Flash friendliness on the Series 3

Written by:	Martin Stamp
Revision:	V1.0
Date:	10 October, 2022

Background

Being 'flash friendly' under EPOC (the operating system on the Series 3, Series 3a and HC) is not always as simple as it seems. Some of the ramifications of the Flash Memory Filing System (FMFS) are not easily predicted.

This document explains the background and the steps that a developer should take to achieve the twin aims of:

- Compactness,
- Access speed.

Flash Solid State Disk(SSD)

The Series 3 uses an MS-DOS filing system. A standard MS-DOS filing system, which is used for RAM SSDs, has an area called the FAT (File Allocation Table). This fixed size area keeps a list of all files, directories and the relationship between them. This approach will not work on a write once media such as Flash.

To get round this problem the Flash Memory Filing System was developed by Psion in conjunction with Microsoft. The data is written exactly in the order it is saved. Hidden blocks are used to keep track of the files and directories. When part of a file is deleted or updated those blocks are marked as deleted and new blocks are added.

This has three implications:

- When reading the file it will always have to traverse over deleted blocks,
- The space is not recovered
- Every time the attributes of a file, (eg read only or the date time stamp) changes extra data must be written to the SSD.

When a call is made to determine the size of a file, the size that DOS would report is given. No allowance is given for the hidden header blocks that are also necessary (just as in a FAT based system no allowance is made for the FAT table itself).

Worst cases

It is interesting to consider the worst case. This is when two (or more) files are intermingled (i.e. a record from one file is followed by a record by the other) and the file closed or flushed after each record is saved. Each time the file changes a hidden block is written pointing to the

next valid data in the file. Not only does this waste space on the pack it also dramatically slows down access.

To give an idea of the problem:

Each file has 100 records with 50 characters per record	File A mixed with File B	Copied to a second Flash device	On RAM SSD
Size	14000	10000	10000
Search time - not found	700ms	10ms	10ms

To avoid these overheads:

- Use the built-in flash friendly DBF (database file) if possible; it has a particularly low overhead when deleting and changing records,
- When data is modified in a file with a record structure just rewrite the data that is modified not the whole file,
- Write data in a particular file as contiguously as possible,
- Encourage users to copy files if access is getting slow,
- Don't flush or close files unnecessarily - each time this is done a hidden block is written with the date time stamp,
- Do not create and delete files unnecessarily,
- Use the single byte write facility (see below).

Single byte write

Flash memory can selectively be changed from a set state (1) to a clear state (0) - except when being formatted when all bits are returned to the set state. So if a byte is, say, $37 it can be modified to be $31. It could not be modified to $34.

The PLIB call p_write of **a single byte** will, if bits are only to be changed from set to clear, modify the byte at the current file position in situ rather than by adding an extension record. This feature should be used with care. If the byte can not be modified in the requested way, because a bit is requested to be set when it is currently clear, the call will result in an extension record. This single byte write is very useful for small changes to data after it has been written. It could be used, for example, for marking a record in a accounts package as reconciled.

There are two important uses of this feature for applications not using the built-in DBF. Firstly when a record is written it is done in two stages:

- The size word, the record data with the illegal record type of $FF is written,
- The record type is over written with the correct type (say $F1), with a single byte write.

The reason for this becomes clear if you consider a write failure during the first stage.

Some space on the SSD is wasted but the data is marked as invalid. The second stage will normally succeed or to do nothing. In rare cases one bit might be cleared and another bit fails. So for the greatest level of security use one bit to indicate validity and blow that bit last.

Secondly when a record is deleted the type byte for that record is written to a special deleted value (say $7x, or $00). If the single byte write was not used the entire block, which might contain much more than just that record would be rewritten as a new extension block. This not only fills up the SSD but also slows access down.

The built-in DBF uses both these techniques and is to be recommended for most situations.

Files not closed down

For the date/time stamp to be added the file should be closed before the SSD is removed. Users should be encouraged to exit programs properly not to use 'User Abandoned' to exit.

If this is not done it can lead to the unexpected error on opening a file "Can not write to pack" (if the battery has insufficient power) as it tries to fix up the file structure!