cb.collatingSequence = sort4general ;

     .       /* All database created will be stamped as using code page 1252. */
     .       /* All index files created will be sorted according    to the general */
     .       /*  collating sequence. */

    code4initUndo( &cb ) ;
}
```

CODE4.createTemp

Usage: int CODE4.createTemp = 0

Description: This true/false flag specifies whether CodeBase should create temporary files. When **CODE4.createTemp** is set to true (non-zero), any file that is created by **d4create** or **file4create** will be regarded as temporary by CodeBase and thus deleted once it is closed.

This is used for creating temporary files that are only required once.

The default setting is false (zero).

Note

When a file is created, CodeBase checks the **CODE4.createTemp** flag to determine whether the file should be a temporary one. If this member is altered, the change will only affect files that are created afterwards.

See Also: **d4create, file4create**

CODE4.errCreate

Usage: int CODE4.errCreate = 1

Description: This true/false flag specifies whether CodeBase functions should generate an error message if a file cannot be created. This flag is initialized to true (non-zero).

See Also: **error4**

```
/*ex3.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /*for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    FILE4 *temp ;

     code4init( &cb ) ;
    cb.errCreate = 0 ;

    if( file4create( temp, &cb, "NEWFILE.TXT", 1 ) == r4noCreate)
    /* File exists. Try in the temp directory.*/
         file4create( temp, &cb, "C:\\TEMP\\NEWFILE.TXT", 1 ) ;/*TEMP must exist*/

    if( cb.errorCode < 0 )
       code4exit( &cb ) ;

     /* Some other code*/
     code4initUndo( &cb ) ;
}
```

CODE4.errDefaultUnique

Usage: int CODE4.errDefaultUnique = r4uniqueContinue

Description **CODE4.errDefaultUnique** is set to **r4uniqueContinue** by default. **CODE4.errDefaultUnique** may be set to: **r4uniqueContinue**, **e4unique** or **r4unique**. Only unique tags are affected by this setting.

See Also: **i4create**

```
/*ex4.c*/
#include "d4all.h"
```

```
void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
    int rc ;

    /* Do not add duplicate records to unique tags or the data file and
       return r4unique when attempted.*/

    cb.errDefaultUnique = r4unique ;

    data =  d4open( &cb, "INFO" ) ;
    d4top( data ) ;
    d4appendStart( data, 0 ) ;

    rc = d4append( data ) ; /* append a duplicate copy of the top record*/

    if( rc == r4unique )
        printf( "Attempt to add a duplicate record failed.\n") ;
    else
    {
         printf( "Attempt to add a duplicate record succeeded\n") ;
         printf( "Record in both data and index file\n") ;
     }
    d4close( data ) ;
    code4initUndo( &cb ) ;
}
```

CODE4.errExpr

Usage: int CODE4.errExpr = 1

Description: This true/false flag specifies whether **expr4parse** should generate an error message if an invalid expression is specified. This is useful to turn off if a user is providing the expression to be evaluated. This flag is initialized to true (non-zero).

See Also: **expr4parse**

```
/*ex5.c*/
#include "d4all.h"
extern unsigned _stklen = 15000; /* for all Borland compilers */

void main( void )
{
    CODE4 code ;
    DATA4  *data;
    EXPR4 *expression ;
    char badE xpr[ ] = "NAME = 5" ;

    code4init( &code ) ;
    data = d4open( &code, "INFO" );
    expression = expr4parse( data, badExpr ) ;
    printf( "\nAn error message just displayed \n" ) ;

    code.errorCode = code.errExpr = 0 ;
    expression = expr4parse( data, badExpr ) ;
    printf( "No error message displayed." ) ;
    expr4free( expression ) ;
    code4initUndo( &code ) ;
}
```

CODE4.errFieldName

Usage: int CODE4.errFieldName = 1

Description: This true/false flag specifies whether **d4field** and **d4fieldNumber** should generate an error message if the field name, specified as a parameter, does not exist.

This flag is initialized to true (non-zero).

See Also: **d4fieldNumber**, **d4field**

```
/*ex6.c*/
#include "d4all.h"

void main( void )
{
    CODE4 code ;
    DATA4  *data;
    FIELD4  field;

    char badField[] = "notAField" ;

    code4init( &code ) ;
    data = d4open( &code, "INFO" ) ;
    field = d4field( data, badField ) ;

    printf( "\nAn erro r message just displayed\n" ) ;

    code.errorCode = code.errFieldName = 0 ;
    field = d4field( data, badField ) ;

    printf( "No error message displayed.\n") ;
    code4initUndo( &code ) ;
}
```

CODE4.errGo

Usage: int CODE4.errGo = 1

Description: This true/false flag specifies whether **d4go** should generate an error message when attempting to go to a non-existent record.

This flag is initialized to true (non-zero).

See Also: **d4go**

```
/*ex7.c*/
#include "d4all.hpp"
extern unsigned _stklen = 10000; /* for all Borland compilers*/

void main( )
{
   CODE4 cb ;
   DATA4 *data

   code4init( &cb ) ;
   data = d4open( &cb, "INFO" ) ;
   d4go( data, d4recCount( data ) + 1 ) ;

   printf( "\nAn error message was displayed\n" ) ;

   cb.errorCode = cb.errGo = 0 ;
   d4go(data, d4recCount( data ) + 1 ) ;
   printf( "No error message was displayed\n" ) ;

   code4initUndo( &cb ) ;
}
```

CODE4.errOff

Usage: int CODE4.errOff = 0

Description: This switch disables the CodeBase standard error functions from displaying error messages. When **CODE4.errOff** is false (zero) all error messages are displayed. If true (non-zero), no error messages are displayed.

This switch also disables the 'pause' created when **E4PAUSE** is defined.

The default setting is false (zero).

See Also: **error4, E4PAUSE**

CODE4.errOpen

Usage: int CODE4.errOpen = 1

Description: This true/false flag specifies whether CodeBase functions should generate an error message if a data file cannot be opened.

This flag only applies to the physical act of opening the file. If the file is corrupt or is not a data file, an error message may appear.

This flag is initialized to true (non-zero).

See Also: **d4open, code4logOpen, i4open, file4open**

```
/*ex8.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers */

FIELD4INFO fields[] =
{
    { "NAME_FLD", 'C', 20, 0 },
    { "AGE_FLD", 'N', 3, 0 },
    { 0,0,0,0 }
};

void main( void )
{
    CODE4 cb ;
    DATA4  *data

     code4init( &cb ) ;
     cb.errOpen = 0 ;
    /* no error message is displayed if NO_FILE does not exist*/
     data = d4open( &cb, "NO_FILE" ) ;

   if(data == NULL )
   {
       /* Data file does not exist */
       cb.safety = 0 ;
       data = d4create( &cb, "NO_FILE", fields, 0 ) ;
       if( data == NULL )
         printf( "Could not create NO_FILE" ) ;
    }

    code4initUndo( &cb ) ;
}
```

CODE4.errorCode

Usage: int CODE4.errorCode = 0

Description: This is the current error code. A zero or positive value means that there is no error. Any value less than zero represents an error. Occasionally, a function may set this member to a positive value, indicating a non-error condition.

Any returned error code will correspond to one of the error constants in the header file "D4DATA.H." These constants are documented in "Appendix A: Error Codes".

This variable is initialized to zero.

See Also: **error4set**

CODE4.errRelate

Usage: int CODE4.errRelate = 1

Description: This true/false flag specifies whether relation functions **relate4skip**, **relate4doAll**, **relate4doOne**, **relate4top**, and **relate4bottom** should generate an error message if a slave record cannot be found during a lookup. This is only applicable for exact match and scan relations whose error action is set to **r4terminate**. If this flag is false (zero), the error message is suppressed and these functions return a value of **r4terminate**. This flag is initialized to true (non-zero).

See Also: **relate4skip**, **relate4doAll**, **relate4doOne**, **relate4top**, **relate4bottom**

CODE4.errSkip

Usage: int CODE4.errSkip = 1

Description: This true/false flag specifies whether **d4skip** should generate an error message when it attempts to skip from a non-existent record. If **CODE4.errSkip** is true (non-zero), an error message is generated.

This flag is initialized to true (non-zero).

See Also: **d4skip**

CODE4.errTagName

Usage: int CODE4.errTagName = 1

Description: This true/false flag specifies whether **d4tag** should generate an error message when the specified tag name is not located.

This flag is initialized to true (non-zero).

See Also: **d4tag**

CODE4.fileFlush

Usage: int CODE4.fileFlush = 0

Description: This true/false flag specifies whether CodeBase should perform a file flush every time a write to a file is performed. Normally, this functionality is not required, since the standard C **write()** function physically writes to disk.

Some operating systems and disk caching software, do not write to disk automatically, but rather the buffer waits until it is convenient to write to disk. Setting **CODE4.fileFlush** to true (non-zero) forces the operating system to write to disk by calling **file4flush**.

The default setting is false (zero).

Client-Server: This setting does not apply to files opened by the server. However, **CODE4.fileFlush** does apply to files opened by client applications.

See Also: **file4flush**

CODE4.hInst

Usage: HANDLE CODE4.hInst

Description: This member is only used in Microsoft Windows programs. It is the "instance handle" of a Microsoft Windows application. The application should assign the "instance handle" supplied by WinMain to **CODE4.hInst** just after **code4init** is called. **CODE4.hInst** is used by CodeBase function **sort4assignCmp** when the CodeBase DLL is used.

See Also: **CODE4.hWnd**, **sort4assignCmp**

```
/*ex9.c*/
#include <windows.h>
#include "d4all.h"

extern unsigned _stklen = 20000;

long FAR PASCAL WndProc( HWND, UINT, WPARAM, LPARAM );

CODE4 cb ;
DATA4 *data ;

int PASCAL WinMain( HINSTANCE hInstance, HANDLE hPrevInstance, LPSTR lpszCmdParam, int nCmdShow )
{
   static char szAppName[] = "testapp";
   HWND hwnd;
   MSG msg;
   WNDCLASS wc;

   if ( ! hPrevInstance )
   {
      wc.style = CS_HREDRAW | CS_VREDRAW;
      wc.lpfnWndProc = WndProc;
      wc.cbClsExtra = 0;
      wc.cbWndExtra = 0;
      wc.hInstance = hInstance;
      wc.hIcon = LoadIcon( NULL, IDI_APPLICATION );
      wc.hCursor = LoadCursor( NULL, IDC_ARROW );
      wc.hbrBackground = GetStockObject( WHITE_BRUSH );
      wc.lpszMenuName = NULL;
      wc.lpszClassName = szAppName;

      RegisterClass( &wc );
   }

   hwnd = CreateWindow( szAppName,
                        "CodeBase++ Test Program",
                        WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, CW_USEDEFAULT,
                        NULL, NULL, hInstance,
                        NULL );

   ShowWindow( hwnd, nCmdShow );
   UpdateWindow( hwnd );

   code4init( &cb );
   cb.hWnd = hwnd;
   cb.hInst = hInstance;
   cb.autoOpen = 0;

   data = d4open( &cb, "INFO" );

   /* Cause a CodeBase message box to appear*/
   d4go( data, d4recCount( data ) + 1 ) ;

   return TRUE ;
}

long FAR PASCAL WndProc( HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam )
{
   return( DefWindowProc(hwnd, msg, wParam, lParam) );
}
```

CODE4.hWnd

Usage: HANDLE CODE4.hWnd

Description: Microsoft Windows applications can assign a window handle to **CODE4.hWnd** just after **code4init** is called. This member variable is used in the report functions under Microsoft Windows. If reporting capabilities are not used in an application, it is unnecessary to set **CODE4.hWnd**.

See Also: **CODE4.hInst**, CodeBase Overview; Windows Programming.

Example: See **CODE4.hInst**

CODE4.lockAttempts

Usage: int CODE4.lockAttempts = WAIT4EVER

Description: **CODE4.lockAttempts** defines the number of times CodeBase will try any given lock attempt. This includes group locks set with **code4lock**. If **CODE4.lockAttempts** is **WAIT4EVER**, CodeBase retries indefinitely until it succeeds. Unfortunately, using this setting can result in dead lock under some circumstances.

The default setting is **WAIT4EVER**.

Valid settings for **CODE4.lockAttempts** is **WAIT4EVER** (-1) or any value greater than or equal to 1. Any other value is undefined.

```
/*ex10.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers */

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;

    code4init( &cb ) ;
    cb.readLock = 1 ;
    cb.lockAttempts = 3 ;

    data = d4open( &cb, "INFO" ) ;
    if( d4top( data ) == r4locked )
    {
       printf( "Top record locked by another user\n") ;
       printf( "Lock attempted %d time(s)\n", cb.lockAttempts );
    }
    code4initUndo( &cb ) ;
}
```

See Also: **code4lock, CODE4.lockAttemptsSingle, code4unlockAuto**

CODE4.lockAttemptsSingle

Usage: int CODE4.lockAttemptsSingle = 1

Description: **CODE4.lockAttemptsSingle** defines the number of times CodeBase will try any individual lock when performing a set of locks with **code4lock**. If the individual lock is not successful after **CODE4.lockAttemptsSingle** attempts, all successful locks in the group are removed. **code4lock** may then try again, depending on the value of **CODE4.lockAttempts**.

The default value is 1 attempt.

See Also: **code4lock, CODE4.lockAttempts**

CODE4.lockDelay

Usage: unsigned int CODE4.lockDelay = 100

Description: This member variable is used to determine how long CodeBase should wait between attempts to perform a lock. **CODE4.lockDelay** is measured in 100ths of a second (i.e. the default setting of 100 means that a lock attempt is made once a second).

Setting this member variable to a smaller setting will cause CodeBase to try the lock more often, thus increasing network traffic (in a network setting) and potentially slowing down overall network performance.

See Also: **CODE4.lockAttempts, code4lock, d4lock**

CODE4.lockEnforce

Usage: int CODE4.lockEnforce = 0

Description: This true/false flag can be used to ensure that an application has explicitly locked a record prior to an attempt to modify it with a field function or the following data file functions: **d4blank**, **d4changed**, **d4delete** or **d4recall.** If **CODE4.lockEnforce** is set to true (non-zero) and an attempt is made to modify an unlocked record, the modification is aborted and an **e4lock** error is returned.

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.

The default setting is false (zero).

See Also: **CODE4.readLock, code4unlockAuto, CODE4.accessMode**

CODE4.log

Usage: int CODE4.log = LOG4ON

Description: The modifications that occur while an application is running can be recorded in a log file automatically. This setting specifies whether the automatic logging will occur.

Note

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 changes to the data files are always recorded in a log file automatically. The logging may NOT be turned off for the current data file by calling **d4log**.

LOG4ON The changes to all the data files are recorded in a log file automatically but the logging may be turned off and on for the current data file by calling **d4log**. This is the default value for **CODE4.log**.

LOG4TRANS Only the changes that occur during a transaction are recorded in a log file automatically. Any changes made to the current data file outside the scope of a transaction may be recorded in a log file if the logging is turned on by calling **d4log**.

Note

When a temporary file is opened, it is opened as though the **CODE4.log** variable is set to **LOG4TRANS**, regardless of the **CODE4.log** setting.

Client-Server: This setting may or may not have an effect in the client-server configuration. Refer to the server configuration file documentation and the catalog file documentation for details.

See Also: **d4log**

CODE4.memExpandBlock

Usage: int CODE4.memExpandBlock = 10

Description: When an index file is opened, it allocates a pool of memory blocks for its use. Extra memory blocks are allocated when the initial memory pool is fully utilized. **CODE4.memExpandBlock** is the number of extra blocks allocated. The default value is 10.

If **CODE4.memExpandBlock** is changed, it is best to change it before any index file is opened. This lets CodeBase manage its memory efficiently.

Index files with the same block size can share from the same memory block pool.

See Also: **CODE4.memSizeBlock, CODE4.memStartBlock**

CODE4.memExpandData

Usage: int CODE4.memExpandData = 5

Description: When a data file is opened, a block of memory is allocated by CodeBase to contain the associated **DATA4** structure.

CODE4.memExpandData is the number of **DATA4** structures to allocate when the initial pool of memory is fully utilized.

The default value is 5.

Note

In operating systems that support hardware moves of memory, such as Windows and OS/2, this value may be set to a lower value, since the operating system attempts to prevent fragmented memory.

See Also: Memory Functions, **CODE4.memStartData**

CODE4.memExpandIndex

Usage: int CODE4.memExpandIndex = 5

Description: This is identical to **CODE4.memExpandData** except that the memory structure in question is the **INDEX4** structure used for index files.

See Also: **CODE4.memExpandData, CODE4.memStartIndex**

CODE4.memExpandLock

Usage: int CODE4.memExpandLock = 10

Description: If more than **CODE4.memStartLock** group locks are established at any one time, CodeBase allocates memory for **CODE4.memExpandLock** more group locks.

See Also: **CODE4.memStartLock**

CODE4.memExpandTag

Usage: int CODE4.memExpandTag = 5

Description: This is identical to **CODE4.memExpandData** except that the memory in question is for the **TAG4** structures used for the index tag files.

See Also: **CODE4.memExpandData, CODE4.memStartTag**

CODE4.memSizeBlock

Usage: unsigned CODE4.memSizeBlock = 1024

Description: A "block" is the number of continuous bytes written or read in a single disk operation. **CODE4.memSizeBlock** is used by memory optimization routines to determine the size of memory blocks. dBASE IV MDX index files are designed to be used with variable block sizes. The size of an index file block is determined when the index is created. CodeBase uses the **CODE4.memSizeBlock** setting when creating new dBASE IV MDX index files. The block size must be a multiple of 512; the maximum block size is 63*512 or 32256 bytes. Memory optimization works most efficiently when all index files use the same block size.

Note

FoxPro CDX block sizes are fixed at 512 bytes. Clipper NTX index file block sizes are fixed at 1024 bytes.

CODE4.memSizeBuffer

Usage: unsigned CODE4.memSizeBuffer = 32768

Description: Packing or zapping a data file is much faster when two large memory buffers can be allocated: one for reading data and one for writing data. This is the size of the two buffers, in bytes, that **d4pack** and **d4zap** initially attempt to allocate. In addition, this is the size of each memory optimization buffer when memory optimizations are being used.

This variable is initialized to 32,768 and should be a multiple of 1024. It also should be a multiple of **CODE4.memSizeBlock**.

Note

If CodeBase does not succeed in allocating this buffer size, smaller buffer sizes are tried. The buffer sizes will decrease until **CODE4.memSizeBlock** is reached.

See Also: User's Guide Optimizations Chapter

CODE4.memSizeMemo

Usage: unsigned CODE4.memSizeMemo = 512

Description: When a dBASE IV or a FoxPro memo file is created, the memo file can have its own 'block size'. **CODE4.memSizeMemo** specifies the block size of the memo file. The block size is the unit by which extra disk space is allocated for a memo entry. For example, if the block size is 1024, then every memo entry uses at least 1024 bytes of disk space. If more than 1024 bytes of disk space is required, the memo entry would use some multiple of 1024 bytes of disk space.

Clipper memo file block sizes are fixed at 512. dBASE IV memo file block sizes must be a multiple of 512 (to a maximum of 32256 bytes) and FoxPro memo block sizes can be any value between 33 and 16384.

By default, **CODE4.memSizeMemo** is 512. If an illegal value is specified, CodeBase rounds up to the closest legal value.

Each block used contains an overhead of 8 bytes. Consequently, if a block size is 512, only 504 bytes are actually available for the memo entries.

CODE4.memSizeMemoExpr

Usage: unsigned CODE4.memSizeMemoExpr = 1024

Description: This member is used by the expression functions. If a memo field is longer than **CODE4.memSizeMemoExpr**, the excess is ignored by the expression evaluation functions. For example, **expr4vary** ignores all the memo field's information that is over **CODE4.memSizeMemoExpr**. In addition, if the memo field's length is less than **CODE4.memSizeMemoExpr**, then **expr4vary** pads the result with null characters. This effect is the same as when dealing with trimmed fields.

expr4len assumes that the length of the memo field is exactly **CODE4.memSizeMemoExpr**.

CODE4.memSizeMemoExpr is initially set to 1024.

See Also: Expression functions.

CODE4.memSizeSortBuffer

Usage: unsigned CODE4.memSizeSortBuffer = 4096

Description: When sorting large amounts of information using the sort module, information often has to be temporarily stored on disk. When this condition occurs, CodeBase sort functions allocate a sequential read/write buffer to speed up this part of the sort operation. **CODE4.memSizeSortBuffer** specifies the size of the buffer in bytes.

Its default value is 4096 and should always be a multiple of 2048. If **CODE4.memSizeSortBuffer** is too large, this read/write buffer can take too much memory from the sorting and as a consequence, increase the amount of spooling that occurs.

See Also: **CODE4.memSizeSortPool,** Sort Functions

CODE4.memSizeSortPool

Usage: unsigned CODE4.memSizeSortPool = 0xF000

Description: Initially the sort module attempts to allocate **CODE4.memSizeSortPool** bytes of memory.

Its default value is 0xF000 which is close to the maximum amount of memory which can be allocated at once under DOS.

Note

The sort module tries values lower than the default if the attempted default allocation fails.

See Also: **CODE4.memSizeSortBuffer,** Sort Functions

CODE4.memStartBlock

Usage: int CODE4.memStartBlock = 10

Description: When an index file is initially opened, it allocates **CODE4.memStartBlock** memory blocks for its use. The initial value for **CODE4.memStartBlock** is 10.

If the index file block size is the same for more than one index file, the files can share the same pool of available memory blocks. Consequently, it is most efficient to set these numbers before opening any index files and then never change them.

Note

In operating systems that support hardware moves of memory, such as Windows and OS/2, this value may be set to a lower value, since the operating system itself attempts to prevent fragmented memory.

See Also: **CODE4.memSizeBlock, CODE4.memExpandBlock**

CODE4.memStartData

Usage: int CODE4.memStartData = 10

Description: When a data file is opened , a block of memory is allocated by CodeBase to contain the **DATA4** structure.

The CodeBase memory functions are used to allocate the **DATA4** structures in groups so as to avoid memory fragmentation that results from many allocations, and to speed up the allocation process.

The first time a data file is opened, memory for **CODE4.memStartData** **DATA4** structures is allocated.

Note

In operating systems that support hardware moves of memory, such as Windows and OS/2, this value may be set to a lower value, since the operating system itself attempts to prevent fragmented memory.

See Also: **CODE4.memExpandData**

CODE4.memStartIndex

Usage: unsigned CODE4.memStartIndex = 10

Description: This is identical to **CODE4.memStartData** except that the memory in question is used for index files. This is the initial number of **INDEX4** structures to allocate.

See Also: **CODE4.memExpandIndex, CODE4.memStartData**

CODE4.memStartLock

Usage: int CODE4.memStartLock = 5

Description: When a group lock function (**d4lockAdd, d4lockAddAppend, d4lockAddFile, d4lockAddAll, relate4lockAdd**) is first called, CodeBase allocates memory to store the lock information. **CODE4.memStartLock** is the number of locks that may be added to the locking queue before **code4lock** is called. When **CODE4.memStartLock** locks are added to the queue, more memory is allocated according to **CODE4.memExpandLock**.

If it is known exactly how many group locks are placed at a single time during the course of an application, fragmentation can be reduced by setting this member variable before placing any group locks.

Note

In operating systems that support hardware moves of memory, such as Windows and OS/2, this value may be set to a lower value, since the operating system itself attempts to prevent fragmented memory.

See Also: **CODE4.memExpandLock, d4lockAdd, d4lockAddAppend, d4lockAddFile, d4lockAddAll, relate4lockAdd, code4lock**

CODE4.memStartMax

Usage: long CODE4.memStartMax

Description: When memory optimization is being used, CodeBase allocates a number of memory buffers in which disk information is stored. This reduces the number of times CodeBase has to access the disk and consequently improves performance.

CODE4.memStartMax is the maximum amount of memory CodeBase uses for its memory optimization. Generally, the more memory CodeBase can use, the faster its potential performance.

However, making **CODE4.memStartMax** too large is not recommended. If most of the memory could not be allocated, the memory optimization will work slightly less efficiently. In addition, some operating environments will simply provide all of the requested memory as "virtual memory" and then automatically swap information back and forth between memory and disk. Having "virtual memory" allocated is counter-productive.

On the other hand, if too little memory can be allocated, the benefits of memory optimization do not warrant the overhead. Consequently, if too little memory can be allocated or if **CODE4.memStartMax** is too small, the function **code4optStart** returns a failure.

CODE4.memStartMax is initialized to 0x50000L (320K) under DOS and 0xF0000L (960K) under Windows, OS/2 and Unix. This value should be modified as dictated by the resources of the operating system and the needs of the application before **code4optStart** is called.

See Also: User's Guide Memory Optimizations Chapter

CODE4.memStartTag

Usage: int CODE4.memStartTag = 10

Description: **CODE4.memStartTag** is identical to **CODE4.memStartData** except that the memory in question is used for tag files. This is the initial number of **TAG4** structures to allocate.

See Also: **CODE4.memExpandTag, CODE4.memStartData**

CODE4.optimize

Usage: int CODE4.optimize = OPT4EXCLUSIVE

Description: This member specifies the initial memory read optimization status to be used when files are opened and created. The default can be overridden afterwards by calling **file4optimize** or **d4optimize**.

Possible choices for **CODE4.optimize** are as follows:

OPT4EXCLUSIVE Read-optimize when files are opened exclusively, when the **S4STAND_ALONE** compilation switch is defined, or

when the DOS read-only attribute is set for the file. Otherwise, do not read optimize. This is the default value.

Note that there is a distinction between a DOS read-only attribute and a network read file permission. A DOS read-only attribute means that no one may write to the file, while a network permission setting for any given file may be different from user to user.

OPT4OFF Do not read optimize.

OPT4ALL Read optimize all files, including those opened/created in shared mode.

Note

Use memory optimization on shared files with caution. When using memory read optimization on shared files, it is possible for inconsistent data to be returned if another application is updating the data file. This means that any data returned, from the memory optimized file, could potentially be out of date.

See Also: **CODE4.accessMode, CODE4.optimizeWrite, d4optimize, file4optimize, code4optStart**

```
/*ex11.c*/

#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers */

void main ()
{
    CODE4 cb ;
    DATA4 *inventory ;
    FIELD4 *minOnHand, *onHand, *stockName;

    int oldOpt, oldExcl ;
    int count ;

    code4init( &cb );

    oldOpt = cb.optimize ; /* save old optimization setting.*/
    oldExcl = cb.accessMode ;
    cb.optimize = OPT4EXCLUSIVE ; /* optimize all new files */
    cb.accessMode = OPEN4DENY_RW ;

    inventory = d4open( cb, "INVENT.DBF") ; /* Read optimized */
    minOnHand = d4field( inventory, "MIN_ON_HND" ) ;
    onHand = d4field( inventory,   "ON_HAND" ) ;
    stockName = d4field( inventory, "ITEM" )  ;

    count = 0 ;
    if( cb.errorCode >= 0 )
    {
       code4optStart( &cb ) ;

       for( d4top( inventory ) ; ! d4eof( inventory) ; d4skip( inventory, 1L) )
          if( f4long(onHand) < f4long( minOnHand ) )
            count++ ;
         code4optSuspend( &cb ) ;
    }
    cb.optimize = oldOpt ;
    cb.accessMode = oldExcl ;
    d4close( inventory) ;
    printf( "%d items need to be restocked", count) ;
    code4initUndo( &cb ) ;
}
```

CODE4.optimizeWrite

Usage: int CODE4.optimizeWrite = OPT4EXCLUSIVE

Description: This member specifies the initial write optimization status to be used when files are opened and created. The default can be overridden afterwards by calling **file4optimizeWrite** or **d4optimizeWrite**. Read optimization must be enabled for write optimization to take effect. In addition, to write optimize shared data, index and memo files, it is important to lock the files. Call **d4lockAll,** or call **d4lockAddAll** followed by **code4lock** to lock the files. This ensures that performance is not degraded through unbuffered writes to index and/or memo files.

Possible choices for this member variable are as follows:

OPT4EXCLUSIVE Write-optimize when files are opened exclusively or when the **S4STAND_ALONE** compilation switch is defined. Otherwise, do not write optimize. This is the default value.

OPT4OFF Do not write optimize.

OPT4ALL Write optimize all files, including those opened/created in shared mode. Shared files must be locked before write optimization takes effect.

WARNING

Use memory optimization on shared files with caution. When doing so, it is possible for inconsistent data to be returned if another application is updating the data file.

Note

Write optimization does not improve performance unless the entire data file is locked over a number of operations. For example, write optimization would be useful when appending many records at once.

See Also: **CODE4.optimize, d4optimizeWrite, code4optStart**

```
/*ex12.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ;

void main ()
{
     CODE4 code ;
     DATA4 *d ;
     FIELD4 *dateField ;

     char today[8];
     int oldLockAttempts, oldOpt, oldOptWrite ;

     code4init( &code ) ;
     oldLockAttempts = code.lockAtt  empts ;
     oldOpt = code.optimize;
     oldOptWrite = code.optimizeWrite ;

    code.lockAttempts = WAIT4EVER ;
    code.optimize = OPT4ALL ;
    code.optimizeWrite = OPT4ALL ;

    d = d4open( &code, "DATEFILE" ) ;
    if( code.errorCode < 0 )
       code4exit( &code ) ;

    d4lockAll( d ) ; /* lock the file for optimizations to take place*/
```

```
    dateField = d4field( d, "DATE" ) ;
    date4today( today );

    code4optStart( &code ) ;
    for( d4top( d ) ; ! d4eof( d ) ; d4skip( d, 1L ) )
       f4assign( dateField, today ) ;
    code4optSuspend( &code ) ;

    d4close( d ) ;
    code.lockAttempts = oldLockAttempts ;
    code.optimize = oldOpt ;
     code.optimizeWrite = oldOptWrite ;
     code4initUndo( &code );
}
```

CODE4.readLock

Usage: int CODE4.readLock = 0

Description: This true/false flag specifies whether functions should automatically lock a data record before reading it. Specifically, this flag applies to functions **d4top**, **d4bottom**, **d4seek**, **d4seekDouble**, **d4seekN**, **d4seekNext**, **d4seekNextDouble**, **d4seekNextN**, **d4skip**, **d4go**, **d4position** and **d4positionSet**.

This flag is initialized to false (zero).

Setting **CODE4.readLock** to true (non-zero) can reduce performance, since locking a record often takes as long as a write to disk. For the best performance, set **CODE4.readLock** to true (non-zero) only when modifying several records one after another.

Note

Note that **CODE4.readLock** does NOT specify whether files should be locked when performing on a relation. If the files are to be locked while performing a relation function, then **relate4lockAdd** and **code4lock** must be called explicitly. If the files are locked in this way, then no other users may modify any of the relation's data files until after **code4unlock** is called. This ensures that relate functions will always return results that are completely up to date.

See Also: **d4lock, file4lock, relate4lockAdd, code4lock, CODE4.lockEnforce, code4unlockAuto, code4unlock**

```
/*ex13.c*/
#include "d4all.h"

CODE4 cb ;
DATA4 *d ;

int retry(  )
{
    char rc ;
    printf("Record locked by another user.\n") ;
    printf("Retry? (Y or N)\n") ;
    rc = getchar( ) ;
    if( rc == 'N' ) return 0 ;
    return 1 ;
}

void modifyRecordValues( )
{
     int rc ;
    char buf[8+2] ;
     FIELD4 *field ;

    cb.readLock = 1 ;
    cb.lockAttempts = 3 ;
```

```
    field = d4field( d, "DATE") ;
    while( ((rc = d4top( d )) == r4locked) && (retry( ) == 1) ) ;
    if( rc == r4locked ) return ;

    while( !d4eof( d ) )
    {
       printf("\nEnter new record value: \n") ;
       fgets(buf, sizeof(buf), stdin) ;
       f4assign( field, buf ) ;

       while( ((rc = d4skip( d, 1L )) == r4locked) && (retry( ) == 1)) ;
       if( rc == r4locked ) return ;
    }
}
void main()
{
    code4init( &cb ) ;
    d = d4open( &cb, "DATEFILE" ) ;
    modifyRecordValues();
    code4initUndo( &cb ) ;
}
```

CODE4.readOnly

Usage: int CODE4.readOnly = 0

Description: This true/false flag specifies whether files are to be opened in read-only mode.

There are two reasons why this switch is used. First, the application may only have read-only permission on the file and any attempt to open it with read and write permissions would fail. Secondly, if the application is designed to only read files and not modify them, then opening in read-only mode would protect against application bugs that may modify the file accidentally.

This flag has no effect on how files are created.

The default setting is false (zero).

Note

There is a distinction between a DOS read-only attribute and a network read file permission. A DOS read-only attribute means that no one may write to the file, while a network permission setting for any given file may be different from user to user.

CodeBase can make optimizations internally if the DOS read-only attribute is set. These optimizations will not be possible if only the network write permission is denied.

Client-Server: Catalog files have an independent setting that determines whether the file has read-only access. This setting overrides that of **CODE4.readOnly**. Therefore, if **CODE4.readOnly** is set to false (zero) and the catalog file readOnly setting for a file is true (non-zero), the file would have read-only access.

```
/*ex14.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    DATA4 *prices ;
```

```
    TAG4 *tag ;

    code4init( &cb );
    cb.readOnly = 1 ;

    /* open a file on a drive without write access*/

    prices = d4open( &cb, "W:\\INFO.DBF" ) ;
    error4exitTest( &cb ) ;

    tag = d4tagDefault( prices ) ;
    d4tagSelect( prices, tag ) ;

    if( d4seek( prices, "SMITH" ) == 0 )
        printf( "SMITH is found\n") ;
    else
        printf( "SMITH is not found\n" ) ;

    code4initUndo( &cb ) ;
}
```

CODE4.safety

Usage: int CODE4.safety = 1

Description: This true/false flag determines if files are protected from being automatically over-written when an attempt is made to re-create them. This flag is initialized to true (non-zero).

Possible settings for **CODE4.safety** are:

Non-Zero — Files are protected from erasure by functions such as **d4create**, **i4create**, and **file4create**. Instead these functions will return **r4noCreate** and possibly generate an error message. In addition, **r4noCreate** will be returned following an attempt to create a file to which the user has neither write nor create access (for instance, if the DOS read-only attribute is set).

0 — Files are automatically erased when an attempt is made to re-create them.

Client-Server: Catalog files have an independent safety setting. This setting overrides that of **CODE4.safety**. Therefore, if the **CODE4.safety** was false (zero) and the catalog safety setting for a file is true (non-zero), then the file will not be overwritten when an attempt to re-create it occurs.

See Also: **CODE4.errCreate**

```
/*ex15.c*/
#include "d4all.h"

CODE4 cb ;
DATA4 *data ;

int createFiles( )
{
    FIELD4INFO fields [] =
    {
        { "NAME", 'C', 20, 0 },
        { "AGE", 'N', 3, 0 },
        { "BIRTHDATE", 'D', 8, 0 },
        { 0, 0, 0, 0 },
    };

    TAG4INFO tags [] =
    {
        { "NAME_TAG", "NAME", 0, 0, 0 },
        { 0, 0, 0, 0, 0 },
```

```
    };

    cb.safety = 0 ; /* Turn off safety -- overwrite files*/

    data = d4create( &cb, "INFO.DBF", fields, tags ) ;

   return cb.errorCode ;
}

void main()
{
    int rc ;

    code4init( &cb );
    rc = createFiles();
    code4initUndo( &cb );
}
```

CODE4.singleOpen

Usage: int CODE4.singleOpen = 1

Description: This true/false flag determines whether data files or index files may be opened more than once by the same application. If **CODE4.singleOpen** is false (zero), an application may open the same file several times.

When **CODE4.singleOpen** is set to true (non-zero), a file may only be opened once. Any attempts to open it again will generate an **e4instance** error.

The default setting is true (non-zero).

See Also: **d4open**

CodeBase Function Reference
code4calcCreate

Usage: EXPR4CALC *code4calcCreate(CODE4 *codebase,
EXPR4 *expr, const char *name)

Description: This function creates a user-defined function, which is recognizable in any other dBASE expression and in the **EXPR4** structure expressions. The function is not accessible outside of the expression evaluation routines of CodeBase.

Calculation names are defined without parentheses. When used in a dBASE expression, however, a set of parentheses -- () -- are appended to the calculation name. There may not be any characters (including spaces) between the calculation's parentheses.

If an expression is too long to be parsed by **expr4parse,** a CodeBase error is generated. **code4calcCreate** can be used to break up an expression into manageable pieces, thus allowing longer expressions than would otherwise be possible.

Parameters:

codebase A pointer to a **CODE4** structure.

expr This is the **EXPR4** structure that specifies what the user-defined function returns.

name This is the name of the user-defined function. *name* can not exceed 19 characters in length.

Returns:

Null Error. The user-defined function was not created.

Not Null A non-zero return indicates success. The **EXPR4CALC** structure pointer is used inside CodeBase.

See Also: **code4calcReset**

```
/*ex16.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    DATA4  *db ;
     EXPR4 *ex, *ex2 ;
     EXPR4CALC *names ;
     char *result;

     code4init( &cb ) ;
     db = d4open( &cb, "DATA" ) ;
    d4top( db ) ;

     ex = expr4parse( db, "TRIM(  LNAME)+', '+TRIM(FNAME)" ) ;
    names = code4calcCreate( &cb, ex, "NAMES" ) ;
     expr4vary(ex, &result) ;
    printf( "%s is the result\n", result ) ;

    ex2 = expr4parse( db, "'HELLO '+NAMES()" ) ; /*no space in dBASE function
                                                    calls.*/
     expr4vary(ex2, &result ) ;
     printf( "%s is the second result\n", result ) ;

     expr4free( ex2 ) ;
    code4calcReset( &cb ) ;
     code4initUndo(&cb ) ;
}
```

code4calcReset

Usage: void code4calcReset(CODE4 *codebase)

Description: All of the user-defined functions created with **code4calcCreate** are removed. Any parsed expressions that reference any of the removed user-defined functions must not be used subsequently.

See Also: **code4calcCreate**

code4close

Usage: int code4close(CODE4 *codebase)

Description: **code4close** closes all open data, index and memo files. Before closing the files, any necessary flushing to disk is done. In addition, the time stamps of the files are updated if the files have been updated. **code4close** is equivalent to calling **d4close** for each and every data file opened. **code4close** has no effect on files opened with **file4open**.

Locking: **code4close** does any locking necessary to accomplish flushing to disk. After any necessary updating is done, **code4close** unlocks everything that has been locked.

Before closing the file, **code4close** attempts to flush the most recent changes to the record buffers. If a flush attempt does not succeed, **code4close** continues and closes the files anyway. Consequently, **code4close** never returns **r4locked** or **r4unique**. If it is important to trap these return codes, consider using **code4flush** before calling **code4close**.

Returns:

r4success Success.

< 0 Error.

See Also: **d4close, code4flush**

```
/*ex17.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void openAFile( CODE4 *cb )
{
     DATA4 *d ;

   /* 'd' falls out of scope.  Data file is still open*/
   d = d4open( cb, "INFO" ) ;
}


void main( void )
{
    CODE4 cb ;
     DATA4 *d ;

     code4init( &cb ) ;
    cb.autoOpen = 0 ;
    openAFile( &cb ) ;

    d = d4open( &cb, "DATA" ) ; /* open a second file */
    printf("Number of records in DATAFILE: %d \n",d4recCount( d ) ) ;

    code4close( &cb ) ; /* INFO and DATAFILE are both closed*/
    code4initUndo( &cb ) ;
}
```

code4connect

Usage: int code4connect(CODE4 *codebase, const char *serverId,
const char *processId,
const char *userId,
const char *password,
const char *protocol)

Description: This function performs all of the operations necessary to connect a client application to the server.

Note

If **code4connect** has not been explicitly called by the application, **d4open** or **d4create** automatically call **code4connect** with the default parameters in order to gain access to the file.

Parameters:

codebase This is a pointer to the **CODE4** structure.

serverId This string specifies the identification string of the server to which the application is attempting to connect. If this parameter is null, the default server is specified by **DEF4SERVER_ID** = "S4SERVER".

processId This string specifies the identification process string of the server to which the application is attempting to connect. If this parameter is null, the default process identification is specified by **DEF4PROCESS_ID** = "23165".

userId The server uses this string to specify the registered name of the user who is attempting to access the server. This parameter may be null if the server does not use name authorizations, or if public access is desired and the default *userId* "PUBLIC" is used.

password The server uses this string as a password to validate the registered name of the user identified by *userId*. This parameter may be null if the server does not use password authorizations.

protocol This string specifies the name of the network communication protocol DLL that the client application will be using. The server should be using a similar 'S4' communications DLL to match the 'C4' DLL that you specify (e.g. S4SPX.DLL corresponds to C4SPX.DLL).

C4SPX.DLL	Use the IPX/SPX communication protocol.
C4SOCK.DLL	Use the Windows Sockets communication protocol.

DOS applications ignore this parameter.

Returns:

r4connected A connection to the server already exists.

r4success The server was successfully located and connected.

< 0 An error occurred. This could indicate a general or network error, which may be caused by the inability to locate or attach to the server. Refer to "Appendix A: Error Codes", for information specific to the error code.

See Also: **code4init**, **code4initUndo**, **d4create**, **d4open**

code4data

Usage: DATA4 *code4data(CODE4 *codebase, const char *alias)

Description: **code4data** tries to find an opened data file that has *alias* as its alias. If successful, **code4data** returns a **DATA4** structure for the specified data file. The returned structure may be used in the same manner as a regularly constructed **DATA4** structure.

Parameters:

codebase A pointer to a **CODE4** structure.

alias This is the alias name to look for.

Returns:

Not Null A pointer to the **DATA4** structure of the data file corresponding to the *alias* parameter.

Null The alias was not found.

```
/*ex18.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void openAFile( CODE4 *cb )
{
     DATA4 *d ;
    /* 'd' falls out of scope.  Data file is still open*/
    d = d4open( cb, "INFO" ) ;
}


void main( void )
{
    CODE4 cb ;
     DATA4 *d ;

     code4init( &cb ) ;
    openAFile( &cb ) ;

    d = code4data( &cb, "INFO" ) ; /* obtain a new DATA4 structure*/
   if( d != NULL )
    {
       printf("INFO has %d records.\n", d4recCount( d )) ;
       d4top( d ) ;
       d4close( code4data( &cb, "INFO") );/*an alternative way to close the file*/
    }
    code4initUndo( &cb ) ;
}
```

code4dateFormat

Usage: const char *code4dateFormat(CODE4 *codeBase)

Description: This function returns the current default date format for the client application. This is equivalent to accessing the **CODE4.date_format** member in CodeBase 5.1. The initial date format is "MM/DD/YY". The date format is used when an expression involving the dBASE functions CTOD() or DTOC() are involved.

See Also: **code4dateFormatSet**

code4dateFormatSet

Usage: int code4dateFormatSet(CODE4 *codeBase, const char *fmt)

Description: This functions sets the current date format for the client application. Note that you cannot simply set the **CODE4.date_format** member, as in previous versions of CodeBase products, since the server will not be aware of this change.

See Also: **code4dateFormat**

code4exit

Usage: void code4exit(CODE4 *codebase)

Description: **code4exit** causes the program to exit immediately. This could be considered an emergency exit function. **code4exit** automatically calls **code4initUndo** to free memory and terminates the server connection. **CODE4.errorCode** is passed as a return code to the operating system.

```
/*ex19.c*/
#include "d4all.h"

void exitToSystem( CODE4 *cb )
{
    printf("\nShutting down application ... \n") ;
    code4close( cb ) ;
    code4exit( cb ) ;
}

void main( )
{
    CODE4 cb ;

    code4init ( &cb ) ;
    exitToSystem( &cb ) ;
}
```

code4flush

Usage: int code4flush(CODE4 *codebase)

Description: This function flushes all CodeBase data, index and memo files to disk. Effectively, this is equivalent to calling **d4flush** for every open data file.

Returns:

r4success Success.

r4locked A required lock attempt did not succeed. All of the files are not flushed.

r4unique The record was not written due to the following circumstances: First, writing the record caused a duplicate key in a unique key tag. Secondly, **t4unique** returned **r4unique** for the tag.

< 0 Error.

Locking: If changes have been made to any field, the record is locked. The index files and append bytes may also be locked during updates. After the **code4flush** is finished, only the current record remains locked for each data file.

See Also: **d4flush**

code4indexExtension

Usage: const char *code4indexExtension(CODE4 *codebase)

Description: This function returns the file extension that corresponds to the index format being used.

Returns:

Not Null This is a pointer to a character string that contains the file extension that corresponds to the index format.

Null If CodeBase does not know the index format, null is returned. Null is also returned when the **S4OFF_INDEX** switch is defined.

Client-Server: The server will return the file extension if it can be determined. Null is returned if the application is not connected to the server.

See Also: **S4OFF_INDEX**

code4init

Usage: int code4init(CODE4 *codebase)

Description: This function is used to initialize the **CODE4** structure. One of these structures should be declared for every application.

Initialization of the **CODE4** is necessary before calling most CodeBase functions, therefore **code4init** must be called before any other CodeBase functions. Normally, **code4init** should be called only once from your application.

When **code4initUndo** is called, **code4init** must be called to re-initialize the **CODE4** structure prior to calling any CodeBase function.

Parameters:

codebase A pointer to a **CODE4** structure.

Returns:

r4success The **CODE4** structure was successfully initialized.

< 0 An error has occurred. This is usually due to an error in the configuration of the application (e.g. the stack is too low, etc.).

See Also: **code4initUndo**

code4initUndo

Usage: int code4initUndo(CODE4 *codebase)

Description: This function un-initializes CodeBase. All data, memo, and index files are flushed and closed, and any memory associated with the files is freed back to the CodeBase memory allocation pool. Furthermore, in the client-server configuration, the server connection are terminated. Once **code4initUndo** is called, no CodeBase function should be called unless **code4init** is first called.

WARNING

If **code4initUndo** is called during a transaction, the transaction is automatically rolled back. Therefore, if a transaction is to be committed, then call **code4tranCommit** before **code4initUndo.**

Returns:

r4success Success.

< 0 Error. See **CODE4.errorCode** for the exact error.

Locking: Locking occurs on files that require flushing. **code4initUndo** makes sure that any locks are unlocked when its finished.

See Also: **code4init**

code4lock

Usage: int code4lock(CODE4 *codebase)

Description: This function performs a lock of a group of records, files, and/or append bytes. A group lock functions as if it were a single lock on a single record, however many interdependent records and files, etc. may be locked.

The entire lock group is either in a state of having all locks held, or no locks held. If any of the locks fail, all successful locks are removed and an error is returned. That is, if all of the locks were successfully performed, but the last lock failed, all of the successful locks would be removed and **code4lock** would report **r4locked**.

This high-level approach to locking minimizes the possibility of deadlock, while giving maximum flexibility.

The process involved in performing a group lock is as follows:

- Set the **CODE4.lockAttempts** and **CODE4.lockAttemptsSingle** to desired values.
- Queue one or more locks with **d4lockAdd**, **d4lockAddAll**, **d4lockAddFile**, **d4lockAddAppend** and/or **relate4lockAdd.**
- Clear the queued locks, if desired, with **code4lockClear** and begin the process again, or
- Attempt to place the queued locks with a single call to **code4lock**. The return code from this function indicates whether or not the locks were successful.

code4lock automatically clears the queue of locks once the locks are successfully performed.

Returns:

r4success	All locks placed in the queue where successfully performed. The queue of locks is emptied.
r4locked	A required lock attempt did not succeed and as a result no locks were placed. The queue of lock entries are maintained for a further attempt.
< 0	Error. The queue of lock entries are maintained for a further attempt.

See Also: **code4unlock, code4unlockAuto, code4lockUserId, code4lockNetworkId, code4lockItem, code4lockFileName, CODE4.lockAttempts, CODE4.lockAttemptsSingle, d4lockAdd, d4lockAddFile, d4lockAddAppend, relate4lockAdd**

```
/*ex20.c*/
#include "d4all.h"

void main( void )
{
   CODE4 cb ;
   DATA4  *data1, *data2;
   long numRec;

   code4init( &cb ) ;
```

```
   data1 = d4open( &cb, "DATA1" ) ;
   data2 = d4open( &cb, "DATA2" ) ;

   d4top( data1 ) ;  d4top( data2 ) ;

   d4lockAddFile( data1 ) ;
   d4lockAddAppend( data2 ) ;

   numRec = d4recCount( data2 ) ;
   d4lockAdd( data2, numRec ) ;
   d4lockAdd( data2, numRec-1 ) ;

   if( code4lock( &cb ) == r4success )
      printf( "All locks were successfully performed\n") ;

   code4initUndo( &cb ) ;
}
```

code4lockClear

Usage: void code4lockClear(CODE4 *codebase)

Description: This function removes any group locks previously placed with **d4lockAdd, d4lockAddAll, d4lockAddAppend** , **d4lockAddFile** and/or **relate4lockAdd**. No locking or unlocking is performed by this function, but rather, any queued locks are removed from the queue.

See Also: **code4lock, d4lockAdd, d4lockAddAll, d4lockAddAppend, d4lockAddFile, relate4lockAdd**

code4lockFileName

Usage: const char *code4lockFileName (CODE4 *codebase)

Description: When a locking function (e.g. **code4lock**) returns **r4locked** – indicating that another user has locked the required information – it is possible to identify the file name of the item that was locked. This function returns a character string, which contains the name of the file that has been locked.

Note

This information is available for a limited time (usually until the next CodeBase Call). Therefore the information should be copied if it is required for a prolonged period.

Returns: The name of the file on which the lock failure has occurred is returned. Null is returned when the file name is not available.

See Also: **code4lock, code4lockUserId, d4lock, code4lockItem, code4lockNetworkId**

code4lockItem

Usage: long code4lockItem (CODE4 *codebase)

Description: When a locking function (e.g. **code4lock**) returns **r4locked** – indicating that another user has locked the required information – it is possible to identify the type of item that was locked. This function returns a long integer indicating whether the item locked was a record, a file or append bytes.

Note

This information is available for a limited time (usually until the next CodeBase Call). Therefore the information should be copied if it is required for a prolonged period.

Returns:

> 0 This is the record number of a locked record.

0 Zero indicates that the append bytes were locked.

-1 This indicates that a file was locked.

-2 This indicates that the locked item cannot be identified.

See Also: **code4lock, code4lockUserId, d4lock, code4lockFileName, code4lockNetworkId**

code4lockNetworkId

Usage: const char *code4lockNetworkId(CODE4 *codebase)

Description: When a locking function (e.g. **code4lock**) returns **r4locked** – indicating that another user has locked the required information – it is sometimes possible to retrieve the network-specific identification of the user that placed the offending lock.

This information is available for a limited time (usually until the next CodeBase Call). Therefore the information should be copied if it is required for a prolonged period.

Returns:

Null The identification for the user that placed the lock was unavailable. This could be due to a limitation in the network protocol being used.

NULL is also returned if the most recent lock attempt succeeded.

Not Null A null-terminated string containing the network-specific identification of the user that placed the lock.

See Also: **code4lock, code4lockUserId, d4lock, code4lockItem, code4lockFileName**

code4lockUserId

Usage: const char *code4lockUserId(CODE4 *codebase)

Description: When a locking function (e.g. **code4lock**) returns **r4locked** – indicating that another user has locked the required information – it is sometimes possible to retrieve the user name of the person who placed the offending lock.

code4lockUserId returns the name of the user holding the lock as registered with the server in the *userId* parameter of **code4connect**.

This information is available for a limited time (usually until the next CodeBase Call). Therefore the information should be copied if it is required for a prolonged period.

Returns:

Null Null may be returned for one of the following reasons: the name of the person holding the lock was not registered, the most recent lock attempt

succeeded, the application is in the stand alone configuration, or the user id can not be determined.

Not Null A null-terminated string containing the name of the person holding the lock.

See Also: **code4lock, code4lockNetworkId, d4lock, code4lockItem, code4lockFileName, code4connect**

code4logCreate

Usage: int code4logCreate(CODE4 *codebase, const char *name, const char *userId)

Description: This function manually creates a CodeBase log file.

If a log file does not exist in the stand-alone configuration, it must be explicitly created by calling **code4logCreate** before calling **d4create** or **d4open**.

If a log file already exists and the logging status is on, then **d4open** (**d4create** or **code4tranStart**) will try to open a log file using **code4logOpen**.

WARNING

When **code4logCreate** is called, **CODE4.safety** is used to determine what to do if the specified log file already exists.

Parameters:

name This is the name of the log file that is to be created. If a path is provided in the single user or multi-user configuration, it is used. If not, the file is assumed to be in the current directory. If this parameter is null, then the default name of "C4.LOG" is used as the log file name.

userId For each change made, the user who made the change is also recorded in the log file. The parameter *userId* specifies the user identification associated with the changes. If this parameter is null, "PUBLIC" is used as the default *userId*.

Returns:

r4success Success.

r4logOpen This indicates that a log file has already been opened. **code4logCreate** can not be used to create a new log file while another log file is open.

< 0 An error occurred and the log file was not created.

Client-Server: In the client-server configuration, this function does not apply and **r4success** is always returned**.**

See Also: **code4logFileName**, **code4logOpen**, **code4logOpenOff**, **CODE4.safety**, **d4log**, **d4open, CODE4.log**

code4logFileName

Usage: const char *code4logFileName(CODE4 *codebase)

Description: This functions returns the name of the log file that is currently being used.

Returns:

Not Null The name of the log file that is currently being used.

Null A log file is not being used currently.

Client-Server: This function returns null in the client-server configuration .

See Also: **code4logCreate**, **code4logOpen**, **d4log**

code4logOpen

Usage: int code4logOpen(CODE4 *codebase, const char *name,
const char *userId)

Description: This function manually opens a CodeBase log file.

If a log file already exists and the logging status is on, then **d4open (d4create** or **code4tranStart)** will try to open a log file using **code4logOpen**.

To open a log file explicitly, call **code4logOpen** before calling **d4open** or **d4create.**

Parameters:

name This is the name of the log file that is to be opened. If a path is provided in the single user or multi-user configuration, it is used. If there is no path, the file is assumed to be in the current directory. If this parameter is null, then the default name of "C4.LOG" is used as the log file name.

userId For each change made, the user who made the change is also recorded in the log file. The parameter *userId* specifies the user identification associated with the changes. If this parameter is null, "PUBLIC" is used as the default *userId*.

Returns:

r4success Success.

r4logOpen This indicates that a log file has already been opened. **code4logOpen** can not be used to open a new log file while another log file is open.

< 0 An error occurred and the log file was not opened.

Client-Server: In the client-server configuration, this function does not apply and **r4success** is always returned**.**

See Also: **code4logCreate**, **code4logOpenOff**, **code4logFileName**, **d4log**, **d4open**

code4logOpenOff

Usage: void code4logOpenOff(CODE4 *codebase)

Description: This function instructs **d4open**, **d4create** or **code4tranStart** not to automatically open the log file. When this function is called, no transactions can take place unless the log file has been explicitly opened.

Client-Server: This function does not apply in the client-server configuration.

See Also: **code4logOpen, code4logCreate**

code4optAll

Usage: int code4optAll(CODE4 *codebase)

Description: This function ensures that memory optimization is completely implemented. To do this, **code4optAll** locks and fully read/write optimizes all data, index and memo files. Finally memory optimization is turned on through a call to **code4optStart**.

By calling **code4optAll**, you can get an idea of how fast your application could run when it fully uses memory optimization. Call this function after you have opened all of your files.

Use this function with caution in multi-user applications. Refer to **d4optimize** and **d4optimizeWrite** for more information.

Returns:

r4success Success.

r4locked A required lock failed, so no optimization is implemented.

< 0 Error. The flushing failed when the optimization was disabled or **CODE4.errorCode** contained a negative value.

See Also: **code4optStart, d4lockAll, d4lockAddAll, code4lock, d4optimize, d4optimizeWrite**

code4optStart

Usage: int code4optStart(CODE4 *codebase)

Description: Use this function to initialize CodeBase memory optimization. It is appropriate to call **code4optStart** after files are opened/created or after memory optimization is suspended as a result of a call to **code4optSuspend**. If **code4optSuspend** has been called, **code4optStart** does not necessarily reallocate the same amount of memory for memory optimization. The application could have allocated extra memory, which would make less memory available for the optimization, or the setting of **CODE4.memStartMax** could have changed in the interim. These factors can change the maximum amount of memory **code4optStart** allocates.

Memory optimization can use a substantial amount of memory. Consequently, it is often best to open/create data, index and memo files before calling **code4optStart**. CodeBase uses the function **u4allocFree** when allocating memory in order to open files. This temporarily suspends optimization when memory cannot be allocated. However, it is more efficient to just start memory optimization once.

Returns:

r4success Success. Memory optimization has been implemented.

< 0 Failure. Memory optimization has not been implemented because there is a lack of available memory or the **S4OFF_OPTIMIZE** compiler define switch has been defined. This is not considered an error condition since CodeBase may still function without memory optimizations, therefore **CODE4.errorCode** is not affected.

See Also: **code4optSuspend, d4optimize, file4optimize, CODE4.optimize, CODE4.optimizeWrite, S4OFF_OPTIMIZE**

```
/*ex21.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ;

void main( void )
{
    CODE4 code ;
    DATA4 *dataFile ;
    int delCount, rc ;

    code4init( &code ) ;
    code.accessMode = OPEN4DENY_RW ;

    dataFile = d4open( &code, "INFO" ) ;
    error4exitTest( &code ) ;

    /* initialize optimization with default settings. */
    code4optStart( &code ) ;

    delCount = 0 ;
    for( rc = d4top( dataFile ); rc == r4success ; rc = d4skip( dataFile, 1L ) )
       if( d4deleted( dataFile) )
          delCount++ ;

    printf("%d records are marked for deletion.\n", delCount ) ;
    code4initUndo( &code ) ;
}
```

code4optSuspend

Usage: int code4optSuspend(CODE4 *codebase)

Description: This function suspends CodeBase memory optimization. **code4optSuspend** frees the memory used by memory optimization back to the operating system. This freed memory can then be used by the application.

To restart memory optimization, re-call **code4optStart**. CodeBase remembers which files are memory optimized and how they are memory optimized.

Returns:

r4success Success.

< 0 Error.

Locking: Since files are only write optimized as long as they are locked or opened exclusively, **code4optSuspend** only flushes those write optimized files that are already locked. **code4optSuspend** does not alter the locking status of any files.

See Also: **d4optimize, d4optimizeWrite, CODE4.optimizeWrite, CODE4.optimize**

```
/*ex22.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ;

void main( void )
{
    CODE4 cb ;
    DATA4 *data;

     code4init( &cb );
     data = d4open( &cb, "INFO") ;

     d4lockAll( data ) ;
    d4optimizeWrite( data, OPT4ALL ) ;

    code4optStart( &cb ) ;
                 /* .. some other code    */
    code4optSuspend( &cb ) ; /* flush & free optimization memory.*/

    d4unlock( data ) ; /* let other users make modifications.*/

     /*  ... some other code*/

     d4lockAll( data ) ;
    code4optStart( &cb ) ;
                    /* ... other code */
    code4initUndo( &cb ) ;
}
```

code4timeout

Usage: long code4timeout(CODE4 *codebase)

Description: This function returns the setting of the **CODE4** member variable **CODE4.timeout**, which is used by CodeBase to determine how long it should wait to receive a message from the server, before disconnecting.

See Also: **code4timeoutHook**, **S4TIMEOUT_HOOK, code4timeoutSet**

code4timeoutSet

Usage: void code4timeoutSet(CODE4 *codebase, long setting)

Description: This function sets the value of the **CODE4** member variable **CODE4.timeout**, which is used by CodeBase to determine how long it should wait to receive a message from the server, before disconnecting. **CODE4.timeout** measures time in seconds. A value of **WAIT4EVER** indicates that the application will wait forever for a response from the server.

The default value is **WAIT4EVER** if the **S4TIMEOUT_HOOK** switch is not defined. If the switch is defined, then the default value is 100.

If the **S4TIMEOUT_HOOK** switch is not defined, then CodeBase will generate an error when it times out. If the switch is defined, CodeBase will call the time out hook function (**code4timeoutHook**) when it times out.

See Also: **code4timeoutHook**, **S4TIMEOUT_HOOK, code4timeout**

code4tranCommit

Usage: int code4tranCommit(CODE4 *codebase)

Description: This function commits the active transaction. Changes reflected in the transaction may not be undone with **code4tranRollback**. The changes are stored in the transaction log file, and may be viewed and/or recovered by using a utility program.

All of the data files are flushed before the transaction is committed. If an error occurs during the flushing, the transaction will not be committed.

Note

Depending upon the setting returned by **code4unlockAuto**, performing a **code4tranCommit** may not unlock the records affected by the transaction. It may be necessary to call **code4unlock** after calling **code4tranCommit**.

Returns:

r4success The transaction was successfully committed.

r4locked A required lock attempt did not succeed.

r4unique This value is returned when a record flush causes a duplicate key in a unique tag, and **t4unique** returned **r4unique** for the tag.

< 0 An error occurred during the attempt to commit the changes.

In general **code4tranCommit** does not fail. If **code4tranCommit** does fail, then a critical error has occurred and recovery can NOT be accomplished by calling **code4tranRollback**. Instead, try to recover by calling the utility programs.

Locking: If record buffer flushing is required, the record and index files are locked. See **d4flush** and **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

See Also: **code4unlockAuto, d4flush**

code4tranRollback

Usage: int code4tranRollback(CODE4 *codebase)

Description: This function is used to eliminate any changes made to the data files while a transaction has been active. All changes that have been made to any open data files since **code4tranStart** was called are removed. **code4tranRollback** restores the original values to the data files and terminates the currently active transaction. If new transaction is desired, **code4tranStart** must be called**.**

Note

Depending upon the setting returned by **code4unlockAuto**, performing a **code4tranRollback** may not unlock the records affected by the transaction. It may be necessary to call **code4unlock** after calling **code4tranRollback**.

code4tranRollback can not reverse the actions of CodeBase functions that create, open or close any type of file.

After calling **code4tranRollback**, all data files are set to an invalid position. Explicit positioning (e.g. **d4top**, **d4go**, etc.) must occur before any access to the data files is possible.

Returns:

r4success The data files were successfully restored to their original forms.

< 0 An error occurred during the attempt to restore the data files to their original form.

code4tranStart

Usage: int code4tranStart(CODE4 *codeBase)

Description: This function initiates a transaction. A log file must be opened explicitly or implicitly before a transaction can be initiated.

All modifications made to a data file that are beyond the scope of a transaction are automatically recorded in a log file by CodeBase. The automatic logging may be turned off or on depending on the value of **CODE4.log**, by calling **d4log**

Returns:

r4success The transaction was successfully initiated.

< 0 An error occurred while attempting to initiate the transaction.

Locking: Throughout a transaction, CodeBase acts as though the **code4unlockAuto** is set to **LOCK4OFF**. In addition, if **d4unlock** or **code4unlock** are called during a transaction, an error is returned and no unlocking is performed.

See Also: **d4log, code4logOpen, codelogCreate**

code4tranStatus

Usage: int code4tranStatus(CODE4 *codebase)

Description: This function indicates whether a transaction is in progress.

Returns:

r4active **code4tranStart** has been called to initiate a transaction.

r4inactive A transaction is not in progress.

code4unlock

Usage: int code4unlock(CODE4 *codebase)

Description: **code4unlock** removes all locks on all open data, index, and memo files.

Returns:

r4success Success.

< 0 Error. An error will be returned if this function is called during a transaction.

Locking: All data, index and memo files are unlocked once **code4unlock** completes.

code4unlock calls **d4unlock** for each open data file. 'Success' is returned even if one or more of the **d4unlock** calls returns **r4unique**.

See Also: **d4unlock, code4unlockAuto**

```
/*ex23.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    DATA4 *info, * data;

    code4init( &cb );
    info = d4open( &cb, "INFO" ) ;
    data = d4open( &cb, "DATA" ) ;

    d4lock( info, 1L ) ;
    d4lockAll( data ) ;

    code4unlock( &cb ) ;/*unlocks all open files.*/

    code4initUndo( &cb ) ;
}
```

code4unlockAuto

Usage: int code4unlockAuto(CODE4 *codebase)

Description: This function returns the setting of the automatic unlocking capability of CodeBase. The possible settings are listed below. By default, CodeBase performs automatic unlocking as defined under **LOCK4ALL**, below.

Returns:

LOCK4OFF CodeBase performs no automatic unlocking. It is up to the application developer to unlock a record, file, or append bytes after it is no longer necessary. This setting gives maximum flexibility to the developer, but should be used with care, since it can possibly result in deadlock.

LOCK4DATA CodeBase performs automatic unlocking on a data file by data file basis. Before placing a new lock on a data file, CodeBase removes any previous locks placed on the same file. This setting is only provided for backwards compatibility with CodeBase 5.x, and may not be supported in future versions of CodeBase.

LOCK4ALL CodeBase performs automatic unlocking on the entire set of open data files. Any new lock placed using any method forces an automatic removal of every lock placed on every open data file associated with the **CODE4** structure. This setting is the default action taken by CodeBase.

The following scenarios illustrate how a lock is placed in CodeBase. Consider the case when a data file function such as **d4go** must flush a

record before it can proceed. A record must be locked before it can be flushed. At this point the record may already be locked or may need to be locked.

If a new lock must be placed, all the previous locks are unlocked according to **code4unlockAuto** before the new lock is be placed. In this case, three possible results can occur depending on the setting of **code4unlockAuto**. If **code4unlockAuto** is set to **LOCK4OFF,** no automatic unlocking is done and new lock is placed. Thus after the flushing takes place all the previous locks remain in place as does the new lock. If the **code4unlockAuto** is set to **LOCK4DATA,** all the locks on the current data file are removed before the new lock is placed. When the flush is completed only the new locks will remain in place on the current data file, while the locks on any other open data files remain unchanged. If the **code4unlockAuto** is set to **LOCK4ALL,** all the locks on all open data files are removed before the new lock is placed and when the flush is completed only the new locks remain in place.

If the record is locked already, then no new locking is needed and the flushing can proceed. Since there are no new locks to be placed, no unlocking is required, so the setting of **code4unlockAuto** is irrelevant. The locks that were in place before the flushing are still in place after the flushing is completed.

The above discussion of locking procedures not only applies to flushing but to any case where locking is performed.

Note

The purpose of automatic unlocking is to make application code simpler and shorter by making it unnecessary to program unlocking protocols. In addition, automatic unlocking can prevent deadlock from occurring.

See Also: **code4lock, code4unlock, d4lock, d4unlock, d4flush**

code4unlockAutoSet

Usage: void code4unlockAutoSet(CODE4 *codebase, int autoUnlock)

Description: This function sets the automatic unlocking capabilities of CodeBase. By default, CodeBase performs automatic unlocking as defined under **LOCK4ALL**, below.

Parameters: *autoUnlock* is used to set the type of automatic unlocking used within the application. The possible settings for *autoUnlock* are listed below.

LOCK4OFF CodeBase performs no automatic unlocking. It is up to the application developer to unlock a record, file, or append bytes after it is no longer necessary. This setting gives maximum flexibility to the developer, but should be used with care, since it can possibly result in deadlock.

LOCK4DATA CodeBase performs automatic unlocking on a data file by data file basis. Before placing a new lock on a data file, CodeBase removes any previous locks placed on the same file. This setting is only provided for backwards

compatibility with CodeBase 5.x, and may not be supported in future versions of CodeBase.

LOCK4ALL CodeBase performs automatic unlocking on the entire set of open data files. Any new lock placed using any method forces an automatic removal of every lock placed on every open data file associated with the **CODE4** structure. This setting is the default action taken by CodeBase.

Note

In order to understand this function better, consider groups of locks placed with **code4lock** to be a single lock.

See Also: **code4lock, code4unlock, d4lock, d4unlock, code4unlockAuto**

Conversion Functions

c4atod
c4atoi
c4atol
c4encode
c4trimN

Conversion functions change information from one format to another. They are different from ANSI C runtime library conversion functions because they all have string length parameters. This is useful when dealing with strings that may not be null terminated.

Function **c4encode** is useful for character transformations. In addition, function **c4trimN** is useful because it guarantees a null terminated trimmed result and because it can safely take the result of a **sizeof** operation as a parameter.

Conversion Function Reference

c4atod

Usage: double c4atod(const char *str, int lenStr)

Description: This function converts a string of characters into a **(double).** The string does not have to be null terminated because *lenStr* specifies the number of characters to convert.

The maximum string length is **(int)** 49.

WARNING

There is a replacement function for Borland compiler development using pre-built Microsoft DLL's under windows. Refer to the COMPILER.TXT file.

Returns: The converted **(double)** result.

```
/*ex24*/
#include "d4all.h"

double function() /*this function returns '(double) 67.3" */
{
     double d ;

     /* only the first five characters are used */
     d = c4atod("67.37 Garbage", 5 ) ;

     return d ;
}
```

c4atoi

Usage: int c4atoi(const char *str, int lenStr)

Description: This function converts a string of characters into a **(int).** The string does not have to be null terminated because *lenStr* specifies the number of characters to convert.

The maximum string length is **(int)** 127.

Returns: The converted integer result.

```
/*ex25.c*/
/* 'f4int()' uses 'c4atoi' because database field data is not null terminated */

#include "d4all.h"

int f4int( FIELD4 *field )
{
    /*convert the field data into an 'int' */
    return c4atoi(f4ptr(field), (int) f4len(field)) ;
}
```

c4atol

Usage: long c4atol(const char *str, int lenStr)

Description: This function converts a string of characters into a **(long).** The string does not have to be null terminated because *lenStr* specifies the number of characters to convert.

The maximum string length is **(int)** 127.

Returns: The converted long result.

```
/*ex26*/
#include "d4all.h"

void main()
{
    CODE4 codeBase ;
    FIELD4 *value ;
    DATA4 *data ;
    long l ;

    code4init( &codeBase ) ;

    data = d4open( &codeBase, "VALUES" ) ;
    value = d4field( data, "VALUE" ) ;

    d4go( data, 1L ) ;
    l = c4atol( f4ptr( value ), f4len( value ) ;

    printf( "\n Value: %ld", l ) ;
    code4initUndo( &codeBase ) ;
}
```

```
/*ex27.c*/
/* 'long result' will be '3' since it only converts the 2 characters as specified by the
    second parameter. */
long result = c4atol("3547", 2) ;
```

c4encode

Usage: void c4encode(char *to, const char *from, char *temTo, const char *temFrom)

Description: Characters are moved from the string *from* to the string *to*. The format of the result in *to* is specified by the template *temTo*. The format of the source string *from* is specified by the template in *temFrom*.

There are no predetermined formatting characters. However, if a character in *temFrom* matches a character in *temTo*, the corresponding character in *from* is moved to the corresponding position in *to*.

Parameters:

to The destination of the converted information.

from The information to be converted.

temTo A template of the destination formatting.

temFrom A template of the source information.

Note

As in the example, any characters in *temTo* which do not match characters in *temFrom* are copied to corresponding positions in *to.*

```
/*ex28.c*/
/* The result put into 'to' will be "C B A" */
c4encode( to, "ABC", "3 2 1", "123" ) ;

/* The result put into 'to' will be "A&B&C" */
c4encode( to, "30-12/1990", "789A4512", "123456789A" ) ;

/* The result put into 'to' will be "12/30/90" */
c4encode( to, "19901230", "EF/GH/CD", "ABCDEFGH" ) ;
```

c4trimN

Usage: void c4trimN(char *str, int numBytes)

Description: This function trims the blanks off a string *str*. Regardless, a null character is placed in the last byte in the string to ensure it is null terminated. This null termination guarantee is useful for reducing uncertainty in order to build solid C programs which never fail.

Note

When trimming character arrays with specified sizes, use the **sizeof** operator. Then if the dimension of the array is changed at a later time, it is not necessary to perform additional maintenance. Avoid hard coding fixed length values.

Parameters:

str A pointer to a string.

numBytes The number of bytes of memory declared for the string.

```
/*ex29.c*/
#include "d4all.h"

void disp( char *ptr )
{
    char buf[80] ;
    strncpy( buf, ptr, sizeof(buf) ) ;

    /* A null will be placed in the 80th byte of 'buf' to guarantee that it is null
    terminated */

    c4trimN( ptr, sizeof(buf) ) ;
    printf( "Display Result: %structure", ptr) ;
}
```

Data File Functions

d4alias	d4fieldNumber	d4memoCompress	d4seek
d4aliasSet	d4fileName	d4numFields	d4seekDouble
d4append	d4flush	d4open	d4seekN
d4appendBlank	d4freeBlocks	d4openClone	d4seekNext
d4appendStart	d4go	d4optimize	d4seekNextDouble
d4blank	d4goBof	d4optimizeWrite	d4seekNextN
d4bof	d4goEof	d4pack	d4skip
d4bottom	d4index	d4position	d4tag
d4changed	d4lock	d4positionSet	d4tagDefault
d4check	d4lockAdd	d4recall	d4tagNext
d4close	d4lockAddAll	d4recCount	d4tagPrev
d4create	d4lockAddAppend	d4recNo	d4tagSelect
d4delete	d4lockAddFile	d4record	d4tagSelected
d4deleted	d4lockAll	d4recWidth	d4tagSync
d4eof	d4lockAppend	d4refresh	d4top
d4field	d4lockFile	d4refreshRecord	d4unlock
d4fieldInfo	d4log	d4reindex	d4write
d4fieldJ	d4logStatus	d4remove	d4zap

The data file functions correspond to high level dBASE commands. They are used to store and retrieve information from data files.

Each data file has a current record number, a record buffer, and a selected tag. Whenever a function changes any of these, it is noted in the documentation.

In addition, the record buffer has a "record changed" flag attached to it. When the record buffer is changed through use of a field function, the "record changed" flag is set to true (non-zero). The "record changed" flag tells the data file functions to automatically write the changed record to the data file before a different record is read. The automatic writing of changed records is called "record buffer flushing".

The data file functions also keep track of an end of file (eof) and a beginning of file (bof) flag. When the program skips past the last record, the end of file flag is set to true (non-zero) and the record buffer becomes blank. When the program attempts to skip before the first record, the record buffer stays the same as the first record and the beginning of file flag is set to true (non-zero). These bof/eof flags are reset when a record is read or written.

To work with a data file, use the **d4open** to open the file or use **d4create** to create a new file. Both of these functions return a pointer to a **DATA4** structure. All of the other data file functions use this pointer as their first parameter. This parameter tells the function which data file to operate on. Once the storage and/or retrieval of information is completed, use **d4close**, **code4close** or **code4initUndo** to close the data file.

Note

Often the data file functions require that a file or a portion of the file be locked before the function can proceed. If the file or portion of the file has the required locks already, then the data file functions recognize this and proceed without doing any additional locking.

After most data file functions, **code4unlockAuto** is used to

determine what kind of unlocking occurs. In addition, if a field function writes to the database, the **CODE4.lockEnforce** setting is checked to ensure the record is locked by the application prior to performing the write.

```
/*ex30.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    DATA4 *info ;
    FIELD4 *field ;
    long iRec ;

    code4init( &cb ) ;
    cb.accessMode = OPEN4DENY_RW ;
    info = d4open( &cb, "DATA1.DBF") ;
    code4optStart( &cb ) ;

    field = d4field( info, "NAME") ;

    for(iRec = 1L ; iRec <= d4recCount( info ) ; iRec++ )
    {
      d4go( info, iRec ) ;
      f4assign( field, "New Data" ) ;
    }

    d4close( info ) ;
    code4initUndo( &cb ) ;
    code4exit( &cb ) ;
 }
```

Data File Function Reference
d4alias

Usage: const char *d4alias(DATA4 *data)

Description: This function returns a character string containing the alias of the data file.

The data file alias is a string of characters which identifies the data file. By default, it is assigned the name that is passed to **d4open** or **d4create** (minus extension and path). However, once the data file is opened, the user may assign a different alias by using **d4aliasSet.** The alias is used by **code4data** to return a data file pointer. It can also be used as part of a dBASE expression. Refer to the "Appendix C: dBASE Expressions".

Parameters:

data A pointer to the **DATA4** structure.

Return: This function returns a null terminated string containing the alias of the data file.

See Also: **d4aliasSet**

```
/*ex31.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ;  /* for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
```

```
   DATA4 *data ;

    code4init( &cb ) ;
    data = d4open( &cb , "INFO.DBF" ) ;
    printf( "%s is the alias of the file INFO.DBF \n", d4alias( data )) ;
    code4initUndo( &cb ) ;
}
```

d4aliasSet

Usage: void d4aliasSet(DATA4 *data, const char *alias)

Description: The data file alias is a string of characters which identifies the data file. After the data file is opened, the user may assign a different alias by using **d4aliasSet.** See **d4alias** for more information.

Parameters:

data A pointer to the **DATA4** structure.

alias The data file alias is set to the character array pointed to by *alias*.

See Also: **d4alias**

d4append

Usage: int d4append(DATA4 *data)

Description: **d4append** works in conjunction with **d4appendStart** to add a new record to the end of a data file.

First, **d4appendStart** is called; next, the appropriate changes to the record buffer (if any) are made; and finally, **d4append** is called to create the new record.

d4append maintains all open index files by adding keys to respective tags. In addition, **d4append** ensures that any existing memo fields are handled in accordance with the *useMemoEntries* setting of **d4appendStart**.

The current record is set to the newly appended record.

Note

When many records are being appended at once, the most efficient method for locking the files is to use either **d4lockAll**, or **d4lockAddAll** followed by **code4lock**. In addition, using write optimization while appending many records will improve performance.

WARNING

If a file is write optimized, the file length on the disk will not be updated until the changes to the file are flushed. This can sometimes result in delayed "out of disk space" errors. For example, memory buffers could hold hundreds of appended records before determining that some could not be saved to disk.

If your application could run into this problem and you would

like to avoid it, compile the library with the **S4SAFE** switch defined. This will result in the disk file length always being explicitly set, giving "out of disk space" errors as soon as an attempt to exceed available disk space occurs. The cost is reduced performance.

Returns:

r4success Success

r4locked The record was not appended because a required lock attempt did not succeed.

r4unique The record was not appended because to do so would result in a non-unique key for a unique key tag. In this case **t4unique** returns **r4unique** for the tag so an error is not generated.

< 0 Error

Locking: The append bytes and the new record to be appended, must be locked before the append can proceed.

In the case where the append bytes required locking, the append bytes are unlocked according to **code4unlockAuto** after the append is completed. On the other hand, if the append bytes were previously locked, they remain locked after the append.

The new record remains locked after the append regardless of whether it was previously locked or newly locked.

See **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

See Also: **d4appendBlank**, **d4appendStart, code4unlockAuto**

```
/*ex32.c*/
#include "d4all.h"

CODE4 cb ; /* CODE4 may be constructed globally.*/
DATA4  *data ;
FIELD4 *field ;

void main( void )
{
    code4init( &cb ) ;
    cb.accessMode = OPEN4DENY_RW ;
    data = d4open( &cb, "DATA1" ) ;
   code4optStart( &cb ) ;

   d4appendBlank( data ) ;

   /* Append a copy of record two.  (Assume record two exists.)*/
   d4go( data, 2 ) ;
   d4appendStart( data, 0) ;/* a false useMemoEntries parameter */
                /* Append a copy of record 2 with a blank memo entry.*/
   d4append( data ) ;

   d4go( data, 2 ) ;
   d4appendStart( data, 1 ) ; /* a true parameter means use memo entries */
   d4append( data ) ;     /*append a copy of record 2 with its memo entry*/

   /* Set the record buffer to blank, change a field's value, and append
        the resulting record.*/
    d4appendStart( data, 0  ) ;
   d4blank( data ) ;

    field = d4field( data, "NAME" ) ;
   f4assign( field, "New field value" ) ;
```

```
    d4append( data ) ;
    /* close all open files and release any allocated memory */
    code4initUndo( &cb ) ;
}
```

d4appendBlank

Usage: int d4appendBlank(DATA4 *data)

Description: A blank record is appended to the end of the data file. Any changes to the current record buffer are flushed to disk prior to the creation of the blank record. Any open index files are automatically updated.

The current record is set to the newly appended record.

Note

If a data file is locked or opened exclusively and many records are being appended, the operation can be speeded up by using write optimization.

WARNING

If a file is write optimized, the file length on the disk will not be updated until the changes to the file are flushed. This can sometimes result in delayed "out of disk space" errors. For example, memory buffers could hold hundreds of appended records before determining that some could not be saved to disk.

If your application could run into this problem and you would like to avoid it, compile the library with the **S4SAFE** switch defined. This will result in the disk file length always being explicitly set, giving "out of disk space" errors as soon as an attempt to exceed available disk space occurs. The cost is reduced performance.

Returns:

r4success Success

r4locked The record was not appended because a required lock attempt did not succeed.

r4unique The record was not appended. A non-unique key was detected in a unique tag when either flushing or when appending a blank record. In this case **t4unique** returns **r4unique** for the tag so an error is not generated.

< 0 Error

Locking: If record buffer flushing is required, the record and index files must be locked. The append bytes and the new record to be appended, must be locked before the append can proceed.

In the case where the append bytes required locking, the append bytes are unlocked according to **code4unlockAuto** after the append is completed.

On the other hand, if the append bytes were previously locked, they remain locked after the append.

The new record remains locked after the append regardless of whether it was previously locked or newly locked.

See **d4flush** and **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

See Also: **d4append, d4flush, code4unlockAuto**

```
/*ex33.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
     int i ;

     code4init( &cb ) ;
     data = d4open( &cb, "DATA" ) ;
     error4exitTest( &cb ) ;

    /* Add 20 blank records */
    for(i = 20 ; i ; i-- )
       d4appendBlank( data ) ;

    /* Close the data file.*/
    d4close( data ) ;
    code4initUndo( &cb ) ; /* Free up any memory used. */
}
```

d4appendStart

Usage: int d4appendStart(DATA4 *data, int useMemoEntries)

Description: **d4appendStart** is used in conjunction with function **d4append** in order to append a record to a data file.

After **d4appendStart** is called, the current record number becomes zero. This lets CodeBase know that **d4appendStart** has been called; and that if changes are made to the record buffer, no automatic flushing should be done.

d4appendStart does not change the current record buffer. To initialize the record buffer to blank after **d4appendStart** is called, use **d4blank**.

WARNING

The first byte of the record buffer contains the 'record deletion' flag. This flag does not change when **d4appendStart** is called. To ensure that the 'record deletion' flag is not set (ie. blank) when appending a potentially deleted copy of another record, call **d4recall** after calling **d4appendStart**.

Note

It is not necessary to call **d4append** after **d4appendStart**. For example, after a **d4appendStart** call, an end user could decide to "Cancel Append". In this case, there would be no corresponding **d4append** call, and the record would not be appended. If flushing is suspended using **d4appendStart**, the field functions may be used to freely manipulate the record

buffer without fear of corrupting the data on disk.

An example that illustrates this, is to read a record, suspend flushing, modify the record, and then write the record buffer to another record with **d4write**.

Another example is to store data file records in memory, copy them into the record buffer using **d4record**, and then access the record with field functions.

Parameters: Parameter *useMemoEntries* is true (non-zero) in order to make a copy the current record's memo entries for the new record. If *useMemoEntries* is false (zero), the new record starts with blank memos.

Returns:

r4success Success

r4locked The required flushing was not successful because a required lock attempt did not succeed.

r4unique The "append start" did not succeed. A non-unique key was detected in a unique tag when flushing the record buffer. In this case **t4unique** returns **r4unique** for the tag so an error is not generated.

< 0 Error

Locking: If the record changed flag is set prior to a call to **d4appendStart**, the current record buffer is flushed to disk. The record and index files must be locked before the flushing can proceed. See **d4flush** and **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

See Also: **d4append**, **d4write**, **d4changed, d4flush, code4unlockAuto**

Example: See **d4append**.

d4blank

Usage: void d4blank(DATA4 *data)

Description: The record buffer for the data file is set to spaces. In addition, the "record changed" flag is set to true (non-zero) and the "deleted" flag is set to false (zero).

If the current record has one or more memo entries, calling **d4blank** will remove the record's reference to the entries. The orphaned memo entries may be removed from the memo file by calling **d4memoCompress**.

```
/*ex34.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
```

```
    code4init( &cb ) ;
    data = d4open( &cb, "DATA1") ;

   for( d4top( data ) ; !d4eof( data ) ; d4skip( data, 1L ) )
      d4blank( data )  ;
  /* blank out all records in the data file*/
   code4initUndo( &cb ) ;  /* close all files and release any memory */
}
```

d4bof

Usage: int d4bof(DATA4 *data)

Description: This function returns true (non-zero) after **d4skip** attempts to skip backwards past the first record in the data files. When the beginning of file condition becomes true (non-zero) , it remains true until the data file is repositioned or written to.

It is impossible to skip backwards to record zero.

See Also: **d4skip**

```
/*ex35.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
    FIELD4 *field ;

    code4init( &cb ) ;
    data = d4open( &cb, "INFO" );
    field = d4fieldJ( data, 1 ) ;

    /* output the first field of every record in reverse sequential order.*/
    for( d4bottom( data ) ; !d4bof( data ) ; d4skip( data, -1 ) )
          printf("%s \n",f4str( field)) ;
    d4close( data ) ;
    code4initUndo( &cb ) ;
}
```

d4bottom

Usage: int d4bottom(DATA4 *data)

Description: The bottom record of the data file is read into the record buffer and the current record number is updated. The selected tag is used to determine which is the bottom record. If no tag is selected, the last record of the data file is read.

Returns:

r4success Success.

r4locked A required lock attempt did not succeed.

r4eof End of File. There are no records in the data file.

r4unique This value is returned when a record flush causes a duplicate key in a unique tag, and **t4unique** returned **r4unique** for the tag.

< 0 Error

Locking: The record and index files must be locked if record buffer flushing is required. If **CODE4.readLock** is true (non-zero), the new record must be locked before it is read. See **d4flush** and **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

See Also: **d4flush, code4unlockAuto, CODE4.readLock**

```
/*ex36.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
     FIELD4 *field ;

     code4init( &cb ) ;
     data = d4open( &cb, "INFO" ) ;
     field = d4field( data, "NAME" ) ;

    d4bottom( data ) ;
    printf("Last Name added: %s\n", f4str( field ) ) ;
    d4close( data ) ;
     code4initUndo( &cb ) ;
}
```

d4changed

Usage: int d4changed(DATA4 *data, int flag)

Description: Normally, after the record buffer is changed using a field function, the change is automatically written to the data file at an appropriate time. (ie. When a data file function such as **d4go** or **d4skip** is called.) CodeBase accomplishes this by maintaining a 'record changed' flag for each data file. When a field function changes the record buffer, this flag is set to true (non-zero). When the changes are written to disk this flag is changed to false (zero). **d4changed** changes and returns the previous status of this 'record changed' flag.

Parameters:

flag The record changed flag is set to the value of *flag*. The possible settings are:

- `> 0` The record buffer is flagged as 'changed'. This is useful when the record buffer is modified directly by the application. CodeBase will then know to flush the changes at the appropriate time.
- `0` The record buffer is flagged as 'not changed'. This effectively tells CodeBase not to flush any record buffer changes.
- `< 0` Nothing is done except that the current status of the 'record changed' flag is returned.

Note

A typical data file edit function could directly change the data file record buffer, using the field functions, to save editing changes. If the end-user decides to abort the changes, **d4changed(datafile, 0)** could be called before any positioning statements to instruct CodeBase not to flush the changes to

the data file.

Returns: The previous status of the "record changed" flag is returned.

Non-Zero The flag status was ‘changed’.

Zero The flag status was ‘not changed’.

Note

Changes that have been "written" to the data file but have not been flushed to disk, due to memory write optimization, may not be aborted.

See Also: **CODE4.lockEnforce, d4appendStart**

```
/*ex37.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
    FIELD4 *field ;

    code4init( &cb ) ;
    data = d4open( &cb, "DATA2" ) ;
    field = d4field( data, "NAME" ) ;
    d4top( data ) ;
    if ( d4changed( data, -1 ) != 0 )       /*Displays FALSE */
        printf( "Changed status: TRUE\n") ;
    else
    printf( "Changed status: FALSE\n") ;

    f4assign( field, "TEMP DATA") ;
    if ( d4changed( data, -1 ) != 0 )     /*Displays TRUE */
        printf( "Changed status: TRUE\n") ;
    else
    printf( "Changed status: FALSE\n") ;

    d4changed( data, 0 ) ;

    d4close( data ) ;      /* The top record is not flushed.*/
    code4initUndo( &cb ) ;
 }
```

d4check

Usage: int d4check(DATA4 *data)

Description: The contents of all the open index files corresponding to the data file are verified against the contents of the data file.

This function is provided mainly for debugging purposes. It can take as long as reindexing.

WARNING

This member function may not be used to check a data file while a transaction is in progress. In fact, any attempt to do so will fail and generate an **e4transViolation** error.

Returns:

r4success Success

r4locked A required lock attempt did not succeed.

< 0 Error. An index file is not up to date.

Locking: The data file and corresponding index and memo files are locked during and after **d4check**.

```
/*ex38.c*/
#include "d4all.h"
extern unsigned  _stklen =  10000 ;  /* Borland Only */

/* Check the validity of an index file. */
void main( int argc, char *argv[ 2 ] )
{
    if ( argc > 2 )
    {
         CODE4 cb ;
         DATA4 *checkData ;
         INDEX4 *testIndex ;

         code4init( &cb ) ;
         cb.accessMode = OPEN4DENY_RW ;
         cb.autoOpen = 0 ;  /* open index file manually.*/

         checkData = d4open( &cb, argv[1] ) ;
         testIndex = i4open( checkData, argv[2]);

         error4exitTest( &cb ) ;
         code4optStart( &cb ) ;

         if ( d4check( checkData ) == r4success )
           printf("\nIndex is OK !!\n") ;
         else
             printf("\nProblem with Index !!\n") ;
         code4initUndo( &cb );
    }
    else
       printf( "\nUsage: PROGRAM  DataFile  IndexFile\n") ;
 }
```

d4close

Usage: int d4close(DATA4 *data)

Description: **d4close** closes the data file and its corresponding index and memo files. Before closing the files, any necessary flushing to disk is done. If the data file has been modified, the time stamp is also updated.

If **d4close** is called during a transaction, the action of **d4close** is delayed until the transaction is committed or rolled back. However, the **DATA4** structure is no longer valid after **d4close** is called.

Returns:

r4success Success. The data file was successfully closed. The **DATA4** pointer is invalid after calling **d4close** and it should not be used.

< 0 Error

Locking: **d4close** does any locking necessary to flush changes to disk. After any necessary updating is done, **d4close** unlocks all files associated with the data file.

Before closing the file, **d4close** attempts to flush the most recent changes to the record buffer. If the flush attempt does not succeed, **d4close** continues and closes the file anyway. Consequently, **d4close** never returns **r4locked** or **r4unique**. If these values must be checked for prior

to the closure, call **d4flush**, **d4write** or **d4append**, as appropriate, before calling **d4close**.

d4create

Usage: DATA4 *d4create(CODE4 *code, const char *name,
const FIELD4INFO *fields, const TAG4INFO *tags)

Description: **d4create** creates a data file, possibly a "production" index file, and possibly a memo file. A "production" index file is created if the *tags* parameter is not null. A memo file is created if the *fields* structure contains a memo field. See the chapters on field and tag functions for more information.

Note

In the client-server configuration, **code4connect** will automatically be called if connection to the server has not been previously established.

Parameter *fields* defines the fields of the data file to be created. It points to an array of **FIELD4INFO** entries with each entry specifying the different field attributes. Specifically, the members of **FIELD4INFO** are defined as follows:

- **(char *) name** This is the name of the field. Each field name should be unique to the data file. A field name is up to ten alphanumeric or underscore characters - except for the first character, which must be a letter. Any characters in the field name over ten are ignored.

- **(char *) type** This is the type of the field. Fields must be one of the following types: Character, Date, Numeric, Floating Point, Logical, Memo, Binary or General.

- **(int) len** This is the length of the field. Date, Memo and Logical fields, have pre-determined lengths. These pre-determined lengths are used regardless of the lengths specified for the **FIELD4INFO** structure by the application.

- **(int) dec** This is the number of decimals in Numeric or Floating Point fields.

Specifics on the types of fields and their limitations are listed in the following table:

Type	*Abbreviation*	*Length*	*Decimals*	*Information Type*
Binary	'B' or r4bin	Set to 10. Actual data is in a separate	0	Binary fields are handled in the same way as memo fields. It stores binary information. The

		file.		amount of information is dependent upon the size of an **(unsigned int)**.
Character	'C' or r4str	1 to 65533 1 to 254 to keep dBASE and FoxPro file compatibility	0	Character fields can store any type of information including binary.
Date	'D' or r4date	8	0	Date Fields store date information only. It is stored in CCYYMMDD format.
Floating Point	'F' or r4float	The length depends on the format **S4CLIPPER** 1 to 19 **S4FOX** 1 to 20 **S4MDX** 1 to 20	The number of decimals depends on the format **S4CLIPPER** is the minimum of (len - 2) and 15 **S4FOX** (len - 1) **S4MDX** (len - 2)	CodeBase treats this field like it was a Numeric field. This field is useful for compatibility with dBASE and FoxPro, which treat Floating point and Numeric fields differently. Use this field to store values that will be used in floating point calculations.
General	'G' or r4gen	Set to 10. Actual data is in a separate file.	0	General fields are handled in the same way as memo fields. It stores OLEs. The amount of information is dependent upon the size of an **(unsigned int)**.
Logical	'L' or r4log	1	0	Logical fields store either true or false. The values that represent true are 'T','t','Y','y'. The values that represent false are 'F','f','N','n'.
Memo	'M' or r4memo	Set to 10. Actual data is in a separate file.	0	Memo fields store the same type of information as the Character type. The amount of information is dependent upon the size of an **(unsigned int)**.
Numeric	'N' or r4num	The length depends on the format **S4CLIPPER** 1 to 19 **S4FOX** 1 to 20	The number of decimals depends on the format **S4CLIPPER** is the minimum of	Numeric fields store numerical information. It is stored internally as a string of digits. This field is useful for compatibility with dBASE and FoxPro, which treat Floating

		S4MDX 1 to 20	(len - 2) and 15 **S4FOX** (len - 1) **S4MDX** (len - 2)	point and Numeric fields differently. Use this field to store values that will be NOT be used in floating point calculations.

WARNING

The Binary and General field types are NOT compatible with some versions of dBASE, FoxPro, Clipper and other dBASE compatible products. Only use Binary and General fields with products that support these field types.

Parameters:

code This is a pointer to a **CODE4** structure. **code4init** must be executed before calling **d4create**. The **CODE4** structure contains default settings, which are used to determine how the data file is opened and accessed. See **CODE4** member variables for more information.

name The name for the data file, index file, and memo file. The default data file name extension is **.DBF**. See **d4open** for a list of default index and memo file extensions.

If a path is provided in the single user or multi-user configuration, it is used. Otherwise, the file is assumed to be in the current directory.

When using the client-server configuration, refer to the Catalog file documentation in the CodeServer manual for the details on how to specify the file name.

The default alias for the data file is initially set to *name*, disregarding path and extension.

fields *fields* is a pointer to an array of **FIELD4INFO** structures. This parameter is used to specify the field definitions for the new data file. Each entry in the **FIELD4INFO** array specifies a field to be created in the new data file. The last entry is null to signal that there are no additional entries.

tags *tags* is a pointer to an array of **TAG4INFO** structures. This parameter is used to specify the tag definitions for the production index file. Each entry in the **TAG4INFO** array specifies a tag to be created in the new index file. The last entry is null to signal that there are no additional entries. If *tags* is null, no production index file is created.

Returns:

Not Null Success. A pointer to the corresponding **DATA4** structure is returned.

Null The data, index or memo file was not successfully created. Inspect the **CODE4.errorCode** for more detailed information. If a negative value is returned then an error has occurred.

A file cannot be created when a file with the same name already exists and **CODE4.safety** is true (non-zero). Refer to **CODE4.safety** for more details.

CODE4.errorCode may contain **r4unique** or **r4noCreate**. **r4unique** indicates that a duplicate key was located for that tag and **TAG4INFO.unique** was **r4unique** for that tag. **r4noCreate** indicates that a file could not be created and **CODE4.errCreate** is false (zero).

Locking: Nothing related to the data file is locked upon completion.

See Also: **code4connect**, **i4create, CODE4.safety**

```
/*ex39.c*/
#include "d4all.h"

static FIELD4INFO fieldArray[ ] =
{
    { "NAME_FIELD", 'C', 20, 0 },
    { "AGE_FIELD",  'N',  3, 0 },
    { "BIRTH_DATE", 'D',  8, 0 },
    { 0,0,0,0 }
};

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;

     code4init( &cb ) ;
     cb.safety = 0 ;       /* overwrite the file if it exists*/
    data = d4create( &cb, "NEWDBF", fieldArray, 0 ) ;
    error4exitTest( &cb ) ;

    if( cb.errorCode )
       printf( "An error occurred, NEWDBF not created\n") ;
    else
       printf( "Created successfully!\n") ;

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

d4delete

Usage: void d4delete(DATA4 *data)

Description: The current record buffer is marked for deletion. In addition, the record changed flag is set to true (non-zero), so that the record -- including the deletion mark -- is flushed to disk at an appropriate time.

The deletion mark may be removed by calling **d4recall**.

See Also: **d4deleted**, **d4recall**, **d4pack**

```
/*ex40.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* Borland compilers only  */

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;

     code4init( &cb ) ;
    cb.accessMode = OPEN4DENY_RW ; /* open file exclusively to speed pack*/

    data = d4open( &cb, "DATA1" ) ;
    error4exitTest( &cb ) ;
    code4optStart( &cb ) ;

    for( d4top( data ); ! d4eof( data ); d4skip( data, 2 ) )
        d4delete( data ) ;        /* Mark the record for deletion*/

     /*  Physically remove the deleted records from the disk*/
    d4pack( data ) ;
```

```
    d4close( data ) ;
    code4initUndo( &cb ) ;
}
```

d4deleted

Usage: int d4deleted(DATA4 *data)

Description: This function returns whether the record in the current record buffer is marked for deletion. If there is no current record, the result is undefined.

Returns:

non-zero The record is marked for deletion.

0 The record is NOT marked for deletion.

See Also: **d4delete**, **d4recall**

```
/*ex41.c*/
#include "d4all.h"

void main( void )
{
    CODE4 codeBase ;
    DATA4 *file ;
     long count ;

     code4init( &codeBase ) ;
     file = d4open( &codeBase, "INFO") ;

    code4optStart( &codeBase ) ;

     count = 0 ;

    for( d4top( file ); !d4eof( file ); d4skip( file, 1L ) )
       if( d4deleted( file ) )
          count++ ;
     printf( "INFO has %d deleted records\n", count) ;
     code4initUndo( &codeBase ) ;
 }
```

d4eof

Usage: int d4eof(DATA4 *data)

Description: If you attempt to position past the last record in the data file with **d4skip** or **d4seek**, the End of File flag is set and **d4eof** returns a true (non-zero) value. At any other point in the data file, a false (zero) value is returned.

Returns:

> 0 An end of file condition has occurred. **d4eof** will return this value until the data file is repositioned or modified.

0 This return indicates that an end of file condition has not occurred.

< 0 The **DATA4** structure is invalid or contains an error value.

```
/*ex42.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    DATA4 *infoFile ;

     code4init( &cb ) ;
```

```
    infoFile = d4open( &cb, "INFO" ) ;

    /* Go to the end of file and set the End of file flag*/
    d4goEof( infoFile ) ;

    /* Check to see if the end of file flag is set*/
    if( d4eof( infoFile )  )
    {
       printf( "This is always true\n") ;
       d4bottom( infoFile ) ; /* reset the eof flag*/
    }

    d4close( infoFile ) ;
    code4initUndo( &cb ) ;
}
```

d4field

Usage: FIELD4 *d4field(DATA4 *data, const char *name)

Description: A field name is looked up in the data file. A pointer, which can be used with field functions, is returned.

Parameters:

name A null terminated character array containing the name of a field in the data file.

Returns:

Not Null A pointer to the **FIELD4** structure corresponding to the field specified is returned.

Null The field was not located. This is an error condition if **CODE4.errFieldName** is true (default).

WARNING

The field pointer returned becomes obsolete once the data file is closed.

See Also: **CODE4.errFieldName**

d4fieldInfo

Usage: FIELD4INFO *d4fieldInfo(DATA4 *data)

Description: A pointer to a **FIELD4INFO** array, which corresponds to the data file, is returned.

This structure is created so it can be used a parameter to **d4create**.

WARNING

The return value becomes obsolete once the data file is closed. This is because the field names of the data file are referenced by the returned **FIELD4INFO** array.
The **FIELD4INFO** return value needs to be freed with function **u4free**.

Returns: A null return means that not enough memory could be allocated.

See Also: **d4create**

d4fieldJ

Usage: FIELD4 *d4fieldJ(DATA4 *data, int jField)

Description: A pointer to the *jField*th data file field is returned.

The parameter *jField* must be between one and the number of fields.

(ie. 1<= *jField* <= **d4numFields**)

WARNING

This field pointer becomes obsolete once the data file is closed.

See Also: Field functions

d4fieldNumber

Usage: int d4fieldNumber(DATA4 *data, const char *name)

Description: A search is made for the specified field name and its position in the data file, starting from one, is returned.

Parameters:

name A null terminated character array containing the name of the field to search for.

Returns:

> 0 The field number of the located field.

-1 The field name was not found. If **CODE4.errFieldName** is true (non-zero), this is an error condition.

See Also: **CODE4.errFieldName, f4number**

```
/*ex43.c */
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    DATA4 *infoFile ;
     FIELD4 *field ;
     int num ;

     code4init( &cb ) ;
     infoFile = d4open( &cb, "INFO" );

    for(num = d4numFields( infoFile ) ; num ; num -- )
    {
         field = d4fieldJ( infoFile, num ) ;
       if( num == d4fieldNumber( infoFile, f4name( field )) )
          printf( "This is always true.\n") ;
    }
     code4initUndo( &cb ) ;
}
```

d4fileName

Usage: const char *d4fileName(DATA4 *data)

Description: The file name of the data file is returned as a null terminated character string complete with the file extension and all path information. This information is returned regardless of whether a file extension or a path was specified in the name parameter passed to **d4open** or **d4create**. This value is not altered by **d4alias**.

The pointer that is returned by this function, points to internal memory, which may not be valid after the next call to a CodeBase function. Therefore, if the name is needed for an extended period of time, the file name should be copied by the application.

d4flush

Usage: int d4flush(DATA4 *data)

Description: The data file, its index files (if any), and its memo files (if any) are all explicitly written to disk. This includes any field value changes. Once completed, the "record changed" flag is reset to false (zero).

In addition, **d4flush** tries to guarantee that all file changes are physically written to disk through a call to **file4flush**. Refer to function **file4flush** for more information.

If the current record number is unknown due to **d4appendStart** being called or due to the file having been opened but not positioned to a record, **d4flush** discards memo field changes and resets the "record changed" flag to false (zero).

Note

If you are flushing a data file on a regular basis it is best to disable memory write optimization for the data file. This is because explicit flushing negates the benefits of write optimization. Refer to **CODE4.optimizeWrite**.

Returns:

r4success Success.

r4locked A required lock attempt did not succeed.

r4unique The record was not written due to the following circumstances: writing the record caused a duplicate key in a unique key tag and **t4unique** returned **r4unique** for the tag.

< 0 Error.

Locking: If changes have been made to any field, the record must be locked before it can be flushed. If a new lock is required, everything is unlocked according to the **code4unlockAuto** setting and then the required lock is placed. If the required lock is already in place, **d4flush** does not unlock or lock anything and after the flush is completed, the previous locks remain in place.

The index files and append bytes may need to be locked during updates. If index files require locking, they are locked prior to the flush. After the flushing is complete, the index files are unlocked. If the index files are

already locked, no new locking is required and after the flushing is finished, the index files remain locked.

The above discussion on locking procedures for index files not only applies to flushing but to any case where index locking is performed.

Client-Server: In the client-server configuration, the changes are flushed to the server.

See Also: **file4flush**, **CODE4.optimizeWrite, code4unlockAuto**

```
/*ex44.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
     FIELD4 *age ;

     code4init( &cb ) ;
     data = d4open( &cb, "INFO" );
     d4top( data );
    d4go( data, 2 ) ;
     age = d4field ( data, "AGE" ) ;
     d4lockAll( data );
    f4assignLong( age, 49 ) ;

    /* Explicitly flush the change to disk in case power goes out*/
    d4flush( data ) ;

    /* some other code*/

    d4close( data ) ;
    code4initUndo( &cb ) ;
}
```

d4freeBlocks

Usage: int d4freeBlocks(DATA4 *data)

Description: All tag blocks in memory for the tags of the data file are flushed to disk and freed for use by other tags.

Returns:

r4success Success.

< 0 Error.

See Also: "Appendix D: CodeBase Limits"

d4go

Usage: int d4go(DATA4 *data, long recordNumber)

Description: Function **d4go** reads the specified record into the record buffer and *recordNumber* becomes the current record number. Before reading the new record, **d4go** writes the current record buffer to disk if the record changed flag is set.

If memory optimizations are being used, use **d4skip** instead of **d4go** when sequentially reading data file records. When memory optimizations are used, CodeBase detects the sequential skipping and automatically optimizes the operations when **d4skip** is used.

Parameters:

recordNumber — This **long** value specifies the physical record number to read into the record buffer. To succeed, *recordNumber* must be > 0 and <= **d4recCount**.

Returns:

r4success — Success.

r4locked — A required lock attempt did not succeed.

r4entry — **CODE4.errGo** is false (zero) and the data file record did not exist.

r4unique — This value is returned when a record flush causes a duplicate key in a unique tag, and **t4unique** returned **r4unique** for the tag.

< 0 — Error.

Locking: If record buffer flushing is required, the record and index files must be locked. If **CODE4.readLock** is true (non-zero), the new record must be locked before it is read. See **d4flush** and **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

If the data file is shared and memory optimization is being used, the new record might not reflect recent changes or additions made by other users. To avoid this, disable memory optimization and set **CODE4.readLock** to true (non-zero), or explicitly lock the record before calling **d4go**, to ensure the new record is up to date.

See Also: **d4top**, **d4bottom**, **CODE4.errGo, CODE4.readLock, d4recCount, d4flush, code4unlockAuto**

```
/*ex45.c */
#include "d4all.h"

void main( void )
{
    CODE4  cb ;
    DATA4 *data ;
     int rc ;

     code4init( &cb ) ;
     data = d4open( &cb, "INFO" ) ;

    cb.lockAttempts = 1 ;
    cb.readLock = 1 ;

     rc =  d4go( data, 2L ) ;
    if ( rc == r4locked )
    {
       printf( "\nRecord 2 was locked by another user.") ;
       cb.readLock = 0 ; /* Turn automatic locking off.*/

       rc =  d4go( data, 2L ) ;
       if ( rc == r4locked )
       {
          printf("This will never happen because") ;
          printf("'CODE4.readLock' is false.");
       }
    }
    d4close( data ) ;
    code4initUndo( &cb ) ;
}
```

d4goBof

Usage: int d4goBof(DATA4 *data)

Description: The record number becomes one and the bof flag becomes true (non-zero).

Returns:

r4bof Success. The beginning of file flag was set.

r4locked A lock attempt, which was necessary to flush field changes, did not succeed.

r4unique The record was not written when attempting to flush because a duplicate key was encountered in a unique tag. In this case **t4unique** returns **r4unique** for the tag so an error is not generated.

< 0 Error.

Locking: The current record and the index files must be locked if flushing is required. See **d4flush** and **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

See Also: **d4flush, code4unlockAuto**

d4goEof

Usage: int d4goEof(DATA4 *data)

Description: The record number becomes one past the number of records in the data file, the record is blanked and the eof flag becomes true (non-zero).

Returns:

r4eof Success. The end of file flag was set.

r4locked A lock attempt, which was necessary to flush field changes, did not succeed.

r4unique The record was not written when attempting to flush because a duplicate key was encountered in a unique tag. In this case **t4unique** returns **r4unique** for the tag so an error is not generated.

< 0 Error.

Locking: The current record and the index files must be locked if flushing is required. See **d4flush** and **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

See Also: **d4flush, code4unlockAuto**

Example: See **d4eof**

d4index

Usage: INDEX4 *d4index(DATA4 *data, const char *indexName)

Description: A search is done to determine if the data file has an open index file under the name of *indexName*. If it does, a pointer to the corresponding **INDEX4** structure is returned. Otherwise, null is returned. **d4index** is not used to open index files.

Parameters:

indexName A null terminated character string containing the name of the index file to locate. If *indexName* is null, **d4index** uses the data file alias as the name of the index file.

See Also: **i4tag**, **d4tag**, **i4open**. Refer to **d4create** and **i4create** to create an index file.

```
/*ex46.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
     INDEX4 *index ;

     code4init( &cb ) ;
     data = d4open( &cb, "INFO" ) ;

    /* Since CODE4.autoOpen is by default true (non-zero),
       the INFO index file should have been opened.*/
    index = d4index( data, "INFO" ) ;
     if ( index != NULL )
         printf( "INDEX: INFO has been opened\n") ;

     index = d4index( data, "JUNK" ) ;
    if( index == NULL )
        printf( "INDEX: JUNK has not been opened\n") ;

    d4close( data ) ;
    code4initUndo( &cb ) ;
 }
```

d4lock

Usage: int d4lock(DATA4 *data, long recordNum)

Description: **d4lock** locks the specified record. If the current application already has locked the entire file or the specific record, **d4lock** recognizes this and does nothing except return success.

If a new lock is required, **d4lock** checks the setting of the **code4unlockAuto** function and performs the specified automatic unlocking, before performing the lock.

Note

Locking several records with **d4lock** while **code4unlockAuto** is set to **LOCK4OFF** can be used to simulate group locks. However, in the interests of avoiding deadlock, it is strongly suggested that **code4lock** be used to perform locks on multiple records.

Parameters:

recordNum This is the record number of the physical record to be locked.

Returns:

r4success Success.

r4locked The record was locked by another user. Locking did not succeed after **CODE4.lockAttempts** tries.

< 0 Error.

Locking: If successful, the specified record is locked.

See Also: **d4unlock**, **code4lock**, **CODE4.lockAttempts, code4unlockAuto, CODE4.readLock**

```
/*ex47.c*/
#include "d4all.h"
extern unsigned  _stklen =  10000 ;  /* Borland Only */

void main( void )
{
    CODE4  cb ;
    DATA4 *data ;
     int rc ;

     code4init( &cb ) ;
     data = d4open( &cb, "DATA" ) ;

    cb.lockAttempts = 4 ; /* Try lock four times */

     rc = d4lock( data, 4 ) ;
    if( rc == r4success )
       printf( "Record 4 is now locked.\n") ;
    else if( rc == r4locked )
       printf( "Record 4 is locked by another user.\n" ) ;

    cb.lockAttempts = WAIT4EVER ; /*Try forever, d4lock will not return r4locked*/
    rc = d4lock( data, 4 ) ;

     if ( rc == r4locked )
         printf( "This should never Happen\n");

    d4close( data ) ;
    code4initUndo( &cb ) ;
 }
```

d4lockAdd

Usage: int d4lockAdd(DATA4 *data, const long recordNumber)

Description: This function is used to add the specified record to the list of locks placed with the next call to **code4lock**.

Parameters: *recordNumber* contains the physical record number of the record to be placed in the queue to lock with **code4lock**.

Note

This function performs no locking. It merely places the specified record on the list of records to be locked by **code4lock**.

Returns:

r4success The specified record number was successfully placed in the **code4lock** list of pending locks.

< 0 Error. The memory required for the record lock information could not be allocated.

See Also: **code4lock, d4lockAddAppend, d4lockAddFile, d4lockAddAll**

d4lockAddAll

Usage: int d4lockAddAll(DATA4 *data)

Description: The data file, along with corresponding index and memo files, are added to the list of locks placed with the next call to **code4lock**.

Note

This function performs no locking. It merely places the specified files on the list of locks to be locked by **code4lock**.

Returns:

r4success The files were successfully placed in the **code4lock** list of pending locks.

< 0 Error.

See Also: **d4unlock**, **CODE4.lockAttempts, code4lock, code4unlockAuto**

d4lockAddAppend

Usage: int d4lockAddAppend(DATA4 *data)

Description: This function is used to add the data file's append bytes to the list of locks placed with the next call to **code4lock**.

Note

This function performs no locking. It merely places the specified record on the list of records to be locked by **code4lock**.

WARNING

When appending many records to a datafile, use **d4lockAddAll** instead of **d4lockAddAppend.** Using **d4lockAddAll** will significantly improve performance.

Returns:

r4success The data file's append bytes were successfully placed in the **code4lock** list of pending locks.

< 0 Error. The memory required for the append byte lock information could not be allocated.

See Also: **code4lock, d4lockAdd, d4lockAddFile, d4lockAddAll**

d4lockAddFile

Usage: int d4lockAddFile(DATA4 *data)

Description: This function is used to add the entire data file, including all records and the append bytes, to the list of locks placed with the next call to **code4lock**.

Note

This function performs no locking. It merely places the specified record on the list of records to be locked by **code4lock**.

WARNING

If multiple updates are being made, use **d4lockAddAll** instead of **d4lockAddFile.** Using **d4lockAddAll** will significantly improve performance.

Returns:

r4success The data file and its append bytes were successfully placed in the **code4lock** list of pending locks.

< 0 Error. The memory required for the data file and append byte lock information could not be allocated.

See Also: **code4lock, d4lockAdd, d4lockAddAppend, d4lockAddAll**

d4lockAll

Usage: int d4lockAll(DATA4 *data)

Description: The data file, along with corresponding index and memo files, are all locked. If the locking attempt fails on any of the files, everything is unlocked according to **code4unlockAuto** and **r4locked** is returned.

Note

If modifications are to be made on more than one data file, then use **d4lockAddAll** and **code4lock** to lock the files, instead of calling **d4lockAll**.

Returns:

r4success Success.

r4locked A required lock attempt did not succeed after the lock had been tried **CODE4.lockAttempts** times.

< 0 Error.

See Also: **d4unlock**, **CODE4.lockAttempts, code4lock, d4lockAddAll, code4unlockAuto**

```
/*ex48.c*/
#include "d4all.h"

void main( void )
{
    CODE4 code ;
    DATA4 *df ;
     int rc ;

     code4init( &code ) ;
     df = d4open( &code, "INFO" ) ;

    rc = d4lockAll( df ) ;

    if( rc == r4success )
        printf( "Lock is successful.\n ") ;
```

```
   code4initUndo( &code ) ;
}
```

d4lockAppend

Usage: int d4lockAppend(DATA4 *data)

Description: This function locks the append bytes. If the entire data file is locked or if the append bytes have been explicitly locked, then this function does nothing and **r4success** is returned. If the append bytes require locking, **d4lockAppend** removes any locks in accordance with the **code4unlockAuto** setting and then locks the append bytes.

While the append bytes are locked, no other application may add new records to the data file.

WARNING

When appending many records to a datafile, use **d4lockAll** instead of **d4lockAppend.** Using **d4lockAll** will significantly improve performance.

Calling this function before **d4recCount** will guarantee that **d4recCount** returns the exact number of records in the data file.

Usually, there is no need to call this function directly from application programs.

Returns:

r4success The append bytes were successfully locked.

r4locked The append bytes were locked by another user and locking did not succeed after **CODE4.lockAttempts** tries.

< 0 Error.

Locking: Once **d4lockAppend** finishes successfully, the append bytes are locked.

See Also: **code4lock, d4lock**, **d4unlock**, **CODE4.lockAttempts, d4lockAll, d4recCount**

d4lockFile

Usage: int d4lockFile(DATA4 *data)

Description: This function locks the entire data file. Locking the datafile ensures that it may not be modified by any other user.

If the data file has already been locked, this function does nothing and returns **r4success**.

WARNING

If multiple updates are being made, use **d4lockAll** instead of **d4lockFile.** Using **d4lockAll** will significantly improve performance.

Returns:

r4success The data file was successfully locked.

r4locked The file was locked by another user and locking did not succeed after **CODE4.lockAttempts** tries.

< 0 Error.

Locking: If **code4unlockAuto** is set with **LOCK4DATA** or **LOCK4OFF**, the data file will remain locked until it is explicitly unlocked or closed. On the other hand if **code4unlockAuto** is set with **LOCK4ALL**, the data file will be unlocked the next time a new lock is to be placed on a different open data file.

See Also: **code4lock, d4lock**, **d4unlock**, **CODE4.lockAttempts, code4unlockAuto**

```
/*ex49.c*/
#include "d4all.h"
extern unsigned  _stklen =  10000 ;  /* for all Borland compilers  */

void main( void )
{
    CODE4  cb ;
    DATA4 *data ;

     code4init( &cb );
     data = d4open( &cb, "INFO" ) ;

    if ( d4lockFile( data ) == r4success )
    {
       printf( "Other users can read this data file, but can make ") ;
       printf( "no modifications until this lock is removed.\n") ;
    }

    code4initUndo( &cb ) ; /* implicitly unlock the file. */
}
```

d4log

Usage: int d4log(DATA4 *data, int logging)

Description: All modifications made to the current data file, which are beyond the scope of a transaction, can be recorded in a log file, depending on the setting of **CODE4.log** when the data file was opened. **d4log** may be used to turn the logging off or on for an open data file. The log file must be explicitly opened, or implicitly opened by **d4open** or **d4create** before **d4log** may be called. **d4log** only has an effect on the logging status if the data file was opened with **CODE4.log** set to **LOG4ON** or **LOG4TRANS**. Setting **CODE4.log** to **LOG4ALWAYS,** before the data file is opened, ensures that the data file changes are always logged and can not be turned off by using **d4log**.

Changes are not logged for temporary files by default. **d4log** can be used to set the logging status to true (non-zero) for temporary files if desired.

It is useful to turn the logging off when copying data files or when a back up copy of the changes is not required. When the logging is off, the changes are made more quickly and less disk space is used.

Turning logging off with **d4log** will have no effect on logging during a transaction. See **code4tranStart** and **code4tranCommit** for more details about transactions.

Note that **d4log** will neither create, open nor close log files; it merely starts and stops the recording process.

Parameters:

logging This is the value to which the logging status is set. The possible settings are:

0 Do not record future changes for the current data file in the log file.

Non-Zero Record all future changes for the current data file in the log file.

Returns: The previous setting of the logging status is returned. **r4logOn** is returned if an attempt is made to turn off the logging when **CODE4.log** is set to **LOG4ALWAYS**.

Client-Server: This function does not have an effect in the client-server configuration.

See Also: **code4logCreate**, **code4logFileName**, **code4logOpen, CODE4.log**

d4logStatus

Usage: int d4logStatus(DATA4 *data)

Description: This function returns the current logging status.

Returns: The setting of the logging status.

0 Does not record future changes of the current data file in the log file.

Non-Zero Records all future changes of the current data file in the log file.

See Also: **code4logCreate**, **code4logFileName**, **code4logOpen, d4log, CODE4.log**

d4memoCompress

Usage: int d4memoCompress(DATA4 *data)

Description: The memo file corresponding to the data file is compressed. If the data file has no memo file, nothing happens.

A call to **d4memoCompress** may be desirable after packing or zapping a data file. This is because these functions do not remove memo entries. The unreferenced memo entries do no harm except waste disk space. When memo files entries are reduced in size, the disk space previously occupied by the entries becomes available for reuse by the memo file. However, the disk space is not necessarily returned to the operating system. This function compresses the memo file to return the disk space to the operating system.

d4memoCompress first makes a temporary memo file in the current directory with the same name as the original memo file but with a "**.TMP**"

extension. The original memo file is then compressed into this file. Finally, when **S4OFF_MULTI** is defined, the original memo file is deleted and the newly created memo file is renamed to the original name. However, in the multi-user case, the contents of the new memo file are copied back into the original file, the file size is shrunk, and the temporary file is deleted.

WARNING

Appropriate backup measures should be taken before calling this function. It does not make a permanent backup file. However, if the function fails for whatever reason, either the original memo file or the temporary file, containing the newly compressed memo file contents, should be present.

It is recommended that data files be opened exclusively (i.e. setting **CODE4.accessMode** to **OPEN4DENY_RW** before opening the datafile) before the memos are compressed. This ensures that other users cannot access the memo file while it is being compressed. Applications that access memo files that have not fully been compressed may generate errors or read random data.

WARNING

This function may not be used to compress a memo file while a transaction is in progress. Attempts to do so will fail and generate an **e4transViolation** error.

Returns:

r4success	Success.
r4locked	The data file is already locked by another user.
r4unique	This value is returned when a record flush causes a duplicate key in a unique tag, and **t4unique** returned **r4unique** for the tag.
< 0	Error.

Locking: The data file and corresponding memo file must be locked before the memo file can be compressed. See **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

See Also: **d4pack**, **d4zap, CODE4.accessMode, code4unlockAuto**

```
/*ex50.c*/
#include "d4all.h"
void compressAll( DATA4 *d )
{
    if( d4pack( d ) == r4success )
       printf( "Records marked for deletion are removed\n" ) ;
    if( d4memoCompress( d ) == r4success )
       printf( "Memo entries are compressed\n") ;
 }
void main ()
{
    CODE4 cb ;
    DATA4 *data ;

    code4init( &cb ) ;
    data = d4open( &cb, "DATA" ) ;
    compressAll( data ) ;
    code4initUndo( &cb ) ;
}
```

d4numFields

Usage: int d4numFields(DATA4 *data)

Description: The number of fields in the data file is returned. If the **DATA4** structure is invalid, a negative value is returned. This function, when used with **d4fieldJ,** is useful for writing general data file utilities.

Returns:

>= 0 This is the number of fields in the data file.

< 0 Error.

See Also: **d4fieldJ**

```
/*ex51.c*/
#include "d4all.h"

void main(  )
{
    CODE4 cb ;
    DATA4 *data ;
     int rc, fieldNum ;

     code4init( &cb ) ;
     data = d4open( &cb, "INFO" ) ;

    code4optStart( &cb ) ;

    for( rc = d4top( data ); rc != r4eof ; rc = d4skip( data, 1L ) )
    {
         printf( "\n" ) ;

       for( fieldNum = 1 ; fieldNum <= d4numFields( data ) ; fieldNum++ )
          printf( " %s \n", d4fieldJ(data, fieldNum)) ;
    }
    code4initUndo( &cb ) ;
 }
```

d4open

Usage: DATA4 *d4open(CODE4 *code, const char *name)

Description: Function **d4open** opens a data file and its corresponding memo file (if applicable). Finally, if **CODE4.autoOpen** is true (non-zero), any "production index file" corresponding to the data file is opened.

Under FoxPro and dBASE IV compatibility, a production index file is an index file created at the same time as the data file, using **d4create**. It can also be created by using **i4create** with a null name parameter. When a production index file is automatically opened, no tag is initially selected.

When an index is created with FoxPro or dBASE IV, it is automatically created as a production index file.

When Clipper index files are being used, and if **CODE4.autoOpen** is true (non-zero), **d4open** assumes that there is a corresponding group file and attempts to open it (refer to the User's Guide for more details about group files). Consequently, when using Clipper, it is often appropriate to set **CODE4.autoOpen** to false (zero). Doing so avoids an **e4open** error message saying that the "**.CGP**" file is not present.

Listed below are the default file extensions for the different compatibilities. If an extension is not provided to **d4open**, the default extensions are used.

	Data	Index	Memo
dBASE IV	".DBF"	".MDX"	".DBT"
Clipper	".DBF"	".CGP"	".DBT"
FoxPro	".DBF"	".CDX"	".FPT"

Note

In the client-server configuration, **code4connect** will automatically be called if connection to the server has not been previously established.

WARNING

d4open does not leave the data file at a valid record. Call a positioning statement such as **d4top** to move to a valid record. It is inappropriate to call **d4skip** until a valid record is loaded into the record buffer.

Parameters:

code This is a pointer to a **CODE4** structure, which was initialized by a call to **code4init**. This pointer is saved in the **DATA4** structure so that all data file functions have access to the settings and information contained in the **CODE4** structure.

name The is the name of the data file to open. If a path is provided in the single user or multi-user configuration, it is used. Otherwise, the file is assumed to be in the current directory.

When using the client-server configuration, refer to the Catalog file documentation in the CodeServer manual for the details on how to specify the file name.

The default alias for the data file is initially set to *name*, disregarding path and extension.

Note

Remember that in the C programming language, the backslash '\' is an escape character (eg. '\n' is the code for newline). Therefore, use a double backslash '\\' to represent a backslash in a static string specifying a path.

Returns:

Not Null Success. A pointer to a **DATA4** structure, corresponding to the opened data file, is returned.

Null The data file was not opened. The problem could be with either the data, index or the memo file. Use **CODE4.errorCode** to determine the error.

If one of the files does not exist and **CODE4.errOpen** was false, then **CODE4.errorCode** is set to **r4noOpen**. Due to the setting of **CODE4.errOpen**, this is not considered an error.

See Also: **code4close**, **code4connect**, **code4init**, **code4logCreate**, **code4logOpen**, **code4logOpenOff**, **CODE4.optimize**, **CODE4.accessMode**, **CODE4.readOnly**, **d4close**, **d4log**

```
/*ex52.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ;

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;

     code4init( &cb ) ;
    cb.accessMode = OPEN4DENY_RW ;

    data = d4open( &cb, "INFO" ) ;

    if( data != NULL )
       printf( "Data file \\INFO.DBF has %d records\n", d4recCount( data )) ;
     d4close( data ) ;
    code4initUndo( &cb ) ;
}
```

d4openClone

Usage: DATA4 *d4openClone(DATA4 *data)

Description: This function opens a data file that is already open. This function ignores the **CODE4.singleOpen** flag thus allowing a file to be opened more than once within the same process.

Parameters:

data This is a pointer to a **DATA4** structure that corresponds to a previously opened data file.

Returns:

Not Null Success, a pointer to a **DATA4** structure, which corresponds to the opened data file,is returned.

Null Error. The data file was not opened. The problem could be with either the data, index or the memo file. Check the **CODE4.errorCode** for details about the error.

See Also: **d4open**

d4optimize

Usage: int d4optimize(DATA4 *data, int optFlag)

Description: This function sets the memory optimization strategy for the data file and corresponding memo and index files.

Memory optimization does not actually take place until **code4optStart** is called. If **code4optStart** has been called, the file's memory optimizations are effective once the file is flagged as memory optimized. This occurs when the file is opened or after a call to **d4optimize**.

If read optimization is turned off, then write optimization is also automatically turned off. If read optimization is turned on, then the write optimization strategy becomes the default as defined by **CODE4.optimizeWrite**.

d4optimize is not present, if the library was built with the **S4OFF_OPTIMIZE** switch defined.

Initially, files are optimized according to the status of the **CODE4.optimize** and **CODE4.optimizeWrite** values at the time the file is opened.

WARNING

Use memory optimization on shared files with caution When using memory read optimization on shared files, it is possible for inconsistent data to be returned if another application is updating the data file. This means that any data returned from the memory optimized file could potentially be out of date.

Parameters: Possible choices for parameter *optFlag* are as follows:

OPT4EXCLUSIVE Read-optimize when files are opened exclusively, when the **S4OFF_MULTI** compilation switch is defined, or when the DOS read-only attribute is set for the file. Otherwise, do not read optimize. This is the default value.

Note

Note that there is a distinction between a DOS read-only attribute and a network read file permission. A DOS read-only attribute means that no one may write to the file, while a network permission setting for any given file may be different from user to user.

OPT4OFF Do not read optimize.

OPT4ALL This is the same as the **OPT4EXCLUSIVE** option except shared files are also read optimized.

Returns:

r4success Success.

< 0 Error. Flushing failed when optimization was disabled, or **CODE4.errorCode** contained a negative value.

Client-Server: In the client-server configuration, the optimization is controlled at the server level, so the function **d4optimize** always returns success.

See Also: **file4optimize**, **d4optimizeWrite**

```
/*ex53.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    DATA4 *data, *extra ;

    code4init( &cb ) ;
    cb.accessMode = OPEN4DENY_RW ;

    data = d4open(  &cb, "INFO" ) ;
```

```
   /* Open the file exclusively, default optimization is the same as if
        d4optimize( data, OPT4EXCLUSIVE ) were called.*/
    /* open a shared file. */
   cb.accessMode = OPEN4DENY_NONE ;
    extra = d4open( &cb, "DATA" ) ;

   d4optimize( extra, OPT4ALL ) ;  /* read optimize the extra "DATA" file */

   code4optStart( &cb ) ;  /* Begin the memory optimizations.*/

   /* .... Some other code .... */

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

d4optimizeWrite

Usage: int d4optimizeWrite(DATA4 *data, int optFlag)

Description: This function sets the memory write optimization strategy for the data file and corresponding memo and index files.

Memory optimization does not actually take place until **code4optStart** has been called. If **code4optStart** has already been called, the file's memory optimizations are effective immediately once it is flagged as a memory optimized file.

The initial write optimization strategy is set when the file is opened, according to the value in flag **CODE4.optimizeWrite**. If write optimization is turned on, then memory optimization, as defined by the **CODE4.optimize** switch, is also turned on.

WARNING

Use write optimization on shared files with extreme care, because write optimized files can return inconsistent data, since parts of a file may not be updated immediately due to the optimization procedures. For shared files, write optimization never actually takes place until the file is locked.

Write optimization does not improve performance unless the entire data file is locked over a number of operations. Write optimization would improve performance, for example, when appending many records at once.

Parameters: The possible values for optFlag are:

OPT4EXCLUSIVE — Write-optimize when files are opened exclusively or when the **S4OFF_MULTI** compilation switch is defined. Otherwise, do not write optimize. This is the default value when **d4optimizeWrite** is not called.

OPT4OFF — Never write optimize.

OPT4ALL — This is the same as the **OPT4EXCLUSIVE** option except shared files, which are locked, are also write optimized. Use this option with care. If concurrently running applications do not lock, they may be presented with inconsistent data.

Returns:

r4success Success.

< 0 An error occurred.

Client-Server: In the client-server configuration, the optimization is controlled at the server level, so the function **d4optimizeWrite** always returns success.

See Also: **file4optimizeWrite**, **d4optimize**, **CODE4.optimizeWrite**

```
/*ex54.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ;  /* for all Borland compilers*/

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
     FIELD4 *ageField ;
     long age ;

     code4init( &cb ) ;
     data = d4open( &cb, "INFO" ) ;

    d4optimizeWrite( data, OPT4ALL ) ;
    /* when doing write optimization on shared files, it is necessary to
         lock the file, preferably with d4lockAll( ) */
    d4lockAll( data ) ;

    code4optStart( &cb ) ; /* begin optimization */

    age = 20 ;
    ageField = d4field( data, "AGE") ;

    /* append a copies of the first record, assigning the age field's
       value from 20 to 65*/
    for( d4top( data ) ;  age < 65 ; d4append( data ) )
    {
       d4appendStart( data, 0 ) ;
       f4assignLong( ageField, age++ ) ;
     }

    code4initUndo( &cb ) ; /* flushes, closes, and unlocks */
}
```

d4pack

Usage: int d4pack(DATA4 *data)

Description: **d4pack** physically removes all records marked for deletion from the data file. **d4pack** automatically reindexes all open index files attached to the data file.

After **d4pack** completes, the contents of the record buffer and the record number are undefined. Explicitly call a positioning function such as **d4top** to position to a desired record.

Appropriate backup measures should be taken before calling this function.

d4pack does not alter the memo file. Consequently, memo entries referenced by removed records will be wasted disk space. Explicitly call **d4memoCompress** to compress the memo file. Alternatively, you could explicitly remove memo entries when records are marked for deletion.

Consider opening the data file exclusively (set **CODE4.accessMode** to **OPEN4DENY_RW** before opening the file) before packing, since **d4pack** can seriously interfere with the data retrieved by other users.

WARNING

This member function may not be used to remove records from a data file while a transaction is in progress. Attempts to do so will fail and generate an **e4transViolation** error.

Returns:

r4success Success.

r4locked A required lock did not succeed.

r4unique An index file could not be rebuilt because doing so resulted in a non-unique key in an unique-key tag. In this case **t4unique** returns **r4unique** for the tag so an error is not generated. The records marked for deletion are removed, but the index file(s) are out of date. Refer to **CODE4.errDefaultUnique**.

< 0 Error

Locking: The data file and its index files must be locked. See **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

See Also: **d4delete**, **d4recall**, **d4memoCompress, code4unlockAuto**

```
/*ex55.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* Borland compilers only  */

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;

     code4init( &cb ) ;
    cb.accessMode = OPEN4DENY_RW ; /* open file exclusively*/

    data = d4open( &cb, "INFO" ) ;
    error4exitTest( &cb ) ;
    code4optStart( &cb ) ;

    for( d4top( data ); ! d4eof( data ); d4skip( data, 2 ) )
       d4delete( data ) ;       /* Mark every other record for deletion*/

      /* Remove the deleted records from the physical disk */
    d4pack( data ) ;

    d4close( data ) ;
    code4initUndo( &cb ) ;
 }
```

d4position

Usage: double d4position(DATA4 *data)

Description: This function is the inverse of **d4positionSet**. **d4position** returns an estimate of the current position in the data file as a **(double)** percentage. For example, if the current position is half way down the data file, **(double)** .5 is returned.

Both **d4positionSet** and **d4position** use the currently selected tag to specify the ordering. If no tag is selected, record number ordering is used.

The purpose of **d4position** and **d4positionSet** is to facilitate the use of scroll bars when developing edit and browse functions. However, due to the nature of the function, it may return inaccurate results when used with a selected tag. The inaccuracy may be reduced by occasionally reindexing and becomes less apparent in larger data files.

Returns:

>= 0 The current position in the data file represented as a percentage. This function returns **(double)** 1.1 if the end of file condition is true, and **(double)** 0.0 if the beginning of file condition is true or if the data file is empty.

< 0 Error. A return of a negative number indicates that there was a problem determining or setting the position. This could be an error return or it could indicate that a tag was already locked by another user. Note that **CODE4.errorCode** can be examined to determine if the return is an error.

```
/*ex56.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
     TAG4 *tag ;

     code4init( &cb ) ;
     data = d4open( &cb, "INFO" ) ;
    tag = d4tagDefault( data ) ;

    error4exitTest( &cb ) ;
    d4tagSelect( data, tag ) ; /* select the default tag. */
    d4positionSet( data, .25 ) ; /* move one quarter through the index file.*/
    printf( "Record number: %d \n", d4recNo( data )) ;

    printf( "The current position is: %f \n", d4position( data )) ;

    code4initUndo( &cb ) ;
 }
```

d4positionSet

Usage: int d4positionSet(DATA4 *data, double pos)

Description: When **d4positionSet** is called, *pos* is taken as a percentage, and the record closest to that percentage becomes the current record. (ie. the record is loaded into the record buffer, and its record number is used as the current record number). Both **d4positionSet** and **d4position** use the currently selected tag to specify the ordering. If no tag is selected, record number ordering is used.

If **d4positionSet** cannot find a record at the precise location specified by *pos*, the next record in sequence is used. For example, if a data file has three records and **d4positionSet(** data, .25 **)** is called the second record (which is actually at .5) is used instead. In this type of case, **d4position** does not return the same value as passed to **d4positionSet**.

When **d4positionSet** is used with a selected tag, it may not position to the exact position within the tag. This is due to the nature of the index file

structure. The inaccuracy may be reduced by occasionally reindexing and becomes less apparent in larger data files.

Parameters:

pos — This percentage indicates which record to make current.

Returns:

r4success — The position was successfully set.

r4entry — Positioning was done with a tag and there was no corresponding data file entry. In addition, **CODE4.errGo** must be false (zero). Note that this implies that the index file is out of date.

r4eof — Either there were no records in the data file or the *pos* was greater than **(double)**1.0.

r4locked — A required lock attempt did not succeed. The lock was attempted for either flushing changes to disk or for reading a record (as required when **CODE4.readLock** is true (non-zero)).

r4unique — This value is returned when a record flush causes a duplicate key in a unique tag, and **t4unique** returned **r4unique** for the tag.

< 0 — Error. A return of a negative number indicates that there was a problem determining or setting the position. Note that **CODE4.errorCode** can be examined to determine if the return is an error.

Locking: If record buffer flushing is required, the record and index files must be locked to accomplish this task. If **CODE4.readLock** is true (non-zero), the new record is locked before it is read. See **d4flush** and **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

See Also: **d4flush, code4unlockAuto**

d4recall

Usage: void d4recall(DATA4 *data)

Description: If the current record is marked for deletion, the mark is removed.

This is done by changing the first byte of the record buffer from an '*' to a ' ' character. In addition, the record buffer is flagged as being changed. Consequently, when other data file functions are called, the change to the record is flushed to disk before a new record is read.

See Also: **d4delete**, **d4deleted**, **d4pack**

```
/*ex57.c*/
#include "d4all.h"

long recallAll( DATA4 *d, int lockTries )
{
    TAG4 *saveSelected ;
     long count ;

     saveSelected = d4tagSelected( d ) ;
    d4tagSelect( d, NULL ) ; /* use record ordering */

    d4lockAll( d ) ;
```

```
    count = 0 ;

    for( d4top( d ) ;  !d4eof( d ) ; d4skip( d, 1 ) )
    {
       d4recall( d ) ;
       count++ ;
    }

    d4tagSelect( d, saveSelected ) ; /* reset the selected tag.*/
    return count ;
}
void main()
{
     CODE4 cb ;
     DATA4 *data ;
     int lockTries = WAIT4EVER ;
     long count  ;

     code4init( &cb ) ;
     data = d4open( &cb , "INFO" ) ;
     count = recal lAll( data, lockTries ) ;
     printf( "record count is %d \n", count ) ;
     code4initUndo( &cb ) ;
}
```

d4recCount

Usage: long d4recCount(DATA4 *data)

Description: The number of records in the data file is returned.

If the data file is shared, the record count might not reflect the most recent additions made by other users. An accurate count can be obtained by manually locking the append bytes prior to calling **d4recCount**, since no other user can modify the number of records in the data file.

Returns:

>= 0 The number of records in the data file.

< 0 Error.

See Also: **d4recNo, d4lockAppend, d4lockAddAppend**

```
/*ex58.c*/
#include "d4all.h"

long recsInFile( CODE4 *cb, DATA4 *d )
{
     long count ;

    if( cb->errorCode ) return -1L ; /* an error occurred */

     count = d4recCount( d ) ; /* save the record count */

    return count ;
}

void main( )
{
     CODE4 cb ;
     DATA4 *data ;
     long count ;

     code4init( & cb ) ;
     data = d4open( &cb, "INFO" ) ;
     count = recsInFile( &cb, data ) ;
     printf( "the number of records in the file is %d \n", count ) ;
     code4initUndo( &cb ) ;
}
```

d4recNo

Usage: long d4recNo(DATA4 *data)

Description: The current record number of the data file is returned. If the record number returned is greater than the number of records in the data file, this indicates an end of file condition.

Returns:

>= 1	The current record number.
0	There is no current record number. **d4appendStart** has just been called to start appending a record.
-1	There is no current record number. The file has just been opened, created, packed or zapped. -1 is also returned when the **DATA4** structure is invalid.
< -1	Error.

See Also: **d4recCount**

```
/*ex59.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
     TAG4 *tag ;
     long count = 0 ;

     code4init( &cb ) ;
     data = d4open( &cb, "INFO" ) ;
     tag = d4tagDefault( data ) ;

    d4tagSelect( data, tag  ) ;  /* select the default tag*/

    for( d4top( data ); !d4eof( data ) ; d4skip( data, 1 ) )
    {
       printf( "Tag position: %d\n", ++count ) ;
       printf( "      Record Position: %d\n",d4recNo( data ) ) ;
    }
    code4initUndo( &cb ) ;
}
```

d4record

Usage: char * d4record(DATA4 *data)

Description: **d4record** returns a pointer to the null terminated record buffer of the data file. This pointer allows you to access the record directly.

WARNING

Unless you are an expert and know exactly what you are doing, it is best to use the field functions to manipulate the record buffer.

See Also: **d4recWidth**

Example: See **d4recWidth**

d4recWidth

Usage: unsigned int d4recWidth(DATA4 *data)

Description: The width of the internal record buffer is returned. This value includes the deleted flag of the record, but does not include the record buffer's null terminator.

Returns:

> 0 The length of the record buffer.

0 An error has occurred in determining the length of the record buffer.

See Also: **d4record**

```
/*ex60.c  Copy records from one database to another. */
#include "d4all.h"

void main( void )
{
    CODE4  cb ;
    DATA4 *from, *to ;
     long iRec ;

     code4init( &cb ) ;
     from = d4open ( &cb, "DATA1" ) ;
    code4optStart( &cb ) ;

    to = d4open( &cb, "DATA2" ) ;
     /* Database 'DATA2' has an identical database
        structure as database 'DATA1'. */

    if ( d4recWidth( from ) != d4recWidth( to ) )
    {
        error4( &cb, e4result, 0 ) ;
       code4exit( &cb ) ;
    }

    error4exitTest( &cb ) ;
    for ( iRec = 1L; iRec <= d4recCount( from ); iRec++ )
    {
       /* Read the database record. */
       d4go( from, iRec ) ;
        /* Copy the database buffer. */
       d4appendStart( to, 0 ) ;
       memcpy( d4record( to ), d4record( from ), d4recWidth( to ) ) ;
        /* Append the database record. */
       d4append( to ) ;
    }
    code4close( &cb ) ;
    code4initUndo( &cb ) ;
}
```

d4refresh

Usage: int d4refresh(DATA4 *data)

Description: If memory optimization is being used, all buffered information for the data file and its corresponding index and memo files are flushed to disk and then discarded from memory.

Effectively, this 'refreshes' the information because the next time the information is accessed, it must be read directly from disk.

Note

There is no point to calling this function in single-user applications (**S4OFF_MULTI**), in client-server applications (**S4CLIENT**), or if memory optimization is not being used (**S4OFF_OPTIMIZE**). In addition, calling this function regularly defeats the purpose of memory optimization. If this function is needed frequently, remove all memory optimizations, and calls to **d4refresh** will be unnecessary.

Returns:

r4success Success.

< 0 Error.

See Also: **d4refreshRecord**

```
/*ex61.c*/
#include "d4all.h"

void main( void )
{
    CODE4 codeBase ;
    DATA4 *dataFile ;

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

    d4optimize( dataFile, OPT4ALL ) ;
    code4optStart( &codeBase ) ;

    d4top( dataFile ) ;
    printf( "Press ENTER when you want to refresh your data.\n") ;
    getchar( ) ;

    d4refresh( dataFile ) ;
    d4top( dataFile ) ;
  /* re-read the record from disk. */
    printf( "The latest information is: %s \n",d4record( dataFile ) ) ;

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

d4refreshRecord

Usage: int d4refreshRecord(DATA4 *data)

Description: This function refreshes the current record, directly from disk, into the record buffer. In addition, the 'record changed' flag is reset. Consequently, any recent changes to the current record buffer are not flushed.

This function may be used to update a record from disk when another application may have changed it.

Returns:

r4success Success.

< 0 Error.

See Also: **d4refresh**

```
/*ex62.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void update( DATA4 *d )
{
    if( d4changed( d, -1 ) )
       printf( "Changes not discarded.\n") ;
    else
       d4refreshRecord( d ) ;
}
void main()
{
    CODE4 cb ;
    DATA4 *data ;

    code4init( &cb ) ;
```

```
    data = d4open( &cb, "DATA") ;
    d4top( data ) ;
    update( data ) ;
    code4initUndo( &cb ) ;
}
```

d4reindex

Usage: int d4reindex(DATA4 *data)

Description: All of the index files corresponding to the data file are recreated. It is generally a good idea to open the files exclusively (set **CODE4.accessMode** to **OPEN4DENY_RW** before opening the data file) before reindexing.

After **d4reindex** completes, the contents of the record buffer and the record number are undefined. Explicitly call a positioning function such as **d4top** to position to a desired record.

Returns:

r4success Success.

r4unique A unique key tag has a repeated key and **t4unique** returns **r4unique** for that tag.

r4locked A required lock attempt did not succeed.

< 0 Error.

Locking: The data file and corresponding index files are locked. It is recommended that the index files be opened exclusively if a reindex is to occur. This will ensure that other users cannot access the file while it is being reindexed. Applications that access index files that have not been fully reindexed may generate errors or obtain incorrect database information.

See Also: **i4create**, **d4check, i4reindex**

```
/*ex63.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;

    code4init( &cb ) ;
    cb.accessMode = OPEN4DENY_RW ;
    data = d4open( &cb, "INFO" ) ;

    printf( "Reindexing %s Please Wait\n", d4alias( data ) ) ;
    if( d4reindex( data ) != 0 )
       printf( "Reindex NOT successful.\n ") ;
    else
       printf( "Successfully reindexed.\n") ;
    code4initUndo( &cb ) ;
}
```

d4remove

Usage: int d4remove(DATA4 *data)

Description: This member function permanently removes the data file, its associated index and memo files, from disk.

WARNING

This member function may not be used to delete a data file while a transaction is in progress. Attempts to do so fail and generate an **e4transViolation** error.

WARNING

This member function irrevocably removes the database from disk. If the database contains useful information, consider performing a backup on the database prior to calling **d4remove**.

Returns:

r4success The data file and any open index and memo files associated with the data file are deleted from disk.

< 0 An error occurred removing the files from disk.

Client-Server: This function will also remove all references to the data file contained in all the system tables held by the server (eg. catalog file, table authorization file).

d4seek

Usage: int d4seek(DATA4 *data, const char *ptr)

Description: Function **d4seek** searches using the default tag, which is the currently selected tag if one is selected (See **d4tagDefault**). Once the search value is located in the tag, the corresponding data file record is read into the record buffer. Searching always begins from the first logical record in the database as determined by the selected tag.

d4seek may be used with any tag type, as long as it is formatted correctly. Seeking on memo fields is not allowed.

Parameters:

ptr *ptr* points to a character array containing the value for which the search is conducted.

If the tag is of type Date, the date search value should be formatted "CCYYMMDD" (Century, Year, Month, Day).

If the tag is of type Character, the *ptr* may have fewer characters than the tag's key length, provided that the search value is null terminated. In this case, a search is done for the first key which matches the supplied characters. If *ptr* points to data longer than the tag key length, the extra characters are ignored.

If the tag is of type Numeric, the character value should represent a valid number. eg. d4seek(db, "33.7"). This number is internally converted to a double value before the seek is performed.

Returns:

r4success Success. The key was found and the record was positioned.

r4after The search value was not found. The data file is positioned to the record after the position where the search key would have been located if it had existed.

r4eof The search value was not found and the search value was greater than the last key of the tag. Consequently, **d4goEof** was called to turn the EOF condition on.

r4entry **CODE4.errGo** is false (zero) and the data file record did not exist.

r4locked A required lock attempt did not succeed. The lock was required either for flushing changes to disk or to lock the found record as required when **CODE4.readLock** is true (non-zero).

r4unique This value is returned when a record flush causes a duplicate key in a unique tag, and **t4unique** returned **r4unique** for the tag.

r4noTag The seek could not be accomplished because no tag exists for the data file.

< 0 Error.

Locking: If record buffer flushing is required, the record and index files are locked to accomplish this task. If **CODE4.readLock** is true (non-zero), the new record is locked before it is read. See **d4flush** and **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

See Also: **d4tagSelect, d4seekDouble, d4seekN, d4seekNext, d4seekNextDouble, d4seekNextN, CODE4.readLock, code4unlockAuto, d4flush**

```
/*ex64.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    DATA4 *people ;
     FIELD4 *age, *birth ;
     int rc ;
     char result[12] ;

     code4init( &cb ) ;
     people = d4open( &cb, "people.dbf" ) ;
    /* Assume 'PEOPLE.DBF' has a production index file with tags
       NAME_TAG, AGE_TAG, BIRTH_TAG */

    d4tagSelect( people, d4tag( people, "NAME_TAG" ) ) ;

   if( d4seek( people,  "fred" ) == r4success )
       printf("fred is in record # %d\n",d4recNo( people )) ;

   if( d4seek( people, "HANK STEVENS" ) == r4success )
       printf( "HANK STEVENS is in record # %d\n",d4recNo( people )) ;

    d4tagSelect( people, d4tag( people, "AGE_TAG" ) ) ;
    age = d4field( people, "AGE" ) ;

    rc = d4seekDouble( people, 0.0 ) ;

    if( rc == r4success || rc == r4after )
       printf( "The youngest age is: %d\n", f4int( age )) ;

    /* Seek using the char * version */
    rc = d4seek( people, "0" ) ;

    if( rc == r4success || rc == r4after )
          printf( "The youngest age is: %d\n", f4int( age )) ;

    /* Assume BIRTH_TAG is a Date key expression */
```

```
    d4tagSelect( people, d4tag( people, "BIRTH_TAG" )) ;
    birth = d4field( people, "BIRTH" ) ;
     date4format( "19600415", result, "MMM DD, CCYY" );

     if( d4seek( people, "19600415") == r4success)
                                /* Char. array in CCYYMMDD format*/
         printf( "Found: %s\n", result ) ;

    if( d4seekDouble( people, date4long( "19600415" )) == r4success )
       printf( "Found: %s\n", result ) ;

    d4close( people ) ;
    code4initUndo( &cb ) ;
}
```

d4seekDouble

Usage: int d4seekDouble(DATA4 *data, double d)

Description: Function **d4seekDouble** searches using the default tag which is the currently selected tag if one is selected (See **d4tagDefault**). Once the search value is located in the tag, the corresponding data file record is read into the record buffer. Searching always begins from the first logical record in the database as determined by the selected tag.

In order to use **d4seekDouble,** the tag key must be of type Numeric or Date.

Parameters:

d *d* is a (**double)** value used to seek in numeric or date keys. If the key type is of type Date, *d* should represent a Julian day.

Returns: Refer to **d4seek** for the possible return values.

Locking: Refer to **d4seek** for locking procedures.

See Also: **d4tagSelect, d4seek**

d4seekN

Usage: int d4seekN(DATA4 *data, const char *ptr, int len)

Description: Function **d4seekN** searches using the default tag which is the currently selected tag if one is selected (See **d4tagDefault**). Once the search value is located in the tag, the corresponding data file record is read into the record buffer. Searching always begins from the first logical record in the database as determined by the selected tag.

If a character field is composed of binary data, **d4seekN** may be used with to seek without regard for nulls because the length of the key is specified. Seeking on memo fields is not allowed.

Parameters:

ptr *ptr* points to a character array containing the value for which the search is conducted. See **d4seek** for more details.

len This is the length of the data pointed to by *ptr*. This should be less than or equal to the length of the key size for the selected tag. If *len* is greater than the tag key length, then the extra characters are ignored. If *len* less than

zero, then *len* is treated as though it is equal to zero. If *len* is greater than the key length, then *len* is treated as though it is equal to the key length.

ptr can point to null characters and still remain a valid search key, since the *len* specifies the length of data pointed to by *ptr*.

Returns: Refer to **d4seek** for the possible return values.

Locking: Refer to **d4seek** for locking procedures.

See Also: **d4tagSelect, d4seek**

d4seekNext

Usage: int d4seekNext(DATA4 *data, const char *ptr)

Description: **d4seekNext** function differs from **d4seek**, in that the current position within the tag is used, instead of the beginning of the tag. This provides the capability of performing an incremental search through the database.

If the index key at the current position in the tag is not equal to that of the search key, **d4seekNext** calls **d4seek** to find the first occurance of the search key in the tag. Should the seek fail to find a match, **r4after** is returned.

If the index key at the current position in the tag is equal to the search key then **d4seekNext** tries to find the next occurrance of the search key. If **d4seekNext** fails to find an index key that matches the search key, **r4entry** is returned and the data file is positioned to the record after the position where the search key would have been located if it had existed.

d4seekNext may be used with any tag type, as long as it is formatted correctly. (e.g. **d4seekNext(** dataptr, " 123.44" **)** for seeking on a numeric tag). Seeking on memo fields is not allowed.

Parameters:

ptr *ptr* points to a character array containing the value for which the search is conducted. See **d4seek** for more details.

Returns:

r4success Success. The key was found and the record was positioned.

r4after This value is returned when there is no index key in the tag that matches the search value. The data file is positioned to the record after.

r4eof The search value was not found and the search value was greater than the last key of the tag. Consequently, **d4goEof** was called to turn the EOF condition on.

r4entry **CODE4.errGo** is false (zero) and the data file record did not exist. This value is also returned when the seek fails to find the next occurrance of the search key. The data file is positioned to the record after.

r4locked A required lock attempt did not succeed. The lock was required either for flushing changes to disk or to lock the found record as required when **CODE4.readLock** is true (non-zero).

r4unique This value is returned when a record flush causes a duplicate key in a unique tag, and **t4unique** returned **r4unique** for the tag.

r4noTag The seek could not be accomplished because no tag exists for the data file.

< 0 Error.

Locking: If record buffer flushing is required, the record and index files are locked to accomplish this task. If **CODE4.readLock** is true (non-zero), the new record is locked before it is read. See **d4flush** and **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

See Also: **d4seek, d4tagSelect**

```
/*ex65.c*/
#include "d4all.h"

int SeekSeries( DATA4 *d, const char *s )
{
   int rc ;

   rc = d4seekNext( d, s ) ;

   if( rc == r4noTag || rc == r4entry || rc == r4locked || rc < 0 )
      return rc ;

   if( rc == r4after || rc == r4eof )
      rc = d4seek( d, s ) ;

    printf( " the found record %s \n", d4record( d ) ) ;
   return rc ;
}
void main()
{
    CODE4 cb ;
    DATA4 *data ;
    TAG4 *nameTag ;
    int rc ;

    code4init( &cb ) ;
    data = d4open( &cb, "PEOPLE" ) ;
    nameTag  = d4tag( data, "NAME_TAG" ) ;
    d4tagSelect( data, nameTag ) ;
    d4seek( data, "mickey") ;
    rc = SeekSeries( data, "mickey" ) ;
    code4initUndo( &cb ) ;
}
```

d4seekNextDouble

Usage: int d4seekNextDouble(DATA4 *data, double d)

Description: **d4seekNextDouble** function differs from **d4seekDouble**, in that the current position within the tag is used, instead of the beginning of the tag. This provides the capability of performing an incremental search through the database.

If the index key at the current position in the tag is not equal to that of the search key, **d4seekNextDouble** calls **d4seekDouble** to find the first occurance of the search key in the tag. Should the seek fail to find a match, **r4after** is returned.

If the index key at the current position in the tag is equal to the search key then **d4seekNextDouble** tries to find the next occurrance of the search key. If **d4seekNextDouble** fails to find an index key that matches the search key, **r4entry** is returned and the data file is positioned to the record

after the position where the search key would have been located if it had existed.

In order to use **d4seekNextDouble**, the tag key must be of type Numeric or Date.

Parameters:

d *d* is a double value used to seek in numeric or date keys. If the key type is of type Date, *d* should represent a Julian day.

Returns: Refer to **d4seekNext** for the possible return values.

Locking: Refer to **d4seekNext** for locking procedures.

See Also: **d4seekDouble, d4tagSelect, d4seekNext**

d4seekNextN

Usage: int d4seekNextN(DATA4 *data, const char *ptr, int len)

Description: **d4seekNextN** function differs from **d4seekN**, in that the current position within the tag is used, instead of the beginning of the tag. This provides the capability of performing an incremental search through the database.

If the index key at the current position in the tag is not equal to that of the search key, **d4seekNextN** calls **d4seekN** to find the first occurance of the search key in the tag. Should the seek fail to find a match, **r4after** is returned.

If the index key at the current position in the tag is equal to the search key then **d4seekNextN** tries to find the next occurrance of the search key. If **d4seekNextN** fails to find an index key that matches the search key, **r4entry** is returned and the data file is positioned to the record after the position where the search key would have been located if it had existed.

d4seekNextN may be used to seek without regard to nulls. Seeking on memo fields is not allowed.

Parameters:

ptr *ptr* points to a character array containing the value for which the search is conducted. See **d4seek** for more details.

len This is the length of the data pointed to by *ptr*. See **d4seekN** for more details.

Returns: Refer to **d4seekNext** for the possible return values.

Locking: Refer to **d4seekNext** for locking procedures.

See Also: **d4seekN, d4tagSelect, d4seekNext, d4seek**

d4skip

Usage: int d4skip(DATA4 *data, long numRecords)

Description: This function skips *numRecords* from the current record number. The selected tag is used. If no tag is selected, record number ordering is used.

If there is no current record, either because the data file has no records or the data file has just been opened, **d4skip** generates an error since there is no record to skip from.

The new record is read into the record buffer and becomes the current record.

If there is no entry in the selected tag for the current record, the closest entry, as determined by a call to **d4tagSync**, is used as the starting point.

If the data file is shared and memory optimization is being used, the new record might not reflect recent changes or additions made by other users. Disable memory optimization or set **CODE4.readLock** to true (non-zero), to ensure access to the latest file changes.

It is possible to skip one record past the last record in the data file and create an end of file condition. Refer to **d4eof** for the exact effect.

Parameters:

numRecords The number of logical records to skip forward. If *numRecords* is negative, the skip is made backwards.

Returns:

r4success Success.

r4bof The current record is the top record. In addition, the last skip call attempted to skip before the top record of the data file.

r4eof Skipped to the end of the file.

r4locked A required lock attempt did not succeed. The locking attempt was either for flushing changes to disk or an attempt to lock the new record as required when **CODE4.readLock** is true (non-zero).

r4unique This value is returned when a record flush causes a duplicate key in a unique tag, and **t4unique** returned **r4unique** for the tag.

< 0 Error.

Locking: If record buffer flushing is required, the record and index files are locked to accomplish this task. If **CODE4.readLock** is true (non-zero), the new record is locked before it is read. See **d4flush** and **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

See Also: **CODE4.lockEnforce, code4unlockAuto, d4eof, d4flush**

```
/*ex66.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
     FIELD4 *name ;

     code4init( &cb ) ;
     data = d4open( &cb, "NAMES" ) ;
    /* Skip to the last record in the file whose NAME field is "John"*/

    code4optStart( &cb ) ;

    name = d4field( data, "F_NAME" ) ;
```

```
   for( d4bottom( data ) ;! d4bof( data ) ; d4skip( data, -1L ) )
   {
      if ( strcmp( "John        ", f4str( name )) == 0 ) /* 0 indicates a find */
         break ;
   }
   if( d4bof( data ) )
      printf( "John not located\n") ;
   else
      printf( "The last John is in record: %d\n",d4recNo( data ) ) ;

   d4close( data ) ;
   code4initUndo( &cb ) ;
}
```

d4tag

Usage: TAG4 *d4tag(DATA4 *data, const char *tagName)

Description: A search is made for the tag name in one of the index files of the data file. If the tag name is located, a pointer to the corresponding **TAG4** structure is returned.

Parameters:

tagName This is a pointer to a character string, which contains the name of the tag to search for.

Returns:

Not Null A pointer to the tag structure corresponding to the tag name is returned.

Null If the tag name could not be located, then a null pointer is returned. This is an error condition if **CODE4.errTagName** is true (non-zero).

See Also: **CODE4.errTagName**

d4tagDefault

Usage: TAG4 *d4tagDefault(DATA4 *data)

Description: The default tag is the currently selected tag, if one is selected. Otherwise, it depends on the index type. If the **S4MDX** switch is defined, then the default tag is the first tag created in the index file. If the **S4FOX** switch is defined, then it is the tag with the lowest alphabetical name. If the **S4CLIPPER** switch is defined, it is the first index listed in the Control Group File or if there is no file, then it is the first index opened.

d4tagDefault returns a pointer to the **TAG4** structure of the default tag. If there are no tags for the data file, a null pointer is returned.

d4tagNext

Usage: TAG4 *d4tagNext(DATA4 *data, TAG4 *tagOn)

Description: **d4tagNext** allows you to iterate sequentially through all of the tags corresponding to the data file.

Parameters:

tagOn This is the current tag in the iteration.

Returns:

Not Null The tag following *tagOn* is returned. If *tagOn* is null, then the first tag is returned.

Null On the other hand, if *tagOn* is the last tag, then null is returned.

See Also: **d4tagPrev**

```
/*ex67.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( void )
{
    CODE4 settings ;
    DATA4 *info;
     TAG4 *tag ;

     code4init( &settings ) ;
     info = d4open( &settings, "INFO" ) ;

    /* List the names of the tags in any production index file corresponding
     to "INFO" */

    printf( "Production index file tags for data file: %s\n", d4alias( info ) ) ;
             /*d4tagNext returns the first tag when NULL is passed to it */
    for( tag = d4tagNext(info, NULL); tag != NULL ; tag = d4tagNext( info, tag ))
       printf( "Tag name: %s\n", t4alias( tag ) ) ;

    code4initUndo( &settings ) ;
}
```

d4tagPrev

Usage: TAG4 *d4tagPrev(DATA4 *data, TAG4 *tagOn)

Description: **d4tagPrev** allows you to iterate backwards through all of the tags corresponding to the data file.

Parameters:

tagOn This is the current tag in the iteration.

Returns:

Not Null The tag occurring before *tagOn* is returned. If *tagOn* is null, then the last tag is returned.

Null On the other hand, if *tagOn* is the first tag, then null is returned.

See Also: **d4tagNext**

```
/*ex68.c*/
#include "d4all.h"

int searchAll( DATA4 *d, char *value )
{
    TAG4 *origTag, *tag ;
     long origRecNo ;

     origTag = d4tagSelected( d ) ; /* Save the current tag*/
     origRecNo = d4recNo( d ) ;
                           /*d4tagPrev returns the last tag when NULL is passed to it*/
    for( tag = d4tagPrev( d, NULL); tag != NULL; tag = d4tagPrev( d, NULL ))
    {
       d4tagSelect( d, tag ) ;
       if( d4seek( d, value ) == 0 )
       {
          d4tagSelect( d, origTag ) ;
          return d4recNo( d ) ;
       }
    }
    d4tagSelect( d, origTag ) ;
    d4go( d, origRecNo ) ;
```

```
    return -1 ;
}
void main()
{
    CODE4 cb ;
    DATA4 *data ;
    char key[] = "Abbott" ;
    int rc ;

    code4init( &cb ) ;
    data = d4open( &cb, "INFO" ) ;

    rc = searchAll( data, key ) ;
    if (rc != -1 )
    {
        d4go( data, rc ) ;
        printf( "%s\n", d4record( data ) ) ;
    }
    code4initUndo( &cb ) ;
}
```

d4tagSelect

Usage: void d4tagSelect(DATA4 *data, TAG4 *tag)

Description: **d4tagSelect** selects a tag to be used for the next data file positioning statements. The selected tag is used by positioning calls to **d4skip**, **d4seek**, **d4seekNext**, **d4position**, **d4top**, and **d4bottom**. To select record number ordering,, make the parameter *tag* null.

Parameters:

tag This is a pointer to a **TAG4** structure, which identifies the tag to make the "selected" tag. If tag is null, then CodeBase positioning functions access the records in physical order.

See Also: **d4skip**, **d4seek**, **d4seekNext**, **d4position**, **d4top**, **d4bottom**.

```
/*ex69.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
    TAG4 *nameTag, *defaultTag ;

    code4init( &cb ) ;
    data = d4open( &cb, "INFO" ) ; /* automatically open data & index file.*/
    nameTag = d4tag( data, "NAME_TAG" ) ;
    defaultTag = d4tagDefault( data ) ;

    d4tagSelect( data, defaultTag ) ; /* Select the default tag*/
    d4seekDouble( data, 32 ) ;     /* Seek using default tag 'AGE_TAG'*/

    d4tagSelect( data, nameTag ) ; /* Select the 'NAME_TAG' tag*/
    d4seek( data, "Fred" ) ; /* Seek using 'NAME_TAG */

    d4tagSelect( data, NULL ) ;  /*Select record ordering */
    d4seek( data, "ginger" );    /*The seek uses the default tag, which is
                                       AGE_TAG, so this seek fails even though "ginger"
                               is in the data file */

    code4initUndo( &cb ) ;
}
```

d4tagSelected

Usage: TAG4 *d4tagSelected(DATA4 *data)

Description: A pointer to the selected tag is returned. If there is no selected tag, then null is returned.

d4tagSync

Usage: int d4tagSync(DATA4 *data, const TAG4 *tag)

Description: This function is used to position the current record to a valid record position within the currently selected tag. Changes to the current record are flushed to disk if required.

This function is useful for moving to a new record when changes to the record cause it to no longer be found within the tag. For example, if the change to the record causes it to contain a duplicate key entry in a unique tag, the record no longer is in a valid position. Calling **d4tagSync** ensures that the record and the tag are in valid positions within the tag.

Note

This function is only useful prior to a call to **d4skip, d4seekNext**, **d4seekNextDouble** or **d4seekNextN** when the current record is not found within the tag. Other positioning functions, such as **d4go**, **d4top**, and **d4seek** perform their movements regardless of the state of the current record.

Returns:

r4success The record is in a valid position.

r4after The record was not found. The data file is positioned to the record after.

r4eof The record was not found and the search value was greater than the last key of the tag. Consequently, **d4goEof** was called to turn the *EOF* condition on.

r4locked A required lock failed. The database is in an invalid position and a explicit positioning statement such as **d4top** should be called prior to calling **d4skip**.

r4unique This value is returned when a record flush causes a duplicate key in a unique tag, and **t4unique** returned **r4unique** for the tag.

< 0 An error has occurred during the repositioning.

Locking: If record buffer flushing is required, the record and index files are locked. If **CODE4.readLock** is true (non-zero), the new record is locked before it is read. See **d4flush** and **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

See Also: **d4flush, code4unlockAuto**

```
/*ex70.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* Borland only*/

void main( void )
{
   CODE4 cb ;
   DATA4 *data ;
     TAG4 *tag ;
```

```
    code4init( &cb ) ;
    data = d4open( &cb, "DBF" ) ;
    tag = d4tag( data, "NAME_TAG" ) ; /* A tag with '.NOT.DELETED()' filter */

   d4top( data ) ; /* Position to the first record that is not deleted.*/
   d4delete( data ) ; /* The current record no longer is located in the tag*/

   d4tagSync( data, tag );/*The change to the record is flushed, and the datafile
                    is positioned on a valid record.*/
   d4skip( data, 1L) ;
   /*... some other code ...*/

   code4initUndo( &cb ) ;
}
```

d4top

Usage: int d4top(DATA4 *data)

Description: The top record of the data file is read into the data file record buffer and the current record number is updated. The selected tag is used to determine which is the top logical record. If no tag is selected, the first physical record of the data file is read.

Returns:

r4success Success.

r4eof End of File (Empty tag or data file.)

r4locked A required lock attempt did not succeed. This was either a lock to flush changes to disk, or an attempt to lock the top record as required when **CODE4.readLock** is true (non-zero).

r4unique This value is returned when a record flush causes a duplicate key in a unique tag, and **t4unique** returned **r4unique** for the tag.

< 0 Error.

Locking: If record buffer flushing is required, the record and index files are locked to accomplish this task. If **CODE4.readLock** is true (non-zero), the top record is locked before it is read. See **d4flush** and **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

See Also: **d4bottom**, **d4bof**, **d4eof, CODE4.readLock, code4unlockAuto, d4flush**

```
/*ex71.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    DATA4 *info ;
    TAG4 *defaultTag ;
    long count = 0 ;

    code4init( &cb ) ;
    info = d4open( &cb, "INFO" ) ; /* automatically open data & index file*/
    defaultTag = d4tagDefault( info ) ;
    d4tagSelect( info, defaultTag ) ; /* Select default tag*/

    for( d4top( info ) ; ! d4eof( info ) ; d4skip( info, 1 ) )
       count ++ ;

    printf( "%d records in tag  ", count ) ;
    printf( "%s\n", t4alias( defaultTag ) );
```

```
    d4close( info ) ;
    code4initUndo( &cb ) ;
}
```

d4unlock

Usage: int d4unlock(DATA4 *data)

Description: **d4unlock** writes any changes to disk and removes any file locks on the data file and its corresponding index and memo files. Locks on individual records and/or the append bytes are also removed.

d4unlock writes any changes to disk prior to removing the file locks. When disk caching software is used, these disk writes may not immediately be placed on disk. If this is a concern, call **d4flush** to bypass the disk caching. Doing this, however, sacrifices some application performance.

If conditional compilation switch **S4OFF_MULTI** is defined, **d4unlock** does nothing.

Returns:

r4success Success.

r4unique The record was not flushed due to the following circumstances: first, writing the record caused a duplicate key in a unique key tag. In addition, **t4unique** returns **r4unique**. Regardless, the data and any corresponding index and memo files are unlocked.

< 0 Error.

Locking: If record buffer flushing is required, the record and index files are locked. See **d4flush** and **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished. If a required lock fails, then the flushing does not occur.

d4unlock removes all the locks from the data file and its index and memo files regardless of whether the flushing was successful.

See Also: **code4lock**, **d4flush**, **d4lockAdd**, **d4lock, d4flush, code4unlockAuto**

```
/*ex72.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers */

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
     FIELD4 *name ;
     TAG4 *nameTag ;
     int rc ;

     code4init( &cb ) ;
     data = d4open( &cb, "INFO" ) ;
    nameTag = d4tag( data, "NAME_TAG" ) ;
    name = d4field( data, "NAME" ) ;

    error4exitTest( &cb ) ; /* check for errors*/
    d4lockAll( data ) ;
    d4tagSelect( data, nameTag ) ;

    for(rc = d4seek( data, "Fred" ) ; rc == r4success ; rc = d4skip( data, 1) )
    {
```

```
      if( memcmp ( f4ptr( name ), "Fred", 4 ) == r4success )
         printf( "Fred in record %d", d4recNo( data ) );
      else
         break ;
   }
   d4unlock( data ) ;

   d4close( data ) ;
   code4initUndo( &cb ) ;
}
```

d4write

Usage: int d4write(DATA4 *data, long recordNumber)

Description: **d4write** explicitly writes the current record buffer to a specific record in the data file. If *recordNumber* is **(long)** -1, the current record number is used.

The 'record changed flag' for the current record buffer is reset to unchanged (zero). The record buffer is NOT flushed. In fact **d4flush** uses **d4write** to perform flushing.

The record is written regardless of the status of the 'record changed flag'. **d4write** also locks and updates all open index files. Finally, **d4write** checks to see if any memo fields have been changed. If they have, they are updated to disk and the record buffer references to the memo entries are updated.

Parameters:

recordNumber *recordNumber* specifies the record to which the record buffer is written. This may reference any valid record number. If *recordNumber* is **(long)** -1, the current record buffer is written to the current record.

Returns:

r4success Success.

r4locked A required lock attempt did not succeed. If there was a problem locking the record or an index file tag, the record will not have been written. However, if the problem was due to not being able to lock the memo file when extending the memo file, only that memo file entry will not have been written.

r4unique The record was not written due to the following circumstances: first, writing the record caused a duplicate key in a unique key tag. Secondly, **t4unique** returned **r4unique** for the tag.

< 0 Error.

Note

Usually, it is not necessary to explicitly call **d4write**. If the data file is modified using a field function, the record is written automatically upon skipping, seeking, flushing, going, or closing. This is possible because the record changed flag determines whether the data file record buffer has been written to disk since it was last changed. When using **d4write** on a modified record buffer, a call to **d4appendStart** is

recommended, to suspend flushing before changing the record buffer with the field functions.

Locking: **d4write** locks the record and may lock the index files during the write. See **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

```
/*ex73.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    DATA4 *data;
    long numRecs ;

    code4init( &cb ) ;
    cb.accessMode = OPEN4DENY_RW ; /* open exclusively to avoid corruption*/
    data = d4open( &cb, "INFO1" ) ;
    d4go( data, 1L ) ;

    /* Make all of the records in the data file the same as the first record*/

    for( numRecs = d4recCount( data ) ; numRecs-1 ; numRecs -- )
       if( d4write( data, numRecs ) != 0 )
          break ;

    d4close( data ) ;
    code4initUndo( &cb ) ;
}
```

d4zap

Usage: int d4zap(DATA4 *data, long startRec, long endRec)

Description: **d4zap** removes the stated range of records from the data file. This range is always specified using record number ordering. Consequently, if *endRec* is less than *startRec*, no records are removed. Once the records are removed, **d4zap** also reindexes all open index files.

To zap the entire data file, call **d4zap** with *startRec* equal to 1 and *endRec* equal to a really large number or equal to **d4recCount**.

After **d4zap** completes, the contents of the record buffer and the record number are undefined. Explicitly call a positioning function such as **d4top** to position to a desired record.

d4zap does not alter the memo file. Consequently, memo entries referenced by removed records will be wasted disk space. Call **d4memoCompress** to compress the memo file.

WARNING

Be careful when using this function since it can immediately remove large numbers of records. It can also take a long time to complete. Appropriate backup measures should be taken before calling this function. **d4zap** does not make a backup file.

WARNING

This member function may not be used to remove records from a data file while a transaction is in progress. Attempts to do so fail and generate an **e4transViolation** error.

Parameters:

startRec This is the first record to remove from the data file. If *startRec* is greater than the number of records in the data file, nothing happens.

endRec This is the last record to remove from the data file. This parameter may be greater than the actual number of records in the data file.

Returns:

r4success Success.

r4locked A required lock attempt did not succeed.

r4unique This is returned when **t4unique** returned **r4unique** for the tag and a repeat key occurred while rebuilding a unique-key tag. Consequently, one of the index files was not reindexed correctly. The data file was successfully zapped.

< 0 Error.

Locking: **d4zap** locks the data file and its index files. See **code4unlockAuto** for details on how any necessary locking and unlocking is accomplished.

See Also: **d4pack**, **d4memoCompress, code4unlockAuto**

```
/*ex74.c*/
#include "d4all.h"

long zapLast( DATA4 *info, long toDelete )
{
    printf( "%s has %d records\n",d4alias( info ), d4recCount( info )) ;

    d4zap( info, d4recCount( info) - toDelete + 1L, 1000000) ;
    printf( "%s now has %d records\n",d4alias( info ), d4recCount( info ));
    return d4recCount( info ) ;
}
void main()
{
     CODE4 cb ;
     DATA4 *data ;
     int rc ;
     long toDelete = 3;

     code4init( &cb ) ;
     cb.accessMode = OPEN4DENY_RW ;
     data = d4open( &cb, "INFO" ) ;
     rc = zapLast( data, toDelete ) ;
     code4initUndo( &cb ) ;
}
```

Date Functions

date4assign	date4isLeap
date4cdow	date4long
date4cmonth	date4month
date4day	date4timeNow
date4dow	date4today
date4format	date4year
date4init	

The date functions are used to perform basic manipulations on dates. This is necessary because dBASE, FoxPro, Clipper, and CodeBase store dates in "CCYYMMDD" format on disk (eg. January 1, 1990 is stored as "19900101"). This date format, however, is not one that most people use in day to day life, nor are character strings particularly easy to use in mathematical computations.

Note

Date arithmetic is done using Julian days. A Julian day is defined as the number of days since Jan. 1, 4713 BC. The smallest Julian day that can be used is 1721425L (Dec. 30, 0000).

The date functions are low level. Consequently, it is not necessary to have a **CODE4** structure initialized before these functions may be used.

```
/*ex75.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( )
{
     CODE4 cb ;
    DATA4 *data;
     FIELD4 *birthField ;
     TAG4 *birthTag ;
     char myBirthDate[8] = "19690225" ;
    char today[8], result[18], tomorrow[8], yesterday[8] ;

     date4today( today ) ; /* set today equal to the system clock date.*/
     date4format( today, result, "MMMMMMMM DD, CCYY") ;
    printf( "Today is %s, %s\n", date4cdow( today ), result ) ;

     date4assign( tomorrow, date4long( today )+ 1L) ;
     date4format( tomorrow, result, "MMMMMMMM DD, CCYY") ;
    printf( "Tomorrow is %s, %s\n", date4cdow( tomorrow ), result ) ;

     date4assign( yesterday, date4long( tomorrow ) -2L );
     date4format( yesterday, result, "MMMMMMMM DD, CCYY") ;
    printf( "Yesterday was %s, %s\n", date4cdow( yesterday ), result ) ;

     code4init( &cb ) ;
     data = d4open( &cb, "INFO" ) ;
     birthField = d4field( data, "BIRTH_DATE" ) ;
     birthTag = d4tag( data, "BIRTH_TAG" ) ;
    d4tagSelect( data, birthTag ) ;

    if( d4seek( data, myBirthDate ) == 0 )
       printf( "I'm in record %d\n", d4recNo( data )) ;

    /* change all birthdate fields to my birth date.*/
    for( d4top( data ) ; !d4eof( data ) ; d4skip( data, 1 ) )
       f4assign( birthField, myBirthDate ) ;
    code4initUndo( &cb ) ;
}
```

Date Function Reference

date4assign

Usage: void date4assign(char *date, long julianDay)

Description: A **(long)** , in Julian date form, is converted into a character date and copied into *date*.

This function is the inverse function of **date4long**.

Parameters:

date The parameter *date* must point to an array of eight characters. The character form is in the “CCYYMMDD” (century, year, month, day) format.

julianDay This is a **(long)** value**,** which is a date in Julian date form.

See Also: Date functions chapter in User's Guide

```
/*ex76.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ;

void main( )
{
    char date[8]= "19900101" ;
    char dayBefore[8], result[11], resultBefore[11] ;

    date4assign( dayBefore, date4long(date) - 1L ) ;
    date4format( date, result, "MMM DD, 'YY") ;
    date4format( dayBefore, resultBefore, "MMM DD, 'YY") ;

    printf("%s is after %s\n", result, resultBefore);
}
```

date4cdow

Usage: const char *date4cdow(const char *date)

Description: A pointer to the day of the week corresponding to *date* is returned.

Parameters:

date This is a pointer to a character array in the form “CCYYMMDD”.

Returns: A pointer to a character array containing the day of the week is returned.

If *date* does not contain a valid date, a null pointer is returned.

See Also: **date4day**, **date4cmonth**

```
/*ex77.c*/
#include "d4all.h"

void main( )
{
    char birthDate[8+2] ;

    printf( "Enter your birthdate in CCYYMMDD format\n") ;
    fgets( birthDate, 10, stdin ) ;

    printf( "You were born on a %s\n", date4cdow( birthDate )) ;
    /* displays "You were born on a Monday" if a monday was entered. */
}
```

date4cmonth

Usage: const char *date4cmonth(const char *date)

Description: A pointer to the month of the year corresponding to *date* is returned.

Parameters:

date This is a pointer to a character array in the form “CCYYMMDD”.

Returns: A pointer to a character array containing the month of the year is returned.

If *date* does not contain a valid date, a null pointer is returned.

See Also: **date4month**

```
/*ex78.c*/
#include "d4all.h"

void main( )
{
    char today[8] ;

    date4today( today ) ;
    printf( "The current month is %s\n", date4cmonth( today )) ;
    /* displays "The current month is January" if the system clock says
       that it is. */
}
```

date4day

Usage: int date4day(char *date)

Description: The day of the month, from 1 to 31, corresponding to *date*, is returned as an integer. If the *date* contains an invalid date, zero is returned.

See Also: **date4cdow**, **date4dow**

date4dow

Usage: int date4dow(const char *date)

Description: The day of the week, from 1 to 7, is returned as an integer according to the following table:

Day	Numeric Day of Week
Sunday	1
Monday	2
Tuesday	3
Wednesday	4
Thursday	5
Friday	6
Saturday	7

See Also: **date4cdow**, **date4day**

```
/*ex79.c*/
#include "d4all.h"

void main( )
```

```
{
    char date[8] ;
    int tillEnd ;

    date4today( date ) ;
    tillEnd = 7 - date4dow( date ) ;
    printf( "%d days left till end of the week", tillEnd ) ;
}
```

date4format

Usage: void date4format(const char *date, char *result, char *picture)

Description: *date* is formatted according to the date picture parameter *picture* and copied into parameter *result*. **date4format** is guaranteed to be null terminated. The special formatting characters are 'C' - Century, 'Y' - Year, 'M' - Month, and 'D' - Day. If there are more than two 'M' characters, a character representation of the month is returned.

Parameters:

date A string in standard date format “CCYYMMDD”.

result This is a character array pointing to at least **strlen(** *picture* **)** characters. The date, formatted according to *picture*, is stored in *result*. The result will be null terminated.

picture This is a null-terminated character string that contains a date picture.

```
/*ex80.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( )
{
    char dt[8] = "19901002" ;
    char  result[20] ;

    date4format( dt, result, "YY.MM.DD" ) ;   /* 'result' will contain "90.10.02"*/
     printf("%s\n", result ) ;
    date4format( dt, result, "CCYY.MM.DD" ) ; /* 'result' will contain "1990.10.02"*/
     printf("%s\n", result ) ;
    date4format( dt, result, "MM/DD/YY" ) ;   /* 'result' will contain "10/02/90"*/
     printf("%s\n", result ) ;
    date4format( dt, result, "MMM DD/CCYY" ) ;
    printf("%s\n", result) ;  /* outputs "Oct 02/1990"*/
}
```

date4init

Usage: void date4init(char *date, const char *value, char *picture)

Description: The *date* is initialized from parameter *value*. Parameter *value* must be formatted according to picture parameter *picture*.

This is the inverse function of **date4format**.

If any of the date is missing, the missing portion is filled in using the date January 1, 1980.

Parameters:

date After the call to **date4init** is completed, date will contain a string in standard date format “CCYYMMDD”.

value This is a character string that represents a date.

picture This null terminated character array specifies the format of parameter *value*.

See Also: Date Functions chapter in the Users Guide, **date4assign**

```
/*ex81.c*/
#include "d4all.h"

void main( )
{
    char day[8] ;
    /*date4init puts the given date in CCYYMMDD format */
    date4init( day, "Oct 07/90", "MMM DD/YY" ) ;
    printf( "Oct 07/90 becomes %s\n", day ) ;
    date4init( day, "08/07/1989", "MM/DD/CCYY") ;
    printf( "08/07/1989 becomes %s \n", day );
}
```

date4isLeap

Usage: int date4isLeap(const char *date)

Description: This member function is used to determine whether *date* contains a date within a leap year.

Returns:

Non-zero The date is within a leap year.

0 The date is invalid, or the date is not within a leap year.

date4long

Usage: long date4long(const char *date)

Description: **date4long** converts *date* from standard format to a Julian day.

Note

You can use **date4long** to verify whether a date value is legitimate by checking for a negative return code.

Returns:

> 0 A Julian date value representing the date.

0 The date is blank.

< 0 The date value was not legitimate.

See Also: **f4long, f4assignLong**

```
/*ex82.c*/
#include "d4all.h"

void main( )
{
    long yesterday ;
    char today[8], tomorrow[8], result[13] ;

    date4today( today ) ; /* Get the current date from the system clock*/

    yesterday = date4long( today ) - 1L ;

    date4assign( tomorrow, yesterday + 2L );
    date4format( today, result, "MMM DD, CCYY" ) ;
    printf( "Today is %s\n", result ) ;
    printf( "The Julian date   for yesterday is %d\n", yesterday ) ;
    printf( "The Julian date   for tomorrow is %d\n", date4long( tomorrow )) ;
}
```

date4month

Usage: int date4month(char *date)

Description: The month of the *date*, from 1 to 12, is returned as an integer. If the date stored in *date* is invalid, 0 is returned.

See Also: **date4cmonth**

```
/*ex83.c*/
#include "d4all.h"

static int daysInMonth[] = { 0,31,28,31,30,31,30,31,31,30,31,30,31 } ;

void main( )
{
    int endOfMonth  ;
    char today[8] ;

    date4today( today ) ;
    endOfMonth   = daysInMonth[ date4month( today ) ] ;
    if( date4month( today ) == 2 && date4isLeap( today ) )
       endOfMonth++ ;
    printf("there are %d days left till the end of the month\n",
                                                          endOfMonth - date4day( today ) ;
}
```

date4timeNow

Usage: void date4timeNow(char *time)

Description: This function initializes *time* from the current time of system clock.

Parameters:

time The parameter *time* must point to 8 characters of memory. This memory is filled with the time in “HH:MM:SS” format. HH represents the hour which is between 0 and 24; MM represents the minute which is between 0 and 59; and SS represents the second which between 0 and 59. Note that “24:00:00” is the same time as “00:00:00”. However, “00:00:00” represents the start of a new day and “24:00:00” represents the end of an old day.

See Also: **date4assign**

```
/*ex84.c*/
#include "d4all.h"

void main( )
{
    char time[9] ;

    date4timeNow( time ) ;
    time[9] = 0 ;         /* Add the null for the printf() */
    printf( "\nThe current time is %s.", time ) ;
}
```

date4today

Usage: void date4today(char *date)

Description: **date4today** sets *date* to the current date from the system clock.

See Also: **date4assign**

```
/*ex85.c*/
#include "d4all.h"
```

```
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( )
{
    char d[8] ;
     int daysToWeekEnd ;

     date4today( d ) ;
    printf("Today is %s\n", date4cdow( d )) ;

    daysToWeekEnd = 7 - date4dow( d ) ;
    if( daysToWeekEnd == 0 || daysToWeekEnd == 6 )
       printf("Better enjoy it!\n") ;
    else
       printf("Only %d more to go till the weekend\n", daysToWeekEnd ) ;
}
```

date4year

Usage: int date4year(char *date)

Description: The century/year of *date* is returned as an integer. If the date contains blanks, 0 is returned.

See Also: **date4format**, **date4month**, **date4day**

```
/*ex86.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( )
{
    CODE4 cb ;
    DATA4 *data ;
    FIELD4 *bdate ;
     char date1[8], date2[8] ;

     code4init( &cb ) ;
     data = d4open( &cb, "INFO" ) ;
     bdate = d4field( data, "BIRTH_  DATE" ) ;
    d4top( data ) ;

     strcpy( date1, f4str( bdate ) ) ; /* make a copy of the field's contents*/
     d4skip( data, 1 ) ;
     strcpy( date2, f4str( bdate ) ) ; /* make a copy of the field's contents*/

     if( date4year( date1 ) != date4year( date2 ) )
       printf("The people in the 1st and 2nd records were born in different years %d, %d"
                                                    , date4year( date1 ), date4year( date2 )) ;
     else
         printf("The people in the 1st and 2nd record were born in the same year %d"
                                                    , date4year( date1 )  ) ;
    d4close( data ) ;
    code4initUndo( &cb ) ;
}
```

Error Functions

error4	error4hook
error4describe	error4set
error4exitTest	error4text
error4file	

Once a CodeBase error is generated, CodeBase functions from most modules will do nothing, until instructed to do otherwise. In addition, once any CodeBase function generates an error, functions from most modules return an error.

This feature is very useful because it becomes unnecessary to constantly check for error returns. For example, it may be appropriate to open a number of files and then check the error code. Note that it is easy to reset the error code. An exception to the above rule are functions, such as **d4close**, which close files. These functions always close any file that is open and free appropriate memory.

The modules which do nothing except return an error once an error has occurred are as follows: data, file, index file, tag, expression evaluation, field, memo, sort and relate. All of these modules are initialized with a pointer to the **CODE4** structure and thereby have access to the error code.

CodeBase displays most error messages through the error functions **error4** and **error4describe**. When the error function is called, the first parameter specifies an integer constant defined in the header file "D4DATA.H". The error function uses this constant to lookup and display a small error description. Refer to Appendix A for a list of the integer constants, their small error descriptions and a more detailed explanation.

To turn off or alter error reporting, define the conditional compilation switch **S4ERROR_HOOK**. This defines function **error4hook** which is documented below.

Error Function Reference

error4

Usage: int error4(CODE4 *codebase, int errCode, long extraInfo)

Description: This function sets the error code and displays an error message.

Once an error code has been set, functions from many modules do nothing except return an error. Refer to the introduction.

Parameters:

codebase A pointer to a **CODE4** structure.

errCode This is an error code corresponding to the error. Its potential values are defined in "e4error.h". This value is assigned to variable **CODE4.errorCode**.

extraInfo This stores a code which may be used to contain some additional diagnostic information on the error and where it originated within the CodeBase library. The pre-defined constant values which are passed by CodeBase may be found in file 'e4defs.h'. The usage of *extraInfo* is determined by the **E4OFF_STRING** conditional compilation switch. When defined, this value is output in the error message as a numeric value. When **E4OFF_STRING** is not defined, **error4text** is called to retrieve an additional error string which is outputted.

Returns: The parameter *errCode* is returned.

See Also: **CODE4.errOff**, **CODE4.errorCode, error4text, E4OFF_STRING**, "Appendix A: Error Codes"

error4describe

Usage: int error4describe(CODE4 *codebase, int errCode, long extraInfo, const char *desc1 = 0, const char *desc2 = 0, const char *desc3 = 0)

Description: These functions are used to report errors to the program and the end user. When an error occurs within the CodeBase library, either **error4** or **error4describe** is called.

Parameters:

codebase A pointer to a **CODE4** structure.

errCode This is an error code corresponding to the error. CodeBase recognizes the values specified in "Appendix A: Error Codes". This value is assigned to **CODE4.errorCode**.

extraInfo This is a variable which contains extra information for the error processing. It is only used internally.

desc1 This is the first line of the error message. *desc1* must point to a null terminated character array or a null value. If *desc1* is null, no additional information on the error is displayed.

desc2 This is the second line of the error message. *desc2* must point to a null terminated character array or a null value. If *desc2* is null, only the message in *desc1* is displayed.

desc3 This is the final line of the error message. *desc3* must point to a null terminated character array or a null value. If *desc3* is null, only *desc1* and *desc2* messages are displayed.

Returns: *errCode* is returned.

See Also: **CODE4.errOff**, **CODE4.errorCode, error4, error4text, E4OFF_STRING**, "Appendix A: Error Codes"

```
/*ex87.c*/
```

```
#include "d4all.h"

int display( CODE4 *cb, char *p )
{
    if( p == NULL )
       return error4describe( cb, e4parm, 0, "Null display string", 0, 0 ) ;
    printf( "%s\n", p ) ;
    return 0 ;
}
void main()
{
     CODE4 cb ;
     char p[] = " some string" ;

     code4init( &cb ) ;
     display( &cb, p ) ;
     display( &cb, 0 ) ;
     code4initUndo( &cb ) ;
}
```

error4exitTest

Usage: void error4exitTest(CODE4 *codebase)

Description: This function tests to see if there has been an error. If **CODE4.errorCode** is negative, **code4exit** is called to exit the application. If **CODE4.errorCode** is zero or a positive value, **error4exitTest** returns and the application continues to execute.

```
/*ex88.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;

     code4init( &cb ) ;
     data = d4open( &cb, "FILE" ) ;
    error4exitTest( &cb ) ; /* the application will exit if FILE cannot be opened */
    /*... other code ... */
     code4initUndo( &cb ) ;
}
```

error4file

Usage: int error4file(CODE4 *codebase, const char *fileName, int overwrite)

Description: This function re-directs the messages of the standard error functions to a file instead of displaying them on the screen. This is useful for tracking error messages without interrupting program execution.

If the **S4ERROR_HOOK** conditional compilation switch is used, this setting is ignored, and error messages are not re-directed to the file.

Parameters:

codebase A pointer to a **CODE4** structure.

fileName *fileName* is a null terminated string containing the file name in which the error messages are written. If *fileName* contains a path, it is used, otherwise the file is written in the current directory. If the file does not exist, it is created.

overwrite If *fileName* already exists, *overwrite* is used to determine whether the new error messages are added to the end of the file, or any existing errors should be overwritten. If *overwrite* is true (non-zero), the default, the contents of the file are erased as it is opened. If *overwrite* is false (zero), new error messages are appended to the end of the file.

Returns:

r4success The error file was successfully opened or created.

r4noCreate The error file could not be opened or created with the file name and path provided.

See Also: **error4**, **file4open**, **file4create**, **S4ERROR_HOOK**

error4hook

Usage: void error4hook(CODE4 *codebase, int errCode1, long errCode2,
const char *desc1, const char *desc2,
const char *desc3)

Description: The purpose of this function is to allow the programmer to easily alter or turn off error reporting. When CodeBase is built with the conditional compilation switch **E4HOOK**, **error4hook** is called by **error4** and **error4describe** to display messages. By default, **error4hook** does nothing.

Consequently, to turn off error reporting, just define **E4HOOK** and do nothing else.

To alter error reporting, alter **error4hook** to display error messages as desired. Function **error4hook** is defined statically in “C4HOOK.C”.

Parameters: The parameters to **error4hook** correspond to parameters passed to **error4** and **error4describe**.

If **error4hook** is displaying error messages for **error4** then the parameters *desc2* and *desc3* are null. The parameter *desc1* may also be null depending on whether the error code has any auxiliary information associated with it.

Call **error4text** to get the error string associated with either *errCode1* or *errCode2*. Note that if **E4OFF_STRING** or **E4OFF** is defined, the **error4text** function may return the string "Invalid or Unknown Error Code".

See Also: **error4**, **E4HOOK, E4OFF_STRING, E4OFF, error4text, error4describe**

error4set

Usage: int error4set(CODE4 *codebase, int errCode)

Description: This function sets the **CODE4.errorCode** member variable to *errCode* and returns the previous setting. **error4set** does not display an error message, even if *errCode* is an error value.

Returns: The previous setting of is returned.

See Also: **CODE4.errorCode**

error4text

Usage: const char *error4text(CODE4 *codebase, long errCode)

Description: This function retrieves a pointer to the error message string associated with an error code. Often, this error code is obtained from **CODE4.errorCode**.

This is a string that CodeBase displays, by default, when an error is generated.

Returns: A string containing the error message.

See Also: **CODE4.errorCode**

Expression Evaluation Functions

expr4data	expr4source
expr4double	expr4str
expr4free	expr4true
expr4len	expr4type
expr4parse	expr4vary

This module evaluates dBASE expressions. This is necessary because dBASE expressions are used to specify tag keys and filters. dBASE expression evaluation could also be useful in applications where a user enters an expression interactively in order to specify relation queries.

Note

Avoid using this module to perform calculations on fields. Instead use the field functions and regular C functions. Otherwise, your application will execute slower than necessary.

CodeBase evaluates expressions as a two step process. First, the expression is pseudo-compiled. Then the pseudo-compiled expression is executed to return the result. This is efficient when expressions are evaluated repeatedly, since the pseudo-compiled form only needs to be generated once.

"Appendix C: dBASE Expressions" describes the supported dBASE expressions in-depth.

The following example uses the expression functions to return the contents of the fields "FNAME" and "LNAME".

```
/*ex89.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main()
{
    CODE4  cb ;
    DATA4  *data ;
     EXPR4 *expr ;
     char *result ;

     code4init( &cb ) ;
     data = d4open( &cb, "DATA" ) ;

    d4go( data, 1L ) ;
    /* "FNAME" and "LNAME" are Character field names of data file "DATA.DBF"*/

     expr = expr4parse( data, "FNAME+\' \'+LNAME") ;
     expr4vary( expr, &result ) ;
    printf( "FNAME and LNAME for Record One: %s\n", result ) ;

    expr4free( expr ) ;
    d4close( data ) ;
    code4initUndo( &cb ) ;
}
```

Expression Function Reference

expr4data

Usage: DATA4 *expr4data(EXPR4 *expr)

Description: A pointer to a **DATA4** structure for *expr's* database is returned.

Example: See **expr4double**

expr4double

Usage: double expr4double(EXPR4 *expr)

Description: The expression is evaluated and the result is returned as a **(double)**. This function assumes that if the dBASE expression evaluates to a character result, the result is a character representation of a decimal number. If the result is a numeric result, it is cast to a **(double)**. If the expression evaluates to a date value, **expr4double** converts the resulting date into a Julian date value.

Returns: **expr4double** returns the **(double)** value of the evaluated expression. Since there is no error return on this operator, check the **CODE4**.**errorCode**, or call **error4exitTest** to determine if an error occurred.

```
/*ex90.c*/
#include "d4all.h"
#define VOTE_AGE 18.0
extern unsigned _stklen = 10000 ; /* for all Borland compilers */

void main( )
{
    CODE4 cb ;
    DATA4 *data ;
     EXPR4 *expr ;
     long count = 0;
     int rc ;

     code4init( &cb ) ;
     data = d4open( &cb, "INFO" ) ;
    expr = expr4parse( data, "AGE" ) ;

    for( rc = d4top( data ) ; rc != r4eof ; rc = d4skip( data, 1 ) )
       if( expr4double( expr ) >= VOTE_AGE )
          count ++ ;

    printf( "Possible voters: %d\n", count ) ;

    expr4free( expr ) ;
    code4initUndo( &cb ) ;
}
```

expr4free

Usage: void expr4free(EXPR4 *expr)

Description: All of the memory associated with the parsed expression is freed. If *expr* has already been freed then the result is undefined. There should be exactly one call to **expr4free** for each call to **expr4parse**.

If the call to **expr4parse** is unsuccessful (invalid), a call to **expr4free** is optional.

Parameters:

expr A pointer to the expression's **EXPR4** structure.

See Also: **expr4parse**

Example: See **EXPR4** introduction.

expr4len

Usage: int expr4len(EXPR4 * expr)

Description: The length of the expression is returned. This maximum length is determined when the expression is initially pseudo-compiled with **expr4parse**.

The length of the evaluated expression is not altered by using the dBASE functions TRIM() or LTRIM(), since these functions do nothing when a field is filled.

Returns: The length of the expression is returned.

See Also: **expr4vary**

```
/*ex91.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( )
{
    CODE4 settings ;
    DATA4 *db ;
     EXPR4 *fullName ;
     char *result, *name ;

     code4init( &settings ) ;
     db = d4open( &settings, "DATA" ) ;

    d4top( db ) ;
    fullName = expr4parse( db, "TRIM( LNAME )+', '+FNAME" ) ;
     name = (char * )malloc( expr4len( fullName ) ) ;
     expr4vary( fullName, &result ) ;
     strcpy( name, result ); /*make copy of result which is copied over the next time
                               expr4vary is called.*/
     d4skip( db, 1 ) ;
     expr4vary( fullName, &result ) ;
     /* For illustration purposes only:
        Avoid using the expression module when the field functions will suffice*/

    printf("%s is the first person in the data file\n", name ) ;
     printf("%s is the second person in the data file\n", result ) ;
     free( name ) ;
    code4initUndo( &settings ) ;
 }
```

expr4parse

Usage: EXPR4 *expr4parse(DATA4 *data, const char *expression)

Description: A dBASE expression is pseudo-compiled (parsed) and an **EXPR4** structure is created to contain the newly parsed expression.

expr4parse dynamically allocates memory. Once the expression is no longer needed, it is a good idea to free the memory with **expr4free**.

Parameters:

data If a field name without a data file qualifier is specified in the expression it is assumed to be associated with the data file referenced by *data*. Parameter *data* is also used to access a pointer to the **CODE4** structure for error message generation.

expression *expression* points to a null terminated string that contains a valid dBASE expression, which is to be parsed.

Returns:

Not Null A pointer to a **EXPR4** structure containing the parse information.

Null The expression could not be parsed.

See Also: **CODE4.errExpr**, **expr4vary** , **E4MISC**

```
/*ex92.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ;

void main( )
{
    CODE4 cb ;
    DATA4 *data, *info ;
     EXPR4 *expr ;
     char *result ;

     code4init( &cb ) ;
     data = d4open( &cb, "DATA" ) ;
    info = d4open( &cb, "INFO" ) ;
    expr = expr4parse( data, "FNAME+' '+DTOS( INFO->BIRTH_DATE)" ) ;

    d4top( data ) ;
    d4top( info ) ;
     expr4vary( expr, &result ) ;
    printf( "First name from DATA and birth date from INFO: %s", result ) ;

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

expr4source

Usage: const char *expr4source(EXPR4 *expr)

Description: **expr4source** returns a pointer to a null terminated copy of the original character form of the parsed dBASE expression.

Parameters:

expr A pointer to the expression's **EXPR4** structure.

Example: See **expr4type**

expr4str

Usage: const char *expr4str(EXPR4 *expr)

Description: The expression is evaluated and the parsed string is returned. The result should be used immediately, since the memory containing the result is overwritten when another expression evaluation function is called.

Returns: If the result of the evaluated expression is a **r4date** type, a string of the form "CCYYMMDD" is returned. If the result is a **r4str** type, then a character string is returned. If the result is a **r4num**, **r4numDoub**, **r4log**

or **r4dateDoub** type, an error is generated. Refer to the return values of **expr4type** for details about the different types of dBASE expressions. Check **CODE4.errorCode** to determine whether an error has occurred.

See Also: **expr4type**

expr4true

Usage: int expr4true(EXPR4 *expr)

Description: The expression is evaluated and assuming the expression evaluates to a logical result, either true (> zero) or false (zero) is returned.

If the expression is not logical, an error message is generated and **CODE4.errorCode** is set to an appropriate error value. In this case, the return code from **expr4true** should be ignored.

Parameters:

expr A pointer to the expression's **EXPR4** structure.

Returns:

> 0 The evaluated expression was true (> zero) for the current record.

0 The evaluated expression was false (zero) for the current record.

< 0 Error.

See Also: **expr4type**, **expr4source**

expr4type

Usage: int expr4type(EXPR4 *expr)

Description: The type of the evaluated dBASE expression is returned.

Parameters:

expr A pointer to the expression's **EXPR4** structure.

Returns: The specific format returned is the format of the information returned when the expression is evaluated using function **expr4vary**.

r4date A date formatted as a character array in "CCYYMMDD" format.

r4dateDoub A Julian date, formatted as a double, is returned. A **(double)** 0 value represents a blank date.

r4log An integer with a true (non-zero) or false (zero) value.

r4num A numeric value formatted as displayable characters.

r4numDoub A numeric value formatted as a **(double)**.

r4str A string of characters.

See Also: **expr4vary**, **expr4double**

```
/*ex93.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */
```

```
void showExpr( EXPR4 *ex )
{
    switch( expr4type( ex ) )
    {
       case  r4date:
             printf( "type is r4date\n" ) ;
             break ;
       case  r4dateDoub:
             printf( "type is r4dateDoub\n" ) ;
             break ;
       case  r4log:
             printf( "type is r4log\n" ) ;
             break ;
       case  r4num:
             printf( "type is r4num\n" ) ;
             break ;
       case  r4numDoub:
             printf( "type is r4numDoub\n" ) ;
             break ;
       case  r4str:
             printf( "type is r4str\n" ) ;
             break ;
       case  r4memo:
             printf( "type is r4m emo\n" ) ;
             break ;
    }
 }

void main( )
{
    CODE4 cb ;
    DATA4 *db ;
    EXPR4 *ex ;

    code4init( &cb ) ;
    db = d4open( &cb, "info" ) ;
    d4top( db ) ;

    ex = expr4parse( db, "NAME" ) ;
    showExpr( ex ) ;
    expr4free( ex ) ;

    ex = expr4parse( db, "AGE" ) ;
    showExpr( ex ) ;
    expr4free( ex ) ;

    ex = expr4parse( db, "BIRTH_DATE" ) ;
    showExpr( ex ) ;
    expr4free( ex ) ;

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

expr4vary

Usage: int expr4vary(EXPR4 *expr, const char **result)

Description: The dBASE expression parsed by **expr4parse** is evaluated for the current record. A pointer to the result is assigned to **result* . Memory for the result is allocated by CodeBase. However, the result should be used immediately, since the memory containing the result is overwritten when another expression evaluation function is called.

The exact result type is chosen to reduce the number of conversions to an absolute minimum. That is, whenever the expression contains a non-character calculation, a conversion to a numeric or a date value is necessary to evaluate the result. Rather than wasting time converting the integer or double back to characters, the numeric result is returned.

Consequently, **expr4vary** has more than one way in which it formats Numeric or Date results. The exact format of the result may be determined by calling **expr4type**.

Parameters:

expr A pointer to the expression's **EXPR4** structure.

result This is the address of the pointer to the results.

Returns: The length of the resulting value is returned. A negative return indicates an error. For more information, refer to **expr4len.**

See Also: **expr4len**, **expr4type**, **expr4double**

Example: See **expr4type**

Field Functions

f4assign	f4long
f4assignChar	f4memoAssign
f4assignDouble	f4memoAssignN
f4assignField	f4memoFree
f4assignInt	f4memoLen
f4assignLong	f4memoNcpy
f4assignN	f4memoPtr
f4assignPtr	f4memoStr
f4blank	f4name
f4char	f4nCpy
f4data	f4number
f4decimals	f4ptr
f4double	f4str
f4int	f4true
f4len	f4type

The field functions are used to access and store information in the data file record buffers and to obtain information about fields.

Access to the contents of a database field depend upon the database being positioned to a valid record. That is, no assignments or retrieval of information may be done if the database is just opened or created and has not been explicitly positioned (e.g. calling **d4top**, **d4go**, etc.), if the database is in an End of File condition, or if the database is in any other invalid position as described by the data file functions.

Field Types

dBASE data files, including those created and used by CodeBase, have several possible field types:

Character Fields

Character fields usually store character information. The maximum width, for dBASE/FoxPro file compatibility, is 254 characters. However, you can increase the width to the CodeBase maximum, 32K, and still maintain Clipper data file compatibility.

CodeBase lets you store any binary data, including normal alphanumeric characters, in a Character field.

Date Fields

Date fields, of width 8, contain information in the following character format: CCYYMMDD (Century, Year, Month, Day).

Eg. "19900430" is April, 30th, 1990

For more information on dates refer to the date functions chapter in the User's Guide.

Floating Point Fields

dBASE IV introduced this field type. With regard to how data is stored in the data file, this field type is identical to a Numeric field. CodeBase treats this field type as a Numeric field.

Logical

This field type, of width 1, stores logical data as one of the following characters: Y, y, N, n, T, t, F or f.

Memo Fields

This is a memo field. It is more complicated than other field types because the variable length memo field data is stored in a separate memo

file. The data file contains a ten byte reference into the memo file.

By using the memo field functions this extra complexity is hidden. From a user perspective, the memo fields are similar to Character fields.

There can be lots of data for a single memo field entry. Most 16 bit compilers are limited to 64K memo entries, while 32 bit compilers can store gigabytes per memo entry. A memo entry may store binary as well as character data.

The field functions do not manipulate the memo entries. In order to manipulate memo entries, refer to the memo field functions.

Numeric Fields

This field type is used to store numbers in character format. The maximum length of the field depends on the format of the file.

File Format	Field Length	Maximum Number of Decimals
Clipper	1 to 19	minimum of (length - 2) and 15
FoxPro	1 to 20	(length - 1)
dBASE IV	1 to 20	(length - 2)

In the data file, the numbers are represented by using the following characters: '+', '-', '.', and '0' through '9'.

Binary Fields

CodeBase treats this field type as though it was a memo field, except that the associated memo file contains binary information. The memo field functions should be used to manipulate the binary entry. This field type provides compatibility with other products that can manipulate binary fields.

General Fields

CodeBase treats this field type as though it was a memo field, except that the associated memo file contains OLEs. This field type is not directly supported by CodeBase, but it provides compatibility with other products, such as FoxPro, which can manipulate OLEs.

Note

Since the dBASE data file standard (used by Clipper and FoxPro) stores all information in the data file as characters, the character based assignment and retrieval functions may be used no matter the defined type of the field.

The Record Buffer

For those interested in dBASE data file internals, field information is stored consecutively without any kind of separator.

Example Record:

"*T19900430Mary17.2"

The first byte represents the single character deletion flag in which the '*' means the record is marked for deletion. Next comes the character 'T' which is likely to be logical field data. Following the 'T' is "19000430"

which could correspond to a date field. The data "Mary" is likely to correspond to a Character field of width 4. Finally, "17.2" is probably a numeric field with a width of 4 and 1 decimal.

```
/*ex94.c*/
#include "d4all.h"
extern unsigned _stklen = 10000;


void main( void )
{
    CODE4  cb ;
    DATA4  *info ;
    FIELD4 *birthDate ;
     char today[8], result[13] ;
     long ageInDays ;
     int rc ;

     code4init( &cb ) ;
     cb.accessMode = OPEN4DENY_RW ;
     info = d4open( &cb , "INFO" ) ;
     birthDate = d4field( info, "BIRTH_DATE" ) ;
    error4exitTest( &cb ) ;

    d4go( info, 1L ) ;

    date4today( today ) ;

     ageInDays =  date4long( today ) - date4long( f4str( birthDate ) ) ;

     printf( "Age in days: %d\n", ageInDays ) ;

    /* display all current birth dates in a formatted manner */
    for(rc = d4top( info ); rc == 0; rc = d4skip( info, 1L ) )
    {
         date4format( f4str( birthDate ), result, "MMM DD, CCYY" ) ;
         printf( "%s\n", result ) ;
    }


    /* assign today's date to all birth dates in the data file*/

     for( rc = d4top( info ); rc == 0; rc = d4skip( info, 1L ) )
    {
       f4assign( birthDate, today ) ;
     }

    code4initUndo( &cb );
}
```

The f4memo Functions

Many memo field functions are similar to corresponding field functions except that they make memo file entries act like the contents of a Character field. Note that memo entries are stored in separate memo files. All that is kept in the data file is a reference to the memo file.

The memo field functions support all of the field types. Consequently it is best to use them when writing generic functions which need to work with all field types.

Field Function Reference

f4assign

Usage: void f4assign(FIELD4 *field, const char *ptr)

Description: The existing field's value is replaced by a null terminated character array pointed to by *ptr*.

If the length of the new data is less than the field length then the extra space in the field is filled with blanks. On the other hand, if *ptr* points to too much data then the extra data is ignored.

WARNING

CodeBase does not do any field type checking. It is the programmer's responsibility to ensure that appropriate data is being assigned. Refer to **f4assignDouble**.
Example Error:

```
/* This creates a formatting problem unless
/* the numeric field is of width two with zero decimals.
f4assign( numericFieldPtr, "33");
```

Parameters:

field A pointer to a field's **FIELD4** structure.

ptr A pointer to a null terminated string.

See Also: **f4assignDouble**

f4assignChar

Usage: void f4assignChar (FIELD4 *field, int chr)

Description: The contents of the specified field are replaced by the single character *chr*. If the field width is greater than one then the extra characters are filled with blanks.

Parameters:

field A pointer to a field's **FIELD4** structure.

chr A character that will replace the first character of the field.

See Also: **f4char**

f4assignDouble

Usage: void f4assignDouble(FIELD4 *field, double value)

Description: The contents of the specified field are replaced by parameter *value*. There is right justified formatting.

If the field is of type Numeric or Floating Point, the number of decimals is used to help determine the formatting. Otherwise, zero decimals are used.

Parameters:

field A pointer to a field's **FIELD4** structure.

value A **(double)** value that will be assigned to the field.

See Also: **f4double**

f4assignField

Usage: void f4assignField(FIELD4 *fieldTo, const FIELD4 *fieldFrom)

Description: The contents of *fieldTo* are replaced by the contents of *fieldFrom*.

Parameters:

fieldTo A pointer to a field's **FIELD4** structure whose contents will be replaced.

fieldFrom A pointer to a field's **FIELD4** structure whose contents will be copied.

Type of *fieldTo*	Copying Method
Character	The characters in *fieldFrom* are copied into *fieldTo* regardless of the type of *fieldFrom*. If *fieldTo* has a longer width, it is padded with blanks.
Numeric or Floating Point	If *fieldFrom* is of type Numeric or Floating Point and *fieldFrom* has the same number of decimals and the same width, then the value in *fieldFrom* is efficiently copied into *fieldTo*. Otherwise, regardless of the type of *fieldFrom*, the data in *fieldFrom* is converted into a **double** using **f4double** and then assigned using **f4assignDouble**.
Date	Information is copied only if *fieldFrom* is of type Date.
Logical	Information is copied only if *fieldFrom* is of type Logical.
Memo, Binary or General	Nothing is copied if *fieldTo* is of type Memo, Binary or General. An error is also generated.

```
/*ex95.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ;

void main( void )
{
    CODE4 cb ;
    DATA4 *info, *data ;
     FIELD4 *infoName, *dataLname ;

     code4init( &cb ) ;
     cb.accessMode = OPEN4DENY_RW ;
     info = d4open( &cb, "INFO" ) ;
    data = d4open( &cb, "DATA" ) ;
    error4exitTest( &cb ) ;

     infoName = d4field( info, "NAME" ) ;
     dataLname = d4field( data, "LNAME" ) ;

    for( d4top( info ), d4top( data ) ; !d4eof( info ) && !d4eof( data ) ;
                                                    d4skip( info, 1 ), d4skip( data, 1 ) )
       f4assignField( infoName, dataLname ) ; /* copy 'LNAME' into 'NAME' */

    code4initUndo( &cb ) ;
}
```

f4assignInt

Usage: void f4assignInt (FIELD4 *field, int value)

Description: The contents of the specified field are replaced by the parameter *value*. There is right justified formatting.

If the field is of type Numeric or Floating Point then any decimals are filled with zeroes.

Parameters:

field A pointer to a field's **FIELD4** structure.

value An **(integer)** that will replace the value in the field.

See Also: **f4int**

f4assignLong

Usage: void f4assignLong (FIELD4 *field, long value)

Description: The contents of the specified field are replaced by the parameter *value*. There is right justified formatting.

If the field is of type Date then *value* should represent a Julian day.

If the field is of type Numeric or Floating Point then any decimals are filled with zeroes.

Parameters:

field A pointer to a field's **FIELD4** structure.

value A **(long)** that will replace the value in the field.

See Also: **f4long**

f4assignN

Usage: void f4assignN (FIELD4 *field, const char *ptr, unsigned ptrLen)

Description: The contents of the specified field are replaced by the character array pointed to by *ptr*.

Parameters:

field A pointer to a field's **FIELD4** structure.

ptr A pointer to a character array of length *ptrLen*. If the length of the new data is less than the field length, then the extra space in the field is filled with blanks. On the other hand, if *ptr* points to too much data then the extra data is ignored.

ptrLen The length of the character array.

f4assignPtr

Usage: char *f4assignPtr (FIELD4 *field)

Description: This function is identical to the function **f4ptr.** The difference is that the **recordChanged** flag is set. Consequently, CodeBase will assume that the record buffer is changed and thus it will flush it at the appropriate time.

Parameters:

field A pointer to a field's **FIELD4** structure.

See Also: **f4ptr**

f4blank

Usage: void f4blank (FIELD4 *field)

Description: The contents of the specified field are filled with blanks.

Parameters:

field A pointer to a field's **FIELD4** structure.

See Also: **f4assign**

f4char

Usage: int f4char (const FIELD4 *field)

Description: The first character of the field is returned as an **(integer).**

Parameters:

field A pointer to a field's **FIELD4** structure.

See Also: **f4assignChar**

f4data

Usage: DATA4 *f4data(const FIELD4 *field)

Description: This function returns a pointer to **DATA4** structure corresponding to the field.

```
/*ex96.c*/
#include "d4all.h"

void displayFieldStats( FIELD4 *f )
{
    DATA4 *db ;

    db = f4data( f ) ;
    printf( "-----------------------------------------------------\n") ;
    printf( "DataFile: %s Field: %s\n", d4alias( db ), f4name( f ) ) ;
    printf( "Length: %d     Type : %c\n", f4len( f    ), f4type( f ) ) ;
    printf( "Decimals: %d\n", f4decimals( f ) ) ;
    printf( "-----------------------------------------------------\n" ) ;
    return ;
}
void main()
{
    CODE4 cb ;
    DATA4 *data ;
    FIELD4 *field ;

    code4init( &cb ) ;
    data = d4open( &cb, "INFO" ) ;
    field = d4field( data, "NAME" ) ;

    displayFieldStats( field );
    code4initUndo( &cb ) ;
}
```

f4decimals

Usage: int f4decimals(const FIELD4 *field)

Description: **f4decimals** returns the number of decimals in the field. This number is always zero for any type other than Numeric or Floating Point fields.

See Also: **f4type**

Example: See **f4data**

f4double

Usage: double f4double(const FIELD4 *field)

Description: The value of the field is returned as a **(double).** This function assumes that the field contains a numeric value and converts that value into a **(double).**

Note

There is a replacement function for Borland compiler development using pre-built Microsoft DLL's under windows. Refer to the COMPILER.TXT file.

See Also: **f4assignDouble**

f4int

Usage: int f4int(const FIELD4 *field)

Description: The value of the field is returned as an **(integer). f4int** also works for Character fields containing numeric values since **f4int** assumes the value of the field is a number.

Any decimals are truncated. If the value of the field overflows the maximum value which can be contained by an integer then the result is undefined.

See Also: **f4assignInt**

f4len

Usage: unsigned f4len(const FIELD4 *field)

Description: The length of the field is returned. This is the length specified for the field when the data file was originally created.

```
/*ex97.c*/
#include "d4all.h"

char *createBufCopy( FIELD4 *f )
{
    char *buf ;

     buf = (char *) malloc(f4len( f ) +1 ) ;
    memcpy( buf, f4ptr( f ), f4len( f ) );
    buf[f4len( f )] = 0;
    return buf ;
}
void main()
{
     CODE4 cb ;
     DATA4 *data;
     FIELD4 *field ;
     char *buffer ;

     code4init( &cb ) ;
     data = d4open( &cb, "INFO" ) ;
     field = d4field( data, "NAME" ) ;
     d4top( data ) ;
     buffer = createBufCopy( field ) ;
     printf( "the copy of the buffer is %s\n", buffer ) ;
     code4initUndo( &cb ) ;
}
```

f4long

Usage: long f4long(const FIELD4 *field)

Description: The value of the field is returned as a **(long)** . However, the value returned depends on the type of the field.

Specifically if the field is of type Date then the date is returned as a Julian day. **f4long** also works for Character fields containing numeric values since **f4long** assumes the value of the field is a number.

Any decimals are truncated. If the value of the field overflows the maximum value which can be contained by a long integer then the result is undefined.

See Also: **date4long**, **f4assignLong**

f4memoAssign

Usage: int f4memoAssign (FIELD4 *field, const char *ptr)

Description: This function assigns a character string to a memo entry. It is the same as **f4assign** except that it supports memo fields.

The memo entry is automatically flushed to disk at a later time. In order to update the disk file immediately use the **d4flush** function.

Returns:

r4success Success.

< 0 Error.

See Also: **f4memoAssignN, f4assign, d4flush**

f4memoAssignN

Usage: int f4memoAssignN(FIELD4 *field, const char *ptr, unsigned ptrLen)

Description: This function assigns the information pointed to by *ptr* to a memo entry. *ptrLen* represents the length of the information, in bytes. The information may be in any format and it may contain null characters..

The memo entry is automatically flushed to disk at a later time. In order to update the disk file immediately use the **d4flush** function.

Returns:

r4success Success.

<0 Error.

See Also: **f4memoAssign, f4assign, d4flush**

f4memoFree

Usage: int f4memoFree (FIELD4 *field)

Description: This function explicitly causes CodeBase to free internal CodeBase memory corresponding to the memo field. It is not generally necessary to use this function since the internal CodeBase memory is freed automatically when the data file is closed.

However, it is worthwhile to use this function if the application is short of memory and the memo entries are large.

Note

The next time the memo field is used, internal CodeBase memory corresponding to the memo field is automatically allocated again.

WARNING

Any changes made to the memo entry that have not yet been flushed to disk will be lost when calling **f4memoFree**. To avoid data loss, use the **d4flush** function before using **f4memoFree**.

Returns:

r4success Success.

<0 An error can occur if a bad parameter is passed to the function.

See Also: **d4flush**

f4memoLen

Usage: unsigned f4memoLen(FIELD4 *field)

Description: This function is used to determine the length of the field or memo entry. If the field does not refer to a memo field, then the length of the field is returned.

Returns:

> 0 The length of the field or memo entry, in bytes, is returned.

<= 0 If the length could not be determined, zero is returned. Check **CODE4.errorCode** for a negative value to determine if an error has occurred. A return of zero may also indicate that there is no memo entry associated with the memo field.

```
/*ex98.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
     FIELD4 *comments ;
     long count ;

     code4init( &cb ) ;
     data = d4open( &cb , "DATA3" ) ;
     comments = d4field( data, "COMMENTS" ) ;

    error4exitTest( &cb ) ;
     count = 0 ;
     for( d4top( data  ) ; !d4eof( data ) ; d4skip( data, 1) )
         if( f4memoLen( comments ) )
          count ++ ;

    printf( "There were %ld memo entries out of %ld records\n", count,
                                                                    d4recCount( data ));
     code4initUndo( &cb ) ;
}
```

f4memoNcpy

Usage: unsigned f4memoNcpy(FIELD4 *field, char *memPtr, unsigned memLen)

Description: This function copies the field's contents into a character array *memPtr*.

f4memoNcpy guarantees that *memPtr* will be null terminated provided that *memLen* is greater than zero. The function **f4memoNcpy** also guarantees that memory will not be overwritten when the **sizeof** operator is used as the parameter *memLen* . To do this **f4memoNcpy** simply does not copy all of the field data if the *memPtr* array is not large enough.

Returns:

> 0 The number of characters actually copied is returned. This value is always less than *memLen* (unless *memLen* is zero) due to the null termination.

0 Zero is returned if an error occurred or if *memLen* was zero. The **CODE4.errorCode** can be checked to determine which possibility occurred.

See Also: **f4nCpy**, **f4memoLen**

f4memoPtr

Usage: char *f4memoPtr(FIELD4 *field)

Description: Function **f4memoPtr** returns a pointer to a null terminated character array containing the memo entry. The memo entry is read in from disk if necessary.

If the field is not a memo type then **f4memoPtr** acts like **f4ptr**.

WARNING

The value returned by **f4memoPtr** can become obsolete when the memo field is assigned a new value or when the record buffer is changed by a data file function. Once this happens the returned pointer must not be used.

Returns:

Not Null Any non-null value is a valid pointer to the field.

Null **f4memoPtr** returns this value when either an error occurs or when the corresponding record is locked by another user. **CODE4.errorCode** can be used to determine if it is an error condition.

See Also: **f4ptr**, **f4memoStr**

```
/*ex99.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
     FIELD4 *comments, *name ;

     code4init( &cb ) ;
```

```
    data = d4open( &cb, "DATA3" ) ;
    comments = d4field( data, "COMMENTS" ) ;
    name = d4field( data, "NAME" ) ;

    d4top( data ) ;
    /* display the null terminated contents of the memo field*/
   printf( "Memo field contents: %s", f4memoPtr( comments ) ) ;

    /* display the non-null terminated contents of the NAME field.
        this displays NAME plus any additional fields in the record buffer*/
    printf( "NAME field contents: %s", f4ptr( name ) ) ;

    code4initUndo( &cb ) ; /* close all files and free memory */
}
```

f4memoStr

Usage: const char *f4memoStr(FIELD4 *field)

Description: The function returns a pointer to a copy of the field's contents. This copy is null terminated.

The function is implemented by calling **f4memoPtr** if the field is a memo type. Otherwise it is implemented by calling **f4str**.

Returns:

Not Null A pointer to a null terminated version of the field is returned.

Null **f4memoStr** returns this value when either an error occurs or when the memo is locked by another user. **CODE4.errorCode** can be used to determine if it is an error.

See Also: **f4memoPtr, f4str, f4memoLen**

```
/*ex100.c*/
#include "d4all.h"

void displayTheRecord( DATA4 *d )
{
    int numFields, curField = 1 ;
    FIELD4 *genericField ;

    numFields = d4numFields( d );

    for( ; curField <= numFields; curField++ )
    {
       genericField = d4fieldJ( d, curField ) ;
       printf("%s\t", f4memoStr( genericField ) ) ;
    }
    printf( "\n" ) ;
}
void main()
{
    CODE4 cb ;
    DATA4 *data ;

    code4init( &cb ) ;
    data = d4open( &cb, "INFO" ) ;
    d4top( data ) ;
    displayTheRecord( data ) ;
    code4initUndo( &cb ) ;
}
```

f4name

Usage: const char *f4name(const FIELD4 *field)

Description: The name of the field is returned as a null terminated character pointer. This is the name of the field originally specified when the data file was created.

```
/*ex101.c*/
#include "d4all.h"

void main()
{
    CODE4 cb ;
    DATA4 *data ;
    FIELD4 *field ;

    code4init( &cb );
    data = d4open( &cb, "INFO" ) ;
    field = d4fieldJ( data, 1 );
    printf( "the first field is called %s\n", f4name( field ) ) ;
    code4initUndo( &cb  ) ;
}
```

f4nCpy

Usage: unsigned f4nCpy(FIELD4 *field, char *memPtr, unsigned memLen)

Description: This function copies the field's contents into a character array *memPtr*.

f4nCpy guarantees that *memPtr* will be null terminated provided that *memLen* is greater than zero. The function **f4nCpy** also guarantees that memory will not be overwritten when the **sizeof** operator is used as the parameter *memLen*. To do this **f4nCpy** simply does not copy all of the field data if the *memPtr* array is not large enough.

Returns:

> 0 The number of characters actually copied is returned. This value is always less than *memLen* (unless *memLen* is zero) due to the null termination.

0 Zero is returned if an error occurred or if *memLen* was zero. The **CODE4.errorCode** can be checked to determine which possibility occurred.

See Also: **f4memoNcpy, u4ncpy**

f4number

Usage: int f4number(const FIELD4 *field)

Description: **f4number** returns the position of the current field in the data file.

For example, if a data file had three fields (ordered LNAME, FNAME and ADDRESS) and FNAME field is being referenced then the function would return 2.

Returns: **f4number** returns the position of the field being referenced. This number is always greater than or equal to 1 and less than or equal to **d4numFields**.

f4ptr

Usage: char *f4ptr(const FIELD4 *field)

Description: This function returns a character pointer to the field in the record buffer.

f4ptr does not return a null terminated string, so **f4len** is often used in conjunction with **f4ptr**.

For a null terminated copy of the field, use the function **f4str**.

WARNING

If the corresponding database is closed and then reopened, the pointer must be reassigned.

See Also: **f4assignPtr, f4str, f4len**

f4str

Usage: char *f4str(FIELD4 *field)

Description: The field's contents are copied to an internal CodeBase buffer and a pointer to the buffer is returned. The buffer is terminated by a null character.

WARNING

The buffer is overlaid with data from the new field each time **f4str** is called. Consequently if the field's value needs to be saved, it is necessary to copy the field's value to a memory area declared by the application.
INCORRECT Example:

```
printf( "Field One %s Field Two %s", f4str(d4fieldJ(data, 1)),
                                                f4str(d4fieldJ(data,
2)));
```

In the above example, **f4str** is evaluated twice before **printf** is called. Since **f4str** always returns the same pointer to the same internal buffer, either field one or field two is printed out twice depending on which parameter is evaluated first.
Refer to the CORRECT example below.

Returns:

!0 A pointer to the field's value is returned.

0 An error has occurred.

See Also: **f4memoStr**

```
/*ex102.c*/
#include "d4all.h"
void main ()
{
    CODE4 cb ;
    DATA4 *data ;
    FIELD4 *field1, *field2 ;

    code4init( &cb ) ;
    data = d4open( &cb , "INFO" ) ;
    field1 = d4fieldJ( data, 1 ) ;
    field2 = d4fieldJ( data, 2 ) ;
    d4top( data ) ;
    printf( "\nField 1: %s", f4str(field1) );
    printf( "\nField 2: %s", f4str(field2) );
    code4initUndo( &cb ) ;
}
```

f4true

Usage: int f4true(const FIELD4 *field)

Description: This function **f4true** is used to determine if a logical field is true of false.

Returns:

1	The field is true.
0	The field is false.
<0	An error has occurred.

f4type

Usage: int f4type(const FIELD4 *field)

Description: The type of the field, as defined when the data file was created, is returned.

Returns:

r4bin or 'B'	Binary Field
r4str or 'C'	Character Field
r4date or 'D'	Date Field
r4float or 'F'	Floating Point Field
r4gen or 'G'	General Field
r4log or 'L'	Logical Field
r4memo or 'M'	Memo Field
r4num or 'N'	Numeric or Floating Point Field

File Functions

file4close	file4optimize
file4create	file4optimizeWrite
file4flush	file4read
file4len	file4readAll
file4lenSet	file4refresh
file4lock	file4replace
file4name	file4unlock
file4open	file4write

The file functions are used to perform low-level file operations such as creating, opening, reading, writing, locking and so forth.

The main advantage of using the file functions instead of the standard C runtime functions is that they implement the CodeBase memory optimizations and error handling.

CodeBase uses these functions internally instead of the standard C runtime functions to provide greater portability between environments and compilers.

Note

When using the file functions in a shared environment, it is the programmers responsibility to lock the files prior to calling **file4write**. In addition, files marked for optimization (using **file4optimize**) are not actually optimized until **code4optStart** is called and the file is locked.

```
/*ex103.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    FILE4 file ;
    char readInfo[50] ;
     unsigned lenRead ;

     code4init( &cb ) ;

    file4create( &file, &cb, "TEXT.FIL", 0 ) ;
     error4exitTest( &cb ) ;

    file4write( &file, 0, "Some File Information", 21 ) ;
     lenRead = file4read( &file, 10, readInfo, sizeof( readInfo) ) ;

    if( memcmp (readInfo, "Information" , lenRead ) == 0 )
       printf( "This is always true\n" ) ;
    if( lenRead == 11 )
       printf( "This is always true, too\n" ) ;
    file4close( file ) ;
    code4initUndo( &cb ) ;
}
```

File Function Reference
file4close

Usage: int file4close(FILE4 *file)

Description: The file is flushed to disk and closed. All optimizations and locks for the file are removed prior to the closing of the file. If the file is already closed, nothing happens.

Returns:

r4success Success.

< 0 An error occurred while attempting to close the file. A low-level C function returned an error value.

See Also: **file4optimize**, **file4open**

```
/*ex104.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ;  /* for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    FILE4 autoexec ;

     code4init( &cb ) ;
     file4open( &autoexec, &cb, "C:\\AUTOEXEC.BAT", 0 ) ;

    file4lock( &autoexec, 0, file4len( &autoexec ) ) ; /* lock the entire file*/

    /* ... some other code */

    file4close( &autoexec ) ; /* save changes and close the file*/
    code4initUndo( &cb ) ;
}
```

file4create

Usage: int file4create(FILE4 *file, CODE4 *code, const char *name, int doAlloc)

Description: A new file is created and opened. If **CODE4.createTemp** is true (non-zero), **file4create** creates a file that is automatically deleted from disk when it is closed.

Parameters:

code This **CODE4** reference is used for memory allocation for the file, optimization, and error messages.

name This null terminated string contains the name of the file to be created. If *name* does not contain a drive and/or directory, the current drive and directory are used.

If *name* is not provided, CodeBase creates a temporary file in the system's temporary directory with a unique name. This file name may be retrieved using **file4name**. CodeBase automatically allocates memory for this file name as the file is created, and frees it when the file is closed.

doAlloc The *doAlloc* flag determines whether memory should be allocated to store a copy of *name*, or whether simply the pointer *name* should be stored. If *doAlloc* is false (zero), only the pointer is stored. If *name* points to temporary memory, set *doAlloc* to true (non-zero).

Returns:

r4success Success.

r4noCreate The new file could not be created. This is usually due to attempting to create a file over an existing file of the same name with the **CODE4.safety** flag set to true (non-zero). This only occurs if **CODE4.errCreate** is false (zero).

< 0 Error.

See Also: **file4name, file4close**

```
/*ex105.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers*/

void main( void )
{
    CODE4 cb ;
    FILE4 textFile ;
     int rc ;

     code4init( &cb ) ;
    cb.accessMode = OPEN4DENY_RW ;
    cb.errCreate = 0 ; /* Handle the errors at the application level*/
     rc = file4create( &textFile, &cb, "C:\\TEMP\\TEXT.FIL", 0 ) ;

    if( rc < 0 || rc == r4noCreate)
    {
       printf( "File 'TEXT.FIL' NOT created\n") ;
       code4exit( &cb ) ;
    }
    file4write( &textFile, 0, "Some Sample Text", 16 ) ;

    file4close( &textFile ) ;
    code4initUndo( &cb ) ; /* flush changes and close file. */
}
```

file4flush

Usage: int file4flush(FILE4 *file)

Description: This function flushes all buffers to disk for the specified file. This ensures that file changes have been saved thus avoiding data loss and allowing other users access to these changes.

Since CodeBase automatically performs file flushing, it is generally unnecessary to call this function. Files are automatically flushed when they are closed and when memory optimizations are turned off. This function is useful when the file is being write-optimized or when Microsoft Windows is being used and data is saved to a local drive. In both of these situations, data is sometimes not immediately written to disk after a **file4write** call has been performed.

WARNING

This function does not work as expected with some cache software -- in particular RAM disk software. To determine whether **file4flush** works for any particular operating system configuration, flush some information to the file and turn the computer's power off. Then turn the computer back on and check if the information is present.

Returns:

r4success Success.

< 0 Error.

Locking: Any locking required for the flush is done. After the flush is completed, the file is unlocked according to **code4unlockAuto**.

See Also: **code4optStart**, **CODE4.fileFlush**, **file4write, file4refresh**

```
/*ex106.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    FILE4 testFile ;

    code4init( &cb ) ;
     cb.safety =  0 ;
    file4create( &testFile, &cb, "TEMP.FIL", 0 ) ;
    code4optStart( &cb ) ;
    file4write( &testFile, 0, "Is this information written?", 27 ) ;
     /* Written to memory, not disk*/

    file4flush( &testFile ) ; /* Physically write to disk*/

    printf( "Flushing complete. \n") ;
     printf( "Check TEMP.FIL after you power off the computer.\n") ;

     code4initUndo( &cb ) ;
}
```

file4len

Usage: long file4len(FILE4 *file)

Description: The length of the file is returned.

If the file is memory optimized, the returned length may differ from the physical file length.

Returns:

>=0 The length of the file is returned.

< 0 Error.

```
/*ex107.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers */

void main( void )
{
    CODE4 cb ;
    FILE4 testFile ;

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

    /* overwrite existing file if it exists */
    file4create( &testFile, &cb, "TEST.FIL", 0 ) ;

    file4write( &testFile, 0, "0123456789", sizeof( "0123456789" ) ) ;
    printf( "Length of the file is: %ld", file4len( &testFile ) ) ;

    file4close( &testFile ) ;
    code4initUndo( &cb ) ;
}
```

file4lenSet

Usage: int file4lenSet(FILE *file, long newLen)

Description: The length of the file is changed to *newLen*.

Parameters:

newLen This is the new length, in bytes, of the file referenced by the **FILE4** structure.

Returns:

r4success Success.

< 0 Error.

WARNING

If the file is write optimized, the file length won't be changed on disk until the file changes are flushed. This can sometimes result in delayed "out of disk space" errors. For example, memory buffers could hold hundreds of appended records before determining that some could not be saved to disk.

To avoid this problem, compile the library with the **S4SAFE** switch defined. This results in the disk file length always being explicitly set, giving "out of disk space" errors as soon as an attempt to exceed the disk space occurs.

See Also: **file4len**

Example: See **file4len**

file4lock

Usage: int file4lock(FILE4 *file, long posStart, long numBytes)

Description: The specified byte range is locked. Depending on the setting of **CODE4.lockAttempts**, **file4lock** tries to lock once, several times, or even keeps trying until it succeeds.

In multi-user applications, a portion of the file or the entire file should be locked before any call to **file4write**. If the portion of the file being written to is not locked, other applications reading the file may get corrupted data.

If write optimizations are being used in multi-user applications with the file functions, it is strongly suggested that the entire file be locked prior to calling **file4optimizeWrite** and **file4write**. In addition, the file should not be unlocked until **file4optimizeWrite** is called to disable write optimizations.

Note

If the file was opened with the **CODE4.accessMode** set to either **OPEN4DENY_RW** or **OPEN4DENY_WRITE** or if the file's read only attribute is set, **file4lock** returns **r4success** but actually does nothing. This function also returns **r4success** and does nothing if the application was compiled with the **S4OFF_MULTI** switch.

Parameters:

posStart A byte offset within the file. For example, 0L represents the first byte of the file.

numBytes The number of bytes to lock.

Returns:

r4success Success.

r4locked The bytes were not locked since they were locked by another user.

< 0 Error.

See Also: **file4unlock**, **file4optimizeWrite**

```
/*ex108.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    FILE4 test ;

     code4init( &cb ) ;

    if( file4open( &test, &cb, "TEST.FIL", 0 ) < 0 )
       code4exit( &cb ) ;

    cb.lockAttempts = 20 ; /* attempt a lock 20 times*/
    if( file4lock( &test, 0, LONG_MAX ) != 0 )
    {
       printf( "Unable to lock the file\n" ) ;
       file4close( &test ) ;
       code4initUndo( &cb ) ;
       code4exit( &cb ) ;
    }
    /* double the file size for fun */
    file4lenSet( &test, file4len( &test ) * 2 ) ;
     printf( "the new length of the file is %ld", file4len( &test ) ) ;
    file4close( &test ) ; /* file4close automatically unlocks the file*/
    code4initUndo( &cb ) ;
}
```

file4name

Usage: const char *file4name(FILE4 *file)

Description: This function returns a null terminated string containing the file name of the file. This is either a pointer to a copy of the name used to open/create the file or a pointer to the *name* parameter passed to **file4open** / **file4create**. The *doAlloc* parameter of **file4open** / **file4create** determines which pointer is returned..

Note that the return value differs from that returned by **d4fileName** and **i4fileName**. Both **d4fileName** and **i4fileName** return the file name complete with the file extension and path information whereas **file4name** just returns the information that was passed to **file4open** or **file4create**.

Returns: A null terminated string containing the name of the file.

See Also: **file4open**, **file4create, d4fileName, i4fileName**

```
/*ex109.c*/
#include "d4all.h"

void main()
{
     CODE4 cb ;
```

```
    FILE4 file ;

    code4init( &cb ) ;
    file4open( &file, &cb, "TEXT.FIL", 0 ) ;
    error4exitTest( &cb ) ;
    printf( "File name: %s", file4name( &file ) ) ;
    printf( "Length: %s", file4len( &file ) ) ;
    code4initUndo( &cb ) ;
}
```

file4open

Usage: int file4open(FILE4 *file, CODE4 *cb, const char *name, int doAlloc)

Description: A file is opened for reading and (possibly) writing.

Parameters:

cb This is a pointer to the **CODE4** structure in order to handle optimization, memory allocation, and error messaging in a manner consistent with the rest of CodeBase.

name *name* should be a null terminated string containing the drive, path, and file name (including extension) of the file to open. If the drive and path are not provided in *name*, the current working directory is assumed.

name may point to temporary memory. However, if temporary memory is used, *doAlloc* should be set to true (non-zero).

doAlloc If *doAlloc* is true (non-zero), **file4open** allocates memory and creates a copy of the *name* for future use. Any memory allocated as a result is freed by **file4close**. If *doAlloc* is false (zero) pointer *name* is saved for future use.

Returns:

r4success Success.

r4noOpen The new file could not be opened and the **CODE4.errOpen** is false (zero).

< 0 Error.

Locking: If **CODE4.accessMode** is set to **OPEN4DENY_RW**, the file is opened in exclusive mode, otherwise it is opened in shared mode. If **CODE4.readOnly** is set to a true (non-zero) value, the file is opened as a read only file.

See Also: **CODE4.accessMode**, **CODE4.readOnly**, **CODE4.errOpen**

file4optimize

Usage: int file4optimize(FILE4 *file, int optFlag, int fileType)

Description: The default memory optimizations for the file are replaced with the specified settings. When files are opened using the **FILE4** structure, the **CODE4.optimize** setting is used as the default memory optimization. This function does nothing if the library was built with the **S4OFF_OPTIMIZE** switch defined.

Note

If optimization has been enabled for a file, it is still necessary to call **code4optStart** before optimization actually takes place.

Parameters:

optFlag This flag determines how memory optimizations should be handled for the file. Valid values for *optFlag* are:

OPT4EXCLUSIVE Read-optimize when files are opened exclusively or when the **S4OFF_MULTI** compilation switch is defined. Otherwise, do not read optimize. If **file4optimize** is not called, this is the default value.

OPT4OFF Do not read optimize.

OPT4ALL This is the same as the **OPT4EXCLUSIVE** option except shared files are also read optimized.

fileType Different files are optimized differently. This flag allows CodeBase to optimize the file in the most efficient manner. The possible values are:

Setting	Meaning
OPT4DBF	Data file optimization.
OPT4INDEX	Index file optimization.
OPT4OTHER	Generic optimization.

Returns:

r4success Success.

< 0 Error. **file4flush** returned an error.

Locking: **file4optimize** does no locking. However, if optimizations are disabled after having been enabled, a call to **file4flush** is made internally. To maintain data integrity when disabling optimizations, lock the file.

See Also: **d4optimize**, **file4flush**, **CODE4.optimize**, **CODE4.optimizeWrite**

file4optimizeWrite

Usage: int file4optimizeWrite(FILE4 *file, int optFlag)

Description: The default write optimizations for the file are replaced with the specified settings. This function is identical to **d4optimizeWrite** except that it only applies to the specific file specified by parameter *file*. **CODE4.optimizeWrite** setting is used as the default for write optimization.

This function does nothing if the library was built with the **S4OFF_OPTIMIZE** switch defined.

If optimization has been enabled for a file, it is still necessary to call **code4optStart** before optimization actually takes place.

Note

Note

In general, it is not recommended that shared files be write-optimized unless the file is completely locked. It is the caller's responsibility to ensure that any writes to the file are flushed prior to unlocking.

Parameters:

optFlag This flag determines how memory optimizations should be handled for the file. Valid values for *optFlag* are:

OPT4EXCLUSIVE Write-optimize when files are opened exclusively or when the **S4OFF_MULTI** compilation switch is defined. Otherwise, do not write optimize. If **file4optimizeWrite** is not called, the **CODE4.optimizeWrite** member variable is used as the default.

If a file is not opened exclusively or **S4OFF_MULTI** is not defined, this option disables write optimizations, and any buffered writes are flushed to disk with a call to **file4flush**.

OPT4OFF Do not write optimize. If write optimizations are disabled after having been enabled, buffered writes are flushed to disk using **file4flush**.

OPT4ALL This is the same as the **OPT4EXCLUSIVE** option except shared files are also write optimized. Use this option with care. If concurrently running applications do not lock, they may be presented with inconsistent data. In addition, write optimizations does not improve performance unless several write operations take place.

Returns:

r4success Success.

< 0 Error. **file4flush** returned an error.

Locking: **file4optimizeWrite** does no locking. However, if optimizations are disabled after having been enabled, a call to **file4flush** is made internally. To maintain data integrity when disabling optimizations, it is necessary for the programmer to lock the file.

See Also: **d4optimize**, **file4flush**, **CODE4.optimize**, **CODE4.optimizeWrite**

file4read

Usage: unsigned file4read(FILE4 *file, long pos, void *ptr, unsigned len)

Description: Information is read from the file, starting at byte *pos*, and stored in *ptr*. An attempt is made to read *len* bytes into *ptr*.

If the end of the file is reached before the requested number of bytes are read, as many bytes as possible are read.

Parameters:

pos A byte offset within the file. For example, 0L represents the first byte in the file.

ptr Points to where the file information is to be stored.

len The number of bytes to read from the file.

Returns:

>=0 The number of bytes actually read. If the return is zero, it may be an error condition. **CODE4.errorCode** can be checked to determine whether an error actually occurred.

< 0 An error has occurred.

See Also: **file4readAll**, **file4refresh**, **file4optimize**

```
/*ex110.c*/
#include "d4all.h"

void main()
{
    CODE4 cb ;
    FILE4 file ;
    char buf[11] ;
    int pos ;

    code4init( &cb  ) ;
    file4open( &file, &cb, "TEST.FIL", 0 ) ;

    memset( buf, 0, sizeof( buf ) ) ; /* ensure null termination for output */
    pos = file4read( &file, 0L, buf, sizeof( buf ) - 1 ) ;
   if( cb.errorCode < 0 ) return ;
   printf( "%s\n", buf ) ;
    code4initUndo( &cb ) ;
}
```

file4readAll

Usage: unsigned file4readAll(FILE4 *file, long pos, void *ptr, unsigned len)

Description: Information is read from the file, starting at byte *pos*, and stored in *ptr*. An attempt is made to read *len* bytes into *ptr*.

If the end of the file is reached before the requested number of bytes are read, an error message is generated, and a negative value is returned.

Parameters:

pos A byte offset within the file. For example, 0L represents the first byte in the file.

ptr Points to where the file information is to be stored.

len The number of bytes to read from the file.

Returns:

r4success Success.

< 0 Error. Check the **CODE4.errorCode** setting for the specific error.

See Also: **file4read**

```
/*ex111.c*/
#include "d4all.h"
typedef struct
{
    int id ;
    char password[9] ;
} MY_STRUCT ;

int readUserInfo( FILE4 *file, MY_STRUCT *ms, int user)
{
    int rc ;

    rc = file4readAll( file, user*sizeof(MY_STRUCT), ms, sizeof(MY_STRUCT) ) ;
    if( rc != 0 )
     {
       printf( "Could not read user # %d\n", user ) ;
       return rc ;
    }
    printf( "id: %d password: %s\n", ms->id, ms->password ) ;
    return 0 ;
}

void main()
{
    CODE4 cb ;
    FILE4 testFile ;
    MY_STRUCT *info ;
    int userNum ;

    code4init( &cb  );
    file4open( &testFile, &cb, "TEST.FIL", 0 ) ;
    info = ( MY_STRUCT * )malloc( sizeof( MY_STRUCT ) ) ;
    for (userNum = 0; userNum <= 9 ; userNum++ )
        readUserInfo( &testFile, info, userNum ) ;
    file4close( &testFile ) ;
    code4initUndo( &cb ) ;
}
```

file4refresh

Usage: int file4refresh(FILE4 *file)

Description: This function, which is only relevant to memory optimized files, discards all 'read' information buffered in memory for the file. This ensures that the next file read returns information read directly from disk. In addition, **file4flush** is called to ensure information that is 'write' buffered is written to disk.

When **S4OFF_OPTIMIZE** is defined, all file functions read directly from disk. As a result, **file4refresh** always returns 'Success' when **S4OFF_OPTIMIZE** is defined.

Returns:

r4success Success.

< 0 Error.

See Also: **d4refresh**, **file4flush**, **file4optimize**

```
/*ex112.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ;

void main( void )
{
    CODE4 cb ;
    FILE4 file ;
    char before[6], after[6] ;
```

```
    code4init( &cb ) ;
    file4open( &file, &cb, "TEST.FIL", 0 ) ;
    memset( before, 0, sizeof( before ) ) ;
    memset( after, 0, sizeof( after ) ) ;
    memset( before, ' ', sizeof( before )-1 ) ;
    memset( after, ' ', sizeof( after )-1 ) ;

    file4optimize( &file, 1, OPT4OTHER) ;

    /* read the first 5 bytes and buffer it.*/
    file4read( &file, 0, before, sizeof( before )-1 ) ;
    file4read( &file, 0, after, sizeof( after )-1 ) ; /* read from memory, not disk*/

    if( strcmp( before, after ) )
        printf( "This will always be true, since the read was from memory\n") ;

    printf( "Press ENTER to re-read information" ) ;
    getchar( ) ;

    file4refresh( &file ) ; /* next read will be from disk*/

    file4read( &file, 0, after, sizeof( after )-1 ) ;
    if( strcmp( before, after ) )
        printf( "No changes detected\n" ) ;
    else
        printf( "Good thing it was read from disk... \nsomeone has changed it\n") ;

    file4close( &file ) ;
    code4initUndo( &cb ) ;
}
```

file4replace

Usage: int file4replace(FILE4 *keepName, FILE4 *newFile)

Description: The function renames the file specified by *newFile* with the name from *keepName* and then it closes and deletes the original file specified by *keepName*. Under other circumstances the renaming of *newFile* takes place in different manner. For example in the multi-user configuration, the contents of *keepName* are copied over by the contents of *newFile* and *newFile* is closed and deleted.

This is useful when a temporary copy of a file is made and then the original file needs to be replaced by the temporary file (such as when compressing a memo file).

Parameters: *keepName* and *newFile* both refer to open files.

Returns:

r4success Success.

< 0 Error.

Locking: No locking is done. In a multi-user environment it is the program's responsibility to ensure that the entire original file is locked prior to calling **file4replace**, and that the user has delete and rename privileges for the files.

```
/*ex113.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main(  )
{
    CODE4 cb ;
    FILE4 primary, secondary ;
    int rc ;
```

```
    char buffer[15] ;

    code4init( &cb ) ;
    code4optStart( &cb ) ;
    cb.safety = 0 ;

    file4create( &primary, &cb, "PRI", 0 ) ;
    file4create( &secondary, &cb, "SEC", 0 ) ;

    file4write( &primary,  0, "PRIMARY FILE", 12 ) ;
    file4write( &secondary, 0, "SECONDARY FILE", 14 ) ;

    rc = file4replace( &primary, &secondary ) ;

    if( rc < 0 )
       printf( "An error occurred in file4replace\n" ) ;
    cb.errOpen = 0 ;
    rc = file4open( &secondary, &cb, "SEC", 0 ) ;
    if( rc == 0 )
      printf( "This should never happen\n") ;

    memset( buffer, 0, sizeof( buffer ) )   ;
    memset( buffer, ' ', sizeof( buffer ) - 1 ) ;
   file4read( &primary, 0, buffer, sizeof( buffer )-1 ) ;

   if( strcmp( buffer, "SECONDARY FILE" ) == 0 )
      printf( "This should always be true\n" ) ;

   file4close( &primary ) ;
   code4initUndo( &cb ) ;
}
```

file4unlock

Usage: int file4unlock(FILE4 *file, long posStart, long numBytes)

Description: The specified range of bytes is unlocked. It is the programmer's responsibility to ensure that exactly the same range of bytes were previously locked with function **file4lock**.

Conditional compilation switch **S4LOCK_CHECK** can be used for debugging when using **file4unlock**. This switch adds a check to ensure that each call to **file4unlock** has a corresponding call to **file4lock**.

Parameters:

posStart A byte offset within the file. For example, 0L represents the first byte in the file.

numBytes The number of bytes to lock.

Returns:

r4success Success.

< 0 Error.

See Also: **file4lock**

```
/*ex114.c*/
#include "d4all.h"

int addToFile( CODE4 *cb, FILE4 *file, char *string )
{
    int oldLockAttempts ;
    long fileSize ;

    oldLockAttempts = cb->lockAttempts ;
    cb->lockAttempts = 1 ;
    fileSize = file4len( file ) ;
```

```
    if( file4lock( file, fileSize, LONG_MAX ) == r4locked )
    {
       printf( "Cannot add to file, another user is writing\n") ;
       cb->lockAttempts = oldLockAttempts ;
       return 1 ;
    }
    /* lock succeeded, I may add to the file without corrupting anyone else's
     writes */

    file4write( file, fileSize, string, strlen(string) ) ;

    file4unlock( file, fileSize, LONG_MAX ) ;
    cb->lockAttempts = oldLockAttempts ;
    return 0 ;
}
void main()
{
    CODE4 cb ;
    FILE4 file ;
    char buffer[]= "Add this string to the file" ;

    code4init( &cb );
    file4open( &file, &cb, "TEST.FIL", 0 ) ;

    addToFile( &cb, &file, buffer ) ;
    file4close( &file ) ;
    code4initUndo( &cb ) ;
}
```

file4write

Usage: int file4write(FILE4 *file, long pos, void *ptr, unsigned len)

Description: The information pointed to by *ptr* is written to *file*, beginning at the *pos* byte.

Parameters:

pos The position, within the file, to write the information.

ptr A pointer to the information to be written.

len The number of bytes to write to the file.

Returns:

r4success Success.

< 0 Error.

Note

If the file is write-optimized, **file4write** calls will not usually write the changes directly to disk, but instead will buffer the changes. The changes are written to disk at a later time in order to speed up overall performance.

See Also: **file4flush**, **file4read**, **file4optimizeWrite**

```
/*ex115.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers */
#define DATE_OFFSET 16

void main( void )
{
    CODE4 cb ;
    FILE4 file ;
    char runDate[9] ;

    code4init( &cb ) ;
    cb.accessMode = OPEN4DENY_RW ;
```

```
    date4today( runDate ) ; /* initialize to the system clock*/
     runDate[8] = 0 ;

    if( file4open( &file, &cb, "TEST.FIL", 0 ) !=0 )
      return ;

    /* file is opened exclusively - no need to lock*/
    file4write( &file, DATE_OFFSET, runDate, sizeof(runDate) ) ;
    file4close( &file ) ;
     code4initUndo( &cb );
}
```

Sequential Read Functions

file4seqRead
file4seqReadAll
file4seqReadInit

The file sequential read functions are used to efficiently read information from a file in a sequential manner. The file sequential read functions boost performance by buffering read information in memory. This gives subsequent reads the advantage of quickly reading from memory instead of having to wait for the comparatively slow hardware to locate and read the information.

The file sequential read functions are independent of the memory optimizations used by the file module. Instead of performing customized buffering for data or index files, the data is buffered directly and sequentially using a program-allocated buffer. With the ability to allocate a buffer directly in the calling program, CodeBase sequential read functions provide a low-memory method of buffering read information. It is therefore unnecessary to call **file4optimize** for a sequential read function**.**

It is possible to have more than one sequential read and/or sequential write happening at once on the same file. However, if information is written using sequential writing and then immediately read using sequential reading, there is no guarantee that the read information will be the most current. This is because the buffers is not flushed until it is full or re-read until it is empty.

When the read buffers must be refreshed, the file sequential read functions read directly from disk. However, since they are read in large blocks, instead of smaller chunks, the disk access time is significantly reduced.

The sequential read functions do no locking.

Note

In general, the file functions should be used for file reading and writing, since they use the CodeBase memory optimizations. The sequential read functions are provided for programmers who may be in a low memory situation where the added memory for read optimizations is unavailable.

```
/*ex116.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ;

typedef struct myStructSt
{
    int id ;
    char password[9] ;
} MY_STRUCT ;

void main( void )
{
    CODE4 cb ;
    FILE4SEQ_WRITE writePassFile ;
    FILE4SEQ_READ readPassFile ;
```

```
    FILE4 passFile ;
    MY_STRUCT person ;
     char buffer[ 0x1400 ] ; /* 5K bytes... space for 200 structures */
     int i ;

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

    file4create( &passFile, &cb, "TEST.FIL", 0 ) ;
    file4seqWriteInit( &writePassFile, &passFile, 0, buffer, sizeof( buffer ) ) ;
    for( i = 10 ; i ; i -- )
    {
         person.id = i ;
         strcpy( person.password, "PASSWORD") ;
         person.password[8] = 0 ;
         file4seqWrite( &writePassFile, &person, sizeof( MY_STRUCT ) ) ;
    } /* physically write only once.*/
    file4seqWriteFlush( &writePassFile ) ;

    file4seqReadInit( &readPassFile, &passFile, 0, buffer, sizeof(buffer) );

    for( i = 10 ; i ; i -- )
    {
       /* only one physical read occurs... the rest are in memory*/
       file4seqRead( &readPassFile, &person, sizeof( MY_STRUCT ) ) ;
       printf( "id: %d password: %s\n", person.id, person.password ) ;
    }
    file4close( &passFile ) ; /* writePassFile and readPassFile are invalid now*/
    code4initUndo( &cb ) ;
}
```

Sequential Read Function Reference
file4seqRead

Usage: unsigned file4seqRead (FILE4SEQ_READ *seqRead, void *ptr, unsigned len)

Description: Information is read from the file, starting at the next position, and stored in *ptr*. An attempt is made to read *len* bytes. If there are not *len* bytes left in the buffer, a physical disk read occurs to refill the buffer.

Parameters:

seqRead A pointer to a structure containing information for the sequential read access.

ptr This a pointer to the program allocated memory to where len bytes of the buffered file is copied. *ptr* must point to at least len bytes of memory.

len *len* is the number of bytes to copy into *ptr*.

Returns: **file4seqRead** returns the number of bytes actually read. If zero is returned, check **CODE4.errorCode** for a potential error.

See Also: **file4read**

```
/*ex117.c*/
#include "d4all.h"

typedef struct myStructSt
{
    int id ;
     char password[9] ;
} MY_STRUCT ;

int getNextStructure( FILE4SEQ_READ *seqFile, MY_STRUCT *ms )
{
    if( file4seqRead( seqFile, ms, sizeof( MY_STRUCT ) ) != sizeof( MY_STRUCT ) )
     {
```

```
            memset( ms, 0, sizeof(MY_STRUCT ) ) ;
            return 1 ;
    }
    return 0 ;
}
void main()
{
    CODE4 cb ;
    FILE4 passFile ;
    FILE4SEQ_READ readFile ;
    MY_STRUCT *person ;
    char buffer[0x1400] ;

    code4init( &cb ) ;
    file4open( &passFile, &cb, "TEST.FIL", 0 ) ;
    file4seqReadInit( &readFile, &passFile, 0, buffer, sizeof(buffer) );
    person = ( MY_STRUCT *) malloc( sizeof( MY_STRUCT ) ) ;
    getNextStructure( &readFi  le, person ) ;
    if ( person != NULL )
        printf( "id: %d password: %s\n", person->id, person->password ) ;

    file4close( &passFile ) ;
    code4initUndo( &cb  ) ;
}
```

file4seqReadAll

Usage: int file4seqReadAll(FILE4SEQ_READ *seqRead, void *ptr, unsigned len)

Description: Information is read from the file, starting at the next position, into *ptr*.

An attempt is made to read *len* bytes. If there are not *len* bytes left in the buffer, a physical disk read occurs to refill the buffer. If the end of file is reached before the requested number of bytes are read, an error is returned and an error message is generated.

Parameters:

seqRead A pointer to a structure containing information for the sequential read access.

ptr This a pointer to program allocated memory where *len* bytes of the buffered file are copied. *ptr* must point to at least *len* bytes of memory.

len *len* is the number of bytes to copy into *ptr*.

Returns:

r4success Success.

< 0 Error.

See Also: **file4readAll**, **file4seqRead**

file4seqReadInit

Usage: void file4seqReadInit(FILE4SEQ_READ *seqRead, FILE4 *file, long startPos, void *buffer , unsigned bufferLen)

Description: **file4seqReadInit** initializes a file for fast sequential reading. The file must have already been opened or created using the file functions.

Parameters:

seqRead This is a pointer to a structure that is initialized with the information provided by the parameters sent to **file4seqReadInit.** It contains the pertinent data required for the other sequential read functions.

file This specifies the file to be read. The file must previously have been opened or created using this structure. All previous read optimizations for the file are bypassed when sequential read functions are used.

startPos This is the position from which to sequential reading is to begin. This is the number of bytes from the start of the file.

buffer This is a pointer to the program-allocated buffer to be used for the physical disk reads.

bufferLen This is the length, in bytes, of *buffer*. The larger the buffer used, the faster the **file4seqRead** function operates. The number of buffer bytes must be greater than 1024 and should be a multiple of 1024. The memory is only used in multiples of 1024, so any extra memory allocated is ignored.

See Also: **file4seqRead**, **file4open**, **file4create**

Sequential Write Functions

file4seqWrite
file4seqWriteFlush
file4seqWriteInit
file4seqWriteRepeat

The file sequential write functions are used to efficiently write information to a file in a sequential manner. The file sequential write functions boost performance by buffering information in memory before it is written to disk. This gives the advantage of quickly writing to memory instead of continually having to wait for the comparatively slow hardware to locate and write the information.

The file sequential write functions are used to augment the memory optimizations used by the file module. When the memory allocated for the sequential write is filled, **file4write** is called to write to disk. If write optimizations have been enabled for the file, this write will then be buffered in the CodeBase write pool. However, if write optimizations are not enabled, the program-allocated buffer can provide a low-memory method of buffering written information. It is therefore not critical to call **file4optimizeWrite** from a **file4seqWrite** function if write buffering is desired.

It is possible to have more than one sequential read and/or sequential write happening at once on the same file. For example, it is possible to have two sequential writes occurring on the same file. However, if the two sequential writes occur on overlapping sections of the file, the results will be undefined. This is because the buffer is not flushed until it is full or re-written until it is empty.

The sequential write functions do no locking.

Note

In general, the file functions should be used for file reading and writing, since they use the CodeBase memory optimizations. The sequential write functions are provided for programmers who may be in a low memory situation where the added memory for write optimizations is unavailable.

An example of using the sequential write functions may be found in the introduction to sequential read functions.

Sequential Write Function Reference

file4seqWrite

Usage: int file4seqWrite (FILE4SEQ_WRITE *seqWrite, const void *info, unsigned infoLen)

Description: Information is written to the buffer starting at the next position and an attempt is made to write *infoLen* bytes. If there are not *infoLen* bytes left in the buffer or if there is not a buffer provided, a physical disk write occurs to flush changes to the file.

Parameters:

seqWrite A pointer to a structure containing information for the sequential write access.

info This is a pointer to the information to be written to disk.

infoLen *infoLen* is the number of bytes in *info* to write to disk.

Returns:

r4success Success.

<0 Error. Not all of the information was written. Check **CODE4.errorCode** for the error return.

See Also: **file4seqWriteFlush**, **file4write**

Example: See Sequential Read Functions introduction.

file4seqWriteFlush

Usage: int file4seqWriteFlush(FILE4SEQ_WRITE *seqWrite)

Description: Any information written with **file4seqWrite** is flushed to disk.

Returns:

r4success The file was successfully flushed.

< 0 There was an error flushing the file.

See Also: **file4seqWrite**, **file4seqWriteInit**, **file4flush**

Example: See Sequential Read Function introduction

file4seqWriteInit

Usage: void file4seqWriteInit(FILE4SEQ_WRITE *seqWrite, FILE4 *file, long startPos, char *buffer, unsigned bufferLen)

Description: **file4seqWriteInit** initializes a file for fast sequential writing. The file must have already been opened or created using the file functions.

Parameters:

seqWrite This is a pointer to a structure that is initialized with the information provided by the parameters sent to **file4seqWriteInit.** It contains the pertinent data required for the other sequential write functions.

file This is a reference to a previously opened file that is to be used for fast sequential writing. *file* must reference a valid file.

startPos This is the position from which sequential writing is to begin. This is the number of bytes from the start of the file. The default value is the beginning of the file.

buffer This is a pointer to memory that is used to buffer disk writes.

When CodeBase determines that the provided *buffer* is full of written information, it is physically written to disk. If *buffer* is NULL, the sequential write functions do no buffering and call **file4write** directly.

Therefore, if write optimizations are used (see **CODE4.optimize** and **CODE4.optimizeWrite**) and *buffer* is not NULL, a two tiered buffering occurs. When *buffer* is full, it is written using **file4write**, but with CodeBase optimizations active **file4write** also provides buffering as directed through **CODE4.optimizeWrite**.

If write optimizations are disabled and buffering is desired, *buffer* should point to program-allocated memory.

bufferLen This is the length, in bytes, of *buffer*. The larger the buffer used, the faster the **file4seqWrite** function operates. The number of buffer bytes must be greater than 1024 and should be a multiple of 1024 The memory is only used in multiples of 1024, so any extra memory is ignored.

See also: **file4seqWrite**, **file4open**, **file4create**

file4seqWriteRepeat

Usage: int file4seqWriteRepeat(FILE4SEQ_WRITE *seqWrite, long numRepeat, char ch)

Description: A character is repeatedly written to the file.

Parameters:

seqWrite A pointer to a structure containing information for the sequential write access.

numRepeat The number of times to write the character.

ch The character to write.

Returns:

r4success Success.

< 0 Error.

Index Functions

i4close	i4reindex
i4create	i4tag
i4fileName	i4tagAdd
i4open	i4tagInfo

The CodeBase data file functions use the index functions to create sorted orderings of the information contained in data files. The sorted orderings can then be used when searching and skipping through the data file. These functions may be used by application programmers to open and create additional index files.

Each index file, represented by an **INDEX4** structure, can contain an unlimited number of sorted orderings (except **.MDX** indexes which can only store 47). Each of these orderings corresponds to a tag within the index file. When, **i4open**, **i4create** or **d4index** are called, a pointer to the **INDEX4** structure is returned. When other index functions are to be called, this pointer is passed as the first parameter.

When information is written to a data file, all open index files corresponding to the data file are automatically updated.

Index Function Reference

i4close

Usage: int i4close(INDEX4 *index)

Description: **i4close** closes an index file that has been opened with **i4open**. If a production index file has been opened with **d4open**, then the index file must be closed with **d4close**. The index file is flushed to disk if necessary, and then closed. If the record buffer of the corresponding data file has been changed, the record is flushed to disk and the index file is updated before it is closed.

If the index must be updated, **i4close** locks the file, performs the flushing, and then closes the file. **i4close** temporarily sets the **CODE4.lockAttempts** flag to **WAIT4EVER** to ensure the index file is locked and updated before returning. As a result, **i4close** never returns **r4locked**. If **i4close** encounters a non-unique key in a unique index tag while flushing the data file, the index file is closed, but not updated.

An error will be generated if **i4close** is called during a transaction.

Returns:

r4success Success.

< 0 Error.

Locking: If flushing is required, the index file is locked. When **i4close** returns, the file is closed and all locks on the index file are removed.

See Also: **d4close**, **code4close**, **i4open**, **code4tranStart**

```
/*ex118.c*/
#include "d4all.h"

int addLotsOfRecords( DATA4 *d )
{
     INDEX4 *production ;
     int i ;

     production = d4index( d, d4alias( d ) ) ; /* get the production index file*/

     if( production != NULL )
        i4close( production ) ;

    d4top( d ) ;
    for( i = 200 ; i ; i -- )
    {
       d4appendStart( d, 0 ) ;
       d4append( d ) ; /* make 200 copies of record 1*/
    }

    /* open the index file and update it*/
    production = i4open( d, d4alias( d ) ) ;
    return i4reindex( production ) ;
}
void main()
{
     CODE4 cb ;
     DATA4 *data ;

     code4init( &cb ) ;
     data = d4open( &cb, "DATA") ;
     addLotsOfRecords( data ) ;
     code4initUndo( &cb ) ;
}
```

i4create

Usage: INDEX4 *i4create(DATA4 data, const char *fileName, const TAG4INFO *tags)

Description: **i4create** creates a new index file and associates it with the data file *data*. The **TAG4INFO** array is used to specify the sort orderings stored in the index file.

An index file may also be created using **d4create**.

WARNING

In the multi-user configuration, open the data file exclusively, which can be done by setting **CODE4.accessMode** to **OPEN4DENY_RW** before opening or creating the data file. If the data file is not opened exclusively before **i4create** is called, the other applications may not be aware of the newly created index file and as a result the new index file may not be updated correctly.

TAG4INFO structure

The first two members of every **TAG4INFO** structure must be defined. The last three members are used to specify special properties of the tag, which are 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 tag name must be unique to the data file and have a length of ten characters or less.

If you are using the **.NTX** index format, then this name includes the index file name with a path. In this case, the index file 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. Refer to the dBASE Expression appendix for more information on possible key expressions. This is often just a field name. This member cannot be null, except to indicate that there are no more tags.
- **(char *)filter** This is a Logical dBASE expression. If this filter expression evaluates to true (non-zero) for any given data file record, a key for the record is included in the tag. If a null string is specified, keys for all data file records are included in the tag.
- **(int) unique** This integer code specifies how to treat duplicate keys. See below for more information.
- **(int) descending** This flag must be zero or **r4descending**. If it is set to **r4descending**, the keys are in a reverse order compared to how they would otherwise be arranged.

Following is information about the possible values for the **unique** member of **TAG4INFO**. The integers are defined in 'D4DATA.H'.

- **0** Duplicate keys are allowed.
- **r4uniqueContinue** Any duplicate keys are discarded. In this case, there may not be a tag entry for a particular record.
- **e4unique** Generate an **e4unique** error if a duplicate key is encountered.
- **r4unique** Do not generate an error if a duplicate key is encountered. However, the operation is aborted and **r4unique** is returned.

Unique Tags

The dBASE file format, which CodeBase uses, only saves to disk a TRUE/FALSE flag, which indicates whether a tag is unique or non-unique. No information on how to respond to a duplicate key for unique key tags is saved.

If a duplicate key is encountered for a unique key tag, dBASE responds by ensuring there is no corresponding key for the record. Any duplicate keys are ignored and are not saved in the tag. Consequently, there may be records in the data file that do not have a corresponding tag entry.

CodeBase mimics the dBASE response to non-unique keys when **t4uniqueSet** is called with **r4uniqueContinue**. CodeBase also provides extra flexibility by allowing different responses when a non-unique key is encountered in unique key tags. **t4uniqueSet** can accept either **e4unique**, **r4unique** or **r4uniqueContinue** as an argument.

t4uniqueSet can be set directly by passing the desired value to the function or indirectly when an index file is created or opened. When an index file is created, the **TAG4INFO.unique** value is passed to **t4uniqueSet**. When an index file is opened, the **t4uniqueSet** is initialized from **CODE4.errDefaultUnique** for any unique tags.

code4init initializes **CODE4.errDefaultUnique** to **r4uniqueContinue** by default. Note that this setting only applies to unique key tags. For non-unique key tags, **t4uniqueSet** is internally initialized to false (zero), meaning there can be duplicate keys.

See the Indexing chapter of the User's Guide for more information.

Parameters:

data The data file corresponding to the index file to be created.

fileName This is the name of the index file to be created.

When using FoxPro **.CDX** or dBASE IV **.MDX** files, it is possible to create a "production" index file with **i4create** when a data file already exists. This is done by passing a null pointer for *fileName*. In this case, the index file name is the same as the data file name. Open the data file exclusively before calling **i4create**. The data file can be opened exclusively by setting the **CODE4.accessMode** to **OPEN4DENY_RW** before the data file is opened. In this way a production index file, which is automatically opened when the data file is opened, is created.

When the **S4CLIPPER** switch is defined, the *fileName* parameter specifies the name of the index group file. This index group file is filled with a list of the created tag files. If the *fileName* parameter is not provided, all of the tag files are created but no index group file is created. Even if the index group file is not created, CodeBase still creates an **INDEX4** structure, which can be used by all of the index file functions. For more information on group files, refer to the section on Clipper support in the User's Guide.

If a path is provided in the single user or multi-user configuration, it is used. Otherwise, the file is assumed to be in the current directory.

tags This points to an array of **TAG4INFO** structures. The last entry is always null to signal that there are no more tags. Refer to the example below.

Returns:

Not Null Success. A pointer to the corresponding **INDEX4** structure is returned.

Null The index file was not successfully created. Inspect **CODE4.errorCode** for more detailed information. If the **CODE4.errorCode** is negative, then an error has occurred.

Locking: The data file is locked. In general, consider opening a data file exclusively before calling **i4create**. The data file can be opened exclusively by setting **CODE4.accessMode** to **OPEN4DENY_RW** before opening or creating the data file.

See Also: **d4create, CODE4.accessMode**

```
/*ex119.c*/
#include "d4all.h"

static FIELD4INFO fieldInfo[ ] =
{
    { "FIELD_NAME", 'C', 10, 0 },
    { "VALUE", 'N', 7, 2 },
    { 0,0,0,0 }
} ;

TAG4INFO tagInfo[ ] =
{
    { "T_NAME", "FIELD_NAME", "FIELD_NAME > 'A'", 0,0 },
    { "NAME_TWO", "VALUE", "", e4unique, r4descending },
    { 0,0,0,0,0 }
} ;

extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
    INDEX4 *index ;

    code4init( &cb ) ;
    data = d4create( &cb, "DB_NAME", fieldInfo, 0 ) ;
    index = i4create( data, "name", tagInfo ) ;

    d4close( data ) ;
    code4initUndo( &cb ) ;
}
```

i4fileName

Usage: const char *i4fileName(INDEX4 *index)

Description: **i4fileName** returns a null terminated character string containing the index file name complete with the file extension and any path information. This information is returned regardless of whether a file extension or a path was specified in the name parameter passed to **i4open** or **i4create**.

The pointer that is returned by this function, points to internal memory, which may not be valid after the next call to a CodeBase function. Therefore, if the name is needed for an extended period of time, the file name should be copied by the application.

See Also: **d4fileName**, **file4name**

i4open

Usage: INDEX4 *i4open(DATA4 *data, const char *name)

Description: An index file is opened. Note that if **CODE4.autoOpen** is true (non-zero), a production index file is automatically opened when **d4open** is called.

Parameters:

data The data file corresponding to the index file being opened.

name This is normally the name of the index file. Alternatively, if *name* is null, the name of the data file (with the appropriate index file extension) is used as the name of the index file. This feature is used by **d4open** to open production index files.

Opening an index file does not change which tag is selected.

When Clipper index files are being used, by default CodeBase attempts to open a CodeBase group file. However, specifying a **.NTX** Clipper index file name extension for *name*, causes CodeBase to open a single index file. In this case, the index file contains a single tag, which has the same name as the index file.

If a path is provided in the single user or multi-user configuration, it is used. Otherwise, the file is assumed to be in the current directory.

Returns:

Not Null A pointer to the **INDEX4** structure corresponding to the opened index file.

Null The index file could not be opened. This is usually an error condition. However, if **CODE4.errOpen** is false and the file does not exist, then there is no error and **CODE4.errorCode** is set to **r4noOpen**.

See Also: **d4open**, **i4close**

```
/*ex120.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers */

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
    INDEX4 *index ;

    code4init( &cb ) ;
    cb.autoOpen = 0 ; /* don't automatically open index file */

    data = d4open( &cb, "INFO" ) ;
    index = i4open( data, "INFO2" ) ; /* open a secondary index file */

    cb.lockAttempts = WAIT4EVER ; /* wait until the lock succeeds*/
    d4lockAll ( data )  ;
    if( i4reindex( index ) == 0 )
       printf( "Reindexed successfully\n" ) ;

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

i4reindex

Usage: int i4reindex(INDEX4 *index)

Description: All of the tags in the index file are rebuilt using the current data file information. This compacts the index file and ensures that it is up to date.

After **i4reindex** completes, the contents of the record buffer and the record number are undefined. Explicitly call a positioning function such as **d4top** to position to a desired record.

Returns:

r4success Success.

r4unique A unique key tag has a repeat key and **t4unique** returned **r4unique** for that tag.

r4locked A lock was attempted and failed **CODE4.lockAttempts** for either the data file or index file. The index was not updated.

< 0 Error.

Locking: The corresponding data file and the index file are locked. It is recommended that index files be opened exclusively if a reindex is to occur. This will ensure that other users cannot access the file while it is being reindexed. Applications that access index files that have not been fully reindexed may generate errors.

See Also: **d4reindex**, **i4create**

Example: See **i4open**

i4tag

Usage: TAG4 *i4tag(INDEX4 *index, const char *name)

Description: **i4tag** looks up the tag name and returns a pointer to the corresponding tag structure.

Parameters:

name A null terminated string containing the name of the tag to be looked up in the index file.

Returns:

Not Null A pointer to the tag structure.

Null The tag name was not located.

See Also: **d4tagSelect**, **d4index**

i4tagAdd

Usage: int i4tagAdd(INDEX4 *index, const TAG4INFO *newTag)

Description: **i4tagAdd** adds new tags to an existing index file. This function is only available for **.MDX** and **.CDX** index files.


WARNING

In the multi-user configuration, the data file should be opened exclusively before calling **i4tagAdd**. The data file can be opened exclusively by setting **CODE4.accessMode** to **OPEN4DENY_RW** before opening or creating the data file. Unless the data file is opened exclusively, the other applications, which have opened the index file before the new tag was added, will not recognize the new tag. Consequently, these applications might not update the index file correctly.


Note

It is more efficient to add all the tags necessary for an index file at one time using **d4create** or **i4create**, than to add them later with **i4tagAdd**.


Note

If you want to add a new tag in Clipper, just edit the .CGP file and add the name of the new tag to the list of tags. See the Group Files section of the CodeBase User's Guide for details.

Parameters:

newTag This is a pointer to an array of **TAG4INFO** structures defining the tags to add. The array of structures must be null terminated in the same manner as the structure used in **d4create**.

Returns:

r4success Success.

r4locked The index file could not be locked.

< 0 Error.

Locking: **i4tagAdd** locks the index file and unlocks it upon completion.

See Also: **d4unlock**

i4tagInfo

Usage: TAG4INFO *i4tagInfo(INDEX4 *index)

Description: This function creates a **TAG4INFO** array that corresponds to the index file. **i4tagInfo** then returns a pointer to the **TAG4INFO** array, which can be used as a parameter to **d4create** or **i4create**.


WARNING

The return value becomes obsolete once *index* is closed. This is because the tag names of the index file *index* are referenced by the returned **TAG4INFO** array.


WARNING

The **TAG4INFO** return value needs to be freed with the function **u4free**.

Returns:

Not Null A pointer to the **TAG4INFO** array.

Null A null return indicates that not enough memory could be allocated.

See Also: **d4create, i4create**

Linked List Functions

LIST4.selected	l4last
l4add	l4next
l4addAfter	l4numNodes
l4addBefore	l4pop
l4first	l4prev
l4init	l4remove

The functions in this module manipulate linked lists. A linked list is composed of series of individual memory objects called nodes. CodeBase linked lists are double linked lists, which means that each node in the list has a reference to the previous and next node in the list. CodeBase also maintains the pointers to the first and last nodes in the list. In addition, the linked list functions maintain a count of the number of nodes in the linked list, and provide for a user specified "selected" node.

Creating a list of nodes

To implement a linked list, two things are necessary. The first is a **LIST4** structure and the second is a user defined structure for the nodes.

The LIST4 Structure

The **LIST4** structure contains the control information about the linked list. It contains pointers to the first and last nodes, a counter containing the number of nodes in the linked list and a pointer to the selected node.

Each linked list in your application 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 **l4init**.

The Node Structure

The node of a linked list is a data structure that you define called the node structure. Nodes can be any 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 the pointers to the next and previous nodes. These pointers are internally maintained by the list functions so they may safely be ignored.

Note

Pointers to nodes are passed to CodeBase linked list functions as **void** pointers. These **void** pointers must be cast to your structure before members may be accessed.

For more information on using the linked list functions and for example code, see the CodeBase User's Guide.

LIST4 Structure Variables
LIST4.selected

Usage: LINK4 *LIST4.selected

Description: This is a pointer that points to the selected node in the list. Generally, the application program initially sets the selected node. The linked list functions then ensure that the selected member variable always references a valid node in the linked list.

When a selected node is removed from the list, the previous node becomes the selected node. When there are no nodes in the linked list, the selected member variable becomes null.

Linked List Function Reference
l4add

Usage: void l4add(LIST4 *list, void *item)

Description: The node pointed to by *item* is added to the end of the linked list. Consequently, the node pointed to by *item* becomes the last node in the linked list.

The selected node does not change, but the number of nodes in the linked list is incremented.

Parameters:

list A pointer to the list's **LIST4** structure.

item A pointer to the node to be added to the linked list.

See Also: **l4numNodes**, **l4addAfter**, **l4addBefore**

l4addAfter

Usage: void l4addAfter(LIST4 *list, void *anchor, void *item)

Description: The node pointed to by *item* is added to the linked list just after the node pointed to by *anchor*. It is the user's responsibility to ensure that anchor is currently in the linked list. However, if the **E4LINK** switch is on, internal diagnostics verify that this is the case.

If *anchor* is null, it is assumed that the linked list is empty. In this case, *item* becomes both the first and the last node in the linked list. Again, if the **E4LINK** switch is on, internal diagnostics verify this.

If *anchor* is the last node in the linked list, *item* becomes the new last node. The selected node does not change, and the number of nodes is incremented.

Parameters:

list A pointer to the list's **LIST4** structure.

anchor A pointer to a node currently in the linked list.

item A pointer to the node to be added to the linked list.

See Also: **l4add**, **l4addBefore**

l4addBefore

Usage: void l4addBefore(LIST4 *list, void *anchor, void *item)

Description: **l4addBefore** is just like **l4addAfter**, except that *item* is placed in the list before *anchor*.

It is the user's responsibility to ensure that anchor is currently in the linked list. However, if the **E4LINK** switch is on, internal diagnostics verify that this is the case.

If *anchor* is null, it is assumed that the linked list is empty. In this case, *item* becomes both the first and the last node in the linked list. Again, if the **E4LINK** switch is on, internal diagnostics verify this.

If *anchor* is the first node in the linked list, *item* becomes the new first node. The selected node does not change, and the number of nodes is incremented.

Parameters:

list A pointer to the list's **LIST4** structure.

anchor A pointer to a node currently in the linked list.

item A pointer to the node to be added to the linked list.

See Also: **l4add**, **l4addAfter**

l4first

Usage: void *l4first(const LIST4 *list)

Description: A pointer to the first node in the linked list is returned. If there are no nodes in the linked list, a null pointer is returned.

Parameters:

list A pointer to the list's **LIST4** structure.

Returns:

Not Null This is a pointer to the first node in the linked list.

Null The list is empty.

See Also: **l4last**

l4init

Usage: void l4init(LIST4 *list)

Description: This function initializes the **LIST4** structure, which means that all the pointers are set to null including the selected node and the number of nodes in the list is set to zero.

Parameters:

list A pointer to the list's **LIST4** structure.

l4last

Usage: void *l4last(const LIST4 *list)

Description: A pointer to the last node in the linked list is returned. If there are no nodes in the linked list, a null pointer is returned.

Parameters:

list A pointer to the list's **LIST4** structure.

Returns:

Not Null This is a pointer to the last node in the linked list.

Null The list is empty.

See Also: **l4first**

l4numNodes

Usage: int l4numNodes(const LIST4 *list)

Description: This function returns the number of nodes in the linked list.

Parameters:

list A pointer to the list's **LIST4** structure.

Returns:

0 The list is empty.

>=0 This is the number of nodes in the list.

See Also: **l4add**, **l4remove**, **l4pop**

l4next

Usage: void *l4next(const LIST4 *list , const void *item)

Description: This function is used to iterate sequentially through each node in the linked list. It is the programmer's responsibility to ensure *item* points to a node in the linked list.

Parameters:

list This is a pointer to the list's **LIST4** structure.

item This is a pointer to a node in the linked list. If *item* is not null, the pointer to the next node in the list is returned. If *item* is null, then a pointer to the first node in the linked list is returned.

Returns:

Not Null This is a pointer to the next node in the linked list.

Null The node structure pointed to by *item* was the last node in the linked list.

See Also: **l4prev**, **l4first**, **l4last**

l4pop

Usage: void *l4pop(LIST4 *list)

Description: The last node in the linked list is removed and a pointer to its node structure is returned. However, if the linked list is empty, a null pointer is returned.

If the selected node (i.e. the node pointed to by **LIST4.selected**) is removed by a call to **l4pop**, the previous node becomes the selected node. If there is no previous node, the selected node becomes null.

l4pop only removes the node's pointer from the linked list. Any memory allocated for the node (such as allocated by **malloc**) is not freed.

If there are any nodes to pop, the counter for the number of nodes is decremented.

Returns:

Not Null This is a pointer to the node that used to be the last node in the linked list.

Null There are no nodes in the linked list.

See Also: **l4remove**

l4prev

Usage: void *l4prev(const LIST4 *list , const void *item)

Description: This function is used to iterate backwards through each node in the linked list. Except for the direction of the iteration, **l4prev** is the same as **l4next**. It is the programmer's responsibility to ensure *item* is a node of the linked list.

Parameters:

list This is a pointer to the list's **LIST4** structure.

item This is a pointer to a node in the linked list. If *item* is not null, the pointer to the previous node structure is returned. If *item* is null, then a pointer to the last node in the linked list is returned.

Returns:

Not Null This is a pointer to the previous node in the linked list.

Null The node structure pointed to by *item* was the first node in the linked list.

See Also: **l4next**, **l4first**, **l4last**

l4remove

Usage: void l4remove(LIST4 *list, void *item)

Description: The node pointed to by *item* is removed from the linked list. It is the user's responsibility to ensure that *item* is currently a member of the linked list. When the **E4LINK** switch is set, internal diagnostics verify this.

If *item* was the last node, then the second last node becomes the last node. Similarly, if *item* was the first node, then the second node becomes the first. Finally, if *item* is the only node in the linked list, the linked list becomes empty.

If *item* is the selected node (i.e. the node pointed to by **LIST4.selected**), the previous node becomes the selected node. If *item* was the only node, or if there is no previous node, the selected node becomes null.

Parameters:

item This is a pointer to a node currently in the linked list.

See Also: **l4pop**

Memory Functions

mem4alloc
mem4create
mem4free
mem4release

These memory functions specialize in the repeated allocation and deallocation of fixed length memory. As a result, this module is more efficient at memory management than some operating systems. The general purpose operating system can make few assumptions when allocating memory. Consequently, operating system memory allocation tends to have a high overhead in terms of wasted memory and allocation/deallocation time. On the other hand, the memory functions allocate chunks of memory from the operating system, and then to sub-allocate and free the memory with little overhead.

```
/*ex121.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ;

static MEM4 *memory ;

typedef struct myStructSt
{
    char buffer[9] ;
} MY_STRUCT ;

void main( void )
{
    CODE4 cb ;
    MY_STRUCT *ms1, *ms2, *ms3 ;

    code4init( &cb ) ;
    memory = mem4create( &cb, 2, sizeof( MY_STRUCT ), 2, 0 ) ;

    /* ms1 and ms2 use the first block allocated with mem4create*/
    ms1 = mem4alloc( memory ) ;
    ms2 = mem4alloc( memory );
    ms3 = mem4alloc( memory );
    /* construction of ms3 causes two more units to be allocated*/

    strcpy( ms1->buffer, "I " ) ;
    strcpy( ms2->buffer, "WAS " ) ;
    strcpy( ms3->b uffer, "HERE" ) ;
    printf( "%s%s%s\n", ms1->buffer, ms2->buffer, ms3->buffer ) ;

    mem4free( memory, ms1 ) ;
    mem4free( memory, ms2 ) ;
    mem4free( memory, ms3 ) ;

    /* memory still contains allocated memory enough for four MY_STRUCT
     sized structures */

    mem4release( memory ) ; /* free memory allocated with mem4create*/

    code4initUndo( &cb ) ;
}
```

Memory Function Reference
mem4alloc

Usage: void *mem4alloc(MEM4 *memoryType)

Description: A block of memory, the size of the **mem4create** *unitSize* parameter, is partitioned from the memory pool, and a pointer to that memory is returned. If no more units are available, **mem4alloc** expands the pool of memory. The allocated memory is initialized to null.

Parameters:

memoryType A pointer to a **MEM4** structure that was returned by an earlier call to **mem4create.**

Returns: A null pointer return means that there were no memory blocks left and that the operating system could not allocate any additional memory for new blocks.

mem4create

Usage: MEM4 *mem4create(CODE4 *code, int unitsStart, int unitSize, int unitsExpand, int makeTemp)

Description: A pointer to a **MEM4** structure is returned. This pointer can be used with **mem4alloc** to allocate memory from the pool.

Parameters:

code Under DOS, if the application has multiple **CODE4** structures, it is preferable to send a null pointer for this parameter so that the memory may be shared among the **CODE4s**. If there is only one **CODE4** structure in the application, then pass the **CODE4** pointer, which allows CodeBase to free up memory when it's running low under DOS.

unitsStart This is the number of memory blocks that should be initially allocated, from operating system memory, for the memory type.

unitSize This is the size of each memory block to be allocated in terms of bytes. This value must be greater than or equal to 8 bytes.

unitsExpand This is the number of additional memory blocks that should be allocated, from operating system memory, when all of the memory blocks initially allocated are used.

makeTemp If *makeTemp* (make temporary) is false (zero), a pointer to a previously allocated **MEM4** structure may be returned. This occurs when the *unitSize* parameter is the same as with a previous call. If a **MEM4** structure is used twice, the values of *unitsStart* (if applicable) and *unitsExpand* become equal to either the previous settings or the new parameter values, whichever are the largest.

Note that this capability allows index file blocks of the same size to be allocated and deallocated from the same pool of memory.

If *makeTemp* is true (non-zero), a new **MEM4** entry is always used and the entry is never used twice.

Returns:

r4success Success.

< 0 Error.

See Also: **mem4release**

mem4free

Usage: void mem4free(MEM4 *memoryType , void *ptr)

Description: A block of memory previously allocated by **mem4alloc** is freed. Once freed, it can be allocated again with a subsequent call to **mem4alloc**.

Parameters:

ptr This is a pointer to the memory previously allocated by **mem4alloc**. If *ptr* is null, nothing happens. If the **E4MISC** switch was used when compiling CodeBase, CodeBase checks to ensure that parameter *ptr* was initially returned by **mem4alloc**.

See Also: **mem4alloc**, **mem4release**

mem4release

Usage: void mem4release(MEM4 *memoryType)

Description: CodeBase keeps a count of the number of times an internal **MEM4** entry has been used. **mem4release** reduces this count by one, each time it's called, and when the count becomes zero, the internal **MEM4** entry is freed. When the **MEM4** entry is freed, the memory allocated from the operating system for the **MEM4** is returned to the operating system.

When a **MEM4** is freed, all of the memory allocated using that memory type is also freed. Consequently, it is the programmer's responsibility to ensure that the freed memory is not subsequently used.

See Also: **mem4create**

Relate/Query Module

relate4bottom	relate4master
relate4changed	relate4masterExpr
relate4createSlave	relate4matchLen
relate4data	relate4next
relate4dataTag	relate4optimizeable
relate4doAll	relate4querySet
relate4doOne	relate4skip
relate4eof	relate4skipEnable
relate4errorAction	relate4sortSet
relate4free	relate4top
relate4init	relate4type
relate4lockAdd	

The Relation module is used to define and access a hierarchical master - slave relationship between two or more data files. That is, when a slave data file contains supplementary information for another master data file, they are "related".

The exact interaction between the two data files is called a relation. In addition, a relation can be established between a new data file and a slave data file of another relation. The slave in one relation is then treated as a master data file in the new relation. This process builds a relation "tree" where one data file can be a master data file to many different databases.

Once the data file relations have all been specified, you can conceptualize the result as being a single "composite" data file consisting of all the fields of all the related data files. Since the relations are automatically maintained, you can skip backwards and forwards in the "composite" data file and be assured that the related data files are positioned to the appropriate records.

The largest single benefit in using the relation module is the advanced features of Query Optimization. This gives you access to high performance queries with little or no additional programming. Even though Query Optimization contains an extensive amount of complex code, it is almost transparent to the programmer.

Note

The CodeBase Query Optimization is used in the Relation module when queries are specified and it is automatically enabled when the relation is used. See the User's Guide for more information on Query Optimization.

Glossary

Composite Record	A composite record consists of all of the records in the data files of a relation set.
Composite Data File	A composite data file consists of all of the composite records that satisfy the query condition. A composite data file does not really exist since the information is scattered throughout a number of data files. The relation module makes it seem as if

	the composite data file exists. There are three types of relations: *exact match*, *scan* and *approximate match* relations. It is possible for a composite data file in a scan relation to have more records than the top master. In a scan relation, there can be multiple slave data file records corresponding to one master data file record resulting in a composite record for each of the matching slaves. In exact match and approximate match relation, the composite data file has the same number of records as the top master. Refer to **relate4type** for more information.
Master	A master is the controlling data file in a relation. The slave data file record is looked up based on the master data file record. See Also - Top Master
Relation	A relation is a specification of how a slave data file record can be located from a master data file record. A relation corresponds to a **RELATE4** structure, which is initialized through a call to **relate4createSlave**. Note that **relate4init** initializes a **RELATE4** structure to just specify the top master data file and it does not indicate how to locate particular records. This top master data file does not have a master and its current record is generally determined by the sort order. Consequently, this initial **RELATE4** structure specifies a kind of pseudo-relation.
Relation Data File	This is the data file corresponding to a **RELATE4** structure. This is the new data file (or slave) added in the relation set. The relation data file may be both a slave and a master to another data file.
Relation Set	A relation set consists of a pseudo-relation created by a **relate4init** and all other connected relations created by **relate4createSlave**. The data files specified by a relation set consists of the top master, its slaves, the slaves of its slaves, and so on.
Slave	The slave data file is used to look up supplementary information, based on the record contents of its master data file.
Slave List	A list of slaves of a relation data file.
Slave Family	The slave family of a relation data file consists of its slaves, the slaves of its slaves and so on.
Top Master	A master data file is a master only in the context of a specific relation. It can be a slave in a different relation. However, there is exactly one data file in a relation set that has no master. This data file is called the top master. It is specified when

	relate4init is initially called.

Using the Relate Module

To use the relate module, follow these steps:

- First, initialize the relate module by calling **relate4init**.
- Specify any relations using **relate4createSlave**.
- Change the relation defaults by calling **relate4errorAction** and **relate4type** as needed. These calls can be made anytime after the relevant relation has been created.
- Set a query by calling **relate4querySet** and set the sort order by calling **relate4sortSet,** if desired.
- If there is a possibility of skipping backwards, call **relate4skipEnable**.
- If applicable, call **relate4lockAdd** followed by **code4lock**.
- Ensure Query Optimization can fully be utilized by having the appropriate index files open for the data files. (See "Query Optimization" in the User's Guide.)
- Initiate the relation/query by calling **relate4top** or **relate4bottom**.
- Skip through the resulting composite records using **relate4skip**. Start and skip through the relation/query additional times as necessary. Call **relate4querySet** or **relate4sortSet** as necessary to change the query or sort order.
- Call **code4unlock** if applicable. Free the relation set by calling **relate4free**.

Performance Considerations

When **relate4querySet** is called to specify a subset of the composite data file, the relate module contains two major optimizations, which can improve performance tremendously.

The first optimization is the use of Query Optimization in conjunction with data file tags. Query Optimization is possible when the query expression contains the following:

```
Key Expression   Logical Operator   Constant
```

For example, if a tag contains the key expression "LAST_NAME" and the dBASE query expression is "LAST_NAME='SMITH'", then the relate module uses Query Optimization to drastically improve performance. Performance improvements could be hundreds or even thousands of times faster than traditional algorithms.

Query Optimization is possible even when using more complicated query expressions involving .AND. and .OR. operators.

For example, the query expression could be "LAST_NAME='SMITH' .AND. AGE > 20". If there is a tag on either or both LAST_NAME and AGE, then the expression is optimizable. The optimizations are most effective if there is a tag on both.

The second major optimization involves minimizing data file relation evaluation. If it is possible to reject a potential composite record due to the query condition without reading the entire composite record, then the relate module does so. This can significantly improve performance.

For example, suppose the query expression contains the clause "COUNTRY = 'US'", where COUNTRY is a field in the master data file. In this case, the relate module can determine whether to reject the potential composite record before reading any slave data file records.

Multi- user Considerations

Relate module locking must be handled carefully. Observe the following and no difficulties will be met.

- If current data is required, call **relate4lockAdd** and then **code4lock** before calling **relate4top** or **relate4bottom**. This prevents file modification by other users, which could lead to inconsistent results.

Note

The relation set is NOT automatically locked by **relate4top** and **relate4bottom** when **CODE4.readLock** is true (non-zero). Explicitly lock the relation set by calling **relate4lockAdd** and **code4lock.**

- If completely current results are not essential, then do not explicitly lock the files. This allows other users to change the data; the changes may or may not be reflected in the returned composite records. Calling **d4refresh** prior to calling **relate4top** is permissible. In fact, **d4refresh** may be called at any time; however, once **relate4top** is called, it is not guaranteed that subsequently changed data will be reflected in the returned composite records. Call **relate4changed** and **relate4top** to force the relate module to regenerate the composite data file in order to return more current data.
- Once the composite records have been read, it is a good idea to call **code4unlock** to ensure that all locked files are unlocked.

Memory Optimization

For best performance results, memory read optimization should be enabled on all files involved in the relation. Read optimization can be used whether the relation is locked or not, so in either case the performance will be enhanced.

If memory is unavailable to perform needed sorting, memory optimization may be automatically disabled. Consequently, if memory optimization is desired after the composite data file has been read, it is best to explicitly call **code4optStart** to do so.

Sort Order

There are three possible orderings in which the composite records can be presented:

- In a specified sort order as specified using function **relate4sortSet**.
- In the order specified by the selected tag of the top master data file.
- Using record number ordering if the top master data file has no selected tag.

Note

When a scan relationship is defined, there may be several records in the related data file for each master record. If a sort order is not specified, the sub-ordering of the related data file records is undefined.

The following discussion assumes that **relate4sortSet** specifies the same sort order as the selected tag from the top master. If a query results in a relatively small record set when compared to the size of the database, the best way to specify a sort order is to use **relate4sortSet**. If the query set is almost the same size as the database, then do NOT use **relate4sortSet** to specify the sort order. In this case, the best way to specify a sort order is to use the selected tag from the top master data file or record number ordering.

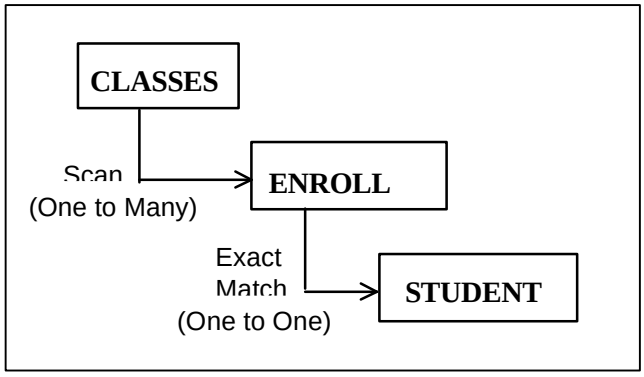
If there is no tag that specifies a desired sort order, use **relate4sortSet** to sort the query set. Using **relate4sortSet** will be faster than creating a new tag to specify the sort order, regardless of the size of the query set.

If **relate4sortSet** is not called, then the selected tag ordering of the top master data file is used. If the top master data file has no selected tag, record number ordering is used.

```
/*ex122.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers */

void main( void )
{   int rc ;
    CODE4 cb ;
    DATA4 *enroll ;
    DATA4 *master ;
    DATA4 *student ;

    TAG4 *enrollTag ;
    TAG4 *studentTag ;

    RELATE4 *MasterRelation ;
    RELATE4 *relation1 ;
    RELATE4 *relation2 ;

    FIELD4 *classCode ;
    FIELD4 *classTitle ;
    FIELD4 *enrollStudentId ;
    FIELD4 *studentName ;

     code4init( &cb ) ;
     enroll = d4open( &cb, "ENROLL" ) ;
     master = d4open( &cb, "CLASSES" ) ;
     student = d4open( &cb, "STUDENT" ) ;

     enrollTag = d4tag( enroll, "C_CODE_TAG" ) ;
     studentTag = d4 tag( student, "ID_TAG" ) ;

     MasterRelation = relate4init( master ) ;
     relation1 = relate4createSlave( MasterRelation, enroll, "CODE", enrollTag ) ;
     relation2 = relate4createSlave( relation1, student, "STU_ID_TAG", studentTag);

     relate4type( relation1, relate4scan ) ;
     relate4sortSet( MasterRelation,  "STUDENT->L_NAME,8,0+ENROLL->CODE" ) ;

     classCode = d4field( master, "CODE" ) ;
     classTitle = d4field( master, "TITLE" ) ;
     enrollStudentId = d4field( enroll, "STU_ID_TAG") ;
     studentName = d4field( studen  t, "L_NAME" ) ;

    error4exitTest( &cb ) ;

    for(rc = relate4top( MasterRelation ); rc != r4eof;
                                          rc = relate4skip( MasterRelation, 1L ) )
    {
```

```
         printf( "%s ", f4str( studentName )) ; /* only one f4str per statement*/
         printf( "%s ", f4str( enrollStudentId ) ) ;
         printf( "%s ", f4str( classCode ) ) ;
         printf( "%s\n", f4str(classTitle ) ) ;
    }

    printf("Number of records in %s is %d\n", d4alias(master),d4recCount(master));

    relate4free( MasterRelation, 1 ) ;
    code4initUndo( &cb ) ;
}
```

The above code creates a two tiered Master - Slave hierarchy. This relation set is illustrated below. Since each class has many students enrolled in it, it is necessary that a scan relationship be established between the CLASSES and ENROLL data files. This is accomplished by calling **relate4type** with a **relate4scan** value. Each entry in the ENROLL data file references a single, exact student, so an Exact match relation (the default) must be established between the ENROLL and STUDENT data files.



Relation Function Reference
relate4bottom

Usage: int relate4bottom(RELATE4 *relate)

Description: This function moves to the bottom of the relation. Essentially, the top master data file is positioned to its bottom according to the sort order of the relation set. Then the slave data files (and their slaves) are positioned accordingly.

If there is a scan relation in the relation set, then the last scan record of the last scan is used to determine the bottom of the relation. **relate4bottom** automatically enables backwards skipping through the relation. Consequently, it is not necessary to call **relate4skipEnable** before **relate4bottom** is called.

Parameters:

relate The parameter *relate* specifies a relation set. Any **RELATE4** structure pointer in the relation set can be used. Regardless, **relate4bottom** acts on the whole relation.

Returns:

r4success Success

r4eof There were no records in the composite data file.

r4terminate A lookup into a slave data file failed and the error action was set to **r4terminate** and **CODE4.errRelate** is set to false (zero).

< 0 Error .

Locking: **relate4bottom** does not do any automatic locking. Explicitly call **relate4lockAdd** and **code4lock** to lock the relation set.

relate4changed

Usage: void relate4changed(RELATE4 *relate)

Description: When a query expression is used, the relate module calculates the resulting set of data at the time that **relate4top** or **relate4bottom** is called. Then when **relate4skip** is called, **relate4skip** returns the information that may just be waiting in an internal CodeBase memory buffer.

If **relate4querySet** or **relate4sortSet** is called -- or if one of the relations in the relation set is modified -- and **relate4changed** is not called to notify the relation set that a change has occurred, **relate4top** and/or **relate4bottom** may just return the same set of information regardless of whether the underlying data might have changed.

Consequently, **relate4changed** should be called to explicitly force the relate module to completely regenerate the result the next time **relate4top** or **relate4bottom** is called.

It is legitimate to call this function more than once. However, once called, do not call **relate4skip** until the relation is positioned through a call to **relate4top** or **relate4bottom**. Otherwise, an error message is generated.

See Also: **relate4querySet**, **relate4sortSet**, **relate4top**, **relate4bottom**

relate4createSlave

Usage: RELATE *relate4createSlave(RELATE4 *master, DATA4 *slave,
char *masterExpr, TAG4 *slaveTag)

Description: This function specifies a relation between a master data file and a slave data file.

When the relation is performed, the *masterExpr* is evaluated, based on the master data file, to obtain either a record number into the slave data file or a key expression that can be used in conjunction with the *slaveTag* to seek to a record in the slave data file.

WARNING

When a new data file is added to the relation set, **relate4top** or **relate4bottom** must be called to reset the entire relation set. Using an out of date relation set can cause unpredictable results.

Parameters:

master This is a pointer to the **RELATE4** structure of the relation data file that is to become the master data file of the new relation. This **RELATE4** pointer could have been returned by **relate4init** or by a previous call to **relate4createSlave**.

slave This **DATA4** pointer specifies a data file that is to become the slave data file.

masterExpr This is a null terminated character array, which specifies a dBASE expression. This expression is evaluated with **expr4parse** using the master data file as the default data file. There is no need to specify the master data file in the expression.
(ie. "MASTER->NAME" may be entered as "NAME") The evaluated expression is then used to locate the corresponding record in the slave data file, when the relation is used.

This expression should evaluate to an index key corresponding to *slaveTag* or a record number if no tag is used on the slave data file.

slaveTag This **TAG4** pointer is a reference to a tag for the slave data file, which corresponds to the evaluated *masterExpr* expression. Any seeking is performed using the *masterExpr* expression on this tag to locate the appropriate record in the slave data file.

If the **TAG4** pointer is null, then the *masterExpr* parameter must evaluate to a record number of the slave data file. **d4go** is then used to locate the appropriate record in the slave data file.

Returns:

Not Null This is a pointer to the **RELATE4** structure of the created relation.

Null Null is returned to indicate that the relation could not be created. Generally this results from an out of memory error condition. It could also mean that the parameter *master* was null. Check the **CODE4.errorCode** for details.

See Also: **relate4querySet**, **relate4sortSet**, **relate4top**, **relate4bottom**, **relate4init**

relate4data

Usage: DATA4 *relate4data(RELATE4 *relate)

Description: **relate4data** returns a pointer to the **DATA4** structure that specifies the relation's data file. It is the data file specified when the *relate* structure was initialized with a call to either **relate4init** or **relate4createSlave**.

See Also: **relate4init**, **relate4createSlave**

relate4dataTag

Usage: TAG4 *relate4dataTag(RELATE4 *relate)

Description: **relate4dataTag** returns the slave tag used in the relation to locate the appropriate records in the slave data file. This is the **TAG4** structure specified as the *slaveTag* parameter to **relate4createSlave.**

This function returns null when the relate structure has no corresponding master or if the lookup expression evaluates to a record number.

See Also: **relate4createSlave**

relate4doAll

Usage: int relate4doAll(RELATE4 *relate)

Description: This function looks up the slave family of the specified parameter *relate*. It assumes that the relation data file specified by this parameter is positioned appropriately.

relate4doAll provides a way to use the relate module to perform automatic lookups. Consequently, you can go to records directly using lower level data file functions (such as **d4go**) and then have related records looked up.

The relation set's query and sort expressions are ignored by **relate4doAll**. Consequently, this function provides somewhat independent functionality. This means using **relate4doAll** in conjunction with relate functions such as **relate4top** and **relate4skip** is not particularly useful. For this reason, it is not necessary to call **relate4top** or **relate4bottom** before calling **relate4doAll**.

Note

To use this function on the entire relation set, position the top master file using a call to a function such as **d4go**. Then call **relate4doAll** using the **RELATE4** pointer returned by **relate4init**.

Note

relate4doAll ignores any query expression set by **relate4querySet**.

Returns:

r4success Success

r4terminate A lookup into a slave data file failed. This was returned because the error action was set to **relate4terminate** and **CODE4.errRelate** is set to false (zero).

< 0 Error

```
/*ex123.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
```

```
    DATA4  *employee ;
    DATA4  *office ;
    DATA4  *building ;

    TAG4 *officeNo ;
    TAG4 *buildNo ;

    RELATE4 *master ;

    RELATE4 *toOffice ;
    RELATE4 *toBuilding ;

     code4init( &cb ) ;
     employee = d4open( &cb, "EMPLOYEE" );
     office = d4open( &cb, "OFFICE" ) ;
     building = d4open( &cb, "BUILDING" ) ;

     /*set up the tags */
     officeNo = d4tag( office, "OFFICE_NO" ) ;
     buildNo = d4tag( building, "BUILD_NO" ) ;

     /* Create the relations */
     master = relate4init( employee ) ;
     toOffice = relate4createSlave(master,office, "EMPLOYEE->OFFICE_NO",officeNo );
     toBuilding = relate4createSlave( toOffice, building, "OFFICE->BUILD_NO",
                                                                        buildNo ) ;
    /* Go to employee, at record 2*/
    d4go( employee, 2L ) ;

    /* Lock the data files and their index files.*/
    relate4lockAdd( master ) ;
     code4lock( &cb ) ;

    /* This call causes the corresponding records in data files "OFFICE" and
     "BUILDING" to be looked up.*/
    relate4doAll( master ) ;

    /* Go to office, at record 3*/
    d4go( office, 3L ) ;

    /* This call causes the building record to be positioned according to its
     master, the office data file*/
    relate4doOne( toBuilding ) ;

    /*  ..  and so on*/

     relate4free( master, 1 ) ;
     code4initUndo( &cb  ) ;
}
```

relate4doOne

Usage: int relate4doOne(RELATE4 *relate)

Description: **relate4doOne** looks up the relation data file using the specified relation. That is, the relation's master expression is evaluated and a seek is performed into the slave data file using the relation's slave tag. The slaves (if any) of the relation's slave data file are not repositioned.

Note

The function **relate4doOne** looks up the relation data file and **relate4doAll** looks up the slaves of the relation data file (and the slaves of those slaves).
For example, if a relation set has exactly one slave, the slave could be looked up using either **relate4doAll** or **relate4doOne**.
However, a different **RELATE4** pointer must be used depending on which function is called. If **relate4doOne** is called, the **RELATE4** pointer parameter must be the one

returned from **relate4createSlave**. If **relate4doAll** is called, the **RELATE4** pointer parameter must be the one returned from **relate4init**.

Note

relate4doOne ignores any query expression set by **relate4querySet**.

Returns:

r4success Success.

r4terminate A lookup into the slave data file failed. This was returned because the error action was set to **relate4terminate** and **CODE4.errRelate** is set to a false (zero) value.

< 0 Error.

See Also: **relate4init**, **relate4doAll**

```
/*ex124.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers */
int seekMaster( DATA4 *master, RELATE4 *r, TAG4 *masterTag, char *seekKey )
{
    int rc ;

    d4tagSelect( master, masterTag ) ;
    rc = d4seek( master, seekKey ) ; /* seek for the requested value*/

    if( rc == r4success )
    relate4doOne( r ) ; /* position the slave data file to the appropriate
                           record according to its master*/
    return rc ;
}

void main( void )
{   int rc ;
    CODE4 cb ;
    DATA4 *enroll ;
    DATA4 *master ;

    TAG4 *enrollTag ;
    TAG4 *codeTag ;
    RELATE4 *MasterRelation ;
    RELATE4 *relation1 ;

    FIELD4 *classCode ;
    FIELD4 *classTitle ;
    FIELD4 *enrollStudentId ;

    code4init( &cb ) ;
    enroll = d4open( &cb, "ENROLL" ) ;
    master = d4open( &cb, "CLASSES" ) ;

    enrollTag = d4tag( enroll, "C_CODE_TAG" ) ;
    codeTag = d4tag( master, "CODE_TAG" ) ;

    MasterRelation = relate4init( master ) ;
    relation1 = relate4createSlave( MasterRelation, enroll, "CODE", enrollTag ) ;

    relate4type( relation1, relate4scan ) ;

    classCode = d4field( master, "CODE" ) ;
    classTitle = d4field( master, "TITLE" ) ;
    enrollStudentId = d4field( enroll, "STU_ID_TAG") ;

    error4exitTest( &cb ) ;

    seekMaster( master, r elation1, codeTag, "MATH521" ) ;
    printf( "%s ", f4str( enrollStudentId ) ) ;
    printf( "%s ", f4str( classCode ) ) ;
    printf( "%s\n", f4str(classTitle ) ) ;
```

```
    relate4free( MasterRelation, 1 ) ;
    code4initUndo( &cb ) ;
}
```

relate4eof

Usage: int relate4eof(RELATE4 *relate)

Description: This function returns whether the relation set is in an end of file position. For example, this would occur when **relate4skip** returns **r4eof**.

Returns:

> 0 The relation set is in an end of file position and this will be returned until the relation set is repositioned.

0 The relation set is not in an end of file position.

< 0 The **RELATE4** structure is invalid or contains an error value.

relate4errorAction

Usage: int relate4errorAction(RELATE4 *relate, int newAction)

Description: At times, a slave data file record cannot be located when the relation is performed. For example, the master key expression has no corresponding entry in the slave tag.

When a slave record cannot be located, the relation module performs one of the following actions, depending on the setting of *newAction*.

relate4blank This is the default action. It means that when a slave record cannot be located, it becomes blank. When using scan relations, a blank composite record will be generated for each slave data file that does not contain a match for the master's 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 module function, possibly **relate4skip**, returns an error code. When using a master with more than one slave, an error is generated if any slave data files do not contain a match for the mater's record.

If the **CODE4.errRelate** member variable is set to false (zero), the error message is suppressed, although the executing function still returns **r4terminate**.

Note

Approximate Match relations are unaffected by this setting. In this type of relation, a blank record is generated if no match is found in the slave data file.

Parameters:

relate This parameter specifies the relation on which the new error action applies. *relate* must be initialized by a call to **relate4createSlave**. If *relate* was initialized by **relate4init** then this function has no effect.

newAction This code specifies the new error action to take. The possible values are **relate4blank**, **relate4skipRec**, and **relate4terminate**.

Returns: **relate4errorAction** returns the previous error action code. If the relation is not initialized, '-1' is returned.

relate4free

Usage: int relate4free(RELATE4 *relate, int closeFiles)

Description: This function frees all of the memory associated with the relation set. Only make one call to **relate4free** for each call to **relate4init**. Do not call **relate4free** for each call to **relate4createSlave**.

Parameters:

relate This pointer specifies the relation set to be freed. Any **RELATE4** structure pointer in the relation may be used since **relate4free** frees the entire relation set.

closeFiles If this parameter contains a true (non-zero) value, all data, index and memo files in the relation tree are flushed and closed once **relate4free** is called. If *closeFiles* is false (zero) all files are left open.

Returns:

r4success Success

< 0 If *closeFiles* is true (non-zero), a negative return indicates there was an error closing a file. A negative value is also returned if the parameter *relate* was null.

Locking: See the locking under **d4close**

relate4init

Usage: RELATE4 *relate4init(DATA4 *topMaster)

Description: **relate4init** initializes the relation set and assigns the top master data file.

Slaves to the top master data file are added with the **relate4createSlave** function.

WARNING

It is important to call **relate4free** prior to calling **relate4init** if a **RELATE4** structure is to be reinitialized with a new top master data file. Failure to do so can result in substantial memory loss.

Parameters:

topMaster *topMaster* specifies the data file to be the top master for the entire relation set.

Returns:

Not Null A pointer to the RELATE4 structure that specifies the new relation set is returned.

Null An has occurred, check the **CODE4.errorCode** for details on which error occurred.

See Also: **relate4free**, **relate4createSlave**

```
/*ex125.c*/
#include "d4all.h"

void main( void )
{
    CODE4 cb ;
    DATA4 *info ;
     RELATE4 *TopMaster ;

     code4init( &cb ) ;
     info = d4open( &cb, "INFO" ) ;

     TopMaster = relate4init( info ) ;

   /* ... other code ...*/

   /* This relation tree is no longer needed. Create a new one*/
    relate4free( TopMaster, 0 ) ;

     TopMaster = relate4init( info ) ;

   /* ... other code ... */
                          /* Automatically close all files in the relation*/
    relate4free( Top Master, 1 ) ;

     code4close( &cb ) ;     /* close any remaining files*/
     code4initUndo( &cb ) ;
}
```

relate4lockAdd

Usage: int relate4lockAdd(RELATE4 *relate)

Description: This function adds all of the data files referenced by the relation set along with their corresponding index files to the list of locks placed with the next call to **code4lock**.

Parameters:

relate This pointer specifies the relation set. Any **RELATE4** structure pointer in the relation may be used since **relate4lockAdd** adds all of the data files in the relation set to the list of pending locks.

Returns:

r4success Success. The specified relation set was successfully placed in the **code4lock** list of pending locks.

< 0 Error. The memory required for the record lock information could not be allocated.

See Also: **code4lock**

relate4master

Usage: RELATE4 *relate4master(RELATE4 *relate)

Description: This function returns a pointer to a **RELATE4** structure, which specifies the relation that is the master of *relate.*

This function returns null if the relation specified by *relate* has no master. In this case, the relate structure was created by **relate4init** and it represents the top master.

Example: See **relate4doOne**

relate4masterExpr

Usage: char *relate4masterExpr(RELATE4 *relate)

Description: **relate4masterExpr** returnsa string that specifies the dBASE expression that was specified in the call to **relate4createSlave,** which initialized *relate*.

This function returns null, if *relate* was initialized by **relate4init**.

See Also: **relate4master, relate4createSlave**

relate4matchLen

Usage: int relate4matchLen(RELATE4 *relate, int len)

Description: A relation's tag has a key length and a relation's master expression has a length. Normally, assuming a Character tag and master expression, the number of characters used from the master expression is the smaller of the two lengths. However, **relate4matchLen** can be called to further decrease the number of characters used from the master expression.

WARNING

If this function is called to change the relation's match length, **relate4top** or **relate4bottom** must be called to reset the relation set. Using a relation with an out of date length can cause unpredictable results.

Parameters:

len Parameter *len* is the number of characters from the evaluated master expression to use. If the value specified is illegal (eg. negative), then the default is used.

Returns: The actual match length is returned. Normally, this is the same as parameter *len*. However, if *len* is an illegal value, then the returned value will be the maximum possible value.

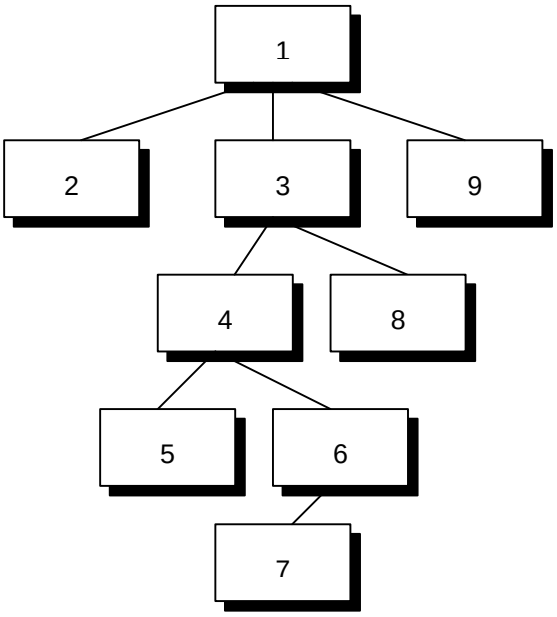
See Also: **relate4masterExpr**

relate4next

Usage: int relate4next(RELATE4 **relatePtrPtr)

Description: This function can be used to iterate through all the relations in the relation set. After **relate4next** is called, it references the next relation in the tree.

relate4next uses the Depth-First algorithm to move through the relation set. The following diagram illustrates the order in which the relation data files are accessed.



Parameters:

*relatePtrPtr This is a pointer which points to the current **RELATE4** pointer in the iteration. The **relatePtrPtr* becomes null, when there are no more relation data files to iterate through. In order to iterate through the entire relation set the first **relatePtrPtr* should point to the top master **RELATE4** structure returned by **relate4init**.

Returns:

r4complete Done. There are no additional relations in the relation set.

r4down The new ***relatePtrPtr* is one further down in the relation set.

r4same The new ***relatePtrPtr* is the next slave of the same master.

-X The new ***relatePtrPtr* is 'X' masters up. For example, a return of **(int)** -1 means that the master had no additional slaves and the master's next slave is the new ***relatePtrPtr*.

```
/*ex126.c*/
#include "d4all.h"

void displayRelationTree( RELATE4 *relate )
{
    DATA4 *data ;
    int i, pos = 0 ;

    for(; relate; )
    {
         printf( "\n" ) ;
         for(i = pos; i > 0; i-- )
          printf( "   " ) ;
         data = relate4data( relate ) ;
         printf( "%s", d4fileName( data )   ) ;

         pos +=  relate4next( &relate ) ;
    }
}
```

```
void main( void )
{
CODE4 cb ;
    DATA4 *master ;
    DATA4 *sl1 ;
    DATA4 *sl2 ;
    DATA4 *sl3 ;
    DATA4 *sl4 ;

    RELATE4 *MasterRelation ;
     RELATE4 *relate1, *relate2, *relate3, *relate4 ;

     code4init( &cb ) ;
     master = d4open( &cb, "M1" ) ;
     sl1 = d4open( &cb, "SL1" ) ;
     sl2 = d4open( &cb, "SL2" ) ;
     sl3 = d4open( &cb, "SL3" ) ;
     sl4 = d4open( &cb, "SL4" ) ;

    /* create the tree*/
     MasterRelation = relate4init( master ) ;
     relate1 = relate 4createSlave( MasterRelation, sl1, "TOSL1",
                                                          d4tag( sl1, "FRM" ) ) ;
     relate2 = relate4createSlave( MasterRelation, sl2, "TOSL2",
                                                          d4tag( sl2, "FRM") ) ;
    relate3 = relate4createSlave( relate2, sl3, "TOSL3",
                                                          d4tag( sl3, "FRMSL2" ) ) ;
    relate4 = relate4createSlave( MasterRelation, sl4, "TOSL4",
                                                          d4tag( sl4, "FRM" ) ) ;
    error4exitTest( &cb ) ;

    relate4top( MasterRelation ) ;
    displayRelationTree( MasterRelation ) ;

    relate4free( MasterRelation, 1 ) ;
     code4initUndo( &cb ) ;
}
```

relate4optimizeable

Usage: int relate4optimizeable(RELATE4 *relate)

Description: This function indicates whether Query Optimization can be used for a particular query expression. **relate4optimizeable** returns true (non-zero) if Query Optimization can be used and false (zero) if not. When false is returned, a programmer can create a new index file with the appropriate tags so that Query Optimization can be fully utilized.

Note that even when this function returns true, Query Optimization may not be used if there is insufficient memory.

relate4querySet

Usage: int relate4querySet(RELATE4 *relate, const char *query)

Description: This function sets a query for the relation set. The dBASE expression *query* is evaluated for each composite record. If the expression is true (non-zero), the record is kept. Otherwise, it is ignored as if it did not exist.

WARNING

It is legitimate to call this function more than once. However, once this function is called, do not call **relate4skip** until the relation is positioned through a call to **relate4top** or **relate4bottom**. Otherwise, an error is generated.

Parameters:

relate This pointer specifies the relation set. Any **RELATE4** structure pointer in the relation may be used since the query expression applies to the entire relation set.

query This is a logical dBASE expression, which can be parsed by function **expr4parse**. Field names in queries must be qualified with a data file name unless the field belongs to the top master data file.

Example:`"PEOPLE->LAST_NAME = 'SMITH'"`

A NULL *query* parameter (the default) cancels the query.

Returns:

r4success Success

< 0 Error or *relate* was null.

relate4skip

Usage: int relate4skip(RELATE4 *relate, long numSkip)

Description: Conceptually, the relation set defines a composite data file with a set of composite records. This function skips forward or backwards in the composite data file.

Parameters:

relate *relate* specifies the relation set. Any **RELATE4** structure pointer in the relation set can be used. Regardless, skipping is done in the composite data file corresponding to the entire relation set.

numSkip The number of records to skip. If *numSkip* is negative, then skipping is done backwards. If you pass **relate4skip** a negative parameter for *numSkip* without first calling **relate4bottom** or **relate4skipEnable**, an error message is generated.

Returns:

r4success Success.

r4terminate A lookup into a slave data file failed. This value was returned because the error action was set to **relate4terminate** and **CODE4.errRelate** is set to false (zero).

r4bof An attempt was made to skip before the first record in the composite data file. The 'beginning of file' condition becomes true (non-zero) for the master data file.

r4eof An attempt was made to skip past the last record in the composite data file. The 'end of file' condition becomes true (non-zero) for the master data file.

< 0 Error.

relate4skipEnable

Usage: int relate4skipEnable(RELATE4 *relate, int doEnable)

Description: In order to allow skipping backwards the relate module needs to perform some extra work and save some extra information.

Calling **relate4skip** with a negative parameter for *numSkip* causes an error condition unless skipping backwards is explicitly enabled for the relation set.

Skipping backwards is enabled either through a call to **relate4skipEnable** or a call to **relate4bottom**.

WARNING

It is legitimate to call this function more than once. However, once this function is called, do not call **relate4skip** until the relation is positioned through a call to **relate4top** or **relate4bottom**. Otherwise, an error is generated.

Parameters:

relate *relate* specifies the relation set. Any **RELATE4** structure pointer in the relation set can be used. Regardless, skipping is enabled or disabled for the entire relation set.

doEnable Skipping backwards is enabled if *doEnable* is true (non-zero). Otherwise, skipping backwards is disabled.

Returns:

r4success Success

< 0 Error or *relate* was null.

relate4sortSet

Usage: int relate4sortSet(RELATE4 *relate, const char *sort)

Description: This function specifies the sorted order in which **relate4skip** returns the various composite records.

WARNING

It is legitimate to call this function more than once. However, once this function is called, do not call **relate4skip** until the relation is positioned through a call to **relate4top** or **relate4bottom**. Otherwise, an error is generated.

Note

Only call **relate4sortSet** when the Query Optimization has reduced the record set to a relatively small size compared to the size of the database. Calling this function to specify the sorted order of a very large record set will be slow compared to an equivalent sort order specified by the selected tag of the master data file. See the **Relation\Query** introduction for more details on alternative methods of specifying the sorted order.

Parameters:

relate *relate* specifies the relation set. Any **RELATE4** structure pointer in the relation set can be used. Regardless, the specified sort order applies to the entire relation set.

sort This is a dBASE expression that specifies the sort order. This expression can produce a result of type Character, Date or Numeric. However, if it is a Logical expression, an error is generated when **relate4top** or **relate4bottom** is called.

Field names in sort expressions must be qualified with a data file name unless the field belongs to the top master data file.

Example: `"INVENTORY->PART_NAME"`

A NULL parameter, the default, cancels the explicit sorting.

Returns:

r4success Success

< 0 Error or *relate* was null.

See Also: The Sort Order subsection of the relate module introduction describes the three possible ways to order the composite records.

relate4top

Usage: int relate4top(RELATE4 *relate)

Description: This function moves to the top of the composite data file. Essentially, the top level master data file is positioned to the top of the composite data file and then the slave data files (and their slaves) are positioned accordingly.

Parameters:

relate *relate* specifies the relation set. Any **RELATE4** structure pointer in the relation set can be used. Regardless, **relate4top** acts on the entire relation set.

Returns:

r4success Success

r4terminate A lookup into a slave data file failed. This was returned because the error action was set to **relate4terminate** and **CODE4.errRelate** is set to false (zero).

r4eof There were no records in the composite data file.

< 0 Error

Locking: **relate4top** does not do any automatic locking. Explicitly call **relate4lockAdd** and **code4lock** to lock the relation set.

relate4type

Usage: int relate4type(RELATE4 *relate, int newType)

Description: There are three ways data files may be related. Either an exact one-to-one relationship, an approximate one-to-one relationship, or a one-to-many relationship. This function specifies the type.

None of these types apply if the relation expression evaluates directly to a record number.

WARNING

If this function is called to change the relation type, **relate4top** or **relate4bottom** must be called to reset the relation set. Using a relation where a relation type has changed can cause unpredictable results.

Parameters:

relate *relate* specifies the relation set to which the new relation type applies. This is the relation that was initialized by a call to **relate4createSlave**. If *relate* was initialized by **relate4init** then this function has no effect.

newType This value determines how records in the slave data file are located. The possible values for *newType* are:

relate4exact This means that for a record to be located, the key value evaluated from the relation expression must be identical to the key value in the tag. However, the lengths of character values do not need to match. If the lengths are different, comparing is done using the shorter of the two lengths. An even shorter length may be used due to a call to **relate4matchLen**.

This is the only option in which a lookup error, as described under function **relate4errorAction**, can occur.

This is the default value.

relate4approx This means that if a key value cannot be exactly located, the first one after is used instead. If the key value is greater than any key value in the tag, then a blank record is used.

This option is useful for looking up a range of values using a single high value. See the User's Guide for an example of using this type of relation.

relate4scan A scan relation means that zero or more records can be located for each master record.

A record is located for each key value in the tag that exactly matches the evaluated relation expression.

Consider the case in which a single master data file has several slave data files, all specified as scan relation types. In this case, when one slave is being scanned, the other slave records are set to blank.

Returns: The previous type code is returned. If the relation has not been initialized, or is invalid, **relate4type** returns '-1'.

Sort Functions

sort4assignCmp
sort4free
sort4get
sort4getInit
sort4init
sort4put

The sort module provides functions to sort large or small amounts of data. Unlike standard sorting routines, which only work on one type of data, CodeBase's sort routines are generic and work on any type including those that you have defined.

Sorts are performed in memory using an efficient quick sort. If the amount of information is too large to be sorted in memory, it is divided up into segments and spooled to disk. A merge sort is then used when the information is retrieved.

Using the Sort Module

To use the sort routines, the following six steps should be performed:

1. A **SORT4** structure must be constructed in your application. This structure contains internal information that is vital to the sort operations. A pointer to this structure will be passed to all of the sort module functions.
2. The **SORT4** structure must be initialized by a call to **sort4init**. This creates an empty list called the sort list, which will eventually contain the sorted data items.
3. If necessary, a comparison function is specified by calling **sort4assignCmp**.
4. The sort items are added to the sort list by calls to **sort4put**.
5. **sort4getInit** is called to specify that all of the calls to **sort4put** are completed.
6. The data items are returned in sorted order by consecutive calls to **sort4get**.
7. Any allocated memory is freed and any temporary files are removed by calling **sort4free**.

```
/*ex127.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ; /* for all Borland compilers  */

void main( void )
{
    CODE4 cb ;
    DATA4 *data ;
    FIELD4 *name ;
     int rc ;
     long recNo ;
     char *info, *otherInfo ;
```

```
    SORT4 dbSort ;

    code4init( &cb ) ;
    data = d4open( &cb, "INFO" ) ;
    name = d4field( data, "NAME" ) ;

    sort4init( &dbSort, &cb, f4len( name ), d4recWidth( data ) + 1 ) ;

    for( rc = d4top( data ); rc == r4success; rc = d4skip( data, 1L ) )
       sort4put( &dbSort, d4recNo(data), f4ptr(name), d4record(data) ) ;

    d4close( data ) ; /* database stored in dbSort.*/

    sort4getInit( &dbSort ) ; /* no more items to add.*/

    printf( "Database sorted on NAME: \n" ) ;
    while( sort4get( &dbSort, &recNo, &info, &otherInfo ) == 0 )
    {
       info[f4len(name) - 1] = 0 ;
       printf( "Record # %ld, Info %s, OtherInfo %s\n", recNo, info, otherInfo ) ;
    }

    sort4free( &dbSort ) ;
    code4initUndo( &cb ) ;
}
```

The Comparison Function

Since the sort routine sorts data of any type, the sort module must be given a method to compare two sort items. This is the purpose of the comparison function. The comparison function is a user defined function that accepts two sort items and returns the item that has a greater value.

Default Comparison Function

If **sort4assignCmp** is not used to specify a comparison function, then the C library **memcmp** function is used instead. If structures are going to be sorted then **sort4assignCmp** should be called..

The comparison function should be declared as follows with *cFunct* replaced by the name of your comparison function:

```
int S4CALL cFunct( S4CMP_PARM p1,S4CMP_PARM p2, size_t len)
```

Although the parameters *p1* and *p2* are declared as S4CMP_PARM (defined as either **(void *)** or **(const void *)** depending on your compiler), they are actually pointers to two sort items. The *len* parameter is the size of one sort item. The body of the comparison function should compare *p1* and *p2* and return the following values:

Return	Meaning
< 0	The value pointed to by *p1* is less than *p2*.
0	The values pointed to by *p1* and *p2* are equal.
> 0	The value pointed to by *p1* is greater than *p2*.

Here is an example sort item and its comparison function:

```
/*ex128.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ;

/* the Sort Item */
typedef struct myStructSt
{
    int number ;
    char otherStuff ;
} NUM ;

int compNum( S4CMP_PARM p1, S4CMP_PARM p2, size_t len )
{
    if( ((NUM *) p1)->number > ((NUM *) p2)->number ) return 1 ;
    if( ((NUM *) p1)->number < ((NUM *) p2)->number ) return -1 ;
```

```
    return 0 ;
}

void main( void )
{
    CODE4 cb ;
    SORT4 sort ;
    NUM st1, st2, st3, *st ;
    long recNo ;
    char *notUsed ;

    code4init( &cb ) ;
    sort4init( &sort, &cb, sizeof( NUM ), 0 ) ;
    sort4assignCmp( &sort, compNum ) ;

    st1.number = 123 ;
    st2.number = 432 ;
    st3.number = 321 ;

    sort4put( &sort, 1L, &st1, NULL ) ;
    sort4put( &sort, 2L, &st2, NULL ) ;
    sort4put( &sort, 3L, &st3, NULL ) ;

    sort4getInit( &sort ) ;

    while( sort4get( &sort, &recNo, &st, &notUsed ) == 0 )
    {
        printf( "Sorted Item: %d\n", st->number ) ;
    }

    sort4free( &sort ) ;
    code4initUndo( &cb ) ;
}
```

Sort Function Reference
sort4assignCmp

Usage: void sort4assignCmp(SORT4 *sort, S4CMP_FUNCTION *fp)

Description: Function **sort4assignCmp** specifies a C comparison function that compares two sort items. This comparison function determines the resulting sort order.

Call this function just after calling **sort4init. sort4assignCmp** uses the **CODE4.hInst** member when the CodeBase DLL is used.

By default, if **sort4assignCmp** is not called, the equivalent of C runtime library function **memcmp** is used instead.

Parameters:

sort A pointer to the **SORT4** structure.

fp A pointer to a comparison function. The comparison function should be declared as follows with "cFunct" replaced by the name of your comparison function:

```
int S4CALL cFunct( S4CMP_PARM p1,S4CMP_PARM p2, size_t len)
```

See Also: **CODE4.hInst**

Example: See the Sort introduction.

sort4free

Usage: int sort4free(SORT4 *sort)

Description: Any memory allocated for the sort is freed. In addition, any temporary file, which might have been created, is closed and removed.

It is not necessary to call **sort4free** once **sort4get** returns a return code indicating no more entries are available. However, calling **sort4free** more than once does no harm as long as **sort4init** has been called.

After **sort4free** is called, **sort4init** must be called again to start a new sort.

Returns:

r4success Success.

< 0 Error.

See Also: **sort4get**

Example: See Sort introduction

sort4get

Usage: int sort4get(SORT4 *sort, long *recPtr, void **sortPtrPtr, void **otherPtrPtr)

Description: This function is used to retrieve the next item in the sorted order.

Once an item is retrieved, it is removed from the memory associated with the sort. When all items are removed, all memory is freed.

Parameters:

sort A pointer to a **SORT4** structure.

recPtr The calling application supplies a long integer of memory to which *recPtr* points. The record number corresponding to the sort item is placed into **recPtr*.

sortPtrPtr This is the address of the pointer (ie. a pointer to a pointer) to the next sort item.

The next sorted item is temporarily contained in the memory supplied by the sort module. Once another call is made to **sort4get** the returned pointer value becomes obsolete and should no longer be used.

Note

This sort item was initially supplied by the parameter *sortInfo* from the function **sort4put**. The length of the sort item is specified in the initial call to **sort4Init**.

otherPtrPtr This is the address of a pointer to memory supplied by the calling application. This memory becomes a pointer to some information corresponding to the sort item.

The information is temporarily contained in memory supplied by the sort module. Once another call is made to **sort4get** the returned pointer value becomes obsolete and should no longer be used.

Note

This corresponding information was initially supplied to the sort module through the parameter *otherInfo* from the function **sort4put**. The length of this information is specified in the initial call to **sort4init**.

Returns:

r4success Success.

r4done There are no more items to retrieve.

< 0 Error.

See Also: **sort4getInit**, **sort4put**

sort4getInit

Usage: int sort4getInit(SORT4 *sort)

Description: This function is called to signal that all of the calls to **sort4put** have been completed. **sort4getInit** must be called before calling **sort4get** to retrieve the sorted items.

Internal memory is allocated to begin the sorting.

Returns:

r4success Success.

< 0 Error.

See Also: **sort4get**, **sort4put**, **sort4free**

sort4init

Usage: int sort4init(SORT4 *sort, CODE4 *code, int sortSize, int otherSize)

Description: Initialization is done to prepare for the sort. Specifically, the initial comparison function is set to **memcmp** and some initial memory is allocated.

Parameters:

sort A pointer to a **SORT4** structure.

code A pointer to a **CODE4** structure. **CODE4.memSizeSortPool** and **CODE4.memSizeSortBuffer** specify default memory allocation for the sort. Please refer to the chapter on **CODE4** settings.

sortSize This is the size of each sort item passed to **sort4put** and the size of each sort item retrieved by **sort4get**. Generally this value is obtained by using the **sizeof** operator.

otherSize This is the size of the additional "other" information for each sort item. This additional information is also passed to **sort4put** and retrieved by **sort4get**. Generally this value is obtained by using the **sizeof** operator.

Returns:

r4success Success.

< 0 Error.

See Also: **sort4put**, **sort4get**

```
/*ex129.c*/
#include "d4all.h"
extern unsigned _stklen = 10000 ;

typedef struct myStructSt
{
    char name[7] ;
} MY_STRUCT ;

int myStructCmp( S4CMP_PARM p1, S4CMP_PARM p2, size_t len )
{
   return strcmp( ((MY_STRUCT *)p1)->name, ((MY_STRUCT *)p2)->name ) ;
}

void main( void )
{
    CODE4 cb ;
    SORT4 sort ;
    MY_STRUCT st1, st2, st3, st4, *st ;
    long recNo ;
    char *notUsed ;

    code4init( &cb ) ;
    sort4init( &sort, &cb, sizeof( MY_STRUCT ), 0 ) ;
    sort4assignCmp( &sort, myStructCmp ) ;

    strcpy( st1.name, "hello" ) ;
    strcpy( st2.name, "this" ) ;
    strcpy( st3.name, "apples" ) ;
    strcpy( st4.name, "fa ce" ) ;

    sort4put( &sort, 1L, &st1, NULL ) ;
    sort4put( &sort, 2L, &st2, NULL ) ;
    sort4put( &sort, 3L, &st3, NULL ) ;
    sort4put( &sort, 4L, &st4, NULL ) ;

    sort4getInit( &sort ) ;
    while( sort4get( &sort, &recNo, &st, &notUsed ) == 0 )
    {
       printf( "Sorted Item: %s\n", st->name ) ;
    }
    sort4free( &sort ) ;
    code4initUndo( &cb ) ;
}
```

sort4put

Usage: int sort4put(SORT4 *sort, long rec, const void *sortInfo, const void *otherInfo)

Description: This function is used to specify some information to be sorted.

Note

Function **sort4put** immediately makes a copy of the sort information and the "other" information. Consequently, the application memory area containing the data passed to **sort4put** can immediately be used again. For example, the application could read the data to sort into a temporary buffer, call **sort4put** and then read some more data into the temporary buffer.

Parameters:

sort A pointer to a **SORT4** structure.

rec — A long integer that will correspond to the sort item. A sub-sort is done on this value. This means that if the sort items are identical, the sort item with the largest *rec* value goes last.

sortInfo — This pointer points to a sort item.

otherInfo — This is some information that is attached to the sort item. This information is retrieved with the sort item by function **sort4get**. For example, when a data file is sorted the attached information could be the data file record.

Returns:

r4success — Success.

< 0 — Error.

See Also: **sort4get, sort4init**

Example: **sort4init**

Tag Functions

TAG4.index
t4alias
t4close
t4expr
t4filter
t4open
t4unique
t4uniqueSet

A tag corresponds to a sorted order stored in an index file. The tag functions are used by the index and data functions to manipulate the sort orderings of an index file.

TAG4 Structure Variable
TAG4.index

Usage: INDEX *TAG4.index

Description: This member of the **TAG4** structure is a pointer to the corresponding **INDEX4** structure which can be used with the index file functions.

See Also: Index File Functions

Tag Function Reference
t4alias

Usage: const char *t4alias(TAG4 *tag)

Description: **t4alias** returns the unique name used to identify the tag in the index file. This name is specified when the index file is initially created.

Returns: **t4alias** returns a null terminated character array containing the name of the tag alias.

See Also: **i4create, d4tag**

t4close

Usage: int t4close(TAG4 *tag)

Description: A Clipper tag file is flushed to disk, if necessary, and closed. This function is only available when the **S4CLIPPER** switch is used.

If the record buffer of the corresponding data file has been changed, the record is flushed to disk and the tag file is updated before it is closed.

If the tag must be updated, **t4close** locks the file, performs the flushing and closes the file. **t4close** temporarily sets the **CODE4.lockAttempts** to **WAIT4EVER** to ensure that the tag file is locked and updated before returning. As a result, **t4close** never returns **r4locked**. If **t4close**

encounters a non-unique key in a unique tag while flushing the data file, the tag file is closed but not updated.

An error will be generated if **t4close** is called during a transaction.

Returns:

r4success Success.

< 0 Error.

See Also: **i4close, t4open, S4CLIPPER**

t4expr

Usage: const char *t4expr(TAG4 *tag)

Description: **t4expr** returns a pointer to a character string containing the expression that determines the order in which records are added to the tag.

Do not use the string returned from **t4expr** to alter the sort expression. Doing so can cause unpredictable results, including the corruption of the tag.

Returns: An expression string for the sort expression is returned.

See Also: Expression functions.

t4filter

Usage: const char *t4filter(TAG4 *tag)

Description: **t4filter** returns a pointer to a string, which contains the filter expression that determines which records are added to the tag.

Do not use the string returned from **t4filter** to alter the filter expression. Doing so can cause unpredictable results, including the corruption of the tag.

Returns: A pointer to a string containing the filter expression is returned.

t4open

Usage: TAG4 *t4open(DATA4 *data, INDEX4 *index, const char *name)

Description: A Clipper tag file is opened. Normally, tag files are opened using **i4open**.

This function is only available when the **S4CLIPPER** switch is defined. dBASE IV and FoxPro index files can be automatically opened when the data file is opened, or **i4open** can be used to open an index file of tags.

Parameters:

data Specifies the data file for which the tag file was created.

index This is the **INDEX4** structure corresponding to the tag. If this parameter is not provided, **t4open** creates an internal **INDEX4** structure which corresponds to the tag.

name This is the name of the tag file. The default file extension is .NTX. If another extension is specified, it is used.

Returns: The function returns a pointer to the corresponding **TAG4** structure. A return value of null indicates an error.

See Also: **CODE4.autoOpen, i4open**

t4unique

Usage: int t4unique(const TAG4 *tag)

Description: **t4unique** returns the setting for the way the tag handles attempts to add duplicate records.

Returns:

0 The tag is not a unique tag.

r4unique The tag is a unique tag. When an attempt is made to add a record to the data file that has a duplicate key value, this value is returned, and the record is not added.

r4uniqueContinue The tag is a unique tag. When an attempt is made to add a record to the data file that has a duplicate key value, this value is returned, but the record is added. The tag only contains a reference to the first record added.

e4unique Although **e4unique** has a negative value, it does not indicate an error when it is returned in this case. This return indicates that the setting is equal to **e4unique**, which means that an error is generated if a duplicate key is encountered.

< 0 Error.

See Also: **CODE4.errDefaultUnique**

t4uniqueSet

Usage: int t4uniqueSet(TAG4 *tag, int uniqueCode)

Description: **t4uniqueSet** sets the setting for the way the tag handles attempts to add duplicate records for unique tags only.

Note

t4uniqueSet can only be used to change the setting of a unique tag. Setting a unique tag to a non-unique tag or setting a non-unique tag to a unique tag will generate a CodeBase error.

Parameters: If *uniqueCode* is specified then the way duplicate records are handled while the tag is open is changed. Changing the value of the tag only effects the way subsequent duplicate records are handled, but does not alter any previously stored keys. In addition, the unique setting is initialized to the **CODE4.errDefaultUnique** setting each time the tag is opened. The possible values of *uniqueCode* are as follows:

r4unique	The tag is a unique tag. When an attempt is made to add a record to the data file that has a duplicate key value, this value is returned, and the record is not added.
r4uniqueContinue	The tag is a unique tag. When an attempt is made to add a record to the data file that has a duplicate key value, this value is returned, but the record is added. The tag only contains a reference to the first record added.
e4unique	This value indicates that an error is generated if a duplicate key is encountered.

Returns:

r4success	Success.
< 0	Error.

See Also: **CODE4.errDefaultUnique**

Utility Functions

u4alloc	u4nameChar
u4allocAgain	u4nameExt
u4allocErr	u4namePiece
u4allocFree	u4ncpy
u4free	u4yymmdd

The utility functions perform miscellaneous tasks such as manipulating file names, copying memory, allocating memory and so on.

u4alloc

Usage: void *u4alloc(long len)

Description: This function allocates memory from the operating system.

Parameters: Parameter *len* is the number of bytes to allocate. Even though *len* is defined as a long integer, it does not necessarily mean that more than an unsigned integer worth of memory can be allocated under the operating system.

Returns: A pointer to the allocated memory is returned. The memory is initialized to null. A null pointer is returned to indicate that no memory was available.

u4allocAgain

Usage: int u4allocAgain(CODE4 *codeBase, char **ptrPtr ,unsigned *lenPtr, unsigned newLen)

Description: This function re-allocates a larger piece of memory if the required memory size is larger than the current memory size. If **u4allocAgain** needs to reallocate the memory, it does so by calling **u4allocFree**.

u4allocAgain is convenient because most programs need to save the current length of an allocated memory area. **u4allocAgain** maintains this length through parameter *lenPtr*.

Parameters:

codeBase This points to the **CODE4** structure declared for the application. It is used for displaying an error message if the memory could not be allocated. *codeBase* can be null. In this case, no error is displayed if memory could not be allocated.

ptrPtr **ptrPtr* contains either null or a pointer to memory previously returned by **u4alloc** or **u4allocAgain**. If **ptrPtr* is not zero, a copy of the previously

allocated memory is copied into the new memory and the previously allocated memory is freed.

lenPtr This points to an unsigned integer containing the number of bytes of memory previously allocated using **u4alloc** or **u4allocAgain**. It is used to determine the number of bytes of memory to copy from the old memory to the new memory before freeing the old memory. **lenPtr* is updated with the new length.

newLen This is the number of bytes to allocate.

Returns:

0 Zero is returned to indicate that the new memory was successfully allocated.

e4memory **e4memory** is returned to indicate that the memory was not allocated due to the memory not being available.

u4allocErr

Usage: void *u4allocErr(CODE4 *codeBase, long len)

Description: This function allocates memory from the operating system. It is identical to **u4allocFree** except that if memory cannot be allocated, an error is generated.

Returns: A pointer to the allocated memory is returned. The memory is initialized to null. A null pointer is returned to indicate that no memory was available. In this case, if the *codeBase* parameter is not null an error is generated.

u4allocFree

Usage: void *u4allocFree(CODE4 *codeBase, long len)

Description: This function allocates memory from the operating system. If the operating system cannot allocate the memory, CodeBase tries to free up some lower priority memory and then tries again. CodeBase considers memory allocated for memory optimization as being lower priority. Consequently, if memory optimization is being used and memory cannot be allocated, **code4optSuspend** is called, another attempt is made to allocate the memory from the operating system, and then **code4optStart** is called to start up the memory optimization once again.

Returns: A pointer to the allocated memory is returned. The memory is initialized to null.

u4free

Usage: void u4free(void *ptr)

Description: This function frees memory previously allocated by **u4alloc**, **u4allocErr**, **u4allocFree** or **u4allocAgain**. *ptr* should point to memory allocated with one of these functions. When the **E4MISC** switch is defined, **u4free**

ensures that the memory had previously been allocated and not previously deallocated. In addition, **u4free** checks that there was no overwriting just before or just after the allocated memory.

u4nameChar

Usage: int u4nameChar(unsigned char ch)

Description: This function returns true (non-zero) if *ch* is a valid file name character. Otherwise, false is returned.

u4nameExt

Usage: void u4nameExt(char *name, int lenMax, const char *ext , int doReplace)

Description: The file name extension of a file name is added or replaced. The resulting file name is converted to uppercase and a null character is added to the end of the file name.

Parameters:

name Parameter *name* points to the file name.

lenMax This the number of bytes of memory that *name* points to. If there is not enough memory for the operation, a severe error results.

ext This is the new file name extension.

doReplace If *doReplace* is false (zero), the new extension in parameter *ext* is added only if the file name currently has no extension. However, if *doReplace* is true (non-zero), the new extension replaces any existing file name extension.

u4namePiece

Usage: void u4namePiece(char *result, int lenResult, const char *from , int givePath, int giveExt)

Description: Part of the file name, including the main part of the file name, is extracted and copied into *result*. If *givePath* is true, any path is included and if *giveExt* is true, any file name extension is included.

Parameters:

result Parameter *result* points to where the resulting file name is copied.

lenResult This is the number of bytes of memory that *result* points to. If there is not enough memory for the operation, a severe error results.

from Parameter *from* points to the file name.

givePath This is true if the file name path should be copied into result.

giveExt This is true if the file name extension should be copied into result.

u4ncpy

Usage: unsigned u4ncpy(char *to, const char *from, unsigned lenTo)

Description: This function copies one string to the other. It will guarantee null termination, provided *lenTo* is greater than zero, and will guarantee that memory will not be overwritten when the **sizeof** operator is used as the last parameter. For these reasons, **u4ncpy** tends to be more useful than standard C runtime library functions **strcpy** and **strncpy**.

Parameters:

to This points to the storage where *from* is copied to.

from *from* points to the string which is to be copied.

lenTo This is the number of bytes of allocated memory that *to* points to.

Returns: The number of characters actually copied is returned. This value is always less than or equal to parameter *lenTo* and does not include the null termination character.

```
void  display20( char *ptr )
{
    char buf[21] ;

    u4ncpy( buf, ptr, sizeof(buf) ) ;
    cout << "Display: " << buf ;
}
```

u4yymmdd

Usage: void u4yymmdd(char *result)

Description: The current year, month and day is formatted into *result*.

Parameters: Parameter result should point to three bytes of storage. The non-century part of the year is stored in the first byte, the month in the second and the day in the third. For example, if the year is 1990, *result*[0] becomes **(char)** 90.

Appendix A: Error Codes

The following tables list the error codes that are returned by CodeBase functions, which signal that an error has occurred. The tables display the integer constants and the corresponding small error descriptions accompanied by a more detailed explanation.

For more information concerning error code returns and error functions, please refer to the chapter Error Functions.

General Disk Errors

Constant Name	Value	Meaning
e4close	**-10**	**Closing File** An error occurred while attempting to close a file.
e4create	**-20**	**Creating File** This error could be caused by specifying an illegal file name, attempting to create a file which is open, having a full directory, or by having a disk problem. Refer to the **CODE4.safety** and **CODE4.errCreate** flags in the CodeBase chapter of this manual for more information on how to prevent this error from occurring. This error also results when the operating system doesn't have enough file handles. See **e4numFiles**, below, for more information.
e4len	**-30**	**Determining File Length** An error occurred while attempting to determine the length of a file. This error occurs when CodeBase runs out of valid file handles. See **e4numFiles**, below, for more information.
e4lenSet	**-40**	**Setting File Length** An error occurred while setting the length of a file. This error occurs when an application does not have write access to the file or is out of disk space.
e4lock	**-50**	**Locking File** An error occurred while trying to lock a file. Generally this error occurs when the **CODE4.lockEnforce** is set to true (non-zero) and an attempt is made to modify an unlocked record. This error can also occur when **file4lock** is called more than once for the same set of bytes without calling **file4unlock** between the calls to **file4lock**.

e4open	**-60**	**Opening File** A general file failure occurred opening a file. This error may also include of the **-6x** errors listed below if the selected compiler or operating system does not allow for distinguishing between various file errors.
e4permiss	**-61**	**Permission Error Opening File** Permission to open the file as specified was denied. For example, another user may have the file opened exclusively.
e4access	**-62**	**Access Error Opening File** Invalid open mode was specified. This would usually occur if there was a discrepancy between CodeBase and the implementation on a compiler or operating system (i.e. a compatibility problem).
e4numFiles	**-63**	**File Handle Count Overflow Error Opening File** Maximum file handle count exceeded. The number of file handles available to an application or DLL is determined in the 'startup' code of the 'C' runtime library that the application or DLL is linked with. The default value is 20 file handles. The server executable has been built with modified runtime libraries that support up to 255 file handles being available. Therefore, this error is unlikely to occur in client-server applications, where the server is opening all files. If this error does occur in a client-server application, you must modify your application to use less files at any given time. In non client-server 'C/C++' applications, this error is much more likely to occur, due to the smaller number of default file handles available. However, this default value can be changed to accommodate the opening of more files. Refer to the COMPILER.TXT file, installed in your compiler directory (e.g. \MSC8), for instructions on how to do this.
e4fileFind	**-64**	**File Find Error Opening File** File was not found as specified.
e4instance	**-69**	**Duplicate Instance Found Error Opening File** An attempt to open a duplicate instance of a file has been denied. The **CODE4.singleOpen** setting influences how duplicate accessing of a file from within the same executable is performed. This error indicates one of two possibilities: **1.** An open request has occurred but an active data handle in the same executable is inhibiting the open. **2.** In a client-server environment, a different client

		application has explicitly requested and has been granted exclusive client-access to the specified file.
e4read	**-70**	**Reading File** An error occurred while reading a file. This could be caused by calling **d4go** with a nonexistent record number.
e4remove	**-80**	**Removing File** An error occurred while attempting to remove a file. This error will occur when the file is opened by another user or the current process, and an attempt is made to remove that file.
e4rename	**-90**	**Renaming File** An error occurred while renaming a file. This error can be caused when the file name already exists.
e4unlock	**-110**	**Unlocking File** An error occurred while unlocking part of a file. This error can occur when an attempt is made to unlock bytes that were not locked with **file4lock**.
e4write	**-120**	**Writing to File** This error occurs when the disk is full.

Data File Specific Errors

Constant Name	Value	Meaning
e4data	**-200**	**File is not a Data File** This error occurs when attempting to open a file as a **.DBF** data file when the file is not actually a true data file. If the file is a data file, its header and possibly its data is corrupted.
e4fieldName	**-210**	**Unrecognized Field Name** A function, such as **d4field**, was called with a field name not present in the data file.
e4fieldType	**-220**	**Unrecognized Field Type** A data field had an unrecognized field type. The field type of each field is specified in the data file header.
e4recordLen	**-230**	**Record Length too Large** The total record length is too large. The maximum is **USHRT_MAX-1** which is 65534 for most compilers.
e4append	**-240**	**Record Append Attempt Past End of File**
e4seek	**-250**	**Seeking**

		This error can occur if int **d4seekDouble** tries to do a seek on a non-numeric tag.

Index File Specific Errors

Constant Name	Value	Meaning
e4entry	**-300**	**Tag Entry Missing** A tag entry was not located. This error occurs when a key, corresponding to a data file record, should be in a tag but is not.
e4index	**-310**	**Not a Correct Index File** This error indicates that a file specified as an index file is not a true index file. Some internal index file inconsistency was detected.
e4tagName	**-330**	**Tag Name not Found** The tag name specified is not an actual tag name. Make sure the name is correct and that the corresponding index file is open.
e4unique	**-340**	**Unique Key Error** An attempt was made to add a record or create an index file which would have resulted in a duplicate tag key for a unique key tag. In addition, **t4unique** returned **e4unique** or, when creating an index file, the member **TAG4INFO.unique** specified **e4unique**.
e4tagInfo	**-350**	**Tag information is invalid** Usually occurs when calling **d4create** or **i4create** with invalid information in the input **TAG4INFO** structure.

Expression Evaluation Errors

Constant Name	Value	Meaning
e4commaExpected	**-400**	**Comma or Bracket Expected** A comma or a right bracket was expected but there was none. For example, the expression "SUBSTR(A" would cause this error because a comma would be expected after the 'A'.
e4complete	**-410**	**Expression not Complete** The expression was not complete. For example, the expression "FIELD_A +" would not be complete because there should be something else after the '+'.

e4dataName	**-420**	**Data File Name not Located** A data file name was specified but the data file was not currently open. For example, if the expression was "DATA->FIELD_NAME", but no currently opened data file has "DATA" as its alias. Refer to **d4alias** and **d4aliasSet**.
e4lengthErr	**-422**	**IIF() Needs Parameters of Same Length** The second and third parameters of dBASE function IIF() must resolve to exactly the same length. For example, IIF(.T., "12", "123") would return this error because character expression "12" is of length two and "123" is of length three.
e4notConstant	**-425**	**SUBSTR() and STR() need Constant Parameters** The second and third parameters of functions SUBSTR() and STR() require constant parameters. For example, SUBSTR("123", 1, 2) is fine; however, SUBSTR("123", 1, FLD_NAME) is not because FLD_NAME is not a constant.
e4numParms	**-430**	**Number of Parameters is Wrong** The number of parameters specified in a dBASE expression is wrong.
e4overflow	**-440**	**Overflow while Evaluating Expression** The dBASE expression was too long or complex for CodeBase to handle. Such an expression would be extremely long and complex. The parsing algorithm limits the number of comparisons made in a query. Thus, very long expressions can not be parsed. Use **code4calcCreate** to 'shorten' the expression.
e4rightMissing	**-450**	**Right Bracket Missing** The dBASE expression is missing a right bracket. Make sure the expression contains the same number of right as left brackets.
e4typeSub	**-460**	**Sub-expression Type is Wrong** The type of a sub-expression did not match the type of an expression operator. For example, in the expression "33 .AND. .F.", the "33" is of type numeric and the operator ".AND." needs logical operands.
e4unrecFunction	**-470**	**Unrecognized Function** A specified function was not recognized. For example, the expression "SIMPLE(3)" is not valid.

e4unrecOperator	**-480**	**Unrecognized Operator** A specified operator was not recognized. For example, in the dBASE expression "3 } 7", the character '}' is in a place where a dBASE operator would be expected.
e4unrecValue	**-490**	**Unrecognized Value** A character sequence was not recognized as a dBASE constant, field name, or function.
e4unterminated	**-500**	**Unterminated String** According to dBASE expression syntax, a string constant starts with a quote character and ends with the same quote character. However, there was no ending quote character to match a starting quote character.
e4tagExpr	**-510**	**Expression Invalid for Tag** The expression is invalid for use within a tag. For example, although expressions may refer to data aliases, tag expressions may not. This error usually occurs when specifying **TAG4INFO** expressions when calling **d4create** or **i4create**.

Optimization Errors

Constant Name	Value	Meaning
e4opt	**-610**	**Optimization Error** A general CodeBase optimization error was discovered.
e4optSuspend	**-620**	**Optimization Removal Error** An error occurred while suspending optimization.
e4optFlush	**-630**	**Optimization File Flushing Failure** An error occurred during the flushing of optimized file information.

Relation Errors

Constant Name	Value	Meaning
e4relate	**-710**	**Relation Error** A general CodeBase relation error was discovered.
e4lookupErr	**-720**	**Matching Slave Record Not Located** CodeBase could not locate the master record's corresponding slave record.

e4relateRefer	**-730**	**Relation Referred to Does Not Exist or is Not Initialized** Referenced a non-existent or improperly initialized relation. Possible cases are: non-initialized memory or an invalid pointer has been passed to a relate module function, or function calls have occurred in an invalid sequence (for example, **relate4skip** may not be called unless **relate4top** has previously been called).

Severe Errors

Constant Name	Value	Meaning
e4info	**-910**	**Unexpected Information** CodeBase discovered an unexpected value in one of its internal variables.
e4memory	**-920**	**Out of Memory** CodeBase tried to allocate some memory from the heap, in order to complete a function call, but no memory was available. This usually occurs during a database update process, which happens when a record is appended, written or flushed to disk. During the update, if a new tag block is required, CodeBase will attempt to allocate more memory. If the memory is not available, CodeBase will return the "Out of Memory" error. If this error occurs during the updating process, the index file will most likely become corrupt. It is virtually impossible to escape this error so it is advantageous to allocate all the memory required before any updates are made. Set **CODE4.memStartBlock** to the maximum number of blocks required before opening any index files. See the "Frequently Asked Questions" document for more details.
e4parm	**-930**	**Unexpected Parameter** A CodeBase function was passed an unexpected parameter value. This can happen when the application programmer forgets to initialize some pointers and thus null pointers are passed to a function.
e4parmNull	**-935**	**Null Input Parameter unexpected** Unexpected parameter - null input.
e4demo	**-940**	**Exceeded Maximum Record Number for Demonstration** Exceeded maximum support due to demo version of CodeBase.
e4result	**-950**	**Unexpected Result**

		A CodeBase function returned an unexpected result to another CodeBase function.
e4verify	**-960**	**Structure Verification Failure** Unexpected result while attempting to verify the integrity of a structure.
e4struct	**-970**	**Data Structure Corrupt or not Initialized** CodeBase internal structures have been detected as invalid.

Not Supported Errors

Constant Name	Value	Meaning
e4notIndex	**-1010**	**Library compiled with S4OFF_INDEX** An attempt was made to call an indexing function when the library was compiled without them.
e4notMemo	**-1020**	**Library compiled with S4OFF_MEMO** An attempt was made to call a memo function when the library was compiled without them.
e4notWrite	**-1040**	**Library compiled with S4OFF_WRITE** An attempt was made to write to a file when the library was compiled without this capability.
e4notClipper	**-1050**	**Function unsupported: library compiled with S4CLIPPER** Function not supported in **S4CLIPPER** implementation.
e4notSupported	**-1090**	**Function unsupported** Operation generally not supported
e4version	**-1095**	**Application/Library version mismatch** Version mismatch (eg. client version mismatches server version).

Memo Errors

Constant Name	Value	Meaning
e4memoCorrupt	**-1110**	**Memo File Corrupt** A memo file or entry is corrupt.
e4memoCreate	**-1120**	**Error Creating Memo File** For example, the **CODE4.memSizeMemo** is set to an invalid value.

Transaction Errors

Constant Name	Value	Meaning
e4transViolation	**-1200**	**Transaction Violation Error** Attempt to perform an operation within a transaction which is disallowed. (eg. **d4pack**, **d4zap**, etc.)
e4trans	**-1210**	**Transaction Error** Transaction failure. A common occurrence is if the transaction file is detected to be in an invalid state upon opening.
e4rollback	**-1220**	**Transaction Rollback Failure** An unrecoverable failure occurred while attempting to perform a rollback (eg. a hard disk failure)
e4commit	**-1230**	**Transaction Commit Failure** Transaction commit failure occurred.
e4transAppend	**-1240**	**Error Appending Information to Log File** An error has occurred while attempting to append data to the transaction log file. One possibility is out of disk space. In the client-server configuration, all clients will likely be disconnected and the server will shut down after this failure.

Communication Errors

Constant Name	Value	Meaning
e4corrupt	**-1300**	**Communication Information Corrupt** Connection information corrupt. In general would indicate a network hardware/software failure of some sort. For example, out of date device drivers may be being used on either a client or a server machine. Alternatively, the client application may have been compiled with an unsupported compiler, using unsupported compiler switches, or under an unsupported operating system, resulting in perceived network problems.
e4connection	**-1310**	**Connection Failure** A connection failure. For example, a connection failed to be established or got terminated abruptly by the network.
e4socket	**-1320**	**Socket Failure** A socket failure. All CodeBase software use sockets as their

		basis for communications. This error indicates a failure in the socket layer of the communications. For example, the selected communication protocol may be unsupported on the given machine. Alternatively, an unsupported version of the networking software may be being used (eg. Windows Sockets 1.0 or Novell 2.x).
e4net	**-1330**	**Network Failure** A network error occurred. Some CodeBase communications protocols are dependent on network stability. For example, if the local file-server is shut-down, CodeServer or CodeBase may be unable to continue operations, and may therefore fail with an **e4net** error. Alternatively, a physical network error may be detected (for example, if a network cable is physically cut or unplugged, thus removing the physical connection of the computer from the network.)
e4loadlib	**-1340**	**Failure Loading Communication DLL** An attempt to load the specified communication DLL has failed. Ensure that the requested DLL is accessible to the application. This error may also occur if attempting to start a client or server under Windows if Windows is unstable.
e4timeOut	**-1350**	**Network Timed Out** This error occurs whenever CodeBase has timed out after **CODE4.timeout** seconds have elapsed.
e4message	**-1360**	**Communication Message Corrupt** A communication message error has been detected. For example, a client may have not been able to properly send a complete message to the server.
e4packetLen	**-1370**	**Communication Packet Length Mismatch** A packet length error has been detected. Possibly the CodeBase client software mismatches the server implementation.
e4packet	**-1380**	**Communication Packet Corrupt** A packet corruption has been detected. Check **e4corrupt** for potential causes of this failure.

Miscellaneous Errors

Constant Name	Value	Meaning
e4max	**-1400**	**CodeBase Capabilities Exceeded (system maxed out)** The physical capabilities of CodeBase or CodeServer have been maxed out. For example, the maximum allowable connections for a computer may have been exceeded by the

		CodeServer. Often these errors can be solved by modifying system or network configuration files which have placed arbitrary limits on the system. This error will also be generated when the maximum number of users for the server is exceeded.
e4codeBase	**-1410**	**CodeBase in an Unacknowledged Error State** CodeBase failed due to being in an error state already. Generally comes out as an error return code if a high-level function is called after having disregarded a CodeBase error condition.
e4name	**-1420**	**Name not Found error** The specified name was invalid or not found. For example, **d4index** was called with a non-existent index alias or the specified name was not found in the catalog file.
e4authorize	**-1430**	**Authorization Error (access denied)** The requested operation could not be performed because the requester has insufficient authority to perform the operation. For example, a user without creation privileges has made a call to **d4create.**

Server Failure Errors

Constant Name	Value	Meaning
e4server	**-2100**	**Server Failure** A client-server failure has occurred. In this case, the client connection was probably also lost.
e4config	**-2110**	**Server Configuration Failure** An error has been detected in the server configuration file. The configuration file is only accessed when the server is first started, so once the server is operational, this error cannot occur.
e4cat	**-2120**	**Catalog Failure** A catalog failure has occurred. For example, the catalog file may exist but may be corrupt.

Appendix B: Return Codes

When CodeBase programs use return codes, they are documented as integer constants. These integer constants are defined in header file "D4DATA.H" and are listed below:

Constant Name	Value	Meaning
r4success, r4same	**0**	In general, a return of zero means that a function call was successful. **r4same** is returned by the function **relate4next** and it means that the new relation is the next slave of the same master.
r4found, r4down	**1**	**r4found** indicates that a search key was located. **r4down** is returned by the function **relate4next** and it means that the new relation is one level down in the relation set.
r4after, r4complete	**2**	**r4after** means that a search call was not successful and that a index or data file is positioned after the requested search key. **r4complete** is returned by the function **relate4next** and it means that there are no more relations left to iterate through.
r4eof	**3**	This constant indicates an end of file condition.
r4bof	**4**	This constant indicates a beginning of file condition.
r4entry	**5**	This return indicates that a record or tag key is missing.
r4descending	**10**	This code specifies that a tag should be in descending order.
r4unique	**20**	This code indicates that a tag should have unique keys or an attempt was made to add a non-unique key.
r4uniqueContinue	**25**	This code indicates to continue reindexing or adding keys to other tags when an attempt is made to add a non-unique key. The non-unique key is not added to the tag.
r4locked	**50**	This return indicates that a part of a file is locked by another user.
r4noCreate	**60**	This return indicates that a file could not be created.
r4noOpen	**70**	This return indicates that a file could not be opened.
r4noTag	**80**	This return indicates that a tag name could not be found.
r4terminate	**90**	This return indicates that a slave record was not located during a lookup
r4inactive	**110**	There is no active transaction
r4active	**120**	There is an active transaction

r4authorize	**140**	User lacks authorization to perform requested action
r4connected	**150**	An active connection already exists.
r4logOpen	**170**	An attempt was made to open or create a new log file when an open log file already exists.

CodeBase also has a set of character constants that can be used in functions as parameters or return codes. These constants represent the different field types or describes how the information is formatted.

Constant Name	Value	Meaning
r4bin	'B'	Binary field.
r4str	'C'	Character field.
r4date	'D'	Date field.
r4float	'F'	Floating Point field.
r4gen	'G'	General field.
r4log	'L'	Logical field.
r4memo	'M'	Memo field.
r4num	'N'	Numeric or Floating Point field.
r4dateDoub	'd'	A date is formatted as a **(double)**.
r4numDoub	'n'	A numeric value is formatted as a **(double)**.

Appendix C: dBASE Expressions

In CodeBase, a dBASE expression is represented as a character string and is evaluated using the expression evaluation functions. dBASE expressions are used to define the keys and filters of an index file. They can be useful for other purposes such as interactive queries.

WARNING

The dBASE functions listed in this appendix are not C functions to be called directly from C programs. They are dBASE functions which are recognized by the CodeBase expression evaluation functions. In the same manner, C functions and variables cannot appear in an dBASE expression.

General dBASE Expression Information

All dBASE expressions return a value of a specific type. This type can be Numeric, Character, Date or Logical. A common form of a dBASE expression is the name of a field. In this case, the type of the dBASE expression is the type of the field. Field names, constants, and functions may all be used as parts of a dBASE expression. These parts can be combined with other functions or with operators.

Example dBASE Expression: "FIELD_NAME"

Note

In this manual all dBASE expressions are contained in double quotes (" "). The quotes are not considered part of the dBASE expression. Any double quotes that are contained within the dBASE expression will be denoted as ' \" '. This method is used to remain consistent with the format of C++ string constants.

Field Name Qualifier

It is possible to qualify a field name in a dBASE expression by specifying the data file.

Example dBASE Expression: "DBALIAS->FLD_NAME"

Observe that the first part, the qualifier, specifies a data file alias (see **d4alias**). This is usually just the name of the data file. Then there is the "->" followed by the field name.

dBASE Expression Constants

dBASE Expressions can consist of a Numeric, Character or Logical constant. However, dBASE expressions which are constants are usually not very useful. Constants are usually used within a more complicated dBASE expression.

A Numeric constant is a number. For example, "5", "7.3", and "18" are all dBASE expressions containing Numeric constants.

Character constants are letters with quote marks around them. " 'This is data' ", " 'John Smith' ", and " \"John Smith\" " are all examples of dBASE expressions containing Character constants. If you wish to specify a character constant with a single quote or a double quote contained inside it, use the other type of quote to mark the Character constant. For example," \"Man's\" " and " ' \"Ok\" ' " are both legitimate Character constants.

Unless otherwise specified, all dBASE Character constants in this manual are denoted by single quote characters.

Constants .TRUE. and .FALSE. are the only legitimate Logical constants. Constants .T. and .F. are legitimate abbreviations.

dBASE Expression Operators

Operators like '+' , ' * ', or '<' are used to manipulate constants and fields. For example, "3+8" is an example of a dBASE expression in which the Add operator acts on two numeric constants to return the numeric value "11".

The values an operator acts on must have a type appropriate for the operator. For example, the divide '/' operator acts on two numeric values.

Precedence

Operators have a precedence which specifies operator evaluation order. The precedence of each operator is specified in the following tables which describe the various operators. The higher the precedence, the earlier the operation will be performed. For example, 'divide' has a precedence of 6 and 'plus' has a precedence of 5 which means 'divide' is evaluated before 'plus'. Consequently, "1+4/2" is "3". Evaluation order can be made explicit by using brackets. For example, "1+2 * 3" returns "7" and "(1+2) * 3" returns "9".

Numeric Operators

The numeric operators all operate on Numeric values.

Operator Name	Symbol	Precedence

Add	+	5
Subtract	-	5
Multiply	*	6
Divide	/	6
Exponent	** or ^	7

Character Operators

There are two character operators, named "Concatenate I" and "Concatenate II", which combine two character values into one. They are distinguished from the Add and Subtract operators by the types of the values they operate on.

Operator Name	Symbol	Precedence
Concatenate I	+	5
Concatenate II	-	5

Examples:

" 'John ' + 'Smith' " becomes " 'John Smith' "

" 'ABC' + 'DEF' " becomes " 'ABCDEF' "

Concatenate II is slightly different as any spaces at the end of the first Character value are moved to the end of the result.

" 'John'-'Smith ' " becomes " 'JohnSmith ' "

" 'ABC' - 'DEF' " becomes " 'ABCDEF' "

" 'A ' - 'D ' " becomes " 'AD ' "

Relational Operators

Relational Operators are operators which return a Logical result (which is either true or false). All operators, except Contain, operate on Numeric, Character or Date values. Contain operates on two character values and returns true if the first is contained in the second.

Operator Name	Symbol	Precedence
Equal To	=	4
Not Equal To	<> or #	4
Less Than	<	4
Greater Than	>	4
Less Than or Equal To	< =	4
Greater Than or Equal To	> =	4
Contain	$	4

Examples:

" 'CD' $ 'ABCD' " returns ".T."

" 8<7 " returns ".F."

Logical Operators

Logical Operators return a Logical Result and operate on two Logical values.

Operator Name	Symbol	Precedence
Not	.NOT.	3
And	.AND.	2
Or	.OR.	1

Examples " .NOT. .T. " returns ".F."

" .T. .AND. .F." returns ".F."

dBASE Expression Functions

A function can be used as a dBASE expression or as part of an dBASE expression. Like operators, constants, and fields, functions return a value. Functions always have a function name and are followed by a left and right bracket. Values (parameters) may be inside the brackets.

Function List

ALLTRIM(CHAR_VALUE)

This function trims all of the blanks from both the beginning and the end of the expression.

ASCEND(VALUE)

This function is not supported by dBASE, FoxPro or Clipper.

ASCEND() accepts all types of parameters, except complex numeric expressions. ASCEND() converts all types into a Character type in ascending order. In the case of numeric types, the conversion is done so that the sorting will work correctly even if negative values are present.

CHR(INTEGER_VALUE)

This function returns the character whose numeric ASCII code is identical to the given integer. The integer must be between 0 and 255.

Example: CHR(65) returns “A”.

CTOD(CHAR_VALUE)

The character to date function converts a character value into a date value:

eg. " CTOD("11/30/88") "

The character representation is always in the format specified by the code4dateFormatSet member function which is by default "MM/DD/YY".

DATE()

The system date is returned.

DAY(DATE_VALUE)

Returns the day of the date parameter as a numeric value from "1" to "31".

eg. "DAY(DATE())"

Returns "30" if it is the thirtieth of the month.

DESCEND(VALUE)

This function is not supported by dBASE or FoxPro. DESCEND() is compatible with Clipper, only if the parameter is a Character type.

DESCEND() accepts any type of parameter, except complex numeric expressions. DESCEND() converts all types into a character type in descending order.

For example, the following expression would produce a reverse order sort on the field ORD_DATE followed by normal sub-sort on COMPANY.

eg. DESCEND(ORD_DATE) + COMPANY

See also ASCEND().

DELETED()

Returns .TRUE. if the current record is marked for deletion.

DTOC(DATE_VALUE)

DTOC(DATE_VALUE, 1)

The date to character function converts a date value into a character value. The format of the resulting character value is specified by the code4dateFormatSet member function which is by default "MM/DD/YY".

eg. " DTOC(DATE()) "

Returns the character value "05/30/87" if the date is May 30, 1987.

If the optional second argument is used, the result will be identical to the dBASE expression function *DTOS*.

For example, DTOC(DATE(), 1) will return "19940731" if the date is July 31, 1994.

DTOS(DATE_VALUE)

The date to string function converts a date value into a character value. The format of the resulting character value is "CCYYMMDD".

e.g. ." DTOS(DATE()) "

Returns the character value "19870530" if the date is May 30, 1987.

IIF(LOG_VALUE, TRUE_RESULT, FALSE_RESULT)

If 'Log_Value' is .TRUE. then IIF returns the 'True_Result' value. Otherwise, IIF returns the 'False_Result' value. Both True_Result and False_Result must be the same length and type. Otherwise, an error results.

eg. "IIF(VALUE << 0, "Less than zero ", "Greater than zero")"

	e.g. ."IIF(NAME = "John", "The name is John", "Not John ")"
LEFT(CHAR_VALUE, NUM_CHARS)	
	This function returns a specified number of characters from a character expression, beginning at the first character on the left. eg. "LEFT('SEQUITER', 3)" returns "SEQ". The same result could be achieved with "SUBSTR('SEQUITER', 1, 3)".
LTRIM(CHAR_VALUE)	
	This function trims any blanks from the beginning of the expression.
MONTH(DATE_VALUE)	
	Returns the month of the date parameter as a numeric. eg. " MONTH(DT_FIELD) " Returns 12 if the date field's month is December.
PAGENO()	
	When using the report module or CodeReporter, this function returns the current report page number.
RECCOUNT()	
	The record count function returns the total number of records in the database: eg. " RECCOUNT() " Returns 10 if there are ten records in the database.
RECNO()	
	The record number function returns the record number of the current record.
STOD(CHAR_VALUE)	
	The string to date function converts a character value into a date value: eg. " STOD("19881130") " The character representation is in the format "CCYYMMDD".
STR(NUMBER, LENGTH, DECIMALS)	
	The string function converts a numeric value into a character value. "Length" is the number of characters in the new string, including the decimal point. "Decimals" is the number of decimal places desired. If the number is too big for the allotted space, *'s will be returned. eg. " STR(5.7, 4, 2) " returns " '5.70' " The number 5.7 is converted to a string of length 4. In addition, there will be 2 decimal places.

eg. " STR(5.7, 3, 2) " returns " '***' " The number 5.7 cannot fit into a string of length 3 if it is to have 2 decimal places. Consequently, *'s are filled in.
SUBSTR(CHAR_VALUE, START_POSITION, NUM_CHARS) A substring of the Character value is returned. The substring will be 'Num_Chars' long, and will start at the 'Start_Position' character of 'Char_Value'. eg. " SUBSTR("ABCDE", 2, 3)" returns " 'BCD' " eg. "SUBSTR("Mr. Smith", 5, 1)" returns " 'S' "
TIME() The time function returns the system time as a character representation. It uses the following format: HH:MM:SS. eg. " TIME() " returns " 12:00:00 " if it is noon. eg. " TIME() " returns " 13:30:00 " if it is one thirty PM.
TRIM(CHAR_VALUE) This function trims any blanks off the end of the expression.
UPPER(CHAR_VALUE) A character string is converted to uppercase and the result is returned.
VAL(CHAR_VALUE) The value function converts a character value to a numeric value. eg. “VAL('10')” returns "10". eg. “VAL('-8.7')” returns "-8.7".
YEAR(DATE_VALUE) Returns the year of the date parameter as a numeric: eg. "YEAR(STOD('19920830')) " returns " 1992 "

Appendix D: CodeBase Limits

Following are the maximums of CodeBase:

Description	Limit
Block Size	32768 (32K)
Data File Size	1,000,000,000 Bytes
Field Width	254 for dBASE compatibility, 32767 otherwise.
Floating Point Field Width	19
Memo Entry Size	(maximum value of an unsigned integer) minus (overhead) The range of the integer depends on how many bytes are used to store an unsigned integer, which in turn depends on the system. The overhead may vary from compiler to compiler or o/s to o/s and depending on CodeBase switches (**E4MISC** causes extra memory to be allocated for corruption detection). The overhead should never exceed 100 bytes. Therefore, if the system uses a 2 byte unsigned integer, then the memo entry size is: 65,536 - 100 = 65, 436 bytes (approx. 64K) If the system uses a 4 byte unsigned integer, then the memo entry size is: 4,294,967,296 - 100 = 4,294,967,196 bytes (approx. 4G)
Number of Fields	128 for dBASE compatibility. 1022 for Clipper compatibility.
Number of Open Files	The number of open files is constrained only by the compiler and the operating environment.
Number of Tags per Index	Unlimited FoxPro 47 dBASE IV 1 Clipper
Numeric Field Width	19
Record Width	65500 (64K)

Index

—D—

—E—

—F—

—G—

—H—

—I—

—J—

—L—

—M—

—N—

—O—

—P—

—Q—

—R—

—S—

—T—

—U—

—V—

—W—

—Y—