Inter-Office Memorandum

To	Cedar Interest	Date	Oct 1, 1982	
From	Dave Gifford, revised by MDS		Location	Palo Alto
Subject	CIFS Manual (3.4)	Organization	PARC/CSL	

XEROX

Filed on: [Indigo]<Cedar>Documentation>CIFSManual.(tioga press)

Cedar Interim File System

Abstract

The Cedar Interim File System provides basic file storage and directory services in support of the Cedar Programming Environment. This manual describes the Cedar Interim File System's concepts, facilities, and interface in two parts. The first part of the manual is intended for the casual Cedar user and describes basic file system concepts. The second part is intended for programmers who will make direct use of the CIFS programming interface. The appendix contains an example program that uses CIFS.

Part A: Users Guide

A.1 Concepts

CIFS is a *client file system*; it runs with Cedar and provides a unified view of *server file systems* such as IFS, Juniper, and Maxc. Like other file systems, CIFS implements the *file* as the basic object for long-term information storage. In CIFS a file is an array of 512 byte pages. Clients can create, read, alter, and delete files.

Files are cataloged in *directories.* A directory is a set of *entries*, and every entry has an *entry name* that is unique within its directory. An entry describes either a file, a directory, or a link (links will be discussed later). Every entry can include a *comment*, which is a text string that is not interpreted by CIFS. Like files, directories can be created, examined, altered, and deleted by a client. Whenever a directory is created it must be permanently cataloged in a single *parent directory*. This requirement causes directories to be arranged in a tree, and appropriately enough the root of the tree is called

the *root directory*. Every directory and file can be reached from the root directory.

An object's *absolute path name* specifies the way the object is reached from the root. Thus, absolute path names are unambiguous. Examples of absolute path names are:

/indigo/cedar/top/rigging.df!2

/phylum/lad/bookpress/

By convention the leading slash on these names means they are relative to the root directory. Subsequent slashes cause CIFS to walk down the tree of directories.

Directories that are children of the root directory correspond to file servers. This correspondence determines where a child of the root and its descendants are physically stored. Thus, */ iris/mesa* is stored on the file server named *iris.*

A *link* is a directory entry that points to another directory entry. The entry pointed to is called the *target* of the link. Links are named by absolute path names in the same manner as files and directories. An example of a link is:

/ivy/cassatt/history/paint.tioga

target / louvre/renoir/paint.tioga

A reference to */ ivy/cassatt/history/paint.tioga* will be resolved to */ louvre/renoir/paint.tioga* by CIFS. Once a link is established its target can be changed. Thus, links are the basic indirection mechanism of CIFS. CIFS restricts links to point at file entries.

This would be the end of the concepts section except that it is very cumbersome to always type absolute path names. Thus, CIFS has a mechanism called *path name expansion* built into it that allows clients to abbreviate names. If a path name does not begin with slash it is called a *relative path name*. Such names are interpreted relative to a *context* that includes *search rules* (described later) and a single *working directory.* Relative path names are interpreted relative to the working directory. For example, if the working directory is */ ivy/jefferson* then the relative path name *congress/constitution.tioga* is expanded by CIFS to the absolute path name

/ivy/jefferson/congress/constitution.tioga

In addition to traversing the tree of directories out from the root directory CIFS is also willing to work the other way. The backslash character causes CIFS to traverse in reverse. Consider the path name:

/ivy/jefferson\\indigo/franklin/kite.tioga

The first backslash tells CIFS to go back up to */ ivy* and the second backslash causes it to go back one more level to the root. This facility is intended for use in relative path names. For example, if the working directory is */ ivy/freud* then CIFS expands *\skinner/rat.mesa* to:

/ivy/skinner/rat.mesa

A client can thus go up and down the directory tree and use the working directory as a convenient reference point.

There is one final mechanism that is brought into play for relative path names that only consist of an entry name (e.g. the path *letter.tioga)*. If the path names a file, and the file is not found in the working directory, a list of other directories is tried in turn. This list of directories is called the *search rules*. Logically the working directory is the first member of the search rules because it is tried before any of the other directories. For example, if the search rules are:

/ivy/ravel	(working directory)
/ivy/beethoven	[1]
/ivy/debussy	[2]

Then *emperor.music* would be expanded to

/ivy/beethoven/emperor.music

and *LaMer.music* would be expanded to

/ivy/debussy/LaMer.music

assuming my musical history is correct.

A.2 Pragmatics

It would be nice if you did not need to know anything about how CIFS worked, but alas there are a few important things that will help you comprehend what is going on when something funny happens.

CIFS makes copies of files and directories on the local disk of the machine you are using. These copies are cataloged by an invisible directory called the *local system directory* (or LSD). The **lsd** command can be used to list the contents of the LSD.

When you alter a file, the changes are not reflected at the file's permanent home until you issue a **backup** command (see Section A.3). This has two important consequences. First, if two people modify a file or update a directory at the same time the last one to do a **backup** wins. The **catalog** command will fix any directory problems that might result, but in general concurrent modification of a file or a directory is not a good idea. Second, if you modify a file that you can not write on CIFS will not complain at the time. When **backup** runs it will be unable to copy your modification to the file's permanent home, and you will have to copy your modifications out of the file and use the **reset** command to flush the changes.

CIFS supplements the directory systems of file servers with its own set of directories. If the absolute path name of a directory is *path*, then the CIFS directory is stored in a file called *path/dir.bt*. The **catalog** command will initialize existing directories for use with CIFS. The **mkdir** command automatically initializes CIFS directories and **catalog** is not needed in this case.

In order to use files in non-CIFS directories the file's absolute path name must be given. For example:

/iris/redell/pilot/paper.press

For compatibility, the directory */local* refers to the Pilot Common Software Directory.

A.3 How to use CIFS

CIFS is in the Cedar boot file. To use CIFS commands, type:

run filesystemcommands

The User Exec will then understand the following set of simple file system commands:

asr *path*

Adds the directory *path* to the beginning of the search rules.

backup

Copies all of the files that have been modified to permanent storage.

catalog *path*

Initialize a CIFS directory system on the directory tree that starts at *path*. If there already is a CIFS directory system, it is checked for consistency and repaired if necessary.

comment *path comment*

Replaces the comment on the entry *path* by *comment*. If *comment* is not specified, the comment on *path* is deleted.

cd ***path***

Changes the working directory to be *path.*

CIFSDelete ***path***

If *path* is a file it is deleted. If *path* is a link it is deleted but its target is not. If *path* is a directory an error is signalled

dd ***path***

Deletes the directory *path.*

dsr ***path***

Deletes the directory *path* from the search rules.

groupreset ***prefix***

Reset all files that whose names begin with *prefix.*

link ***path target***

Creates a link called *path*, and makes its target path *target.*

ls ***[pattern] [-dir path] [-pat pattern]***

List the entries in directory *path*. If -dir is not specified, the working directory is assumed. The pattern can contain regular characters and the reserved characters "*" (match any sequence of characters) and "#" (match any one character).

lsd

List the contents of the local system directory. The listing is printed in the following format:

AbsolutePathname (time last modified) {Dirty}

If {Dirty} is not present, then the local copy of the file has not been modified.

lsr

List the search rules.

make ***path***

A simple way of creating a text file.

mkdir ***path***

Creates a directory called *path.*

pbf

Prints the number of pages that CIFS will try to keep free.

pwd

Prints the working directory.

CIFSRename ***fromPath toPath***

Renames the file named *fromPath* to be named *toPath.*

reset ***absolutePath***

Discards any updates that have been made to *absolutePath* that have not been permanently recorded with **backup**. *absolutePath* must be an absolute path name.

sbf ***pages***

Set base free pages sets the number of pages that CIFS will try to keep free. If no argument is supplied, CIFS will ensure that there are enough free pages according to the current setting. Setting the number of base free pages to be more than the size of the local disk will cause CIFS not to cache files.

swap *filea fileb*

Swap the contents of *filea* and *fileb.*

where *path*

Applies path name expansion to *path* and prints the result.

type *path*

Types the contents of *path.*

unlock

Emergency brute force unlock of all files.

A.4 The Salvager

CIFS includes a salvager that will rebuild the LSD from the leader pages of all files on the disk. The salvager is invoked by booting Cedar with the "l" switch and answering the questions in the obvious way. Salvaging can take several minutes on a Dorado.

Part B: Programmers Guide

The programmer's interface to CIFS is implemented by three definitions files: CIFS, CIFSFeedback, and FileLookup. These files are reproduced below, with the comments in the files serving as its documentation. The file CIFS contains the interfaces of most use to the applications programmer. CIFSFeedback is for use by the "Watcher". FileLookup provides access to a generally useful single packet protocol for interrogating IFS file servers.

```
-- CIFS.mesa  June 2, 1982 6:55 pm
-- Interface to the Cedar Interim File System.
-- Coded August, 1981 by D. Gifford
-- Last edited by
--   MBrown on August 30, 1982 1:06 pm


DIRECTORY
  Ascii: TYPE USING [NUL],
  File: TYPE USING [Capability],
  Rope: TYPE USING [ROPE];

CIFS: CEDAR DEFINITIONS = {

-- Public Procedures
-- Operations that everyone will want to use

  OpenFile: TYPE = REF FileObject;
  FileObject: TYPE;
    -- This type represents an open file.
    -- If an instance is not NIL, it represents an open file.

  Context: TYPE = REF ContextObj;
    -- If NIL is supplied as a context, CIFS will use its default context
```

```
ContextObj:  TYPE;


Mode:  TYPE  =  CARDINAL;
-- any combination  of modes  can  be  used,  but  useful  sets
-- of combinatatons  are:
--  {read},  {write},  {write,  create},  {write,  replace},  {write,  create,  replace}
-- dontCheck  can  be  included  in  any  of  these  sets
read:  Mode  =  1;
  -- read  mode  sets  a non-exclusive  lock  on  the  file
write:  Mode  =  2;
  -- write  mode  sets  an  exclusive  lock  on  the  file
  -- includes  read  by  default
create:  Mode  =  4;
  -- create  mode  will  create  a file  if  it  doesn't  exist
  -- normally  write  and  create  modes  are  used  together
replace:  Mode  =  8;
  -- Imagine  that  the  file  opened  is  "foo"
  -- and  the  file  that  "foo"  is  stored  in  is  F
  -- replace  mode  does  the  following:
  -- (1)  it  creates  a new  empty  file  F'  at  Open  time
  -- (2)  this  is  the  file  returned  to  the  client  by  GetFC
  -- (3)  at  close  time  it  makes  "foo"  point  to  F'
  -- (4)  F  is  made  temporary  so  it  will  be  deleted  at  the
  --  next  boot
dontCheck:  Mode  =  16;
  -- If  file  is  on  the  local  disk,  assume  that  the  copy  is  current  and  don't
  -- check  the  remote  sever
  -- Useful  as  a performance  optimization  for  files  that  are  "immutable"

maxPath:  CARDINAL  =  200;
    -- Maximum  path  length  is  200  characters
maxComment:  CARDINAL  =  300;
    -- Maximum  comment  length  is  300  characters

Close:  PROC  [fh:  OpenFile];
    -- Close  a  file
    -- Errors:  None

Connect:  PROC  [name,  password:  Rope.ROPE];
    -- Set  connect  credentials
    -- Errors:  None

CreateDir:  PROC  [name:  Rope.ROPE,  c:  Context  _  NIL];
    -- Create  a  directory
    -- Errors:  {illegalFileName,  localDiskFull,  noSuchDirectory,  ConnectionErrors,
    --  CredentialsErrors}

Delete:  PROC  [name:  Rope.ROPE,  c:  Context  _  NIL];
    -- Delete  a  file
    -- Files  are  created  by  Open
    -- Errors:  {illegalFileName,  noSuchFile,  requestRefused,  localDiskFull,
    --  fileBusy,  noSuchDirectory,  ConnectionErrors,  CredentialsErrors}
```

```
  DeleteDir: PROC [name: Rope.ROPE, c: Context _ NIL];
      -- Delete a directory
      -- Errors: {noSuchDirectory, directoryNotEmpty, ConnectionErrors, CredentialsErrors}

  GetFC: PROC [fh: OpenFile]
    RETURNS [fc: File.Capability];
      -- Returns the Pilot File Capability of an open file
      -- Errors: None

  GetPathname: PROC [fh: OpenFile]
    RETURNS [path: Rope.ROPE];
      -- Returns the pathname of an open file
      -- Errors: None

  Open: PROC [name: Rope.ROPE, mode: Mode, c: Context _ NIL]
    RETURNS [fh: OpenFile];
      -- Open a file
      -- Errors: {ConnectionErrors, CredentialsErrors, FileErrors, localDiskFull, fileBusy,
      -- noSuchDirectory}

  Rename: PROC [from: Rope.ROPE, to: Rope.ROPE, c: Context _ NIL];
      -- Rename a file
      -- Errors: {ConnectionErrors, CredentialsErrors, FileErrors, localDiskFull, fileBusy,
      -- noSuchDirectory, fileAlreadyExists}

  Swap: PROC [filea: Rope.ROPE, fileb: Rope.ROPE, c: Context _ NIL];
      -- Swap the contents of filea and fileb
      -- Errors: {ConnectionErrors, CredentialsErrors, FileErrors, localDiskFull, fileBusy,
      -- noSuchDirectory}

-- Public Procedures
-- Operations that some people will want to use

  AddSearchRule: PROC [path: Rope.ROPE, before: BOOLEAN, c: Context _ NIL];
      -- Add a search rule
      -- If before is T it will add it before the other search rules
      -- Errors: {ConnectionErrors, CredentialsErrors, localDiskFull, noSuchDirectory}

  CopyContext: PROC [from: Context _ NIL]
    RETURNS [c: Context];
      -- Copy a context
      -- If from is omitted, CopyContext will return a copy of the default
      -- context.
      -- Errors: None

  CreateContext: PROC
    RETURNS [c: Context];
      -- Create a new context for the interpretation of names
      -- The "Jerry Brown" Operation
      -- Errors: None

  CreateLink: PROC [path, targetPath: Rope.ROPE, c: Context _ NIL];
      -- causes path to point to targetPath
      -- whenever path is used as a file name, it will be resolved to
```

```
    -- targetPath
    -- Errors: {ConnectionErrors, CredentialsErrors, localDiskFull, noSuchDirectory,
    --  linkAlreadyExists}

DeleteContext: PROC [c: Context _ NIL];
    -- Destroy a context for the interpretation of names.
    -- The "Edward Kennedy" Operation
    -- Errors: None

DeleteSearchRule: PROC [path: Rope.ROPE, c: Context _ NIL];
    -- Delete a search rule
    -- Errors: None

EProc: TYPE = PROC [name, link, comment: REF TEXT]
  RETURNS [stop: BOOLEAN];
    -- Procedure that is called for dir enumeration
    -- name is the entry name
    -- if link.length#0 then link is the path that name will be resolved to
    -- if comment.length#0 then comment is a comment for name

Enumerate: PROC [dir: Rope.ROPE, pattern: Rope.ROPE, p: EProc, c: Context _ NIL];
    -- Enumerate the contents of a directory
    -- pattern can contain "*" and "#"
    -- a pattern of "*" enumerates the entire directory.
    -- If dir = NIL or dir = "" Enumerate assumes the wdir
    -- Errors: {ConnectionErrors, CredentialsErrors, localDiskFull, noSuchDirectory}

GetSearchRules: PROC [c: Context _ NIL]
  RETURNS [LIST OF Rope.ROPE];
    -- Returns the list of paths in the search rules
    -- Errors: None

GetWDir: PROC [c: Context _ NIL]
  RETURNS [path: Rope.ROPE];
    -- Return the path of the working directory
    -- Errors: None

Login: PROC [name, password: Rope.ROPE];
    -- Set credentials
    -- Errors: None

SetDefaultContext: PROC [c: Context];
    -- Set the default context
    -- Errors: None

SetComment: PROC [path, comment: Rope.ROPE, c: Context _ NIL];
    -- Sets the comment on the specified path to be comment
    -- Errors: {ConnectionErrors, CredentialsErrors, FileErrors, localDiskFull,
    -- noSuchDirectory}

SetWDir: PROC [path: Rope.ROPE, c: Context _ NIL];
    -- Set working directory
    -- Errors: {ConnectionErrors, CredentialsErrors, localDiskFull,
    -- noSuchDirectory}
```

```
-- Public Procedures
-- Operations that hardly anyone will want to use

  Expand: PROC [name: Rope.ROPE, c: Context _ NIL]
    RETURNS [path: Rope.ROPE];
      -- Expands a name into a full path name
      -- May raise CIFS.Error if the file is not in the dir
      --  system.
      -- Errors: {ConnectionErrors, CredentialsErrors, FileErrors, localDiskFull,
      -- noSuchDirectory}

  ExplicitBackup: PROC [name: Rope.ROPE, c: Context _ NIL]
    RETURNS [version: INT];
      -- Explicit backup of the named file to its remote server home
      -- The server's version number for the backing file is returned
      -- If name has a version number then ExplicitBackup will use
      --  that number for the backing file
      -- Errors: {ConnectionErrors, CredentialsErrors, FileErrors, fileBusy}

  GetBaseFreeSpace: PROC RETURNS[pages: INT];
      -- Gets the number of free pages that should be maintained
      -- on the local disk
      -- Errors: None

  OpenButDontHandleError: PROC [name: Rope.ROPE, mode: Mode, c: Context _ NIL]
    RETURNS [fh: OpenFile];
      -- Open a file
      -- Same as Open, except that it will not automatically handle
      -- a credential error.
      -- Errors: {ConnectionErrors, CredentialsErrors, FileErrors, localDiskFull, fileBusy,
      -- noSuchDirectory}

  Reset: PROC [name: Rope.ROPE, c: Context _ NIL];
      -- Reset a file from its backing store
      -- name must be an absolute path name
      -- Errors: {noSuchFile}

  SetBaseFreeSpace: PROC[pages: INT _ 0];
      -- Sets the number of free pages that should be maintained
      -- on the local disk
      -- If pages is omitted, CIFS will just ensure that there are
      -- enough free pages according to the current setting
      -- Errors: None

-- Exceptional conditions
-- Unfortunately, everyone has to know about these
-- In each CIFS call the most likely errors are listed. However, a client
-- should be prepared for variant of ErrorCode. The message "error" is
-- essential, and should be shown to the user.

  HandleError: PROC [code: ErrorCode, error: Rope.ROPE, reply: CHARACTER _ Ascii.NUL]
    RETURNS [resume: BOOLEAN];
    -- Attempts to handle a credential error. The usage of this procedure is:
```

```
    -- CIFS.Operation[  args !
    --    CIFS.Error  =>  {IF CIFS.HandleError[code,   error,  reply] THEN  RESUME}];
    -- NOTE:  The  above  code  is included  in CIFS,  so a client  does  not  have
    --  to write  it.  This  procedure  is exported  only  for  completeness.

  Error:  SIGNAL  [code: ErrorCode,  error:  Rope.ROPE,  reply:  CHARACTER  _  Ascii.NUL];
    -- error  usually  contains  an interesting  message
    -- reply  is the  code  passed  from  server  (see STPReplyCode.mesa)

  ErrorCode:  TYPE  =  {
    -- connection  errors
    noSuchHost,  noRouteToNetwork,  noNameLookupResponse,   alreadyAConnection,
    noConnection,  connectionClosed,  connectionRejected,
    connectionTimedOut,
    -- credentials  errors
    accessDenied,  illegalUserName,  illegalUserPassword,  illegalUserAccount,
    illegalConnectName,  illegalConnectPassword,  credentialsMissing,
    -- protocol  errors
    protocolError,
    -- file  errors
    illegalFileName,  noSuchFile,  requestRefused,
    -- remote  stream  errors
    accessError,
    -- catch  all
    undefinedError,
    -- local  disk  full
    localDiskFull,
    -- file  is locked
    fileBusy,
    -- file already  exists  for  a rename  command
    fileAlreadyExists,
    -- directory  does  not  exist
    noSuchDirectory,
    -- directory  contains  files and  can  not  be  deleted
    directoryNotEmpty,
    -- link  alrady  exists
    linkAlreadyExists
    };

  ConnectionErrors:  TYPE  =  ErrorCode[noSuchHost..connectionTimedOut];
  CredentialsErrors:  TYPE  =  ErrorCode[accessDenied..credentialsMissing];
  FileErrors:  TYPE  =  ErrorCode[illegalFileName..requestRefused];

}..


-- CIFSFeedback.mesa
-- Interface  for  CIFS  to  report  STP  activities
-- Coded  by  M. D. Schroeder,  July  16,  1982  1:49  pm
-- Last  edited  by
--   MBrown  on  August  30,  1982  1:24  pm


DIRECTORY
```

```
  Rope: TYPE USING [ROPE];

CIFSFeedback: CEDAR DEFINITIONS = {

Register: PROC [p: PROC [Rope.ROPE] _ NIL];
  -- CIFS will call the registered procedure with a Rope describing each
  --file manipulation activity on a remote server. Registering p ~ NIL will
  --stop such reporting.
  -- The style of use is: pass a non-empty rope at the start of an activity,
  --pass an empty rope (e.g. NIL) at the end of the activity.

  }..


-- FileLookup.mesa
-- M. D. Schroeder, September 7, 1982 11:54 am

DIRECTORY
  Rope: TYPE USING [ROPE],
  System: TYPE USING [GreenwichMeanTime];

FileLookup: CEDAR DEFINITIONS = BEGIN

Result: TYPE = {noResponse, noSuchPort, noSuchName, ok};

LookupFile: PROC [server, file: Rope.ROPE ]
    RETURNS [result: Result, version: CARDINAL,
      create: System.GreenwichMeanTime, count: LONG CARDINAL];

-- This procedure uses the LookupFile packet exchange protocol to obtain the
-- version number, create time, and byte length of a file on a remote file server.
-- The file name may be specified complete with version number, with "!h", with "!l",
-- or with no version. The file names can be specified either with the "<..>..>.." syntax
-- or with the ".../.../..." syntax (be sure that the name does not start with a '/).

-- If the result is "ok" then the requested file exists with the returned
-- version number, create time, and byte length. "noSuchName" means that either
-- the server name was nonsense or that the file does not exist on that server.
-- "noSuchPort" means that the server responded with a no-such-port error packet
-- when prodded on the LookupFile socket. "noResponse" means that either the NLS
-- didn't respond to the server name lookup, the LookupFile packet didn't get to the
-- server, or the server did not respond to it.

-- LookupFile caches the state of the server as derived from previous lookup attempts.
-- If the cached state for a server is either "noResponse" or "noSuchPort" then
-- LookupFile immediately returns that result without attempting any communication.
-- Such negative cache entries are flushed after five minutes. Positive cache entries
-- contain the Pup address of the server, eliminating the need to do a NLS lookup
-- on the server name.

-- The longest that you should have to wait for a response is ~ 4*(2 + 0.5*MIN[8,hopCount])
-- seconds. This is the time it will take to decided that a server isn't going to respond in the
-- case of an uncached down server. Most answers are determined much more quickly.
```

END...

Appendix: Example Program

```
-- CType.mesa
-- L. Stewart  25-Mar-82  11:52:42

DIRECTORY
  CIFS,
  UserExec: TYPE USING [RegisterCommand, UserAbort, ResetUserAbort, CommandProc],
  FileIO: TYPE USING [StreamFromOpenFile],
  IO,
  Rope,
  UECP;

CType: PROGRAM
  IMPORTS CIFS, UECP, UserExec, FileIO, IO
  = {

  -- Type a file
  typeC: Rope.ROPE = "Type Usage:
  type path
The contents of path will be typed.";

  Type: UserExec.CommandProc = {
    fh: CIFS.OpenFile;
    riS: IO.Handle;
    argv: UECP.Argv _ UECP.Parse[exec.commandLine];
    n: NAT;
    h: IO.Handle = exec.out;
    buffer: REF TEXT _ NEW[TEXT[256]];
    IF argv.argc=1 THEN RETURN;
    {
    ENABLE {
      CIFS.Error => {
                h.PutF["%s\n", IO.rope[error]];
                CONTINUE;
                };
      IO.EndOfStream => {
        h.PutChar['\n];
                CONTINUE;
                };
      };
    fh _ CIFS.Open[name: argv[1], mode: CIFS.read];
    riS _ FileIO.StreamFromOpenFile[openFile: fh];;
    DO
      IF UserExec.UserAbort[] THEN EXIT;
      n _ riS.GetBlock[buffer];
      h.PutBlock[buffer];
      IF n # 256 THEN EXIT;
```

```
    ENDLOOP;
  IF  n>0 AND  buffer[n-1]#'\n    THEN  h.PutChar['\n];
  };
  UserExec.ResetUserAbort[];
  IF  riS#NIL    THEN  riS.Close[];
  IF  fh#NIL    THEN  CIFS.Close[fh];
  };


-- Mainline
UserExec.RegisterCommand["Type.~",      Type,  "Type  file",  typeC];

}.
```