YAFFS Direct Interface (YDI)

(C) 2005-2010, Aleph One Ltd.

Table of Contents

1 1 Purpose....................................1
2 2 What are YAFFS and YAFFS Direct Interface?....................................2
3 3 Why use YAFFS?....................................3
4 4 Source Code and YAFFS Resources 3
5 5 System Requirements....................................4
6 6 How to integrate YAFFS with an RTOS/Embedded system....................................4
6.1 Source Files....................................5
6.2 POSIX Application Interface....................................6
6.3 RTOS Integration Interface....................................8
7 7 YAFFS NAND Model....................................9
7.1 NAND Model considerations for YAFFS1....................................9
7.2 NAND Model for YAFFS2....................................10
8 8 NAND Configuration and Access Interface....................................11
8.1 Common configuration items (YAFFS1 and YAFFS2)....................................12
8.2 Common Access Functions (YAFFS1 and YAFFS2)....................................13
8.3 YAFFS1 Access Functions....................................13
8.4 YAFFS2 Access Functions....................................14

1 Purpose

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.

2 What are YAFFS and YAFFS Direct Interface?

YAFFS stands for *Yet Another Flash File System*. YAFFS is the first file system that has been 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[1].

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.

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.

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. The YDI code was written to provide a compatibility layer between the core yaffs code and the application.

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

1 Various comparison and and analysis documents are available at http://www.aleph1.co.uk/yaffs

as a dedicated RAM file system, this does allow one well proven file system to use both RAM and flash.

3 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.

4 Source Code and YAFFS Resources

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

This document refers to the YAFFS source code extensively. There are two CVS code base repositories: yaffs and yaffs2. The yaffs repository is largely depreciated and use of the yaffs2 repository is suggested. You can find the code on Aleph One's website using the web-based CVS interface here:

http://www.aleph1.co.uk/cgi-bin/viewcvs.cgi/

Or the full CVS interface described here:

http://www.aleph1.co.uk/cvsuse.html

All general YAFFS information (including licensing information) is available here:

http://www.aleph1.co.uk/yaffs/

5 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.
- 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*[2] 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.

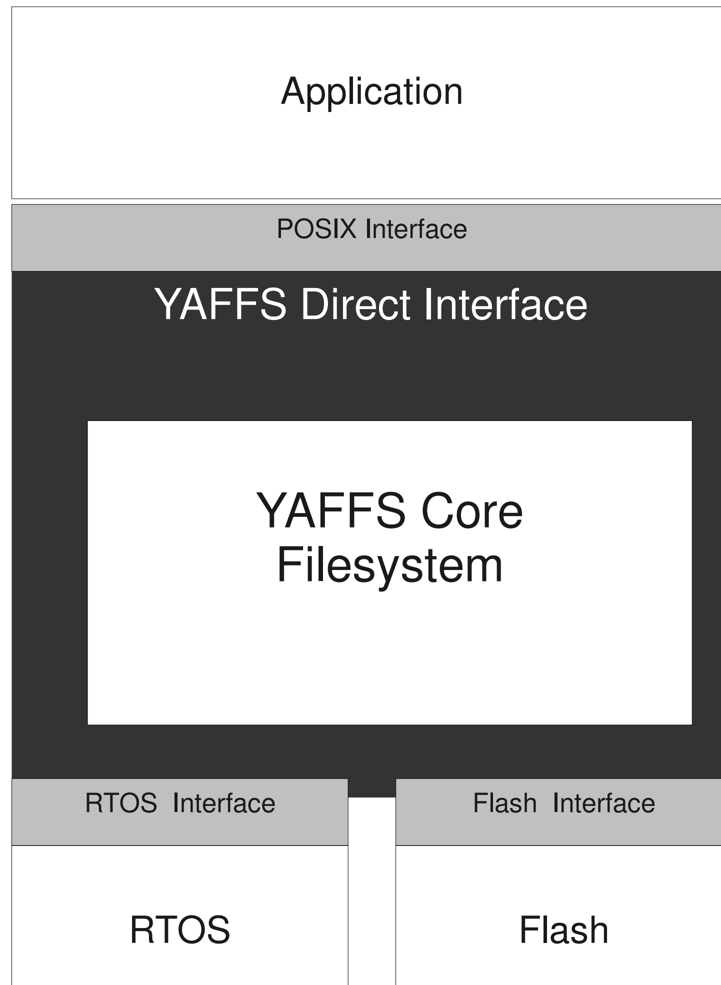
6 How to integrate YAFFS with an RTOS/Embedded system

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:

- **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 much 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).

2 More YAFFS terms are defined in the section entitled YAFFS NAND Model



6.1 Source Files

The following source files contain the core file system

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 yaffs_packedtags2.c	Tags packing code
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.

Example and testing build files:

dtest.c	A test harness
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.
yaffs_ramdisk.c yaffs_ramem2k.c	Simulations using RAM.
ynorsim.c	Another Nor simulation.

6.2 POSIX Application Interface

The application interface is defined in yaffsfs.h[3]. 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:

3 See the section on Source Code and YAFFS Resources to find the files referred to in this document.

```
#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.

6.3 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");
...
```

7 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.

7.1 NAND Model considerations for YAFFS1

The historical YAFFS1 tags structure was layed 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.

7.2 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:

void yaffs_PackTags2(yaffs_PackedTags2 * pt, const yaffs_ExtendedTags * t);

void yaffs_UnpackTags2(yaffs_ExtendedTags * t, yaffs_PackedTags2 * pt);

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

8 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.

8.1 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.

```
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.startBlock = 0; // First block.
bootDev.endBlock = 127; // Last block in 2MB.

// /disk
...
diskDev.startBlock = 128; // First block after 2MB
diskDev.endBlock = 1023; // Last block in 16MB
```

All *yaffs_Device* structures must be configured with the following fields:

- `totalBytesPerChunk`: Number of bytes per chunk. This must be 512 bytes for YAFFS1. If you are using inband tags then this number includes the data + inband tags. If not this only includes the data area.
- `nChunksPerBlock`: Number of chunks per erasable block.
- `nReservedBlocks`: Number of erasable blocks that YAFFS must keep in reserve for garbage collection and to cover for block failures. This must be a minimum of 2, but 5 or so is better. If you are using a media that is not expected to fail (eg. RAM for a RAM disk, or a host file system emulation then 2 is OK).
- `startBlock`: Start block for this *yaffs_Device*.
- `endBlock`: Last block in this *yaffs_Device*.
- `useNANDECC`: If this is non-zero, then YAFFS will not perform ECC and it is assumed that the hardware ECC or specified NAND access functions (or underlying drivers) will perform ECC checks. This only applies to YAFFS1 format.
- `nShortOpCaches`: This configures the number of entries in the YAFFS cache for this *yaffs_Device*. A value of zero turns off the cache. A value of 10 to 20 is recommended for most systems.
- `isYaffs2`: Set according to the type of *yaffs_Device*: 0 for YAFFS1,1 for a YAFFS2.
- `genericDevice`: This is an arbitrary value used to identify the device context. It is important that each *yaffs_Device* has a different value for the yaffs_fstat() POSIX function to work correctly. Typically using `(void *) n`, is sufficient, but this can also be a pointer or some other value as appropriate to establish a system context.

8.2 Common Access Functions (YAFFS1 and YAFFS2)

Both YAFFS1 and YAFFS2 partitions 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.).

8.3 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.

8.4 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.