uFS Project

Micro-Filesystem

SD-card filesystem for embedded systems

27/12/06

Please excuse my lousy english, thanks ;)

Table of Contents

1.What is it, Why is it

UFS is a simple file system, especially designed for small microcontrollers like the famous 8-bit AVR from Atmel (http://www.atmel.com). It can be used with memories, that support blocks (further called 'pages') of 512 bytes. For example it can be used with NAND SD and MMC cards. The theoretical maximum size of the disk is 4gibibyte (4,294,967,296 byte). (SD cards only support up to 2gibibyte).

The maximum size of a file is nearly 2gibibyte. 'Nearly' because the uFS needs some space on memory to manage itself.

Remember, that there is no support for Windows or Linux to access this file system on a SD card for example. So, if you use it, be sure, that only devices, that support uFS, have access to this file system.

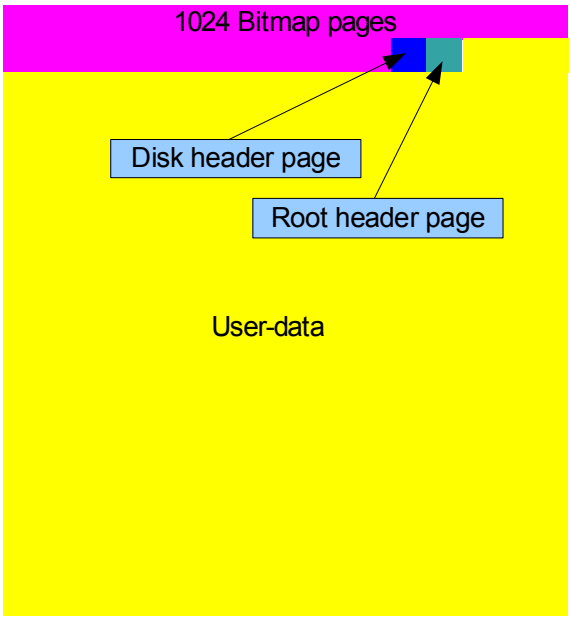
There are many reasons, why this file system is much better for “low-speed” and “low-memory” devices such an embedded system with a microcontroller.

- Internal References (Pointer) are designed as byte-pointer. Not as page-pointer. As the second variant would be less memory-wasting, the first variant will avoid multiple multiplications and divisions, that would waste CPU time.
- The microcontroller, that supports the uFS does not need to hold any tables or something like that in its memory like FAT does.
- File and dir-names are designed as zero-terminated strings. This will waste 1byte per file or dir-name, but whill make handling with them much easier than, for example FAT does.
- Each page (512 byte block) holds the adress of the next page. So there is no need of an extern table, like FAT has.
- A large bitmap is used, to determine which page is used and which is free-to-use. This should result in a fast page allocation and freeing.
- The size of each used variable is designed as a multiple of 8 bit. There are no variables that have a sice of 3 bit 5 bit, or something like that. This makes the compiler avoiding to use bitfields and should result in a higher speed of the whole file system.

2.Disk description

2.1.Disk layout

In this chapter, I will describe the layout, that each uFS-formatted disk must have.



The first 1024 pages (The first 1024 * 512byte = 512kibibyte) are reserved for bitmap pages. Those bitmap pages are holding information about which page on the disk is used, and which page is free to use. 1024 Pages equals to 1024 * 512byte * 8bit = 4194304 bits overall. Because of that each bit will indicate the status of one single page, these 1024 bitmap pages make it possible to indicate the page-status of 4194304 pages on disk. When you say that each page has a size of 512 byte, then the area, that can be controlled by this bitmap-page-bits has a size of 4194304 pages * 512 byte per page = 2gibibyte. And this is exactly the size, that an SD card has at it maximum. (Note, that if your SD card has NOT a size of 2gibibyte, the count of bitmap pages still will be 1024. The bits, that do not hold useable information, because the disk is too small, MUST be set to '1'. So the system will never allocate those (not existent) pages.). As already said, those bitmap-pages will only be used by the allocating and freeing system included in the uFS-core. No other part of a program will get access to those pages. The reason, that those bitmap pages are at the first 1024 pages on the disk is, that it makes the page-adressing for the page-allocating and freeing functions much easier.

After those 1024 bitmap pages lays one page, that exist only once on the whole disk: the disk header page. It has a size of 512 byte and holds a disk name and the disk size.

After that, there is only one page left, that has a fixed adress on the disk: The dir header of the root directory. For a description of this header page, look at the following chapters.

In sum, there are 1026 pages, that are pre-defined and MUST always be at the described adresses. The whole disk space, that comes after those 1026 pages can be of the type a)header page b)filedata page or c)dirdata page. It holds the user-information of what dirs and files are on disk and what the content of those files is. (See next chapters)

Because of the fact, that the first 1026 pages are a kind of 'reserved', the first 1026 bits of the bitmap pages are ALWAYS set to logical ONE (means that these pages are used and are NOT free-to-use).

3.Page description

3.1.Page types with pre defined positions

About this pages, we already talked about in Chapter 2.

a)Bitmap page

The Bitmap page only consist of 512 bytes raw bits, that indicate which page is used, and which is free to use. In C it would like this:

```
struct t_bitmappage
{
	uint8_t byte[512];
};
```

b)Disk header page

The disk header is a simple page. It holds some information about the disk. For example the disk name and the disk size. Just to be complete, here what it would look like in C:

```
struct t_diskheader
{
	uint8_t diskname[255];
	uint32_t disksize;
};
```

3.2.Page types that describe user data

All of the pages in the “user data” space have one thing in common, as the bitmap page and disk header page have NOT.: a page header. This page header is used to show the number of valid bytes, that are in this page and to show where the “page-chain” continues. We add to this information some empty bytes, so that the header will have a size of exactly 8 bytes.To put this in C:

```
struct
{
	uint32_t hdr_pNextpage;
	uint16_t hdr_nValidbytes;
	uint16_t hdr_reserved;
};
```

a)Page with file/dir header information

Now, to get in detail, we will first talk about a header page. This header page is ALWAYS the first page of a file or directory-chain. It holds some information about the data that follows this header page. This information are things like a name (file or directory name) a filesize or attributes, that a file or directory can have (for example 'hidden', 'protected', or something like that). For a complete description i will first put the C code here:

```
struct t_headpage
{
      uint8_t objname[255];
      uint8_t attrflags;
      uint32_t tCreate;
      uint32_t tLastaccess;
      uint32_t tLastchange;
      uint32_t pParent;
      uint32_t filesize;
      uint8_t reserved[228];
};
```

Now in detail:

'objname' is the name of the dir or file. (254 chars, zero-terminated)
'attrflags' has the following flags:

- ATTR_ISDIR (bit 0) (this header is a dir (0) or a file (1) header)
- ATTR_ISPROT (bit 1) (this file/dir is protected)
- ATTR_ISARCH (bit 2) (this file/dir is archived)
- ATTR_ISHIDDEN (bit 3) (this file/dir is hidden)
- ATTR_ISSYSTEM (bit 4) (this file/dir is a system file/dir)

'tCreate' is a timestamp, when this file was created.

'tLastaccess' is a timestamp, when this file was last accessed.

'tLastchange' is a timestamp, when this file was last changed.

'pParent' holds an adress to a directory header, where this file/dir is located in.

'filesize' holds the size, if this is a file.

b)Page with raw file data

A page with raw file data is followed by a headerpage or by another page with raw file data. It simply holds the raw content of a file, if the header page at the beginning of this chain says, that this is a file chain (attrflags & ATTR_ISDIR is not set). The C equivalent is very simple:

```
struct t_filepage
{
      uint8_t data[504];
};
```

c)Page with dir entry pointers

If the first page in a chain says, that this is a dir-chain (attrflags & ATTR_ISDIR is set), then the following pages are pages with dir entries. When you imagine, that the only information, that a directory must have, is what sub-dirs and files it has in. This is in case of the uFS made very simple. The 504bytes, that remain after the page header gives enough space to hold exactly 126 32-bit pointer. Each pointer can point to another sub-directory or file. Now, this directory is the parent directory of each sub-directory or file in this directory. If not all 126 pointers are used, the unused will be set to '0'. This indicates, that this is no valid directory-entry and should be ignored for example in a directory listing. The C equivalent looks similar simple to the page with raw file data:

```
struct t_dirpage
{
      uint32_t entry[126];
};
```

3.3.Overall page description

Now, if we think, that each page (except the first 1026 pages) is built of this system, we simply can create a structure, that is able to represent every case, that can occur in the uFS.

```
struct t_page
{
      struct
      {
            uint32_t hdr_pNextpage;
            uint16_t hdr_nValidbytes;
            uint16_t hdr_reserved;
      };

      union
      {
            struct t_headpage headpage;
            struct t_dirpage dirpage;
            struct t_filepage filepage;
      };
};
```

Now, t_page will always have a size of 512 byte and always have 8 bytes page header. The remaining 504 bytes are overlayed with an union, so that you can choose on your own, on which way you want to access those 504 bytes.

4.File / Dir-chain description

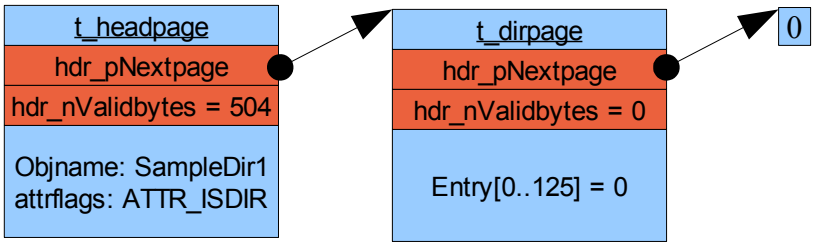
Now, where the layout of a page should be clear, we can talk about how files and directorys are built up.

Files and directorys are organized in “chains”. I call it “chains”, because after every page can come a next page, depending if there is still space for data needed.

Let us say, we want to make a directory.

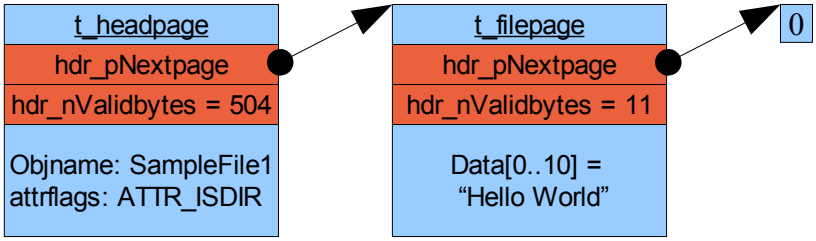
On first place in the Chain, there is a t_headpage. In this headpage, the 'attrflags' 'ATTR_ISDIR' is set to one. This will indicate, that now begins a directory. Further, there is the name of the directory, for example 'SampleDir1', in the headpage. Now, we will set the pNextpage pointer in this directory to a next page – a t_dirpage. We set all dir-entries to zero, because in this directory will no sub-directory or file. Also, in this dirpage, the pNextpage pointer is set to zero too, because there is no following page in this chain.

This was everything, that is needed to create a directory. Link the t_headpage of it in another dirpage (for example root) and this was it. I made a small drawing to make it a bit clearer.

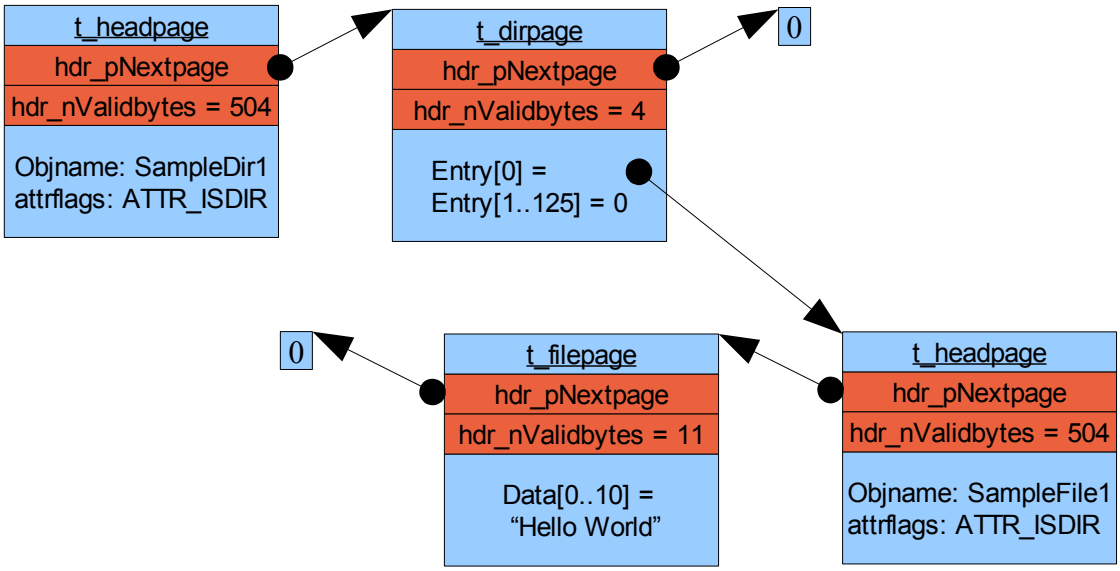


Now, let us make a simple File, containing the text “Hello World”:

In the first place, we will put a t_headpage again. But this time, we clear the ATTR_ISDIR attribute in 'attrflags' to define, that the following pages hold raw data. We put the name in the headpage again, too (for example 'SampleFile1'). We set the pNextpage pointer to a t_filepage and write the text “Hello World” ASCII based in the raw-data space. At last, we set the nValidbytes to 11, because the String “Hello World” has a size of 11 bytes. A second picture will demonstrate it.



Let us link 'SampleFile1' in 'SampleDir1':



5.Summary

As you can see, this file system is intentionally kept very simple. So everyone should be able to understand, how this file system is built up. I think, that this file system is not comparable to a file system like FAT or NTFS. There are many facts like Journaling that are not implemented in uFS, but are a crucial part of a today's file system. The audience this file system aims at is another. I think this file system could be usable for embedded systems, as said in Chapter 1. This file system could be used for simple file management for example, if you plan to design a logging-micro controller, that creates a new file or directory for each day it logs data from, for example an temperature sensor. This data can be read another time by the micro controller again and the data then be processed.