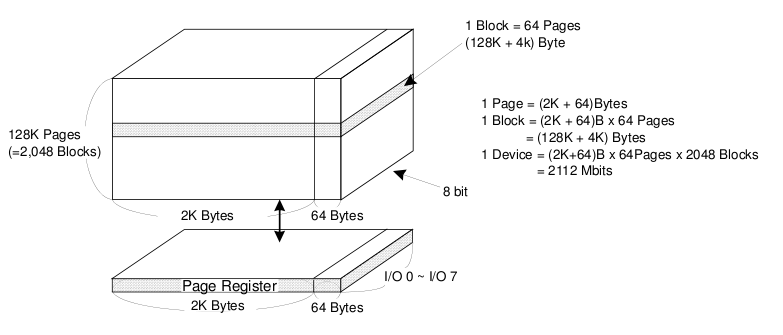
Yaffs2 for MQX Porting User’s Guide

## Introduction

## The purpose of this document is to describe how to bring up Yaffs2 on a board previously unsupported using the Yaffs Direct Interface, with an example using MQX on TWR-K70F120M board.

## NAND Flash Properties

NAND flash is generally arranged in pages and blocks. A page contains a section of data bytes and then a section of out-of-band (OOB) data bytes. Blocks contain a number of pages.  
[](http://www.yaffs.net/yaffs-embedded-structure)

The OOB section of each page is used keep track of various things like file system state, error correction data and bad block marking.

NAND flash is programmed (written to) and read on a largely page basis (though with some level of random access) and is erased on a block basis. That is, to erase a page, you must erase the rest of the containing block along with it.

When programming NAND flash only 1’s can be programmed to 0’s, so to change a byte containing 0x00 to 0xFF it must be erased, along with the rest of the block. However subsequent programs may still be made without an erase if no 0s are to be changed to 1s, this might be useful in a scenario where subsequent data might regularly have this property.

When manufactured NAND flash contains bad blocks, these are blocks that are not reliable and any data in them will commonly corrupt easily. These must be kept track of and the OOB data section is used to mark a bad block.

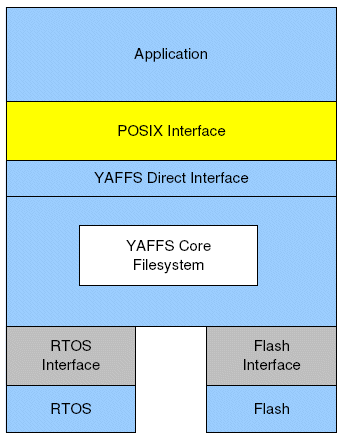
NAND in general is prone to bit errors and for that reason the OOB data is also used to write error correction codes (ECC) so that some errors can be corrected.

## Yaffs and Yaffs Direct Interface

Yaffs stands for Yet Another Flash File System, and it is a file system designed specifically for the characteristics of NAND flash. Its well-proven primary features are:

* Fast - typically much faster than alternatives
* Easily ported (currently ported to GNU/Linux, WinCE, eCOS, pSOS, VxWorks, and various bare-metal systems)
* Log structured, providing wear-levelling and making it very robust
* Supports many flash geometries including 2K-Byte and 512-Byte page NAND flash chips
* Supports MLC and SLC flash
* Very fast mount - almost immediate startup
* Typically uses less RAM than comparable File Systems
* Flexible [Licensing](http://yaffs.net/yaffs-licensing-overview) suitable for most circumstances

Yaffs2 supports 2K-byte page flash as well as 512-byte page flash. (Yaffs1 only supports the original 512-byte page flash.) There is a minimal subset of the file system called Yaffs Direct Interface which is intended to be used in embedded systems. The following figure is the Yaffs embedded structure:



Yaffs embedded structure

Yaffs Direct Interface (YDI) allows Yaffs to be simply integrated with embedded systems, with or without an RTOS. You need to provide a few functions for Yaffs to use to talk to your hardware and OS. The YDI has the following two parts that can be accessed:

* 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.

## Source Files

|  |  |
| --- | --- |
| yaffs\_allocator.c | Allocates Yaffs object and tnode structures. |
| yaffs\_checkpointrw.c  yaffs\_ecc.c  yaffs\_osglue.c | Streamer for writing checkpoint data  ECC Code  RTOS interface to access RTOS system resources |
| 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\_tagsmarshall.c | Tags compatibility code to support Yaffs2 mode. |

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

|  |  |
| --- | --- |
| yaffscfg2k.c | Interface used by yaffs core. |

Detailed source information can refer to the attached package.

## RTOS integration interface

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() andfree(). These, and some other functions can be tuned in file ydirectenv.h

Before the application code uses Yaffs, the yaffs\_start\_up () 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\_start\_up ();  
yaffs\_mount("/nand");

...

## Yaffs NAND Model

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.

## Hardware requirement

TWR-K70F120M

## Porting for MQX on TWR-K70F120M platform

Because NAND driver has been existed in MQX4.1 for TWR-K70F120M, So the driver and some structure should be modified for yaffs2.

The read/write function in the original NAND driver only read/write the data area of the page, not included spare area. But yaffs should used spare area to store file system markers and some useful tags, so yaffs should read/write the whole page including data area and spare area, the full read/write functions should be added into the NAND driver.

The following are porting steps:

1. The main access NAND functions are located in nfc.c. Add the following two functions for accessing the data area and spare area from NAND device:

* uint32\_t nfc\_read\_one\_full\_page(IO\_NANDFLASH\_STRUCT\_PTR nandflash\_ptr, unsigned char \* data\_to\_ptr, unsigned char \* spare\_to\_ptr, uint32\_t page\_number);
* uint32\_t nfc\_write\_one\_full\_page(IO\_NANDFLASH\_STRUCT\_PTR nandflash\_ptr, unsigned char \* data\_from\_ptr, unsigned char \* spare\_from\_ptr, uint32\_t page\_number);

1. Add a BSP configuration macro: BSPCFG\_ENABLE\_NFC\_ECC into user\_config.h. Because NFC of K70 can support hardware ECC calculation. If set BSPCFG\_ENABLE\_NFC\_ECC to 1, ECC would be calculated by NFC and if set BSPCFG\_ENABLE\_NFC\_ECC to 0, ECC would be calculated by Yaffs.
2. Check the configuration: BSPCFG\_ENABLE\_NANDFLASH should be 1, so when startup, the NAND driver can be installed.
3. The structure nandflash\_init\_struct in nandflash.h should be added some members for yaffs2, because yaffs2 should read/write the whole page including data area and spare area, but the original read/write functions only access the data area of the page, not include spare area:

/\* The function to call to read one page of the NAND flash(include spare area) \*/

uint32\_t (\_CODE\_PTR\_ FULL\_PAGE\_READ)(struct io\_nandflash\_struct \*, unsigned char \*, unsigned char \*, uint32\_t); //add for yaffs2

/\* The function to call to write one page of the NAND flash(include spare area) \*/

uint32\_t (\_CODE\_PTR\_ FULL\_PAGE\_WRITE)(struct io\_nandflash\_struct \*,

unsigned char \*, unsigned char \*, uint32\_t); //add for yaffs2

1. As the same above, the structure io\_nandflash\_struct in nandflash.h should be added the same members:

/\* The function to call to read one page of the NAND flash(include spare area) \*/

uint32\_t (\_CODE\_PTR\_ FULL\_PAGE\_READ)(struct io\_nandflash\_struct \*,

unsigned char \*, unsigned char \*, uint32\_t); //add for yaffs2

/\* The function to call to write one page of the NAND flash(include spare area) \*/

uint32\_t (\_CODE\_PTR\_ FULL\_PAGE\_WRITE)(struct io\_nandflash\_struct \*,

unsigned char \*, unsigned char \*, uint32\_t); //add for yaffs2

1. In yaffs2, use IOCTL code to access the NAND driver, so add the following two IOCTL code to nandflash.h:

* #define NANDFLASH\_IOCTL\_READ\_ONE\_FULL\_PAGE \_IO(IO\_TYPE\_NANDFLASH,0x11)
* #define NANDFLASH\_IOCTL\_WRITE\_ONE\_FULL\_PAGE \_IO(IO\_TYPE\_NANDFLASH,0x12)

1. Because the whole data including data area and spare area, the following structure should be added to nandflash.h:

typedef struct nandflash\_page\_data\_struct

{

\_mqx\_uint PAGE\_NUM;

unsigned char \* PAGE\_DATA;

unsigned char \* PAGE\_SPARE;

}NANDFLASH\_PAGE\_DATA\_STRUCT, \*NANDFLASH\_PAGE\_DATA\_STRUCT\_PTR;

1. Then some codes should be changed for adding nfc\_read\_one\_full\_page and nfc\_write\_one\_full\_page functions for yaffs2, these codes are located in init\_nandflash.c and nandflash.c.

## How to use yaffs direct interface

Yaffs direct interface provides some easy-used APIs to access the file system.

The following are some APIs, and detailed information can refer to the source codes:

1. yaffs\_format: format the entire yaffs device.
2. yaffs\_mount: Mount the yaffs device.
3. yaffs\_unmount: Unmount the yaffs device.
4. yaffs\_mkdir: make a directory
5. yaffs\_rmdir: delete a directory.
6. yaffs\_open: Create a new file or open an existed file
7. yaffs\_write: Write the data to the file.
8. yaffs\_read: Read the data from the file.
9. yaffs\_close: Close the file
10. yaffs\_unlink: delete the file from the file system

The following are some examples for how to use YDI:

1. mount a yaffs device which the root is /nand:

yaffs\_start\_up();

yaffs\_mount(“/nand”);

1. make a directory:

yaffs\_mkdir(“/nand/dir1”, S\_IREAD| S\_IWRITE);

1. make a file:

int fHandle = yaffs\_open(“/nand/file1.dat”, O\_RDWR | O\_CREAT | O\_TRUNC, S\_IREAD | S\_IWRITE);

yaffs\_write(fHandle, data\_buf, buf\_length);

yaffs\_close(fHandle);

1. read a file

int fHandle = yaffs\_open(“/nand/file1.dat”,, O\_RDWR, S\_IREAD | S\_IWRITE);

yaffs\_read(fHandle, data\_buf, buf\_length);

yaffs\_close(fHandle);

The more examples can refer to the attached package.

## Support for MCU which has no NFC module

For some platform, for example, TWR-K60F120M, there is not NFC module, so use FlexBus to simulate the NFC, the yaffs2 can also work, and the only effort is to change the NAND driver, the yaffs2 porting codes don’t need to changed.

In other words, yaffs2 for MQX can support NFC and non-NFC simultaneously.

The reference NAND driver code for simulated NFC by Flexbus can refer to the attached “softnand” package.