SIR CMPWN TECHNOLOGIES

KnightOS Filesystem Specification

Version 2.1

Drew DeVault

3/7/2011

This document describes the structure of the KnightOS Filesystem and the means through which programs may access it.

Contents

1: Structure

The structure of the Knight Filesystem is based on ROM pages. The entire filesystem is stored in ROM, and each portion of the filesystem is stored on a single page. This decision is based on the fact that ROM is manipulated one page at a time from the calculator. However, it presents a challenge with Garbage Collection, which is described in section 2: Garbage Collection. The filesystem consists of two main areas – the FAT (File Allocation Table), which describes how each file and directory is stored, and the files themselves, where the actual contents of each file is kept. The latter is referred to as the File Storage Table (FST).

Within a single page, the table itself grows forward in memory. However, if the end of a page is reached, the table continues on the next page down. For example, if the end of page 01 is reached, the FAT continues on page 00. (Note that page 00 is a reserved page, and such an example is not possible).

All values spanning multiple bytes are stored in little endian format. Textual values are ASCII.

1.1: File Allocation Table

The FAT is used to store each file's metadata. This includes the name of the file, the directory it resides in, and more. It is also used to store the directory hierarchy. FAT pages are marked with 0xF0 as the first byte, and they continue from there. The format of each entry is as follows:

Offset	Size (bytes)	Item	Description
0x0000	1	Type	This one byte descriptor identifies what the entry is
0x0001	2 (x)	Length	The length of this entry's data, in bytes
0x0003	x	Data	The data that makes up this entry (unique to each type)

Each entry has its own entry type, which specifies what kind of entry it is. The types are:

ID	Item	Description
0x00	Garbage File	This represents a file to be picked up by the garbage collector (deleted)
0x01	Garbage Dir	This represents a directory to be picked up by the garbage collector
0xC0	Renamed	A renamed entry is one that should be removed, but leave the file intact
0xFC	File	A file entry describes a file in the FST
0xFE	Directory	A directory entry specifies that the entry is a directory
0xFF	Table End	0xFF signals the end of the FAT

All filesystem space is initially set to 0xFF, so that it may be modified at any time. Garbage entries are 0x00 so that any other entry may be turned into garbage.

1.1.1: File Entries

A file entry consists of the following information:

Offset	Size (bytes)	Item	Description
0x0000	3	File Size	The size, in bytes, of the file
0x0001	2	Directory ID	The directory ID this file resides in (root is 0x00)
0x0005	1	Flags	For possible future use
0x0006	1	Flash Page	The page this file begins on within the FST
0x0007	2	File Offset	The offset from the start of Flash Page where this file begins

0x0009	x	Name	The zero-delimited name of the file (up to 245 characters)

1.1.2: Directory Entries

A directory entry consists of the following information:

Offset	Size (bytes)	Item	Description
0x0000	2	Directory ID	The unique ID for this directory (root is 0x00)
0x0002	2	Parent ID	The ID of the parent directory (root is 0x00)
0x0004	1	Flags	For possible future use
0x0005	x	Name	The zero-delimited name of the file

1.1.3: Garbage Entries

Any entry that is marked with 0x00 is considered garbage. In flash, any value may be set to 0x00. This makes it easy to delete any entry. Garbage files should have their corresponding FST entries removed, and garbage directories should have every included file removed (the OS should set all included files to garbage at deletion-time).

1.1.4: Renamed Entries

When an entry needs to be renamed, the original entry is marked as renamed, and the entry is copied to the end of the FAT. See section 1.3.5: Renaming Entries for more information on this.

1.1.5: Table End Entries

A value of 0xFF denotes the end of a table. 0xFF can be changed to anything, which is why it is used.

1.2: File Storage Table

The file storage table does not have any sort of identifiers. It is merely the raw data of every file in the filesystem, appended to each other, including any garbage files made since the last garbage collection. This table begins at page 01 in KnightOS *(subject to change*) and continues forward until it meets the FAT. FST pages are marked with 0x10 as a starting byte. It is generally a good idea to keep a value (page and offset) representing the end of both the FAT and FST sections stored in RAM during OS execution, as well as the largest used directory ID (all of this information may be found at boot time, and during Garbage Collection).

1.3: Modifying the Filesystem

The following information applies only to manual manipulation of the filesystem, generally from within the Operating System.

1.3.1: Formatting the Filesystem

In order to use KFS, ROM must first be prepared. First, all sectors of ROM the filesystem will encompass must be cleared to 0xFF (the pages of the filesystem must span entire sectors). Then, the last page the filesystem will include should have the first byte set to 0xF0 (FAT identifier). The ROM is now prepared for KFS, and all routines should now work.

1.3.2: Creating Files

Making a new file is a relatively simple process. In RAM, collect the following information:

- File Name
- Residing Directory ID
- Flags

All memory within the filesystem past the end of the FAT is set to 0xFF. Because of this, it is all writeable. If at any point your routine encounters a byte outside the FAT or FST that is not 0xFE, the routine should fail. A garbage collect should be able to solve most of these problems in such cases.

Append a file entry to the end of the table, and open the file for writing with a file stream (see section 3. File Streams). Leave the length at 0xFFFFFF until you know for sure how large it will be. As the stream is written to, keep track of the file size in RAM, and when it is closed, write it to the file entry.

1.3.3: Modifying Files

Due to the nature of ROM, you cannot easily modify data. Instead, you should mark the original file entry as garbage, and open a writeable stream to a new entry (mirroring the original entry), which directs to a new entry at the end of the FST. The Garbage Collector will eventually clean up the original file. It is a good idea to leave the size at 0xFFFFFF and update it when the stream is closed.

1.3.4: Creating Directories

Creating a directory is relatively simple, in the same way a file is. Collect the following information:

- Directory Name
- Directory ID (the current highest directory ID plus one)
- Parent Directory ID (0x00 for root)

Once you have this, create a new entry in the FAT with this information, and increment the highest directory ID in RAM (if you choose to store this value in RAM).

1.3.5: Renaming Entries

Any entry may be easily renamed. It is as simple as marking the original entry as renamed, and creating a new entry with the same information, but a different name. The Garbage Collector will clean up the old entry.

2: Garbage Collection

The garbage collection (GC) process is performed whenever the FST meets the FAT, or vice versa. It may also be manually triggered. The OS must also set aside two Flash sectors (see section 2.2: Garbage Removal and Shifting for information on the possibility of using one flash sector and one RAM page), outside the filesystem, that are not used during Garbage Collection, and whose contents may be modified during Garbage Collection. These are called the swap sectors. Garbage collection must be performed from page 00.

2.1: Setup

In order to prepare for garbage collection, the following steps must be performed:

1. Unlock flash
2. Erase the swap sector (set to 0xFF)
3. Load the first FAT page to bank 1, and the first swap sector page in bank 2
4. Set aside working RAM in bank 3

2.2: Garbage Removal and Shifting

Garbage removal is the process of actually removing renamed entries, deleted files, and garbage entries. Shifting refers to the process of shifting the subsequent entries (both FAT and FST) to fill in the remaining space. Both processes occur simultaneously. First, you must clear the swap sector, and load the current FAT sector into it. Then, clear the current FAT sector, and load every page back (from the swap sector), except for the current working page. Repeat the process for the current FST sector. Then, you can begin to process the current FAT page. Simply begin a loop that moves from entry to entry, in order, using the swap page to observe the FAT entries. During this process, if the end of the FAT or FST page is ever reached, the next page should be swapped in, and the offset/page values updated (see section 2.2.1: Temporary RAM below).

Note: On models other than the TI-83+ BE (or any model/OS where an entire RAM page is easily available), it is possible not to use two swap sectors, but to use an extra page of RAM for temporary storage. If you choose to do so, adapt this specification as necessary. When you swap out the RAM page in bank 3 using this method, you should use registers to track the values stored in temporary RAM (described in section 2.2.1: Temporary RAM).

2.2.1: Temporary RAM

The following values need to be kept in RAM, in bank 3:

- The current FAT page
- The current FST page
- The current FAT offset (writing)
- The current FAT offset (reading)
- The current FST offset (writing)
- The current FST offset (reading)

Writing offsets are the new values, and the reading offsets are their place in the swap pages.

2.2.2: File Preservation (Shifting)

If an entry is not marked as garbage, it should be copied back to the FAT page at the current FAT offset. If it is a file entry, the corresponding FST entry should also be copied. To do so, first copy the entry. Then, swap in the current FST page, and copy it back from the second swap page. Swap the FAT page back, and the first swap page, and update the FAT/FST offset values (and page values if applicable).

2.2.3: Garbage Removal

When an entry is found that is marked as garbage, it is not copied. Instead, it is skipped in the FAT write offset, and the FAT read offset moves past it. The same applies to the FST write/read offsets. It is simply not preserved.

2.2.4: Handling Renamed Files

Renamed files are handled the same way as garbage entries, except the FST offset is not updated.

3. File Streams

In order to facilitate manipulation of the filesystem, it is suggested that you use file streams. When a file is open for writing, create a writable file stream, containing the address, page, and size of the file. When writing, the OS should provide a routine that write a byte (or several), and handles page switching and size modification. The operating system should also provide a routine to close streams, which will apply the size changes to the FAT entry (see 1.3.2: Creating Files and 1.3.3: Modifying Files for information). Readable streams will do the same, but should not be allowed to write through OS routines. With this method, only one writable stream may be open at a time, so you should encourage programmers coding against your OS to quickly use writable streams, and close them as fast as possible (especially in multitasking/multithreading OSes).

4. Notes

The KnightOS Filesystem provides an easy way to manage a filesystem tree through ROM, specifically on the TI-83+ family. This filesystem has been optimized to make it run quickly and avoid excessive wear on the Flash chip. Please send any comments or questions you have to sircmpwn@gmail.com, or post them in the KnightOS forum.

5. Implementation in KnightOS

KnightOS is the primary target for KFS. The following documents its implementation.

5.1: Necessary RAM Areas

KnightOS uses the following areas of RAM, as defined by KnightOS.inc, to manage files:

- FileStreamTable: The list of open file stream objects
- TempGCRAM: RAM used by KnightOS during the garbage collection process

5.2: System Routines

The following routines, as defined in KnightOS.inc, may be CALLed by a program to modify the filesystem.

CloseStream

Closes a readable or writeable stream

Inputs:

E: Stream ID

Outputs:

A: Error code

CreateFile

Creates a new file, and opens a writeable stream on it

Inputs:

HL: Pointer to full name of the file

Outputs:

A: Error code

B: Stream ID

OpenFileRead

Opens a new readable file stream

Inputs:

HL: Pointer to full name of file

Outputs:

A: Error code

E: Stream ID

ReadByteStream

Reads the next byte from a readable stream and advances it

Inputs:

E: Stream ID

Outputs:

A: Error code

C: Read byte

ReadWordStream

Reads the next two bytes from a readable stream and advances it

Inputs:

E: Stream ID

Outputs:

A: Error code

HL: Read word

ReadStream

Reads several bytes from a readable stream, writes it to a buffer, and advances the stream

Inputs:

E: Stream ID

BC: Length to read

Outputs:

A: Error code

HL: Pointer to buffer of data

Comments: Don't forget to release the buffer with FreeMem when you are done with it

SeekStream

Moves the pointer in a readable stream

Inputs:

A: Seek mode (0x00: To start, 0x01: To end, 0x02: To exact position)

E: Stream ID

Outputs:

A: Error code

OpenFileWrite

Opens a new writeable stream on an existing file

Inputs:

HL: Pointer to full name of file

Outputs:

A: Error code

E: Stream ID

Comments: When opening an existing file, you can only add to it. If you want to insert or remove bytes, create a new temporary file, and use it to modify the file.

WriteByteStream

Writes a byte to a writable stream and advances it

Inputs:

A: Byte to write

B: Stream ID

Outputs:

A: Error code

WriteWordStream

Writes two bytes to a writeable stream and advances it

Inputs:

E: Stream ID

HL: Word to write

Outputs:

A: Error code

WriteStream

Writes a buffer of bytes to a writeable stream and advances it

Inputs:

E: Stream ID

HL: Pointer to buffer

BC: Length of buffer

Outputs:

A: Error code