Trusted Boot Module

Technical Specification

whitebox

Contents

1 Introduction

This document covers the specification, the design, the motivation and the implementation of the *Trusted Boot Module* or TBM. The TBM is an additional board that consists of a *microcontroller unit* or MCU to manage the boot procedure of the host device in a secure fashion by managing keys, logs and other files related to trusted boot management. Furthermore, the host device will be restricted to only boot from one read-only storage device that will contain a trusted image. Once this image has been booted, the hosted device is in a *Read-Only Trusted Stage* or ROTS from where it will be able to execute a minimal software stack to enumerate the images to boot, to verify these images and to select what image to boot. Once the image has been booted, the host device will enter an untrusted stage and the TBM will only allow for restricted access. This implementation allows the host device to only boot images that are trusted and prevents attackers from tampering with the host device or the TBM to boot untrusted images as long as they don't have physical access and as long as there are no vulnerabilities.

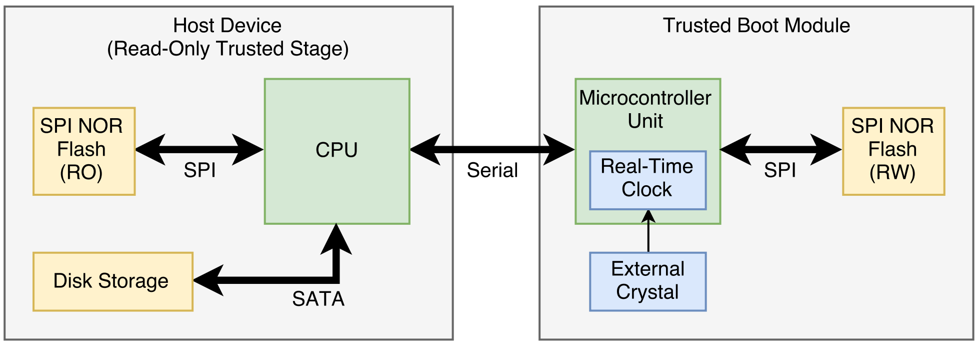

Figure 1: a high-level overview of the interaction between the host device and the Trusted Boot Module

Figure 1 shows a high-level overview of the design. Once the device receives power the *Trusted Boot Module* will boot and at some point the TBM will power on the host device. The host device will then read the trusted image from the SPI NOR flash. Because the device has been configured to be restricted to boot from the SPI NOR flash and because the SPI NOR flash has been configured to be read-only, the host devices enter a *Read-Only Trusted Stage* or ROTS. The image that has been booted is designed to be minimal and only contains the software necessary to perform the boot procedure. Furthermore, the image does not contain a network stack to reduce the amount of possible vulnerabilities and thus to minimise the attack vector. Once the trusted image has been booted, the host device will enumerate the images to boot and co-operate with the TBM to verify images and to select the image to boot. This co-operation happens by means of serial communication with the TBM, where the TBM will grant access to the key storage to the ROTS. Once an image has been selected to boot, the ROTS will inform the TBM that it will boot this image and enter the untrusted stage. From there on the TBM will only allow for restricted access.

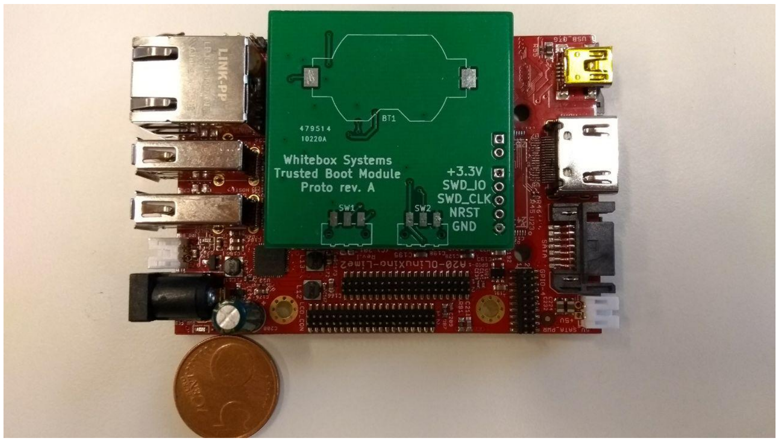

Figure 2: the Trusted Boot Module

1.1 Trust Model

There are different trust models that can be used depending on the use case. These mostly depend on whether the concept of certificate authorities (CAs) is required or not. Furthermore, the key storage also plays an important role in deciding which of the trust models to use.

- "Home router.„
  - **Certificate Authority**: none
  - **Replacement**: by re-flashing read-only flash.
- "Routers in a company with a system administrator.„
  - **Certificate Authority**: none
  - **Replacement**: by re-flashing read-only flash or by signed statements.
- "Routers with a single certificate authority.„
  - **Certificate Authority**: one
  - **Replacement of CA**: by re-flashing read-only flash.
  - **Replacement of keys**: by statements signed by the CA.
- "Routers with multiple certificate authorities.„
  - **Certificate Authority**: many
  - **Replacement of CA**: by threshold.
  - **Replacement of keys**: by statements signed by the CAs.
- "Routers with multiple certificate authorities and with an initial certificate authority pre-configured in ROTS„
  - **Certificate Authority**: many
  - **Replacement of CA**: by re-flashing read-only flash.
  - **Replacement of keys**: by statements signed by the CAs.
  - **Revoking keys**: not the same as removing.

1.2 Key Storage

- Read-only flash
- Trusted Boot Module
- Box storage (e.g. HDD)
- Signed statements by embedding keys in images.

2 Reproducible Builds

The untrusted images to be booted by the Whitebox can be signed by one or more authorities. During the boot procedure of the Whitebox the integrity of the image can be verified. However, a more fine-grained verification can be performed as well. An untrusted image also contains a table of contents listing all the packages that the image consists of together with their hashes allowing for the verification of individual packages within the image. These packages each consist of a manifest that describes where to locate the source code of the package, what tools are needed to build the packages as well as the instructions on how the build the packages, i.e. all the requirements to guarantee the reproducibility of the build.

3 Protocol

After the TBM has powered up, the TBM will power up the host device to ensure that the TBM is in full control of the host device. The host device will then boot the ROTS image from the SPI NOR flash. More specifically, the host device will load and boot the u-boot SPL, which will then load and boot u-boot. Once u-boot has booted up, u-boot will probe the SPI NOR flash device and read the kernel and the initramfs from the SPI NOR flash and execute the kernel. As the SPI NOR flash should be fully write-protected, all these images are read-only and cannot be tampered with. Hence, when the kernel has been booted up and once the initramfs has been mounted, the host device has reached the Read-Only Trusted State (ROTS). In this state, the host device is allowed to send any kind of request to the TBM to fetch the time, the certificates and any other information required to determine which image to boot.

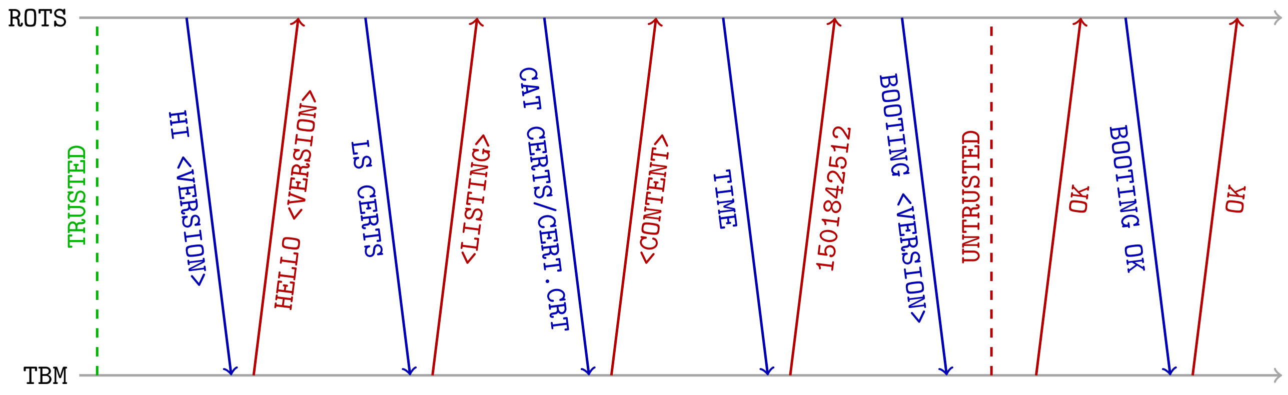

Figure 3: protocol diagram.

Figure 3 shows the interaction between the TBM and the ROTS after both device have booted up and entered the trusted state. In the Read-Only Trusted State, the TBM is passive and will not send any messages to the host device, while the host device is active and will send requests for the TBM to respond to.

Requests start with a command for the TBM to execute followed by zero or more arguments separated by spaces and terminated with a *Carriage Return* (`0x13`). If an argument consists of two or more words separated by spaces, then it must be enclosed in quotes, such that the TBM will treat it is as a single argument instead of multiple. In case input has to be provided to a command, the input must follow the command and must be terminated with an *End of Transmission* character (`0x04`). In all cases, the response of the TBM may include output terminated by an *End of Transmission* character. The *End of Transmission* character must always be present in the response, even is there is no output to signal that there is no output. An exit code or error message must follow the output and must also be terminated with an *End of Transmission* character.

The commands available to the ROTS are: `hi`, `ls`, `cat`, `time`, and `booting`. The `hi` command must be followed with the version of the ROTS on the host device and is used to initiate a handshake with the TBM. The TBM must respond with `hello` followed by the version of the TBM. The `ls` command is used to list the files in a directory and may be followed with a path on the filesystem. If no path has been provided, the root of the filesystem will be assumed. On success, the TBM must respond with a list of filenames. The `cat` command is used to read a file on the filesystem and must be followed with the path of the file to read. On success, the TBM must respond with the file contents. The `time` command is used to get the current time as a UNIX timestamp and requires no arguments. On success, the TBM must respond with the current time encoded as a UNIX timestamp. The `booting` command is used to indicate that the host device will boot an untrusted kernel, thus switch to an untrusted state and must be followed with exactly one argument indicating the version of the image that will be booted. On success, the TBM must restrict access to the commands to the commands that should be available in the untrusted state.

The commands available to the untrusted state are: `booting ok`. The `booting ok` command has no further arguments and is used to indicate that the kernel has been booted.

3.1 Clock Drift

To cope with clock drift the real-time clock has to be synchronised with an external clock. The most straightforward method to support this within the protocol is to allow an absolute time to be set. However, this should only be allowed from the read-only trusted stage. An alternative method is to allow the clock to be corrected by introducing a limited amount of additional ticks or to stop the clock for a limited amount of ticks. By having limited correction, the clock can still be synchronised while an attacker cannot change the clock by more than a negligible factor each year.

4 Features

4.1 Power Cuts

Power cuts when very short can cause certain parts of the hardware to be reset, while others still function properly. However, the probability of this happening is negligible.

4.2 Image Self-Test

The timers and real-time clock of the microcontroller unit can be used to wait for a response from the untrusted stage that it has managed to boot successfully or to reboot the host device if it fails to provide such a response in a given period of time. This would be useful as a self-test for images to check if they work appropriately or not. Furthermore, this would also allow the ability to rollback to previous versions that are known to work reliably.

4.3 Forced Reboot

As it is possible that there is a vulnerability in the untrusted image that has been booted by the TBM and that the image has been compromised by an attacker, it is important to make sure that the time window available to the attacker is limited. Because of this scenario, the TBM has been given full control over the power management of the host device, such that it can reboot the host device by force at any point in time and when the host device fails to co-operate with the TBM. Furthermore, to limit the time window the TBM will periodically reboot where the user can postpone this reboot a few times before the TBM will simply ignore such requests and perform a reboot by force. This design allows the host device to check for new updates and to boot new images where these possible vulnerabilities have been fixed.

4.4 Upgrading

One possible attack vector is that when the image that is running has been compromised in such a way that it prevents the system from running the update process. To prevent this the update process should scheduled to happen as early as possible within the image.

5 Real-Time Clock

To prevent downgrade attacks to known vulnerable firmware versions it is important to have a *Real-Time Clock* or RTC powered by a battery and a crystal to keep track of the time, even if the TBM is offline. To some extent it is also important to have a RTC or timer to force periodic resets to ensure that the Whitebox is in a known trusted state. However, the microcontroller unit already has an integrated RTC and timers for this.

5.1 Attack vector

Assuming that an attack has full control over the drift-correction protocol, the following attacks would be possible:

- **Downgrade attacks**: by drifting the clock too far into the past, it might be possible to reboot to a vulnerable kernel that would otherwise be blacklisted.
- **Denial of service attacks**: by drifting the clock too far into the future, it might be possible that there are no available kernels to boot as they have all been expired. Therefore, it might be possible that the Whitebox can no longer boot.
- **Reboot-prevention attacks**: by sufficiently slowing the clock down, it is possible to extend a compromised state. Even though it is not possible to extend this indefinitely, extending the time a system is compromised would allow for enough time to extract sensitive information or to be in the right time-window to capture privacy-sensitive information.
- **Reboot-encouraging attacks**: by speeding up the clock, the system will be forced to reboot a lot faster than usual, resulting in a denial of service and a frustrated end-user. This might cause the end-user to not further employ the security practices as encouraged by Whitebox, potentially compromising privacy indirectly.

6 SPI NOR Flash

SPI NOR flash is one of many storage devices that can be used for both the host device and the *Trusted Boot Module*. Even though it comes at much smaller capacities than other storage device, it is highly reliable and optimised for writing speeds. Furthermore, the write-protect pin combined with a pre-configured region allows that region to be read-only. This is required to guarantee that the *Read-Only Trusted Stage* can indeed be trusted.

Unlike other common storage devices, SPI NOR flash operates with a different granularity depending on the operation. For instance, reading and programming pages have a very fine granularity of 1 byte, whereas erasing pages must be done on at least pages of 4 kiB. Furthermore, there are faster erase operations that operate on 32 kiB and 64 kiB pages, as well as faster read operations with other limitations. In addition to the granularity, some of the operations such as programming pages also suffer from other limitations such as having a wrap-around limit of 256 bytes. When the offset exceeds 256 bytes while programming a page, the offset will wrap-around and continue writing at the beginning of that 256 byte page. To simplify the implementation of the SPI flash device driver only a select subset of the most common operations have been implemented with care.

6.1 Motivation

SPI NOR flash is a viable option for the *Trusted Boot Module*, because it is a cheap option that can easily be interfaced using SPI. Even though the capacity of SPI NOR flash seems to be on the low size (2 MiB - 16 MiB), this capacity easily meets the requirements of the *Trusted Boot Module* as it will only be used as a storage for keys, logs and a few other files. Furthermore, with an erase cycle of ±1,000,000 per page, SPI NOR flash only wears out slowly and doesn't ship with bad blocks unlike NAND flash. These properties make SPI NOR flash a reliable option for the *Trusted Boot Module*.

On the host device, SPI NOR flash is a viable option as the host device supports booting from SPI NOR flash, as it can be configured to be read-only and as it is the first option to boot from disabling the ability to boot from other devices as long as a valid image has been flashed to the SPI NOR flash.

6.2 Write-Protect

To ensure that the SPI NOR Flash is read-only the write-protected regions should be configured and the $\overline{\texttt{WP}}$ pin should be pulled down by connecting it to the `GND` pin. If the $\overline{\texttt{WP}}$ is not connected to `GND`, then the pin will be in a floating state and it will be possible to disable the write-protection.

The boot order of the host device is pre-configured in hardware and will boot from the SPI NOR flash device if one is connected to the `BOOT0` pins of the device. Furthermore, to properly boot the host device, an image containing *u-boot* and *Linux* will have to be flashed to the SPI NOR flash. This image will be the ROTS-image that also contains a minimal payload to the enumerate existing images, verify their integrity and boot them.

7 Virtual NOR Flash

To be able to quickly develop the current and future subsystems of the software stack that is to be run on the microcontroller unit of the MCU, the code has been designed in such a way that it can be run in a simulated environment. Part of this simulated environment involves simulating a virtual NOR flash device for testing the *Flash Translation Layer* and the file system.

7.1 Motivation

To simplify the development and testing process of other subsystems, it is useful to develop a simulated environment. One of the components of such an environment is the virtual NOR flash which emulates a SPI NOR flash device with similar behaviour as the actual SPI NOR flash device used on the *Trusted Boot Module*. Furthermore, by having a virtual NOR flash device, the wide variety of debugging and analysis tools such as *gdb* and *valgrind* can be used to easily verify the correctness of the other subsystems that are being developed for the *Trusted Boot Module*.

7.2 Design

The flash driver for this virtual flash device implements the same operations as the the SPI flash driver by mapping a file that acts as the flash storage into memory. Because writing on SPI NOR flash can only toggle ones to zeroes but not zeroes to ones, the write operations performs a bitwise AND to simulate the exact same behaviour. Similarly, the virtual flash device also implements an erase operation that simply sets the bits within a page to ones. In addition, a tool has been developed to easily create the file that acts as the flash storage.

8 Flash Translation Layer

Because SPI NOR Flash has some very different characteristics and properties compared to other common storage devices, the *Flash Translation Layer* (FTL) acts a thin layer on top of the SPI NOR Flash to provide an interface that makes the SPI NOR Flash appear more like a common storage device while optimally managing the data pages by maintaining a virtual-address-to-physical-address mapping and keeping these characteristics into account.

8.1 Motivation

While many common storage devices are block devices that operate on blocks of a certain fixed size, most commonly 512 bytes to 4 kiB, for all of their operations, SPI NOR Flash is fundamentally different. In the case of SPI NOR flash bits can only be toggled from one to zero during a write operation, but they cannot be toggled from zero to one. Therefore these bits have to be reset back to ones at some point. To achieve this, the SPI NOR Flash has been subdivided into pages of a certain fixed size, most commonly 4 kiB, that can be erased. However, the amount of times a page can be erased is limited to ±1,000,000 times before the pages start to wear. While it takes quite a lot of erasures before the pages start to wear, it makes sense to load-balance I/O over the pages to extend the lifetime of the flash device. Furthermore, by managing erasure implicitly we can achieve an interface that is more similar to those of common block devices where we can simply perform reads and writes on virtual blocks of a fixed size. In fact, this is similar to what *Solid State Drives* or SSDs do for NAND flash, which have similar characteristics.

8.2 Radix trees

At the very heart of the FTL is the radix or prefix tree that is used to be able to determine whether a virtual address has been mapped, and if it has been, to locate the physical page that corresponds with that virtual address. Each of the pages encode a virtual address as well as an array of pointers to other pages for each of the bits in the virtual address. When the virtual address of a certain page does not fully match the virtual address, the prefix of n bits that does match is used to select the pointer to the next subtree. This process is repeated until an exact match has been found or until there is no further subtree which is an indication that the virtual address has not been mapped.

8.3 Page groups

Because the erase operation only operates on a certain fixed size at minimum, the FTL subdivides the flash device into blocks that are at least large enough to be erased. These blocks are then further subdivided into page groups of 2^n pages, starting with $2^n - 1$ user pages followed by a footer page that consists of $2^n - 1$ page descriptors. Because the size of a logical page is fixed by the user, the amount of pages in a page group and thus the amount of page groups in a block is determined by the amount of page descriptors that fit within a single logical page. Each of the page descriptors correspond with a user page within the block and consist of a virtual address as well as an array of pointers to other subtrees. Once the page descriptor that corresponds with a certain virtual address has been found, the user page that corresponds with that virtual address can be easily found as the position of the page descriptor corresponds with the position of the user page within the page group.

8.4 Journal

When mounting a flash device with a FTL, a scan is performed on the first few blocks to find the initial block of the FTL. If no such block could be found, the FTL is reset to an initial state where it will start writing at the very first block of the device. Once the initial block has been located, a binary search is performed to locate the last written block on the device and finally a binary search is performed within that block to locate the last written page group. Once the last written page group has been found the magic value can be checked to verify the integrity of this footer and the pointer to the last written page descriptor is extracted from this footer to locate the root of the radix tree for the look up of virtual addresses. Once the root has been found, the head is set to the next free user page that is in a erased state or the next block.

When writing data to the FTL, a user page is allocated by incrementing the head pointer to the next user page, by writing the user data and finally by writing the page descriptor. In case of failure, the FTL may end up with a stall user page. However, because the user page will then be inaccessible the write operations appear to be fully atomic, i.e. the operation has either completed or the operation does not seem to have occurred at all. If the head points to the first user page of a block, the block will be fully erased before writing.

8.5 Garbage Collection

Because pages are always written at the head pointer, the head pointer will eventually meet the tail pointer of the FTL leaving no space for further writes. At that point, pages have to be freed from the block the tail pointer points at to be able to claim more space for writing. This is done by copying any pages present within the block at the tail to the head. Furthermore, to guarantee that the garbage collection process can always take place, the FTL always reserves one block of space by reporting a smaller capacity than the actual capacity of the FTL..

8.6 Trim

In addition to read and write operations, the FTL supports trimming, i.e. marking pages as being no longer in use. First the page to trim is located by performing a look up of the virtual address. Then the array of pointers is traversed until a subtree is found, this is the subtree with the shortest matching prefix. The reference to the page to be trimmed is then removed from this page descriptor. Finally, this new page descriptor is written to the head and the user data that was previously associated with this page descriptor is copied over.

8.7 API

- `uint32_t ftl_get_size(const struct ftl_map *map);`
  Returns the amount of bytes that are in use by data.
- `uint32_t ftl_get_capacity(const struct ftl_map *map);`
  Returns the total amount of bytes available for data.
- `size_t ftl_read(struct ftl_map *map, void *data, size_t len, uint32_t va);`
  Reads `len` bytes of data from the given virtual address `va` into the buffer at `data`. Returns the amount of bytes that have been read.
- `size_t ftl_write(struct ftl_map *map, uint32_t addr, const void *data, size_t len);`
  Writes `len` bytes of data from the buffer `data` to the given virtual address `va`. Returns the amount of bytes that have been written.
- `int ftl_trim(struct ftl_map *map, uint32_t va);`
  Trims the page at `va`. Returns 0 on success. -1 otherwise.

9 µ-Filesystem

µ-Filesystem or *µFS* is a filesystem that has been specifically designed for both the microcontroller in the TBM as well as to meet the requirements set for the TBM. As such, it has been to designed to have a very low memory footprint and it has been designed with simplicity in mind while still being both scalable and extensible. Moreover, the filesystem has been designed to store keys, log files and other files of at most a few kilobytes in an organised fashion.

9.1 Motivation

While there are already many file systems to choose from, only a few would be appropriate for the TBM. Primarily because the resources available by the microcontroller unit are very limited. For instance, the STM32F0 discovery board that is used for the development of the software stack for the TBM only ships 8 kiB of RAM. Furthermore, the TBM planned for production will still only have 64 kiB of RAM. Therefore, the implementation of the file system must have a very low memory footprint to run on the microcontroller unit. Furthermore, many of the file systems available support a lot of features that are not needed within the context of the TBM such as file attributes and permissions, symbolic links, hard links, disk quota, sparse files, etc. Moreover, for such a file system to be useful, a library has to be available that has specifically been designed for microcontrollers. Finally, the file system has to be reliable or it should be possible to easily extend the file system to have a journal to guarantee reliability. Because of these very specific requirements, a custom file system, *µFS*, has been designed for the TBM.

9.2 Disk lay-out

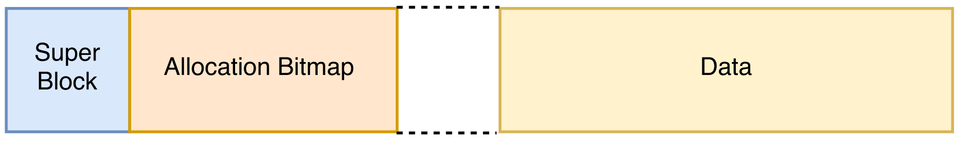

Figure 4: the disk lay-out of µFS.

Figure 4 shows the disk lay-out of µFS. The disk is subdivided into logical block of a fixed size. The first block is the super block which contains an identifier to recognise the existence of a properly formatted µFS file system together with the information necessary to mount a properly working file system. The next few blocks contain a bitmap to mark logical blocks on the disk as either used or available.

9.3 Tree objects

The fundamental data structure of the file system are trees to map virtual addresses within a file to the logical addresses on the disk. The nodes of the tree are logical blocks on the disk that are subdivided into entries that fit a logical address. The intermediate nodes can then be used to map a fragment of the virtual address to a pointer to the next node, whereas the leaf nodes can then be used to map a fragment of the virtual address to the logical address on the disk. Together with the root pointer, the depth of the tree is stored to allow for a scalable virtual address space that can be extended or shrunk to a specific size.

9.4 Directories

The super block consists of a pointer to a tree object that represents the root directory. Each of the logical disk blocks used by this tree object consists of a directory bin up to the size of a logical block on the disk that can contain one or more entries containing metadata such as the name of a file, the file type and a pointer to the tree object of that file.

9.5 API

Because many programmers are familiar with the POSIX standards, the μFS API tries to resemble the functions of this standard as much as possible.

- `struct mufs *mufs_mount(struct flash_dev *dev);`

  Attempts to mount a μFS-formatted flash device. Returns a handle to a filesystem object on success. `NULL` otherwise.

- `void mufs_unmount(struct mufs *fs);`

  Unmounts the mounted filesystems that corresponds with the filesystem handle.

- `int mufs_format(struct flash_dev *dev);`

  Formats μFS on the given flash device. Returns 0 on success. -1 otherwise.

- `char *mufs_abspath(const char *path);`

  Resolves the given path to an absolute path by removing duplicate slashes and resolving both single dot and double dot nodes. The absolute path returned must be freed by the callee.

- `int mufs_stat(struct mufs *fs, const char *path, struct mufs_stat *stat);`

  Gets the file status for a given path including the file type and the size of the file on disk.

- `int mufs_rename(struct mufs *fs, const char *old, const char *new);`

  Renames a file on the filesystem. If the new path already exists and is a directory and the directory does not contain a file with the same name, the file is moved to the directory. If the new path does not exist, the file is renamed. Returns 0 on success. -1 otherwise.

- `struct mufs_dir *mufs_opendir(struct mufs *fs, const char *path);`

  Opens a directory iterator for the given path if it exists and if it is a directory. Returns a directory handle on success. `NULL` otherwise.

- `void mufs_closedir(struct mufs_dir *dir);`

  Closes a directory iterator and frees the memory allocated for it.

- `int mufs_readdir(struct mufs_dir *dir, struct mufs_dirent *dirent);`

  Reads the next directory entry of the directory. Returns 0 on success. -1 otherwise.

- `int mufs_mkdir(struct mufs *fs, const char *path);`

  Creates a directory at the given path if the path does not exist yet and if there is a parent directory. Returns 0 on success. -1 otherwise.

- `int mufs_rmdir(struct mufs *fs, const char *path);`

  Removes the directory at the given path if the path points to an empty directory. Returns 0 on success. -1 otherwise.

- `int mufs_create(struct mufs *fs, const char *path);`

  Creates a file at the given path, if there exists no file at the given path. Returns 0 on success. -1 otherwise.

- `struct mufs_file *mufs_open(struct mufs *fs, const char *path, int mode);`

  Opens the file at the given path if it is a file. The `mode` arguments accepts the following flags:

  - `MUFS_READ`: opens the file for reading.
  - `MUFS_WRITE`: opens the file for writing.
  - `MUFS_APPEND`: opens the file in append mode where writing to the file will always seek to the end before writing.

  Returns a handle to the file on success. `NULL` otherwise.

- `void mufs_close(struct mufs_file *file);`

  Closes a file handle and frees the memory allocated for it.

- `long mufs_seek(struct mufs_file *file, long offset, int whence);`

  Sets the position of the file pointer in the file. The `whence` arguments accepts the following values:

  - `SEEK_SET`: seek from the beginning.
  - `SEEK_CUR`: seek from the current position.
  - `SEEK_END`: seek from the end.

  The `offset` argument contains the relative position to seek. Returns the position of the file handle.

- `size_t mufs_read(struct mufs_file *file, void *data, size_t len);`

  Reads `len` bytes from the file into the buffer given by the `data` argument. Returns the amount of bytes that have been read.

- `size_t mufs_write(struct mufs_file *file, const void *data, size_t len);`

  Writes `len` bytes to the file from the buffer given by the `data` argument. Returns the amount of bytes that have been written.

- `int mufs_unlink(struct mufs *fs, const char *path);`

  Unlinks a file at the given path. Returns 0 on success. -1 otherwise.