yaffs

www.yaffs.net

Yaffs Direct Interface

Charles Manning

2012-07-22

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 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 produces equivalent or better performance. It include Yaffs1 compatibility code so Yaffs1 images still work and migration is quite simple.

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 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.

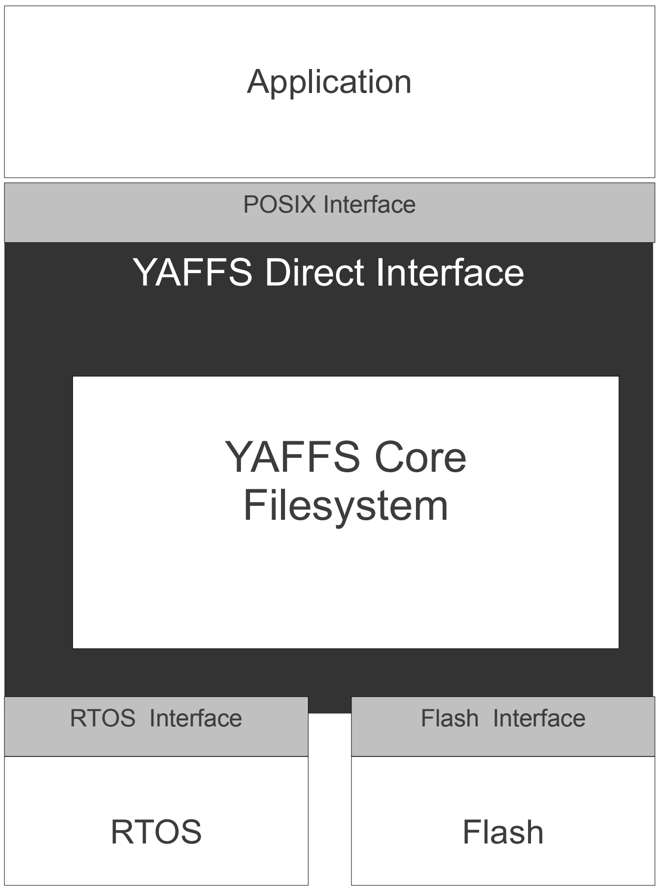
7 How to integrate Yaffs with an RTOS/Embedded system

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 OS. 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 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, set error*)

- **Flash Configuration and Access Interface**: These are the functions that must be provided for Yaffs to access the NAND flash. (*initialise*, *read chunk*, *write chunk*, *erase block*, etc). These functions might be supplied by a chipset vendor or might need to be written by the integrator.



7.1 Source Files

The following source files contain the core file system:

yaffs_allocator.c	Allocates Yaffs object and tnode structures.
yaffs_checkpointrw.c	Streamer for writing checkpoint data
yaffs_ecc.c	ECC code
yaffs_guts.c	The major Yaffs algorithms.
yaffs_nand.c	Flash interfacing abstraction.
yaffs_packedtags1.c	Tags packing code

yaffs_packedtags2.c	
yaffs_qsort.c	Qsort used during Yaffs2 scanning
yaffs_tagscompat.c	Tags compatibility code to support Yaffs1 mode.
yaffs_tagsvalidity.c	Tags validity checking.

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

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.
yaffs_norif1.c	Nor flash simulation and interfacing example. This is configured for M18 flash.
yaffs_ramdisk.c yaffs_ramem2k.c	Simulations using RAM.
ynorsim.c	Another Nor simulation.

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.

7.3 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_LocalInitialisation(void)`**: Called to initialise RTOS context.

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.

Although not shown here, Yaffs also requires memory allocation/free functions which default to **`malloc()`** and **`free()`**. These, and some other functions can be tuned in file ydirectenv.h

Before the application code uses Yaffs, the **`yaffs_StartUp()`** function must be called and the appropriate partitions must be mounted. This is typically done in the system boot code:

```
  ...

  /* System boot code: Start up Yaffs. */
yaffs_StartUp();
yaffs_mount("/boot");

  ...
```

8 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 1010, then programming 1001 will result in the pattern which is the logical AND of these two patterns ie. 1000. 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 more effort is required to interface to the NAND.

8.1 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_Device's useNANDECC filed is not set, then yaffs will perform the ECC calculations. If this iset then yaffs will expect the NAND driver todo any required ECC checks. On devices that are not using ECC, for example NOR, set useNANDECC=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_packedtags1.c

The Yaffs1 model has been used in many NOR-based systems. One particularly instructive example is in yaffs_norif1.c which implements Yaffs1 mode on Intel M18 flash using 1k data pages and simulating a spare area on re-writable areas of flash. This does not use ECC but uses checksumming instead.

8.2 NAND Model for Yaffs2

Yaffs2 uses a different tags layout than Yaffs1. Yaffs2 uses a yaffs_ExtendedTags structure.which is packed according to the code in yaffs_packedtags2.c.

The NAND handler layer must use or populate the fields. The fields have the following meaning:

- **`validMarker0`** Set to 0xAAAAAAAA
- **`chunkUsed`** If 0 then this chunk is not in use
- **`objectId`** The object this chunk belongs to. If 0 then this is not part of an object (ie. unused).
- **`chunkId`** If 0 then this is a header, else a data chunk at the specified position in the file
- **`byteCount`** Number of data bytes in the chunk. Only valid for data chunks

The following fields only have meaning when we read

- **`eccResult`** Result of ECC check when reading
- **`blockBad`** If 0 then the block is not bad.

The following fields have meaning for Yaffs1 and should be zero for Yaffs:

- **`chunkDeleted`** if non-zero, the chunk is marked deleted.
- **`serialNumber`** 2-bit serial number.

The following field has meaning for Yaffs1 and should be zero for Yaffs2:

- **`sequenceNumber`** The sequence number of this block

Optional extra info if this is an object header (Yaffs2 only). Make zero for Yaffs1.

If **`extraHeaderInfoAvailable`** is set, then all the **`extraXXX`** fields must be valid.

- **`extraHeaderInfoAvailable`** There is extra info available if this is not zero.
- **`extraParentObjectId`** The parent object id.
- **`extraIsShrinkHeader`** Is it a shrink header?
- **`extraShadows`** Does this shadow another object?
- **`extraObjectType`** What object type?
- **`extraFileLength`** Length if it is a file.

- **`extraEquivalentObjectId`** Equivalent object Id if it is a hard link.

And finally:

- **`validMarker1`** Must be 0x55555555.

There are quite a few fields, some of which are optional. All the **`extraXXXX`** fields are a way of stuffing more data into object header tags in a way that speeds up scanning. These are optional and should all be zero if not supported.

The easiest way to process the **`yaffs_ExtendedTags`** is to use the functions presented in yaffs_packedtags2.h to do the packing and unpacking. These are:

`yaffs_PackTags2()` and **`yaffs_UnpackTags2()`**.

These functions create or use a packed tags structure which can be stored in the NAND spare area. An example of how to do this is presented in yaffs_fileem2.c

9 NAND Configuration and Access Interface

For an example of how to do the configuration, refer to yaffs_cfg.c and yaffs_cfg2k.c.

This part of the configuration involves configuring the chips and suitable access functions.

This is done in **`yaffs_StartUp().`**

Before launching into the device configuration, it is important to understand that Yaffs supports two different modes of operation:

- Yaffs1: This is the original mode of operation and is only available for 512-byte-per-page devices.
- Yaffs2: This mode of operation supports larger page sizes.

Most configuration options are the same for both, but they have different NAND access functions.

You may have a system that uses a mix of Yaffs1 and Yaffs2 partitions.

9.1 Common configuration items (Yaffs1 and Yaffs2)

The configuration involves two parts:

- Setting up the mount point table.
- Setting up the devices.

A *yaffsfs_DeviceConfiguration* is an entry that has two parts: a mount point name and a pointer to a *yaffs_Device* structure. The mount point name is used to determine where the particular *yaffs_Device* may be found in the directory structure. There can be any number of mount points and they may be nested. For example:

```
static yaffsfs_DeviceConfiguration yaffsfs_config[] = {
        { "/", &ramDev},
        { "/flash/boot", &bootDev},
        { "/flash/flash", &flashDev},
        { "/ram2k", &ram2kDev},
        {(void *)0,(void *)0} /* Terminate list */
};
```

Each logical storage entity is called a *yaffs_Device* and is probably best though of as a partition descriptor. Each *yaffs_Device* can correspond to:

- A whole flash device.
- Part of a flash device.
- Something other than a flash device (e.g. perhaps a RAM emulation). During testing, this emulation capability is often employed to use a host system hard-disk file or nfs-mounted file as an alternative storage mechanism.

Two or more *yaffs_Devices* can reside in a single physical flash device by specifying different start and end blocks. For example here is a case where a single device is split into two partitions:

```
// /boot
......
bootDev.param.startBlock = 0; // First block.
bootDev.param.endBlock = 55; // Last block in 2MB.

// /disk
...
diskDev.param.startBlock = 56; // First block after 2MB
diskDev.param.endBlock = 1023; // Last block in 16MB
```

All *yaffs_Device* structures must be configured according to the Yaffs Tuning document.

9.2 Common Access Functions (Yaffs1 and Yaffs2)

Both Yaffs1 and Yaffs2 configurations must provide pointers to the following functions:

```
  int (*eraseBlockInNAND)(struct yaffs_DeviceStruct *dev,int
blockInNAND)
```

Yaffs calls this function to erase a block of flash.

```
int (*initialiseNAND)(struct yaffs_DeviceStruct *dev)
```

Yaffs calls this function at start up before other functions get called to access the *yaffs_Device*. This allows the system integrator a control point to perform any required initialisation (eg. set up chip selects etc.).

9.3 Yaffs1 Access Functions

For Yaffs1 partitions, the yaffs_Device configuration also specifies pointers to two further functions that Yaffs calls to access the NAND for this yaffs_Device. For a better understanding of these functions, see the section on the Yaffs NAND Model. These functions are:

```
  int (*writeChunkToNAND)(struct yaffs_DeviceStruct *dev,int
chunkInNAND, const __u8 *data, yaffs_Spare *spare)
```

Yaffs calls this function to write data to NAND. The `data` pointer may be NULL if only the spare area is being written, as happens when the chunk is being deleted or retired. If the data pointer is NULL then do not write the data. The `spare` pointer will never be NULL.

```
  int (*readChunkFromNAND)(struct yaffs_DeviceStruct *dev,int
chunkInNAND, __u8 *data, yaffs_Spare *spare)
```

Yaffs calls this function when reading a chunk from flash. The meanings of the arguments are obvious from the above, but note that the data and spare fields might be NULL, in which case those pointers should not be dereferenced.

9.4 Yaffs2 Access Functions

The Yaffs2 mode of operation is far more flexible than the Yaffs1 mode of operation. That allows for far more different types of flash to be used, but slightly increases the complexity of the NAND access functions.

With the Yaffs1 mode of operation, Yaffs performs bad block detection and marking and can optionally perform ECC. It can do this because it assumes that the NAND spare area is structured in a certain way. The Yaffs2 mode of operation can no longer make those assumptions which means that the system integrator must provide slightly more complex functions. However, the interface is still relatively simple, particularly if existing code is used as a basis.

```
  int (*writeChunkWithTagsToNAND) (struct yaffs_DeviceStruct * dev,
                                   int chunkInNAND, const __u8 *
data,
                                   const yaffs_ExtendedTags * tags)
  int (*readChunkWithTagsFromNAND) (struct yaffs_DeviceStruct *
dev,
                                    int chunkInNAND, __u8 * data,
                                                   yaffs_ExtendedTags * tags)
  int (*markNANDBlockBad) (struct yaffs_DeviceStruct * dev, int
blockNo);
  int (*queryNANDBlock) (struct yaffs_DeviceStruct * dev, int
blockNo,
                              yaffs_BlockState * state, int
*sequenceNumber)
```

The **`writeChunkWithTagsToNAND()`** and **`readChunkWithTagsFromNAND()`** functions must save or retrieve the data and extended tags. Please refer to the explanation of the Yaffs2 NAND model for a better understanding of how to handle the tags. Please note that under certain circumstances, the **`data`** and **`tags`** pointers may be NULL and the driver code should ignore transfers from or to pointers that are NULL.

The NAND access functions must also provide a mechanism for marking and tracking bad blocks. If a bad block is detected and Yaffs decides to mark that block bad, the **`markNANDBlockBad()`** function is called.

The **`queryNANDBlock()`** function does two things:

- It determines the block state
- If the block is in use it also retrieves the block's sequence number.

By far the easiest way to understand this all is to refer to the example code provided.

10 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.

10.1 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.

10.2 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.

10.3 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.

10.4 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.

10.5 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 links 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.

10.6 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.
	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 mose.
`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

10.7 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.

10.8 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” */
```

10.9 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.

10.10 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()`**.

10.11 Mount control

```
yaffs_mount(path) ;
yaffs_mount2(path, readOnly);
yaffs_unmount(path);
yaffs_unmount2(path, int force);
yaffs_remount(path, force, readOnly);
yaffs_sync(path) ;
```

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.

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.

10.12 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.

11 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);
                }
            }
```