Chapter 2

EFS: EEPROM File System

Index

1. Introduction......3
2. Structure of EFS......4
2.1. EEPROM space distribution......4
2.2. Entries......4
2.2.1. Entry types......4
2.2.2. Structure of File entries......4
2.2.3. Structure of Data entries......5
3. Using EFS module......6
3.1. Basic principles......6
3.2. Creating and deleting files......6
3.3. Getting information of a file......7
3.4. Reading and writing......7
4. Bibliography......9

1. Introduction

One of the last things I had to do for openplayer was to store the settings into the EEPROM, since it is a non-volatile memory. However, at the time I was still deciding how to store the settings, I also was developing the skins system, where I introduced one feature that lets you to install skins located at your SD card. The question was how to store both data together and even leaving free space for user applications. Settings are easy: they have a fix length, so adding a few reserved bytes to this length should be enough. Skins also have a fix length, but if you can install more, the quantity is not fix. What is more, if a user saves his own values in the EEPROM, they could be overlapped someday by installing a new skin. Therefore, I decided to make a little file system, trying to make it very simple, fast and as much flexible as possible: the EEPROM File System (EFS). Its structure and functioning together with code examples will be explained in this chapter.

2. Structure of EFS

EFS is designed for those devices having at least 4096 bytes of EEPROM. A lower quantity of memory can also be used, but you may prefer to make use of that bytes that EFS takes out organizing EEPROM in your own way.

2.1. EEPROM space distribution

As you know, EEPROM is 4096 bytes long. EFS takes the first 3840 bytes and leaves the rest (256 bytes) for other purposes. These 3840 bytes, are divided into a total of 240 entries (16 bytes each one).

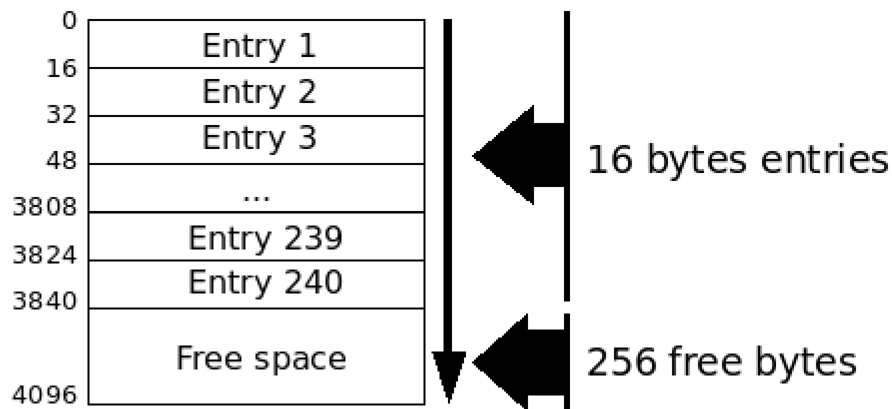

Illustration 2.1: EEPROM space distribution

2.2. Entries

Entries can be classified in different types, depending on its content. As you can see, they are quite small, mainly for one reason: the lack of space. You must write data in blocks of 16 bytes, so imagine if you just need three bytes of an entry to store your data: the rest would become unusable. I also thought that information needed to be stored in the EEPROM, would be in many cases of just a few bytes, which took me to choose that size.

2.2.1. Entry types

There are basically two types of entries: *File* and *Data* entries, and a third one which in fact it is not really a type, just an indicator for those entries which are free. The first byte of each entry is used to indicate its type. The following table shows the corresponding values for each entry type:

Entry type	Indicator value
File	0x01
Data	0x02
Free	0x00

Table 2.1: Entry types

2.2.2. Structure of File entries

File entries are used to store basic information of a file: fomat, size, name, where is data located. Let us see its structure and features:

Byte	Description
0	Entry type indicator (0x01 in this case)
1	File format
2	File size (in bytes, up to 255 bytes)
3..14	File name (11 ASCII characters + NULL character at the end)
15	First data entry

Table 2.2: Structure of File entries

- **File format**: Files can have up to 256 formats. 0x00-0x0f are reserved for system usage. As you can see it is not in a string like format (ex. “MP3”). This way we just only use 1 byte to indicate our file format.
- **File size**: Size (in bytes) of the file. Maximum size is 255 bytes.
- **File name**: The file name in ASCII. It must contain a NULL character at the end of the string. If your file has got no name, it just will contain a NULL character.
- **First data entry**: The first entry where data is located.

2.2.3. Structure of Data entries

Data entries are very simple, just 2 bytes are taken out by EFS: the *indicator* byte and the *next entry* byte. The rest contains data. As you can see, you can avoid the usage of file entries to store your data. You can directly write data entries that will let you to create your personal organization scheme, or just store simple data. What is more, you can use the *free space* to store the number of an specific entry where you stored information, so then you can easily access it. Note that *free space* has enough space to store the whole list of entries if you only want to use them as *Data* type.

Byte	Description
0	Entry type indicator (0x01 in this case)
1-14	Data
15	Next entry in the chain

Table 2.3: Structure of Data entries

- **Next entry in the chain**: Sometimes, the quantity of data that you need to store is too big to fit in a single entry. In these cases, more data entries will be used to put the remaining content. This field indicates which is the next entry.

3. Using EFS module

Openplayer includes a module to work with EFS without having to take care of all things explained in the last point. All applications that make use of EEPROM in openplayer should use this module, except they use the *EEPROM free space*.

3.1. Basic principles

The first thing is to include the header of EFS module so you will be able to access EFS functions inside of EFS module:

```
#include "sys/efs.h"
```

Source 3.1: EFS module header

After that, EFS must be initialized calling *efs_init()*, since this module uses a cache that must be filled before any operation. Note that openplayer already initializes it.

```
efs_init();
```

Source 3.2: Initialization of EFS

Another important thing is the *efs_file* structure template. It uses 4 bytes to store basic information of an entry, and must be given in many functions so this way they know in which file to work.

```
struct efs_file{
         uint8_t entry;
         uint8_t size;
         uint8_t format;
         uint8_t first_data_entry;
};
```

Source 3.3: The EFS file information structure template

3.2. Creating and deleting files

If you need to create a file, you just have to call *efs_create_file()* function, giving an *efs_file* structure (it will be filled with the new file information), the format of the new file and even its name. The following code shows an example on how to create a file named “GameConf” which has a 0xcf format:

```
struct efs_file gameconfig;

efs_create_file(&gameconfig, 0xcf, "GameConf");
```

Source 3.4: Creation of a file

After that, you can use the *gameconfig* structure to write to this file, to delete it, etc.
Let us suppose that you need to delete this file. It would be done by calling *efs_delete_file()* function giving the *gameconfig* structure:

```
efs_delete_file(&gameconfig);
```

Source 3.5: Elimination of a file

3.3. Getting information of a file

If you need to get information of a file already in EFS, (ex. you need to read a file, so you first need to fill the efs_file structure) you must call the *efs_get_file_info()* function. It is located giving the format of the file and its position among the files of that format. By now, getting information giving the name it is not implemented, basically because it would be very slow. Let us see an example:

Imagine that you need to get information of the first file with 0xab format:

```
struct efs_file gameconfig;

efs_get_file_info(1, 0xab, &gameconfig);
```

Source 3.6: Getting information of a file

And the list of files in EFS is the following:

File number	File number (regardind 0xab format)	Format	Size	Name
1	-	0xcc	16	opsettings
2	1	0xab	12	GameConf
3	2	0xab	24	GameConfbck
4	-	0x0f	123	opfile
5	3	0xab	200	GameConfdef

Table 3.1: List of EFS files (example)

As you can see, there are 3 files with the 0xab format, but as you asked for the first, the function will take the file number 2 (marked in yellow). If you had given the number 3, it would have filled the information of the fifth file.

Another interesting function is *efs_get_name()*. It lets you know the name of a file. You just have to give a valid *efs_file* structure and a buffer to store the file name.

```
char buffer[12];
struct efs_file gameconfig;

efs_get_name(&gameconfig, buffer);
```

Source 3.7: Getting file name

3.4. Reading and writing

If you need to read the contents of a file, you need to call *efs_read_file()* function, giving a valid *efs_file* structure, a pointer to a buffer to put read content, the offset where you want to start reading and how many bytes you want to read. For example, if you want to read the last 5 bytes of a file, you would call the function in this way:

```
uint8_t buffer[5];

efs_read_file(&gameconfig, buffer, gameconfig.size-5, 5);
```

Source 3.8: Reading a file

If you want to write content to a file, you can do it, even if the content you want to read is bigger than file size (function will automatically increase file size). For example, if you need to save *the score of a winner in your game* stored in a 2-byte variable at the end of your game configuration file:

```
uint16_t game_score;

efs_write_file(&gameconfig, (uint8_t*)&game_score, gameconfig.size, 2);
```

Source 3.9: Writting to a file

By now, this module just contains basic functions and could be improved. For example, it would be interesting to make a file shorter than its size (delete part of the current content), to make basic functions for those people who want use *Data* entries only, etc.

4. Bibliography

AVR-LIBC, “avr-libc Reference Manual 1.4.6”, http://savannah.nongnu.org/projects/avr-libc/

ATMEL, “Atmega640/1280/1281/2560/2561 Datasheet ”, http://www.atmel.com/avr

http://www.atmel.com/avr

Copyright (c) 2007 Gerard Marull Paretas.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.2
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
Free Documentation License".