

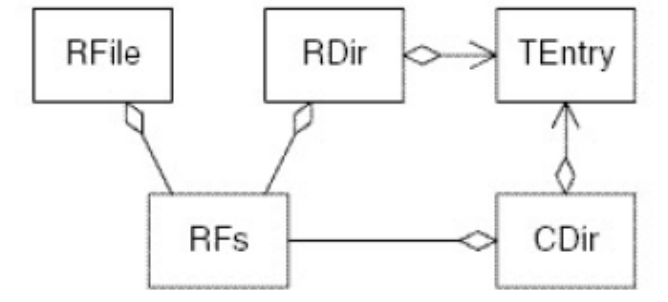
## The File Server

The **file server (F32)** offers a client API that allows user programs to manipulate drives, directories, and files and to read and write data in files. **File server** uses DOS-like conventions to offer up to 26 devices (drives) identified as a: to z:, a fully hierarchical directory structure and long filenames incorporating almost any character ?except those reserved by the file system itself. Directory names are separated by backslashes ('\, as in Windows), rather than by forward slashes ('/', as in UNIX). A period ('.') may be used to indicate an extension; although this has no special meaning to Symbian OS, some applications may assign their own meaning to them.

A filename, *including* its drive and directory portion, may be up to 256 characters long. Like Windows (and unlike UNIX), the file server (VFAT driver) is case preserving, but not case sensitive. In other words, if you create a file called My File, all directory operations will return the name My File with the original case. But if you search for My File, my file, MY file, or any other combination, the file My File will be returned. Apparently, this means that there can not be two or more files in the same directory whose names differ only in the cases of some of their letters.

The file server's API has been designed for easy mapping to **POSIX APIs**. The Symbian OS C standard library is built on top, using **FILE** and its associated functions to map to **RFile** and its member functions and so on for other file-related operations.

The main classes in the file server API are:



An **RFile** is a session from your program to the file server. You need a session for all file related operations and in order to be able to use other classes such as **RFile**, **RDir**, and **CDir**.

## File Server Sessions

All servers use session-based communication so that a client function such as **RFile::MkDir()** or **RFile::Write()** is converted into a message that's sent to the server. The requested function is performed in the server and then any result is passed back to the client. It isn't necessary to understand how servers work in order to use them; all you need to know is that you can't do anything without a connected session ? in the case of the file server, a connected **RFile**.

So the pattern for using the **file server** is:

- connect an **RFs** to the file server,
- open a file, specifying which **RFs** to use,
- perform your required operations,
- close the file,
- close the **RFs** using *Close()*, since **R** objects don't have destructors.

You can open any number of files or directories using a single **RFs**. You can carry an **RFs** as part of your application's object data, open it at the time you open your application, and close it when you finish.

After you have closed the session, you won't be able to call any more member functions on objects that were opened using that session. You should make sure you close and clean up these objects ? preferably before you close the session. In any case, when the session is closed, the server will clean up any server-side resources associated with the session.

**RFs** contains many useful file system-related operations:

- manipulating the current directory.
- making, removing, and renaming directories.
- deleting and renaming files.
- changing directory and file attributes.
- notifying changes.
- manipulating drives and volumes.
- peeking at file data without opening the file (used by some file format recognizers).
- adding and removing file systems.
- system functions to control and check the status of the server. [[Category:Architecture/Design]]