



In addition to an application's document file, the application architecture provides support for an application to open, read and modify a second file. By convention, the file has the same name as the application, but has a .ini extension; hence such files are known as .ini files.

The intention is that a .ini file should be used to store global settings and preferences that are independent of the document data that the application is processing. An application can then, in principle, open a different document without affecting the existing preference settings.

In practice, applications running on mobile phones tend to use only a single document and therefore have no real need to use .ini files. In recognition of this fact, the S60 UI disables the application architecture's support for .ini files, but it is a simple matter to restore it if necessary, by replacing **CAknApplication's OpenIniFileLC()** with a version that calls **CEikApplication's OpenIniFileLC()**, such as:

```
CDictionaryStore* CMyAppApplication::OpenIniFileLC(RFs& aFs) const
{
    return CEikApplication::OpenIniFileLC(aFs);
}
```

As well as opening the application's .ini file for exclusive read/write access ? and pushing it to the cleanup stack ? this function will create the .ini file if it does not exist, and will replace a corrupt file. The function is called by the application architecture, for example to record the last opened file, and can be called by your application code.

The .ini file is represented by an instance of **CDictionaryFileStore**, which derives from the **CDictionaryStore** base class. These classes are slightly misleadingly named since they have no connection with stream dictionaries. Also, since they do not derive from **CStreamStore**, they do not represent stream stores. Despite this, there are similarities of usage: the dictionary store contains streams associated with UIDs, and you read and write the streams by means of **RDictionaryReadStream** and **RDictionaryWriteStream** classes, in the same way as you would with a stream store.

However, a significant difference is that a dictionary store never contains more than a simple list of streams, unlike the complex network that is possible in a stream store. Furthermore, you could use UIDs to access the streams directly, rather than via a stream dictionary.

Header:

```
#include <s32stor.h>
```

Link against:

```
LIBRARY estor.lib
```

```
const TUid KDemoStreamUid = {0x101ffac5};
```

```
//Externalization = write into .ini File
void WriteToIniFileL(RFs& aFs)
```

```
{
    CDictionaryStore* iniFile = Application()->OpenIniFileLC(aFs);
    RDictionaryWriteStream stream;
    stream.AssignLC( *iniFile, KDemoStreamUid ); // direct access by UID
    TUint16 i = 0x3456;
    stream << i;
}
```

Using .ini_Files

```
stream.CommitL();
iniFile->CommitL();
CleanupStack::PopAndDestroy(2); // stream and iniFile
}

//Internalization = read from .ini File
void ReadIniFileL(RFs& aFs)
{
    CDictionaryStore* iniFile = Application()->OpenIniFileLC(aFs);
    RDictionaryReadStream readStream;
    readStream.OpenLC(*iniFile, KDemoStreamUid ); // direct access by UID
    TInt16 i;
    readStream >> i;
    CleanupStack::PopAndDestroy(2); // readStream and iniFile
}
```

Other Related Links

[Saving settings with Dictionary store](#)