Symbian OS Internals/09. The File Server

- Symbian OS Internals Table of Contents

by Peter Scobie

RAM disk is not an installation procedure.
Unknown

The file server component, also referred to as F32, manages every file device on a Symbian OS phone; it provides services to access the files, directories and drives on those file devices. This component also contains the loader, which loads executable files (DLLs and EXEs). The loader is covered in Chapter 10, *The Loader*, and in this chapter I will concentrate on the file server.

Overview

Hardware and terminology

Internal drive hardware

We always designate the main ROM drive as *Z:* on a Symbian OS mobile phone. This drive holds system executables and data files and its contents (known as the ROM image) are created by the mobile phone manufacturer when building the device. In fact, the ROM image is normally programmed into Flash memory - Flash is nonvolatile memory that can be programmed and erased electronically. The use of programmable memory for this read-only drive allows the manufacturer to replace or upgrade the ROM image after initial manufacture. In the past, Symbian OS products sometimes used masked ROM to hold the ROM image (or part of it) but this is rarely done now. It takes time to fabricate a masked ROM device with the image and once this has taken place, it is not possible to upgrade the software.

A Symbian OS phone will also have at least one internal drive which provides read/write access, and which the OS uses for the permanent storage of user data. Again, mobile phone manufactures tend to use Flash memory for this internal drive. Indeed, in certain circumstances, the same memory device can be used for both code and user data storage.

Flash memory is made using either NAND or NOR gates - each having significantly different characteristics. Symbian OS supports the storage of code and user data on both NAND and NOR Flash.

Some early Symbian OS products used a RAM disk as the main user data storage device. RAM disks use the same memory as system RAM. Rather than being of fixed size, the system allocates memory to them from the system pool as files are created or extended. Likewise, it frees the memory as data is deleted from the drive. But RAM is volatile storage - data is lost when power is removed. To provide permanent storage, the device has to constantly power the RAM, even when the device is turned off, and it must supply a backup battery to maintain the data, should the main supply fail. Flash memory, on the other hand, retains its contents when power is removed and is also low power and low cost. Because of this, Flash has replaced RAM for permanent user data storage.

Mobile phones do occasionally still make use of a RAM disk, however If the file server finds that the main user-data drive is corrupt when Symbian OS boots, then it can replace this with a RAM disk, providing a temporary work disk to the OS and allowing the main one to be restored. It can then mount the corrupt disk as a secondary drive, which allows a disk utility to recover data, where possible, and then reformat the drive.

Removable media devices

Many Symbian OS phones support removable media devices such as MultiMediaCard (MMC), Secure Digital card (SD card), Memory Stick or Compact Flash (CF). The file server allocates each removable media socket one or more drives, allowing read/write access while a memory card is present. Being removable, these devices have to be formatted in a manner that is compatible with other operating systems. The devices I have mentioned are all solid state rather than rotating media storage devices, but miniature rotating media devices are likely to be used more widely in future, due to their low cost and high capacity. Rotating media devices require more complex power management because of the higher current they consume and their relatively slow disk spin-up times.

I will discuss Symbian OS support for MultiMediaCards in Section 13.5.

File server terminology

Many types of media device, such as MultiMediaCards and SD cards, require every access to be in multiples of a particular sector size, usually 512 bytes. Thus, the sector is the smallest unit that can be accessed.

Other types of media device, such as the ROM, don't have this constraint and allow access in any multiple of a byte.

Throughout this chapter, I will often refer to a media device as a disk. The memory on a disk may be divided into isolated sections, called partitions. Information on the size and location of each partition is generally stored at a known point on the disk - the partition table.

For example, most MultiMediaCards keep a partition table in the first sector of the disk. Even when a device has only a single partition, it will still generally have a partition table. Each separate partition that is made available on a Symbian OS mobile phone is enabled as a different drive. Drives that are allocated to removable media, may, over time, contain different volumes, as the user inserts and removes different removable media devices. So a volume corresponds to a partition on a disk that has been introduced into the system at some time.

F32 system architecture overview

The entire file server system consists of the (shaded) components displayed in Figure 9.1.

The file server, like any other server in Symbian OS, uses the client/server framework. It receives and processes file-related requests from multiple clients. The file server runs in its own process and uses multiple threads to handle the requests from clients efficiently. Clients link to the F32 client-side library (EFSRV.DLL), whose API will be described in Section 9.2. The file server executable, EFILE.EXE, contains two servers - the file server itself (which I will describe in detail in Section 9.3) and the loader server, which loads executables and libraries. I will cover this in Chapter 10, *The Loader*.

Because of the differing characteristics of the various types of disk that Symbian OS supports, we need a number of different media formats. For example, removable disks are FAT formatted to be compatible with other operating systems, and the ROM drive uses a format scheme which is efficient for read operation, but which wouldn't be suitable if writes to the drive were required. In general, the file server does not concern itself with the detail of each file system; instead we implement the different media formats as separate file system components that are *plugged into* the file server. (The exception to this is the ROM file system, which is built into the file server, for reasons that I will discuss in Section 9.1.2.3.) File system components are polymorphic DLLs that have the file extension *.FSY*. These DLLs are dynamically loaded and registered with the file server, normally at system boot time. Figure 9.1 shows a file server configuration with two file systems loaded, ELOCAL.FSY and ELFFS.FSY. I will describe these and other file systems in Section 9.4.

Before a particular drive can be accessed, it must have a file system associated with it, where upon it can be said that the drive is mounted. Again, the file server generally carries out this process at system boot time, once the file systems have been loaded. Mounting also involves determining the basic parameters associated with the drive (drive capacity, free space and so on) and initializing the drive specific data held by the file server. A loaded file system may be mounted on more than one drive.

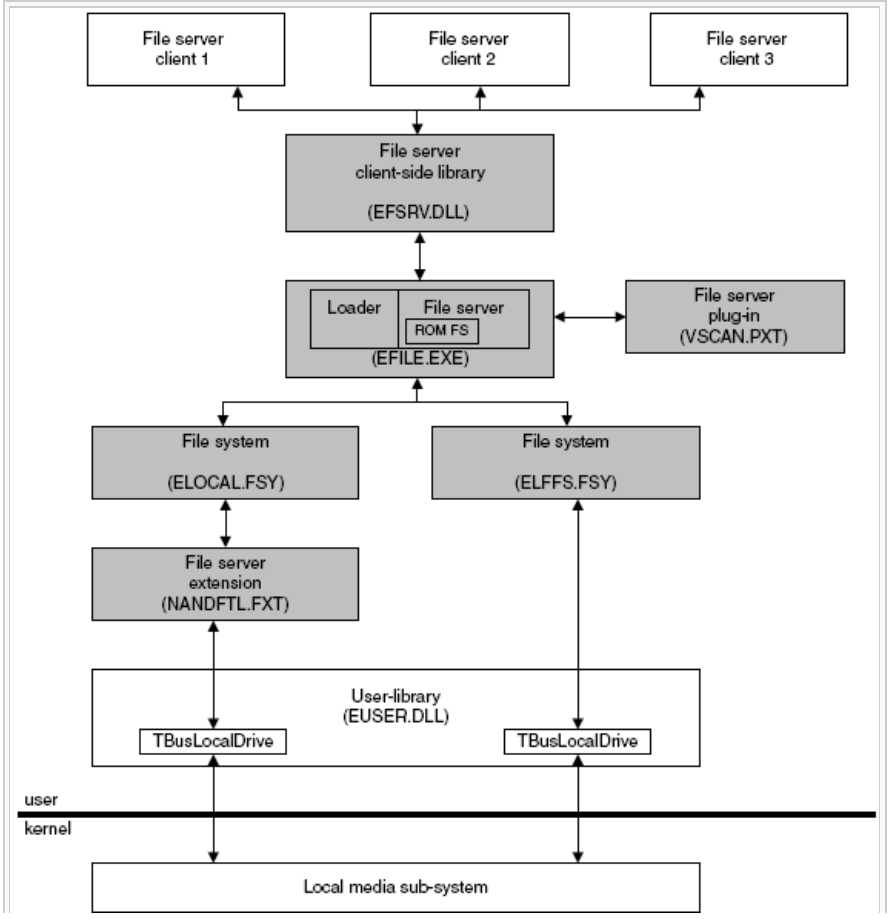

Figure 9.1 *F32 system architecture*

The file systems gain access to the mobile phone's internal and removable disks via a set of device drivers known as the local media sub-system. These drivers are split into a logical device driver layer - the local media LDD (ELOCD.LDD) and a physical device driver layer. The physical device drivers are called media drivers. The user-side interface to the local media sub-system is provided by the class `TBusLocalDrive` whose methods are exported from the user library (EUSER.DLL). The main functions it provides are those to read, write and format regions of each drive's memory area. I describe the local media sub-system in detail in Section 13.3.

Again, the ROM drive is an exception, as we do not access it through the local media sub-system. Instead, the bootstrap maps this memory area to be user-readable, and the file-server accesses it directly.

For other drives though, the `TBusLocalDrive` class provides the user-side interface to the physical media. Often, when a file system is mounted on a particular drive, it will interface directly with the `TBusLocalDrive` instance for that drive. This is the case for the file system ELFFS.FSY shown in the diagram.

However, it is possible to add an extension layer between the file system and the `TBusLocalDrive` object for a particular drive. These extension layers are known as file server extensions and they are executed user-side. They provide away to add functionality to a standard file system that is only required for a certain type of drive. They are built as a separate component, and have the file extension .FXT. The file server provides APIs to install an extension and then associate it with a particular drive. For example, as you will see later, a special translation layer is needed to run a FAT file system over a NAND Flash device. Since this layer is not required when using FAT on a RAM disk or a MultiMediaCard, it would not be appropriate to add this functionality to the FAT file system component itself. Instead, we can implement the Flash translation layer (FTL) as a file server extension (NANDFTL.FXT), and use it only on NAND local drives.

The file server also supports file server plug-ins. Built as separate components, with the file extension .PXT, the plug-ins register with the file server and are then able to intercept requests from any file server clients. Plug-ins can be used to support virus scanning, file compression and file encryption software. The plug-in can set a filter for particular types of request (for example file open) and for requests involving a particular path or drive. When a request is intercepted, the plug-in is given the opportunity to issue its own file server requests using the normal F32 client API (so it can scan the file being opened, for example) before deciding on whether to allow the original request to resume.

Alternatively, the plug-in can fail the request. The file server allows more than one plug-in to register at the same time. It maintains a strict policy on the order in which it notifies the plug-ins, should their request filters coincide. So, for example, if both a virus scanner and a file-compression plug-in were installed, then the compression software would have to decompress a file before the virus scanner scanned it.

Drive letters

The file server supports a maximum of 26 drives, each identified (in DOS-like convention) by a different drive letter (A: to Z:. As I said earlier, the main ROM drive is always designated as the last drive, Z:.

Apart from on the emulator, sixteen of the drives (C: to R:) are normally reserved as local drives - that is, they are available for mounting drives on media devices that are located within the phone. Of these, C: is always designated as the main user data drive, and any removable media device is generally designated as D: or E:.

The remaining 9 drives are available as remote drives, or substituted drives.

F32 on the Symbian OS emulator

On the Symbian OS emulator, any of the 26 drives can be used for mapping to native drives - that is, mapping directories on the host machine's file system to drives on the Symbian OS file server. We do this by using an emulator-specific Symbian OS file system, which converts the Symbian OS file server calls into calls onto the host operating system's file system. There are two default emulator drives - Z:, which represents the target phone's ROM, and C:, which represents the phone's main user-data storage drive. However, by editing the emulator configuration file (usually EPOC.INI), you can reconfigure these drives to map to alternative locations, and you can map additional native drives.

The emulator is generally configured with only these native drives enabled. However, you can enable additional drives that use emulator builds of the file systems that are used on the actual hardware platforms - for example the Symbian OS FAT file system. You can do this either by editing the emulator configuration file or by configuring ESTART (see Section 13.3.1 and Chapter 16, *Boot Processes*). This can be a useful aid when developing a new file system or debugging an existing one. Apart from the ROM file system, all the standard Symbian OS file systems can be used in this way. The majority of the source code for the emulator builds of these file systems is the same as that used for the hardware platforms. Indeed, the same applies for the logical layer of the local media sub-system. However, these drives use emulator specific media drivers which normally read and write to a binary

file on the host machine to emulate the memory area of the disk rather than accessing any media hardware directly.

F32 startup

The file server loads and runs early in the Symbian OS boot sequence, immediately after the kernel has been booted. Obviously, we need a special loading mechanism since at this stage neither the loader nor the ROM file system is available. We use a special statement in the ROM image specification (that is, the ROM obey file), to designate the file server as the secondary process in the system. The result is that the main ROM header holds the address of its image in ROM to allow a kernel extension to start it. (Once the file server is running, the loader can start all the other executables in the system.)

Once the file server is loaded, its main thread runs and installs the ROM file system. Again, without the loader and the ROM drive, this can't be done in the same way as a normal file system - this is why we build the ROM file system into the file server executable, rather than having it as a separate library. Next, the file server starts the loader thread, and then an F32 startup thread.

The startup thread, which is a separate thread to allow it to use the file server client API, now runs and continues file server initialization. It initializes the local media sub-system and then executes ESTART.EXE from Z: before exiting.

ESTART completes the file server initialization process, performing operations such as loading and mounting the file systems. Finally it initiates the startup of the rest of Symbian OS. To do this, it usually launches SYSSTART.EXE, the system startup process, but if this is not present, it launches the window server, EWSRV.EXE, instead. Developers who are creating a new phone platform usually customize ESTART to perform platform specific initialization of the file server or other low-level components. In Chapter 16, *Boot Processes*, I will cover system boot in more depth.

The text shell

Within the F32 source tree, we have implemented a simple text shell (ESHELL.EXE), which presents the user with a DOS-like command prompt. The shell can be used for running console-mode text programs in a minimal Symbian OS configuration that includes only E32 and F32 components. We also provide a minimal window server (EWSRV.EXE) in the E32 source tree too.

The file server client API

We have seen that the file server allows clients to access files, directories and drives on the Symbian OS phone. Client programs access the F32 services by linking to EFSRV.DLL, which provides the client API defined in f32file.h. STDLIB (the Symbian OS implementation of the standard C library) uses this API to add a thin mapping layer to implement POSIX-compliant file services. However, in this section I will describe only the file server client API. For more detail, the reader should refer to Symbian OS C++ for Mobile Phones (Professional Development on Constrained Devices, by Richard Harrison. Symbian Press).

Not surprisingly, most file services involve communication with the file server. (Only a small number are handled client-side, an example being the services provided by the `TParseBase`-derived file name parsing classes - see Section 9.2.4.)

`RFs` class - the file server session

All access from client to file server takes place via a file server session. A session is established thus: the client creates an instance of the file server session class, `RFs`, and connects it to the server using the method `RFs::Connect()`. Clients may have more than one session open simultaneously. The kernel assembles the data for session-based requests into a message and passes a message handle to the server. The file server processes the message and passes the result back to the client.

The file server's clients are normally in different processes to the file server - exceptions are the Symbian OS loader, the F32 startup thread (which I described in Section 9.1.2.3), and file server plug-ins. Because of this, most requests on the file server involve the kernel context switching between processes, which can be an expensive operation. However, the file server is a fixed process, which reduces the impact of the context switch considerably (see Section 7.4.1.3).

The file server doesn't use client-side buffering. This is because the way in which its services are used and the types and formats of files it manages are so varied that there aren't any situations in which this could be consistently beneficial. Instead, higher-level Symbian OS components (such as STORE) implement techniques to minimize the number of calls they make on the file server.

We associate a current path, including a drive letter, with every file server session. Only one path is supported per session. The file server initializes the path when the client connects the `RFs` object, and changes it only as the client directs.

Many of F32's services are provided by the `RFs` class itself. These include the following groups of services:

- Drive and volume information. Examples include:
  - `DriveList()` to get a list of the available drives
  - `Volume()` to get volume information for a formatted device
- Operations on directories and their entries. Examples include:

- `Entry()` to get the entry details of a file or directory
- `GetDir()` to get a filtered list of a directory's contents
- `MkDir()` to create a directory

- Change notification. Examples include:
  - `NotifyChange()` to request notification of changes to files or directories
  - `NotifyDiskSpace()` to request notification when the free disk space on a drive crosses a specified threshold value
- File name parsing. Examples include:
  - `Parse()` to parse a filename specification
- System functions concerning the state of a file system. Examples include:
  - `CheckDisk()` to check the integrity of the specified drive
  - `ScanDrive()` to correct errors due to unexpected power down of the phone
- Management of drives and file systems. Examples include:
  - `SetSubst()` to create a substitute drive - one drive acting as shortcut to a path on another
  - `AddFileSystem()` to add a file system to the file serve
  - `MountFileSystem()` to mount a file system on a drive.

Sub-session objects

A client may use a session to open and access many different files and directories, and F32 represents each of these by a separate object while it is in use by the client. These objects are known as sub-session objects.

There are four sub-session classes:

- The `RFile` class for creating and performing operations on files
- The `RDir` class for reading entries contained in a directory
- The `RFormat` class for formatting a drive
- The `RRawDisk` class, which enables direct drive access.

Creating sub-sessions is much less expensive than opening new sessions. Sub-sessions provide an independent channel of communication within the session. If a client closes a session, the file server automatically closes any associated sub-sessions. This ensures that all file server resources are properly freed.

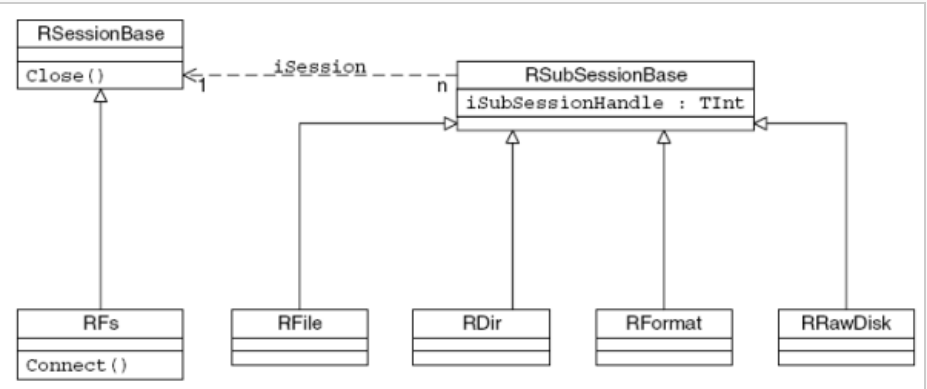

Figure 9.2 *F32 client-side classes*

Figure 9.2 shows these client-side classes. `RSessionBase` is the base class for a handle to a session with any Symbian OS server. We derive from this in the file server to provide the `RFs` class. `RSubSessionBase` is the client-side handle to a sub-session. From this, we derive the `RFile`, `RDir`, `RFormat` and `RRawDisk` sub-session classes that I've already introduced.

As I will explain later, each of these client-side session and sub-session objects has a corresponding object on the server-side. Should the client thread terminate unexpectedly before closing the session, then the kernel will send a disconnect message to the file server which will close the session and all its sub-session objects.

It is good practice to close the sub-session objects as soon as they are finished with - to free the corresponding server-side resources involved. Otherwise they will remain allocated until the owning session is closed.

`RFile` class - for creating and performing operations on files

The `RFile` class represents an individual file. It provides methods to create a new file and open an existing file, as well as methods to read from and write to a file, seek to a position within a file, and get or set a file's attributes.

When a client opens a file, it specifies a sharing mode which indicates how (if at all) other programs can access the same file while it is open. If it opens a file with exclusive access, then no other client can access the file until the first client closes it. Read-only sharing means that other clients may access the file - but only for reading. Finally, a client can open a file and allow shared access with other clients for both reading and writing. If a file is already open for sharing, then another program can only open it using the same share mode as that with which it was originally opened. The client can also specify an access mode indicating how the file will be accessed by this particular `RFile` object. This can specify either read-only access or read/write access. The access mode has to be compatible with the sharing mode.

The `RFile` class also provides methods to lock and unlock a range of bytes within a file. A locked region is accessible only through the `RFile` object that claimed the lock. Locking can be used to synchronize updates to a file when more than one program has read/write access to it.

There are a number of variants of the file read and write methods, including synchronous and asynchronous versions. The synchronous version sends the message to the server and waits for the response, suspending the client thread. The asynchronous version returns as soon as the message is sent, allowing the client thread to continue execution while the server processes the request; the client supplies a reference to a `TRequestStatus` object, which the file server signals on completion.

The `RFile` class also provides methods to read or to change the various attributes of a file, such as `RFile::Size()` to return the size of a file in bytes and `RFile::SetSize()` to change the size of a file.

`RDir` class - for reading directory entries

This class is used to read the entries contained in a directory. Like the `RFile` class, it contains both synchronous and asynchronous read methods.

`RFormat` class - for formatting a drive

This class is only used when formatting a drive. This process can take a great deal of time for a large drive, and so the operation is performed in stages, with control returning to the client at the end of each stage. Stages can be performed synchronously or asynchronously. A drive cannot be formatted while any client has files or directories open on it.

`RRawDisk` class - for direct drive access

This class is used for direct disk access - allowing raw read and write operations to the drive. As with formatting, direct access cannot take place while files or directories are open on the drive.

High level file server services

Symbian OS supports a variety of high-level file server services. We implement this higher-level functionality within the client library rather than the server, and each API generally leads to a sequence of calls being made to the file server.

CFileMan provides file management services such as moving, copying and deleting one or more files in a path. It allows the client to use wildcards in the specification of the paths and files concerned.

CFileMan methods can be configured to operate recursively, which means that they will act on all matching files that they find throughout the source directory's hierarchy. These functions may be performed synchronously or asynchronously. When they operate asynchronously, the operation takes place in a separate thread from the calling client thread. The `CFileManObserver` class allows user notification during the operation.

The `TFindFile` class provides methods to search for files in one or more directories either on a single drive or on every available drive in turn. We provide versions of this that accept wildcards in the file specifier.

CDirScan is used to scan through a directory hierarchy, upwards or downwards, returning a filtered list of the entries contained in each directory.

File names

The file server supports long file names. A full file name may contain up to 256 16-bit characters and consists of four components:

- The drive - a single letter and a colon
- The path - a list of directories separated by backslashes which starts and ends with a backslash
- The filename - this consists of every character from that which follow the last backslash to the character preceding the final dot (if an extension is specified)
- The extension - which consists of every character after the final dot (after the final backslash).

For example: 'c:\dirA\dirB\dirC\file.ext'.

Symbian provides three classes for parsing filenames, each derived from `TParseBase` (the base class for filename parsing). All three classes allow you to test whether a particular component is included in a specified filename, and if so, to extract it:

- `TParse` - this version contains a `TFileName` object as a buffer to store a copy of the parsed filename. `TFileName` defines a descriptor long enough to hold the longest file name. Being 256 characters long, it is a relatively large object and should be used with care. For instance, you should avoid allocating or passing a `TFileName` on the stack
- `TParsePtr` - this version refers to an external, modifiable buffer
- `TParsePtrC` - This version refers to an external buffer that cannot be modified.

The last two versions should be used in preference to the first to minimize stack usage.

Data caging and sharing file handles

The EKA2 version of Symbian OS is normally built with platform security enabled. In this secure version, the file server employs the data caging scheme, which I described in Chapter 8, *Platform Security*.

The central theme of data caging is that the file server designates a certain area on the drives that it controls as a restricted system area. This area is only accessible to programs that are part of the Trusted Computing Base (TCB) - that is the kernel, the file server, and the software installer.

All executables are located within this system area and the OS will refuse to execute code from anywhere else. In addition, each non-TCB process has its own individual private file area that only it (and a small number of other special components) has access to. We provide a third resource area, which is read-only for non-TCB processes and holds read-only files that are to be shared publicly. All remaining file areas are public and any program can read from them and write to them.

So the data caging mechanism allows processes, and the OS itself, to hide private files from other processes. However, there will be circumstances in which a process will need to share a private file with another chosen process. In other words, a situation in which we want to keep a file safe from most processes, but want the ability to grant access to a chosen process without that process having to have special capabilities. Also we don't want to reveal the full (and private) path of the file in order for the recipient to open it (although we can allow the recipient to know the file's name and extension).

To support this, the EKA2 version of the file server provides new `RFile` methods that allow a process to pass an open file to another process. Sharing an open file is a two-stage operation. The owner of the `RFile` object first needs to transfer it to the other process. The `RFile` class provides three methods for this purpose:

- `RFile::TransferToServer()` - for passing from client to server
- `RFile::TransferToClient()` - for passing from server to client
- `RFile::TransferToProcess()` - for passing from one process to another process.

Let's take a closer look at the last of these, as an example:

```
TInt RFile::TransferToProcess(RProcess& aProcess, TInt aFsHandleIndex, TInt
aFileHandleIndex) const;
```

This transfers the `RFile` sub-session object to the process specified by the first argument. In doing this, the file server generates a duplicate handle on this same sub-session object. However, this handle is only useable in the context of the session on which it was created and so the file server must share the session as well as the sub-session object. The duplicate sub-session handle and the session handle are passed to the other process using two of that process's sixteen environment slots. (The slot numbers are specified in the second and third parameters.) The sending process can continue to access the file after it has transferred it. If the sending process closes the file and the session, then the corresponding file server objects are not destroyed - because they have been transferred.

To access the open file that was transferred, the receiving process must adopt the `RFile` object. Again, three different methods are provided, corresponding to the three file transfer methods:

- `RFile::AdoptFromClient()` - for a server adopting a file from a client
- `RFile::AdoptFromServer()` - for a client adopting a file from a server
- `RFile::AdoptFromCreator()` - for one process adopting a file from another.

Again, let's look at the last of these as an example:

```
TInt RFile::AdoptFromCreator(TInt aFsIndex, TInt aFileHandleIndex);
```

This is used to adopt an open file that was sent using the `TransferToProcess()` method. The file server retrieves the session and sub-session handles from the environment data slots specified in the two arguments - these must correspond with those specified in the transfer function. The receiving process can then access the file as it would any other open `RFile` object. The adopted file retains the access attributes that were set when the sending process opened the file.

Although the receiving process shares the session that was used to open and transfer the file, it doesn't have to adopt and manage this shared session directly. Once the receiving process closes the `RFile` object, both the session and sub-session file server objects are destroyed (assuming the sending process has already done likewise).

Because the session is shared between the two processes involved, it is recommended that the sending process opens a file server session specifically for the transfer. Other files opened in the same session by the sending process could be accessible by the receiving process, which could pose a security risk.

The file server

As we have seen, the file server handles requests from its clients for all mounted drives. It is a system server, which means that if it panics, the whole OS is restarted. The main file server thread is always one of the highest priority user threads running on the system.

Interfacing with clients

Figure 9.3 shows a diagram of the server-side classes that form the interface with F32 clients.

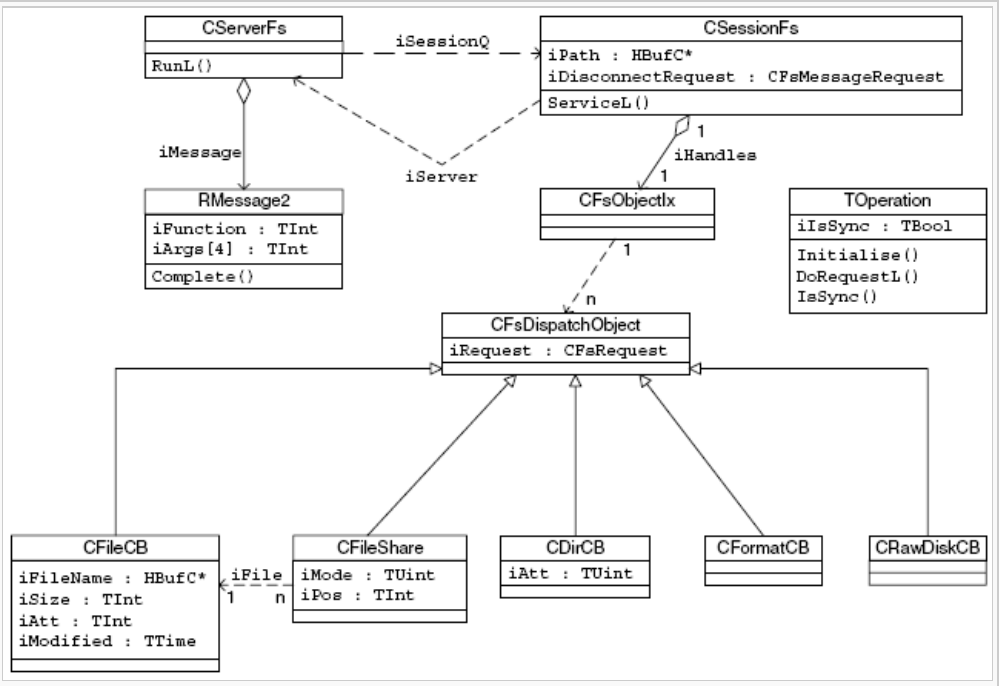

Figure 9.3 *Server-side classes forming the interface with the F32 clients*

CServerFs is a singleton that is derived from `CServer2`, the Symbian OS client/server framework's server class. `CServerFs` is an active object; each time a client request is received, its event-handling method, `RunL()`, is called to accept that request from the client thread and forward it to the relevant server-side client session.

CServerFs handles the creation of the server-side sessions. A client requests a connection by calling `RFs::Connect()`, which sends a connect message to the server. If connection is successful, this leads to the creation of an instance of the class `CSessionFs`, the server-side session object. The session is initialized in the method `CSessionFs::CreateL()`. `CServerFs` maintains a queue, `iSessionQ`, of all the sessions open on it.

Each time the server receives a request from an open session, it calls the request servicing method of the relevant session:

```
void CSessionFs::ServiceL(const RMessage2& aMessage)
```

The details of every client request are contained in a message object, `RMessage2`, which is passed as an argument to the service method. `RMessage2` contains a 32-bit operation identifier, which is read by the server using `RMessage2::Function()`. It also holds a

copy of the request arguments - up to four 32-bit values. After handling a request, the server conveys a 32-bit result back to the client by calling the message object's base class method.

```
void RMessagePtr2::Complete(TInt aReason) const
```

The class `TOperation` encapsulates a type of operation that the server is able to perform. It contains various data members that categorize the operation, as well as two important methods. The first is `TOperation::Initialise()`, which the server calls prior to request execution. This method parses and preprocesses the data supplied by the client. The second method is `TOperation::DoRequestL()`, which performs the requested operation. The server holds a constant array of `TOperation` objects, containing a separate entry for each operation that it is capable of performing. (This is of the order of 90 different operations.)

The session's `ServiceL()` method uses the operation identifier to index this array and obtain the corresponding `TOperation` object for the request.

When a client closes a session, the server will receive a disconnect message and then call this method:

```
void CSessionFs::Disconnect(const RMessage2& aMessage)
```

As we will see shortly, session disconnection can involve more than just deleting the session object.

For each of the client-side sub-session objects that I mentioned in the previous section, there is a corresponding server-side object, and these are all managed by their associated session. The following table lists the server-side sub-session objects and their client/server relationship:

Server-side class	Description	Corresponding client-side class
`CFileShare`	Abstraction for a client view of an open file.	`RFile`
`CDirCB`	Abstraction of an open directory.	`RDir`
`CFormatCB`	Abstraction of a format operation.	`RFormat`
`CRawdiskCB`	Abstraction for direct drive access.	`RRawDisk`

Each time the file server creates a server-side sub-session object, for example because a client calls `RFile::Create()`, it adds this object to the object index (`CSessionFs::iHandles`) of the session to which the object belongs. This generates a unique sub-session handle that the server returns to the client-side object, which stores it. After this, the handle is passed as a message argument in each request involving the same sub-session, to identify the appropriate server-side object. If the client closes the session, the file server will first close any sub-session objects still remaining in the object index. Each sub-session class is derived from `CFsDispatchObject`, which deals with the closing of these objects.

The `CFileCB` class represents an open file - the server has to create an instance of this class before access to a particular file is possible. `CFileCB` contains the full file name (including drive and extensions), the file size, the file attributes and the last time the file was modified.

If you look at the previous table, you can see that an `RFile` object is actually a handle on a `CFileShare` rather than a `CFileCB`. This is because many clients may have the same file open, and `CFileShare` corresponds to one client's particular view of the open file. `CFileShare` stores the current file position for its client, together with the mode in which the file was opened. A `CFileCB` object remains instantiated by the server as long as there are one or more `CFileShare` objects open on it. Once the last share is closed, then the file server closes the `CFileCB` object too.

To further illustrate the relationship between the `CFileCB` and `CFileShare` classes, Figure 9.4 shows two clients, each with files open. Client 1 has a single file open. Client 2 has two files open, but one of them is the same as the one opened by Client 1.

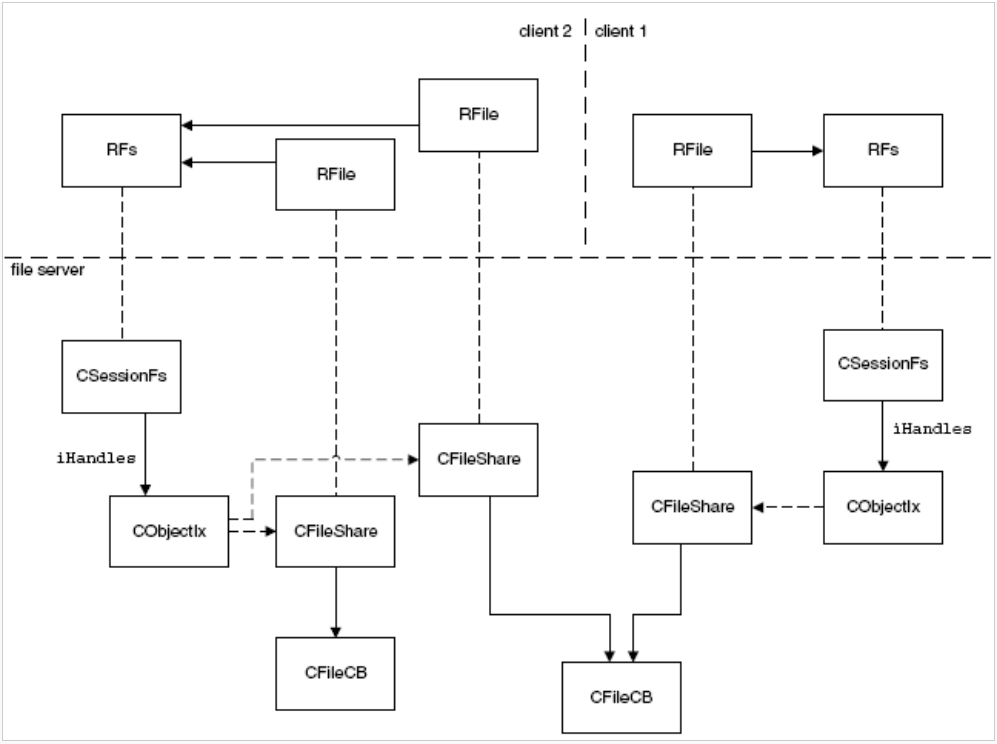

Figure 9.4 *The relationship between* `CFileShare` *and* `CFileCB`

Processing requests

The EKA1 version of the file server is single-threaded. This single thread processes all requests, for all drives. When the thread blocks, waiting on an I/O operation on a particular drive, it is unable to process requests for any other drive.

We took the opportunity to improve the file server design in EKA2. It is multi-threaded and allows concurrent access to each drive. As well as the main file server thread, there is normally a thread for each logical drive, and a thread for handling session disconnection. So, for example, while the server is processing a request to write a large block of data to a multimedia file on a removable media drive, it is still able to accept and process a read request to an INI file on the main internal user data drive. This design also enables the file server to support file systems for remote drives. These are drives that are connected to the mobile phone via a network connection. Requests to a remote drive could take a very long time to complete. Such requests block the thread associated with the remote drive, but, because it is multi-threaded, the file server can still access the other drives in the system.

A client using asynchronous requests can have requests outstanding concurrently on more than one drive from a single session. With the multi-threaded scheme, these can truly be handled concurrently. On EKA1, although the client may be given the impression that they are handled concurrently, in fact they are processed sequentially.

Figure 9.5 illustrates the running F32 threads in a Symbian OS phone that has a single drive. The main file server thread initially handles all client requests. It goes on to service those requests that don't require any access to the media device itself and those that won't block a thread, before completing them and returning the result to the client. These requests must not block since this will delay the entire file server from processing new requests.

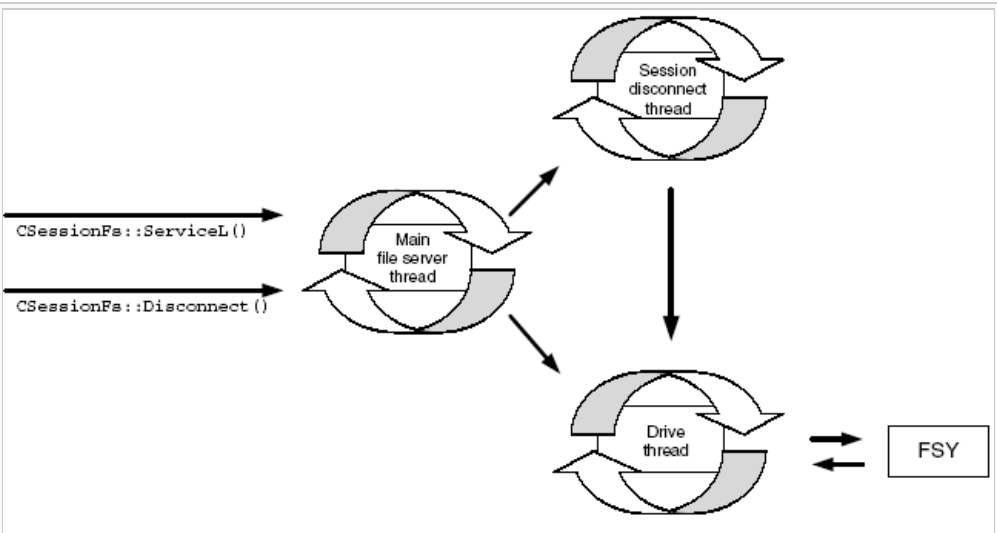

Figure 9.5 *The F32 threads*

The main thread passes any requests that involve a call down to the file system or that may block to a separate drive thread. We allow requests on drive threads to be *long-running* operations. While these drive threads are busy or blocked handling a request, new requests for the drive are added to a drive-thread queue. In most circumstances, it queues requests in a FIFO order. (There is only one exception to this, which I will talk about later.) All drive threads have the same priority, which is slightly less than that of the main file server thread.

There is a certain overhead in transferring requests to a separate drive thread, and so we avoid this where possible. Some types of drive, such as the ROM drive and the internal RAM drive, never perform *long-running* operations and never block the thread. We designate such drives *synchronous drives*, and process all requests for them in the main file server thread - synchronous drives do not have a separate drive thread. However, even with asynchronous drives, we can handle certain requests without access to the media device itself - for example, requests to set or retrieve information held by the file server. We classify these types of operation as *synchronous operations* and the main file server thread always processes these too. (The Boolean member of the `TOperation` class - `iIsSync` indicates which operations are synchronous; see Figure 9.3.) I will now list some examples of synchronous operations:

- `RFs::NotifyChange()`
- `RFs::Drive()`
- `RFs::SetSessionPath()`.

As we have seen, when a client closes a session, this can result in the file server having to close down sub-sessions - and this may mean that it has to write to disk. For example, if closing a `CFileShare` object results in the server closing a `CFileCB` object too, the server may need to flush the current size of the file to the disk. If the file is on an asynchronous drive, then this will have to be handled by the drive thread. Also, before the file server destroys a session, it needs to clean up any outstanding requests for that session - and these may be queued or in progress on one or more drive threads. In this case, we may need to wait for a drive thread to unblock before the requests can be unqueued. Again, we can't tie up the main file server thread while these session termination operations take place, and this is why we use a separate thread to manage session disconnection.

Figure 9.6 shows a diagram of the server-side classes that deal with the processing of a request.

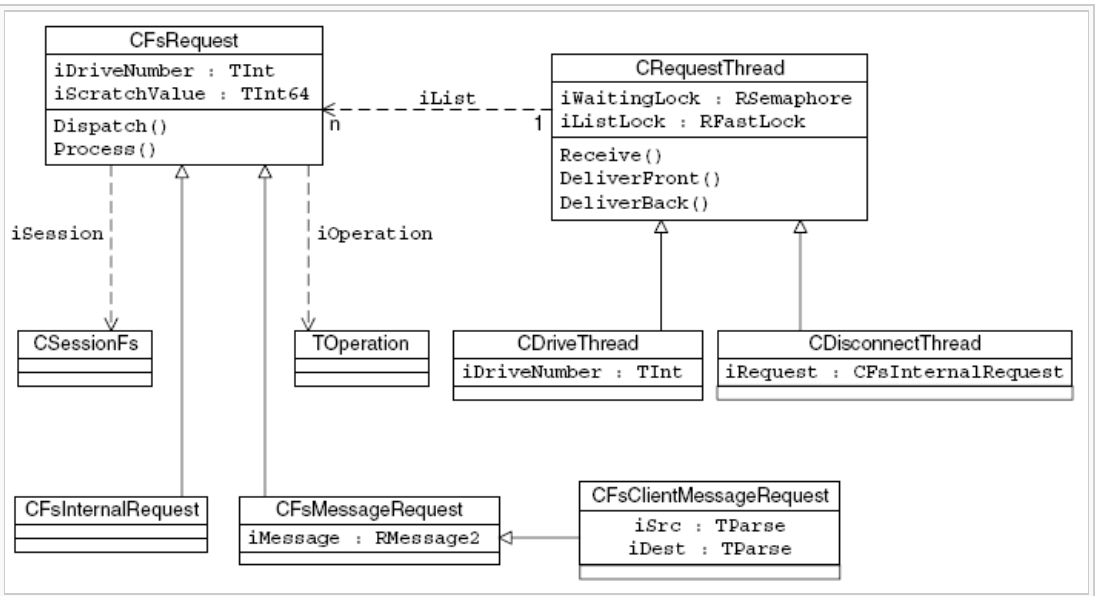

Figure 9.6 *The F32 server-side classes which perform request processing*

Request objects

The abstract class, `CFsRequest`, encapsulates a request within the file server, and we use it to pass these requests from one server thread to another. The initiating thread, which will either be the main file server thread or the session disconnect thread, generates a request object. If the initiating thread cannot handle the request, then this delivers it to the target thread, which will be either a drive thread or the session disconnect thread. The initiating thread may need to store the request in the target thread's request queue until it can be processed. `CFsRequest` has a reference to the corresponding `TOperation` object for the request, `iOperation`. It also has a pointer to the session that originated the request, `iSession`, and a member holding the number of the drive on which the request is to be performed, `iDriveNumber`.

Most requests come from file server clients. However, the server can generate internal requests too:

- `CancelSessionOp`. The session disconnect thread generates this request, and it is delivered to all drive threads, requesting

them to cancel any requests they hold for the session being closed

- `DispatchObjectCloseOp`. This is generated when a sub-session object is closed. As I have already mentioned, sub-session closure can result in a write to disk. Because of this, sub-session closure has to be carried out on the correct drive thread for the object. This means that the initiating thread must issue a DispatchObjectCloseOp request to the appropriate drive thread.

A separate class derived from `CFsRequest` represents each different type of request. `CFsMessageRequest` encapsulates requests originating from a client, and `CFsInternalRequest` represents an internal file server request. Each of these classes has different `Complete()` methods. Completion of a `CFsMessageRequest` results in the request-handling thread signaling back to the client, by calling `RMessagePtr2::Complete()`. Completion of an internal request means that the handling thread will signal the file server thread that initiated the request.

Some client requests involve one or even two file names as arguments, and so `CFsClientMessageRequest`, derived from `CFsMessageRequest` is provided. This contains two `TParse` members to hold this information. The first such member is `iSrc`, which is used by requests which involve a source path name such as `RFile::Create()` and `RFile::Read()`. Requests that involve both a source and a destination path, such as `RFs::Rename()` and `RFile::Replace()`, also use the second `TParse` member, `iDest`.

Each request processed by the server needs a separate request object, and it is the job of the RequestAllocator class to manage the allocation and issuing of new requests. To keep request handling time as short as possible, the request allocator preallocates blocks of empty client request objects. During file server startup, when the first client request is received, it creates the first block of fifteen request objects and adds them to a free list. If ever a request object is required but the allocator's free list is empty, it then allocates a further block of fifteen request objects - up to a maximum of 45. If the allocator ever needs more objects than this, it switches to a new strategy, whereby it allocates and frees individual request objects.

The allocator is responsible for returning the correct type of object for the particular request. When a request object is no longer required, the server informs the allocator, and it returns the object to the free pool.

However, the file server has to be able to handle session and sub-session closure requests without the possibility of failure. This means that for these requests, it must not be necessary for the RequestAllocator to issue new request objects, in case there are none free and there is not enough free memory to allocate a new one. To cope with this, we ensure that the file server always allocates the necessary request objects ahead of receiving a session or sub-session close request. We do this like so:

- Sub-session closure. Each request to open a sub-session results in the request allocator setting aside an internal request object, DispatchObjectCloseOp, to handle sub-session closure
- Session closure. This involves two further types of request in addition to sub-session closure. These are:
  - A request issued from the main thread to the disconnect thread to commence session disconnect. Every `CSessionFs` object has a *SessionDiconnectOp* message request object as one of its private members (iDisconnectRequest) - see Figure 9.3
  - A request to clean up outstanding requests for the session. The session disconnect thread has a CancelSessionOp internal request object as one of its private members (CDisconnect-Thread::iRequest) - see Figure 9.6.

Server threads

As I have already mentioned, as well as the main file server thread, there are two other types of file server threads: the drive threads and the session disconnect thread. Unlike the main thread, which processes each new request as it is received, these other threads may be busy or blocked when a new request arrives, and so they employ a request queue to hold any pending requests. The base class `CRequestThread(shown` in Figure 9.6) encapsulates a file server thread that accepts requests into a queue and then processes them. Requests can be added to either the start or the end of its doubly linked list, `iList`. The fast semaphore `iListLock` prevents access to the list from more than one thread at once. From the `CRequestThread` entry point, the thread initializes itself and then calls `CRequestThread::Receive()` in readiness to receive requests. This method waits for a request to arrive from another thread - at which point it calls the request's `Process()` method. While the request list is empty, `CRequestThread` waits on the semaphore `iWaitingLock`. This is signaled by other threads whenever they deliver a new request and the `CRequestThread` is idle.

The class `CDriveThread`, which is derived from `CRequestThread`, handles requests that are carried out on a particular logical drive. The file server creates an instance of this class for each asynchronous drive on the phone, when the drive is mounted. The FsThreadManager class, which contains only static members, manages drive thread allocation and access. When the file server needs to mount a file system on a logical drive, its main thread calls the FsThreadManager to create the corresponding drive thread. However, the mounting of a file system involves an access to the disk itself, and this can only be done from the correct drive thread, so the main thread then sends a mount request to the newly created drive thread. On drive dismount, the drive thread exits, but the file server does not free the drive thread object.

The class `CDisconnectThread`, also derived from `CRequest`-Thread, handles session disconnect requests. The file server creates

one instance of this class during startup to handle all session disconnect requests. This object is never deleted.

Synchronous request handling

Figure 9.7 shows the program flow for the synchronous request: `RFs::Drive()`. (This request returns drive information to the client.)

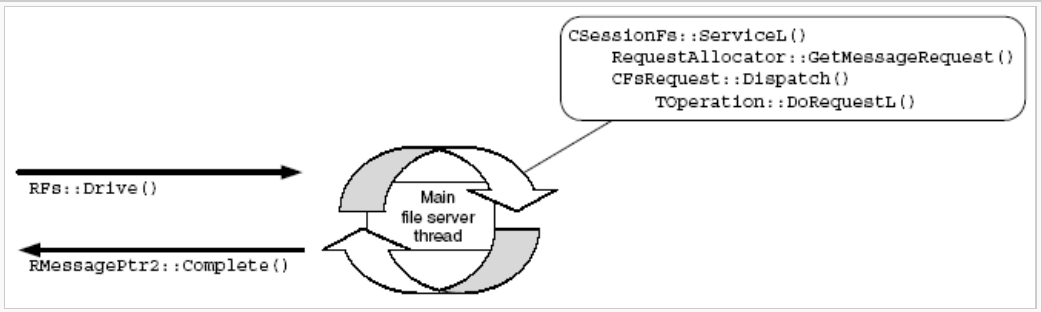

Figure 9.7 *Program flow for a synchronous request*

On receiving the request, the file server calls `CSessionFs::ServiceL()` which executes in the main thread. This acquires a request object, identifies the appropriate `TOperation` object for this request type and then dispatches the request. `CFsRequest::Dispatch()` first calls `TOperation::Initialise()` to validate the arguments supplied by the client. Next it checks whether or not the operation is synchronous using `TOperation::IsSync()`. Since the request is synchronous, the main thread processes the request by calling `TOperation::DoRequestL()`. Finally, the main thread completes the client request by calling `RMessagePtr2::Complete()`.

Asynchronous request handling

Figure 9.8 shows the program flow for the asynchronous request `RFile::Create()` (which creates and opens a file).

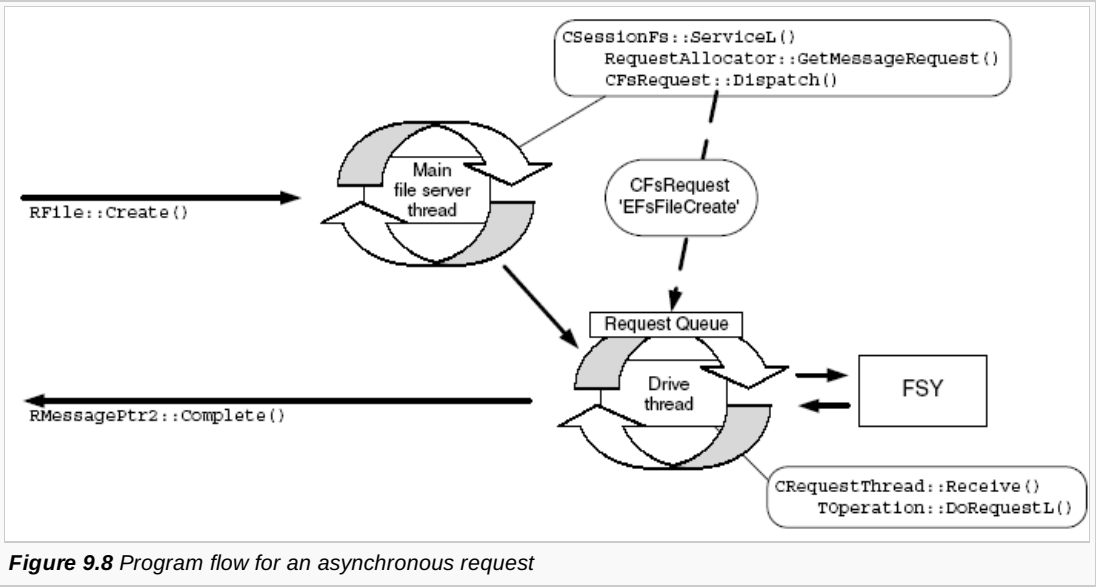

Figure 9.8 *Program flow for an asynchronous request*

On receiving the request, again the file server calls `CSessionFs::ServiceL()` to acquire a request object and initialize it with the appropriate `TOperation` object. Still in the main thread, `CFsRequest::Dispatch()` calls `TOperation::Initialise()`, which parses the name of the file supplied. This time, however, the call to `TOperation::IsSync()` reveals that the operation is asynchronous, and so the main thread dispatches the request to the appropriate drive thread. Once it has done this, it is able to accept other client requests.

When the drive thread retrieves the request from its queue, it processes it by calling `TOperation::DoRequestL()`. This involves interaction with the file system and the underlying media sub-system. Finally, the drive thread completes the client request by calling `RMessagePtr2::Complete()`.

Session disconnection

Figure 9.9 shows the first phase of program flow for session disconnection, `RFs::Close()`. On receiving the request, the file server's main thread calls `CSessionFs::Disconnect()`. This method first closes any open sub-session objects for synchronous drives. (They can't be closed later, when the disconnect thread issues sub-session close requests to the asynchronous drives, because the main thread is not designed to accept internal requests.)

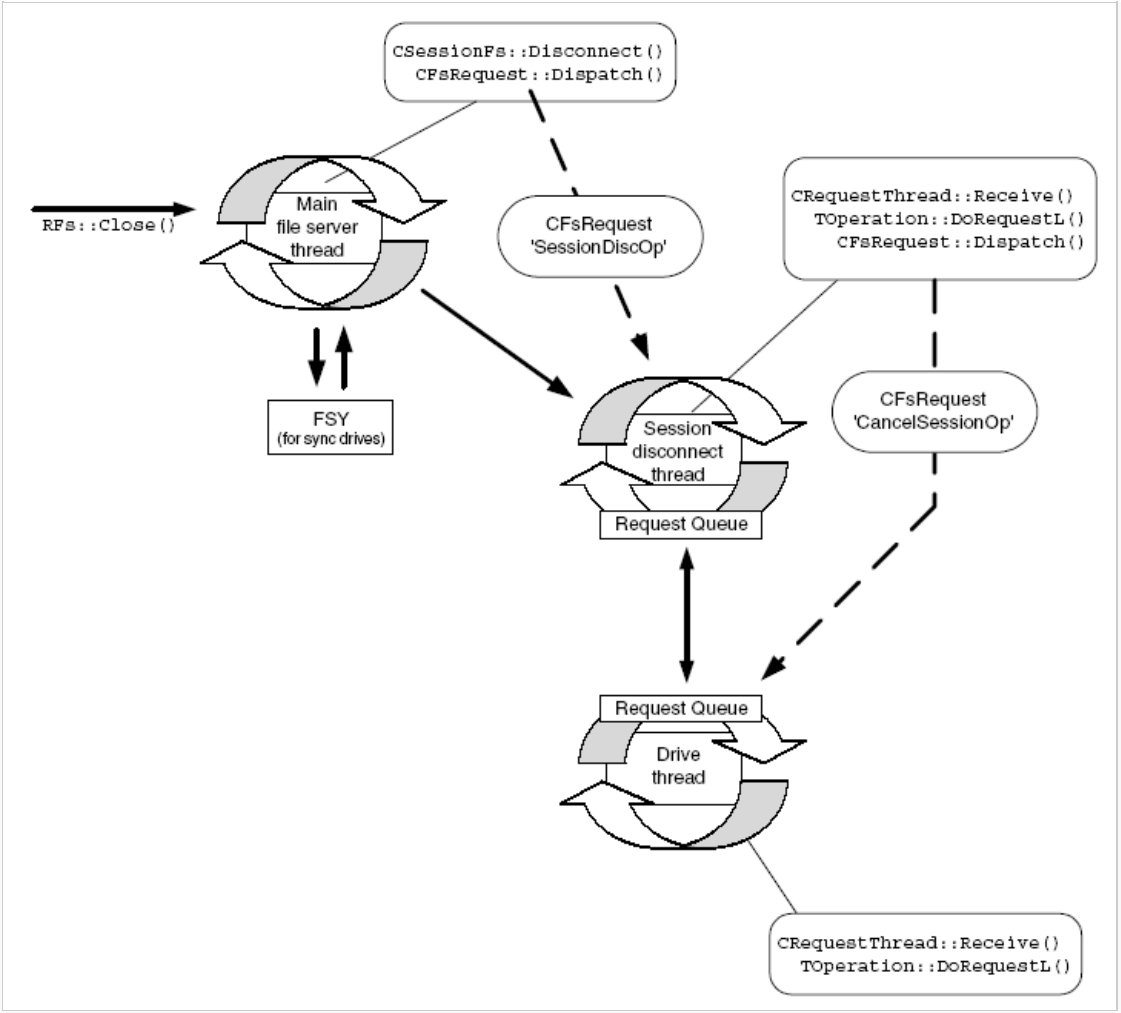

Figure 9.9 *Program flow for the first phase of session disconnection*

The next phase is the cleanup of any outstanding requests for the session. However, the main thread passes the responsibility for this and the completion of session disconnection to the disconnect thread by dispatching a *disconnect session* (SessionDiconnectOp) request to it. (Remember that for session disconnect, the server can't use the request allocator to acquire this request object, and instead it must use the session's request object: `CSessionFs::iDisconnectRequest`. I discussed this in Section 9.3.2.1.*)*

When the disconnect thread retrieves the request from its queue, it issues an internal CancelSessionOp request to each drive thread in turn, asking each thread to cancel any requests queued for the session in question. (Requests in progress will be allowed to complete, since the drive thread doesn't check its queue again until it has completed its current request). The cancel request is inserted at the front of each drive thread queue, so that it will be the next request fetched by each drive thread. Each drive thread will later signal the completion of its CancelSessionOp request to the disconnect thread. Figure 9.10 shows the second phase of the program flow for session disconnection, `RFs::Close()`. Now that the disconnect thread has ensured the cleanup of any outstanding requests for the session, it is able to finish processing the *disconnect session* request it received from the main thread. If any sub-sessions objects remain open - only for asynchronous drives now - then the disconnect thread closes them. Sub-session closure may require the server to write to a disk, so it does this by issuing another internal request, DispatchObjectCloseOp, to each drive concerned.

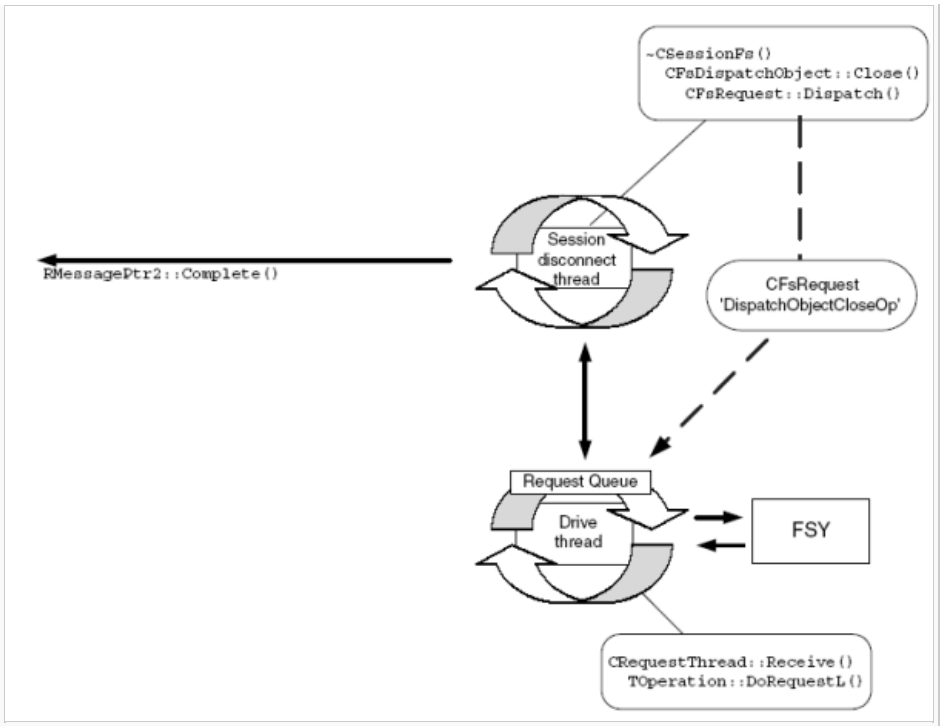

Figure 9.10 *Program flow for the second phase of session disconnection*

Again each drive thread signals back completion to the disconnect thread. Finally, the session-disconnect thread completes the original client request by calling `RMessagePtr2::Complete()`.

Interfacing with the file system

Figure 9.11 shows a diagram of the server-side classes that form the interface with a file system, with a FAT file system implementation shown as an example. I will now describe these classes.

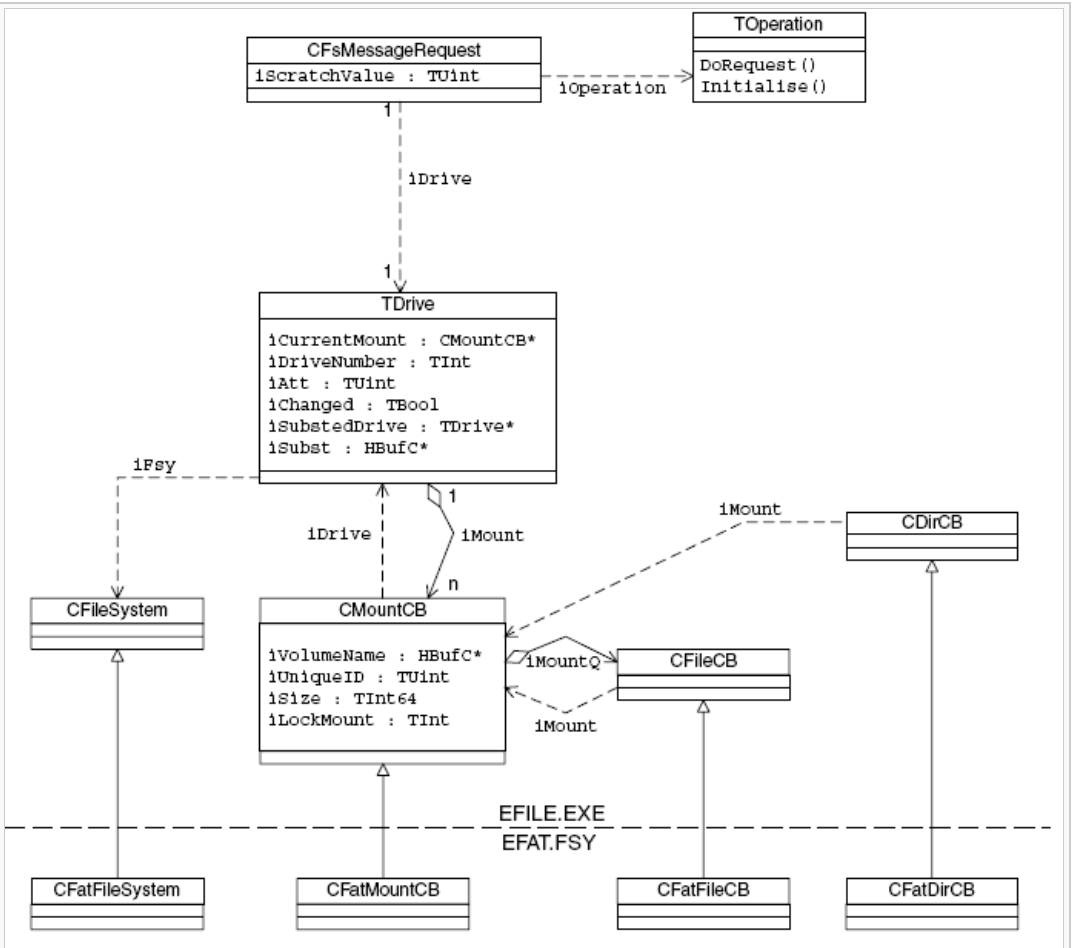

Figure 9.11 *The F32 server-side classes forming the interface with the file system*

`TDrive` class

We've seen that the file server supports 26 drives (A: to Z:); the `TDrive` class is the file server's abstraction for a logical drive. It maintains a `TDrive` instance for each drive, whether or not the drive is currently mounted. As I mentioned in Section 9.1.2, mounting a drive means associating a file system with it. `TDrive` contains the member `iFSys,` which is a pointer to a file system factory class, `CFileSystem`. When `iFSys` is NULL, the drive is not mounted. (Some drives are available for drive substitution but substitute drives have no associated file system.)

The `TDrive` data member `iAtt` holds a bit-mask of the drive attributes. These attributes are fixed while the drive is mounted with a particular file system. The set of possible drive attributes is as follows:

Attribute	Description
`KDriveAttLocal`	Drive is local. Uses the local media sub-system (not ROM or remote).
`KDriveAttRom`	A ROM drive.
`KDriveAttSubsted`	Drive is a substitute to a path on another drive.
`KDriveAttInternal`	Drive is internal (as opposed to removable).
`KDriveAttRemovable`	Drive is removable.
`KDriveAttRemote`	Drive is remote.
`KDriveAttTransaction`	Drive employs a file system which is transactional (this is used by STORE).

In fact, in the context of drives, the term *mount* is a little over-used, since we also talk about a volume being *mounted* on a drive. The class `CMountCB` is an abstraction of a volume (or partition). For removable media devices, the file server creates a different `CMountCB` object for each volume introduced into the system. If the user removes a volume from the phone with a sub-session object still open on it (for example, an open file), then the file server cannot destroy the corresponding `CMountCB` object.

TDrive maintains an object container, `iMount,` holding all the open mounts on its drive. `TDrive` also keeps a separate `CMountCB` pointer, `iCurrentMount,` corresponding to the volume that is currently present on the phone. For a `CMountCB` object to be destroyed, there must be no sub-session objects open on it and it must not be the current mount.

The Boolean member `TDrive::iChanged` indicates a possible change of volume, and is important for removable media drives. At startup, the file server passes down to the local media sub-system the address of `iChanged` for each local drive that is enabled on the phone. The local media sub-system will then update this variable each time there is a card insertion or removal event for the drive concerned.

Each volume contains a unique identifier - for example, FAT partitions contain a unique ID field in their boot sector. The file server reads this ID when it mounts the volume on the drive and stores it in the corresponding mount object, `CMountCB::iUniqueID`. If the user changes the media in the drive, then when the file server next accesses that drive, it will find `iChanged` to be true. The file server then reads the unique ID directly from the new volume to determine if the volume has changed. The server compares the unique ID that it has just read with the ID of each existing mount object stored in the mount queue, to see if it already knows about this volume. If it does, then the corresponding mount object becomes the current mount again. If it does not, then it creates a new mount object.

`CMountCB` class

The volume abstraction, `CMountCB`, has members holding the size of the volume in bytes, iSize, and the volume name, `iVolumeName.` It also has a member `iMountQ,` which is a list of all the files open on the volume.

Its member, `iLockMount,` is a lock counter, which tracks whether files or directories are opened, and whether a format or raw disk access is active on the volume. The server checks `iLockMount` prior to processing format and raw disk requests on the drive, as these can't be allowed while files or directories are still open. Similarly it checks this member before opening files or directories on the drive to ensure that a format or raw disk access is not in progress.

Request dispatch

Now let us look in a little more detail at what happens when a client request is dispatched to a drive thread.

As I described in Section 9.3.2.4, before it dispatches the request to the drive thread, the server's main thread calls `TOperation::Initialise()` to preprocess and validate the data supplied by the client. This may involve assembling a full drive, path and filename from a combination of the data supplied and the session's current path. If the request involves a sub-session object (for example, `CFileShare)` then this process of validation will lead to the identification of the target sub-session object. Rather than discarding this information and recalculating it again in the drive thread when request processing commences, the main thread saves a pointer to the sub-session object in the scratch variable `CFsRequest::iScatchValue` so that the drive thread

can re-use it.

It is also at this initial stage that the main thread translates a request specifying a substituted drive. The data member `TDrive::iSubstedDrive` provides a pointer to the true drive object (or the next one in the chain), and `TDrive::iSubst` holds the assigned path on this drive.

The drive thread commences its processing of the request by calling `TOperation::DoRequestL()`. It identifies the appropriate server object to be used to perform the request (often via the scratch variable). Requests translate into server objects as follows:

Client request	Server object
`RFs`	`CMountCB`
`RFile`	`CFileCB`
`RDir`	`CDirCB`
`RFormat`	`CFormatCB`
`RRawDisk`	`CRawDiskCB`

Request execution continues with the drive thread calling methods on the server object. The first thing it normally does is to check that the target drive is mounted with a volume.

These server object classes form the major part of the API to the file systems. This API is a polymorphic interface - each server object is an abstract class that is implemented in each separate file system DLL. The server manipulates these server objects using the base class's API and this allows it to work with different file systems in different DLLs. In this way, request processing is passed down to the appropriate file system.

Notifiers

As I mentioned in Section 9.2.1, the file server API allows clients to register for notification of various events. These include:

- Standard change notification events:
  - Changes to any file or directory on all drives
  - Disk events such as a drive being mounted, unmounted, formatted, removed and so on
- Extended change notification events. These are changes to a particular file or directory on one or more drives
- Disk space notification events: when free disk space on a drive crosses a specified threshold value.

The client calls to register for notification are asynchronous: they return to the client as soon as the message is sent to the file server. The server doesn't complete the message until the notification event occurs (or the request is canceled). This completion signals the notification back to the client.

The server creates a notification object for each notification request. Figure 9.12 shows a diagram of the classes concerned. `CNotifyInfo` is the base class for each notification. This contains an `RMessagePtr2` member, `iMessage`, which the server uses to complete the request message when the notification event occurs. It also contains the member `iSession`, which is a pointer to the session on which the notification was requested. On session closure, the file server uses this to identify any notifiers still pending for the session and cancel them. `CNotifyInfo` also stores a pointer to the client's request status object, `iStatus`, which the client/server framework signals if the notifier is completed. We need this to handle the client's cancellation of a specific notification request. For example, the client can cancel a request for change notification using `RFs::NotifyChangeCancel(TRequestStatus &aStat)`, where aStat supplies a reference to the request status object of the notifier to cancel. The member `iStatus` is used to identify the specific notifier concerned.

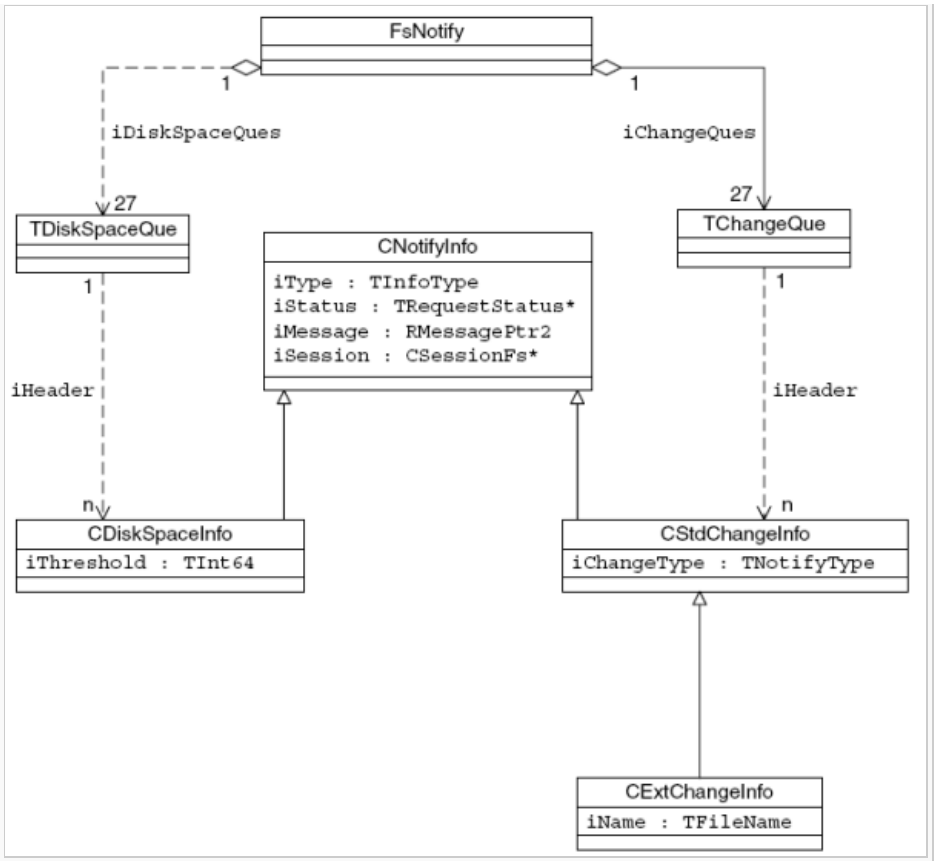

Figure 9.12 *The F32 server-side notifier classes*

We derive `CStdChangeInfo` from `CNotifyInfo` for standard change notifiers. This in turn is a base class for the extended change notifier class, `CExtChangeInfo`. This class's member `iName` holds the name of the specific file or directory associated with the notifier. We also use the `CDiskSpaceInfo` class, which represents a disk space notifier.

The server uses the static class `FsNotify` to manage the notification objects. `FsNotify` keeps two separate sets of queues, one set for change notifiers and one set for disk space notifiers. Each set has 27 queues within it, one for each supported drive and one more to hold notifiers that apply to all drives.

Users of the F32 notification APIs should be aware of the potential for performance degradation. Each time the server completes any request, it may also have to complete a pending notifier. The server checks the `TOperation` object of the request it is completing, to determine if the request type is one which could potentially trigger either a disk space or a change notifier. If it is, then the server iterates through two queues checking for notifiers in need of completion. (Two queues because one is for the current drive, and one is for notifiers that apply to all drives.) In the case of extended change notifiers, a certain amount of pathname comparison is required, and for disk space notifiers, the amount of free disk space needs recalculating. If there are a large number of notifiers pending, then this can have an impact on file server performance.

File systems

The file server receives all client requests, but it never accesses the media devices directly. Instead, it passes any requests that require access to the directories and files on the device to a file system. Each file system employs a media format which is appropriate for the characteristics of the devices that use it. Most file systems are implemented as separate file server plug-in DLLs, the exception being the ROM file system, which is built as part of the file server itself. File system DLLs are assigned the file extension: FSY.

Symbian OS supports the following file systems:

- The ROM file system is used for code storage on execute-in-place (XIP) media such as NOR Flash. XIP refers to the capability to execute code directly out of the memory
- The log Flash file system (LFFS) for user-data storage on NOR Flash
- The FAT file system for user-data storage on NAND Flash, internal RAM drives and removable media
- The Read-Only file system (ROFS) for code storage on non-XIP media such as NAND Flash. Code on non-XIP media first has to be copied into RAM for execution.

It is possible for developers to implement their own file system. Normally they would then customize ESTART to load and mount this file system during file server startup. I describe the necessary APIs in Section 9.4.1.6.

File system API

As I mentioned in Section 9.3.3.3, a loadable file system is a polymorphic DLL providing plug-in functionality to the file server by implementing the predefined file system interface classes, which are abstract classes. Each file system implements the API by defining and implementing concrete classes derived from each abstract class.

File systems are dynamically loaded by the file server at runtime - a new one can be added and mounted on a drive without the need to restart the server or interrupt any connected file server sessions. File systems contain a single exported function, which the file server calls when the file system is added to it. This export is a factory function that returns a pointer to a new file system object - an instance of a `CFileSystem` derived class. The `CFileSystem`-derived class is itself a factory class for creating each of the other file system objects. Having called this export, the server is able to call all other file system functions through the vtable mechanism.

The file system API is defined in f32fsys.h. In the following sections, I will discuss the classes of which it is comprised.

The `CFileSystem` class

This is a factory class, which allocates instances of each of the other objects that form the file system API: `CMountCB`, `CFileCB`, `CDirCB` and `CFormatCB`. The file server has only a single `CFileSystem` instance for each loaded file system - even when that file system is mounted on a number of drives.

The `CMountCB` class

This class, which I introduced in Section 9.3.3.2, is the abstraction of a volume. The file server creates an instance of `CMountCB` for each volume introduced into the system.

The functionality that the file system supplies through this class roughly corresponds to that contained within the `RFs` class. So, taking as an example the method `RFs::MkDir()` to make a directory, we find that program flow moves from this client function into `TDrive::MkDir()` in the file server and then on to `CFatMountCB::MkDirL()` in the FAT file system (FAT.FSY). Here is another example:

```
Client DLL File server FAT.FSY
RFs::Rename() . TDrive::Rename() . CFatMountCB::RenameL()
```

The `CFileCB` class

This class represents an open file. The functionality this class supplies roughly corresponds to the methods contained in the `RFile` class. Again, let's follow the program flow using the FAT.FSY as an example:

```
Client DLL File server FAT.FSY
RFile::Read() . CFileShareclass . CFatFileCB::ReadL()
```

The file server has a single object container, which references every `CFileCB` object, across all the drives in the system.

The `CDirCB` class

This class represents the contents of an open directory. This supplies functionality corresponding to the methods contained in the `RDir` class.

Again, the file server has a single object container that references every `CDirCB` object across all the drives in the system.

The `CFormatCB` class

This class represents a format operation.

Loading and mounting a file system

We add file systems to the file server by calling the client method:

```
TInt RFs::AddFileSystem(const TDesC& aFileName) const
```

The argument aFileName specifies the name of the FSY component to be loaded. As I mentioned in Section 9.1.2.3, ESTART normally does file-system loading during file server startup. Once it has been successfully added, a file system can be mounted on a particular drive using the method:

```
TInt RFs::MountFileSystem(const TDesC& aFileSystemName, TInt aDrive) const
```

In this method, aFileSystemName is the object name of the file system and aDrive is the drive on which it is to be mounted.

The EKA1 version of the file server requires a nominated default file system, which must be called ELOCAL.FSY. The EKA2 version of the file server places no such restriction on the naming of file systems, or in requiring a default file system.

If you are developing file systems, there are two methods available which are useful for debugging:

```
TInt RFs::ControlIo(TInt aDrive, TInt, TAny*, TAny*)
```

This is a general-purpose method that provides a mechanism for passing information to and from the file system on a specified drive. The argument aDrive specifies the drive number, but the assignment of the last three arguments is file system specific.

Additionally, the following method can be used to request asynchronous notification of a file system specific event:

```
void RFs::DebugNotify(TInt aDrive, TUint aNotifyType, TRequestStatus& aStat)
```

The argument aDrive specifies the target drive number, aNotify-Type, specifies the event, and aStat is a reference to a request status object that is signaled when the event occurs. To trigger the notifier, the file system calls the following method, which is exported by the file server:

```
void DebugNotifySessions(TInt aFunction, TInt aDrive)
```

The argument aFunction specifies the event that has occurred and aDrive indicates the drive on which this has occurred.

So for example, if when testing, it is required for the test program to issue a particular request when a certain condition occurs in a file system then using `DebugNotifySessions()`, the file system can be configured to complete a pending debug notification request whenever the condition occurs.

All these methods are only available in debug builds.

The log Flash file system (LFFS)

I introduced Flash memory in Section 9.1.1, where I mentioned the different types of Flash that we support in Symbian OS. We designed the log Flash file system to enable user-data storage on NOR Flash devices.

NOR Flash characteristics

Flash is nonvolatile memory which can be erased and rewritten. Reading from NOR Flash is just like reading from ROM or RAM. However, unlike RAM, data cannot be altered on Flash just by writing to the location concerned. Flash must be erased before a write operation is possible, and we can only do this erasing in relatively large units (called blocks). To erase, the phone software must issue a command and then wait for the device to signal that the operation is complete. The erase sets each bit within a block to one. Write operations, which are issued in the same way as erases, can then change bits from one to zero - but not from zero to one. The only way to change even a single zero bit back to a one is to erase the entire block again.

Imagine that we need to modify just one byte of data in a block, changing at least one bit from zero to one. (Assume also that we cannot perform the modification by writing to an alternative location in the block that has not yet been written to.) Then, to do this, we have to move all the other valid data in the block to another freshly erased location, together with the updated byte. The new location now replaces the original block - which can then be erased and becomes available for reuse later.

Another characteristic of Flash that we had to consider in our design is that it eventually wears out - there is a limit to the number of times a block can be erased and rewritten.

The log

The LFFS is specifically designed to operate with NOR Flash and to protect itself against power loss. To do this, it keeps a log of all its operations (hence the *log* part of the name). It records each modification to the data in the file system by adding an entry at the end of the log describing the operation. So, if a new file is created, this information is added as a log. If the file is subsequently deleted, a log entry indicating that the file is no longer available is added to the log.

Each log entry is of fixed size (32 bytes) and includes a flag that indicates the completion status of the operation. Before each operation is started, the LFFS creates its log entry which it adds to the log with a completion status of *not complete*. It then performs the operation, and only when this is fully complete does it modify the status in the log entry to *complete*. If an operation is incomplete when power is removed then, when power is restored, the LFFS undoes the operation and any space it had consumed is reclaimed. This system ensures that power loss does not corrupt the file system - although data that is only partially written is lost.

The LFFS uses the key characteristic of NOR Flash to implement this scheme. We've seen that generally we can't change Flash contents without a prior erase cycle. However, we implement an *incomplete* flag status using bits in the one state, and so we can rewrite this flag to zero (the *complete* state) without the need for an erase.

A set of operations are often related to each other, in that the whole set must either be completed, or the set of operations should fail. In other words, all the changes must be committed atomically. As an example of this, consider a large file write involving several data blocks. To handle this requirement, the LFFS uses a transaction mechanism. It marks all log entries that are part of the same transaction with the transaction ID number. It also marks the first entry with a transaction start flag, and the last entry with a transaction end flag. This ensures that partial transactions are never regarded as valid. Either the transaction succeeds and all the associated operations are valid, or the transaction fails and all the operations are invalid. As I mentioned earlier, the LFFS undoes invalid operations, and reclaims the space they consume.

File and directory structure

Normal file and directory data storage is completely separate from the log. This data is arranged into File Data Blocks (FDBs), which are, by default, 512 bytes in size. However, you could build the LFFS to use larger blocks (up to 4 KB) by changing a constant in one of its configuration header files. Although using a fixed data block size is wasteful of memory for small files, this allows the FDB pointer information to use an FDB index rather than an absolute address, which reduces the data management overhead.

Each FDB has an associated log entry that describes the purpose of the block and provides a pointer to it. However, the log is mainly intended as a record of changes and does not provide a permanent mechanism to track the FDBs that hold a file's data. Instead, the LFFS uses three structures to hold this information.

The first of these structures is an I-node. Each file has a single I-node that holds file-specific data, such as the file type, the file size and a unique I-node number (which is essential in identifying the file).

An I-node also contains fourteen FDB pointers. These are known as the direct pointers, and they hold address information for up to fourteen FDBs that make up the file. With an FDB size of 512 bytes, this structure alone can handle files of up to 7 KB. For larger files, a second structure is involved - the indirect block (IDB). IDBs contain 64 pointers, each addressing either FDBs or further IDBs. The LFFS supports up to four layers of IDBs, giving a maximum file size of approximately 8 GB. An I-node has an indirect pointer for each layer of IDBs.

The organization of a file with a first-level IDB is shown in Figure 9.13. The LFFS gives the FDBs in a file sequential numbers, starting at zero. It gives IDBs the same number as the first FDB that they point to.

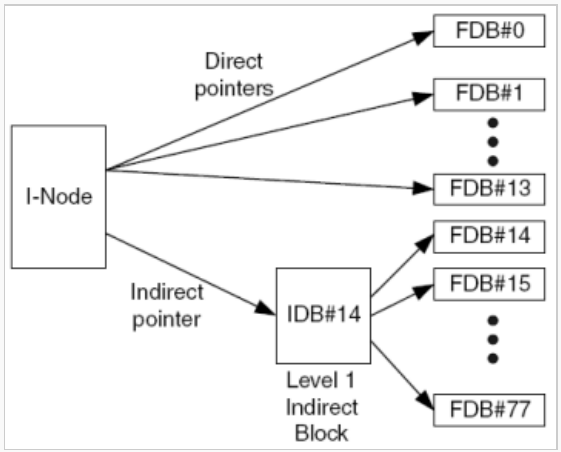

Figure 9.13 *The organization of files in the LFFS which uses a first level IDB*

The following table lists the fields contained in an I-node:

Field	Size (in bytes)	Description
I-node number	4	The I-node number of the file.
Reference count	2	The number of directory entries referring to this I-node. This is always 1.
File type	2	The type of file referred to by the I-node. The value can be any of the following: 1 = User data file 2 = Directory file 3 = Metadata file.
File length	4	The number of data bytes in the file.
Data block size	4	The size of the FDBs referred to by the I-node and IDBs.
Direct pointers	4 * 14	Pointers to the first 14 FDBs in the file. The first pointer is to FDB#0, the second is to FDB#1, etc.
Indirect pointer L1	4	Pointer to a level 1 IDB. The IDB contains pointers to the FDBs following those found through the direct pointers.
Indirect pointer L2	4	Pointer to a level 2 IDB. The IDB is the root of a 2 level tree with pointers to the FDBs following those found through Indirect pointer L1.
Indirect pointer L3	4	Pointer to a level 3 IDB. The IDB is the root of a 3 level tree with pointers to the FDBs following those found through Indirect pointer L2.
Indirect pointer L4	4	Pointer to a level 4 IDB. The IDB is the root of a 4 level tree with pointers to the FDBs following those found through Indirect pointer L3.

The LFFS uses a third structure to track the I-nodes: the LFFS partition contains a single I-file, which holds an array of pointers to the I-nodes. It adds new I-node references to the I-file at the array entry given by the I-node number. When it deletes a reference to an I-node from the I-file, it sets the array entry to zero. This indicates that the I-node is not in use any more, and a new file can reuse the I-node number.

Collectively, these FDB tracking structures are known as the LFFS metadata. However, the metadata doesn't hold filename or directory information. Instead, we store this information in directory files. These are really just normal data files, except that they are used by the file system, and are not directly visible to clients. A directory file contains an entry for each file in that directory. Directory entries contain the name of the file and the number of the I-node that points to the file's data blocks. A directory entry's size depends on the length of the filename, which can be at most 256 characters.

I-node number 2 always points to the root directory.

Segments

To manage the erasing of blocks, the LFFS uses the notion of a segment. A segment is the smallest unit of media space that the file system can erase and consists of one or more consecutive erase blocks. The LFFS views the entire NOR device as a consecutive series of segments. It stores log entries and data entries (file data blocks and metadata) into each segment.

It stores the log starting at the bottom of the segment growing upwards, and the data entries at the top, growing downwards. Figure 9.14 shows the layout of a segment.

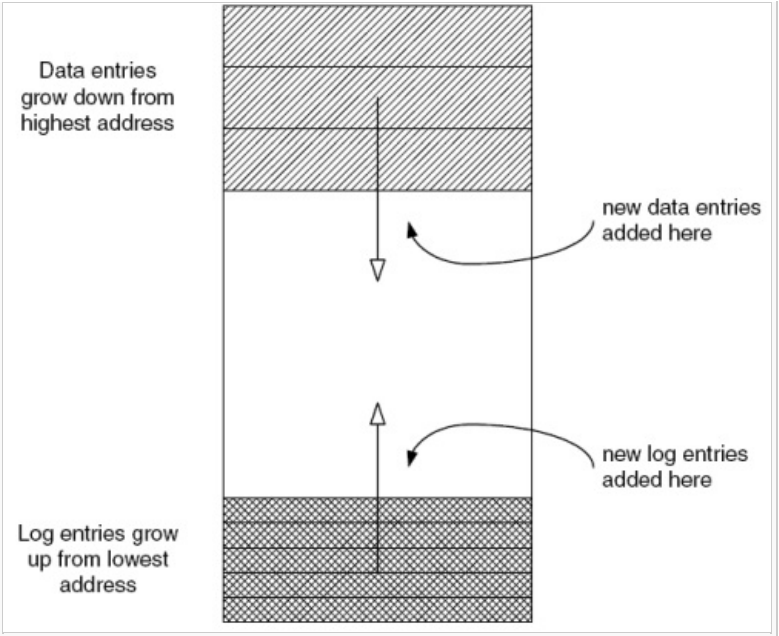

Figure 9.14 *The layout of an individual LFFS segment*

In this way, the log is split across segments, but log entries always occupy the same segment as their corresponding data entries. The LFFS adds log and data entries to the current segment until that segment eventually becomes full - at which point it moves on to the next erased segment.

Figure 9.15 shows a section of an LFFS partition containing four segments. The segment 2 is the current segment. Segment 1 is full, but segments 3 and 4 are empty.

As it adds and modifies file data, the LFFS moves on from one segment to the next, until it approaches a point where it is running out of media space. However, the total amount of valid data on the device will almost certainly be much less than the capacity of the device.

Reclaiming outdated media space

When file data is modified, the LFFS has to replace each FDB affected. It adds the replacement FDBs together with their associated log entries to the current segment. At this point, the old FDBs and associated log entries have become out-dated. The LFFS will eventually have to reclaim this space to allow further updates to the drive. However, the LFFS has not yet finished the file update, since it must also change the metadata to point to the new FDBs. This means modifying the file's I-node and IDBs - which will generate yet more out-dated log and data entries. (However, as I will explain in Section 9.4.2.6, this metadata update is deferred until later.) When file data is deleted, this also leaves out-dated log and data entries, which the LFFS needs to reclaim.

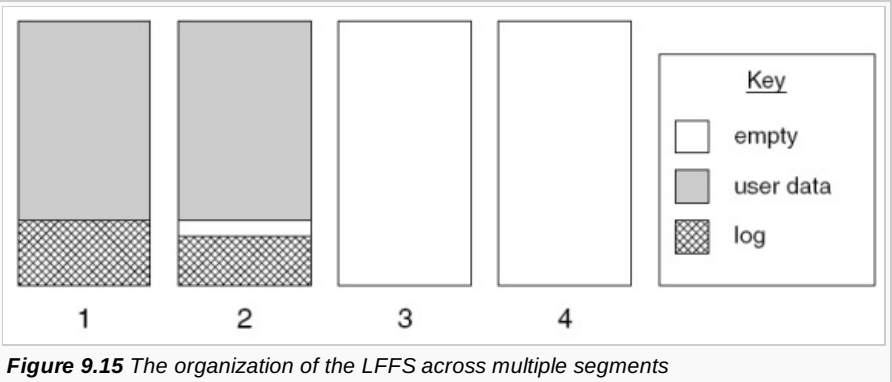

Figure 9.15 *The organization of the LFFS across multiple segments*

Reclaiming out-dated media space is not simple, as this space will normally be spread across many segments, and these segments will also contain valid data. The reclaim process has to identify the segment with the largest amount of dirty data, copy any valid data from that segment, and then erase the segment allowing it to become active again. The LFFS can't allow the device to run out of free space before it attempts a reclaim, because it needs reserve space into which to move the valid data before erasing. It has to reserve at least a segment to ensure this does not happen.

The choice of which segment to reclaim is actually more complex than I have just described. I mentioned earlier that Flash blocks have a limit on the number of erase cycles they can handle. To mitigate this, the LFFS employs a wear-leveling scheme. This scheme aims to keep the erase count of each of the segments roughly equal, to avoid premature failure due to some blocks being used more than others. The LFFS stores the erase count of each segment in a segment header, and the reclaim algorithm takes

this into account when identifying the next segment for reclaim. It avoids a segment containing a lot of dirty data but with a high erase count.

Reclaiming a segment can be a slow process. First, the LFFS must copy valid data to another segment, and then it must erase the blocks contained by that segment. If the LFFS did not perform space reclamation until it needed space to service a request, then its performance would be poor. Instead, the LFFS tries to reclaim early by using a reclaim thread.

This is a low priority thread that uses what would otherwise be CPU idle time to reclaim data.

Roll forward

When the user modifies a file, the LFFS does not update the metadata as part of the same transaction that updates the FDBs. Instead it defers the metadata update until later. It can do this because it can extract the information it needs from the log. We call this update of metadata from the log *roll-forward*. To improve performance, the LFFS generally performs roll-forward in batches, using the same low priority background thread that it uses to perform early reclaim. If power is lost after the LFFS has updated the FDBs, but before it updates the metadata, then the write operation is still regarded as successful, because the LFFS can rebuild the metadata from the log when power is restored.

However, the LFFS cannot allow too many metadata updates to accumulate, because this affects reclaiming. This is because it cannot reclaim any segment that contains FDBs whose metadata it has not rolled forward, because this reclaim could destroy information needed to reconstruct the metadata.

Caching

The LFFS maintains a cache which holds both read data and pending write data. This cache includes all types of data - not only FDBs, but also metadata and log entries. The cache tracks the pending write data for each file separately, and can therefore act on cached data from only one file. For example, data for a single file can be flushed without having to flush data for any other file. If you are building the LFFS for a new phone platform, you can configure the size of this cache by changing a constant in one of the LFFS configuration header files.

There is also a separate read-ahead cache that reads log entries in groups, for efficiency.

Error recovery

If, during a reclaim, the LFFS detects an erase failure on a block, then it marks the entire segment that contains that block as being bad (and no longer available for use). Again, it stores this information in the segment header. The LFFS then identifies another segment for reclaim, leaving the client unaware of the error.

If the LFFS detects a write failure on the current segment, then it reattempts the same write in the next available position on the same segment. It repeats this, moving further through the segment until the write eventually succeeds. This may require the LFFS to move on to the next free segment. Again, the client is unaware of any error. If it has suffered a write error, it is likely that the damaged sector will also suffer an erase error if it is ever reclaimed, which will cause it to be marked as bad at that point. However, assuming the damaged sector is still capable of being read, any valid data it contains will be moved to another segment as part of the normal reclaim process.

By configuring one of the LFFS configuration header files, developers who are creating a new phone platform can build the LFFS so that it keeps segments in reserve to replace bad segments, so that drive capacity isn't reduced in erase failure situations.

The FAT file system

Symbian OS uses the VFAT file system for user-data storage on various media types including removable media, internal RAM and NAND Flash.

On a FAT-formatted volume, the data area is divided into clusters, with a cluster being the smallest unit of data storage that can be allocated to a file. On a given volume, each cluster is the same size and is always a whole number of sectors (see Section 9.1.1.3). Cluster size varies between volumes, and depends on the size of the volume.

A File Allocation Table (FAT) structure is used to track how clusters are allocated to each file. Since the original version of the FAT file system became available, it has been enhanced to support larger volumes. Now there are different FAT file system types, corresponding to differences in the size of the entries used in the FAT data structure. These include FAT12 for 12-bit entries, FAT16 for 16-bit entries and FAT32 for 32-bit entries. Directory information is stored in a directory table, in which each table entry is a 32-byte data structure. The VFAT version of the standard supports long file names, up to 255 characters in length - previous versions supported only *8.3* filenames that can only have eight characters, a period, and a three-character extension. The FAT and VFAT standards are described in many publications and I will not describe them in any more detail here.

To ensure that Symbian OS phones are compatible with other operating systems, it was essential that we used the FAT file

system on our removable media. Data on a removable media card that has been formatted and updated on a Symbian OS phone must be fully accessible when removed and introduced into other computer systems, such as a card reader connected to a PC - and vice versa. In some cases, compatibility with other systems can also mandate that internal drives be FAT formatted.

This is because some Symbian OS phones support remote access from a host machine (such as a desktop computer) to one or more of their internal drives, by exposing these as USB mass storage devices. For this to work without a special driver on the host machine, these internal drives must be FAT-format.

Symbian OS always formats internal RAM drives using FAT16.

Symbian OS actually provides two builds of the FAT file system - one that supports only FAT12 and FAT16, and a second that supports FAT32 as well. This is because not all Symbian OS licensees require support for FAT32.

Rugged FAT version

We build both FAT12/16 and FAT32 in a rugged configuration by default - although developers who are creating a new phone platform can enable the standard version if they wish. The rugged FAT version provides tolerance to situations in which write operations are interrupted due to power failure. This only happens when there is an unexpected loss of battery power, not from the user's normal power down of the phone, because in this case the file server can complete the operation before turning off.

The rugged version alters the way in which the file system updates the FAT tables and directory entries, but this alone can't completely eliminate the possibility of an error due to unexpected power removal. However, it also incorporates a ScanDrive utility, which the file server runs when power is restored to the disk - and this can fix up any such errors. For example, ScanDrive can detect and correct inconsistencies between the FAT table and the directory entries, to reclaim lost clusters. It does this by generating its own version of the FAT table from analyzing the directory entries and then comparing this with the current FAT table. ScanDrive runs at system boot, but only if the phone has been unexpectedly powered down. In normal power down situations, the Symbian OS shutdown server (SHUTDOWNSRVS.EXE) sends notification to the file server of orderly shutdown, using the F32 method `RFs::FinaliseDrives()`. The file server passes notification down to the FAT file system, and this is then able to update a status flag on the FAT disk. When it starts up, the file system checks the status flag, allowing it to only run ScanDrive when it is needed.

However this power-safe, rugged scheme only applies if the underlying local media sub-system can guarantee atomic sector writes - that is, a sector involved in a write operation is never left in a partially modified state due to power removal, but is either updated completely, or left unmodified. We can provide this guarantee for internal FAT drives that use a translation layer over NAND Flash.

For removable media devices, unexpected power removal may result from the removal of the card from the phone, as well as from the loss of power from the battery. For these devices, we can't usually guarantee atomic sector writes. However, we can minimize the possibility of card corruption due to unexpected power removal, using schemes such as not allowing writes when the battery voltage is low. Another way of minimizing corruption is the use of a card door scheme that provides early warning of the possibility of card removal. Unfortunately these schemes can't catch everything - consider the situation where the phone is accidentally dropped, resulting in the battery being released during a write operation. We could protect against corruption in this scenario with the use of a backup battery that preserves system RAM while the main battery is missing, so that we can retry the original write operation when the main supply is restored. Unfortunately, the use of a backup battery is not popular with phone manufacturers, due to the increase in the bill of materials for the phone.

Caching

When we use the FAT file system on either removable media or NAND Flash drives, we always employ two caching schemes to improve performance.

The first of these is a cache for the FAT itself. The file system caches the entire FAT into RAM, for all drives formatted using FAT12 or FAT16, and for any drives formatted using FAT32 whose FAT is smaller than 128 KB. This is a *write-back with dirty bit* type of cache scheme, with the file system flushing all dirty segments at certain critical points throughout each file server operation. This cache is used so that short sequences of updates to the FAT can be accumulated and written in one go. The frequency of cache flushes is higher, for a given operation, if the file system is configured for rugged operation. The segment size within this cache is 512 bytes. This corresponds to the smallest unit of access for the sector-based media that the cache is used for.

However, for larger drives formatted with FAT32, the size of the FAT becomes too large for it to be entirely cached into RAM. Instead, only part of the table is cached. The cache stores up to 256 segments and employs an LRU (Least Recently Used) scheme when it needs to replace a segment. Each segment is still 512 bytes long.

The other type of cache, which we use on NAND Flash and removable media drives, is a metadata cache. This caches the most

recently accessed directory entries together with the initial sectors of files. Caching the first part of a file improves the speed of the file server when it is handling client requests to read file UIDs. Each cache segment is again 512 bytes and the file system is built allowing a maximum cache size of 64 segments. Once more, we use an LRU replacement scheme. However, this is a *write through* type cache - it always reflects the contents of the drive.

User-data storage on NAND Flash

NAND Flash characteristics In Section 9.4.2.1, I described the characteristics of NOR Flash. NAND Flash is similar. It is nonvolatile memory that can be erased and rewritten.

Write operations change bits from one to zero, but an entire block must be erased to change a bit from a zero to a one. Again, there is a limit to the number of times a block of NAND Flash can be erased and rewritten. However, there are a number of differences between the characteristics of NAND and NOR Flash:

- Unlike NOR Flash, NAND Flash devices are not byte-addressable - they can only be read and written in page-sized units. (These pages are 512 bytes or larger, but are always smaller than the erase block size)
- The geometry and timing characteristics of NAND and NOR Flash are different. NAND devices tend to have smaller blocks than NOR devices. Program and erase operations are faster on NAND Flash
- NAND Flash has a low limit on the possible number of partial program cycles to the same page. (After being erased, all bits are in the *one* state. Writing to a page moves some of the bits to a *zero* state. The remaining bits at *one* can still be changed to zero without an erase, using a subsequent write operation to the same page. This is called a partial program cycle.)

As I mentioned at the start of Section 9.4.2, we designed the LFFS specifically to enable user-data storage on NOR Flash. The differences in the NAND Flash characteristics that I have listed mean that LFFS is not a suitable file system for user-data storage on NAND Flash. The most fundamental issue is the low limit on the number of partial page program cycles that are possible on NAND Flash. As we saw, LFFS relies on being able to perform partial programs to update the completion status of each log entry.

The Flash translation layer (FTL) Instead, Symbian OS uses the FAT file system for user-data storage on NAND Flash. FAT is a file system better suited to the page read/write unit size of NAND.

However, because NAND pages have to be erased prior to a write, and because erase blocks contain multiple pages, we need an additional translation layer for NAND, to provide the sector read/write interface that FAT requires. This is the NAND Flash translation layer (FTL).

The translation layer also handles another characteristic of NAND Flash. When NAND devices are manufactured, they often contain a number of faulty blocks distributed throughout the device. Before the NAND device is shipped, its manufacturer writes information that identifies the bad blocks into a spare region of the NAND device. That is not all - as the FTL writes to good blocks on the device, there is a chance that these will fail to erase or program, and become bad. The likelihood of this occurring increases the more a block is erased. To handle these issues, the translation layer implements a Bad Block Manager (BBM), which interprets the bad block information from the manufacturer and updates it with information about any new bad blocks that it detects. The BBM also controls a reservoir of spare good blocks, and it uses these to replace bad ones encountered within the rest of the device.

The translation layer handles wear leveling, employing a scheme very similar to that used by LFFS. It also provides a system for ensuring the integrity of the data in situations of unexpected power removal, making sure that data already successfully committed to the device is not lost in such a situation - even if power removal occurs part-way through a write or erase operation. Indeed, the FTL is so robust that it can handle the situation in which power removal occurs while it is in the process of recovering from an earlier, unexpected, power removal.

The NAND FTL implementation may be split between a user-side file server extension (see Section 9.1.2) and a media driver. It can also be implemented wholly kernel-side in a media driver. The second scheme tends to result in a smaller number of executive calls between user-side code and the media driver which makes it slightly more efficient.

The first NAND FTL version released by Symbian employed a scheme split between a file server extension and a media driver. We had to split it this way on EKA1, and so we chose the same scheme for EKA2 to provide compatibility between the two versions of the kernel. We later implemented a version of the FTL for EKA2 entirely within a NAND media driver. Whichever scheme is used, the FTL and BBM software is generic to any NAND Flash device. However, the media driver contains a hardware interface layer, which is specific to the particular NAND device in use.

Figure 9.16 shows the components required to support user-data storage on NAND Flash memory.

File delete notification The FTL operates more efficiently as the amount of free space on the drive increases, since it can make use of the unallocated space in its sector re-mapping process. When a file is truncated or deleted on a FAT device, any clusters previously allocated to the file that become free are marked as available for reuse within the FAT. Normally, the contents of the clusters themselves are left unaltered until they are reallocated to another file. But in this case, the underlying FTL can't benefit

from the additional free space - it is not aware that the sectors associated with these clusters are now free. So, when the FAT file system is required to free up clusters - for example, in the call `CFatMountCB::DeleteL()` - it calls down to the next layer using the method:

```
CProxyDrive::DeleteNotify(TInt64 aPos, TInt aLength)
```

This provides notification that the area specified within the arguments is now free. If this layer is a file server extension implementing the FTL, then it can now make use of this information. If no extension is present, then the call can be passed down to the `TBusLocalDrive` interface, and on to the media driver where again an FTL can make use of the information.

Removable media systems

Those Symbian OS phones that support removable media devices must provide a hardware scheme for detecting disk insertion or removal, and it is the local media sub-system that interfaces with this. The file server needs to receive notification of these media change events so that it can handle the possible change of volume, and also so it can pass the information on to any of its clients that have registered for disk event notification.

I have already described (in Section 9.3.3.1) how the file server receives notification of a possible change of volume by registering a data member of the appropriate `TDrive` class with the local media sub-system. I also mentioned (in Section 9.3.4) that a client might register for notification of disk events, such as the insertion of a new volume. Instead of using this same `TDrive` mechanism to handle client notifiers, the file server uses a slightly different scheme. It creates an instance of the `CNotifyMediaChange` class for each removable media socket. This is an active object that requests notification of media change events, again via the local media sub-system. Each time a request is completed, the active object handles any pending client notifiers and then reissues a request on the local media sub-system for the next media change event.

Media change events are involved in a third notification scheme. For certain critical media access failures, the file server sometimes needs to display a dialogue box on the screen prompting the user to take some action to avoid disk corruption. This dialogue box is launched by an F32 critical notifier. It is used by the FAT file system in situations where a read or write failure has occurred that could result in corruption of the metadata on a volume. These situations include updates to the FAT tables and directory entries, and also running the ScanDrive utility. The dialogue box is only displayed on particular read/write errors. For removable media, these include errors caused by card power-down as a result of a media change event - in this case, the dialogue box prompts the user to replace the disk immediately to avoid disk corruption.

We implement the critical notifier in the class `CAsyncNotifier`. Each drive can own an instance of this class, and any file system can use it to provide a notifier service. That said, currently only the FAT file system uses it.

CAsyncNotifier uses the `RNotifier` user library class, which encapsulates a session with the extended notifier server - part of the Symbian OS UI system. When a file system needs to raise a user notification, it creates a session with the notifier server and issues a request on it, specifying the text for the dialogue box. Until the user has responded to the notification, and the request completes back to the file server, the drive thread on which the error occurred is suspended. During this time the file server is unable to process any other requests for that drive. Once the notification has completed, the original operation can be reattempted if needed - for example if the user replaced the disk and selected the *retry* dialogue button.

Since the EKA2 file server has a separate thread per drive, the processing of requests on unaffected drives can continue as normal while a notification is active. It is not so simple to support user notification on the EKA1 version of F32. Here, because it has only a single thread, the file server has to nest its active scheduler so that it can accept a limited set of other requests while the notification is being handled.

File server clients can enable or disable critical notifiers on a per session basis using the method `RFs::SetNotifyUser()`. The default state is for notifiers to be enabled.

The read-only file system (ROFS)

The read-only file system is part of the scheme used to support the storage of code (that is, ROM components) on non-XIP media such as NAND Flash.

The core OS image

As I mentioned in Section 9.4.3.3, NAND Flash devices are not byte-addressable, and they can only be read or written in page-sized units. As a result, they do not support code execute in place (XIP). This means that we need a RAM-shadowing scheme for code stored on NAND devices - the code must be read from the Flash into RAM from where it is then executed. Code on the Flash device must be stored in separate partitions from those used for data storage. Since the code partition is a read-only area, we

don't need a FAT format and we can use a simpler linear layout, which is similar to the layout we use for the ROM file system.

One approach we could take is to shadow the entire NAND code area. However, this would use up a lot of RAM! Instead, we normally shadow only a subset of the ROM components permanently, and load the remainder into RAM only when access to them is required.

If you are porting Symbian OS to a new phone platform, you can choose which OS components are permanently shadowed when you specify the contents of the ROM image. At a minimum, this needs to include the kernel, kernel extensions, media drivers, file server, file systems and ESTART. You can include additional components - obviously there is a trade-off between the speed gained by having a component permanently shadowed, against the amount of RAM this consumes. These permanently shadowed components are stored on the Flash device as a separate core OS image. At startup, a core loader program, which is one of the programs used to boot the OS, permanently shadows this entire image in RAM. The core loader does this shadowing before even the standard Symbian OS bootstrap has run.

The phone manufacturer can choose to have the entire core OS image compressed, to reduce the amount of Flash space this consumes. The core loader has to detect whether this is the case and decompress the image where necessary. However, the core OS image is loaded as a single entity and so the core loader does not need the ability to interpret the file system format within it.

The ROFS image

The remaining OS components, which are not included in the core OS image, are only loaded into RAM when they are required. The scheme we use loads entire executables and data files into RAM on demand, in much the same way as we load executables from a removable disk. This is not the same as a demand paging scheme, in which components are loaded at a finer granularity (that of a hardware page, usually 4 KB) and in which we would need a more proactive scheme to unload code from RAM that we deem to be no longer in active use.

The ROFS is the entity that reads these OS components from NAND Flash and interprets the linear format of this code area, which is known as the ROFS image.

The standard Symbian OS loader (which I discuss in Chapter 10, *The Loader*) copies entire executables and library files from the ROFS image to RAM, using the file server and the ROFS to read them. The ROFS image also contains data files, such as bitmap files. Clients of the file server issue requests to read sections of these data files, and again the file server uses ROFS to read them from NAND Flash memory. Individual executable files within the ROFS image may be compressed, and it is the job of the standard loader to detect this, and decompress them at load time.

ROFS uses the NAND media driver to read from the NAND device in the same way as the FAT file system does. The NAND region allocated for the ROFS code image may contain bad blocks and again ROFS uses the bad block manager in the media driver to interpret these.

To improve performance, ROFS caches all its metadata (its directory tree and file entries) in RAM. The ROFS file format places file UIDs in the metadata, as well as at the start of the file itself. This means that these UIDs are permanently cached, avoiding the need to make short, inefficient reads from the Flash to retrieve them.

ROFS also employs a small cache for the file data area.

Figure 9.16 shows all the components needed to support code and data storage on NAND Flash memory - including ROFS and the core loader.

Booting the OS from NAND Flash

Since it is not possible to execute code directly from NAND Flash memory, a phone using NAND Flash for code storage has to provide hardware assistance so that the processor is able to begin fetching code and can start to boot Symbian OS. Many platforms include a hardware boot loader. This is logic associated with the NAND device that includes a small RAM buffer, and it has the effect of making a small part of the start of the NAND device become XIP.

This XIP area is often very small, typically less than 1 KB, and may be too small to contain the core loader program. Instead a smaller miniboot program is normally the first code to execute, and its function is to locate the core loader program's image, copy it into RAM and then execute it.

Although, as we've seen, the NAND manufacturing process leaves a certain number of faulty blocks distributed throughout the device, usually the manufacturer of the device will guarantee that the first block is good. If the core loader can be contained within this block, then the miniboot program doesn't have to deal with bad blocks at all.

Next the core loader program executes. Its function is to locate the core OS image, copy it entirely into RAM and then find and execute the standard Symbian OS bootstrap. The core loader has to handle existing bad blocks within the core OS image, but not

the detection and handling of new bad blocks. It may also have to decompress the core OS image.

The miniboot program and the core loader do not have access to the normal Symbian OS NAND media driver, and so they have to duplicate some of its functionality. If you are creating a new phone platform, you must provide miniboot and core loader programs to suit your particular NAND hardware configuration. Symbian provides reference versions of these, which you can customize.

You must program the following images into the NAND Flash memory for a phone to be able to boot from NAND Flash:

- The miniboot program
- The core loader program
- The core OS image
- The primary ROFS image (and possibly images for secondary ROFS partitions)
- A partition table providing information on the location of all these images.

As well as these, phone manufactures will often also program a preformatted user data image into NAND Flash, so that this is available as soon as the device is first booted.

Phone manufactures must produce tools to program this data into the phone's NAND Flash memory. Symbian provides a reference tool which we use on our standard hardware reference platforms, and which you can refer to. This is the nandloader, NANDLOADER.EXE. When programming NAND Flash, we include this tool in a normal text shell ROM image, which we program onto our platform's NOR Flash memory. The platform boots from this image and runs the nandloader which allows the user to select and program the various component images into the NAND Flash. These images can be included in the nandloader ROM image or supplied separately on a removable media card. Finally the user can restart the platform, which this time boots from the images now programmed onto the NAND Flash.

The composite file system

The composite file system is unlike any of the file systems I've described so far. Although it implements the file system API that I introduced in Section 9.4.1, it doesn't directly use either the local media sub-system or a file server extension, as a standard file system would. Instead, it interfaces with a pair of child file systems. The composite file system is another part of the scheme we use to support code storage on NAND Flash.

Once the core loader has loaded the core OS into RAM, the standard ROM file system (which is embedded into the file server) provides access to its components in RAM. The ROFS provides access to the rest of the OS components, which it copies from NAND Flash on demand. Normally, the file server would mount two different file systems on two separate drives, but file server clients expect to find all the ROM components on a single drive - the Z: drive. So we use the composite file system to combine both file systems. This file system is a thin layer, which simply passes requests from the file server to either (or both of) the ROM file system or the ROFS. The composite file system uses the concept of a primary and secondary file system, where the primary file system is always accessed first. In this case, ROFS is the primary file system, since the majority of files are located there.

At system boot, the standard ROM file system is the only file system that is accessible until ESTART loads and mounts the other file systems. Before this, on a phone using NAND for code storage, only the core OS components are available on the Z: drive. ESTART then loads the composite file system, dismounts the standard ROM file system and mounts the composite file system on the Z: drive in its place. At this point the entire ROM image becomes available on the Z: drive.

Figure 9.16 shows the components required to support code and data storage on NAND Flash, including the composite file system - ECOMP.FSY.

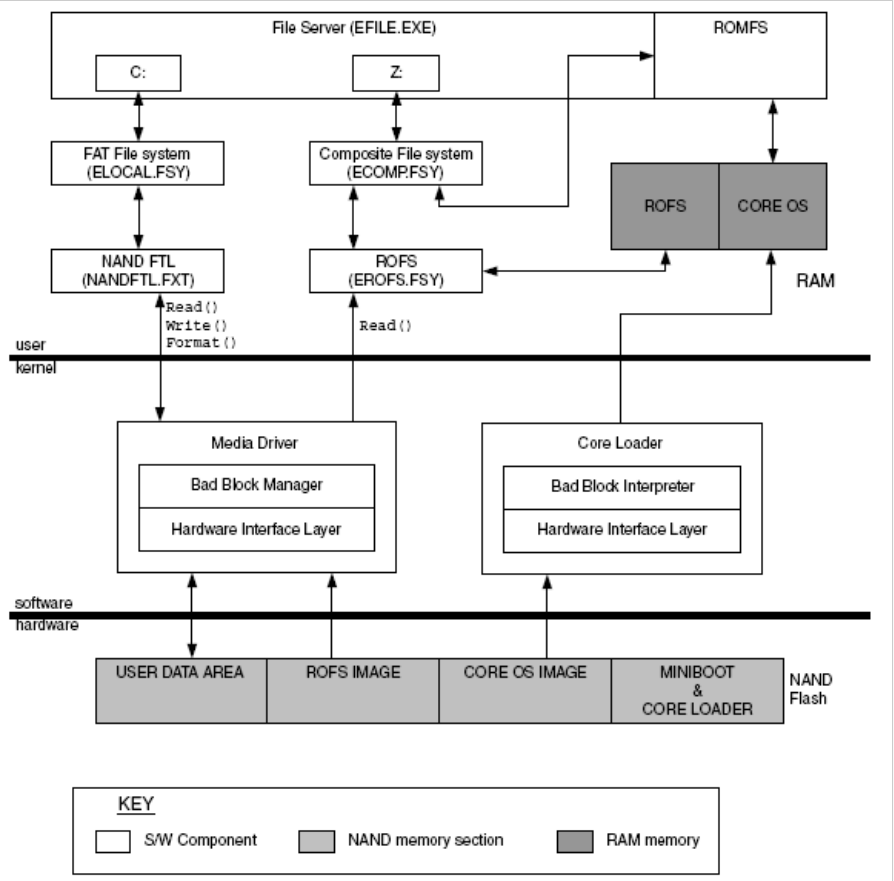

Figure 9.16 *The components involved in supporting code and data storage on NAND Flash*

Summary

In this chapter, I have described the F32 system architecture including a brief overview of the client API, followed by a more detailed description of the design of the file server. Next I described the file systems in general terms before looking in detail at the log Flash file system (LFFS), the FAT file system, the read-only file system (ROFS) and the composite file system. In Chapter 10, *The Loader*, I will describe the loader server, which is also part of the F32 component.