

PRO/5[®]

***C Library
Reference Manual***

Copyright Notice

This manual is copyrighted by BASIS International Ltd., Albuquerque, New Mexico © 1997. All rights reserved. Portions © 1986 by the University of Toronto. Written by Henry Spencer. Not derived from licensed software. Portions © 1980 The Regents of the University of California. All rights reserved. Portions © Informix Software, Inc. All rights reserved. No part of this manual may be reproduced by electronic, digital, or mechanical methods without the express written permission of BASIS International Ltd. Information contained in this document is subject to change without notice.

Trademarks

Visual PRO/5™ and BASIS ODBC Driver™ are trademarks of BASIS International Ltd. BBx®, BBxPROGRESSION/4®, PRO/5 Data Server®, PRO/5®, TAOS: The BBx Developer's Workbench®, and TAOS/Views® are registered trademarks of BASIS International Ltd.

All other product names and brand names are service marks and/or trademarks or registered trademarks of their respective companies.

Second edition, first printing 10/97

BASIS International Ltd.
5901 Jefferson Street NE
Albuquerque, New Mexico 87109-3432

General Telephone	505.345.5232
General Facsimile	505.345.5082
E-Mail	info@basis.com
World Wide Web	www.basis.com
Technical Support Telephone	505.345.5021
Technical Support Facsimile	505.344.9057
Technical Support Internet	support@basis.com

Table of Contents

Introduction.....	1
Overview	1
C Library Function Summary	1
Installation and Use	2
Installation Instructions	2
Use Overview	2
Initializing the C Library	2
Creating, Deleting, and Manipulating Files	3
Closing the C Library	3
Notes.....	3
Input/Output Concepts.....	4
Introduction	4
I/O Channels.....	4
BB ^x Files and Devices	4
Disk File Types.....	5
Summary of File Types.....	5
Files	5
Fields, Records, and Files.....	6
File Pointers.....	6
FIDs and FINs	6
iop.....	7
File System Limits	7
I/O Errors and Error Trapping.....	7
Signals	8
Data Server File Access.....	8
C Library Functions	9
Overview	9
Syntax Typographical Conventions	9
Syntax Element Descriptions.....	9
fs_addkey (Add Segment to FIN)	10
Syntax	10
Description	10
Returned	10
Notes.....	10
See also.....	10
Examples	10
fs_close (Close I/O Channel)	11
Syntax	11
Description	11
Returned	11
See also.....	11
Examples	11
fs_clrfin (Prepare FIN Buffer for Use).....	13
Syntax	13
Description	13
Returned	13
See also.....	13
Examples	13

fs_conf (Pass Configuration Filename to File System)	14
Syntax	14
Description	14
Returned	14
Notes.....	14
See also.....	14
Examples	14
fs_create (Create File from FID)	15
Syntax	15
Description	15
Returned	15
Notes.....	15
See also.....	15
Examples	15
fs_erase (Erase File)	17
Syntax	17
Description	17
Returned	17
Examples	17
fs_fid (Fetch FID Information for Channel)	18
Syntax	18
Description	18
Returned	18
Notes.....	18
See also.....	18
Examples	18
fs_flush (Flush Cached Files)	20
Syntax	20
Description	20
fs_info (Fetch Information about a File/Device)	21
Syntax	21
Description	21
Returned	21
Notes.....	21
See also.....	21
Examples	21
fs_init (Initialize File System)	23
Syntax	23
Description	23
Returned	23
Notes.....	23
See also.....	23
Examples	23
fs_initfile (Erase and then Reinitialize a File)	25
Syntax	25
Description	25
Returned	25
Examples	25
fs_key (Get Key from File)	26
Syntax	26
Description	26
Returned	26
See also.....	26
Examples	26

fs_lock (Lock Opened File)	27
Syntax	27
Description	27
Returned	27
Notes.....	27
See also.....	27
Examples	27
fs_mkeyfin (Generate FIN for MKEYED File)	28
Syntax	28
Description	28
Notes.....	28
See also.....	28
Examples	28
fs_mkfid (Build FID Structure from Components)	30
Syntax	30
Description	30
Returned	30
Notes.....	30
See also.....	30
Examples	30
fs_open (Open File/Device/Pipe)	32
Syntax	32
Description	32
Returned	32
Notes.....	32
See also.....	32
Examples	33
fs_read (Read File or Device)	34
Syntax	34
Description	35
Returned	35
Notes.....	36
See also.....	36
Examples	36
fs_remove (Remove Record from Keyed File)	37
Syntax	37
Description	37
Returned	37
See also.....	37
Examples	37
fs_rename (Rename File)	38
Syntax	38
Description	38
Returned	38
Notes.....	38
Examples	38
fs_shut (Shut Down File System)	39
Syntax	39
Description	39
Returned	39
See also.....	39
Examples	39

fs_unlock (Unlock File)	41
Syntax	41
Description	41
Returned	41
See also.....	41
Examples	41
fs_write (Write to File or Device)	42
Syntax	42
Description	43
Returned	43
Notes.....	43
See also.....	43
Examples	43
Configuration File.....	45
What is the Configuration File?.....	45
Configuration File Components.....	45
Sample Configuration File.....	47
Calling the Configuration File	47
C Library Error Codes.....	48
Overview	48
E_BADARG (String/Number Mismatch).....	48
E_BADCIB (Improper File/Device Usage).....	48
E_BADHEAD (Open of File With Invalid Header).....	48
E_BREAK (Escape)	48
E_BUSY (File, Record, or Device Busy or Inaccessible)	48
E_CANT (Improper File/Device Address).....	49
E_DFAIL (Disk Not Ready).....	49
E_DISK (Invalid Disk Directory).....	49
E_DREAD (Disk Read/Write Error)	49
E_EOB (End of Buffer Overflow).....	49
E_EOF (End of File)	49
E_EOR (End of Record).....	49
E_INTERN (General I/O Error).....	49
E_INVINT (Invalid Integer).....	49
E_INVKLEN (Invalid Key Length)	50
E_INVPARAM (Invalid Parameter)	50
E_KFAIL (Missing or Duplicate Key)	50
E_MEM (System Memory Overflow)	50
E_NOF (Missing or Duplicate File)	50
E_NOTIMP (Function Not Implemented).....	50
E_NOROOM (Disk Full)	50
E_NLEN (Illegal Filename).....	50
E_PNTR (Sector Pointer Out of Range).....	50
E_TOFLOW (Directory or Table Overflow).....	50
E_USERCANT (Illegal Control Operation/ Permissions Error)	51
E_VERIF (Disk Write Error/Data Miscompare)	51
E_ULIMIT (Too Many Users)	51
E_NETERR (Network Error)	51
E_NETGONE (Network Connection Lost)	51
E_NETUSER (Network User Error)	51

Introduction

Overview

The C Library is a portable library that allows developers to access BB^x special file formats from C programs. From C programs, programmers can use the C Library to perform any fundamental BB^x file operation because it fully supports all types of BB^x files. When compiled and linked with the proper library, C programs that use the C Library correctly will run on any system that supports BB^x.

C Library Function Summary

File System Control Functions

Function	Description
fs_conf()	Reads the BB ^x configuration file.
fs_init()	Initializes the C Library.
fs_shut()	Shuts down the C Library.

Closed File Access Functions

Function	Description
fs_create()	Creates a file (may subsequently be opened).
fs_open()	Opens the named file.
fs_erase()	Deletes the named file.
fs_rename()	Renames a file.
fs_initfile()	Erases, then reinitializes a file.

Open File Access Functions

Function	Description
fs_lock()	Locks a file to other processes.
fs_unlock()	Releases a previously locked file.
fs_read()	Reads data, sets position, locks/unlocks a record.
fs_write()	Writes data, sets position, unlocks a record.
fs_flush()	Flushes cached files (for future implementation).
fs_remove()	Removes data from a keyed file.
fs_key()	Reads a key from a keyed file.
fs_fid()	Retrieves a FID for an open channel.
fs_info()	Retrieves information on a file or device.
fs_close()	Closes an open file.

Utility Functions

Function	Description
fs_mkfid()	Builds a FID structure from components.
fs_clrfin()	Prepares a FIN for use by fs_addkey() .
fs_addkey()	Adds a key specification to a FIN.
fs_mkeyfin()	Builds a FIN from textual key specifications.

Installation and Use

Installation Instructions

1. Create a scratch directory into which to install the product. For example:

```
mkdir /tmp/basis
```

2. Make the scratch directory the current directory. For example:

```
cd /tmp/basis
```

3. Copy the contents of the media shipped with this package into the current directory.
4. The current directory will then contain a number of files from the media, including the README file. Read the README file and follow the detailed instructions to perform installation and initial tests of the C Library package.

Use Overview

Any C language modules requiring the services of the C Library must include the header file, `fslib.h`, at the top of the module. For example:

```
#include "fslib.h"
```

When these modules are linked, the C Library must be linked as well, usually by specifying the `-l` (lowercase L) command line option. For example:

```
cc myprog.c -lpro5
```

If the C Library was installed using the procedures recommended in the `readme.txt` file (supplied with the C Library), the library and header files will be located in the system area (usually in `/usr/lib` and `/usr/include`, respectively). The compiler/linker will be able to locate the necessary files without further assistance.

If, however, the library and header files were not installed in the system area, special options can direct the compiler and linker to the location of the files. Typically, `-I` points to the header files, and `-L` points to the library. Check your compiler and linker manuals for clarification of these procedures.

In the following example, the files were not installed in the system area, but rather in a directory called `/users/pat/bbxlib`. The shell command directs the compiler and linker to this directory for the location of the header files and library:

```
cc -I/users/pat/bbxlib -L/users/pat/bbxlib myprog.c -lpro5
```

Initializing the C Library

The procedure for initializing the C Library depends on the name and/or location of the configuration file, as follows:

- If you have a configuration file named `config.bbx` that is located in the current directory, send a call to `fs_init()`.
- If you have a configuration file that is NOT named `config.bbx`, and/or is NOT contained in the current directory, first send a call to `fs_conf()`, then send a call to `fs_init()`.

Input/Output Concepts

Introduction

BB^x is one of the few programming languages that defines complex file structures as well as peripheral device control as part of the language specification. Since, however, input and output involve the outside world, much of this is outside the control of BB^x.

To minimize the impact of moving applications to new environments, BB^x provides programmers with an abstract I/O interface which provides powerful and efficient I/O processing without being tied down to specific hardware or host operating systems. If you are familiar with some of the popular PC versions of BASIC, you will find some similarities and a lot of differences.

BB^x is designed for sophisticated business data processing applications with heavy demands on the file system. Consequently, BB^x uses a very different approach to I/O. For example, you may find different meanings below to the terms “field” and “record.”

I/O Channels

BB^x uses the input/output channel approach to file access:

- A call to **fs_open ()** returns a pointer to a channel (ChanType *).
- All subsequent I/O operations with that file, such as **fs_read ()** and **fs_close ()** use that channel pointer.
- When all desired I/O with that file is complete, the **fs_close ()** function is used to disassociate the channel from the file.

The number of channels available determines the number of files that may be accessed at one time. BB^x will allow as many channels as the host operating system, but cannot exceed the limits imposed by the host system. An application that requires more than 16 concurrently opened files may cause problems when porting to older architectures.

BB^x Files and Devices

During the operation of the **fs_open ()** function, BB^x determines the type of file or device being opened and proceeds appropriately. The following is an example of a BB^x filename:

/<accounts>f:/usa/montana/butte.ajs

Filenames provided to BB^x consist of 1 to 4 components, as described below:

Component	Description	Mandatory?	Example(s)
Server Name	Identifies the server name or files accessed on networks via the Data Server, and must be in the form “/<servername>”.	N	/<accounts>
Disk Drive	Identifies the disk drive, and must be terminated by a colon (“:”).	N	f:
Directory	Identifies the path to the file, and must be separated by forward slash (“/”) or backslash (“\”) characters, whichever are permitted by the operating system.	N	/usa/montana/
Name	Names the individual file.	Y	butte.ajs

In BB^x, the total filename cannot exceed 255 characters.

Disk File Types

Any file that resides on a hard disk or diskette drive is called a “disk file.” (Some systems provide a RAM disk feature that uses memory to simulate a diskette drive, which BB^x considers to be a disk) The following properties pertain to disk files:

- A single disk can contain many files.
- Each file will occupy some region on the disk.
- The locations of files on the disk are maintained in directories, which are used to access the files.
- Some systems require files to exist within fixed regions, whereas some systems allow files to increase in size dynamically as more space is needed. BB^x will work completely within the region allocated for each file to build its information structures.

Summary of File Types

BB^x and the C Library support the following information structures or file types:

File Type	Non-sequential Access Method	Attributes
DIRECTORY	Not Permitted	<ul style="list-style-type: none"> • Read-only file • Sequential read returns next filename • No access by index or key • Must have minimum 256-byte buffer
STRING	Byte offset	<ul style="list-style-type: none"> • No records (ASCII "flat" file)
PROGRAM	Byte offset	<ul style="list-style-type: none"> • STRING file with header
SERIAL	Record number	<ul style="list-style-type: none"> • Series of variable length records • Size of each record saved at beginning of record (invisible to user)
INDEXED	Record number	<ul style="list-style-type: none"> • Fixed record size • Dynamic number of records
DIRECT	Record number or key	<ul style="list-style-type: none"> • Fixed record and file size
SORT	Record number or key	<ul style="list-style-type: none"> • DIRECT file with no record data (record size zero) • Fixed file size
MKEYED	Key or key path	<ul style="list-style-type: none"> • Fixed record size • Dynamic number of records • Access by record number not permitted.

Note: C-ISAM support is not provided by the C Library.

Files

A file in BB^x is either a directory, a STRING file, or a BB^x file:

- A directory is a special case of file that contains the names of files in the directory.
- A STRING file does not contain a structure recognized by BB^x and is thus accessed in the same manner as a big string.
- A BB^x file always has the string "<<bbx>>" as the first 7 bytes of the file. The next 8 bytes are the FID of the file. (See the description of FIDs and FINs.) If BB^x can read the first 7 bytes and they are the proper identifier, the following 8 bytes are checked for validity. If these bytes don't define a proper FID, `fs_open()` will return `fserr=E_BADHEAD` (invalid file header).

If the file fails the "<<bbx>>" test, it will always be considered a STRING file.

Fields, Records, and Files

Regardless of the type of file or device, the BB^x concepts of fields and records apply uniformly. For all file types, attempting to read a nonexistent record or read past the end-of-file causes an error condition. Attempting to overfill a record or a fixed-length file will also result in an error.

Record-Oriented Files

SERIAL, INDEXED, DIRECT, SORT, MKEYED, and DIRECTORY files are “record-oriented” because they are logically divided into parts called “records.” Record-oriented files can only be processed one record at a time. These records may or may not be logically subdivided into fields. When working with record-oriented files, it is up to the C programmer to manage the division of a record into fields, and the concatenation of fields into a record.

Since a read operation from a record-oriented file will always return a whole record, a buffer of sufficient size must be provided. The `fs_read()` function automatically pads (extends) short records with nulls (0x00 bytes) to fill the record in the file.

Byte-Oriented Files

STRING and PROGRAM files are “byte-oriented” because they are not logically organized into records. These files may, however, contain fields separated by field-terminator characters. For example, an ASCII text file (STRING type), which is composed of many lines (fields) separated by carriage returns (field terminators) is byte-oriented .

The C Library supports field processing when reading byte-oriented files through the use of the FFIELDS flag, which is described in the `fs_read()` section on page 34.

File Pointers

As data is read or written, the read/write location moves through the file. BB^x maintains the current read/write location in a file pointer. The file pointer contains the index of the current record for record-oriented files, or the current byte, for byte-oriented files.

A separate file pointer is kept for each open channel, and thus for the file associated with each open channel. The `fs_info()` function can be used to determine the current read/write position within a file by specifying the channel associated with that file. If the file pointer points past the last record/byte in the file, it is at end-of-file. Attempts to read data at end-of-file will result in an error. Attempts to write at end-of-file can cause an error depending on the type of file.

When a channel is opened, the file pointer is automatically set to the first logical record or byte in the file. For keyed files this is the record with the lowest key. If a file has no records (such as an empty keyed file or a SERIAL file) the pointer is set to end-of-file.

All data transfers take place at the current position within the file. The program can move the file pointer at will to control where each data transfer will take place. Usually, BB^x will automatically advance the file pointer to the next logical record/byte in the file after each transfer. Therefore, if the program does not explicitly position the file pointer, successive data transfers will step sequentially through the file. The FINDEQ and FKEYEQ flags on the `fs_read()` and `fs_write()` functions allow non-sequential movement through the file.

FIDs and FINs

A FID is a data structure that contains important file information such as type, size, and name. The header file `fslib.h` details FID structure and other important FID information. The `fs_fid()` function can be used to retrieve the FID information for any open channel. The `fs_mkfid()` function allocates and initializes the FID structure. The only function that requires that a FID be passed to it is `fs_create()`.

A FIN is a 385-byte buffer containing special key information required in order to access MKEYED files, and can be declared as simply as:

```
char myfin[385];
```

The internal structure of a FIN is not documented here, since convenient library functions are provided to manipulate FINs. A valid FIN is always terminated with a 0xff byte.

iop

A global structure variable, **iop**, is used for information passing in certain circumstances. Its structure is defined in **fslib.h**.

File System Limits

BB^x supports files up to 2³¹-1 bytes. Each record can contain up to 32,767 bytes. A key may be up to 1024 bytes. Files can contain any number of records, as long as the maximum file size is not exceeded. The host system will also impose limits. Check your operating system manual for file system limit details.

I/O Errors and Error Trapping

The C Library provides a pair of global variables (**fserr** and **fserrs**) to indicate error conditions, as follows:

- If a function is successful, **fserr** is set to zero.
- If a function is unsuccessful, **fserr** is set to an error code (listed beginning on page 48). Nearly all library functions return the value of **fserr** in addition to setting it.
- In some cases, a system-specific error code will additionally be placed in **fserrs**.

If an error should occur during the I/O process, the programmer can make very few assumptions about the state of the file written or the data read:

- If the error occurred while positioning the file pointer, then the file pointer should be located in the position it was before the error.
- If the error occurred during a read or write, no guarantees are made regarding the amount of data that was actually transferred.
- If an end-of-record error occurs during a write, it is not safe to assume that data up to the end of record has been successfully written.

The BB^x error codes break down I/O errors into several specific causes. Unfortunately, BB^x relies on the host system to provide the cause of the error. Some host systems are not as specific as BB^x. When programming traps for specific I/O errors, I/O error codes may be less specific than documented in this manual.

Error Code Differences

The error codes returned by the C Library are one number higher than those returned by BB^x due to the difference in meaning of Error code zero:

- In BB^x, Error code zero denotes an actual error.
- In C, Error code zero denotes the successful completion of a function.

The error code macros provided in **fserr.h** eliminate the need to use numeric error codes.

Signals

The C Library utilizes two signals SIGPIPE and SIGALRM, which are important to C programmers attempting to catch signals with the **signal (2)** system call:

- The SIGPIPE signal is used to handle spooling on UNIX systems. (The **fs_init ()** function arranges to ignore the SIGPIPE signal.)
- The SIGALRM signal is used to detect timeouts on I/O operations.

Programmers can catch either signal without adversely affecting the C Library functions. However, some C Library functions reset the SIGALRM signal. Programmers may need to arrange to catch these signals repetitively. Even with this precaution, a SIGALRM event could be missed during a C Library function call.

If possible, avoid using the SIGPIPE and SIGALRM signals in applications requiring the services of the C Library.

Data Server File Access

The C Library can access files through a PRO/5 Data Server in the same manner as BB^x if the SETOPTS bit in the configuration file is set. See the *PRO/5 Data Server Guide* for the correct syntax.

C Library Functions

Overview

This section describes the functions supported by the C Library. The listings are in alphabetical order, and include information about the function syntax, purpose and operation, returned data, and important notes. Many listings also contain examples.

Syntax Typographical Conventions

The following describes the syntax entries listed in each C Library function:

- Boldface non-italicized characters and punctuation must be entered exactly as shown.
- Boldface italicized characters represent variables such as filenames, field numbers, field or record length, etc.
- Optional items are indicated by braces { }.

Syntax Element Descriptions

The following describes the common syntax elements used in this manual:

Element	Description
<i>arglist</i>	An argument list
<i>buffer</i>	Pointer to a temporary storage space
<i>channel</i>	An I/O path to a file or device
<i>fid</i>	File identification block
<i>field</i>	Field number within a record
<i>fin</i>	File information block
<i>flags</i>	Binary or mnemonic indicators used to select options
<i>keydef</i>	Key definition for an MKEYED file
<i>knum</i>	Key number
<i>keysiz</i>	Number of bytes in a key
<i>len</i>	Length of a key segment
<i>length</i>	Length of data to be read or written
<i>level</i>	File system release level number
<i>name</i>	Name of a file
<i>offset</i>	Displacement from the beginning of a field or record
<i>records</i>	Number of records in a file
<i>recsiz</i>	Number of bytes per record

fs_addkey (Add Segment to FIN)

Syntax

fs_addkey (fin, knum, field, offset, len, flags)

Syntax Item	Description
char *fin	Pointer to a 385-character block for FIN storage
int knum	Key number (0 to 15)
int field	Record field number (0 to 255)
int offset	Offset into record or field (1 to 1024)
int len	Length of key segment (1 to 120)
int flags	Key flags (bitwise "or" of): <ul style="list-style-type: none"> 0x1 Descending 0x2 Unique 0x4 Business math type numeric key

Description

This function is used to add a key to a FIN buffer. It uses a pointer to access an existing FIN buffer (0xff terminated) and then adds a key definition segment to that FIN buffer.

Returned

Returned Item	Description
fserr	Global error code variable.

Notes

The `fs_clrfin()` function must be called prior to the first invocation of `fs_addkey()`.

`fs_mkeyfin()` is a higher level function that completely builds a FIN, given a BB^x-style key specification. The `fs_addkey()` function probably need not be used directly by the C programmer.

See also

`fs_clrfin()`, `fs_create()`, `fs_fin()`, `fs_mkeyfin()`

Examples

BB^x Example

No equivalent.

C Example

```
char fin[385];

fs_clrfin(fin);
fs_addkey (fin, 0, 3, 1, 5, 0);
fs_addkey (fin, 0, 1, 1, 1, 0);
```

fs_close (Close I/O Channel)

Syntax

`fs_close (channel)`

Syntax Item	Description
<code>ChanType *channel</code>	I/O channel to be closed.

Description

This function closes the file or device associated with an open channel.

Returned

Returned Item	Description
<code>fserr</code>	Global error code variable.

See also

`fs_create()`, `fs_open()`

Examples

BB^x Example

```
1000 OPEN (1,ERR=9000) "MYFILE.DAT"
1010 PRINT "Ok"
1020 REM Code to process file goes here.
...
2000 CLOSE (1,ERR=9000)
2010 PRINT "Done"
2020 END
...
9000 PRINT "Error",ERR
9010 END
```

C Example

```
#include "fslib.h"

#define FSLEVEL 1
main()
{
    ChanType *chan1;
    if (fs_init(FSLEVEL) != 0) {
        printf ("Can't start filesystem\n");
        exit (-1);
    }

    if ( (chan1 = fs_open ("MYFILE.DAT", 0))
        == (ChanType *) 0) {
        printf ("Error %d\n", fserr);
        exit (-1);
    }
    printf ("Ok\n");

    /* Code to process file goes here. */
```

C Example

```
    if (fs_close(chan1) != 0) {
        printf ("Error %d\n", fserr);
        exit (-1);
    }
    printf ("Done\n");
    fs_shut();
}
```

fs_clrfin (Prepare FIN Buffer for Use)

Syntax

```
fs_clrfin (fin)
```

Syntax Item	Description
<code>char *fin</code>	Pointer to a 385 character block for FIN storage.

Description

This macro terminates an existing FIN buffer with the required 0xff byte. This termination is required prior to the first call to `fs_addkey()`. It is recommended that programmers use the `fs_mkeyfin()` function which produces the same result as `fs_clrfin()` and `fs_addkey()` together. It will still be necessary to allocate FIN storage.

Returned

Nothing.

See also

`fs_addkey()`, `fs_create()`, `fs_fin()`, `fs_mkeyfin()`

Examples

BB* Example

No equivalent.

C Example

```
char fin[385];

fs_clrfin(fin);
fs_addkey (fin, 0, 3, 1, 5, 0);
fs_addkey (fin, 0, 1, 1, 1, 0);
```

fs_conf (Pass Configuration Filename to File System)

Syntax

```
fs_conf (name)
```

Syntax Item	Description
<code>char *name</code>	The text "-c", followed by the name of the configuration file.

Description

Upon startup (invocation of `fs_init()`), the BB^x file system tries to read a special configuration file that customizes the system and establishes parameters that are different from the defaults. Normally, this file is named `config.bbx`. The `fs_conf()` function allows a different filename to be specified.

Returned

Returned Item	Description
<code>fserr</code>	Global error code variable.

Notes

The `fs_conf()` function must be called prior to calling `fs_init()` so that the file system can initialize properly according to information in the configuration file.

The configuration filename must be preceded by the text "-c".

See also

`fs_init()`, `fs_shut()`

Examples

BB ^x Example
Invoking BB ^x with <code>-cmyconfig.bbx</code> on command line.

C Example
<code>fs_conf ("-cmyconfig.bbx");</code>

fs_create (Create File from FID)

Syntax

```
fs_create ( fid {, fin})
```

Syntax Item	Description
FID *fid	Pointer to an existing FID structure.
char *fin	Pointer to an existing FIN structure.

Description

This function is called to create a file. The *fin* parameter is needed only if the file being created is a multi-keyed MKEYED type file.

Returned

Returned Item	Description
fserr	Global error code variable.

Notes

Use the `fs_mkfid()` function to build the FID in one step. The FIN is assembled by the `fs_mkeyfin()` function.

See also

`fs_close()`, `fs_fid()`, `fs_mkeyfin()`, `fs_mkfid()`, `fs_open()`

Examples

The following examples create an MKEYED file:

BB* Example

```
1000 MKEYED "MK.DAT", [3:1:5]+[1:1:1], 100, 50, ERR=9000
1010 PRINT "Ok"
1020 END
9000 PRINT "Error ", ERR
9010 END
```

C Example

```
#include "fslib.h"

#define FSLEVEL 1
main()
{
    FID *fid;
    char fin[385];

    if (fs_init(FSLEVEL) != 0) exit (-1);

    fs_mkeyfin (fin, "[3:1:5]+[1:1:1]",
                (char *)0);
    fid = fs_mkfid (T_BTREE, 0, 100, 50,
                  "MK.DAT");
    fs_create (fid, fin);
    free (fid);
}
```

C Example

```
if (fserr != 0)
    printf ("Error %d\n", fserr);
else
    printf ("Ok\n");
fs_shut();
}
```

fs_erase (Erase File)

Syntax

`fs_erase (name)`

Syntax Item	Description
<code>char *name</code>	The name of the file to be erased.

Description

Erases the named file. The operation will fail if the file is currently open.

Returned

Returned Item	Description
<code>fserr</code>	Global error code variable.

Examples

BB^x Example

```
1000 ERASE "MYFILE", ERR=2000
...
2000 PRINT "Error", ERR
2010 END
```

C Example

```
...
if (fs_erase("MYFILE") != 0) {
    printf ("Error %d\n", fserr);
    exit (-1);
}
...
```

fs_fid (Fetch FID Information for Channel)

Syntax

```
fs_fid (channel)
```

Syntax Item	Description
ChanType *channel	Pointer to an open channel.

Description

This function returns a pointer to the FID structure associated with a currently open channel.

Returned

Returned Item	Description
fserr	Global error code variable.
(FID *) (iop.s1)	File FID information.
iop.s1 + sizeof(FID)	Filename (See <code>fslib.h</code>).

Notes

The pointer returned points to the only copy of the FID structure that exists. The programmer is advised to take precautions not to corrupt it.

See also

```
fs_create(), fs_info(), fs_open()
```

Examples

The following examples use the FID function to determine the file type:

BB^x Example

```
1000 OPEN (1,ERR=9000) "X"
1010 A$=FID(1,ERR=9000)
1020 ON 1+DEC(A$(1,1)) GOTO 1800,1100,1200,1300,1400,
1020:1500,1600,1700,1800
1100 PRINT "indexed"; END
1200 PRINT "serial"; END
1300 PRINT "keyed (direct or sort)"; END
1400 PRINT "string"; END
1500 PRINT "program"; END
1600 PRINT "directory"; END
1700 PRINT "multi-keyed"; END
1800 PRINT "invalid type"; END
9000 PRINT "Error ",ERR
9010 END
```

C Example

```
#include "fslib.h"

#define FSLEVEL 1
main()
{
    FID *f;
    ChanType *chan1;
```

C Example

```

    if (fs_init(FSLEVEL)) exit(-1);

    if ( (chan1 = fs_open ("X", 0))
        == (ChanType *)0 ||
        fs_fid (chan1) != 0) {
        printf ("Error %d\n", fserr);
        exit(0);
    }
    f = (FID *) (iop.s1);
    switch (f->type) {
    case T_INDEX:
        printf ("indexed\n");
        break;
    case T_SERIAL:
        printf ("serial\n");
        break;
    case T_KEYED:
        printf ("keyed (direct or sort)\n");
        break;
    case T_STRING:
        printf ("string\n");
        break;
    case T_PROGRAM:
        printf ("program\n");
        break;
    case T_DIRECT:
        printf ("directory\n");
        break;
    case T_BTREE:
        printf ("multi-keyed\n");
        break;
    default:
        printf ("invalid type\n");
        break;
    }
    fs_close(chan1);
    fs_shut ();
}

```

fs_flush (Flush Cached Files)

Syntax

`fs_flush()`

Description

This function is not yet implemented. It is intended to flush cached files in future releases.

fs_info (Fetch Information about a File/Device)

Syntax

`fs_info (channel, function)`

Syntax Item	Description
<code>ChanType *channel</code>	Pointer to an open channel.
<code>int function</code>	One of the following (from <code>fslib.h</code>):
	<code>FI_IND</code> Return current index in <code>iop.index</code>
	<code>FI_HEADER</code> Return FIN pointer in <code>iop.s1</code>

Description

This function returns information about the file/device on an open channel. Currently the FIN data and the file index are the only information this function retrieves.

The file index is the current position in the file. For a record-oriented file, this is the record number. For a byte-oriented file, it is the byte offset.

Returned

Returned Item	Description
<code>fserr</code>	Global error code variable.
<code>iop.index</code>	Current index value (<code>function = FI_IND</code>).
<code>iop.s1</code>	Pointer to FIN (<code>function = FI_HEADER</code>).

Notes

The pointer returned points to the only copy of the FIN structure that exists. The programmer is advised to take precautions not to corrupt it.

See also

`fs_fid()`, `fs_open()`

Examples

The following examples retrieve the current index of an open file:

BB^x Example

```
1000 I=IND(1)
```

C Example

```
ChanType *chan1;
long i;
...
fs_info(chan1, FI_IND);
i = iop.index;
```

The following examples obtain the FIN structure of an open MKEYED file:

BB^x Example

```
1000 A$=FIN(1)
```

C Example

```
ChanType *chan1;  
char *finptr;  
...  
fs_info(chan1, FI_HEADER);  
finptr = iop.s1;
```

fs_init (Initialize File System)

Syntax

```
fs_init (level)
```

Syntax Item	Description
int level	The C Library level number. Use the value 1.

Description

This function sets up system parameters to prepare the file system for use. The `fs_conf()` function is the only call to the C Library that should precede `fs_init()`. The `fs_init()` function should eventually be followed by `fs_shut()` when processing is complete.

The only parameter for this function is the level number, which is used to ensure compatibility with future releases.

Returned

Returned Item	Description
fserr	Global error code variable.

Notes

If this call returns an error condition, then no additional C Library calls should be attempted, including `fs_shut()`.

See also

`fs_conf()`, `fs_shut()`

Examples

BB* Example

```
1000 OPEN (1,ERR=9000) "MYFILE.DAT"
1010 PRINT "Ok"
1020 REM Code to process file goes here.
...
2000 CLOSE (1,ERR=9000)
2010 PRINT "Done"
2020 END
...
9000 PRINT "Error",ERR
9010 END
```

C Example

```
#include "fslib.h"

#define FSLEVEL 1
main()
{
    ChanType *chan1;
    if (fs_init(FSLEVEL) != 0) {
        printf ("Can't start filesystem\n");
        exit (-1);
    }
}
```

C Example

```
    if ( (chan1 = fs_open ("MYFILE.DAT", 0))
        == (ChanType *) 0) {
        printf ("Error %d\n", fserr);
        exit (-1);
    }
    printf ("Ok\n");

    /* Code to process file goes here. */

    if (fs_close(chan1) != 0) {
        printf ("Error %d\n", fserr);
        exit (-1);
    }
    printf ("Done\n");
    fs_shut();
}
```

fs_initfile (Erase and then Reinitialize a File)

Syntax

`fs_initfile (name)`

Syntax Item	Description
<code>char *name</code>	The name of the file to be erased and reinitialized.

Description

Erases the named file and then reinitializes that file. The operation will fail if the file is currently open.

Returned

Returned Item	Description
<code>fserr</code>	Global error code variable.

Examples

BB^x Example

```
...
1000 INITFILE "MYFILE",ERR=2000
...
2000 PRINT "Error", ERR
2010
```

C Example

```
...
if (fs_initfile("MYFILE") != 0) {
    printf ("Error %d\n", fserr);
    exit (-1);
}
...
```

fs_key (Get Key from File)

Syntax

`fs_key (channel, function)`

Syntax Item	Description
<code>ChanType *channel</code>	Channel pointer from a previous <code>fs_open()</code> function.
<code>int function</code>	One of the following (from <code>fslib.h</code>):
	<code>FI_KEYF</code> Return first key
	<code>FI_KEYP</code> Return previous key
	<code>FI_KEY</code> Return current key
	<code>FI_KEYN</code> Return next key
	<code>FI_KEYL</code> Return last key

Description

This function is used to retrieve a key from an open file.

Returned

Returned Item	Description
<code>fserr</code>	Global error code variable.
<code>iop.s1</code>	Key data (can contain embedded nulls).
<code>iop.s1len</code>	Length of key data.

See also

`fs_fid()`, `fs_info()`, `fs_remove()`

Examples

The following examples retrieve the current key from an open file:

```

BBx Example
1000 K$=KEY(1,ERR=9000)
1010 PRINT "Key:",K$
1020 PRINT "Key Length:",LEN(K$)
...
9000 PRINT "Error",ERR
9010 END

```

C Example

```

ChanType *chan1; /* previously opened */
...
if (fs_key(chan1, FI_KEY) != 0) {
    printf ("Error %d\n", fserr);
    exit (-1);
}
printf ("Key: %s\n", iop.s1);
printf ("Key Length: %d\n", iop.s1len);
/*NOT strlen(iop.s1), as key can contain embedded nulls */

```

fs_lock (Lock Opened File)

Syntax

```
fs_lock (channel)
```

Syntax Item	Description
ChanType *channel	Pointer to a previously opened channel.

Description

This function prevents all other write access to an open file. If advisory locking is in use, read access is still permitted on locked files.

Returned

Returned Item	Description
fserr	Global error code variable.

Notes

The `fs_lock()` function is required before writing to a SERIAL file.

See also

`fs_close()`, `fs_unlock()`

Examples

The following examples lock a previously opened file:

BB* Example

```
...
1000 LOCK (1,ERR=9000)
...
9000 PRINT "Error",ERR
9010 END
```

C Example

```
ChanType *chan1;

...
if (fs_lock (chan1) != 0) {
    printf ("Error %d\n", fserr);
    exit (-1);
}
```

fs_mkeyfin (Generate FIN for MKEYED File)

Syntax

```
fs_mkeyfin (fin, keydef1 {, keydef2...}, (char *) 0)
```

Syntax Item	Description
<code>char *fin</code>	Pointer to a 385-character block for FIN storage.
<code>char *keydef</code>	Key definition for an MKEYED file.
	Each <i>keydef</i> has the following syntax: <pre>segment{+ segment...}</pre> Each <i>segment</i> has the following syntax: <pre>[{ field:} offset:len{ :flags}]</pre> <pre>field</pre> Record field number (0 to 255) <pre>offset</pre> Offset into record or field (1 to 1024) <pre>len</pre> Length of key segment (1 to 120) <pre>flags</pre> Key flags, specify none or more of: <pre>U</pre> Unique key <pre>D</pre> Descending key <pre>B</pre> Business math type numeric key
<code>(char *) 0</code>	The <i>keydef</i> list <u>must</u> end with <code>(char *) 0</code>

Description

This function assembles a FIN structure for an MKEYED file, given a BB^x-style textual key specification. It calls `fs_clrfin()` once, then `fs_addkey()` repeatedly for each *keydef* argument.

Each of the *keydef* arguments is parsed into the FIN buffer with the appropriate key number. Bracketed expressions separated by a plus (+) sign, `segment{+ segment...}` indicate a composite key made up of several parts of the record. Up to 48 key segments can be defined for each MKEYED file. The keys will be numbered in the sequence in which they are defined.

Notes

The list of *keydef* arguments must end with `(char *) 0`.

In a key segment, field number zero specifies the whole record with no field parsing.

See also

`fs_addkey()`, `fs_clrfin()`, `fs_create()`, `fs_fin()`, `fs_info()`

Examples

The following examples create an MKEYED file:

BB ^x Example
1000 MKEYED"MK.DAT", [3:1:5]+[1:1:1], 100, 50, ERR=9000
1010 PRINT "Ok"
1020 END
9000 PRINT "Error ", ERR
9010 END

C Example

```
#include "fslib.h"

#define FSLEVEL 1
main()
{
    FID      *fid;
    char  fin[385];

    if (fs_init(FSLEVEL) != 0) exit (-1);

    fs_mkeyfin (fin, "[3:1:5]+[1:1:1]",
               (char *) 0);
    fid = fs_mkfid (T_BTREE, 0, 100, 50,
                  "MK.DAT");
    fs_create (fid, fin);
    free (fid);
    if (fserr != 0)
        printf ("Error %d\n", fserr);
    else
        printf ("Ok\n");
    fs_shut ();
}
```

fs_mkfid (Build FID Structure from Components)

Syntax

FID *fs_mkfid (*type, keysiz, records, recsiz, name*)

Syntax Item	Description
char <i>type</i>	One of the following from <code>fslib.h</code> :
	T_INDEX INDEXED file type T_SERIAL SERIAL file type T_KEYED Keyed file (DIRECT or SORT type) T_STRING STRING file type T_PROGRAM PROGRAM file type T_DIRECT DIRECTORY file type T_BTREE MKEYED file type
char <i>keysiz</i>	Specify for DIRECT and SORT file types only. Enter 0 for all others.
unsigned long <i>records</i>	Number of records in the file. Enter 0 for a dynamic file with no limit.
unsigned short <i>recsiz</i>	Number of bytes per record. Enter 0 if not fixed (SERIAL file type), or actually zero (SORT file).
char * <i>name</i>	Filename.

Description

This function allocates and initializes a FID structure from parameters supplied by the caller. The returned pointer points to a FID object (FID *), immediately followed in memory by the null-terminated name. (See `fslib.h` for details.)

Returned

Returned Item	Description
	Pointer to newly created FID structure.
<code>fserr</code>	Global error code variable.

Notes

It is the caller's responsibility to free the storage when it is no longer needed. Programmers are advised, however, not to free FIDs they did not create.

See also

`fs_create()`

Examples

The following examples create an MKEYED file:

BB* Example
1000 MKEYED "MK.DAT", [3:1:5]+[1:1:1], 100, 50, ERR=9000
1010 PRINT "Ok"
1020 END
9000 PRINT "Error ", ERR
9010 END

C Example

```
#include "fslib.h"

#define FSLEVEL 1
main()
{
    FID *fid;
    char fin[385];

    if (fs_init(FSLEVEL) != 0) exit (-1);

    fs_mkeyfin (fin, "[3:1:5]+[1:1:1]",
               (char *)0);
    fid = fs_mkfid (T_BTREE, 0, 100, 50,
                  "MK.DAT");
    fs_create (fid, fin);
    free (fid);
    if (fserr != 0)
        printf ("Error %d\n", fserr);
    else
        printf ("Ok\n");
    fs_shut();
}
```

fs_open (Open File/Device/Pipe)

Syntax

```
ChanType *fs_open (name, flags {, arglist...})
```

Syntax Item	Description
char *name	The name of the file or device to be opened.
int flags	<p>Flags indicating file access mode. The C Library recognizes a bitwise OR of the following (from fslib.h). For example: to specify both the FMODEQ flag and the FPTHEQ flag, the flags value would be (FMODEQ FPTHEQ).</p> <p>FISZEQ If this flag is set, the file will be processed as a STRING file, regardless of its type.</p> <p>FMODEQ This flag enables a special file mode (implementation-dependent). Fetches char *mode from the arglist.</p> <p>FPTHEQ Normally, the primary key (key #1) of an MKEYED file is used. This flag allows selection of an alternate key chain (key #2 or greater). Fetches int keynum from the arglist.</p>
arglist	<p>One or more of the following parameters, depending upon which flags are set.</p> <p>char *mode This value is implementation dependent. Supplemental documentation will be provided as required.</p> <p>int keynum Number (2 or greater) of the alternate path selected.</p>

Description

This function opens a channel to a device or file, and must precede most other I/O operations. Note that **fs_open()** returns a pointer, but the error condition is still placed in the **fserr** variable.

Returned

Returned Item	Description
fs_open()	Returns a channel pointer if successful or (ChanType *) 0 if unsuccessful.
fserr	Global error code variable.
iop.funk	<p>Channel type codes. One of the following (from fslib.h):</p> <p>FSA_CHR Character dev/file</p> <p>FSA_DFILE Disk file</p> <p>FSA_CFILE Character disk file</p>
iop.index	Record size, if available.

Notes

The **arglist** parameters are always retrieved in the sequence listed above. It is the programmer's responsibility to enter **arglist** parameters in the sequence shown so that the correct parameter is retrieved for each flag that is set.

See also

fs_close()

Examples

BB^x Example

```

1000 OPEN (1,ERR=9000) "MYFILE.DAT"
1010 PRINT "Ok"
1020 REM Code to process file goes here.
...
2000 CLOSE (1,ERR=9000)
2010 PRINT "Done"
2020 END
...
9000 PRINT "Error",ERR
9010 END

```

C Example

```

#include "fslib.h"

#define FSLEVEL 1
main()
{
    ChanType *chan1;
    if (fs_init(FSLEVEL) != 0) {
        printf ("Can't start filesystem\n");
        exit (-1);
    }

    if ( (chan1 = fs_open ("MYFILE.DAT", 0))
        == (ChanType *) 0) {
        printf ("Error %d\n", fserr);
        exit (-1);
    }
    printf ("Ok\n");

    /* Code to process file goes here. */

    if (fs_close(chan1) != 0) {
        printf ("Error %d\n", fserr);
        exit (-1);
    }
    printf ("Done\n");
    fs_shut();
}

```

fs_read (Read File or Device)

Syntax

`fs_read(channel, buffer, length, flags, {, arglist...})`

Syntax Item	Description
<code>ChanType *channel</code>	Channel returned by <code>fs_open()</code> .
<code>char *buffer</code>	Read buffer allocated by programmer.
<code>unsigned int length</code>	Buffer length.
<code>int flags</code>	<p>Any ORed combination of the following (from <code>fslib.h</code>):</p> <p>FEXTRACT Reads and then locks the record. The file pointer is positioned to rewrite the record with the next <code>fs_write()</code> function.</p> <p>FFIELDS Set this flag to enable field parsing for byte-oriented files only. If the flag is on, the <code>fs_read()</code> will stop upon detecting a valid field terminator, rather than attempting to read the full number of bytes requested. Valid field terminators include the following: 0x00, 0x0A (linefeed), 0x0D (carriage return), 0x1C, 0x1D, 0x1E and 0x1F.</p> <p>FLENEQ If field parsing is enabled (FFIELDS flag is on), and the buffer length is reached before encountering a valid field terminator, this flag causes the file pointer to be advanced to the beginning of the next field. Without this flag, the next <code>fs_read()</code> would continue with the current field.</p> <p>FKEYEQ Set this flag to locate a record by key. The <code>fs_read()</code> function fetches <code>char *key</code> (the key may have embedded nulls), and <code>int keylen</code> (the length of the key being sought) from the <code>arglist</code>.</p> <p>FFIND This flag is used only in conjunction with the FKEYEQ flag. Normally, a failed attempt to locate a record by key has the side-effect of moving the file pointer. If this flag is set, <code>fs_read()</code> will move the file pointer only if the key is found.</p> <p>FINDEQ If this flag is set, the <code>fs_read()</code> function locates data to be read by record number (for record-oriented files) or by byte number (for byte-oriented files). It fetches <code>long index</code> from the <code>arglist</code>.</p> <p>FTIMEQ The <code>fs_read()</code> function times out if the request is unsatisfied during the timeout period. If this flag is set, the <code>fs_read()</code> function updates the timeout period to the number of seconds in <code>int timeout</code> from the <code>arglist</code>. The default period is 10 seconds.</p> <p>FTIMOK If this flag is set, an <code>fs_read()</code> function that times out before completion does not cause an error condition to be reported.</p> <p>FPTHEQ Set this flag to select an alternate key chain (key path) when reading an MKEYED file. The <code>fs_read()</code> function fetches <code>int key</code> (the number of the key path selected) from the <code>arglist</code>.</p> <p>FDIREQ Causes the file pointer to be advanced increment records or bytes after</p>

Syntax Item	Description
	the read is completed. The increment may be negative. int increment is fetched from the arglist .
arglist	<p>One or more of the following parameters, depending upon which flags are set.</p> <p>char *key The key value sought (may contain embedded nulls).</p> <p>int keylen Length of the key being sought.</p> <p>long index Record number for record-oriented files, or byte number for byte-oriented files, to which the file pointer is repositioned before reading.</p> <p>int timeout Number of seconds to be established as the timeout period for this fs_read() and any subsequent timed I/O operations.</p> <p>int key The number of the key path selected.</p> <p>int increment The amount to move the file pointer after reading.</p>

For example, to select the FKEYEQ, FFIND, and FTIMEQ options, **flags** would be specified as **(FKEYEQ|FFIND|FTIMEQ)**, and the arguments **char *key**, **int keylen**, **int timeout** would be fetched from the **arglist**, in that sequence.

Description

This function initiates a read operation from the file or device associated with the specified **channel** into the **buffer**. When reading record-oriented files, **fs_read()** attempts to read exactly one record, the next sequential record by default. Non-sequential read operations from record-oriented files can be performed by setting the FINDEQ flag or the FKEYEQ flag and specifying the file pointer offset, or the key and key length, respectively, in the **arglist**.

The **fs_read()** function has the side-effect of unlocking any previously locked records on the specified channel. Such records would have been locked by a previous **fs_read()** with the FEXTRACT flag set. If there were no subsequent I/O operations on that channel, the "extracted" records would have remained locked. Note that **fs_read()** does not unlock files that have been locked by the **fs_lock()** function.

When reading byte-oriented files (STRING and PROGRAM files), **fs_read()** attempts to read the number of bytes specified by **length**, or one field if the FFIELDS flag is set. As with record-oriented files, non-sequential read operations from byte-oriented files can be executed through use of the FINDEQ flag.

The **fs_read()** function requires that the buffer be large enough to contain all of the data requested. If it is not, an error condition will result. Additionally, error conditions will be generated by any of the following: attempting to read past the end of a file, or attempting to locate a nonexistent key, record number, or byte number in a file.

Returned

Returned Item	Description
fserr	Global error code variable.
iop.sllen	Length of data actually read.

Notes

The *arglist* parameters are always retrieved in the sequence listed above. It is the programmer's responsibility to enter *arglist* parameters in the sequence shown so that the correct parameter is retrieved for each flag that is set.

DIRECTORY files require a minimum buffer size of 256 bytes, or if MAXNAMLEN is defined on your system then the buffer size should be MAXNAMLEN + 1.

See also

`fs_key()`, `fs_open()`, `fs_remove()`, `fs_write()`

Examples

The following examples extract a record by key and display it:

BB^x Example

```
1000 EXTRACT (1, key="12345") A$
1010 PRINT "Record data: ", A$
1020 PRINT "Record length:", LEN(A$)
```

C Example

```
ChanType *chan1;
char      buf[100];
...
fs_read (chan1, buf, 100, FEXTRACT|FKEYEQ,
        "12345", 5);
printf ("Record data: %s\n", buf);
printf ("Record length: %d\n", iop.sllen);
/* NOT strlen(buf), as data may contain
   nulls*/
```

The following examples read the next record:

BB^x Example

```
1000 READ RECORD (1) A$
```

C Example

```
ChanType *chan1;
char      buf[100];
...
fs_read (chan1, buf, 100, 0);
```

fs_remove (Remove Record from Keyed File)

Syntax

```
fs_remove (channel, key, keylen)
```

Syntax Item	Description
ChanType *channel	The open channel to the file containing the record to be removed.
char *key	The key value sought (may contain embedded nulls).
int keylen	Length of key argument.

Description

This function removes a key and its associated record from a keyed file.

Returned

Returned Item	Description
fserr	Global error code variable.

See also

```
fs_key(), fs_read(), fs_write()
```

Examples

The following examples remove a key and its associated record from a data file:

BB* Example

```
...
1000 REMOVE (1,KEY="12345",ERR=9000)
...
9000 PRINT "Error",ERR
9010 END
```

C Example

```
ChanType *chan1;

...
if (fs_remove (chan1, "12345", 5) != 0) {
    printf ("Error %d\n", fserr);
    exit (-1);
}
```

fs_rename (Rename File)

Syntax

```
fs_rename (oldname, newname)
```

Syntax Item	Description
char *oldname	Filename to be changed.
char *newname	New filename.

Description

This function changes the name of an existing file.

Returned

Returned Item	Description
fserr	Global error code variable.

Notes

The file to be renamed cannot be open.

Examples

BB^x Example

```
1000 RENAME "OLDFILE", "NEWFILE", ERR=9000
...
9000 PRINT "Error", ERR
9010 END
```

C Example

```
if (fs_rename ("OLDFILE", "NEWFILE") != 0) {
    printf ("Error %d\n", fserr);
    exit (-1);
}
```

fs_shut (Shut Down File System)

Syntax

```
fs_shut ()
```

Description

This function should be called when all file system processing is complete. It closes all open channels and files, and frees all system resources reserved by `fs_init ()`.

Returned

Nothing.

See also

`fs_conf()`, `fs_init()`

Examples

BB^x Example

```
1000 OPEN (1,ERR=9000) "MYFILE.DAT"
1010 PRINT "Ok"
1020 REM Code to process file goes here.
...
2000 CLOSE (1,ERR=9000)
2010 PRINT "Done"
2020 END
...
9000 PRINT "Error",ERR
9010 END
```

C Example

```
#include "fslib.h"

#define FSLEVEL 1
main()
{
    ChanType *chan1;
    if (fs_init(FSLEVEL) != 0) {
        printf ("Can't start filesystem\n");
        exit (-1);
    }

    if ( (chan1 = fs_open ("MYFILE.DAT", 0))
        == (ChanType *) 0) {
        printf ("Error %d\n", fserr);
        exit (-1);
    }
    printf ("Ok\n");

    /* Code to process file goes here. */

    if (fs_close(chan1) != 0) {
        printf ("Error %d\n", fserr);
        exit (-1);
    }
}
```

C Example

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

fs_unlock (Unlock File)

Syntax

```
fs_unlock (channel)
```

Syntax Item	Description
ChanType *channel	Pointer to the open channel to be unlocked.

Description

This function unlocks a file previously locked by the `fs_lock()` function.

Returned

Returned Item	Description
fserr	Global error code variable.

See also

`fs_close()`, `fs_lock()`

Examples

BB^x Example

```
...
1000 UNLOCK (1,ERR=9000)
...
9000 PRINT "Error",ERR
9010 END
```

C Example

```
ChanType *chan1;

...
if (fs_unlock (chan1) != 0) {
    printf ("Error %d\n", fserr);
    exit (-1);
}
```

fs_write (Write to File or Device)

Syntax

`fs_write (channel, buffer, length, flags{, arglist...})`

Syntax Item	Description
<code>ChanType *channel</code>	Open channel associated with file or device where data is to be written.
<code>char *buffer</code>	Pointer to data to be written.
<code>unsigned int length</code>	Length of data to be written.
<code>int flags</code>	<p>Any ORed combination of the following (from <code>fslib.h</code>):</p> <p>FDOMEQ Normally, writing a record with a key that already exists in the file will simply overwrite the previous record. With this flag set, an attempt to overwrite an existing key generates an error condition.</p> <p>FKEYEQ Set this flag to write a record by key. The <code>fs_write()</code> function fetches <code>char *key</code> (the key may have embedded nulls), and <code>int keylen</code> (the length of the key being sought) from the <code>arglist</code>.</p> <p>FINDEQ If this flag is set, the <code>fs_write()</code> function repositions the file pointer to the record number (for record-oriented files) or the byte number (for byte-oriented files) before writing. It fetches <code>long index</code> from the <code>arglist</code>.</p> <p>FTIMEQ The <code>fs_write()</code> function times out if the request is unsatisfied in the timeout period. If this flag is set, the <code>fs_write()</code> function updates the timeout period to the number of seconds in <code>int timeout</code> from the <code>arglist</code>. The default timeout period is 10 seconds.</p> <p>FDIREQ Causes the file pointer to be advanced <code>increment</code> records or bytes after the write is completed. The increment may be negative. Fetches <code>int increment</code> from the <code>arglist</code>.</p>
<code>arglist</code>	<p>One or more of the following parameters, depending upon which <code>flags</code> are set.</p> <p>char *key The key value sought (may contain embedded nulls).</p> <p>int keylen Length of the key being sought.</p> <p>long index Record number for record-oriented files, or byte number for byte-oriented files, to which the file pointer is repositioned before writing.</p> <p>int timeout Number of seconds to be established as the timeout period for this <code>fs_write()</code> and all subsequent timed I/O operations.</p> <p>int increment Amount to move the file pointer after writing.</p>

For example, to select the FKEYEQ, FTIMEQ, AND FDOMEQ options, `flags` would be specified as `(FKEYEQ|FTIMEQ|FDOMEQ)`, and the arguments `char *key`, `int keylen`, and `int timeout` would be fetched from the `arglist` in that sequence.

Description

This function initiates a write operation from the *buffer* to the file or device associated with the specified open *channel*. When writing record-oriented files, `fs_write()` attempts to write exactly one record, the next sequential record by default. If the length specified is greater than the file's fixed record size, an error condition is returned; however, if the *length* is shorter, the record is written successfully, and the remainder of the record is filled in with nulls (0x00). Non-sequential write operations to record-oriented files can be performed by setting the FINDEQ flag or the FKEYEQ flag and specifying the file pointer offset, or the key and key length, respectively, in the *arglist*.

The `fs_write()` function has the side-effect of unlocking any previously locked (extracted) records on the specified *channel*. Note that `fs_write()` does not unlock files that have been locked by the `fs_lock()` function.

When writing byte-oriented files (STRING and PROGRAM files), `fs_write()` attempts to write the number of bytes specified by length. As with record-oriented files, non-sequential write operations to byte-oriented files can be executed by use of the FINDEQ flag and the *arglist*.

In all cases, the file or device must have sufficient space to accommodate the new data being written. If not, an error condition will be reported. It is strongly recommended that programmers arrange to lock either the record (see `fs_read()`) or the entire file (see `fs_lock()`) before writing. This will ensure that processes competing for use of the file will not interfere with each other.

In addition, the following special considerations apply to `fs_write()`:

- DIRECTORY files can never be written, only read.
- Writing more than 32767 bytes in one function is never permissible.
- SERIAL files cannot be written unless the file is locked (see `fs_lock()`).
- SERIAL files are always truncated to the record number most recently written.
- SERIAL files have no fixed record size. Therefore, the number of bytes specified by length is always written, with no padding.

Returned

Returned Item	Description
<code>fserr</code>	Global error code variable.

Notes

The *arglist* parameters are always retrieved in the sequence listed above. It is the programmer's responsibility to enter *arglist* parameters in the sequence shown so that the correct parameter is retrieved for each flag that is set.

See also

`fs_read()`, `fs_remove()`

Examples

The following examples write records by index number:

BB ^x Example
1000 WRITE RECORD (1, IND=30, ERR=9000) A\$
...
9000 PRINT "Error", ERR
9010 END

C Example

```
ChanType *chan1;
char      buf[100];
...
if (fs_write (chan1, buf, 100, FINDEQ, 30)
    != 0) {
    printf ("Error %d\n", fserr);
    exit (-1);
}
```

Configuration File

What is the Configuration File?

The configuration file used with the C Library is an ASCII text file that contains detailed system information that can be passed to C Library programs. The configuration file enables users to provide system-unique limits for the C Library. Although the C Library can automatically set up certain minimum configuration defaults, you can optimize performance by creating a configuration file to set system information such as number of devices, disk files, and I/O channels that can be accessed simultaneously.

Each host system sets different limits on the number of devices that can be configured, as well as the maximum number of files that can be accessed simultaneously. The following system limit considerations apply:

- If the limits are set too low, programmers cannot take full advantage of the host system.
- If the limits are set too high, memory space is wasted.
- The C Library limits can never exceed those imposed by the host system.

Any standard text editor can be used to maintain the configuration file. Each line must conform to those defined in the following sections or they will be ignored.

Configuration File Components

The configuration file contains two separate components:

- The Limit Configuration Component defines limits, such as number of devices, disk files, I/O channels, etc., that are placed upon the system. Remember, the C Library limits can never exceed those imposed by the host system.
- The Functions Configuration Component defines operations functions, such as file caching, file path initialization, etc.

Limit Configuration Component

The configuration file can include some or all of the following Limit Configuration Component options:

```
DEVS=int
FCBS= int
CIBS= int
HANDLES= int
TIMEOUT= int
```

Option	Description	Default
DEVS	Total number of devices that can be accessed simultaneously by a BB ^x process	4
FCBS	Total number of disk files that can be simultaneously accessed by a BB ^x process. (This is not the total for the system, which may also impose limits for this parameter.)	10
CIBS	Total number of I/O channels that can be accessed simultaneously by a BB ^x process. (This is not the total for the system.) BB ^x associates an I/O channel with either a file or device. There is not necessarily a one-to-one relationship between CIBS and DEVS or between CIBS and FCBS. For example, 2 channels opened to 2 different devices will require 2 CIBS and 2 DEVS; however, 2 channels opened to the same device will require 2 CIBS but only 1 DEV.	16
HANDLES	Maximum number of file "handles" to be used by each BB ^x process. (This is not	OS Limit

Option	Description	Default
	the total for the system.) Entering a large number here will cause BB ^x to retain a large number of open files (FCBCACHE enabled), or allow the user to open a large number of files (FCBCACHE not enabled).	
TIMEOUT	Number of seconds BB ^x will wait for a busy record or file before issuing an error. The range is 4 to 255 seconds.	10

Function Configuration Component

The configuration file can include some or all of the following Function Configuration Component options:

FCBCACHE

PREFIX *path*

SETOPTS *hexstring*

Option	Description																		
FCBCACHE	Indicates that BB ^x is to maintain files opened and closed by the user open at the system level in the hopes that the file will be opened again at a later time. If the file is kept open by BB ^x , the later open will take very little time.																		
PREFIX	Initializes the file search path. Similar to the UNIX/XENIX PATH value. The directories in the PREFIX are space separated. For example: PREFIX /usr/source/ms/ /bbx/util/ Unlike the PREFIX command in the BB ^x language, the PREFIX configuration option does NOT put quote characters around the list of directories.																		
SETOPTS	Sets special-purpose features of the BB ^x file system. The default is all options off. The SETOPTS configuration line consists of the keyword SETOPTS followed by a series of hexadecimal digits (Ex. SETOPTS 001). The individual bits represented by each hexadecimal digit have the following meanings: <table border="1"> <thead> <tr> <th>Digit</th> <th>Bit Value</th> <th>Result when Bit is Set</th> </tr> </thead> <tbody> <tr> <td>3</td> <td>1</td> <td>The BB^x file system will attempt to force MS-DOS to update the length of an MKEYED file after any growth in the file. This will slow down MKEYED writes. (For MS-DOS only.)</td> </tr> <tr> <td>5</td> <td>4</td> <td>Enables advisory locking. Read operations are permitted, even on locked data. Normally, locked files or records are not available for other access.</td> </tr> <tr> <td>5</td> <td>2</td> <td>Permits access to the key region of otherwise locked files or records.</td> </tr> <tr> <td>5</td> <td>1</td> <td>Improved file system performance if the operating system supports only advisory locking.</td> </tr> <tr> <td>7</td> <td>2</td> <td>Network access bit. Permits access to remote files via a PRO/5 Data Server.</td> </tr> </tbody> </table> For example, to permit access to the key region of otherwise locked data, but leave all of the other options listed above off, the line would be: SETOPTS 00002	Digit	Bit Value	Result when Bit is Set	3	1	The BB ^x file system will attempt to force MS-DOS to update the length of an MKEYED file after any growth in the file. This will slow down MKEYED writes. (For MS-DOS only.)	5	4	Enables advisory locking. Read operations are permitted, even on locked data. Normally, locked files or records are not available for other access.	5	2	Permits access to the key region of otherwise locked files or records.	5	1	Improved file system performance if the operating system supports only advisory locking.	7	2	Network access bit. Permits access to remote files via a PRO/5 Data Server.
Digit	Bit Value	Result when Bit is Set																	
3	1	The BB ^x file system will attempt to force MS-DOS to update the length of an MKEYED file after any growth in the file. This will slow down MKEYED writes. (For MS-DOS only.)																	
5	4	Enables advisory locking. Read operations are permitted, even on locked data. Normally, locked files or records are not available for other access.																	
5	2	Permits access to the key region of otherwise locked files or records.																	
5	1	Improved file system performance if the operating system supports only advisory locking.																	
7	2	Network access bit. Permits access to remote files via a PRO/5 Data Server.																	

Option	Description
	To accept the default condition (none of these features are enabled), omit the SETOPTS line from the configuration file.

Sample Configuration File

The following illustrates a sample configuration file:

```
cibs=20
devs=5
fcbs=20
PREFIX /bbx/util/ /usr/SOURCE/MS/
```

Configuration File Processing

Although a configuration file is normally named **config.bbx**, and contained in the current directory, it can be given a different name and/or contained in another directory. The following considerations apply:

- If the configuration file is named **config.bbx** and is located in the current directory, send a call to **fs_init()**.
- If the configuration file is NOT named **config.bbx**, and/or is NOT contained in the current directory, first send a call to **fs_conf()**, then send a call to **fs_init()**.

For both initialization options, the **fs_init()** function allocates required space, sets up configuration information and performs other system initialization procedures prior to use. This function needs to be performed only once during a C Library session.

The **fs_init()** function performs the following steps to locate the configuration file:

- If **fs_conf("-cname")** is called, **fs_init()** will try only **name**.
- If an environment variable **BBCONFIG=name** exists, **fs_init()** will try only **name**.
- If neither of the above conditions is met, the **fs_init()** function will attempt to use the file "**config.bbx**" in the current directory, and then in the **/usr/bbx** directory.
- If the configuration file is not found, the C Library sets the minimum defaults.

C Library Error Codes

Overview

This section lists the C Library error codes in alphabetical order. A description of each error is given along with some possible causes. The error codes are also defined in the `fserr.h` file, which is supplied with the C Library.

Error codes returned by the C Library are one number higher than those returned by BB^x. This difference stems from error code zero. In BB^x, error code zero indicates an error, whereas in C, error code zero represents successful completion of the function.

E_BADARG (String/Number Mismatch)

- An invalid parameter was passed to a function.
- A string was used where a number was needed.
- A number was used where a string was needed.

E_BADCIB (Improper File/Device Usage)

- Opening a device that is already in use by another user.
- Opening a channel that is already open.
- Attempting input or output on a channel that is not open.
- Attempting to unlock a file that hasn't been locked or which has been opened with FISZEQ.
- Closing a channel that is not open.
- Locking a file that was opened by a different user.
- Locking a file that was already locked by the same user.
- Erasing a file you do not own.

E_BADHEAD (Open of File With Invalid Header)

Attempting to open a file with an invalid header.

E_BREAK (Escape)

The user interrupted the operation by typing an ESCAPE or INTERRUPT character.

E_BUSY (File, Record, or Device Busy or Inaccessible)

The C Library issues this error when a data transfer or control operation is not yet possible because of the current state of the file or device. Usually, the function call keeps trying for several seconds before reporting the error. The amount of wait time may be controlled by the FTIMEQ flag.

- Trying to access a device, such as a printer, that is not turned on or not online.
- Trying to access a record that has been locked.
- Trying to access a file that has been locked.
- Trying to lock a file that is being used.

E_CANT (Improper File/Device Address)

Occurs when an attempt is made to perform an operation on a file that is not meaningful for that type of file. This error can also occur if the file is not in the necessary state (i.e., does not have correct operating system permissions).

- Reading from a write-only device, such as a printer.
- Writing to a read-only device.
- Writing to a SERIAL file without locking it.
- Attempting to access keys in a non-keyed file.

MKEYED errors:

- Read from an MKEYED file with the FINDEQ flag set.
- Write to an MKEYED file with the FINDEQ flag set.
- Write to a single-key MKEYED file with the FPTHEQ flag set.

E_DFAIL (Disk Not Ready)

- The disk is not online.
- A floppy disk drive is empty or the door is not closed.
- Trying to write to a disk that is write-protected.

E_DISK (Invalid Disk Directory)

Attempting to access a disk that is not properly initialized. To correct, initialize or format the disk according to your host system specifications.

E_DREAD (Disk Read/Write Error)

This error usually means that a disk is damaged or the drive is not correctly aligned.

E_EOB (End of Buffer Overflow)

An operation was attempted that could not be done with the available memory, such as attempting to read or write a large record.

E_EOF (End of File)

This error occurs when an attempt is made to move the file pointer beyond the end of the file.

- Trying to read beyond the end of a file.
- Trying to read a SERIAL file immediately after writing to it.
- Adding a new key to a full keyed file.

E_EOR (End of Record)

This error occurs when an attempt is made to:

- Read more data than is contained in a record.
- Write more data than a record can hold.
- Write a record more than 32767 bytes long.

E_INTERN (General I/O Error)

The C Library issues this error when an I/O operation fails and there is no error code defined for that type of error.

E_INVINT (Invalid Integer)

An integer out of the legal range was passed to a function.

E_INVKLEN (Invalid Key Length)

The length of the key is invalid.

E_INVPARAM (Invalid Parameter)

- Attempting to reference a disk that is not a valid disk.
- Call to fs_mkeyfin() or fs_addkey() with invalid parameters.

MKEYED errors:

- Invalid key table string encountered. This could be the result of a corrupted MKEYED file header.
- FPTHEQ flag is attempting to select a key number that does not exist.

E_KFAIL (Missing or Duplicate Key)

- Attempting to access a record in a keyed file using the FKEYEQ flag when there is no such key in the file.
- Attempting to write a key using the FKEYEQ flag and the FDOMEQ flag when that key already exists.

E_MEM (System Memory Overflow)

Not enough memory exists outside the user workspace to perform a certain operation.

- Attempting to open devices or disk files when insufficient memory exists to allocate buffers and information structures.

E_NOF (Missing or Duplicate File)

- Attempting to access a file that cannot be found, either because it has not been previously defined, or because the disk containing the file is not on-line.
- Attempting to create a new file with the same name as an existing file.
- Defining a disk data file or program where the filename is the same as the name reserved for a system device (for example, PRN or CON) or the disk name.
- Opening an I/O device not included in the configuration.

E_NOTIMP (Function Not Implemented)

Attempting to execute a function not supported for a particular implementation.

E_NOROOM (Disk Full)

Attempting to define or expand a file on a disk that does not have enough available space.

E_NLEN (Illegal Filename)

A filename was given that is null, too long, or contains illegal characters.

E_PNTR (Sector Pointer Out of Range)

Attempting to access a disk sector that does not exist. This can be an indication of a damaged keyed file. To correct, copy the file to a new file to reestablish the pointers.

E_TOFLOW (Directory or Table Overflow)

- Attempting to create a new file on a disk with a full directory.
- Opening too many channels. Maximums can be established by both the configuration file and the host operating system. The C Library cannot exceed the maximum imposed by the operating system. Check your operating system manual.

E_USERCANT (Illegal Control Operation/ Permissions Error)

- Attempting to access a file in violation of the host security system.
- User attempting to perform a privileged operation.

E_VERIF (Disk Write Error/Data Miscompare)

This error can occur in some diagnostic utilities on some systems, indicating that a disk data transfer failed.

E_ULIMIT (Too Many Users)

Too many people using the C Library at the same time.

E_NETERR (Network Error)

This error indicates a network error and only occurs when using the PRO/5 Data Server.

E_NETGONE (Network Connection Lost)

This error indicates the network connection is lost and only occurs when using the PRO/5 Data Server.

E_NETUSER (Network User Error)

This error indicates a network user error and only occurs when using the PRO/5 Data Server.