ZCFS, implementation of log-structured filesystem for IoT devices

Alberto Crosta
Università degli Studi di Milano
Milano, Italy
alberto.crosta@studenti.unimi.it

Matteo Zoia
Università degli Studi di Milano
Milano, Italy
matteo.zoia1@studenti.unimi.it

***Abstract*—Persistence is one fundamental pillar in operating systems and a filesystem is its software implementation: methods and data structures that an operating system uses to keep track of files on a disk or partition. In this paper we propose a filesystem for the STM32F4 family boards. We have evaluated some kind of filesystems with the specific embedded use case in mind; we propose ZCFS a log-structured filesystem suitable for creating logs during the execution of applications in embedded contexts. ZCFS is implemented as middleware on top of the STM32 HAL layer.**

I. Introduction

Some common use case of embedded devices that led us to the choice of the log-structured model are based on the (realistic) assumption that the main activity that involve the filesystem on an embedded device is read configurations file or store continuous data for logging purpose.

For example: one of the main tasks of a quadcopter's flight controller is writing a log file for post-flight data analysis. In this case the main activity of the filesystem is focused on append operation (enhanced on a log-structured filesystem). It is possible to also bring log concepts on a different systems, let's think about a robotic arm or a CNC machine that need to monitor every operation done by the mechanical components, this operations can be carry out very well by a log-structured filesystem. When a fault or anomalies arise is possible to extract the actions logged by the filesystem for a later analysis purpose.

A. System contribution

In this work we try to approach the problem of building from scratch a filesystem in hopes of providing a stable way to store data without embedded device limitation. We discuss the choice of log-structured filesystem in terms of use case for embedded device, and present an implementation of such filesystem by discussing the data exchange protocol and the general architecture. We then tackle the optimization with buffers and timers, discussing the performance with a set of stress test.

II. System architecture

The design goals of ZCFS are to obtain a fully working filesystem on a low-performance CPU like STM32F4. With this project we achieve our goal but we want to point out that this is only a toy example and if you plan to use the ZCFS in a real application project, some improvement are needed to be done before its final release. For example in this PoC the maximum number of file is limited to a fixed number and the directory are not supported, so if you want to use it like a regular filesystem you should be add properly data structure to support directory trees [1].

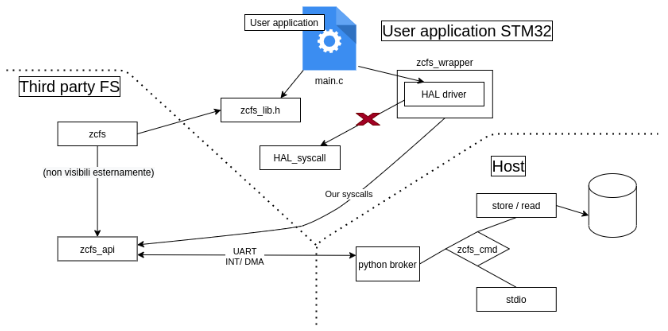

Fig. 1: system architecture

A. Software architecture

1) Disk structure: ZCFS is made of two core data strcture: the superblock (placed at the end of the disk) and the inode table contained in the superblock. Every chunk of a file stored in the disk is divided in data-chunk and data-inode-chunk that are stored in the disk and grows respectelly up and down (like stack and heap does). In this section, we will first show the layout of the whole disk, and then explain the management structure of files.

2) Disk layout: The disk layout of ZCFS is shown in Figure [2], divided into several parts:

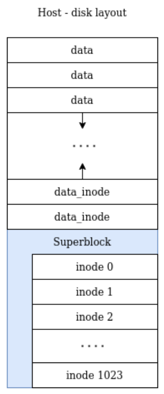

Fig. 2: disk layout

- **superblock**: the superblock data structure [code] keeps every information about the filesystem organization, in particular it keeps the address of the last data chunk written, the address of the last inode data chunk, the file descriptor number for the next request (incremental) and a list of inode (a.k.a. inode table).

- **inode-table**: the inode table [code] is a list of inodes; each inode is a structure that contains the file descriptor number, file name, time of last edit, size of the file (created by summing each data chunk associated within a file descriptor), a flag that tells if the file is open and buffered and a pointer to the last data inode.

- **inode-data-chunk**: the inode data chunk [code] are data structure that contains information about every chunk of data written to the disk. With this information ZCFS is able to reconstruct chunk by chunk an entire file spreaded across the filesystem due to the logic of the log-structured filesystem. The structure is very simple, there is a pointer to the data chunk, the size of the chunk and a pointer to the next dinode data chunk structure.

- **data chunk**: the data chunk are simply bytes append one above the other, there are no structure in it and because of this fact without the proper data structure dinode data chunk are just junk data.

3) Disk management: As explained in the previous [section], all the structures are linked each other. In this paragraph, it will be show how those are linked in memory and how they work.

dinode and data: In the figure [3], there are some example of how each inode (`dinode` in our implementation) is linked to its own data. Every time a chunk of data is written in memory, it must generate a new dinode (data inode) that points to the data just created. Every dinode of the same file is connected to the next one by a linked list: re-creating a file is fairly simple, it consist only in jumping from a dinode to the next one.

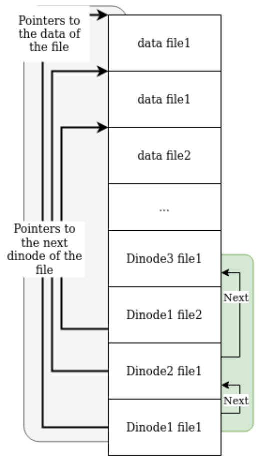

Fig. 3: dinode and data

superblock, inode and data: In the this figure [4], there are some example of how the superblock, inode and dinode are connected. The superblock has always a pointer to an inode, the inode value depends on the user program, if the user open a file the inode have the struct with the file information otherwise the inode value will be null. If the inode contains an opened file and the program has written some data in it, it points to the first dinode with the information about where the raw file data are stored. The inode has two links: *head* and *tail*. These links are useful to access directly to the first dinode, that represents the first data written in the file, and to the last dinode, that represent the end of the file. The first link (head) is useful to read the file and the second one is useful to append new data (new dinode) to the file.

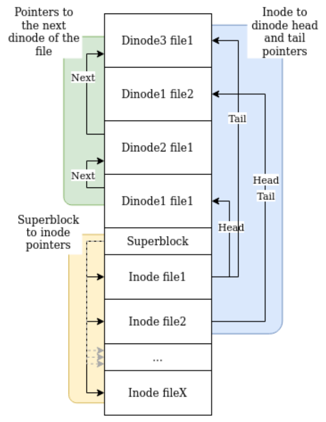

Fig. 4: inode and dinode

Full management: The schema below shows the full management and logical links between all disk parts [5]. Data and the inodes grow like stack and heap: the first grows from lower address to the upper and the second one grows from the bigger to lower.

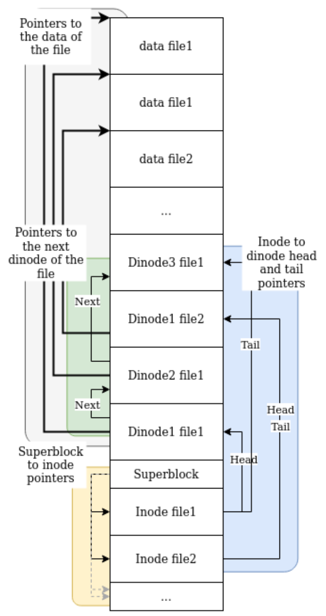

Fig. 5: inode, dinode and data

B. *HW architecture*

1) Board and components: We design the filesystem based on the family board STM32F4, in particular on STM32F407. The necessary hw to run the specific implementation of the filesystem is the board STM32F407 (the board used is the STM32F407 Discovery) and the FTDI (UART/USART) dongle to communicate with the host. The dongle is connected with the board on the PA2 and PA3 pins for the USART communication.

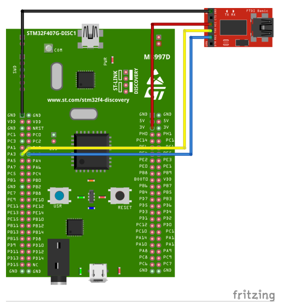

Fig. 6: board and components

III. Broker

The broker is the interface between the user program on the board and the host disk, in our PoC it is based on the USART protocol.

For this PoC the broker was written in Python3 but it could be implemented in every language, just follow the protocols to communicate between the board and the host.

A. *Communication protocol*

***fs_open**:* This functionality is used to create the structures of the file in the filesystem. More precisely it is used to initialize the inode with the file information such as: timestamp, filename and link to the data inode. The following image [7] shows how the function temporally interacts with the different components: the internal RAM and the Host.

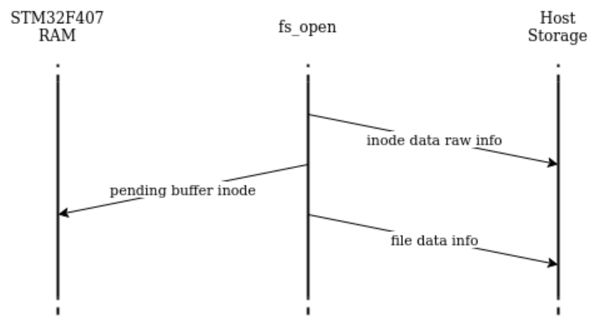

Fig. 7: fs_open protocol

***fs_read**:* This functionality is used to read raw data or structures from the filesystem. The following image [8] shows how the function temporally interacts with the Host which sends back the requested data to save it on RAM (after reading the packet with the ”read” functionality.

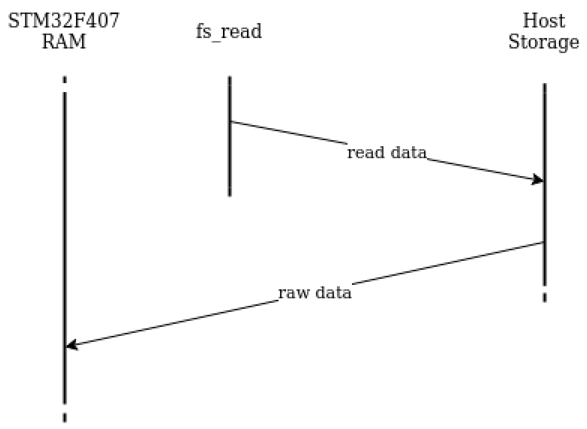

Fig. 8: fs_read protocol

***fs_write**:* This functionality [9] is used to write the data on the filesystem; in particular, this case represents the first data write in the file. Unlike the other schemas, this one has a not atomic component: the box “if buffer full or timer”. The fs_write function writes on the internal buffer but the raw data will be written on the host storage only if the buffer is full or the timer throws the interrupt; so if the buffer has been filled, the timer will throw the writing on disk anyway.

***fs_write - next dinode**:* This functionality [10] is used to write the data on the filesystem; in particular, this case represents the data write on a file that has already some data stored. The difference from the previous write is characterized by the fs_read that is called before the write.

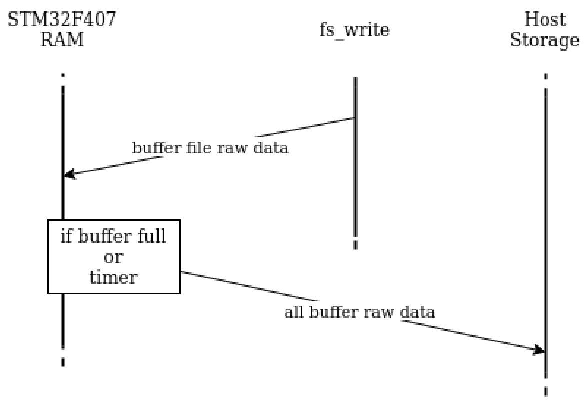

Fig. 9: fs_write protocol

The function call the fs_read to read the last file dinode to update the link of the current data which will be inserted in; this is necessary to maintain the linked list struct.

Also if the protocol is different, the function used in those cases are the same.

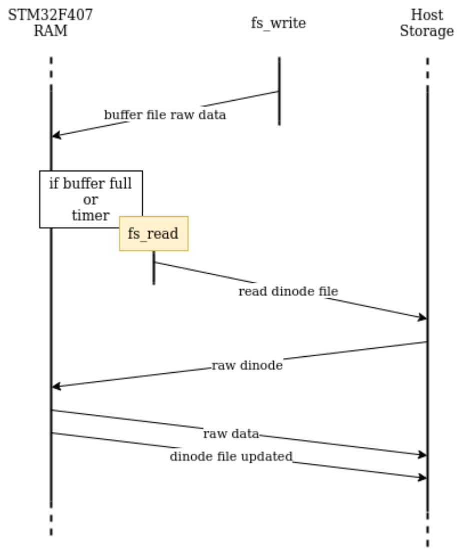

Fig. 10: fs_write next dinode protocol

***fs_close**:* To complete the explanation of the protocol functionalities, the following chart [11] represents the steps used to close the file on the filesystem. Before the closing of the file, the box "flush file buffer" represents the write operation of the last data buffered on the STM32 RAM of the specific file on disk.

IV. Buffer cache

The buffer cache is an important part of modern filesystems. The implementation of ZCFS proposed takes advantages in I/O operations by using a buffer. The buffer is not strictly needed

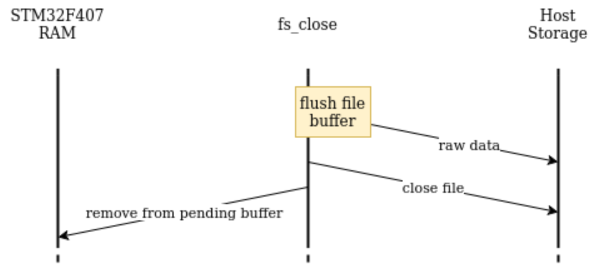

Fig. 11: fs_close protocol

and the filesystem can work without this improvement but in many cases caching a bunch of operations and delivering that once is better than committing every single operation when it happens. Before dive into logic of ZCFS buffer let's have an overview of the layout. We have decided to keep a buffer for every open file, these sub-buffers are called "*file buffers*". To manage all the "*file buffer*" correctly we keep a list in a data-structure called "*main buffer*" that have also some other control information.

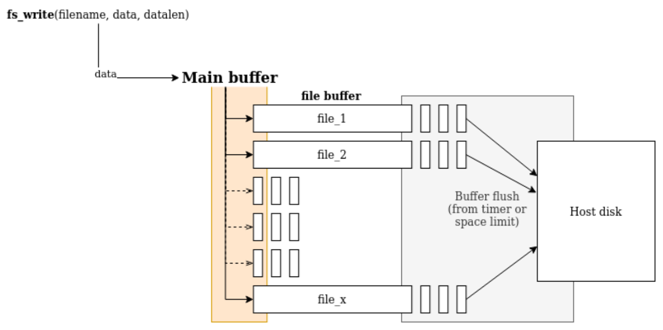
fs_write(filename, data, datalen)
data
Main buffer
file buffer
file_1
file_2
file_x
Buffer flush (from timer or space limit)
Host disk

Fig. 12: Buffer structure

When a write operation is issued through a call of `write` primitive, the buffer logic follow these steps:

- the filename is used as a key to search in the opened file list
- if the file is not open the write request is dropped (with an open operation the filesystem loads the structure to handle the write operation on a specific file, and obviously the file must be first open before a write)
- otherwise a check is performed to ensure that the data in the file buffer **plus** the data that need to be written according to this write operation don't overflow the buffer cache, otherwise the buffer is flushed
- finally the file buffer is filled with the new raw data

The buffer at some point needs to commit all the changes to the disk, this event can be raised by exceeding the buffer size (default set to 4KB) or by a time routine that expires some pre-defined delta-time. Another way that can be used (only for debugging purpose) is to directly call the routine `flush()`, this automatically scans the buffer and unloads its content to the disk.

A crucial point in buffer cache is to define a good expire function to flush the buffer, this is challenging cause the delta-time should be tuned correctly to get the maximum performance gain and the function itself must not alt a write operation that has just been triggered and is ongoing.

The use of the flush function in the buffer timeout routine is quite trivial cause some time it can be concur in a dead lock against a flush operation issued by the timer, to prevent this behavior we have used a lock to ensure that when the time routine start to unload the buffer no one other can call the same function.

V. Parameter settings

A. *buffer*

For the main buffer there is only one parameter to manage its size:

- **_BUFFER_SIZE** (see “fs_buffer.h”): This is the maximum value of characters/bytes that the buffer could contains before it commits data on the disk.

As explained before, the main buffer is only an array of sub-buffers where the real data are stored. Those sub-buffers are created with a space of **12** bytes of capacity, but this value is indicative; it depends on the type and the amount of data that an user want to store in the files.

- **_FBUFFER_INIT_SIZE** (see “fs_buffer.h”): this variable represents the initial size of the sub-buffer used to store the data of the new file created. Every times the main buffer (and so the sub-buffers) were flushed, the size of the sub-buffers are reset to this current value.

B. *memory/disk*

In this PoC, it is used a disk of 5MB and all other parameters were calculated to fit this size (ex. Superblock address, etc). We structured the code to easily modify those parameters, so if anyone needs more space (or less!) could set it as he wants. The following variables are the key of the disk size:

- **_VZCFS_DISK_SIZE**: variable that contains the size of the entire disk is set in the “fs_core.c” file.
- **_INODE_LIST_LIMIT**: it represents the maximum number of inode that the superblock could contains. Modifing this parameter could be useful to save space, if decremented, or to manage more files, if incremented. It is placed in “fs_util.h”

[WARNING] To make the code works, those parameters must be aligned with the ones set in the broker!

VI. Case study: CNC engaving machine

Let’s suppose to have a CNC machine with a robotic arm for engaving metal surfaces, as a case of study we want to support a realistic firmware for this purpose, an example should be like the following:

```c
int main(void){

  perform_components_test();
  go_to_home_position();
  get_input();

  while( !is_work_end() ){
    get_actual_position();
    calculate_new_position();
    set_motor_speed();
  }

  go_to_home_position();
}
```

in brief the `main` procedure starts with a check of all the components by performing some tests, after that it reads the input sketch to be drawn on the metal surface and continuously in a loop get the actual position, perform some calculation and move the motor for the next step. We talk about logging in a few words but just considering this scenario is clear that the function `get_input()` reads the sketch from the outside world. A tipical file for CNC machine is basically a text with a list of coordinate (x,y,z) and a time to reach that specific point in space; if the project is huge the size of the tex should be too large to be loaded in microcontroller memory, so best solution is to store it in a file. With ZCFS is possible to read only a part of a file at time, this is a good news for large CNC project because is possibile to store in memory only a chunk of that file and read all the sketch piece-by-pice as needed.

Another very important scenario is about logging, a CNC machine is a very expensive tool and some logging activity is very important to obviously have and history of the machine works and to have some data prediction about the worn of the motor. Suppose to have a motor that spin 20% more than it should be in a normal situation, in this case through the log activity, is possible to notice this over-power and replace the exhausted draw point without breaking the motor. The following piece of code show how to integrate the logging activity in a firmware with ZCFS.

```c
int main(void){

  perform_components_test();
  go_to_home_position();

  // get input
  uint32_t fd_sketch = zcfs_open("sketch");

  // logging file
  uint32_t fd_log = zcfs_open("log");

  for(int i = 0; i < sketch_size; i = i + 1000){
      // read 1k at time
    zcfs_read(fd_sketch, sketch_buffer, i, i +
        1000);

    while( !is_buffer_consumed(sketch_buffer) ){
      get_actual_position();
      calculate_new_position();
      set_motor_speed();

      // log work and sensors
      zcfs_write(fd_log, str_to_be_logged, strlen(
          str_to_be_logged));
  }

  go_to_home_position();

  zcfs_close(fd_sketch);
  zcfs_close(fd_log);
}
```

VII. Future works

A. Filesystem architecture

ZCFS is built for working on embedded systems, on these systems you had to deal with low memory size so we have decided to fix the maximum number of files to 1024 (that is realistic on embedded system). If we suppose to have 1024 files opened on the STM32 board the total size of memory occupied is about 30%, about 70% of free memory to the user program.

In our PoC we have decided to not support directory, in our use-cases the embedded systems do not necessarily have a tree of directory and sub-directory but are sufficient some files to write configuration parameters or to log the activity.

B. Broker setup

Another step that should be done in a fully working environment is an implementation of a startup communication between the filesystem and the broker to avoid errors, like synchronization between the filesystem and the host.

In a real world application the configuration parameters should be loaded automatically by the last configuration used in the broker. At the startup time ZCFS should communicate automatically with the broker to reload a configuration and restore all filesystem partition. Obviously this suggestion slow down the startup execution but they are necessarily to maintain persistence and to avoid human interaction.

C. Communication protocol

For this PoC we have decided to use the UART standard for communication with the external hard drive but in a real use case the UART should not be used due to its reliability and we suggest upgrading the communication to something like I2C.

For the implementation presented in this work we don't take in account a stafull connection, so if some messages are lost in the communication, there is no way to recover it. This situation is not so common because there is a simple cable between the board and the external PC, but in a real situation it is possible to have some interference and it's better that the protocol supports some acknowledge method to avoid inconsistent states.

Appendix

superblock

```c
typedef struct superblock{
  // last written address for data
  uint32_t ptr_data_address;

  // last written address for inode
  uint32_t ptr_dinode_address;

  // next fd to write
  uint32_t next_fd;

  ifile_t inode_list[_INODE_LIST_LIMIT];
}superblock_t;
```

inode

```c
typedef struct __attribute__((__packed__))
    inode_file{
  // file descriptor
  uint32_t fd;

  // 15 char + '\0'
  char name[FNAME_LENGTH];

  // time of the last edit
  uint32_t time;

  // size of the file
  uint32_t size;

  // is file open
  uint8_t is_open:1;

  // last dinode written for the specific file
  idfile_t* last_dinode;

  // next(first) inode written for specific file
  idfile_t* next_dinode;
} ifile_t;
```

data inode

```c
typedef struct inode_file_data{
  // ptr to the file data
  uint32_t data_ptr;

  // length of the pointed data
  uint32_t data_len;

  // next inode
  uint32_t next_dinode;
} idfile_t;
```

main buffer

```c
typedef struct __attribute__((__packed__)) buffer{
  // list of all files
  file_buffer_t* list[_INODE_LIST_LIMIT];

  // number of files
  uint32_t files;

  // size of the files
  uint32_t size;
} main_buffer_t;
```

buffer of the file

```c
typedef struct __attribute__((__packed__))
    buffer_file{
  // file identifier
  uint32_t fd;

  // buffer of the file
  char* file_buffer;

  // buffer filled
  uint32_t bfill;

  // size of the file buffer
  uint32_t size;
} file_buffer_t;
```