



APPLICATION NOTE 709

Adding An External File System to TINI

Abstract: This application note describes the techniques used to add an external file system to TINI®. By providing a method of accessing other file systems, TINI users will no longer be limited in the type and amount of information they can process. Since TINI also uses its RAM as its operating heap, using a remote file system allows more of the RAM to be used for running applications.

Source code for an example mountable file system is given that uses the built-in TCP/IP capabilities of TINI to access files on a remote server. Step-by-step instructions are given on how to compile and run this example.

Introduction

The standard file system of any embedded system is usually small in size. The storage space of the Tiny Internet Interfaces (TINI®) platform is no exception. TINI's file system resides in the on-board RAM and is limited by the size of this RAM. Because of this constraint, it is important to provide alternatives for data storage.

This application note describes a technique for adding an external file system. By providing a method of accessing other file systems, TINI users no longer are limited in the type and amount of information they can process. Since TINI also uses its RAM as its operating heap, using a remote file system allows more of the RAM to be used for running applications.

Once a file system has been mounted, there appears to be no difference between a file that resides on the local file system and one that is external. All interactions with files occur through the standard `java.io` classes and the `com.dalsemi.fs.DSFile` class.

System Overview

There are three steps to implementing an external file system. First, the `com.dalsemi.fs.FileSystemDriver` interface must be implemented. This class can use a native library, rely on a server running elsewhere (like the example included in this application note), or any number of other options. Second, the class (or classes) must be made available to any process that uses the external file system. This can be done by placing a copy of the class in TINI's classpath or by building it directly into any application that will use the new file system. Finally, the file system needs to be mounted. This is done by either calling the `com.dalsemi.fs.DSFile.mount()` method from inside an application or using the `mount` command in `slush`.

The `com.dalsemi.fs.FileSystemDriver` interface has been written so that a variety of systems can be implemented. The developer has the freedom to handle files in a manner that is appropriate for his design. Possibilities include a Network File System (NFS), an FTP file system where files are written to and read from an FTP server, or an IDE disk drive connected to TINI. This example uses a custom design that uses TINI's on-board TCP/IP network stack to access files from a remote server. **Figure 1** shows the system's configuration. TINI's file system holds a reference to the file system driver. Whenever a remote file is accessed, the appropriate method in the driver is called. The driver then communicates with the remote host over the network to process the request.

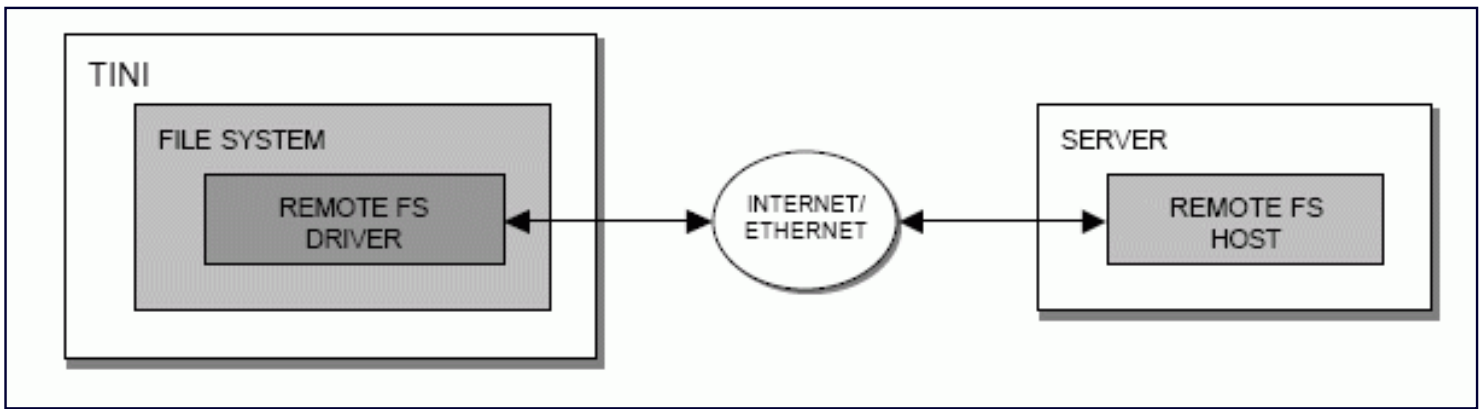


Figure 1. System Block Diagram.

TINI Software

`NetworkFileSystemDriver` is the class that implements the `com.dalsemi.fs.FileSystemDriver` interface. The driver's `init()` method is called automatically whenever a file system is mounted. This gives the driver an opportunity to perform any initialization that might be necessary. Some examples include resetting and initializing hardware, allocating buffers, and establishing network connections. In this example, the `init()` method attempts to establish a socket connection with the remote server specified in the parameters. Upon a successful connection, the server sends a new port number that the driver must use to send file system commands. This allows the original port to remain open for additional connections.

```

DataInputStream in = null;
DataOutputStream out = null;
Socket sck = null;
public void init(String[] args) throws Exception
{
    //args[0] is IP address of file system server.
    //args[1] is the port to connect to.
    int port = Integer.parseInt(args[1]);
    int newPort = 0;
    try
    {
        sck = new Socket(args[0], port);
        //Once connected the server will send back a new port to connect
        //to. This way the original server port will remain available
        //for additional connections.
        in = new DataInputStream(sck.getInputStream());
        newPort = in.readInt();
    }
    finally
    {
        if(in != null) {in.close();}
        if(sck != null) {sck.close();}
    }
    //Open the new socket and get its input and output streams.
    //We'll wrap them in Data streams for convenience.
    sck = new Socket(args[0], newPort);
    in = new DataInputStream(sck.getInputStream());
    out = new DataOutputStream(sck.getOutputStream());
  
```

The `unmount()` method of the driver is called when the file system is unmounted. The driver should use this method to release any resources it may be using (e.g., buffers, network connections, ports, etc.). In the example presented here, the driver attempts to notify the server it is about to disconnect. The driver then closes the connection it has with the server.

```

public void unmount()
{
    try
    {
        try
        {
  
```

```

//Send the command byte.
//This will notify the server it can close its end.
out.writeByte(CLOSE);
}
finally
{
//We don't care what the server does, we still need to close our end.
in.close();
out.close();
sck.close();
}
}
catch(IOException x)
{
}
}

```

The remaining methods in the class are all file system operations (e.g., open, close, read, write, etc.). Each follows the flow chart shown in **Figure 2**. These methods were kept as simple as possible in order to put most of the workload onto the server. An example implementation of one of the file system operations (File.exists()) is given below.

```

//out is a DataOutputStream for the command socket between the client and host.
//in is a DataInputStream for the command socket between the client and host.
//EXISTS is a byte that the server understands as the "does file exist" command.
public boolean exists(String fileName)
{
try
{
//Send the command byte and the necessary arguments.
out.writeByte(EXISTS);
writeString(fileName);
//Get the result.
return in.readBoolean();
}
catch(Exception s)
{
return false;
}
}
}

```

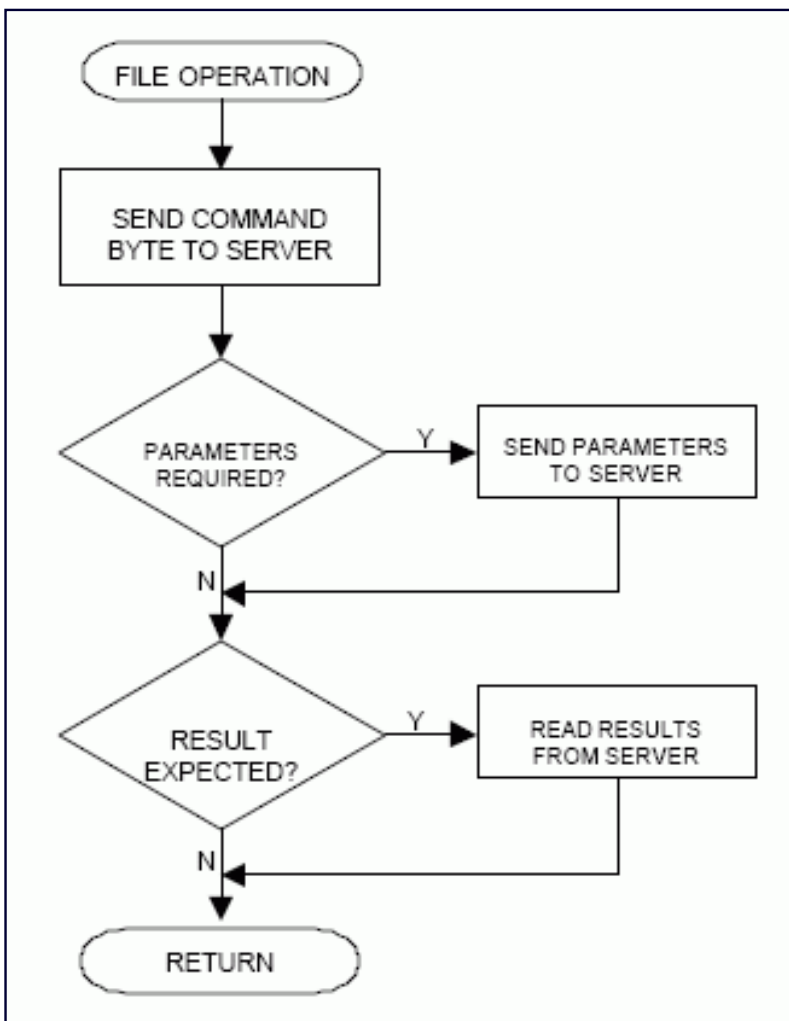


Figure 2. Block Diagram.

Server Software

The `NetworkFileSystemHost` class acts as the file system server that resides on the remote host. This server is multithreaded and can handle multiple simultaneous connections. When the server starts, it creates a server socket that waits for incoming connections.

```

//Create a socket to accept incoming connections.
ServerSocket ss = new ServerSocket(3453);
System.out.println("Server running... Waiting for connections...");
Socket sck = ss.accept();
  
```

When a new connection is requested, the server responds with a new port number to which the client should connect and creates a new thread that listens on the given port for commands from the client.

```

//Open a new server socket and send its port number to the client.
//We want to keep the original free for more connections.
DataOutputStream out = new DataOutputStream(sck.getOutputStream());
ServerSocket ss2 = new ServerSocket(0);
//Start a thread to handle the new connection. The thread will
//be responsible for all file system operations.
new SessionThread(ss2).start();
out.writeInt(ss2.getLocalPort());
out.flush();
  
```

The thread continues handling commands until it receives the disconnect command. The listing below shows how the server

handles this command. The server's counterpart to the `exists()` function of the TINI software is also shown.

```
//out is a DataOutputStream for the command socket between the client and host.
//in is a DataInputStream for the command socket between the client and host.
boolean closed = false;
while(!closed)
{
byte command = in.readByte();
switch(command)
{
case DISCONNECT:
{
//Close the communication socket as requested.
in.close();
out.close();
sck.close();
//Clean up all the open files held by this thread.
closeOpenFiles(openRand);
closeOpenFiles(openRead);
closeOpenFiles(openWrite);
//Set the state to closed. This will stop the main
//loop in the run() method and allow the thread to
//terminate.
closed = true;
break;
}
case EXISTS:
{
//Get the parameters for this operation.
String fName = readString();
//Send the result.
File f = new File(parent, fName);
out.writeBoolean(f.exists());
break;
}
//Handle other commands here.
}
}
```

Figure 3 illustrates the entire program flow for the server.

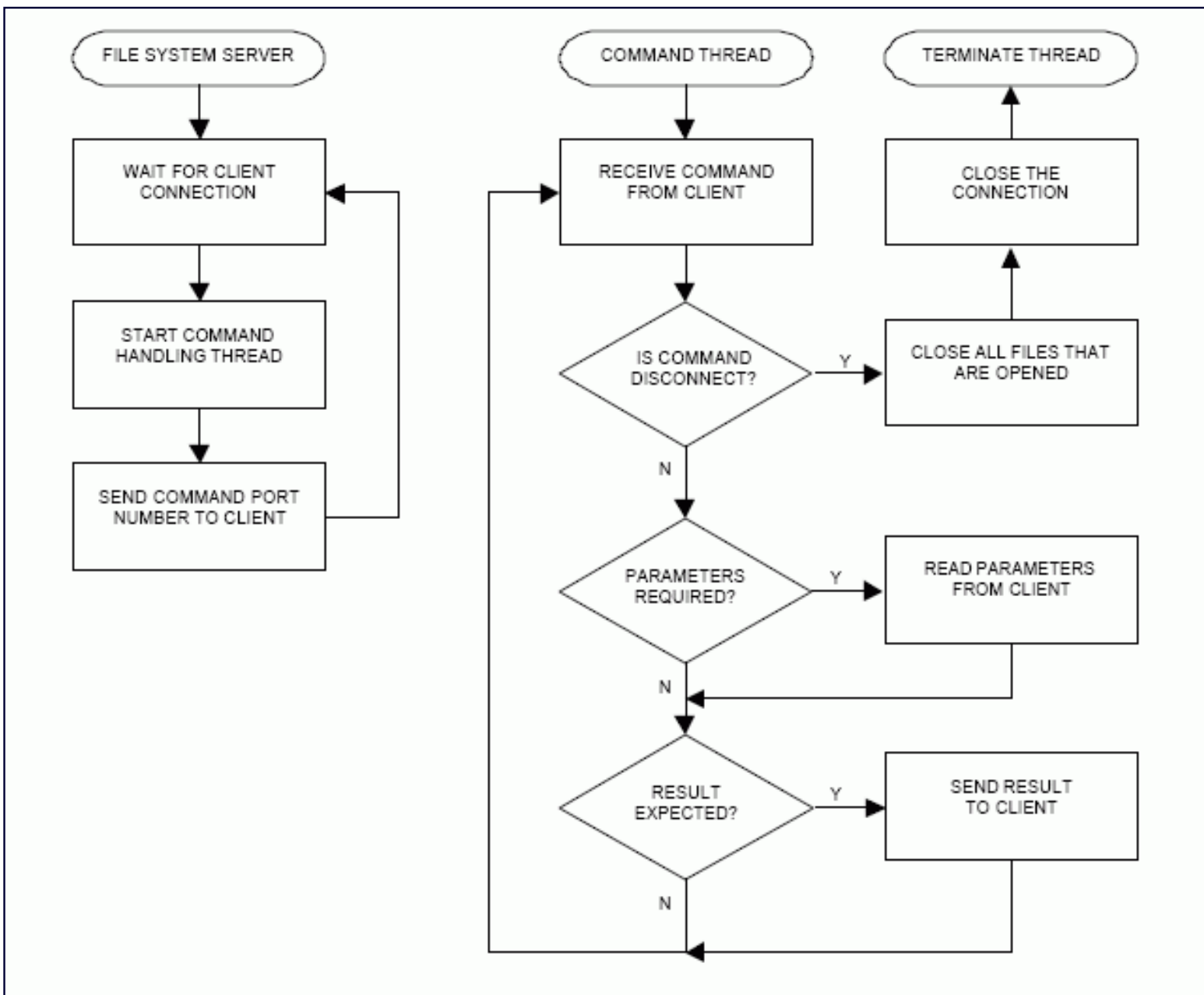


Figure 3. File System Server.

Running the Example

To run the example, complete the following steps:

- Install TINIOS 1.10.
- Include the `mount` and `unmount` commands in slush. To do this, first extract the source for the commands from the `OptionalSlushCommandsSrc.jar` file and compile them. Then FTP the class files into the `/tiniext/com/dalsemi/slush/command` directory on TINI. Run the following commands at the slush prompt:

```
addc com.dalsemi.slush.command.MountCommand mount
```

```
addc com.dalsemi.slush.command.UnmountCommand unmount
```

- Compile the two source files included with this example using the following commands:

```
javac -bootclasspath \bin\tiniclasses.jar NetworkFileSystemDriver.java
```

```
javac NetworkFileSystemHost.java
```

where <TINI1.10> is the TINI 1.10 installation directory.

- FTP the `NetworkFileSystemDriver.class` file into the `/tiniext` directory on TINI.
- Start the file system server with the following command: `java NetworkFileSystemHost <startDir>` where `<startDir>` is the directory that is visible from the TINI file system.
- In slush, mount the file system with the following command:
`mount mnt NetworkFileSystemDriver <host ip> 3453`
- where `<host ip>` is the IP address of the system where the server is running.

There will now be a directory in the root of TINI's file system called "mnt." This directory shows the files that are available from the remote server. These files can be accessed just like files that reside in TINI's internal file system and are available to all processes created by slush.

Associated Files

Files associated with AN709 are available for [download](#).

Conclusion

By using the built-in TCP/IP capabilities of TINI, the standard file system can be extended without much overhead in the TINI system. This frees the applications developed for TINI from the constraints of the internal file system. More RAM also is made available since data files, log files, etc., can now be stored remotely.

Appendix A

The following is the API documentation of the `com.dalsemi.fs.FileSystemDriver` interface.

com.dalsemi.fs Interface FileSystemDriver

public interface **FileSystemDriver** This interface is used to implement external file systems. Anyone wishing to extend TINI's file system must first implement this interface. The class can then be passed to the `com.dalsemi.fs.DSFile.mount()` method for creation.

Nearly every method in this interface takes either a file name or a file descriptor. File names will be Strings that do not include the name of the mount point itself. For example, if you have a mount point named "mnt", and on the mounted file system there is a directory called test, on TINI you would access a file in that directory by using the String `"/mnt/test/myfile."` The name would then be passed to the driver as `"test/myfile."` When accessing mount point itself (`"/mnt"`), it is represented by an empty string (`""`). Methods that use a file descriptor use the file descriptor returned by the `openReadingFD`, `openWritingFD`, and `openRandomFD` methods. What is contained in the file descriptor is up to the developer.

Many methods also use a uid parameter. This is the uid of the person trying to perform the operation. IDs that have the high bit set can be assumed to have administrator privileges.

Method Summary

int	<code>available(Object fd)</code> The number of bytes that can be read without blocking.
boolean	<code>canExec(String fileName, byte uid)</code> Determines if the given file is executable.
boolean	<code>canRead(String fileName, byte uid)</code> Determines if the given file is readable.
boolean	<code>canWrite(String fileName, byte uid)</code> Determines if the given file is writable.
Void	<code>close(Object fd)</code> Closes the file descriptor's stream and releases any system resources used.

boolean	delete(String fileName, byte uid) Removes the specified file from the mounted file system.
boolean	exists(String fileName) Determines if the given file exists on the mounted file system.
byte[]	getContents(String fileName, byte uid)>br> Gets the complete contents of a file on the mounted file system.
long	getLength(Object fd) Gets the length of the file represented by the file descriptor.
long	getOffset(Object fd) Gets the current offset into the file.
int	getOtherPermissions(String fileName) Gets the other (non-owner) permissions for the given file.
int	getUser(String fileName) Gets the owner of a file.
int	getUserPermissions(String fileName) Gets the user/owner permissions for the given file.
void	init(String[] args) This method will be called the first time a mounted file system accessed.
boolean	isDirectory(String fileName) Determines if the given name represents a directory.
boolean	isFile(String fileName) Determines if the given name represents a file and not a directory.
long	lastModified(String fileName) Indicates the time the file was last modified.
long	length(String fileName) Gets the length of the file.
string[]	list(String fileName, byte uid) Retrieves a listing of the files in the directory specified.
boolean	mkdir(String fileName, byte uid) Creates a directory on the mounted file system.
object	openRandomFD(String fileName, byte uid) Opens the given file for random access.
object	openReadingFD(String fileName, byte uid) Opens the given file for reading.
object	openWritingFD(String fileName, boolean append, byte uid) Opens the given file for writing.
int	readBytes(Object fd, byte[] data, int start, int length) Reads from the file represented by the file descriptor.
boolean	rename(String srcname, String destname, byte uid) Changes the name of a file.
Void	seek(Object fd, long n) Moves the file pointer to a given location, measured in bytes from the beginning of the file.
Void	setOtherPermissions(String fileName, int perms, byte uid) Changes the other (non-owner) permissions for the given file.
void	setUser(String fileName, byte newUID, byte uid) Sets the owner of the given file.
void	setUserPermissions(String fileName, int perms, byte uid) Changes the user/owner permissions for the given file.
long	skipBytes(Object fd, long n) Skips the next n bytes of data from the stream.
void	touch(String fileName, byte uid) Updates the last modified time on the given file to the current time.
void	unmount() Allows the driver a chance to clean up and release any resources used when a mount point is removed.
void	writeBytes(Object fd, byte[] data, int start, int length) Writes the given data to the file represented by the file descriptor.

Method Detail

init

```
public void init(String[] args)  
throws Exception
```

This method will be called the first time a mounted file system accessed.

Parameters:

args - Any arguments needed to initialize the file system.

Throws:

Exception -

exists

```
public boolean exists(String fileName)
```

Determines if the given file exists on the mounted file system. This method should match the behavior of `java.io.File.exists()`.

Parameters:

fileName - The file.

Returns:

true if the file exists.

canWrite

```
public boolean canWrite(String fileName, byte uid)
```

Determines if the given file is writable. If the file does not exist, the driver should determine if it can be created and return accordingly. This method should match the behavior of `java.io.File.canWrite()`.

Parameters:

fileName - The file.

uid - The user that is trying to access the file.

Returns:

true if the file can be written by this user.

canRead

```
public boolean canRead(String fileName, byte uid)
```

Determines if the given file is readable. This method should match the behavior of `java.io.File.canRead()`.

Parameters:

fileName - The file.

uid - The user that is trying to access the file.

Returns:

true if the file can be read by this user.

canExec

```
public boolean canExec(String fileName, byte uid)
```

Determines if the given file is executable. This method should match the behavior of `com.dalsemi.fs.DSFile.canExec()`.

Parameters:

fileName - The file. uid - The user that is trying to access the file.

Returns:

true if the file can be executed by this user.

isFile

```
public boolean isFile(String fileName)
```

Determines if the given name represents a file and not a directory. This method should match the behavior of `java.io.File.isFile()`.

Parameters:

`fileName` - The file.

Returns:

true if `fileName` represents a file.

isDirectory

```
public boolean isDirectory(String fileName)
```

Determines if the given name represents a directory. This method should match the behavior of `java.io.File.isDirectory()`.

Parameters:

`fileName` - The file.

Returns:

true if `fileName` represents a directory.

LastModified

```
public long lastModified(String fileName)
```

Indicates the time the file was last modified. This method should match the behavior of `java.io.File.lastModified()`.

Parameters:

`fileName` - The file.

Returns:

the modification time of the file.

length

```
public long length(String fileName)
```

Gets the length of the file. This method should match the behavior of `java.io.File.length()`.

Parameters:

`fileName` - The file.

Returns:

the length of the file.

mkdir

```
public boolean mkdir(String fileName, byte uid)
```

Creates a directory on the mounted file system. This method should match the behavior of `java.io.File.mkdir()`.

Parameters:

`fileName` - The name of the directory to create.

`uid` - The user that is trying to create the directory.

Returns:

true if the directory was created.

rename

```
public boolean rename(String srcname, String destname, byte uid)
```

Changes the name of a file. This method should match the behavior of `java.io.File.renameTo(File dest)`.

Parameters:

srcname - The name of the file to be changed.

destname - The new name for the file.

uid - The user that is trying to rename the file.

Returns:

true if the file was renamed.

list

```
public String[] list(String fileName, byte uid)
```

Retrieves a listing of the files in the directory specified. This method should match the behavior of `java.io.File.list()`.

Parameters:

fileName - The directory to get a listing from.

uid - The user trying to retrieve the list.

Returns:

the list of files, or null if fileName doesn't represent a directory.

delete

```
public boolean delete(String fileName, byte uid)
```

Removes the specified file from the mounted file system. This method should match the behavior of `java.io.File.delete()`.

Parameters:

fileName - The File to delete.

uid - The user trying to delete the file.

Returns: true if the file was removed.

touch

```
public void touch(String fileName, byte uid)
```

throws `IOException`

Updates the last modified time on the given file to the current time. This method should match the behavior of `com.dalsemi.fs.DSFile.touch()`.

Parameters:

fileName - The file to touch.

uid - The user trying to update the file.

Throws:

`IOException` -

setUserPermissions

```
public void setUserPermissions(String fileName, int perms, byte uid)
```

throws `IOException`

Changes the user/owner permissions for the given file. This method should match the behavior of `com.dalsemi.fs.DSFile.setUserPermissions(int perms)`.

Parameters:

fileName - The file.

perms - The new permissions.

uid - The user that is trying to change the file.

Throws:

`IOException` -

setOtherPermissions

```
public void setOtherPermissions(String fileName, int perms, byte uid)
throws IOException
```

Changes the other (non-owner) permissions for the given file. This method should match the behavior of `com.dalsemi.fs.DSFile.setOtherPermissions(int perms)`.

Parameters:

`fileName` - The file.

`perms` - The new permissions.

`uid` - The user that is trying to change the file.

setUser

```
public void setUser(String fileName, byte newUID, byte uid)
throws IOException
```

Sets the owner of the given file. This method should match the behavior of `com.dalsemi.fs.DSFile.setUser(byte uid)`.

Parameters:

`fileName` - The file.
`newUID` - The new owner.

`uid` - The user that is trying to change the file.

Throws:

IOException -

getUserPermissions

```
public int getUserPermissions(String fileName)
throws FileNotFoundException
```

Gets the user/owner permissions for the given file. This method should match the behavior of `com.dalsemi.fs.DSFile.getUserPermissions()`.

Parameters:

`fileName` - The file.

Returns:

the user permissions.

Throws:

FileNotFoundException -

getOtherPermissions

```
public int getOtherPermissions(String fileName)
throws FileNotFoundException
```

Gets the other (non-owner) permissions for the given file. This method should match the behavior of `com.dalsemi.fs.DSFile.getOtherPermissions()`.

Parameters:

`fileName` - The file.

Returns:

the other permissions.

Throws:

FileNotFoundException -

getUser

```
public int getUser(String fileName)
throws FileNotFoundException
```

Gets the owner of a file. This method should match the behavior of `com.dalsemi.fs.DSFile.getUser()`.

Parameters:

`fileName` - The file.

Returns:

the file's owner.

Throws:

FileNotFoundException -

openWritingFD

```
public Object ,b>openWritingFD(String fileName, boolean append, byte uid)
```

```
throws IOException
```

Opens the given file for writing. The file descriptor that is returned will be used to identify the file in other driver calls. A file descriptor can be any length and should hold any information the driver needs to identify the associated file and the stream's state. This method should match the behavior of `java.io.FileOutputStream(String name, boolean append)`.

Parameters:

fileName - The name of the file to open.

append - If true and the file exists, the file should be opened and the file pointer set to the end of the file. If false and the file exists, all contents of the file should be erased and the file's length set to 0.

uid - The user trying to open the file.

Returns:

The file descriptor.

Throws:

IOException -

openReadingFD

```
public Object openReadingFD(String fileName, byte uid)
```

```
throws FileNotFoundException
```

Opens the given file for reading. The file descriptor that is returned will be used to identify the file in other driver calls. A file descriptor can be any length and should hold any information the driver needs to identify the associated file and the stream's state. This method should match the behavior of `java.io.FileInputStream(String name)`.

Parameters:

fileName - The name of the file to open.

uid - The user trying to open the file.

Returns:

The file descriptor.

Throws:

FileNotFoundException -

openRandomFD

```
public Object openRandomFD(String fileName, byte uid) throws IOException
```

Opens the given file for random access. The file descriptor that is returned will be used to identify the file in other driver calls. A file descriptor can be any length and should hold any information the driver needs to identify the associated file and the stream's state. This method should match the behavior of `java.io.RandomAccessFile(String name, String mode)`.

Parameters:

fileName - The name of the file to open.

uid - The user trying to open the file.

Returns:

The file descriptor.

Throws:

IOException -

writeBytes

```
public void writeBytes(Object fd, byte[] data, int start,int length)
```

```
throws IOException
```

Writes the given data to the file represented by the file descriptor. This method should match the behavior of `java.io`.

OutputStream.write(byte[] b, int off, int len).

Parameters:

fd - The file descriptor identifying the file to write to.

data - The data to write.

start - The start offset in the data.

length - The number of bytes to write.

Throws:

IOException -

readBytes

```
public int readBytes(Object fd, byte[] data, int start, int length)
```

```
throws IOException
```

Reads from the file represented by the file descriptor. This method should match the behavior of java.io.InputStream.read(byte[] b, int off, int len).

Parameters:

fd - The file descriptor identifying the file to read from.

data - A buffer to store the data that is read.

start - The start offset in the buffer.

length - The number of bytes to read.

Returns:

the number of bytes read.

Throws:

IOException -

seek

```
public void seek(Object fd, long n)
```

```
throws IOException
```

Moves the file pointer to a given location, measured in bytes from the beginning of the file. This method should match the behavior of java.io.RandomAccessFile.seek(long pos).

Parameters:

fd - The file descriptor identifying the file.

n - The new position for the file pointer.

Throws:

IOException -

skipBytes

```
public long skipBytes(Object fd, long n)
```

```
throws IOException
```

Skips the next n bytes of data from the stream. This method should match the behavior of java.io.InputStream.skip(long n) if the file descriptor represents a FileInputStream and should match the behavior of java.io.RandomAccessFile.skipBytes(long n) if the file descriptor represents a RandomAccessFile.

Parameters:

fd - The file descriptor identifying the file.

n - The number of bytes to skip.

Returns:

The actual number of bytes skipped.

Throws:

IOException -

getOffset

```
public long getOffset(Object fd)
```

```
throws IOException
```

Gets the current offset into the file. This method should match the behavior of java.io.

RandomAccessFile.getFilePointer().

Parameters:

fd - The file descriptor identifying the file.

Returns:

the current position of the file pointer.

Throws:

IOException -

getLength

public long **getLength**(Object fd)

throws IOException

Gets the length of the file represented by the file descriptor. This method should match the behavior of java.io.

RandomAccessFile.length().

Parameters:

fd - The file descriptor identifying the file.

Returns:

the length of the file.

Throws:

IOException -

available

public int **available**(Object fd)

throws IOException

The number of bytes that can be read without blocking. This method should match the behavior of java.io.FileInputStream.available().

Parameters:

fd - The file descriptor identifying the file.

Returns:

the number of bytes available.

close

public void **close**(Object fd) throws IOException

Closes the file descriptor's stream and releases any system resources used. This method should match the behavior of java.io.FileInputStream.close(), java.io.FileOutputStream.close(), or java.io.RandomAccessFile.close() depending on the type of file descriptor passed in.

Parameters:

fd - The file descriptor identifying the file.

Throws:

IOException -

unmount

public void **unmount**()

Allows the driver a chance to clean up and release any resources used when a mount point is removed.

getContents

public byte[] **getContents**(String fileName, byte uid)

throws IOException

Gets the complete contents of a file on the mounted file system. This method will be called when the system attempts to

execute a file located on the mounted file system.

Parameters:

fileName - The file to retrieve.

uid - The user trying to execute the file.

Returns:

the contents of the file.

Throws:

IOException -

Application Note 709: <http://www.maxim-ic.com/an709>

More Information

For technical questions and support: <http://www.maxim-ic.com/support>

For samples: <http://www.maxim-ic.com/samples>

Other questions and comments: <http://www.maxim-ic.com/contact>

Related Parts

DS80C390: [QuickView](#) -- [Full \(PDF\) Data Sheet](#) -- [Free Samples](#)

DS80C400: [QuickView](#) -- [Full \(PDF\) Data Sheet](#) -- [Free Samples](#)

DSTINI400: [QuickView](#) -- [Full \(PDF\) Data Sheet](#)

DSTINIs400: [QuickView](#) -- [Full \(PDF\) Data Sheet](#)

AN709, AN 709, APP709, Appnote709, Appnote 709

Copyright © by Maxim Integrated Products

Additional legal notices: <http://www.maxim-ic.com/legal>