The RFID filesystem whitepaper

Achilleas Anagnostopoulos
(archie@istlab.dmst.aueb.gr)

Athens University of Economics and Business
Department of Management Science and Technology

1 Introduction

1.1 Why a new filesystem?

RFID tags are actually microscopic storage devices. While their storage capacity is quite limited (i.e only a couple of bytes), they don't require any kind of external power supply but are powered by the radiowaves emitted by readers. You can also find battery backed tags (active tags) which boast greater transmission ranges and more memory but they are quite expensive.

In any case, it only seems logical to represent RFID tags as files. By analogy to ordinary files, RFID tags are also *read,written* and have some kind of *modification mask*. We can better manage a large number of tags by organizing them in directories. In addition we can use them as input to the unix tool-chain. Finally we can easily exploit the plethora of file notification daemons available for responding to tag content. Some example uses of this filesystem will be presented in the following sections.

1.2 Why another kernel module?

There is a lot of heated debate on whether new features should be implemented as new *kernel modules* or as *user-land daemons.*

The clean and elegant way to write filesystem drivers is as modules exploiting the interfaces already provided by the linux kernel (Linux VFS). While several libraries for writing user-land filesystems do exist, they are often slow and complex.

Moreover, it is my belief that the actual hardware dictates the use of a kernel module. RFID readers come in many forms and employ different and often proprietary communication interfaces. While a reader with a serial interface could easily be interfaced to a user-land daemon, the same cannot be asserted for a reader in a PCI, ISA or PCMCIA form. In order to provide a unified interface to all reader hardware and to allow for future reader support (i.e 2nd generation readers) I chose to implement a new kernel module. The module's functionality is depicted in figure 1

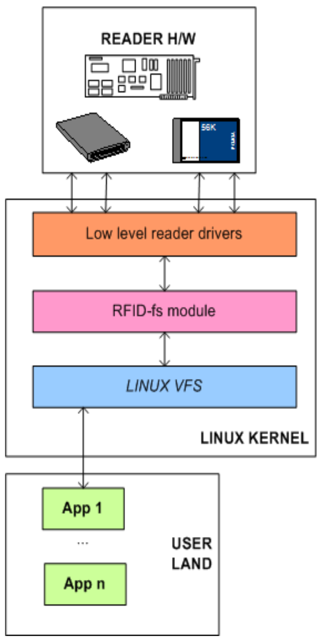


Figure 1: The idea behind RFID-fs

1.3 Tag Types

The current version of the filesystem supports the following types of RFID tags:

- **Read-only Tags (RO)**. These tags have a factory programmed unique 64bit number that cannot be reprogrammed.
- **Read/Write Tags (R/W)**. These tags may be reprogrammed as often as required or they can be locked to turn data into read-only. Their capacity is 80bits. Tag locking is not yet implemented.
- **Multipage Transponders (MPT)**. Similar to the R/W transponder but with increased capacity. MPT tags have 1360bits organized in 17 pages of 80bits each. Paged reads/writes are handled transparently by

the reader driver.

- **Selective Adressable Multipage Transponder (SAMPT)**. Similar to the MPT transponder but may be read or programmed in a group of transponders.

2 Filesystem specifications

2.1 File and Folder types

The filesystem supports 3 kinds of entities.

- **Tag Files**. They represent RFID tags. You can read them like ordinary files and depending on the underlying transponder H/W, also write them. Tag files may also be copied to other filesystems. In this case, a new file will be created whose content is a snapshot (think of a hard-link) of the current tag data.

- **Reader Directories**. These are automatically allocated by the filesystem when a new reader is autodetected. All tags in the reader's range appear inside this folder. These directories may not be renamed, moved or deleted. However it is possible to copy their contents to another filesystem.

- **User Directories**. These are directories that users may create for better organizing his tags. They may be freely copied, deleted, renamed or even symlinked.

2.2 Internal structures

2.2.1 File data

```
typedef struct rfid_file_data
{
  char *name;
  int   name_len;

char type;
char is_visible;

  struct inode *my_inode;
}_rfid_file_data;
```

Each file/folder has a distinctive name, stored in the *name* field. The file's type (tag/reader folder/user folder) is stored in the *type* field while a pointer to the allocated i-node for this file is stored in the *my_inode* field. The *is_visible* field is a boolean flag which is set when a tag is in the reading range of a specific reader. Once a tag moves out of range, it is marked as invisible and does not appear during the enumeration of folder contents.

The primary reason behind this important design decision is the issue of **kernel memory allocation**. Several readers support what is called "interrupt-driven notifications". This means that the actual hardware generates an interrupt when new tags move in/out of range. The reader driver installs an *interrupt handler* to process any hardware requests. This approach has obvious advantages over the periodic polling scheme (namely lower cpu resource usage). Unfortunately, one of the fundamental interrupt handler design rules states that: "*Interrupt handlers cannot invoke functions that might put the thread to sleep or the system might lock-up*". One of this functions is *kmalloc* which is normally used by kernel modules for allocating memory blocks. Fortunately, the Linux kernel API offers several ways to bypass this limitation (tasklets, work queues, etc) but their application is outside the scope of this paper. By employing the visibility scheme described above we minimize the number of memory allocations and deallocations and therefore increase the efficiency of the filesystem.

2.2.2 Tag data

```
typedef struct rfid_tag_data
{
  void *data;
  int  data_len;
  long data_hash;
  int  capacity;
  int  tag_type;

  char address[8];

  struct rfid_file_data *my_file;
}_rfid_tag_data;
```

The *rfid_tag_data* structure holds information about the actual RFID tag data. The data and its length are stored in the *data* and *data_len* fields. A hash value for the tag's data is stored in the *data_hash* field. The hash value is used to prevent tag collisions when multiple readers are used. The tag's capacity (in bits) depends on the tag type (*tag_type* field) and is stored in the *capacity* field. Typical capacity values are 64bits for RO tags, 80bits for R/W tags and 1360bits for multi-page tags. If the tag supports selective addressing (SAMPT type), the *address* field contains a special address code which may be used to selectively read or program the tag. Finally, the *my_file* field contains a pointer to the associated file data.

2.2.3 Inode data

```
typedef struct rfid_inode_data
{
  struct rfid_file_data my_file;

  struct rfid_tag_data  *my_tag;

  /* Child inodes */
  struct rfid_file_list *children;

}_rfid_inode_data;
```

This structure defines the private filesystem data which is stored in the *generic_ip* pointer of the kernel's i-node structure. Each i-node contains file data (stored in the *my_file* field) and for files of type *TYPE_TAG*, the relevant tag data in the appropriate *my_tag* structure. If the file is a directory, the *children* structure contains a linked list of all the files which are stored inside this directory.

2.2.4 Reader

```
typedef struct rfid_reader
{
  /*
   * Must be unique. This is the folder name under which read tags
   * are stored.
   */
   char *name;

   /* Driver specific data */
   void *private_data;

   /* Driver interface */
   char *init( struct rfid_reader *reader );
   char *shutdown( struct rfid_reader *reader );

   void *update_folder_tags( struct rfid_reader *reader,
                             struct rfid_file_list *list );
   void *refresh_tag( struct rfid_reader *reader,
                      struct rfid_tag_data *tag );
   void *sync_tag( struct rfid_reader *reader,
                   struct rfid_tag_data *tag );

}_rfid_reader;
```

Each reader structure contains a *private_data* field for storing reader specific data. In addition, each reader driver should fill in the supplied function pointers. If a driver does not implement a specific function, it should pass NULL to the function pointer.

- **init**. Once all supported readers are autodetected, RFID-fs will call each reader's init function. The role of this callback is to allow the reader driver to initialize the device hardware and set-up any internal structures.

- **shutdown**. RFID-fs will call each reader's shutdown function when the filesystem is unmounted. The role of this callback is to allow the reader driver to perform any required cleanup (free private data).

- **update_folder_tags**. RFID-fs will call this function each time it needs to update the status of all tags in a specific folder.

- **refresh_tag**. RFID-fs will call this function each time it needs to rescan (i.e read again) a specific tag. This function will only be invoked for SMPT tags.

- **sync_tag**. RFID-fs will call this function each time it needs to synchronize a tag's content to the data stored in memory. This function will only be invoked for SMPT tags as we need selective adressing for writing a specific tag.

2.3 Detecting RF-ID tags in range

2.3.1 When is the tag status updated?

My initial thought was to poll all readers at fixed intervals. However, after an fruitful discussion with Diomidis Spinellis, I figured out that there was no need to constantly poll the readers. We only need to update the tag status when a user application attempts to enumerate the tags in a specific folder (i.e using "ls" or system calls like opendir, readdir). RFID-fs will invoke the reader supplied *update_folder_tags* function to update the tag state. However, it's up to the reader driver's implementation to select the most appropriate way for performing the actual update (interrupt handlers or polling).

2.3.2 Handling Multiple Threads

The RFID-fs filesystem was designed with SMP architectures in mind. Even if the host system does not support "true" SMP (like Intel's hyper-threading technology) we should take all necessary steps to ensure that only one thread may access a reader at any given time. RFID-fs handles this by using *spinlocks* to protect all internal structures. Therefore, if two threads simultaneously try to enumerate the same folder, one will block (or sleep) while the other accesses each reader.

2.4 Tag Collision Handling

Since many readers may be connected to the same computer, it is possible that the same tag is successfully read by two or more readers. This produces a collision and makes it difficult to update the tag's content. In order to avoid tag collisions we maintain a global linked list containing all read tags. When a tag is first read, a hashing function is used to calculate a hash value from the tag's data. When a reader successfully scans a new tag, it calculates a hash value and compares it with all the already scanned tags. If another tag with the same hash value exists, the reader driver will assume that this is a collision and ignore this tag. By employing this collision handling mechanism, we ensure that each tag is managed by only one reader.

3 Putting It All Together

3.1 Adding rf-id fs to the 2.6 kernel tree

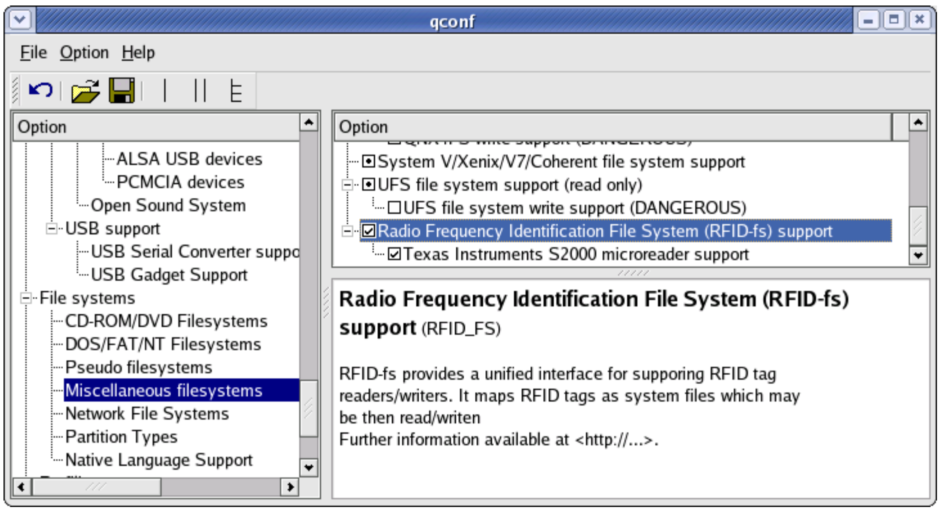
3.1.1 Using the kernel build system

The RFID-fs module has been fully integrated into the kernel build system. By applying some minor patches (available in the rfid-fs source distribution) you can configure and compile RFID-fs as any other aspect of the linux kernel (figure 2).

3.1.2 Compiling

To compile RFID-fs you just have to follow the normal kernel build rules:

```
make bzImage && make modules
make modules install
```



```
ALSA USB devices
PCMCIA devices
Open Sound System
USB support
USB Serial Converter suppo
USB Gadget Support
File systems
CD-ROM/DVD Filesystems
DOS/FAT/NT Filesystems
Pseudo filesystems
Miscellaneous filesystems
Network File Systems
Partition Types
Native Language Support
```

```
⊡ System V/Xenix/V7/Coherent file system support
⊡ UFS file system support (read only)
☐ UFS file system write support (DANGEROUS)
☑ Radio Frequency Identification File System (RFID-fs) support
☑ Texas Instruments S2000 microreader support
```

Radio Frequency Identification File System (RFID-fs) support (RFID_FS)

RFID-fs provides a unified interface for supporing RFID tag readers/writers. It maps RFID tags as system files which may be then read/writen
Further information available at <http://...>.

Figure 2: Configuring the RFID-fs module using the kernel build system.

3.1.3 Testing the module

If you compiled RFID-fs as a loadable module you can load it by typing (as root):

```
/sbin/insmod rfid-fs  OR  /sbin/modprobe rfid-fs
```

You can mount the RFID filesystem to a folder (say rfid_pool) by typing:

```
mount -t rfidfs none /mnt/rfid_pool
```

3.2 Example applications of RFID-fs

3.2.1 Digital Canvas / Personalized MP3 playlists

Digital canvas is a very interesting concept. A liquid crystal or gas plasma display, usually mounted on the living room wall, is directly connected to a computer. High quality multimedia content is stored and categorized in the host computer. Each user of the system carries an RFID card which contain a unique ID number. When the user enters the system's range, the computer randomly selects content based on the user's profile and displays it on screen. We could easily expand the idea to cover all kinds of media like for instance, mp3 files and music visualizations. Of course, this is just the tip of the iceberg. The system could also superimpose important information such as received emails or system alerts onto the displayable content.

3.2.2 Smart fridge

It is expected that by 2008 all retail products will have embedded RFID tags. A smart fridge could use RFID-fs to monitor its contents and alert the user when some products expire or automatically re-order items over the Internet.

3.3 Further Suggestions

- *Maintaining state information.* The current implementation of RFID-fs should be considered a RAM based filesystem. When the filesystem is unmounted, all user changes (user folders, symlinks etc) are lost. However, it is possible to generate a snapshot of the filesystem and dump it to a file residing in another filesystem. This snapshot could then be reloaded when the filesystem is remounted thus allowing for persistency.