Yaffs Direct Interface

Charles Manning

2017-06-07

This document describes the Yaffs Direct Interface (YDI), which allows Yaffs to be simply integrated with embedded systems, with or without an RTOS.

Table of Contents

1 Background

The purpose of this document is to describe the interfacing of the Yaffs Direct Interface (YDI) as well as to provide sufficient information to allow a preliminary evaluation of Yaffs. This document tries to focus on the issues important to the system integrator without getting too detailed about how Yaffs works. Other documents provide an in-depth discussion of how Yaffs works.

2 Licensing

Yaffs was originally released for Linux under the GNU Public License (GPL). Various embedded developers soon identified that Yaffs would be ideal for their applications, but were not able to use GPL based code in their systems. Aleph One has alternative licensing arrangements to support such applications.

3 What are Yaffs and Yaffs Direct Interface?

Yaffs stands for *Yet Another Flash File System*. Yaffs was the first file system designed, from the ground up, for NAND storage.

In 2002 Aleph One set out to identify file system options for using NAND Flash as a file system. Various file systems available at the time were evaluated and all were found lacking in one way or another. The need for a suitable NAND storage file system was identified and Yaffs was designed to fill that need.

Although Yaffs was originally designed for NAND flash, it has been used successfully with NOR flash systems and even as a RAM file system. This allows high reliability file systems to be constructed using NOR flash, with a future migration path to NAND flash for higher density and performance.

Yaffs was originally designed for use with the Linux operating system, but was designed in a very modular way. The operating-system-specific code was kept separate from the main Yaffs file system code. This allows Yaffs to be ported quite cleanly to other operating systems through operating system personality modules. One such personality module is the Yaffs Direct Interface (YDI) which allows Yaffs to be simply integrated with embedded systems, with or without an RTOS.

Yaffs has been used for many products using various operating systems including Windows CE and various RTOSs, including ThreadX, vXworks, pSOS to name just a few.

Note that there is a native port of Yaffs to eCOS that does not use YDI. This is supported and distributed by eCocCentric.(http://www.ecoscentric.com/).

Yaffs2, a more recent release of Yaffs, supports a wider range of NAND flash components including 2k page devices and devices with multi-level cells (MLC), and produces equivalent or better performance. It include Yaffs1 compatibility code so Yaffs1 images still work and migration is quite simple. Yaffs1 or Yaffs2 mode of operation are simply selected by a run-time parameter.

As well as providing a NAND file system, YDI also provides a RAM emulation layer to allow Yaffs to operate as a RAM file system too. While the RAM emulation is perhaps not as efficient as a dedicated RAM file system, this does allow one well proven file system to use both RAM and flash.

4 Why use Yaffs?

Yaffs is the first, and perhaps only, file system designed specifically for NAND flash. This means that Yaffs has been designed to work around the various limitations and quirks of NAND flash, as well as exploit the various features of NAND to achieve an effective file system.

Some features to consider:

- Yaffs has been well proven and has been used to ship in large volumes in several products using many different operating systems, compilers and processors.
- Yaffs is written in portable C and is endian neutral.
- Yaffs provides bad block handling and ECC algorithms to handle deficiencies in NAND flash.
- Yaffs is a log-structured file system which makes it very robust to corruptions caused by power failures etc.
- Yaffs has highly optimised and predictable garbage collection strategies. This makes it high performance and very deterministic when compared with similar file systems.
- Yaffs has a lower memory footprint than most other log-structured flash file systems.
- Yaffs provides a wide range of POSIX-style file system support including directories, symbolic and hard links etc. through standard file system interface calls.
- Yaffs is highly configurable to work with various flash geometries, various ECC options, caching options etc.
- Yaffs Direct Interface is simple to integrate in a system – only a few interface functions are required.
- Yaffs can be used with a wide range of memory technologies.

5 Source Code and Yaffs Resources

Please note the licensing terms that apply to using the Yaffs code.

http://www.yaffs.net serves as the hub for all Yaffs information.

This document refers to the Yaffs source code extensively. and is available via the download link on this site.

There are various other documents and a Yaffs mailing list for Yaffs discussions.

Consulting services are also available.

6 System Requirements

Determining minimum system requirements is often quite difficult. The following are presented as a guideline only. Contact Aleph One for more detailed analysis if required.

- Yaffs is endian-neutral and works fine with little-endian and big-endian processors.
- Yaffs code has been used successfully with many different 32-bit and 64-bit CPUs including MIPS, 68000, ARM, ColdFire, PowerPC and x86 variants. Yaffs has even been used with various DSP architectures. Yaffs should work with 16-bit CPUs too, but this is generally untested and might need some tuning, depending on compiler options.
- Because Yaffs is log structured, RAM is required to build up runtime data structures for acceptable performance. As a rule of thumb, budget approximately 2 bytes per *chunk* of NAND flash, where a *chunk* is typically one page of NAND. For NAND with 512byte pages, budget approximately 4kbytes of RAM per 1Mbyte of NAND. For 2kbyte page devices budget approximately 1kbyte per 1Mbyte of NAND.

7 How to integrate Yaffs with an RTOS/Embedded system

Yaffs Direct Interface (YDI) wraps Yaffs in a way that is simple to integrate. You need to provide a few functions for Yaffs to use to talk to your hardware and RTOS. Yaffs provides a set of POSIX-compliant functions for applications to use to talk to it.

In addition to the Yaffs core file system, the YDI has three parts which are each explained in more detail in further sections:

- **POSIX Application Interface**: This is the POSIX-like interface that the application code uses to access the Yaffs file system. (*open, close, read, write,* etc)
- **RTOS Integration Interface**: These are the functions that must be provided for Yaffs to access the RTOS system resources. (*initialise, lock, unlock, get time, memory allocation, set error, etc*). Note that although we use the term RTOS, the same can be done with other OSs or even with no RTOS at all in a bare-metal system.

- **Flash Device Interface**: This interface support is where the flash device and appropriate device drivers are installed to provide an access path from Yaffs to the flash device(s). Multiple different devices and device types can be supported simultaneously.

7.1 Source Files

The following source files contain the core file system:

yaffs_allocator.c	Allocates Yaffs object and tnode structures.
yaffs_bitmap.c	Block and chunk bitmap handling code.
yaffs_checkpointrw.c	Streamer for writing checkpoint data
yaffs_ecc.c	1-bit Hamming ECC code

yaffs_guts.c	The major Yaffs algorithms.
yaffs_nameval.c	Name/value code for handling extended attributes (xattr).
yaffs_nand.c	Flash interfacing abstraction.
yaffs_packedtags1.c yaffs_packedtags2.c	Tags packing code
yaffs_summary.c	Code for handling block summaries.
yaffs_tagscompat.c	Tags compatibility code to support Yaffs1 mode.
yaffs_tagsmarshall.c	Tags marshalling code.
yaffs_verify.c	Test verification code.
yaffs_yaffs1.c	Yaffs1-mode specific code.
yaffs_yaffs2.c	Yaffs2-mode specific code.

The Yaffs direct interface is in yaffsfs.c, with the interface functions and structures defined in yaffsfd.h.

yaffs_attribs.c	Attribute handling.
yaffs_error.c	Error reporting code.
yaffsfs.c	Yaffs Direct Interface wrapper code.
yaffs_hweight.c	Counts the number of hot bits in a byte, word etc.
yaffs_qsort.c	Qsort used during Yaffs2 scanning

Example flash drivers, simulators and configurations used for testing are are in direct/test-framework directory. These include:

nanddrv.c	A NAND driver layer that performs the commands to access NAND flash in a rudimentary manner. This code should work on many styles of CPU with little modification.
yaffs_nanddrv.c	A wrapper around the NAND driver which plugs it into Yaffs.
nandsim.c	A NAND simulator layer that works with a nandstore_xxx backing store. This simulates a NAND chip.
nandstore_file.c	A storage backend for a NAND simulator that stores the data to file.
nandsim_file.c	A wrapper around nandsim.c and nandstore_file.c which makes a NAND simulator that saves its data in a file.
yaffs_nandsim_file.c	A wrapper which creates a nand simulator instance, hooks up the yaffs NAND driver and then adds it to yaffs for use.
yaffs_nor_drv.c	A driver for CFI-style NOR flash.
yaffs_m18_drv.c	A driver for Intel M18-style NOR flash.
yaffs_osglue.c	An OS glue layer example used in the test harness.

Example and testing build files:

dtest.c	A test harness. Also has sample code that can be used for better understanding of how some function calls work.
yaffscfg2k.c	A test configuration.
yaffs_fileem.c yaffs_fileem2k.c	Nand flash simulation using a file as backing store.

Further testing files are in direct/tests and direct/python.

7.2 Integrating the POSIX Application Interface

The application interface is defined in yaffsfs.h. These provide a POSIX file system interface. For the most part, this interface consists of the standard clib function names prefixed with **`yaffs_`**. For example, **`yaffs_open(), yaffs_close()`** etc.

There is a lot of flexibility in how these functions are used and the system integrator needs to determine the best strategy for the system.

These functions can be used directly, with the code written using these names. For example

```
    int main(...)
{
   // initialisation
    f = yaffs_open(...);
    yaffs_read(f,...);
    yaffs_close(f);
}
```

The yaffs functions can also be wrapped in some way to allow existing code to use yaffs without modification. For example:

```
    #define open(path, oflag, mode)  yaffs_open(path, oflag, mode)
..

int main(...)
{
   // initialisation
    f = open(...);
    read(f,...);
    close(f);
}
```

A more complex approach that can be used if more than one file-system is used is to provide redirection functions. For example:

```
    int open(const char *path,...)
{
    // Determine which file system is being used
    if(it_is_a_yaffs_path(path))
        return yaffs_open(...);
    else {
        // Do something else
    }
}
```

Some RTOS's provide a flexible way to integrate file systems in which case YDI can typically use these interfaces to access yaffs file systems.

8 RTOS Integration Interface

Before Yaffs can be used, it needs to be integrated with the system and configured so that the correct actions are performed.

To do this, you need to modify the configuration file. An example configuration file is presented in yaffscfg.c and yaffscfg2k.c.

This is a relatively straight forward process and defines the RTOS access functions.

The RTOS access functions are:

- **`void yaffsfs_SetError(int err)`**: Called by Yaffs to set the system error.
- **`void yaffsfs_Lock(void)`**: Called by Yaffs to lock Yaffs from multi-threaded access.
- **`void yaffsfs_Unlock(void)`**: Called by Yaffs to unlock Yaffs.
- **`u32 yaffsfs_CurrentTime(void)`**: Get current time from RTOS.
- **`void *yaffsfs_malloc(size_t size)`**: Called to allocate memory.
- **`void yaffsfs_free(void *ptr)`**: Called to free memory.
- **`void yaffsfs_OSInitialisation(void)`**: Called to initialise RTOS context.
- **`void yaffs_bug_fn(const char *file_name, int line_no)`**: Function to report a bug.
- **`int yaffsfs_CheckMemRegion(const void *addr, size_t size, int write_request`**) : Function to check if accesses to a memory region is valid. This function must return zero if the test passes and negative if it fails.

If Yaffs is being used in a multi-threaded environment, then typically **`yaffs_LocalInitialisation()`** will initialise a suitable RTOS semaphore and **`yaffs_Lock()`** and **`yaffs_Unlock()`** will call the appropriate functions to lock and release the semaphore.

`yaffs_CurrentTime()` can be any time increment of use to the system. If this is not required, then it is fine to just use a function that is hard-wired to return zero.

Before the application code uses Yaffs, the **`yaffsfs_OSInitialisation()`** function must be called.

`yaffsfs_CheckMemRegion()` is used to check addresses passed to the various yaffs_xxx functions (yaffs_open(), yaffs_read() etc)). If the function returns -1, then the yaffs_xxx function will return -EFAULT rather than attempt to dereference the address and crash.

An implementation might do something like:

```
   int yaffsfs_CheckMemRegion(const void addr, size_t size,
                 int write_request)
  {
     u32 uaddrb = (u32) addr;
     u32 uaddrt = uaddrb + size -1;

     if( uaddrb >= stack_bottom && uaddrb <= stack_top &&
         uaddrt >= stack_bottom && uaddrt <= stack_top)
         return 0; /* r/w access is OK */

     if( uaddrb >= heap_bottom && uaddrb <= heap_top &&
         uaddrt >= heap_bottom && uaddrt <= heap_top)
         return 0; /* r/w access is OK /

     if( !write_request &&
         uaddrb >= rom_bottom && uaddrb <= rom_top &&
         uaddrt >= rom_bottom && uaddrt <= rom_top)
         return 0; / read access is OK /

     return -1; /* Bad access */

}
```

9 Yaffs NAND Model

Before defining the flash integration interface, it is important to understand the model and corresponding nomenclature used throughout Yaffs and its documentation.

Yaffs uses a fairly abstract model for NAND flash. This allows a lot of flexibility in the way it can be used.

Yaffs is designed for NAND flash and makes the following assumptions and definitions:

- The flash is arranged in ***blocks***. Each block is the same size and comprises an integer number of chunks. Each block is treated as a single erasable item. A block contains a number of chunks.
- A ***chunk*** equates to the allocation units of flash. For Yaffs1, each chunk equates to a 512-byte or larger NAND page (that is, a 512-byte or larger data portion and a 16-byte spare area). For Yaffs2, a chunk will typically be larger (eg. on 2k page devices, a chunk will typically be a single 2k page: 2kbytes of data and 64 bytes of spare). Yaffs2 is capable of working with smaller chunk sizes by using inband tags.
- All accesses (reads and writes) are page chunk aligned. Some reads might only read the spare area where the tags are kept.
- When programming a NAND flash, only the zero bits in the pattern being programmed are relevant, and one bits are "don't care". For example, if a byte already contains the binary pattern b1010, then programming b1001 will result in the pattern which is the logical AND of these two patterns ie. b1000. This is different to NOR flash which would typically abort the attempt to convert a 0 into a 1.

Yaffs identifies blocks by their block number and chunks by the chunk Id. A chunk Id is calculated as:

```
chunkId = block_id * chunks_per_block + chunk_offset_in_block
```

Yaffs treats a blank (0xFF filled) block as being free or erased. Thus, the equivalent of formatting for a Yaffs partition is to erase all the blocks that are not bad.

Yaffs2 has a generalised tags interface to provide better flexibility to cater for a wider range of devices with larger pages and stricter programming limitations. Yaffs2 thus handles more abstract constructs than Yaffs1 and, in older versions of Yaffs, more effort was required to interface to the NAND.

In newer versions of Yaffs, the tags management has been abstracted out of the driver. This makes the driver interface far simpler.

9.1 Formatting

Yaffs treats erased blocks as empty. That makes it really easy to "format" a device: just erase it.. Note that this should be done while the partition is not mounted.

Unmounting will not normally work when the file system is busy (eg. files are open), Some extra non-standard unmount options are provided to enable an unmount to be forced. Note that using these will break any current handles and you will no longer be able to use open files.

There is a **`yaffs_format()`** function that will take care of the details.

9.2 NAND Model considerations for Yaffs1

The historical Yaffs1 tags structure was laid out in accordance with the SmartMedia specification for use with 512-byte pages. There is now much more flexibility in the way things can be configured.

Yaffs1 is capable of being used with a variety of memory layouts so long as the flash supports the ability to overwrite the tags area to set the deletion marker to zero.

If the yaffs_dev's use_nand_ecc field is not set, then yaffs will perform the ECC calculations. If this is set then yaffs will expect the NAND driver to do any required ECC checks. On devices that are not using ECC, for example NOR, set use_nand_ecc=1 and then don't do any ECC anyway.

The Yaffs1 flash model expects an area of 16 spare bytes, some of which are used for tags. The mapping between the spare bytes and the usage of the bytes is specified in yaffs_packed-tags1.c

The Yaffs1 model has been used in many NOR-based systems. One particularly instructive example is in yaffs_nor_drv.c which implements a driver for Yaffs1 mode on CFI-style NOR flash. This does not use ECC but uses checksumming instead.

The yaffs_dev structure has a yaffs_driver structure which holds the function pointers to the driver. For Yaffs1 mode of operations the following are required:

drv_write_chunk_fn	Write a data chunk
drv_read_chunk_fn	Read a data chunk.
drv_erase_fn	Erase a block
drv_initialise_fn	Initialisation.
drv_deinitialise_fn	De-initialise

These functions are described later.

9.3 NAND Model for Yaffs2

The Yaffs2 interface is far more flexible than Yaffs1, but that means a bit more effort must be done in the driver. In particular:

- There is is no standard spare area/ECC layout. The driver must handle all the layout of the spare area and ECC handling.
- There is not a standard implementation for bad block markers. Thus, the driver must provide functions for checking and marking bad blocks.

The yaffs_driver structure for Yaffs2 mode requires the following functions:

drv_write_chunk_fn	Write a data chunk
drv_read_chunk_fn	Read a data chunk.
drv_erase_fn	Erase a block
drv_mark_bad_fn	Mark a block bad
drv_check_bad_fn	Check bad block status of a block.
drv_initialise_fn	Initialisation.
drv_deinitialise_fn	De-initialise

9.4 Overview of flash driver function

The best way to understand how to write these functions is to look at the example drivers.

```
   int (*drv_write_chunk_fn) (struct yaffs_dev *dev, int
nand_chunk,
                                const u8 *data, int data_len,
                                const u8 *oob, int oob_len)
```

This function writes the specified chunk data and oob/spare data to flash.

This function should return YAFFS_OK on success or YAFFS_FAIL on failure.

If this is a Yaffs2 device, or Yaffs1 with use_nand_ecc set, then this function must take care of any ECC that is required.

```
   int (*drv_read_chunk_fn) (struct yaffs_dev *dev, int nand_chunk,
                               u8 *data, int data_len,
                               u8 *oob, int oob_len,
                               enum yaffs_ecc_result *ecc_result)
```

This function reads the specified chunk data and oob/spare data from flash.

This function should return YAFFS_OK on success or YAFFS_FAIL on failure.

If this is a Yaffs2 device, or Yaffs1 with use_nand_ecc set, then this function must take care of any ECC that is required and set the ecc_result.

```
int (*drv_erase_fn) (struct yaffs_dev *dev, int block_no)
```

This function erases the specified block.

This function should return YAFFS_OK on success or YAFFS_FAIL on failure.

```
int (*drv_mark_bad_fn) (struct yaffs_dev *dev, int block_no);
int (*drv_check_bad_fn) (struct yaffs_dev *dev, int block_no);
```

These functions are only required for Yaffs2 mode. They mark a block bad or check if it is bad.

```
int (*drv_initialise_fn) (struct yaffs_dev *dev);
int (*drv_deinitialise_fn) (struct yaffs_dev *dev);
```

These functions provide hooks for initialising or deinitialising the flash driver.

10 Configuration and flash driver installation

For an example of how to do the configuration, refer to yaffs_cfg2k.c and one of the drivers such as yaffs_nand_drv.c

The old mechanism was static and required a table of devices that was installed at start up. This mechanism had shortcomings – in particular it broke modularity and did not allow devices to be added dynamically.

Things are different now. As any time, a yaffs_dev structure can be configured and then registered with Yaffs by using the yaffs_add_device() function.

A yaffs_dev structure has two parts which must be configured before yaffs_add_device() is called: the paramers (dev->param) and the driver (dev->drv). The driver configuration (plugging in the driver functions) has already been described above, so we will now look at the yaffs parameters.

The yaffs_param structure must be set before the yaffs_dev is added with yaffs_add_device() and should not be modified after that.

The params are defined in the Yaffs Tuning Document.

11 Using the POSIX file system interface

This section is directed to readers that might not be very familiar with accessing a file system via POSIX and POSIX-like interfaces.

11.1 Difference to Windows POSIX-like interfaces

MSDOS and Windows provide a POSIX-lke interface. If you are familiar with this then that will help you. There are are however some slight differences, most importantly:

- Windows has no concept of links and has a 1:1 mapping between file names and files.POSIX supports files with zero, one or many names. Refer to the sections on linking and unlinking files. For instance POSIX allows you to unlink a file (ie delete the name for a file) while it is still in use. You can continue to read/write the file but it will be deleted as soon as the last file handle to the file is closed. Windows, on the other hand, does not allow you to unlink a file while it is in use.
- Windows names drives with a letter. POSIX has no drive letters. The whole POSIX file system is in a single directory tree.
- Directories and "folders" are the same thing.

11.2 Fundamental concepts

File	A file is an object that is stored in the file system. yaffs supports the following file types: A regular file is a sequence of bytes stored in the file system. A directory holds links to other files. A symbolic link holds a name to another file. A special file holds device ids or similar information.
Directory	This is a structure that holds links and allows a file to be found by name. Note that a single file may have many links(ie names) and many be found in many places in the directory structure.
Link	This is a name that allows the file to be found. There are two types of link: real links (sometimes called hard links) and symbolic (or soft) links.
Handle	A file handle provides a context from which a file can be accessed and manipulated. A handle has a current file position pointer which points to where the file is being read or written. Multiple handles can simultaneously access a single file.
File mode	The mode is the set of flags that define the type of file as well as the current permissions associated with the file.
Path	This is a string that defines the directories and file name. For example: "/data/direct-1/file1"

File metadata	This is the information about the file, excluding the file contents. The metadata includes the name, mode, size and other details.
Mount, partition, file system	These terms all refer to a single mounted flash area. Each mount is hooked into a common directory structure. The term file system can sometimes be confusing because it is used both for a single mounted partition as well as the the file system code. In other words, we might say that both "/boot" is a file system and might also say that yaffs is a file system.
inode number	The inode number is a unique number that identifies the object within the mount. Note that inodes are not unique between two or more mounts. There is no way to reach a file by inode, only via its links.

11.3 Error code

Most functions result in error codes if the function does not execute successfully. In these cases yaffs calls a function called **`yaffs_SetError()`** with the error value. This function can be configured to deliver the error into the system's error mechanism.

In the case of functions that return integers, a returned value smaller than 0 indicates an error occurred. Functions returning pointers will return NULL if an error occurred.

11.4 Links – the hard kind (not symbolic links)

At first we're just considering real links and not symbolic links. Sometimes real links are also called hard links to differentiate them from symbolic links (which are sometimes called soft links).

The difference between links and files is subtle and can be confusing. Briefly:

- a file is the stored object in the file system
- the link is how it is hooked into the directory structure.

The first link to a file is created by using the file creation calls: **`yaffs_open()`**, **`yaffs_mknod()`**, **`yaffs_symlink()`** and **`yaffs_mkdir()`**.

A link can be destroyed by using the **`yaffs_unlink()`** function. Directories are destroyed using the **`yaffs_rmdir()`** function (oh, and the directory must be empty).

Once a file is both unused (there are no handles open to it) and there are no links to it it is then deleted.

A link can be renamed by using the **`yaffs_rename()`** function.

A file may have many links. These may be created using the **`yaffs_link()`** function. All the links access the same file and there is no one link that is considered “the real link” and others being “the duplicate link”. All links to a file have the same priority. If you look at the inode number for “a” and for “b” then they will be the same because they are refering to exactly the same file.

Thus:

1. If we start with a file called “a” and then call **`yaffs_link(“a”, “b”)`** we now have two links to the same file. We can't tell them apart.
2. If we now unlink “a”, the file still exists under the link “b”.
3. If we now unlink “b” the file will no longer be linked. If there is a handle open to the file then we can still access the file through the handle and continue to read or modify the file. If there is no link to the file it is deleted.
4. If an unlinked file was kept alive because of a handle being open, the file will not show up in the directory structure (because it has no link) and will be deleted as soon as the handle is closed (or if the file system is unmounted).

There are some limitations in how links are used:

- The existing file you are linking to must exist.
- The file you are linking to must be in the same mount (partition) as the link you are creating.
- You cannot create a link to a directory because this would allow the creation of recursive loops.
- The new link name must not exist.

11.5 Symbolic links

Now that you understand how hard links work, we'll introduce symbolic (or soft) links.

A symbolic link holds an alias for a different object. When you use that symbolic link in a path the alias is used instead. To that extent symbolic link and hard links are the same.

You can remove symbolic links with the **`yaffs_unlink()`** function, just like hard links.

There are, however, some important differences. A symbolic link only does its linking function when the path is being evaluated and thus:

- The file you are creating an alias to does not have to exist.
- The file you are creating an alias to does not have to be in the same mount.
- You can create symbolic links to directories.
- A symbolic link will not keep a file alive by itself. Only a hard link can do that.

Thus:

1. If we start with a file called “a” and then call **`yaffs_symlink(“a”, “b”)`** we now have one link to the file called “a” and a symbolic link aliasising from “b” to “a”.
2. If we use **`yaffs_open(“a”)`** or **`yaffs_open(“b”)`** they will open the same file. The path resolution mechanism substitutes in the alias and so will look up “a”. The same applies to almost all functions that takes a path, such as **`yaffs_truncate()`**, **`yaffs_stat()`** etc. The only exceptions are **`yaffs_lstat()`** and **`yaffs_readlink()`** which will treat the path as pointing to the symbolic link and not the file.
3. You can tell “a” and “b” apart by using **`yaffs_lstat()`**. **`yaffs_lstat(“a”)`** will show the information for file “a”. **`yaffs_lstat(“b”)`** will show the information for the symbolic link itself. Note the difference with hard links. With hard links **`yaffs_lstat(“a”)`** and **`yaffs_lstat(“b”)`** would show the same results since they are EXACTLY the same thing being referenced.
4. If we now unlink “a”. then the file is no longer linked and will no longer be accessible from “b”. Note the difference with hard links.

11.6 Handle-based file handling

A handle is file accessor that allows us to access the contents of a regular file. The handle identifies the file being accessed as well as a position pointer tracking where the next read or write operation will take place.

A handle is identified by an integer of value 0 or greater.

A handle is created by using the **`yaffs_open()`** call and returns the handle value. If the handle could not be created then a negative value is returned and the yaffs error code is set.

A handle remains valid until the handle is closed by **`yaffs_close()`**.

The **`yaffs_open()`** call takes three parameters:

- name: Full path name of the file being opened.
- access flags: Flags being used to open this handle
- creation mode: Mode flags used when creating a file.

The access flags are one or more of:

`O_CREAT`	Create file if it does not already exist.
`O_EXCL`	Only use with O_CREAT. Create file if it does not exist. If the file already exists then fail.
`O_TRUNC`	If the file exists, and opening with write access, then truncate the file to zero bytes long.

O_APPEND	Regardless of the handle position, always write to the end of the file.
O_RDWR	Open for read/write access.
O_WRONLY	Open for write access and no read access.
0 (zero)	If neither O_RDWR or O_WRONLY is set then the file will be opened for read-only access.

Where they are not mutually exclusive, these flags are typically combined. For example:

O_CREAT | O_TRUNC | O_RDWR : Create a new file if it does not exist otherwise if the file already exists then truncate to zero length. Typically used to overwrite a file or create if it does not already exist.

O_CREAT | OEXCL | O_WRONLY : Create a new file, opening it for write only. Fails if file already exists.

There are three mode flags that can be controlled or set. These flags may be set by the open that creates the file or later by the **yaffs_chmod()**.

S_IREAD	The file may be opened for read access.
S_IWRITE	The file may be opened for write access.
S_IEXEC	The file may be opened for execution. This is not enforced by yaffs.

Once you have opened a handle you can then do various operations on the file.

yaffs_close(handle)	Close the handle. It is no longer usable.
yaffs_fsync(handle)	Flush out any cached writes to the file, as well as the file meta-data.
yaffs_datasync(handle)	Same as yaffs_fsync() but no file metadata.
yaffs_flush(handle)	Same as yaffs_fsync()
yaffs_dup(handle)	Duplicates a handle.
yaffs_lseek(handle, offset, whence)	Set the handle read/write position. Note that this does not modify the file. You can seek past the end of the file. This does not modify the file size.
yaffs_ftruncate(handle, size)	Change the file size. It can be made bigger or smaller. This does not change the handle read/write position.

`yaffs_fstat(handle, buf)`	Get file status.
`yaffs_fchmod(handle, mode)`	Change the file mode.
`yaffs_read(handle, buffer, nbytes)`	Read data from file at file pointer and update pointer.
`yaffs_pread(handle, buffer, nbytes, position)`	Read data from file at specified position. Does not modify file pointer
`yaffs_write(handle, buffer, nbytes)`	Write data to file at file pointer and update pointer.
`yaffs_pread(handle, buffer, nbytes, position)`	Write data to file at specified position. Does not modify file pointer

11.7 Changing file size

```
yaffs_truncate(path, newSize)
yaffs_ftruncate(handle, newSize)
```

These change a file's size. Note that truncating does not just mean make the file smaller. The file can also be made larger.

Yaffs does not store bytes that it does not need to. If you use truncate to make a file larger then this will not necessarily occupt more flash space.

See also **`yaffs_lseek()`** too. Note that **`yaffs_lseek()`** only changes the position of the write pointer and does not impact on the file size unless the file is written.

11.8 Getting/setting information about files

```
yaffs_chmod(path, mode)
yaffs_fchmod(handle, mode)

yaffs_access(path,amode)

yaffs_stat(path, struct yaffs_stat *buf)
yaffs_fstat(handle, struct yaffs_stat *buf)
yaffs_lstat(path, struct yaffs_stat *buf)
yaffs_readlink(path, buf, bufsize)
```

The chmod calls allow you to set different mode flags for the file. These can only be used to manipulate the permissions flags and not change the mode flags relating to the type of file.

`yaffs_access()` tests that the file can be accessed in the specified mode. This call returns zero on success and -1 is returned

amode is a bitmask of the following values:

`R_OK`	Check read is OK
`W_OK`	Check write is OK
`X_OK`	Check execute is OK.
`0 (zero)`	Just checks the file exists

These can be combined.

For example:

```
        if(yaffs_access(path, R_OK | W_OK) == 0)
          the file exists and can be read and written
        if(yaffs_access(path, 0) == 0)
          the file exists
```

The stat calls retrieve a yaffs_stat structure of data which is defined in yaffsfs.h. There are three versions:

`yaffs_fstat`: Takes a handle.

`yaffs_stat`: Takes a path name and follows symbolic links.

`yaffs_lstat`: Takes a path but does not follow symbolic links.

Thus, if path is a symbolic link, **`yaffs_fstat()`** will give information on the file that the link points to while **`yaffs_lstat()`** will give information about the link itself.

`yaffs_readlink()` will read the read the link value into the buffer provided. For example:

```
          yaffs_symlink(“target”,”/link”); /* create a link called link
    to target */
          yaffs_readlink(“/link”,buffer,10);  /* now buffer contains
    “target” */
```

11.9 Changing the directory structure and names

```
        yaffs_open(path, flags, mode)
```

```
yaffs_mkdir(path, mode)
yaffs_symlink(targetPath, linkPath)
yaffs_mknod(pathname, mode, dev)

yaffs_link(existingPath, newath)

yaffs_unlink(path)
yaffs_rmdir(path)

yaffs_rename(oldPath, newPath)
```

Files of various types, and their first link, are created via the **`yaffs_open()`**, **`yaffs_mkdir()`**, **`yaffs_symlink()`** and **`yaffs_mknod()`** functions.

`yaffs_link()` can be used to create subsequent links to files. Note that you cannot create a link to a directory.

`yaffs_rename()` allows you to rename a link. Note that renaming one link over another link is an atomic operation. That means that even if the operation is interrupted by a power failure, the result will either complete or not happen at all. If the target link is for a directory, then the rename will only work if the target directory is empty.

11.10 Searching directories

```
yaffs_DIR *yaffs_opendir(const YCHAR *dirname) ;
struct yaffs_dirent *yaffs_readdir(yaffs_DIR *dirp) ;
void yaffs_rewinddir(yaffs_DIR *dirp) ;
int yaffs_closedir(yaffs_DIR *dirp) ;
```

Searching is performed by the following sequence of operations:

- Open the directory for reading with **`yaffs_opendir()`**.
- Iterate through the entries in the directory using **`yaffs_readdir()`**. Each read will advance the directory read position.
- You can reset to the start of the directory with **`yaffs_rewinddir()`**.
- Close the directory when finished with **`yaffs_closedir()`**.

11.11 Searching directories (fd interface)

```
int yaffs_open(const YCHAR *dirname, O_RDONLY, 0) ;
struct yaffs_dirent *yaffs_readdir_fd(int fd) ;
void yaffs_rewinddir_fd(int fd) ;
int yaffs_close(int fd) ;
```

This is not a POSIX interface, but provides an alternate interface for reading directories.

Searching is performed by the following sequence of operations:

- Open the directory for reading with **`yaffs_open( name, O_RDONLY, 0)`**.
- Iterate through the entries in the directory using **`yaffs_readdir_fd()`**. Each read will advance the directory read position.
- You can reset to the start of the directory with **`yaffs_rewinddir_fd()`**.
- Close the directory when finished with **`yaffs_close()`**.

11.12 Mount control and similar

```
yaffs_mount(path) ;
yaffs_mount2(path, readOnly);
yaffs_unmount(path);
yaffs_unmount2(path, int force);
yaffs_remount(path, force, readOnly);
yaffs_sync(path) ;
yaffs_format(const YCHAR *path,
    int unmount_flag,
    int force_unmount_flag,
    int remount_flag)
```

A partition must be mounted to be accessed. **`yaffs_mount()`** will mount the partition for read/write access. **`yaffs_mount2()`** can be used to control whether the access is read/write or read only.

A partition can be unmounted to remove it from use. **`yaffs_unmount()`** will fail if any files are in use whereas **`yaffs_unmount2()`** allows you to force the unmount even if files are open. Note that any file handles will then be broken and will no longer access the files.

yaffs_remount() can be used to change the access to a currently mounted partition and is equivalent to making an **unmount2()** call followed by a **mount2()** call.

yaffs_sync() does not change the mount status but flushes out any pending data on a partition that is mounted with write access.

yaffs_format() provides a mechanism to format a device. It has various parameters.
unmount_flag: If set, yaffs_format() will unmount the device before formatting. If this flag is not set and the device is mounted, then the format will fail with EBUSY.

force_unmount: If set, then an unmount is allowed even if the device is busy (eg. files open). If not set, then yaffs_format() will fail (EBUSY) if files are open.

remount_flag: If set, then the device will be remounted after the format if it was mounted.

The force_unmount and remount_flag options only have meaning if unmount_flag is set.

11.13 Extended attributes (xattrt)

Extended attributes provide a mechanism to attach a limited amount of extra data to a file without actually modifying the file. Each xattribute is a named pair of name and some data. Each file can have as many xattribs as there is space for. In Yaffs, this is around chunk-size – 512 bytes. So on devices with 2kbyte chunks you can store up to 1.5kbytes of xattribute information.

Although there are 12 xattr functions, there are really only four basic functions, each with 3 different flavours.

The basic functions are: set an xattr, get an xattr, remove an xattribute and list the xattribute names.

```
int yaffs_setxattr(const char *path, const char *name,
                        const void *data, int size, int flags);
int yaffs_lsetxattr(const char *path, const char *name,
                        const void *data, int size, int flags);
int yaffs_fsetxattr(int fd, const char *name,
                        const void *data, int size, int flags);
```

These functions are called with name being the name of the xattr, a pointer to the data and the size of the data, and flags.

The flags are:

XATTR_CREATE: Create a new xattr, fail with -EEXIST if the xattrib already exists.

XATTR_REPLACE: Replace an existing xattr, fail with -ENOATTR if it does not exist.

zero: Just create with no checks, replace if it already exists.

Setting can fail with -ENOSPC if there is not enough space to store the xattrib.

```
int yaffs_getxattr(const char *path, const char *name,
                        void *data, int size);
int yaffs_lgetxattr(const char *path, const char *name,
                        void *data, int size);
int yaffs_fgetxattr(int fd, const char *name,
                        void *data, int size);
```

These get (fetch) an xattrib.

If the call succeeds, the length of the retrieved xattrib is returned. This call can fail with -ENOATTR if the requested xattr does not exist or -ERANGE if the buffer was too small.

```
int yaffs_removexattr(const char *path, const char *name);
int yaffs_lremovexattr(const char *path, const char *name);
int yaffs_fremovexattr(int fd, const char *name);
```

These functions delete an existing xattr. If the xattr does not exist then -ENOATTR is returned.

```
int yaffs_listxattr(const char *path, char *list, int size);
int yaffs_llistxattr(const char *path, char *list, int size);
int yaffs_flistxattr(int fd, char *list, int size);
```

These functions fetch a list of the xattrib names into the list buffer. The names are NUL terminated. eg. “attrib0\0attrib1\0...”.

The function returns the length of the list, otherwise if the list buffer is too short it returns -ERANGE.

Note: If you call these functions with a list buffer size of zero, this acts as a query of the required buffer size. No data is actually transferred into the list.

For example we might choose to add some xattribs to a media file:

```
//Set description
yaffs_fsetxattr(fd, “user-comment”, “I hate this song!”,18);
yaffs_fsetxattr(fd, “price-in-cents”, &cost, sizeof(cost));
// Get the comment
```

```
yaffs_fgetxattr(fd, “user_comment”, buffer, sizeof(buffer);

//List the attributes in a NUL delimited list
yaffs_flistxattr(fd, list_buf, sizeof(list_buf));
// Now list_buf has user-comment\0price-in-cents\0
yaffs_fremovexattr(fd, ”user-comment”);
```

11.14 Other

```
yaffs_freespace(path);
yaffs_totalspace(path);
yaffs_inodecount(path);
```

These functions provide information on the free and total space (in bytes) for the mount that the path addresses.

`yaffs_inodecount()` returns the number of files on the mount that the path addresses.

12 Example: yaffs_readdir() and yaffs_stat()

This code snippet illustrates using **yaffs_readdir()** and associated functions to iterate through a directory. It also shows how to use **yaffs_lstat()** to get information on a file and interpret it.

```
void dump_directory(const char *dname)
{
  yaffs_DIR *d;
  yaffs_dirent *de;
  struct yaffs_stat s;
  char str[100];

  d = yaffs_opendir(dname);

  if(!d) {
        printf("opendir failed\n");
  } else {
        while((de = yaffs_readdir(d)) != NULL) {
              sprintf(str,"%s/%s",dname,de->d_name);

              yaffs_lstat(str,&s);

              printf("%s inode %d length %d mode %X ",
                          str,s.st_ino,(int)s.st_size,s.st_mode);
              switch(s.st_mode & S_IFMT) {
                    case S_IFREG: printf("data file"); break;
                    case S_IFDIR: printf("directory"); break;
                    case S_IFLNK: printf("symlink -->");
                          if(yaffs_readlink(str,str,100) < 0)
                                printf("no alias");
                          else
                                printf("\"%s\"",str);
                          break;
                    default: printf("unknown"); break;
```

```
                        }
                        printf("\n");

                        if((s.st_mode & S_IFMT) == S_IFDIR)
                                dump_directory_tree_worker(str);
                }

                yaffs_closedir(d);
        }
}
```