

eCIFS API Documentation

Copyright 2002, CodeFX



Software solutions for applications and appliances

www.codefx.com

voice/fax: (619) 269-4274

4645 ½ North Ave.

San Diego, CA 92116

Table of Contents

Preliminary Information	4
Introduction	5
eCIFS General Concepts	6
cifs.h and libCIFS.a.....	6
Clean Namespace.....	6
Program Flow.....	6
Common Data Types	7
API Return Codes	7
EC__STATUS and EC__TERM_STATUS Return Codes	8
Blocking and Non-blocking Function Calls	9
The Callback Function.....	10
The “int magic_num” Function Parameter	13
Unicode support.....	13
API documentation	15
Ec__ca_init().....	15
Ec__ca_deinit().....	16
Ec__ca_connectnb().....	17
Ec__ca_fopen().....	21
Ec__ca_fclose().....	24
Ec__ca_fread().....	26
Ec__ca_fwrite().....	29
Ec__ca_rmfile()	31
Ec__ca_mkdir().....	33
Ec__ca_rmdir()	35
Ec__ca_setattr()	37
Ec__ca_fsquery()	39
Ec__ca_rename()	41
Ec__ca_dirlist().....	43
Ec__ca_dirparse()	48
Ec__ca_checkconn().....	50
Ec__ca_endconn().....	51
Ec__sterror().....	52
API documentation (browsing)	53
Ec__ca_browседomains()	55
Ec__ca_parse_browседomainlist()	57
Ec__ca_browsecomputers().....	59
Ec__ca_parse_browsecomputerlist()	61
Ec__ca_browseshares().....	63
Ec__ca_parse_browseshares()	65
Appendix A – Common Structs	67
Ec__nu_name.....	67
Ec__ca_ipinfo	69
Legal Info	70

Revision History

Version	author	summary	date
1.0	jkleven	initial version	2/28/02
1.1	jkleven	added Ec__strerror() documentation	2/29/02
1.2	jkleven	describe CIFS server guest access	3/6/02
1.3	jkleven	added Unicode documentation	11/15/03

Preliminary Information

Document Overview

This document details CodeFX's eCIFS software library. The following items are discussed.

1. General purpose of the eCIFS library.
2. General eCIFS programming information.
3. Detailed eCIFS application programmer interface (API) information.

Necessary Background Information

In order to understand this document to the fullest, the following background information is highly recommended. Items (1) and (2) below are essential.

1. Some understanding of the CIFS *and* NetBIOS protocols. Please read the CodeFX whitepaper titled "CIFS Explained" to accomplish this.
2. Detailed knowledge of the 'C' programming language.
3. Basic understanding of network communications (specifically TCP/IP).
4. Understanding of execution contexts (i.e. threads/processes).

Introduction

What is eCIFS?

eCIFS is a 'C' language software library designed for use in embedded systems as well as desktop applications. The purpose of this software library is to allow programmers to use the Common Internet File System (CIFS) protocol without having to design, implement, and debug a full CIFS protocol stack. eCIFS presents an API to the programmer which allows for easy network file access without getting bogged down in packet level negotiations.

eCIFS operates in client mode. This means that eCIFS initiates connections to a server, issues requests to the server (such as read a file), and receives responses back. The eCIFS code does not include CIFS server support and therefore does not support *uninitiated incoming* CIFS requests.

For more information on CIFS file sharing, please refer to the CodeFX whitepaper titled "CIFS Explained". "CIFS Explained" details the feature set of the protocol as well as the underlying mechanisms which accomplish these features.

eCIFS General Concepts

The following subsections detail information that relates to the eCIFS library as a whole and therefore would not be appropriate to list in the main API documentation below.

cifs.h and libCIFS.a

The file `cifs.h` is the main interface (header) file for the entire eCIFS library. Programmers who wish to call eCIFS functions and declare eCIFS data types need only `#include "cifs.h"` in their source code. In fact, no other eCIFS header files should *ever* be included by programmers – `cifs.h` contains everything needed. Of course, for the linking stage of a build, `libCIFS.a` must be included as a library to the linker. `libCIFS.a` contains the actual software library routines.

Clean Namespace

All programming constructs that share namespace with user code are prefixed with the string `"Ec__"`. This helps to insure that no conflicts exist between eCIFS code and the programmer's code.

Program Flow

A re-occurring theme in the API is 'C' structures (structs) that are initialized in one function and then used in another. For example, when a programmer wishes to start a CIFS session with a remote server, the `Ec__ca_connectnb()` function is called and is passed a pointer to an `Ec__ca_conn` struct. This function will then establish the connection with the remote server and fill out the `Ec__ca_conn` struct with pertinent connection information. Now that the CIFS connection has been established, *all file operation functions* (such as `Ec__ca_fopen` and `Ec__ca_fread`) *require the Ec__ca_conn struct to be passed in as an argument*. Put another way, the `Ec__ca_conn` structure is what tells the file open and read functions which CIFS connection the programmer is referring to. The `Ec__ca_conn` structure is sometimes referred to as a connection handle.

A similar sequence occurs when opening a file with the `Ec__ca_fopen()` function. A pointer to an `Ec__ca_fdata` struct is passed in, and the function initializes this structure with specific file information once the file has been opened on the server. This structure is then passed into the functions like `Ec__ca_fread()` to indicate which file to operate on. The `Ec__ca_fdata` structure is sometimes referred to as a file handle.

Common Data Types

The following data types are heavily used in cifs.h and the API documentation below.

int8_t	= an 8 bit signed integer
int16_t	= a 16 bit signed integer
int32_t	= a 32 bit signed integer
u_int8_t	= an 8 bit unsigned integer
u_int16_t	= a 16 bit unsigned integer
u_int32_t	= a 32 bit unsigned integer

API Return Codes

The file util\ret_code.h contains all of the possible eCIFS function return codes that the API programmer will see. There are three distinct classes of return codes – each is described below.

Successful return codes:

These return codes indicate that there was no problem and are always greater than or equal to zero. There are only two; EC__SUCCESS and EC__PENDING.

Error return codes:

These negative value codes indicate that the operation that was requested (via an API call) has failed for some reason. The code itself indicates why the operation failed. Examples are EC__BADARG_SHARE, EC__BADARG_BUFFER, and EC__NOSUCHFILE. Error return codes are denoted throughout the rest of this text as EC__XXXX.

Termination error return codes:

These negative value error codes indicate that the requested API operation has failed **AND** that the entire CIFS server connection has terminated. ***After receiving a termination error code it is illegal to call any other API functions with that particular Ec__ca_conn structure, as that connection is now non-existent. In addition, any opened files on that connection will also be invalidated and cannot be used to call any file operation API functions. Violating these rules may cause program failure!***

To help insure that an Ec__ca_conn argument is not used to call an API function after a termination error code has been returned, the EC__CA_TERMD() macro can optionally be used. After calling an API operation (blocking or non-blocking) the return code can be passed into the EC__CA_TERMD() macro. This macro will return EC__TRUE if the return code was indeed a termination error code. If the macro returns EC__TRUE the connection must be re-initialized (via the Ec__ca_connectnb function) if more API operations are to be called. **DO NOT CALL MORE API FUNCTIONS WITHOUT RE-ESTABLISHING THE CONNECTION FIRST.** The Ec__ca_conn struct can be re-used as the argument to Ec__ca_connectnb() if the programmer desires to re-establish the connection.

Example termination return codes are EC__TERM_TCP (TCP connection has been dropped) and EC__TERM_TIMEOUT (the remote server is not responding). All termination return codes have the TERM string as a part of their name. Termination error return codes are denoted throughout the rest of this text as EC__TERM_XXXX.

Please note that although every API function has the possible return codes listed below in the function-by-function documentation, all possible EC__TERM_XXXX return codes can be returned by ANY of the API functions. This is because previously executed non-blocking API requests (i.e. requests which are pending) will return a termination style return code to the *very next API call*. In other words, any API call that the programmer executes could very well return an EC__TERM_XXXX code for any of the currently pending non-blocking functions. This is a necessity because eCIFS could not allow new requests to be made if any of the pending requests actually terminated the entire CIFS connection.

EC__STATUS and EC__TERM_STATUS Return Codes

Every received packet from the CIFS server contains an error class and error code field. These fields are used by the server to indicate an error condition during packet exchanges between the client and server. When a CIFS server returns an error code and class in a packet, eCIFS makes every attempt to decipher the error class and error code into a meaningful return code to pass back to the API programmer (such as EC__NOSUCHFILE or EC__TERM_BADPASSWD). However, because of the vast possible combinations of error codes and classes, the high number of undocumented error codes and classes, and the many CIFS server code bases which handle errors differently, eCIFS cannot always pass back a meaningful return code based on the error class and code. In these cases eCIFS will return either EC__STATUS or EC__TERM_STATUS. This is eCIFS' way of indicating that the CIFS server returned an unknown error class and code combination. If the programmer is accepting input from a user and then calling eCIFS API functions, it is best to indicate to the user that the operation has failed and to have the user re-input the parameters that feed into the eCIFS API call.

EC__STATUS therefore indicates that for unknown reasons the CIFS server did not honor the request and the operation has failed. The underlying CIFS connection however is still valid and operational.

EC__TERM_STATUS also indicates that for unknown reasons the CIFS server did not honor the request and the operation failed. However in this case, the underlying connection is no longer operational, and no more API calls can be made utilizing the Ec__ca_conn struct that represented this connection.

Blocking and Non-blocking Function Calls

Most of the eCIFS functions can either be called in blocking mode or non-blocking mode. The first argument of functions that can have either of these modes is “int call_mode”.

When the #defined token “EC__BLOCK” is passed in for the call_mode parameter, the function will block (not return) until the actual requested event (e.g. file open, file read, CIFS connect...) has completed or failed.

When the #defined token “EC__NOBLOCK” is passed in for the call_mode parameter, the function will return *before* the actual requested event has taken place. The function call will block for a brief time period in which the arguments are checked and a message is sent to schedule the request. At this point (assuming the arguments were OK) the function will return EC__PENDING, which indicates that although the event has not yet taken place, it has been scheduled and will be completed as soon as possible in the future.

Once the requested non-blocking event has actually completed or failed, the programmer is notified via callback execution. This callback is a ‘C’ function call that the programmer specifies for each active CIFS connection. **Please note that no eCIFS API functions can be called from within the callback function. The callback function is executed by internal eCIFS threads, and therefore cannot safely call API functions.**

In many API functions, structures and basic types are passed in by reference. This allows the eCIFS’ functions to modify the parameters that are passed in, effectively passing information *back* to the caller. ***However, because non-blocking operations return before that actual event has taken place, there is no way these API functions can modify parameters passed by reference; the information to pass back has not yet been generated.*** In these cases the appropriate information will be passed to the programmer via the callback execution. The only time that a non-blocking API function will modify an actual function parameter is with a buffer argument. In these cases, the exact same buffer pointer that was passed in as a parameter will be passed back during callback execution. If the execution was successful then the buffer will have been filled out.

Because non-blocking function calls cannot modify function arguments, there are a number of API functions that behave differently when called in non-blocking mode. Function parameters are sometimes not required in non-blocking mode, and NULL can be passed in instead (this is documented function-by-function in the API documentation below). NULL’s can be passed instead of the parameter whenever a function argument exists solely to pass data *back* to the programmer.

In some cases, an argument passed by reference is used to pass data *in to* eCIFS, as well as *out to* the programmer. In these cases, the argument is still necessary to get the data in, but the data will never be changed to pass the information back out to the programmer – the event has not occurred. In these cases, the data that was passed in will be copied, stored, and finally passed back in the callback execution. However, the data object that was passed in by reference to the API function will never be modified by eCIFS, and therefore will never pass data *back* to the programmer. The copy of the data will be modified though, and this will be accessible to the programmer upon callback execution.

The Callback Function

The first eCIFS function call (excluding `Ec__ca_init`) that must be executed before further function calls can be made is `Ec__ca_connectnb()`. One of the arguments to this function is the callback function pointer. By specifying a function pointer here the programmer indicates which function to use for asynchronous callback notification for all non-blocking calls *related to this new connection*. All non-blocking function calls which then terminate (either successfully or unsuccessfully) will execute this callback and indicate the status of the request. Therefore, each active CIFS server connection (created with `Ec__ca_connectnb`) can have its own callback function – although nothing is wrong or illegal with using the same callback for multiple connections (as long as the code contained in the callback is re-entrant).

```
int Ec__ca_connectnb(
    int call_mode,
    u_int32_t dia_list,
    Ec__ca_ipinfo *ipinfo,
    Ec__nu_name *called_name,
    Ec__nu_name *calling_name,
    char username[],
    char passwd[],
    char share[],
    u_int32_t opts,
    Ec__ca_conn *conn,
    void (*callback)(u_int16_t msg_type, int status, int magic_num, void *data),
    int magic_num);
```

The prototype for `Ec__ca_connectnb()`

As you can see, the 11th argument to `Ec__ca_connectnb()` is a pointer to the callback function. The programmer must write this callback function (*if non-blocking function are to be called*) and then pass in a pointer to the function upon calling `Ec__ca_connectnb()`. There are 4 arguments to this callback, and the eCIFS library fills them all in when the function is executed. It is then up to the programmer (who wrote the callback) to monitor and manipulate the passed in data. Each of the 4 arguments is detailed below.

`u_int16_t msg_type`: Because this callback function will indicate when *any* request has finished for the given connection, the `msg_type` parameter indicates which request this callback is executing for. The value will be equal to one of the enum values below.

```
enum operation {
    MSG_CONNECT,
    MSG_FOPEN,
    MSG_FREAD,
    MSG_FWRITE,
    MSG_DIRECTORY_NEW,
    MSG_DIRECTORY_PREV,
    MSG_REMOVE,
    MSG_REMOVE,
    MSG_FCLOSE,
    MSG_RENAME,
    MSG_SETATTR,
    MSG_FSQUERY,
    MSG_BENUM2,
    MSG_BENUM,
    MSG_ENDCONN,
    INVALID_MSG
};
```

For example, if `Ec__ca_connectnb()` is called in non-blocking mode, when the callback is executed the `msg_type` parameter will be equal to `MSG_CONNECT`. In a similar sense, if `Ec__ca_fclose()` is called in non-blocking mode, `msg_type` will be equal to `MSG_FCLOSE` upon callback execution.

int status: The status integer specifies the return code for the finished operation, and hence whether or not the operation was successful. A status parameter equal to EC__SUCCESS indicates that the operation completed successfully. Negative status parameters can also be used for the int status parameter; however, *the status code can never be a terminated return code* (see above explanation of return codes). *The repercussions of this are very important.* This means that no callback execution can ever indicate that the entire connection is dead.

The reason that the callback cannot indicate total termination with a terminated return code stems from the fact that the callback is executed from an internal eCIFS thread. For total termination to occur, the calling thread (the API programmers thread) must close down queues and semaphores to clean up the system. However, because the callback is NOT executing under the API programmers thread there is no way to force the API thread to execute clean up procedures.

Explained in another way, the status message can only be EC__SUCCESS (successful completion), or EC__XXXX (the operation failed), but it can never be EC__TERM_XXXX (would indicate that entire connection has been terminated). A situation can arise however when a requested non-blocking API operation does need to indicate a termination code to a callback. Instead of issuing the term code directly, the callback will instead set status equal to EC__FAIL. This generic token is the programmers sign that something has gone wrong, and most likely the entire connection is dead. When this occurs, the very next API call (either blocking or non-blocking) will return immediately (no callback execution of any kind) and indicate the EC__TERM_XXXX code that caused the termination.

The above situation can sometimes require the use of the Ec__ca_checkconn() API function. This function simply checks to see whether or not the connection is still alive with the CIFS server. This function will therefore either return EC__SUCCESS (indicating that the connection is still alive) or EC__TERM_XXXX (indicating that the connection is terminated). Ec__ca_checkconn() can be called to determine the *real* error code associated with a callback execution that had the status parameter set to EC__FAIL. Note however that Ec__ca_checkconn() cannot be called inside of the callback function directly as calling any API functions within the callback is illegal.

To summarize, upon receiving the status equal to EC__FAIL in a callback function, the very next API function call will most likely result in an immediate return of a termination return code. If API functions are not being called frequently from the programmers thread it can be useful to call Ec__ca_checkconn() (from the programmers thread) to receive the actual termination code.

Typical callback functions usually send some sort of IPC message to the real programmers thread (so as to change out of the eCIFS internal thread) and then continue to call API operations once the message is received.

int magic_num: Whenever an API function is called which has the ability to be non-blocking (i.e. the first parameter of the function is int call_mode), the function will also accept an "int magic_num". When a function is then called in non-blocking

mode, the passed in magic number is stored with the pending request. When the request has finished (successfully or un-successfully) this same magic_num will be presented in the callback execution. In this manner, programmers can multiplex the same request (i.e. multiple non-blocking file open requests) and still have a method for determining which callback execution correlated to which API request.

For more information on the int magic_num parameter, please reference the section below titled "[The 'int magic_num' Function Parameter](#)".

void *data: Often times when a callback executes various types of data must also be passed back to the API programmer. The void *data pointer is used to accomplish this. When the status parameter is equal to EC__SUCCESS, the void *data pointer will point to an API specific structure. The msg_type integer indicates *which* type of structure it points to. The table below indicates the structure that the data pointer should be cast to based on the msg_type.

msg_type	structure to cast *data into
MSG_CONNECT	msg_connect_rsp *
MSG_FOPEN	msg_fopen_rsp *
MSG_FREAD	msg_fread_rsp *
MSG_FWRITE	msg_fwrite_rsp *
MSG_DIRLIST_NEW	msg_dirlist_rsp *
MSG_DIRLIST_PREV	msg_dirlist_rsp *
MSG_RMFILE	data always NULL
MSG_RMDIR	data always NULL
MSG_FCLOSE	data always NULL
MSG_RENAME	data always NULL
MSG_SETATTR	data always NULL
MSG_FSQUERY	msg_fsquery_rsp *

As the table makes evident, some requests do not have any additional data to be passed back to the programmer, and therefore the data pointer is always NULL in these cases. This happens when the status (either EC__SUCCESS or an error code) is all that is necessary to know (i.e. in a file rename operation, it either works or returns an error code, no additional data is required).

All of the above structures are defined in cifs.h, and can be referenced there. ***Please be aware that once the callback function returns, the void *data pointer memory will be freed. Dereferencing the void *data memory pointer after callback completion will have disastrous results.*** Because of this, the structure should always be shallow copied into a malloc'd or global response structure. The new stable memory structure pointer is then typically passed to another thread via an IPC message.

In almost all cases when the status integer is set to anything besides EC__SUCCESS, the void *data pointer will be set to NULL. The idea being that there is not additional

information to pass back if the operation fails. *There is one exception to this rule.* If `Ec__ca_connectnb()` is called in non-blocking mode and fails, `EC__FAIL` will be passed as the status. In this case, the void `*data` pointer *will still point to a `msg_connect_rsp` structure.* This is necessary because another API function (such as `Ec__ca_checkconn`) must still be called (in the main API programmers thread) for code cleanup purposes. However, all API functions require an `Ec__ca_conn` argument (to deduce which CIFS server connection the programmer is referring to). Because of this, failed non-blocking `Ec__ca_connectnb` requests still pass back a valid data pointer that holds an `Ec__ca_conn` structure. The programmer is then capable of copying the `Ec__ca_conn` structure, passing it back to the main API calling thread, and finally calling another API function with it such as `Ec__ca_checkconn`.

For more help on the non-blocking API calls please see the example code which is heavily commented.

The “int magic_num” Function Parameter

All API function calls that can be blocking or non-blocking (indicated when the first argument is `int call_mode`) have an `int magic_num` which is also passed in by the programmer upon function execution. *This number is only used when the function is called in non-blocking mode (`EC__NOBLOCK`).* When called in non-blocking mode, the `magic_num` that is passed in by the programmer will be passed back in the callback upon operation completion (see above). When called in blocking mode, any `magic_num` passed in will not be utilized in any way. CodeFX uses the number 0 in these cases.

Unicode support

The following eCIFS functions support the Unicode character set:

- `Ec__ca_connectnb()`
- `Ec__ca_fopen()`
- `Ec__ca_rmfile()`
- `Ec__ca_mkdir()`
- `Ec__ca_rmdir()`
- `Ec__ca_setattr()`
- `Ec__ca_rename()`
- `Ec__ca_dirlist()`
- `Ec__ca_dirparse()`

In order to utilize the Unicode character set in these eCIFS calls the `Ec__ca_connectnb` function must specify the `EC__USE_UNICODE` option in the `opts` argument when the connection is established. If this argument is not set when the programmer calls `Ec__ca_connectnb` *none* of the eCIFS functions can utilize Unicode.

When the programmer calls the `Ec__ca_connectnb` call with the `EC__USE_UNICODE` flag set in the `opts` argument, the eCIFS code will attempt to connect to the server using the Unicode character set. However, some servers do not support Unicode (win95/98, older versions of Samba, etc.) and in these cases eCIFS will return the error code `EC__TERM_NO_UNICODE`. The only recourse when this is returned is to call the `Ec__ca_connectnb` function again without the `EC__USE_UNICODE` flag.

Note that the following server side entities typically can support Unicode if the server is Unicode enabled:

- usernames
- passwords
- share names
- files
- directories

However, NetBIOS server names do NOT support Unicode because the actual NetBIOS specification does not allow for chars outside of a very specific ASCII range.

If the server does accept Unicode strings the connection will succeed and return `EC__SUCCESS` as usual. At this point, many of the arguments to the above API's can accept Unicode strings. In the detailed "API Documentation" section below, all arguments that can accept Unicode strings are noted with the phrase (*UTF-8 OK*). Again, these function arguments can only accept Unicode if the `Ec__ca_connectnb` function was called with the `EC__USE_UNICODE` flag set in the `opts` argument.

Anytime the programmer has successfully established a Unicode enabled connection and wishes to pass a Unicode string to the eCIFS function library, the string MUST be encoded in the UTF-8 format. UTF-8 is a "serialization method" for the Unicode character set. UTF-8 is the most common serialization method used for applications and operating systems – thereby making it a defacto standard. The following table shows how Unicode characters (on the left) are mapped into UTF-8 byte strings.

Unicode characters	Same characters in UTF-8
U-00000000 - U-0000007F	0xxxxxxx
U-00000080 - U-000007FF	110xxxxx 10xxxxxx
U-00000800 - U-0000FFFF	1110xxxx 10xxxxxx 10xxxxxx

Although Unicode provides for 3 and 4 byte characters, the CIFS protocol does not support transmission of these Unicode characters. The programmer cannot call any of the eCIFS API's with a UTF-8 string which maps to a Unicode character greater than 2 bytes in size.

Unicode and UTF-8 are covered extensively at the following web sites:

<http://eyegene.ophthy.med.umich.edu/unicode/>

<http://www.cl.cam.ac.uk/~mgk25/unicode.html>

<http://www-106.ibm.com/developerworks/linux/library/l-linuni.html>

API documentation

Ec__ca_init()

Prototype:

```
int Ec__ca_init(int log_level)
```

Purpose:

This function is used to prepare the eCIFS library for API execution. It also indicates the level of debug information that the eCIFS library should log.

Notes:

This function *must* be called before any other eCIFS API function is called.

Parameters:

int log_level: This integer must be passed in as one of the following defined values:
EC__LOG_ALL (logs informational, anomaly, error, and hard error debug messages)
EC__LOG_MOST (logs anomaly, error, and hard error debug messages)
EC__LOG_ERRS (logs error and hard error messages)
EC__LOG_BIGERRS (logs only hard error messages)
EC__LOG_OFF (no logging)

Return:

EC__SUCCESS

EC__TERM_INIT

Indicates init failure due to port specific reasons.

Ec__ca_deinit()

Prototype:

```
int Ec__ca_deinit(void)
```

Purpose:

This function is used to cleanup any system resources which were utilized by execution of the eCIFS software library.

Notes:

It is illegal to call any other eCIFS API functions after this function has been executed. Before calling `Ec__ca_deinit()` the programmer should insure that all currently active CIFS server connections have been terminated. This function will not tear down CIFS server connections; it is the programmer's responsibility to do this. If the programmer wishes to call eCIFS API functions after calling this function, `Ec__ca_init()` must be called again.

Parameters:

Return:

`EC__SUCCESS`

The function successfully released all of the utilized resources.

`EC__TERM_INIT`

The function could not release all of the utilized resources.

Ec__ca_connectnb()

Prototype:

```
int Ec__ca_connectnb(int call_mode, u_int32_t dia_list,
    Ec__ca_ipinfo *ipinfo, Ec__nu_name *called_name, Ec__nu_name
    *calling_name, char username[], char passwd[], char share[],
    u_int32_t opts, Ec__ca_conn *conn,
    void (*callback)(u_int16_t msg_type, int status,
    int magic_num, void *data), int magic_num)
```

Purpose:

This function is used to connect to a CIFS server.

Notes:

This function connects to a CIFS server. This means that the following items must be identified in the function arguments:

- server name (NetBIOS name or IP)
- username (optional)
- password (optional)
- server share name

Because a great deal of information must be specified here, this is the most complex API function.

Many CIFS servers can grant guest access to a CIFS client (if the server administrator has enabled guest access). Unfortunately, not all CIFS servers require the same username/password combination to grant guest access. The following table describes common username and password combinations that are likely to grant guest access on the corresponding CIFS server type.

CIFS server type	Username	Password
Windows	“Guest”	NULL
Samba	“”	NULL

It is OK to call `Ec__ca_connectnb()` twice if the programmer is attempting to gain guest access and does not know the CIFS server type. If the first function execution is not successful try the second “Samba” username/password combination. If neither of these username/password combinations work then it is likely that guest access is not enabled and the programmer will have to enter an actual username and password combination.

Parameters:

int call_mode: As described above in the “eCIFS General Concepts” section under the [“Blocking and Non-blocking Function Calls”](#) heading, this argument can either be `EC_BLOCK` or `EC_NOBLOCK`.

u_int32_t dia_list: Currently, the only dialect supported “NT LM 0.12”. Therefore this argument must be `EC_NTLM012`.

Ec__ca_ipinfo *ipinfo: This struct specifies how the CIFS server's NetBIOS name (specified directly below) is resolved into an IP address. This struct is detailed [here](#) in Appendix A.

Ec__nu_name *called_name: This struct specifies the NetBIOS name of the CIFS server that the programmer wishes to connect to. This struct must be declared, filled out, and finally passed in as a pointer. The struct and its various fields are detailed in [Appendix A](#).

char *nb_name: Use this struct member field to point to the name of the server that you wish to connect to.

char *scope_id: Either NULL or the scope_id.

Ec__nu_serv_type serv_type: Normally set to *fileserv*.

Ec__nu_name *calling_name: This struct describes *the client's* (as in the system that the programmer is executing code on) NetBIOS name. *This argument can be NULL.* When this argument is NULL, the eCIFS software library will specify the NetBIOS name as "eCIFS_CLIENT". For typical CIFS server access, the NetBIOS calling name is not important and therefore it is usually acceptable to leave this argument as NULL. Details on the Ec__nu_name struct can be found in [Appendix A](#).

char username[] (UTF-8 OK): This string specifies the user name that will be used to login to the CIFS server. This argument can be NULL when connecting to a share-level security CIFS server. Note that if the CIFS server does utilize share-level security the username string will be ignored regardless of whether it is specified or not. If specified, this string cannot exceed 255 bytes.

char passwd[] (UTF-8 OK): This string specifies the password. This argument can also be NULL if the programmer is attempting to login as a guest. If specified, this string cannot be greater than 255 bytes.

char share[] (UTF-8 OK): This string specifies the share name that the programmer wishes to access. This string cannot be greater than 255 bytes.

u_int32_t opts: This 32 bit integer can either be 0 or EC__USE_UNICODE. The EC__USE_UNICODE flag must be turned on if this or any other eCIFS function will be called with UTF-8 arguments. See the [Unicode Support](#) segment for more info.

Ec__ca_conn *conn: This structure will be filled out by the Ec__ca_connectnb() function after a successful connection has occurred. The programmer will then pass this filled in conn structure to all other API function calls which will use this connection. When calling Ec__ca_connectnb(), do not declare a pointer and then pass it to Ec__ca_connectnb(). Instead, declare the actual struct, and then pass in the pointer to the struct. More information can be found in the "eCIFS General Concepts" section above under the "[Program Flow](#)" heading.

If this function is called in non-blocking mode the conn parameter may be NULL. In non-blocking mode, the conn parameter will not be utilized in any way. A valid Ec__ca_conn structure will then be made available upon callback execution. You may still pass in a conn pointer, but the structure will *never* get initialized. The programmer must obtain the valid Ec__ca_conn structure during callback execution.

void (*callback)(u_int16_t msg_type, int status, int magic_num, void *data): This function pointer is used to tell the Ec__ca_connectnb() function how to notify the programmer when non-blocking API calls have finished execution. In other words, an internal eCIFS thread will execute the function that is specified here whenever a non-blocking API call has finished. This argument can be NULL as long as no API functions will be called in non-blocking mode. More information can be found in the “eCIFS General Concepts” section above under the “[Blocking and Non-Blocking Function Calls](#)” heading.

int magic_num: When this function is called in non-blocking mode this magic number will be passed to the callback function upon function completion. More information can be found in the “eCIFS General Concepts” section above under the “[int magic_num](#)” heading.

Return:

(Note that all non-successful return codes are EC__TERM_XXXX type codes. If the connect function fails, it is definitely illegal to use the Ec__ca_conn struct for further API calls. Therefore any failures here must return termination codes.)

EC__SUCCESS

Successfully connected to CIFS server. The Ec__ca_conn argument was filled out.

EC__PENDING

Indicates that the non-blocking request is currently pending. Callback will execute when operation has finished.

EC__TERM_BADARG_BLOCKMODE

The call_mode argument above was invalid.

EC__TERM_BADARG_DIALECT

The dia_list argument above had an invalid dialect bit set.

EC__TERM_BADARG_NBNSIP

The ipinfo struct above specified an nb_mode that requires an NBNS server IP to be specified, but no such IP was set.

EC__TERM_BADARG_NBQUERYMODE

The ipinfo struct above specified an invalid nb_mode.

EC__TERM_BADARG_CALLEDNAME

The called_name argument above specified an invalid NetBIOS name.

EC__TERM_BADARG_CALLINGNAME

The calling_name argument above specified an invalid NetBIOS name.

EC__TERM_BADARG_USERNAME

The username argument above is invalid (too long or bad UTF-8).

EC__TERM_BADARG_PASSWD

The passwd argument above is invalid (too long or bad UTF-8).

EC__TERM_BADARG_SHARE

The share argument above is either NULL, too long, or bad UTF-8

EC__TERM_BADARG_OPTS

The opts argument above was invalid.

EC__TERM_NB_NOTFOUND

The NetBIOS called_name specified above did not resolve to an IP address.

EC__TERM_TCP

A TCP connection could not be established with the CIFS server or the underlying TCP stack returned an error after the connection was established.

EC__TERM_TIMEOUT

The CIFS server did not reply to one of the pending requests for 10-20 seconds.

EC__TERM_CPVIOLATION

An incoming CIFS packet was malformed.

EC__TERM_DIALECT

The CIFS server will not communicate in any of the dialects selected in the dia_list argument above.

EC__TERM_NOENCRYPT

The server wanted to send passwords in plaintext, this is not allowed.

EC__TERM_BADPASSWD

The server indicated that the password was incorrect.

EC__TERM_STATUS

The server returned an unknown error class and status in response to an eCIFS packet. Please reference the “eCIFS General Concepts” section above under the “[EC__STATUS and EC__TERM_STATUS Return Codes](#)” heading for more information.

EC__TERM_SHARE

The CIFS server indicated that the specified share name does not exist.

EC__TERM_NO_UNICODE

Indicates that the server does not support Unicode.

EC__TERM_VIOLATION_UNICODE

Indicates that a Unicode protocol violation occurred.

Ec__ca_fopen()

Prototype:

```
int Ec__ca_fopen(int call_mode, Ec__ca_conn *conn,  
    Ec__ca_fdata *f_data, char *name, int access_mode,  
    int opts, int share_mode, int magic_num)
```

Purpose:

This function is used to open or create a file.

Notes:

Parameters:

int call_mode: As described above in the “eCIFS General Concepts” section under the “[Blocking and Non-blocking Function Calls](#)” heading, this argument can either be EC__BLOCK or EC__NOBLOCK.

Ec__ca_conn *conn: This argument must point to a valid Ec__ca_conn structure that was *successfully* initialized by the Ec__ca_connectnb function.

Ec__ca_fdata *f_data: This argument is what holds information about the opened or created file. The Ec__ca_fopen function will initialize this structure to the appropriate values upon *successful* completion. For all other API functions that operate on files this Ec__ca_fdata structure will be passed in so the function knows which file to operate on. More information can be found in the “eCIFS General Concepts” section above under the “[Program Flow](#)” heading.

If this function is called in non-blocking mode the f_data pointer may be NULL. In non-blocking mode the f_data pointer will not be utilized in any way. A valid Ec__ca_fdata structure will then be made available upon callback execution. It is still valid to pass in an f_data pointer, but the structure will *never* get initialized. The programmer must obtain the valid Ec__ca_fdata structure during callback execution.

char *name (UTF-8 OK): This argument specifies the file name string that is to be created or opened.

int access_mode: This argument specifies the type of file access the programmer desires. It should be set to one of the following enumerated values:

EC__AM_READ (read only access)

EC__AM_WRITE (write only access)

EC__AM_READWRITE (read and write access)

int opts: This argument specifies the create disposition of the file. It should be set to one of the following enumerated values:

EC__OPT_EXCLOPEN (exclusive open – only open this file, if it does not exist fail)

EC__OPT_EXCLCREATE (exclusive create – only create this file, if it exists fail)

EC__OPT_OPENCREATE (open or create – if the file exists open it, if not create it)

int share_mode: This argument specifies the level of file access that other processes are allowed to have with this file. It should be set to one of the following enumerated values:

EC__SM_READ (allow other processes to read from the file)

EC__SM_WRITE (allow other processes to write to the file)

EC__SM_ALL (allow other processes full uninhibited access)

EC__SM_DENY (deny other processes any type of access)

int magic_num: When this function is called in non-blocking mode this magic number will be passed to the callback function upon function completion. More information can be found in the “eCIFS General Concepts” section above under the [“int magic_num”](#) heading.

Return:

EC__SUCCESS

Successfully opened or created the file. The Ec__ca_fdata argument was initialized.

EC__PENDING

Indicates that the non-blocking request is currently pending. Callback will execute when operation has finished.

EC__TERM_TCP

Indicates that the underlying TCP stack caught an error while trying to send or receive packets.

EC__TERM_TIMEOUT

The CIFS server did not reply to one of the pending requests for 10-20 seconds.

EC__TERM_CPVIOLATION

Indicates that a significant CIFS protocol violation has occurred. This typically results if eCIFS receives heavily malformed or partial packets.

EC__CPVIOLATION

A minor CIFS protocol violation has occurred.

EC__STATUS

The server returned an unknown error class and status in response to an eCIFS packet. Please reference the “eCIFS General Concepts” section above under the [“EC__STATUS and EC__TERM_STATUS Return Codes”](#) heading for more information.

EC__BADARG_BLOCKMODE

The call_mode argument above was invalid.

EC__BADARG_CONN

The conn pointer above does not point to a valid and initialized Ec__ca_conn structure.

EC__BADARG_FNAME

The name argument above does not point to a valid file name (it is either NULL, too long, or bad UTF-8).

EC__BADARG_ACCESSMODE

Indicates that the `access_mode` argument was not set to a valid enumerated access mode.

`EC__BADARG_OPTS`
Indicates that the `opts` argument was not set to a valid enumerated option.

`EC__BADARG_SHAREMODE`
Indicates that the `share_mode` argument was not set to a valid enumerated share mode option.

`EC__FILEEXISTS`
CIFS server indicated that the file already exists.

`EC__NOSUCHFILE`
CIFS server indicated that no such file exists.

`EC__NOACCESS`
CIFS server indicated that the client is not allowed to access this file.

`EC__TERM_VIOLATION_UNICODE`
Indicates that a Unicode protocol violation occurred.

Ec__ca_fclose()

Prototype:

```
int Ec__ca_fclose(int call_mode, Ec__ca_conn *conn,  
    Ec__ca_fdata *f_data, int magic_num)
```

Purpose:

This function closes a previously opened file and invalidates the `Ec__ca_fdata` file handle.

Notes:

It is good programming practice to close files that the programmer is no longer working with to avoid running out of open file resources on the server. However, when a connection terminates in any way the CIFS server is supposed to de-allocate all connection oriented resources – which should also clean up any opened files.

Once this function has successfully completed the `f_data` structure is invalid and cannot be used in other API functions unless it is re-initialized by calling `Ec__ca_fopen` with it.

Parameters:

int call_mode: As described above in the “eCIFS General Concepts” section under the “[Blocking and Non-blocking Function Calls](#)” heading, this argument can either be `EC__BLOCK` or `EC__NOBLOCK`.

Ec__ca_conn *conn: This argument must point to a valid `Ec__ca_conn` structure that was *successfully* initialized by the `Ec__ca_connectnb` function.

Ec__ca_fdata *f_data: This argument must point to a valid `Ec__ca_fdata` structure that was *successfully* initialized by the `Ec__ca_fopen()` function. This file handle will be closed after this function successfully completes.

int magic_num: When this function is called in non-blocking mode this magic number will be passed to the callback function upon function completion. More information can be found in the “eCIFS General Concepts” section above under the “[int magic_num](#)” heading.

Return:

`EC__SUCCESS`

Successfully closed the file.

`EC__PENDING`

Indicates that the non-blocking request is currently pending. Callback will execute when operation has finished.

`EC__TERM_TCP`

Indicates that the underlying TCP stack caught an error while trying to send or receive packets.

EC__TERM_TIMEOUT

The CIFS server did not reply to one of the pending requests for 10-20 seconds.

EC__TERM_CPVIOLATION

Indicates that a significant CIFS protocol violation has occurred. This typically results if eCIFS receives heavily malformed or partial packets.

EC__CPVIOLATION

A minor CIFS protocol violation has occurred.

EC__STATUS

The server returned an unknown error class and status in response to an eCIFS packet. Please reference the “eCIFS General Concepts” section above under the [“EC__STATUS and EC__TERM_STATUS Return Codes”](#) heading for more information.

EC__BADARG_BLOCKMODE

The call_mode argument above was invalid.

EC__BADARG_CONN

The conn pointer above does not point to a valid and initialized Ec__ca_conn structure.

EC__BADARG_FDATA

The f_data pointer does not point to a valid and initialized Ec__ca_fdata structure.

EC__BADFID

CIFS server indicated that the Ec__ca_conn structure contained a bad file handle. This usually indicates that the conn pointer did not point to valid or initialized data.

EC__TERM_VIOLATION_UNICODE

Indicates that a Unicode protocol violation occurred.

Ec__ca_fread()

Prototype:

```
int Ec__ca_fread(int call_mode, Ec__ca_conn *conn,  
    Ec__ca_fdata *f_data, u_int8_t *buf, u_int32_t offset,  
    u_int32_t *size, int magic_num)
```

Purpose:

This function reads from a previously opened file.

Notes:

The file to read from must have been previously successfully opened with read access enabled.

Parameters:

int call_mode: As described above in the “eCIFS General Concepts” section under the “[Blocking and Non-blocking Function Calls](#)” heading, this argument can either be EC__BLOCK or EC__NOBLOCK.

Ec__ca_conn *conn: This argument must point to a valid Ec__ca_conn structure that was *successfully* initialized by the Ec__ca_connectnb function.

Ec__ca_fdata *f_data: This argument must point to a valid Ec__ca_fdata structure that was *successfully* initialized by the Ec__ca_fopen() function. This argument indicates which file you wish to read from.

u_int8_t *buf: Specifies where eCIFS should store the data that is read. Note that the buffer must be at least as big as the read size requested.

u_int32_t offset: Indicates the byte offset (from the beginning of the file) that the read is requested for.

u_int32_t *size: A pointer to a 32-bit integer that specifies how many bytes to read from the file. This argument is specified as a pointer because upon *blocking* completion of this function the 32-bit integer pointed to by size will indicate the *actual* amount of data that was read into the buffer. If this value is less than the requested read size after a successful Ec__ca_fread() call it is very likely that the read size and offset actually went past the bounds of the file (i.e., the programmer read past the EOF marker on the file).

If called in non-blocking mode the size parameter will indicate the actual amount of data read *upon callback execution* – the size integer that is passed in here will not be modified.

int magic_num: When this function is called in non-blocking mode this magic number will be passed to the callback function upon function completion. More

information can be found in the “eCIFS General Concepts” section above under the [“int magic_num”](#) heading.

Return:

EC__SUCCESS

Successfully read from the file. A successful read can sometimes read only 0 bytes of data. Be sure to check the actual size read.

EC__PENDING

Indicates that the non-blocking request is currently pending. Callback will execute when operation has finished.

EC__TERM_TCP

Indicates that the underlying TCP stack caught an error while trying to send or receive packets.

EC__TERM_TIMEOUT

The CIFS server did not reply to one of the pending requests for 10-20 seconds.

EC__TERM_CPVIOLATION

Indicates that a significant CIFS protocol violation has occurred. This typically results if eCIFS receives heavily malformed or partial packets.

EC__CPVIOLATION

A minor CIFS protocol violation has occurred.

EC__STATUS

The server returned an unknown error class and status in response to an eCIFS packet. Please reference the “eCIFS General Concepts” section above under the [“EC__STATUS and EC__TERM_STATUS Return Codes”](#) heading for more information.

EC__BADARG_BLOCKMODE

The call_mode argument above was invalid.

EC__BADARG_CONN

The conn pointer above does not point to a valid and initialized Ec__ca_conn structure.

EC__BADARG_FDATA

The f_data pointer does not point to a valid and initialized Ec__ca_fdata structure.

EC__BADARG_BUFFER

The buf pointer is NULL.

EC__BADARG_SIZE

The size pointer is NULL.

EC__NOACCESS

CIFS server indicated that the client is not allowed to access this file in the specified manner (e.g. file was opened for write only).

EC__BADFID

CIFS server indicated that the Ec__ca_conn structure contained a bad file handle. This usually indicates that the conn pointer did not point to valid or initialized data.

EC__LOCKED

The requested bytes to read were locked.

EC__TERM_VIOLATION_UNICODE

Indicates that a Unicode protocol violation occurred.

Ec__ca_fwrite()

Prototype:

```
int Ec__ca_fwrite(int call_mode, Ec__ca_conn *conn,  
    Ec__ca_fdata *f_data, u_int8_t *buf, u_int32_t offset,  
    u_int32_t *size, int magic_num)
```

Purpose:

This function writes to a previously opened file.

Notes:

The file to write to must have been previously successfully opened with write access enabled.

Parameters:

int call_mode: As described above in the “eCIFS General Concepts” section under the “[Blocking and Non-blocking Function Calls](#)” heading, this argument can either be EC__BLOCK or EC__NOBLOCK.

Ec__ca_conn *conn: This argument must point to a valid Ec__ca_conn structure that was *successfully* initialized by the Ec__ca_connectnb function.

Ec__ca_fdata *f_data: This argument must point to a valid Ec__ca_fdata structure that was *successfully* initialized by the Ec__ca_fopen() function. This argument indicates which file you wish to write to.

u_int8_t *buf: Specifies the buffer that is to be written.

u_int32_t offset: Number of bytes to offset into the file (from the beginning) before writing.

u_int32_t *size: A pointer to a 32-bit integer that specifies how many bytes to write to the file. This argument is specified as a pointer because upon *blocking* completion of this function the 32-bit integer pointed to by size will indicate the *actual* amount of data that was written to the file. It should be a rare occurrence for this value to differ from the initial requested value.

If called in non-blocking mode the size parameter will indicate that actual amount of data written *upon callback execution* – the size integer that is pointed to by size here will not be modified.

int magic_num: When this function is called in non-blocking mode this magic number will be passed to the callback function upon function completion. More information can be found in the “eCIFS General Concepts” section above under the “[int magic_num](#)” heading.

Return:**EC__SUCCESS**

Successfully wrote to the file. Please check the size integer to insure all required bytes were actually written.

EC__PENDING

Indicates that the non-blocking request is currently pending. Callback will execute when operation has finished.

EC__TERM_TCP

Indicates that the underlying TCP stack caught an error while trying to send or receive packets.

EC__TERM_TIMEOUT

The CIFS server did not reply to one of the pending requests for 10-20 seconds.

EC__TERM_CPVIOLATION

Indicates that a significant CIFS protocol violation has occurred. This typically results if eCIFS receives heavily malformed or partial packets.

EC__CPVIOLATION

A minor CIFS protocol violation has occurred.

EC__STATUS

The server returned an unknown error class and status in response to an eCIFS packet. Please reference the “eCIFS General Concepts” section above under the “[EC__STATUS and EC__TERM_STATUS Return Codes](#)” heading for more information.

EC__BADARG_BLOCKMODE

The call_mode argument above was invalid.

EC__BADARG_CONN

The conn pointer above does not point to a valid and initialized Ec__ca_conn structure.

EC__BADARG_FDATA

The f_data pointer does not point to a valid and initialized Ec__ca_fdata structure.

EC__BADARG_BUFFER

The buf pointer is NULL.

EC__BADARG_SIZE

The size pointer is NULL.

EC__NOACCESS

CIFS server indicated that the client is not allowed to access this file in the specified manner (e.g. file was opened for read only).

EC__BADFID

CIFS server indicated that the Ec__ca_conn structure contained a bad file handle. This usually indicates that the conn pointer did not point to valid or initialized data.

EC__LOCKED

The file (or a particular byte subsection) was locked.

EC__TERM_VIOLATION_UNICODE

Indicates that a Unicode protocol violation occurred.

Ec__ca_rmfile()

Prototype:

```
int Ec__ca_rmfile(int call_mode, Ec__ca_conn *conn,  
char file[], int magic_num)
```

Purpose:

This function removes (deletes) a file or a set of files from the CIFS server.

Notes:

Parameters:

int call_mode: As described above in the “eCIFS General Concepts” section under the “[Blocking and Non-blocking Function Calls](#)” heading, this argument can either be EC__BLOCK or EC__NOBLOCK.

Ec__ca_conn *conn: This argument must point to a valid Ec__ca_conn structure that was *successfully* initialized by the Ec__ca_connectnb function.

char file[] (UTF-8 OK): Specifies the file or files to delete. If no wildcards are used then only the exact matching file will be deleted. If either the ‘*’ or ‘?’ wildcard is used, any files matching the criteria will be deleted.

int magic_num: When this function is called in non-blocking mode this magic number will be passed to the callback function upon function completion. More information can be found in the “eCIFS General Concepts” section above under the “[int magic_num](#)” heading.

Return:

EC__SUCCESS

Successfully deleted the file(s).

EC__PENDING

Indicates that the non-blocking request is currently pending. Callback will execute when operation has finished.

EC__TERM_TCP

Indicates that the underlying TCP stack caught an error while trying to send or receive packets.

EC__TERM_TIMEOUT

The CIFS server did not reply to one of the pending requests for 10-20 seconds.

EC__TERM_CPVIOLATION

Indicates that a significant CIFS protocol violation has occurred. This typically results if eCIFS receives heavily malformed or partial packets.

EC__CPVIOLATION

A minor CIFS protocol violation has occurred.

EC__STATUS

The server returned an unknown error class and status in response to an eCIFS packet. Please reference the “eCIFS General Concepts” section above under the [“EC_STATUS and EC_TERM_STATUS Return Codes”](#) heading for more information.

EC__BADARG_BLOCKMODE

The call_mode argument above was invalid.

EC__BADARG_CONN

The conn pointer above does not point to a valid and initialized Ec__ca_conn structure.

EC__BADARG_FNAME

The file name is invalid (either NULL, too long, or bad UTF-8).

EC__NOSUCHFILE

The CIFS server indicated that no such file(s) actually existed.

EC__NOACCESS

CIFS server indicated that the client is not allowed to delete some or all of the files specified.

EC__TERM_VIOLATION_UNICODE

Indicates that a Unicode protocol violation occurred.

Ec__ca_mkdir()

Prototype:

```
int Ec__ca_mkdir(int call_mode, Ec__ca_conn *conn,  
char dir[], int magic_num)
```

Purpose:

This function creates a directory.

Notes:

Parameters:

int call_mode: As described above in the “eCIFS General Concepts” section under the “[Blocking and Non-blocking Function Calls](#)” heading, this argument can either be EC__BLOCK or EC__NOBLOCK.

Ec__ca_conn *conn: This argument must point to a valid Ec__ca_conn structure that was *successfully* initialized by the Ec__ca_connectnb function.

char dir[] (UTF-8 OK): Specifies the directory name to create. This name cannot exceed 2256 bytes.

int magic_num: When this function is called in non-blocking mode this magic number will be passed to the callback function upon function completion. More information can be found in the “eCIFS General Concepts” section above under the “[int magic_num](#)” heading.

Return:

EC__SUCCESS

Successfully created the directory.

EC__PENDING

Indicates that the non-blocking request is currently pending. Callback will execute when operation has finished.

EC__TERM_TCP

Indicates that the underlying TCP stack caught an error while trying to send or receive packets.

EC__TERM_TIMEOUT

The CIFS server did not reply to one of the pending requests for 10-20 seconds.

EC__TERM_CPVIOLATION

Indicates that a significant CIFS protocol violation has occurred. This typically results if eCIFS receives heavily malformed or partial packets.

EC__CPVIOLATION

A minor CIFS protocol violation has occurred.

EC__STATUS

The server returned an unknown error class and status in response to an eCIFS packet. Please reference the “eCIFS General Concepts” section above under the

[“EC__STATUS and EC__TERM_STATUS Return Codes”](#) heading for more information.

EC__BADARG_BLOCKMODE

The call_mode argument above was invalid.

EC__BADARG_CONN

The conn pointer above does not point to a valid and initialized Ec__ca_conn structure.

EC__BADARG_SRCDIR

The directory name is invalid (either NULL, too long, or bad UTF-8).

EC__NOACCESS

Denotes one or more of the following problems: directory already exists, illegal directory name, path for directory is bad. See EC__FILEEXISTS notes below.

EC__FILEEXISTS

The directory to be created already exists. Unfortunately, EC__NOACCESS can also indicate this and there is no way to return EC__FILEEXISTS exclusively for this situation. Call Ec__ca_dirlist() to obtain the most accurate data.

EC__INVNAME

CIFS server indicated that the directory name was invalid. EC__NOACCESS can also indicate this problem.

EC__TERM_VIOLATION_UNICODE

Indicates that a Unicode protocol violation occurred.

Ec__ca_rmdir()

Prototype:

```
int Ec__ca_rmdir(int call_mode, Ec__ca_conn *conn,  
char dir[], int magic_num)
```

Purpose:

This function removes a directory.

Notes:

The directory to be deleted **MUST** be empty. There is no way to indicate a recursive directory delete or delete directories that are not empty.

Parameters:

int call_mode: As described above in the “eCIFS General Concepts” section under the “[Blocking and Non-blocking Function Calls](#)” heading, this argument can either be EC__BLOCK or EC__NOBLOCK.

Ec__ca_conn *conn: This argument must point to a valid Ec__ca_conn structure that was *successfully* initialized by the Ec__ca_connectnb function.

char dir[] (UTF-8 OK): Specifies the directory name to delete. This name cannot exceed 2256 bytes.

int magic_num: When this function is called in non-blocking mode this magic number will be passed to the callback function upon function completion. More information can be found in the “eCIFS General Concepts” section above under the “[int magic_num](#)” heading.

Return:

EC__SUCCESS

Successfully deleted the directory.

EC__PENDING

Indicates that the non-blocking request is currently pending. Callback will execute when operation has finished.

EC__TERM_TCP

Indicates that the underlying TCP stack caught an error while trying to send or receive packets.

EC__TERM_TIMEOUT

The CIFS server did not reply to one of the pending requests for 10-20 seconds.

EC__TERM_CPVIOLATION

Indicates that a significant CIFS protocol violation has occurred. This typically results if eCIFS receives heavily malformed or partial packets.

EC__CPVIOLATION

A minor CIFS protocol violation has occurred.

EC__STATUS

The server returned an unknown error class and status in response to an eCIFS packet. Please reference the “eCIFS General Concepts” section above under the [“EC__STATUS and EC__TERM_STATUS Return Codes”](#) heading for more information.

EC__BADARG_BLOCKMODE

The call_mode argument above was invalid.

EC__BADARG_CONN

The conn pointer above does not point to a valid and initialized Ec__ca_conn structure.

EC__BADARG_SRCDIR

The directory name is invalid (either NULL, too long, or bad UTF-8).

EC__NOSUCHFILE

Denotes one or more of the following problems: directory does not exist, the specified directory was a file. See the EC__NOACCESS note below.

EC__NOACCESS

Denotes one or more of the following problems: directory permissions do not allow the delete, the specified directory was a file, directory is not empty. Note that the EC__NOSUCHFILE return code can also indicate that the specified directory is a file. If more information is needed please call Ec__ca_dirlist().

EC__TERM_VIOLATION_UNICODE

Indicates that a Unicode protocol violation occurred.

Ec__ca_setattr()

Prototype:

```
int Ec__ca_setattr(int call_mode, Ec__ca_conn *conn,  
    char name[], u_int16_t attr, int magic_num)
```

Purpose:

This function changes the attributes of a given directory or file.

Notes:

Parameters:

int call_mode: As described above in the “eCIFS General Concepts” section under the “[Blocking and Non-blocking Function Calls](#)” heading, this argument can either be EC_BLOCK or EC_NOBLOCK.

Ec__ca_conn *conn: This argument must point to a valid Ec__ca_conn structure that was *successfully* initialized by the Ec__ca_connectnb function.

char name[] (UTF-8 OK): Specifies the name of the file or directory to be altered. The length of this character string cannot exceed 2256 bytes.

u_int16_t attr: Specifies the new attributes that the file or directory will have. This is specified by bitwise OR’ing any of the following values together:

```
EC_READONLY  
EC_HIDDEN  
EC_SYSTEM  
EC_ARCHIVE
```

For example, to make a file or directory read only and hidden *attr* should be set to EC_READONLY | EC_HIDDEN.

int magic_num: When this function is called in non-blocking mode this magic number will be passed to the callback function upon function completion. More information can be found in the “eCIFS General Concepts” section above under the “[int magic_num](#)” heading.

Return:

EC_SUCCESS

Successfully changed the file or directory attributes.

EC_PENDING

Indicates that the non-blocking request is currently pending. Callback will execute when operation has finished.

EC_TERM_TCP

Indicates that the underlying TCP stack caught an error while trying to send or receive packets.

EC__TERM_TIMEOUT

The CIFS server did not reply to one of the pending requests for 10-20 seconds.

EC__TERM_CPVIOLATION

Indicates that a significant CIFS protocol violation has occurred. This typically results if eCIFS receives heavily malformed or partial packets.

EC__CPVIOLATION

A minor CIFS protocol violation has occurred.

EC__STATUS

The server returned an unknown error class and status in response to an eCIFS packet. Please reference the “eCIFS General Concepts” section above under the [“EC__STATUS and EC__TERM_STATUS Return Codes”](#) heading for more information.

EC__BADARG_BLOCKMODE

The call_mode argument above was invalid.

EC__BADARG_CONN

The conn pointer above does not point to a valid and initialized Ec__ca_conn structure.

EC__BADARG_FNAME

The name character string is either NULL, too long, or bad UTF-8.

EC__BADARG_ATTR

The u_int16_t attr argument is not a combination of EC__READONLY, EC__HIDDEN, EC__SYSTEM, and EC__ARCHIVE.

EC__NOSUCHFILE

Indicates that either the file or directory did not exist.

EC__NOACCESS

Indicates that access to the file or directory was denied. This is commonly due to either sharing violations (e.g. another process currently is accessing this file and won't allow an attribute change) or permission problems.

EC__TERM_VIOLATION_UNICODE

Indicates that a Unicode protocol violation occurred.

Ec__ca_fsquery()

Prototype:

```
int Ec__ca_fsquery(int call_mode, Ec__ca_conn *conn,  
                  Ec__ca_fsinfo *fsinfo, int magic_num)
```

Purpose:

This function queries the remote CIFS filesystem that the share resides on. The total Kilobytes (KB) on the remote disk and available Kilobytes (KB) on the remote disk are returned.

Notes:

The returned KB totals are not always 100% percent accurate. They should be used only as a ballpark estimate of total and available remote disk space. 1 KB in these measurements is equal to 1024 bytes, NOT 1000 bytes.

Parameters:

int call_mode: As described above in the “eCIFS General Concepts” section under the “[Blocking and Non-blocking Function Calls](#)” heading, this argument can either be EC_BLOCK or EC_NOBLOCK.

Ec__ca_conn *conn: This argument must point to a valid Ec__ca_conn structure that was *successfully* initialized by the Ec__ca_connectnb function.

Ec__ca_fsinfo *fsinfo: In blocking mode a pointer to an Ec__ca_fsinfo struct must be passed in for the function to fill out. In non-blocking mode there is no reason to pass in a pointer here and NULL can be used. Upon callback execution a filled out Ec__ca_fsinfo struct will returned to the programmer. The Ec__ca_fsinfo struct is detailed below.

```
typedef struct {  
    u_int32_t total_KB;  
    u_int32_t avail_KB;  
} Ec__ca_fsinfo;
```

The total_KB and avail_KB fields indicate the total KB on disk and the available KB on disk respectively.

int magic_num: When this function is called in non-blocking mode this magic number will be passed to the callback function upon function completion. More information can be found in the “eCIFS General Concepts” section above under the “[int magic_num](#)” heading.

Return:

EC_SUCCESS

Successfully queried the remote filesystem and filled in the Ec__ca_fsinfo struct.

EC__PENDING

Indicates that the non-blocking request is currently pending. Callback will execute when operation has finished.

EC__TERM_TCP

Indicates that the underlying TCP stack caught an error while trying to send or receive packets.

EC__TERM_TIMEOUT

The CIFS server did not reply to one of the pending requests for 10-20 seconds.

EC__TERM_CPVIOLATION

Indicates that a significant CIFS protocol violation has occurred. This typically results if eCIFS receives heavily malformed or partial packets.

EC__CPVIOLATION

A minor CIFS protocol violation has occurred.

EC__STATUS

The server returned an unknown error class and status in response to an eCIFS packet. Please reference the “eCIFS General Concepts” section above under the [“EC__STATUS and EC__TERM_STATUS Return Codes”](#) heading for more information.

EC__BADARG_BLOCKMODE

The call_mode argument above was invalid.

EC__BADARG_CONN

The conn pointer above does not point to a valid and initialized Ec__ca_conn structure.

EC__BADARG_FSINFO

Blocking mode was specified and the fsinfo pointer was NULL.

EC__NOACCESS

The filesystem query failed due to access issues.

EC__TERM_VIOLATION_UNICODE

Indicates that a Unicode protocol violation occurred.

Ec__ca_rename()

Prototype:

```
int Ec__ca_rename(int call_mode, Ec__ca_conn *conn,  
char old_name[], char new_name[], int magic_num)
```

Purpose:

This function renames a file or directory on the remote CIFS server.

Notes:

The CIFS server will typically fail the rename request if “new_name” already exists as either a file or directory. However, when the CIFS server is a Samba Unix server and the old_name is a directory and the new_name is an already existing empty directory the server will copy the old_name over the new_name (no error will be returned).

Parameters:

int call_mode: As described above in the “eCIFS General Concepts” section under the [“Blocking and Non-blocking Function Calls”](#) heading, this argument can either be EC_BLOCK or EC_NOBLOCK.

Ec__ca_conn *conn: This argument must point to a valid Ec__ca_conn structure that was *successfully* initialized by the Ec__ca_connectnb function.

char old_name[] (UTF-8 OK): This string specifies the name of the file or directory that is to be renamed. This string cannot exceed 2256 bytes in length.

char new_name[] (UTF-8 OK): This string specifies the new name of the file or directory. This string cannot exceed 2256 bytes in length.

int magic_num: When this function is called in non-blocking mode this magic number will be passed to the callback function upon function completion. More information can be found in the “eCIFS General Concepts” section above under the [“int magic_num”](#) heading.

Return:

EC_SUCCESS

Successfully renamed the file or directory.

EC_PENDING

Indicates that the non-blocking request is currently pending. Callback will execute when operation has finished.

EC_TERM_TCP

Indicates that the underlying TCP stack caught an error while trying to send or receive packets.

EC_TERM_TIMEOUT

The CIFS server did not reply to one of the pending requests for 10-20 seconds.

EC__TERM_CPVIOLATION

Indicates that a significant CIFS protocol violation has occurred. This typically results if eCIFS receives heavily malformed or partial packets.

EC__CPVIOLATION

A minor CIFS protocol violation has occurred.

EC__STATUS

The server returned an unknown error class and status in response to an eCIFS packet. Please reference the “eCIFS General Concepts” section above under the [“EC__STATUS and EC__TERM_STATUS Return Codes”](#) heading for more information.

EC__BADARG_BLOCKMODE

The call_mode argument above was invalid.

EC__BADARG_CONN

The conn pointer above does not point to a valid and initialized Ec__ca_conn structure.

EC__BADARG_FNAME

Indicates that either new_name or old_name was too long, NULL, or bad UTF-8.

EC__NOSUCHFILE

Indicates one or more of the following conditions: file or directory to rename does not exist, new_name contains invalid characters.

EC__NOACCESS

Indicates one or more of the following conditions: old_name cannot be modified (invalid permissions or it is currently in use), new_name already exists, new_name contains invalid characters. Calling Ec__ca_dirlist() can help to deduce if the problem is “new_name already exists”.

EC__FILEEXISTS

The new_name file or directory already exists.

EC__TERM_VIOLATION_UNICODE

Indicates that a Unicode protocol violation occurred.

Ec__ca_dirlist()

Prototype:

```
int Ec__ca_dirlist(int call_mode, Ec__ca_conn *conn,
                  Ec__ca_search *search, Ec__ca_listmode *mode,
                  char src_dir[], char filter[], uint8_t *buf,
                  uint32_t bufsize, uint32_t *hits, int magic_num)
```

Purpose:

This function obtains the contents of a directory on the CIFS server.

Notes:

This function can be used to start a directory listing *or* continue a previous directory listing. Because the caller does not know how many entries may exist in a directory it is impossible to know how large of a buffer to pass in to `Ec__ca_dirlist()`. Therefore it is possible to pick up a search where it left off. In this manner the programmer does not always need to utilize a very large buffer and can instead call `dirlist()`, parse and manipulate the results, and then call `dirlist()` again starting from where it left off.

Upon successful completion of this function, the `buf` argument will point to the raw data that was returned from the directory listing which executed on the CIFS server. To help parse this raw data into meaningful file information, the `Ec__ca_dirparse()` function should be used.

CIFS servers will typically return the “.” and “..” directory listings when the filter (details below) is set to “*” or “*.*”. However, the root directory listing of the share (i.e. “\” as the `src_dir` argument) of *some* CIFS servers will not indicate these two entries.

Parameters:

int call_mode: As described above in the “eCIFS General Concepts” section under the “[Blocking and Non-blocking Function Calls](#)” heading, this argument can either be `EC__BLOCK` or `EC__NOBLOCK`.

Ec__ca_conn *conn: This argument must point to a valid `Ec__ca_conn` structure that was *successfully* initialized by the `Ec__ca_connectnb` function.

Ec__ca_search *search: This pointer must reference an `Ec__ca_search` struct. This struct is used by `dirlist` to allow programmers to continue where a previous search left off (in essence this struct is a search handle). When `mode` (described below) is set to `EC__SEARCH_NEW`, this search struct is filled in with new search data. When `mode` is set to `EC__SEARCH_PREV`, this search struct **MUST** reference a previously filled in search struct (and `dirlist` will continue the directory listing that was started before).

In blocking mode the programmer must always pass in a search struct pointer when call `dirlist()`.

In non-blocking mode, the programmer only needs to pass in a search struct pointer when continuing a previous search (i.e. mode is set to EC__SEARCH_PREV). In non-blocking mode with mode set to EC__SEARCH_NEW the search argument can be NULL (if it is not NULL the pointer will not be utilized in any way). As in other non-blocking function calls, if a search struct pointer is passed in the actual struct will NOT be modified. The search struct will only be made available to the programmer *upon callback execution*. The search struct pointed to in the actual function call will not be changed ever in non-blocking mode – it will only be read (not modified) if mode is equal to EC__SEARCH_PREV.

Ec__ca_listmode *mode: Upon calling dirlist, mode must always be set to either EC__SEARCH_NEW or EC__SEARCH_PREV and then passed into dirlist by reference. For example:

```
Ec__ca_listmode mode = EC__SEARCH_NEW;  
// call dirlist() with &mode for the Ec__ca_listmode pointer
```

Setting mode equal to EC__SEARCH_NEW indicates that dirlist should begin a new directory listing for the specified folder. Setting mode equal to EC__SEARCH_PREV indicates that dirlist() should pick up where it left off. Dirlist() knows what search the programmer wants to continue by utilizing the Ec__ca_search struct pointer that is passed in above.

In addition to indicating whether or not to start a new dirlist or continue an old one, dirlist also returns information to the programmer.

When called in ***blocking mode***, the Ec__ca_listmode type will be modified upon successful execution of the function to equal either EC__SEARCH_MORE or EC__SEARCH_END. EC__SEARCH_MORE indicates that there are more directory entries to be listed. This also indicates that the Ec__ca_search struct (above) was filled out and that it is OK to call dirlist() again with mode set to EC__SEARCH_PREV (insuring that the corresponding search struct above is also passed in again). If mode is set to EC__SEARCH_END it indicates that there are no more entries from the directory to be returned and it is *illegal* to call dirlist() again with mode set to EC__SEARCH_PREV and the same search struct.

When called in ***non-blocking mode***, mode must still be set to either EC__SEARCH_NEW or EC__SEARCH_PREV upon calling the function, but the value will not be set to either EC__SEARCH_MORE or EC__SEARCH_END when the function returns (as the actual dirlist event has not yet occurred). However, upon callback execution the programmer will be passed a copied Ec__ca_listmode data type that will be set to either EC__SEARCH_MORE or EC__SEARCH_END.

char src_dir[] (UTF-8 OK): This string indicates the directory that the programmer wishes to obtain a content listing from. This string must have a final ‘\’ character as

the last character of the string (directly before the NULL terminating character). Note that in 'C' and 'C++' the escape character in string literals is '\ ' – therefore to indicate that the programmer wants a listing from the temp directory the *src_dir* string would be "\\temp\\". When *Ec__ca_listmode* is set to *EC__SEARCH_PREV* this argument can be NULL and the listing will be continued with the previous *src_dir* argument.

char filter[] (UTF-8 OK): This string allows a programmer to filter directory listings. A filter must always be present, unless *Ec__ca_listmode* is set to *EC__SEARCH_PREV* in which case the search is continued with the previous filter. Example filters follow: "*.*", "*", "asdf?.txt". An exact file match can also be requested by setting filter to a specific file name with no wildcards, as in "myfile.pdf". This is an easy way to obtain attributes and filesize for a single file.

u_int8_t *buf: This points to a byte buffer that eCIFS will store the directory hits in. The size of this buffer is specified below and must be at least 400 bytes long. When calling *dirlist()* in non-blocking mode insure that this buffer is allocated from non-volatile memory (i.e. be careful if you call this in non-blocking mode and pass in stack memory for this argument).

u_int32_t bufsize: This argument indicates the size of the above buffer in bytes. This number must be greater than 400 or an error will result.

u_int32_t *hits: This argument must point to a valid 32-bit integer. Upon successful termination of *dirlist()* this integer will contain the number of file/directory entries that were found and copied into the *buf* memory chunk passed in above. When *dirlist()* is called in non-blocking mode this argument can be NULL and will not be utilized in any way whether it is passed in or not.

int magic_num: When this function is called in non-blocking mode this magic number will be passed to the callback function upon function completion. More information can be found in the "eCIFS General Concepts" section above under the "[int magic_num](#)" heading.

Return:

EC__SUCCESS

Successfully obtained the directory listing.

EC__PENDING

Indicates that the non-blocking request is currently pending. Callback will execute when operation has finished.

EC__TERM_TCP

Indicates that the underlying TCP stack caught an error while trying to send or receive packets.

EC__TERM_TIMEOUT

The CIFS server did not reply to one of the pending requests for 10-20 seconds.

EC__TERM_CPVIOLATION

Indicates that a significant CIFS protocol violation has occurred. This typically results if eCIFS receives heavily malformed or partial packets.

EC__CPVIOLATION

A minor CIFS protocol violation has occurred.

EC__STATUS

The server returned an unknown error class and status in response to an eCIFS packet. Please reference the “eCIFS General Concepts” section above under the [“EC__STATUS and EC__TERM_STATUS Return Codes”](#) heading for more information.

EC__BADARG_BLOCKMODE

The call_mode argument above was invalid.

EC__BADARG_CONN

The conn pointer above does not point to a valid and initialized Ec__ca_conn structure.

EC__BADARG_SEARCH

Ec__ca_search argument could not be NULL under the specified conditions.

EC__BADARG_SEARCHMODE

The list mode argument was invalid, or the Ec__ca_search argument was not previously initialized and should have been.

EC__BADARG_SRCDIR

The src_dir string was invalid.

EC__BADARG_FILTER

The filter string was invalid.

EC__BADARG_BUFFER

The buf pointer was invalid.

EC__BADARG_SIZE

The bufsize argument was invalid.

EC__BADARG_HITS

The hits argument was invalid.

EC__NOACCESS

The server denied access.

EC__NOSUCHFILE

Some CIFS servers will return this to indicate that there were no hits or matches for this directory listing. Therefore this should not necessarily be considered an error, just that no directory contents existed, or no directory contents matched the specified filter.

However, this error code can also be used to indicate that the specified src_dir was either a file or completely non-existent. Issuing a dirlist() call on the parent directory of the current src_dir and checking file attributes can help deduce the problem.

EC__NOMATCH

No matches were found.

EC__BUF_TOOSMALL

The server indicated that the buffer that was passed in was not big enough. This is different from EC__BADARG_SIZE in that the server returns this error. This error should not be generated if the 400-byte minimum was met.

EC__TERM_VIOLATION_UNICODE

Indicates that a Unicode protocol violation occurred.

Ec__ca_dirparse()

Prototype:

```
int Ec__ca_dirparse(u_int8_t *dirdata,  
                   Ec__ca_dirent *entry, u_int32_t entry_num)
```

Purpose:

This function parses the buffer built by `Ec__ca_dirlist` and populates an `Ec__ca_dirent` structure with a single directory or file listing from that buffer. The programmer can then easily extrapolate data from the `Ec__ca_dirent` structure, utilize the data in any way, and then get the next file or directory entry from the raw buffer by calling `Ec__ca_dirparse()` again.

Notes:

Parameters:

u_int8_t *dirdata: This argument must point to a buffer that was filled out by the `Ec__ca_dirlist()` function on a SUCCESSFUL CALL. If `dirdata` points to anything but a well-formed `Ec__ca_dirlist` buffer this code could crash.

Ec__ca_dirent *entry: This must point to a valid `Ec__ca_dirent` structure. Once the `Ec__ca_dirparse()` function successfully executes, the programmer can directly view the fields of this structure (which `Ec__ca_dirparse` fills out) in order to determine specific characteristics of a particular file or directory. The structure is detailed below.

```
typedef struct {  
    Ec__pu_time creation_time;  
    Ec__pu_time last_access_time;  
    Ec__pu_time last_write_time;  
    Ec__pu_time attr_change_time;  
    u_int32_t filesize;  
    u_int16_t attr;  
    u_int32_t namelength;  
    char name[EC__MAXFNAME+1];  
} Ec__ca_dirent;
```

The first four struct members specify the various times associated with the file. Note that not all operating systems track these times and therefore some might not be set.

u_int32_t filesize: The `filesize` member indicates the size of the file in bytes.

u_int16_t attr: The `attr` integer specifies the attributes of this particular file or directory. These can be tested against the following bit masks to see if a particular attribute is present:

EC__READONLY
EC__HIDDEN
EC__SYSTEM
EC__VOLUME
EC__DIR
EC__ARCHIVE

Note that not all operating systems support all these attributes. For example, Unix based operating systems indicate hidden files by preceding the filename with a '.' character. Setting the EC__HIDDEN attribute on a Unix-based CIFS server will do nothing.

u_int32_t namelength: Specifies the length of the name below.

char name: A NULL terminated ASCII string that contains the name of the file or directory. If Unicode was enabled on this connection, the name will be a valid UTF-8 string with NULL termination.

u_int32_t entry_num: This specifies which file or directory from the `u_int8_t *dirdata` buffer to parse into the `Ec__ca_dirent` entry struct. This number must be less than the `u_int32_t hits` argument that was filled out by the `Ec__ca_dirlist()` function documented above. For example, if `Ec__ca_dirlist()` filled in the `hits` integer as 15, the acceptable range for the `entry_num` argument would be 0-14. By looping through each of these numbers, the programmer would be able to extract data on every single file and directory that was returned by `Ec__ca_dirlist()`.

Return:

EC__SUCCESS

Successfully parsed a file or directory entry in the `Ec__ca_dirent` struct.

EC__BADARG_HITS

The `entry_num` was not less than the `hits` integer specified by `Ec__ca_dirlist()`.

EC__ENTRYHUGE

This rare return code indicates that the `filesize` was greater than a 32 bit number.

Although this is an error code, the `Ec__ca_dirent` struct should still be filled out for the file. However, the `filesize` member will be incorrect as this number won't be able to contain the total file size.

EC__BADARG_DIRDATA

The `dirdata` buffer pointer was invalid.

EC__BADARG_DIRDATA

The `dirdata` buffer pointer was invalid.

Ec__ca_checkconn()

Prototype:

```
int Ec__ca_checkconn(Ec__ca_conn *conn)
```

Purpose:

Checks to see if the specified connection is still alive.

Notes:

If the connection is dead, this function will return an `EC__TERM_XXXX` code that correlates to the error that killed it. This function is often useful when a non-blocking callback indicates a status of `EC__FAIL`. In this case, the programmer should call the `Ec__ca_checkconn()` function back in the API calling thread (not in the callback) in order to obtain the real termination error code.

Parameters:

Ec__ca_conn *conn: This argument must point to a valid `Ec__ca_conn` structure that was *successfully* initialized by the `Ec__ca_connectnb` function. This argument specifies the connection to check.

Return:

`EC__SUCCESS`

The connection is still active.

`EC__BADARG_CONN`

The `conn` pointer above does not point to a valid and initialized `Ec__ca_conn` structure.

`EC__TERM_XXXX` (any of the possible termination error codes – see `ret_code.h`)

Because any non-blocking function could have terminated the CIFS connection, any of the possible termination error codes could be discovered when checking the status of the connection.

Ec__ca_endconn()

Prototype:

```
int Ec__ca_endconn(Ec__ca_conn *conn)
```

Purpose:

Terminates the connection to the server and invalidates the passed in Ec__ca_conn connection handle.

Notes:

Parameters:

Ec__ca_conn *conn: This argument must point to a valid Ec__ca_conn structure that was *successfully* initialized by the Ec__ca_connectnb function. Upon successful execution of this function the connection represented by conn will be terminated.

Return:

EC__SUCCESS

Successfully ended the CIFS server connection.

EC__BADARG_CONN

The conn pointer above does not point to a valid and initialized Ec__ca_conn structure.

Ec__strerror()

Prototype:

```
char * Ec__strerror(int ret_code)
```

Purpose:

This function is passed an integer that was returned from a previous eCIFS function execution. This function then returns a text string that describes the return code in more detail.

Notes:

If eCIFS was built with the macro `SMALL_FOOTPRINT` defined, this function will always return "small build: no strings". In addition, the function can return either "no description" or "UNKNOWN_ERROR" if the return code was not described or does not exist.

Parameters:

int ret_code: This argument should be the return code of a previously executed eCIFS API function.

Return:

char *: Used to return a string which describes the return code in more detail.

API documentation (browsing)

Browsing is CIFS terminology for network resource discovery: the process of determining other computing resources on a network. The three types of network resources that eCIFS can discover are workgroups/domains, CIFS servers, and available share lists. Each of these resources is discussed below.

Note that for the browsing API, there is no distinction between termination style error codes and regular error codes. Because there is no connection handle there is no need for this division of error codes. In addition, none of the browsing API functions can be non-blocking. All of these functions will block until the operation successfully completes or an error condition is encountered.

There is currently no Unicode support for the browsing API functions.

Workgroups/domains: CIFS enabled computers are commonly grouped together into functional workgroups or domains. For example, if a publishing company had a network setup with CIFS file sharing enabled, it might be useful to classify the many computers into functional divisions. There would be an “authors” group, an “editors” group, and a “sales” group. Every PC on the company network would be grouped into one or more of these divisions. With a setup like this, users of the network could quickly determine which computers belonged to which groups, and this would help to organize the information contained in the network. For example, if an author needed to see a list of CIFS file servers related only to the “authors” workgroup she could do this. Likewise, an editor looking for a list of printers for his division would be able to obtain this list easily.

The eCIFS browsing API allows the programmer to obtain a list of the available domains and workgroups on the network. eCIFS makes no distinction between domains and workgroups; they are both logical groups of computers.

Computer lists: As indicated above, each workgroup or domain has a list of computers that are members of the grouping. Computers can join and leave workgroups at anytime, but typically join upon boot-up and leave upon shutdown. The eCIFS browsing API allows the programmer to obtain a list of computers that belong to a given workgroup or domain.

Share lists: Computers that belong to a domain or workgroup frequently share a directory for other computers to access via the CIFS protocol. These shared directories are sometimes referred to simply as “shares”. Shares form the core file resource that the eCIFS API functions (documented above) work with extensively.

The eCIFS browsing API allows the programmer to obtain a list of shares that an individual computer is “exporting”, or making available. Once this list is obtained, specific file resources and directory listings can be obtained by utilizing the main eCIFS API detailed above.

Ec__ca_browsedomains()

Prototype:

```
int Ec__ca_browsedomains(u_int8_t *dom_buf,  
    int *bufsize, Ec__nu_name *calling_client,  
    char *calling_domain, int *numdomains)
```

Purpose:

This function obtains a list of the domains and workgroups that exist on the network.

Notes:

Parameters:

u_int8_t *dom_buf: This argument specifies where the raw domain/workgroup list should be stored. The buffer pointed to by `dom_buf` must be at least 574 bytes long.

int *bufsize: The integer that `bufsize` points to must indicate the size of the buffer that `dom_buf` points to. This value must be at least 574.

Ec__nu_name *calling_client: This structure (to be filled out by the programmer) is used to indicate the NetBIOS name of the client that will execute the `get domains/workgroups` command on the network. Put another way, the name pointed to by `calling_client` will specify the NetBIOS name of the system which the eCIFS code is executing on. This structure is detailed [here](#) in Appendix A.

The NetBIOS name of the client is typically not important. For this reason, `calling_client` can be set to NULL and a default NetBIOS name (“eCIFS_CLIENT”) will be used.

char *calling_domain: This argument can be specified in order to reduce the number of packets that are sent on the network obtaining the list of domains and workgroups. If the programmer is aware of any current workgroup or domain this argument can be used to specify the known domain or workgroup. With this knowledge, eCIFS can effectively bypass some of the packet transactions and still obtain a complete list of *all* the workgroups and domains.

This argument is not necessary and only reduces network traffic modestly. NULL should typically be used for this argument.

int *numdomains: This argument must point to an integer. Upon successful execution of this function the integer pointed to by `numdomains` will be equal to the number of workgroups and domains that are currently available in raw format in the `dom_buf` buffer.

Return:

EC__SUCCESS

Successfully obtained a list of workgroups/domains and stored them in `dom_buf`.

EC__BADARG_OPTS

The numdomains pointer was NULL.

EC__BUF_TOOSMALL

The buffer was either equal to NULL or not big enough.

EC__DOMAIN_NOTFOUND

No domain found on network.

EC__NODESTATUS

Unable to reverse-lookup the master browser IP.

EC__BADARG_DOMAIN

The passed in domain string was invalid.

EC__BADARG_CALLINGCLIENT

The passed in calling_client NetBIOS name was invalid.

EC__NETWORK

Could not bind/send with UDP protocol.

EC__NOLMBBROWSER

Local master browser did not respond.

EC__BKUPBROWSER

Backup browser query failed.

Ec__ca_parse_browsedomainlist()

Prototype:

```
int Ec__ca_parse_browsedomainlist(  
    uint8_t *dom_buf, int index, char **domain)
```

Purpose:

This function takes the workgroup/domain buffer and parses out a single workgroup/domain from it.

Notes:

This function allows the programmer to obtain meaningful information from the `Ec__ca_browsedomains` function. The `dom_buf` filled out by `Ec__ca_browsedomains` is passed in, along with the index of the workgroup/domain that the programmer is interested in. The function walks through the `dom_buf` buffer until it finds the pointer to the particular workgroup/domain string that correlates to the passed in index integer. It then passes this string pointer back via the `domain` argument.

Note that the `dom_buf` buffer must have been obtained from a *successful* execution of the `Ec__ca_browsedomains()` function above. In addition, the index integer passed in here must be less than the `numdomains` integer that was filled out by the `Ec__ca_browsedomains()` function. For example, if `numdomains` was set to 15 by `Ec__ca_browsedomains`, then `Ec__ca_parse_browsedomainlist` could be called in a loop with `index` varying from 0 to 14. In this manner, the programmer could obtain a string pointer for each of the workgroups/domains in the `dom_buf` buffer.

The character pointer which is returned by `Ec__ca_parse_browsedomainlist` via `char **domain` points directly into the `dom_buf` buffer. If the `dom_buf` buffer is free'd or exits scope, any character pointer obtained from the `domain` argument will reference invalid memory.

Parameters:

uint8_t *dom_buf: This argument must point to a buffer that was *successfully* filled out by the `Ec__ca_browsedomains()` function.

int index: Specifies which workgroup/domain from the above buffer that the programmer wants a string pointer to.

char **domain: This pointer to a character pointer is used to return a character pointer to the programmer. Upon successful execution of this function, `*domain` will contain a pointer to a workgroup/domain character string. The programmer should declare a `char` pointer, and then dereference this pointer to pass into this function. I.e. `char *my_workgrp; Ec__ca_parse_browsedomainlist(buf, 3, &my_workgrp);`

Return:

EC__SUCCESS

Successfully placed a pointer to a workgroup/domain in *domain.

EC__BADARG_OPTS

The dom_buf argument was NULL.

EC__CORRUPT

The buffer was corrupt or the index specified was greater than or equal to the numdomains integer specified by Ec__ca_browseedomains.

Ec__ca_browsecomputers()

Prototype:

```
int Ec__ca_browsecomputers(uint8_t *comp_buf,  
    int *bufsize, Ec__nu_name *calling_client,  
    char *calling_domain, int *numcomputers)
```

Purpose:

This function obtains a list of computer names for the specified workgroup/domain.

Notes:

Parameters:

uint8_t *comp_buf: This argument specifies where the raw computer name list should be stored. The buffer pointed to by *comp_buf* must be at least 574 bytes long.

int *bufsize: The integer that *bufsize* points to must indicate the size of the buffer that *comp_buf* points to. This value must be at least 574.

Ec__nu_name *calling_client: This structure (to be filled out by the programmer) is used to indicate the NetBIOS name of the client that will execute the get computer name list command on the network. Put another way, the name pointed to by *calling_client* will specify the NetBIOS name of the system which the eCIFS code is executing on. This structure is detailed [here](#) in Appendix A.

The NetBIOS name of the client is typically not important. For this reason, *calling_client* can be set to NULL and a default NetBIOS name (“eCIFS_CLIENT”) will be used.

char *calling_domain: This argument must specify the workgroup/domain that the programmer wishes to obtain a computer name list for.

int *numcomputers: This argument must point to an integer. Upon successful execution of this function the integer pointed to by *numcomputers* will be equal to the number of computer names that are currently available in raw format in the *comp_buf* buffer for the specified workgroup/domain.

Return:

EC__SUCCESS

Successfully obtained a list of computer names and stored them in *comp_buf*.

EC__BADARG_OPTS

The *numcomputers* pointer was NULL.

EC__BUF_TOOSMALL

The buffer was either equal to NULL or not big enough.

EC__DOMAIN_NOTFOUND

The domain could not be found on the network.

EC__NODESTATUS

Unable to reverse-lookup the master browser IP.

EC__BADARG_DOMAIN

The passed in domain string was invalid.

EC__BADARG_CALLINGCLIENT

The passed in calling_client NetBIOS name was invalid.

EC__NETWORK

Could not bind/send with UDP protocol.

EC__NOLMBROWSER

Local master browser did not respond.

EC__BKUPBROWSER

Backup browser query failed.

Ec__ca_parse_browsecomputerlist()

Prototype:

```
int Ec__ca_parse_browsecomputerlist(  
    uint8_t *comp_buf, int index, char **computername)
```

Purpose:

This function takes the comp_buf computer list buffer and parses out a single computer name entry from it.

Notes:

This function allows the programmer to obtain meaningful information from the Ec__ca_browsecomputers function. The comp_buf filled out by Ec__ca_browsecomputers is passed in, along with the index of the computer name that the programmer is interested in. The function walks through the comp_buf buffer until it finds the pointer to the particular computer name string that correlates to the passed in index integer. It then passes this string pointer back via the computername argument.

Note that the comp_buf buffer must have been obtained from a *successful* execution of the Ec__ca_browsecomputers() function above. In addition, the index integer passed in here must be less than the numcomputers integer which was filled out by the Ec__ca_browsecomputers() function. For example, if numcomputers was set to 15 by Ec__ca_browsecomputers, then Ec__ca_parse_browsecomputerlist could be called in a loop with index varying from 0 to 14. In this manner, the programmer could obtain a string pointer for each of the computer names in the comp_buf buffer.

The character pointer which is returned by Ec__ca_parse_browsecomputerlist via char **computername points directly into the comp_buf buffer. If the comp_buf buffer is free'd or exits scope, any character pointer obtained from the computername argument will reference invalid memory.

Parameters:

uint8_t *comp_buf: This argument must point to a buffer that was *successfully* filled out by the Ec__ca_browsecomputers() function.

int index: Specifies which computer name from the above buffer that the programmer wants a string pointer to.

char **computername: This pointer to a character pointer is used to return a character pointer to the programmer. Upon successful execution of this function, *computername will contain a pointer to a computer name character string. The programmer should declare a char pointer, and then dereference this pointer to pass into this function. I.e. char *my_comp_name; Ec__ca_parse_browsecomputerlist (buf, 3, &my_comp_name);

Return:

EC__SUCCESS

Successfully placed a pointer to a computer name string in *computername.

EC__BADARG_OPTS

The comp_buf argument was NULL.

EC__CORRUPT

The buffer was corrupt or the index specified was greater than or equal to the numcomputers integer specified by Ec__ca_browsecomputers.

Ec__ca_browseshares()

Prototype:

```
int Ec__ca_browseshares(u_int8_t *share_buf, int *bufsize,
    Ec__nu_name *calling_client, Ec__nu_name *called_server,
    Ec__ca_ipinfo *server_ipinfo, int *numshares,
    char username[], char passwd[])
```

Purpose:

This function obtains a list of the available shares on a specified CIFS server.

Notes:

Parameters:

u_int8_t *share_buf: This argument specifies where the raw share info data should be stored. The buffer pointed to by *share_buf* must be at least 574 bytes long.

int *bufsize: The integer that *bufsize* points to must indicate the size of the buffer that *share_buf* points to. This value must be at least 574.

Ec__nu_name *calling_client: This structure (to be filled out by the programmer) is used to indicate the NetBIOS name of the client that will obtain the list of available shares from the CIFS server. Put another way, the name pointed to by *calling_client* will specify the NetBIOS name of the system which the eCIFS code is executing on. This structure is detailed [here](#) in Appendix A.

The NetBIOS name of the client is typically not important. For this reason, *calling_client* can be set to NULL and a default NetBIOS name (“eCIFS_CLIENT”) will be used.

Ec__nu_name *called_server: This argument specifies the name of the CIFS server that the programmer wishes to query for available shares. This structure is detailed [here](#) in Appendix A.

Ec__ca_ipinfo *server_ipinfo: This argument specifies how the CIFS server’s NetBIOS name (specified above) is resolved into an IP address. This struct is detailed [here](#) in Appendix A.

int *numshares: This argument must point to an integer. Upon successful execution of this function the integer pointed to by *numshares* will be equal to the number of shares that the CIFS server is currently exporting.

char username[]: Specifies the username that the programmer wishes to login to the CIFS server as. This argument may be NULL if the server operates in share level security or the programmer wishes to attempt a guest login.

char passwd[]: Specifies the password for the CIFS server. This argument may be NULL if the server does not require a password or guest access is being attempted.

Return:

EC__SUCCESS

Successfully obtained a list of shares from the CIFS server.

EC__BADARG_OPTS

Either numshares, bufsize, called_server, or server_ipinfo was NULL.

EC__BUF_TOOSMALL

Either the buffer is equal to NULL or bufsize is not large enough.

EC__BADPASSWD

The server rejected the specified password.

EC__NB_NOTFOUND

The NetBIOS name of the server could not be resolved to an IP address.

EC__TCP

The TCP connection died or could not be established with the server.

EC__CANT_CONNECT

The connection to the server failed for unspecified reasons.

Ec__ca_parse_browseshares()

Prototype:

```
int Ec__ca_parse_browseshares(u_int8_t *share_buf,  
int index, Ec__ca_shareinfo *shareinfo)
```

Purpose:

This function takes the share_buf share list buffer and parses out a single entry into an Ec__ca_shareinfo structure.

Notes:

This function allows the programmer to obtain meaningful information from the Ec__ca_browseshares function. The share_buf filled out by Ec__ca_browseshares is passed in, along with the index of the share that the programmer is interested in. The function walks through the share_buf buffer until it finds the share data that correlates to the passed in index integer. It then fills in the passed in Ec__ca_shareinfo structure with data for this particular share.

Note that the share_buf buffer must have been obtained from a *successful* execution of the Ec__ca_browseshares() function above. In addition, the index integer passed in here must be less than the numshares integer which was filled out by the Ec__ca_browseshares() function. For example, if numshares was set to 15 by Ec__ca_browseshares, then Ec__ca_parse_browseshares could be called in a loop with index varying from 0 to 14. In this manner, the programmer could obtain the share data for each of the shares in the share_buf buffer.

The Ec__ca_parse_browseshares function fills in the Ec__ca_shareinfo structure with a *shallow copy* of the data from the share_buf. Therefore if the share_buf is deallocated or goes out of scope, any Ec__ca_shareinfo structs that were previously filled in using that share_buf will reference invalid memory.

Parameters:

u_int8_t *share_buf: This argument must point to a buffer that was *successfully* filled out by the Ec__ca_browseshares() function.

int index: Specifies which share from the above buffer that the programmer wants the Ec__ca_shareinfo struct filled out by.

Ec__ca_shareinfo *shareinfo: This pointer must point to a valid Ec__ca_shareinfo structure. Upon successful execution of this function the shareinfo structure will contain information regarding one particular share on the CIFS server. This structure is detailed below:

```
typedef struct {  
    char *sharename;  
    Ec__ca_sharetype type;  
    char *comment;
```

```
} Ec__ca_shareinfo;
```

*char *sharename*: This string pointer contains the name of the share.

Ec__ca_sharetype type: This enumerated value indicates the type of the specified share. Current possible values are listed below:

```
EC__SHRTYPE_FILE (shared resource is a file directory or volume)
EC__SHRTYPE_PRINTER (shared resource is a printer)
EC__SHRTYPE_COMDEV
EC__SHRTYPE_IPC
```

Please note that only shares of type `EC__SHRTYPE_FILE` are suitable to be connected to via the `Ec__ca_connectnb()` function. In addition, there could be other values specified for type that are not one of these 4 values. However, these are the 4 common types that are documented.

*char *comment*: This string pointer will contain the share comment. The share comment is typically a brief sentence or phrase that indicates the general purpose of the share. Many times there is no share comment and this struct member will be set to `NULL`.

Return:

`EC__SUCCESS`

Successfully placed a particular share's information into the `Ec__ca_share info` struct.

`EC__BADARG_OPTS`

Indicates one or more of the following problems:

- Specified index was greater than or equal to the `numshares` argument specified by the `Ec__ca_browseshares` function.
- Specified index was less than zero.
- The `share_info` argument was `NULL`.
- The `share_buf` argument was `NULL`.

`EC__CORRUPT`

The `share_buf` buffer was invalid/corrupt.

Appendix A – Common Structs

Ec__nu_name

This struct holds a NetBIOS computer name (see the CIFS Explained whitepaper for more details on this). The struct is detailed below.

```
typedef struct {  
    char *nb_name;  
    char *scope_id;  
    Ec__nu_serv_type serv_type;  
} Ec__nu_name;
```

char *nb_name: This member must point to the first character of a string that contains a NetBIOS name. This string can contain any number of characters, but only the first 15 will be used to build the NetBIOS name (any characters past the 15th will be ignored). In addition, this string can be uppercase or lowercase but will always be converted to uppercase before being used. Valid characters for a NetBIOS name are listed below:

Sets of chars:

[A-Z]

[a-z]

[0-9]

Individual chars:

!	@	#	\$
%	^	&	(
)	-	'	{
}	.	~	_

char *scope_id: Can either be NULL or point to the first character of a NetBIOS scope ID. The scope ID must conform to standard internet domain name rules (specified in RFC 1034). In short, the string pointed to by *scope_id* must not be longer than 255 characters total, and any individual label within the name cannot exceed 63 characters. For example, in the scope ID “code-serv.codefx.com” there would be three labels, “code-serv”, “codefx”, and “com”. Valid characters for a *scope_id* (and any DNS name in general) are listed below. Please note that a ‘.’ character is allowed, but has special meaning as a label delimiter and is therefore not included in the valid characters list.

Sets of chars:

[A-Z]

[a-z]

[0-9]

Individual chars:

-

Ec_nu_serv_type serv_type: This member specifies the final 16th byte of a computer's NetBIOS name (please see "CIFS Explained" whitepaper). *serv_type* must be set to one of the following enumerated values:

Enumerated value	Meaning
std_workstation_srvc	Typically indicates "normal CIFS client". Use when specifying the client's NetBIOS name.
messenger_service	
ras_server_service	
domain_master_browser	
primary_domain_controller	
master_browser_name	
netdde_service	
fileserver	Typically indicates "normal CIFS server". Use when specifying the server's NetBIOS name.
ras_client_service	
netmon_agent	
netmon_util	

Ec__ca_ipinfo

This struct holds an IP address or details the NetBIOS name resolution method for converting a NetBIOS name into an IP address.

```
typedef struct {  
    u_int32_t ip;  
    Ec__nb_querymode nb_mode;  
    u_int32_t nbns_server;  
    u_int16_t port;  
} Ec__ca_ipinfo;
```

u_int32_t ip: This member can either be set to a 32-bit IP address or set to the #defined token EC__NOIP. If this IP address is set to an actual IP the nb_mode and nbns_server struct members are not checked in any way.

Ec__nb_querymode nb_mode: This member sets the NetBIOS to IP address resolution mode when the ip struct member above is set equal to EC__NOIP. The possible enumerated values are indicated below:

```
EC__NB_BNODE /* Broadcast mode */  
EC__NB_PNODE /* Unicast mode (requires nbns server) */  
EC__NB_MNODE /* Broadcast first, then Unicast */  
EC__NB_HNODE /* Unicast first, then Broadcast (minimizes broadcast traffic) */
```

More information on these resolution modes can be found in the CodeFX whitepaper titled “CIFS Explained”.

u_int32_t nbns_server: If nb_mode above is set to a resolution mode that requires an NBNS (WINS) server, the IP address of the NBNS server must be set here. The following nb_mode values require the NBNS address to be set: EC__NB_PNODE, EC__NB_MNODE, EC__NB_HNODE.

u_int16_t port: This struct member specifies the TCP port that should be utilized to connect to the CIFS server. This value should typically be set to EC__DEFAULT_FILEPORT (equal to 139) because this is the well known port number set aside for CIFS server file services.

Legal Info

eCIFS is a large software library and keeping the documentation in perfect alignment with the source code is a difficult task. In addition it is very likely that bugs exist within the eCIFS software package. For these reasons the eCIFS software package will not always behave as indicated in this document, and may cause catastrophic system failures (e.g. security breaches, system reboots, system lockups, memory leaks...). You have been warned and users of this documentation and software do so at their own risk.

CODEFX DISCLAIMS ALL WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com).