

# Use Yaffs2 with i.MXRT

DAWEI  
2020



CONFIDENTIAL AND PROPRIETARY



SECURE CONNECTIONS  
FOR A SMARTER WORLD

# 文件系统

现有适用于嵌入式系统的文件系统主要包括以下几类

1. 非掉电恢复，基于block的文件系统，常见的有FAT和EXT2。这两个文件系统在写入文件时是原地更新的，不具备掉电恢复的特性。
2. 日志式的文件系统，比如JFFS,YAFFS等，具备掉电恢复的特性。支持均衡读写和坏块管理。需要占用较多的RAM资源。
3. EXT4和COW类型的btrfs具有良好的恢复性和读写性能，但是需要的资源过多，不适合小型的嵌入式系统。
4. littlefs综合了日志式文件系统和COW文件系统的优点。从sub-block的角度来看，littlefs是基于日志的文件系统，提供了metadata的原子更新；从super-block的角度，littlefs是基于block的COW树。

在i.MXRT系列的生态系统中，SDK已经包含了Littlefs及fatfs文件系统的源代码及示例工程。

- Littlefs作为轻量级且功能完备的小型文件系统，常被与SPI NOR Flash一起使用。从介绍来看支持均衡读写，掉电恢复占用资源小等特点。
- Fatfs分离于磁盘IO，因此不依赖于硬件平台。在SDK中常被用于具有自身FTL(Flash Translation Layer) 功能的存储设备。比如，U 盘、SD 卡、MMC 卡以及固态硬盘等。

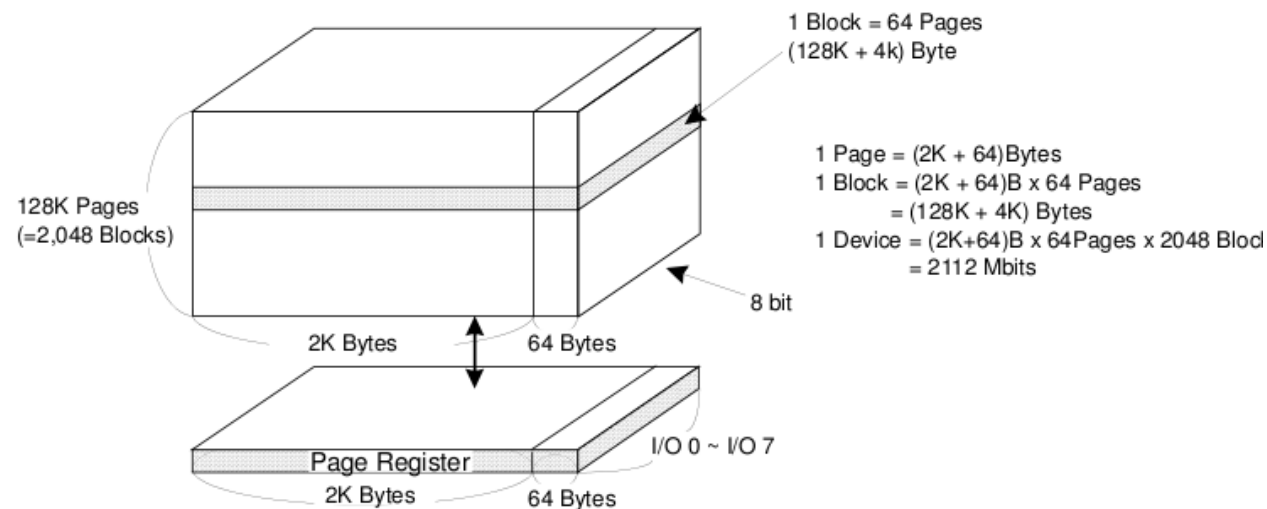
基于大容量的Nand Flash文件系统在MCUX SDK生态环境中缺少示范。

本文主要介绍Yaffs文件系统在MCUX SDK及i.MXRT系列上使用的可能性。

# Yaffs文件系统

# Nand Flash

- Nand flash成本相对低，说白了就是便宜，缺点是使用中数据读写容易出错，所以一般都需要有对应的软件或者硬件的数据校验算法，统称为ECC。但优点是，相对来说容量比较大，现在常见的Nand Flash都是1GB，2GB，更大的8GB的都有了，相对来说，价格便宜，因此适合用来存储大量的数据。其在嵌入式系统中的作用，相当于PC上的硬盘，用于存储大量数据。
- Nand Flash内部是几个Device所组成的，每个Device内部由Block，Block由Page组成。每个page又由Data区域和Spare区域组成。



ONFI规定了Raw NAND内存单元从大到小最多分为如下5层：Device、LUN(Die)、Plane、Block、Page（如下图所示），其中Page和Block是必有的，因为Page是读写的最小单元，Block是擦除的最小单元。而LUN和Plane则不是必有的（如没有，可认为LUN=1, Plane=1），一般在大容量Raw NAND（至少8Gb以上）上才会出现。



Figure 31 Target Memory Organization

# Spare Area(坏块管理, ECC 等)

- **oob / Redundant Area / Spare Area**

每一个页，对应还有一块区域，叫做空闲区域（spare area）/冗余区域（redundant area），而Linux系统中，一般叫做OOB（Out Of Band），这个区域，是最初基于Nand Flash的硬件特性：数据在读写时候相对容易错误，所以为了保证数据的正确性，必须要有对应的检测和纠错机制，此机制被叫做EDC(Error Detection Code)/ECC（Error Code Correction, 或者 Error Checking and Correcting），所以设计了多余的区域，用于放置数据的校验值。

Oob的读写操作，一般是随着页的操作一起完成的，即读写页的时候，对应地就读写了oob。关于oob具体用途，总结起来有：

- 标记是否是坏快
- 存储ECC数据
- 存储一些和文件系统相关的数据。如jffs2就会用到这些空间存储一些特定信息，而yaffs2文件系统，会在oob中，存放很多和自己文件系统相关的信息。

对于MMC、SD卡等自己集成了FTL控制器功能的存储单元而言，对外已经屏蔽了坏块信息，所以就不需要自己管理坏块。

# Nand Flash文件系统需要包含的功能

在选择一个可靠的Nand文件系统需要考虑如下功能：

## ➤ Bad Block Management坏块管理

Nand Flash由于其物理特性，只有有限的擦写次数，超过那个次数，基本上就是坏了。在使用过程中，有些Nand Flash的block会出现被用坏了，当发现了，要及时将此block标注为坏块，不再使用。于此相关的管理工作，属于Nand Flash的坏块管理的一部分工作。

## ➤ Wear-Leveling负载平衡

Nand Flash的block管理，还包括负载平衡。正是由于Nand Flash的block，都是有一定寿命限制的，所以如果你每次都往同一个block擦除然后写入数据，那么那个block就很容易被用坏了，所以我们要去管理一下，将这么多次的对同一个block的操作，平均分布到其他一些block上面，使得在block的使用上，相对较平均，这样相对来说，可以更能充分利用Nand Flash。

## ➤ ECC错误校验码

Nand Flash物理特性上使得其数据读写过程中会发生一定几率的错误，所以要有个对应的错误检测和纠正的机制，于是才有此ECC，用于数据错误的检测与纠正。Nand Flash的ECC，常见的算法有海明码和BCH，这类算法的实现，可以是软件也可以是硬件。不同系统，根据自己的需求，采用对应的软件或者是硬件。基于硬件的ECC也即可以由MCU的控制器实现也可以由NAND Flash本身实现。在i.MXRT10XX系列中，SEMC控制器缺乏硬件ECC，所以ECC可以通过软件或者Nand Flash本身生成。

# NXP SDK/Yaffs License

在开始介绍Yaffs在NXP SDK上的使用时，首先需要了解这两类软件的License。

- NXP MCUX SDK遵循BSD-3-Clause协议，对于用户来说限制非常少。
- Yaffs拥有GPL协议及商用协议两种，在其主页上明确列明了：如果用户试用，可以使用GPL协议，如果需要商用需要单独购买商用软件，无需遵项GPL协议。

<https://yaffs.net/commercial-licences>

<https://yaffs.net/get-yaffs>



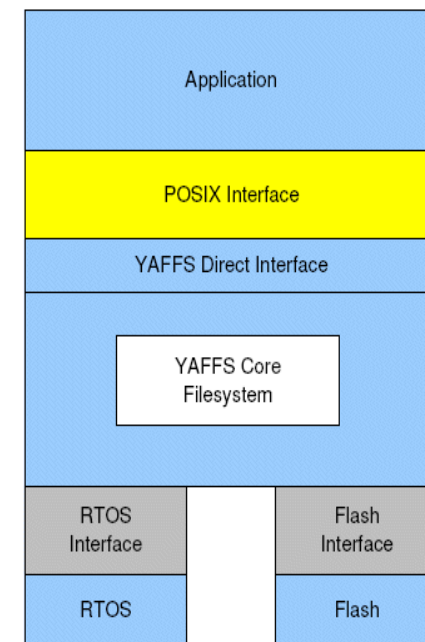
# Yaffs and Yaffs Direct Interface

**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](#) 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.

**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.





# RAM资源占用

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.

对于一般2Gb的Nand flash，现在2KB page的较多，需要的RAM资源256KB往上走，在选择RT系列的时候需要注意RAM资源问题。

Nand Page(Bytes)	RAM Budget(Bytes/1MBytes)	RAM Consumption(Kbytes/1Gbit)
512	4K	4K*128=512
2048	1K	1K*128=128

# Yaffs移植

由此可知，移植Yaffs需要完成3个工作：

1. 预先准备：测试板载Nand操作接口
2. 预先准备：下载Yaffs 源代码
3. 编译环境相关的类型定义
4. RTOS接口的集成：因为一般文件系统都是和RTOS一起工作的多任务环境，需要实现基于文件系统的接口
5. Nand Flash接口系统：比如接在Flexspi/SEMC接口上，需要实现基于这个硬件接口的Flash访问接口。

# SEMC接口访问Nand

1. 以野火的核心板上上面挂载了一颗W29N01HVSINA Nand Flash为例，用户可以采用类似的硬件设计，或者使用Flexspi接口的Nand Flash。
2. 使用MCUX config tool导出一个基于Freertos的SDK工程。鉴于可能需要使用较多的RAM，可以将变量链接到SDRAM，使用DCD初始化SRAM作为内存区域。
3. 通过SEMC Nand访问接口实现并测试如下函数：

Nand\_Flash\_Init

Nand\_Flash\_Read\_Page

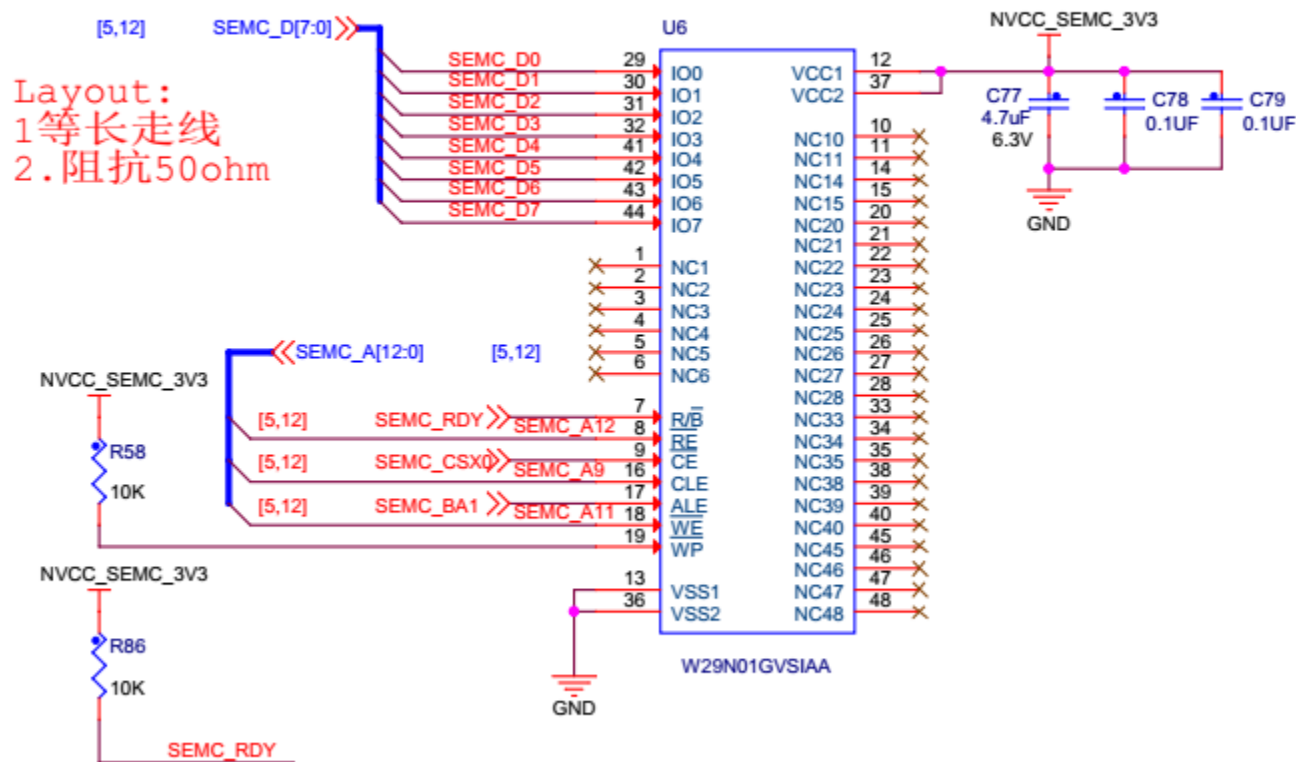
Nand\_Flash\_Read\_Page\_Partial

Nand\_Flash\_Page\_Program

Nand\_Flash\_Page\_Program\_Partial

Nand\_Flash\_Erase\_Block

## NAND FLASH



# SEMC接口访问Nand

Nand\_Flash\_Init(&nandConfig, &nandHandle);

nandConfig初始化设置了Semc及Nand的配置, 有些可以通过ONFI参数读取后续Update, 有些是在这边设定的。尽量根据Nand Flash的数据手册, 填对这边的信息。比如Memory size, 寻址长度等

```
semc_nand_config_t semcNandConfig = {
    .cePinMux = kSEMC_MUXCSX0, /*!< 设置片选 */
    .axiAddress = EXAMPLE_SEMC_NAND_AXI_START_ADDRESS, /*!< 基地址 */
    .axiMemsize_kbytes = 2 * 1024 * 1024, /*!< AXI接口flash大小 8*1024*2*1024*1024 = 16Gb. */
    .ipgAddress = EXAMPLE_SEMC_NAND_IPG_START_ADDRESS, /*!< The base address. */
    .ipgMemsize_kbytes = 2 * 1024 * 1024, /*!< IPG接口flash大小 8*1024*2*1024*1024 = 16Gb. */
    .rdyactivePolarity = kSEMC_RdyActiveLow, /*!< RDY极性 */
    .arrayAddrOption = kSEMC_NandAddrOption_4byte_CA2RA2, //kSEMC_NandAddrOption_5byte_CA2RA3,
    .edoModeEnabled = false, /*!< 地址模式 */
    .columnAddrBitNum = kSEMC_NandColum_12bit,
    .burstLen = kSEMC_Nand_BurstLen1, /*!< 突发长度 */
    .portSize = kSEMC_PortSize8Bit, /*!< 接口位宽 */
    .timingConfig = NULL,
};

semc_mem_nand_config_t semcMemConfig = {
    .semcNandConfig = &semcNandConfig, /*控制器配置结构体. */
    .delayUS = delayUS, /*外部提供的延时函数 */
    .onfiVersion = kNandOnfiVersion_1p0, /*仅支持onfi nand flash */
    .readyCheckOption = kNandReadyCheckOption_SR, //kNandReadyCheckOption_RB /*仅支持onfi nand flash */
    .eccCheckType = kNandEccCheckType_DeviceECC, //dawei: 虽然是软件ecc, 但是program&erase还是需要deviceECC定义去检查
};

nand_config_t nandConfig = {
    /*内存控制器配置应分配特定的控制器配置结构指针*/
    .memControlConfig = (void *)&semcMemConfig,
    .driverBaseAddr = (void *)EXAMPLE_SEMC, /*驱动基地址. */
};
```

# SEMC接口访问Nand

1. ONFI: 定义了Nand的标准操作命令

并可以通过读取Nand内部的参数表获取Nand的生产厂家，容量参数，操作时间参数，地址线模式等。当然这些参数也可以通过数据手册阅读设置。 semc\_nand\_get\_onfi\_timing\_configure ()

<https://www.cnblogs.com/henjay724/p/9152535.html>

## 2.Spare/ECC

每个Page是2KB，而ECC计算块是512Bytes，因此Page区域被均分为4块，分别是Main 0、1、2、3，每块大小为512Bytes，而相应的Spare Area也被均分为4块，分别是Spare 0、1、2、3，每块大小为16bytes，与Main区域一一对应。每个Spare x由2bytes坏块信息、8bytes ECC码、6bytes用户数据组成。要特别说一下的是ECC区域，当芯片硬件ECC功能开启时，8bytes ECC码区域会被自动用来存储ECC信息，而如果芯片没有硬件ECC功能，这个区域可以用来手动地存放软件ECC值。

如图美光的一颗Nand Flash（MT29F4G08ABBxA）的ECC配置：

本身可以打开internal ecc的功能，不需要软件或者SEMC控制器生成ECC。不过我们当前测试的这颗华邦的Nand Flash没有这个功能，需要借助于软件ECC及坏块管理功能。

Command	O/M	1 <sup>st</sup> Cycle	2 <sup>nd</sup> Cycle	Acceptable while Accessed LUN is Busy	Acceptable while Other LUNs are Busy	Target level commands
Read	M	00h	30h		Y	
Copyback Read	O	00h	35h		Y	
Change Read Column	M	05h	E0h		Y	
Read Cache Enhanced	O	00h	31h		Y	
Read Cache	O	31h			Y	
Read Cache End	O	3Fh			Y	
Block Erase	M	60h	D0h		Y	
Interleaved	O		D1h		Y	
Read Status	M	70h		Y	Y	
Read Status Enhanced	O	78h		Y	Y	
Page Program	M	80h	10h		Y	
Interleaved	O		11h		Y	
Page Cache Program	O	80h	15h		Y	
Copyback Program	O	85h	10h		Y	
Interleaved	O	85h	11h		Y	
Change Write Column	M	85h			Y	
Read ID	M	90h				Y
Read Parameter Page	M	ECh				Y
Read Unique ID	O	EDh				Y
Get Features	O	EEh				Y
Set Features	O	EFh				Y
Reset	M	FFh		Y	Y	Y

Table 14 Command set

Figure 80: Spare Area Mapping (x8)

Max Byte Address	Min Byte Address	ECC Protected	Area	Description
1FFh	000h	Yes	Main 0	User data
3FFh	200h	Yes	Main 1	User data
5FFh	400h	Yes	Main 2	User data
7FFh	600h	Yes	Main 3	User data
801h	800h	No		Reserved
803h	802h	No		User metadata II
807h	804h	Yes	Spare 0	User metadata I
80Fh	808h	Yes	Spare 0	ECC for main/spare 0
811h	810h	No		Reserved
813h	812h	No		User metadata II
817h	814h	Yes	Spare 1	User metadata I
81Fh	818h	Yes	Spare 1	ECC for main/spare 1
821h	820h	No		Reserved
823h	822h	No		User metadata II
827h	824h	Yes	Spare 2	User metadata I
82Fh	828h	Yes	Spare 2	ECC for main/spare 2
831h	830h	No		User data
833h	832h	No		User metadata II
837h	834h	Yes	Spare 3	User metadata I
83Fh	838h	Yes	Spare 3	ECC for main/spare 3

Bad Block Information	ECC Parity	User Data (Metadata)
2 bytes	8 bytes	6 bytes

# Yaffs的坏块管理及ECC

- 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 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.

所以Yaffs2的ECC及坏块管理需要在驱动层实现

1. Yaffs中Spare区域前两个字节用于标识坏块；每256Bytes计算3个字节的ECC， 2KBytes page共需要24字节的ECC数据。从26字节开始保存oob数据。
2. 坏块的标识采用像Spare区域的前两个字节写入值来标定
3. 对于由控制器/Nand chip本身产生的ECC校验，坏块标定，在接口驱动中采用自己的方式进行控制。

```
/* Calc ECC and marshall the oob bytes into the buffer */
memset(buffer, 0xff, chip->spare_bytes_per_page);

for(i = 0, e = buffer + 2; i < chip->data_bytes_per_page; i+=256, e+=3)
    yaffs_ecc_calc(data + i, e);

memcpy(buffer + 26, oob, oob_len);
```

```
} « end yaffs_nand_drv_eraseblock »
/*block的第一个page, extra区域写入*/
static int yaffs_nand_drv_MarkBad(struct yaffs_dev *dev, int block_no)
{
    struct nand_chip *chip = dev_to_chip(dev);
    u8 *buffer = dev_to_buffer(dev);
    int nand_chunk = block_no * chip->pages_per_block;
    // struct nand_transfer tr[1];
    status_t status;

    memset(buffer, 0xff, chip->spare_bytes_per_page);
    buffer[0] = 'Y';
    buffer[1] = 'Y';
}
```

# Yaffs and Yaffs Direct Interface

预先准备，下载yaffs资源：在下面站点了解Yaffs的相关介绍及下载Yaffs的可移植代码包

- <https://yaffs.net/documents/yaffs-direct-interface>
- <https://yaffs.net/>
- Yaffs Direct Interface (YDI)可以将Yaffs移植到RTOS/Non-RTOS的环境中。
- 如右，下载的Yaffs源码包包含了核心文件及接口文件

## 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_checkpointw.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





# Yaffs and Yaffs Direct Interface

另外压缩包中也集成了各种Flash驱动接口文件，Nand，Nor，甚至一个nanddrv驱动文件。

一个test配置文件：yaffscfg2k.c

在test-framework下包括了一些基本测试构成文件dtest.c






使用时，需要将yaffs2根目录及direct根目录下的文件迁移到目标工程，并将可能需要修改移植的文件专门放入port文件夹中。

Example flash drivers, simulators and configurations used for testing 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	
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	Nand flash simulation using a file as backing store.
yaffs_fileem2k.c	

-  test\_framework      移植至test\_framework的dtest.c
-  yaffs\_nand\_drv.c      底层Nand Flash基本读写参数函数接口实现，依赖于之前准备工作
-  yaffs\_osglue\_freertos.c      osglue.c是驱动程序的适配层，面向Freertos，提供Freertos的接口层
-  yaffscfg2k.c      Yaffs文件系统的入口，配置文件系统属性，装载回调函数，将设备加入管理列表
-  yportenv.h      环境变量移植



# Yaffs and Yaffs Direct Interface-环境变量

➤ 编译环境相关的类型定义

/\*

Dawei add

yaffs有很多不同的配置选项，这里给出一些你可能会用于Yaffs直接接口的选项。

CONFIG\_YAFFS\_DIRECT - 使用YAFFS直接接口

CONFIG\_YAFFS\_YAFFS2 - 使用YAFFS2

CONFIG\_YAFFS\_PROVIDE\_DEFS - 提供一些文件类型的定义以及 #includes linux/types.h, linux/fs.h, linux/stat.h (参见 yaffs2/devextras.h).

CONFIG\_YAFFSFS\_PROVIDE\_VALUES - 可能使用此选项，这样你可以自定义错误值，以及一些常数，比如O\_RDONLY和系统数值匹配, 如果没有定义这个选项，那么yaffs会包含errno.h,sys/stat.h and fcntl.h (参见yaffs2/direct/yaffsfs.h).

\*/

#define NXP\_YAFFS\_NAME "/yaffs\_nxp"

#define CONFIG\_YAFFS\_DIRECT

#define CONFIG\_YAFFS\_PROVIDE\_DEFS

#define CONFIG\_YAFFSFS\_PROVIDE\_VALUES

#define CONFIG\_YAFFS\_DEFINES\_TYPES

#define CONFIG\_YAFFS\_YAFFS2

typedef long off\_t;

typedef unsigned long loff\_t;

typedef long dev\_t;

typedef int mode\_t;

#define printf PRINTF

/\*Dawei add end\*/

移植环境变量需要注意的：

➤提供一个文件系统名称，在后面测试及加载时统一使用

➤Yaffs文件系统中使用printf打印信息，这边适配SDK的PRINTF

➤/\* 有些RTOS(eg. VxWorks) 需要提供 strlen实现. \*/

size\_t strlen(const char \*s, size\_t maxlen);

```
unsigned int strlen(const char *s, unsigned int max) {  
    register const char *p;  
    for(p = s; *p && max--; ++p);  
    return(p - s);  
}
```

# Yaffs and Yaffs Direct Interface- RTOS接口

- RTOS接口的集成：因为一般文件系统都是和RTOS一起工作的多任务环境，需要实现基于文件系统的接口  
将yaffs\_osglue.c改名为yaffs\_osglue\_freertos.c，实现以下接口：
  - ◆ void yaffsfs\_Lock(void); --提供一个Freertos的信号量全局变量，并实现资源锁定，互斥访问
  - ◆ void yaffsfs\_Unlock(void); --Lock/unlock接口是对接RTOS需要实现的。
  - ◆ u32 yaffsfs\_CurrentTime(void); --返回RTOS的tick count
  - ◆ int yaffsfs\_GetLastError(void);
  - ◆ void yaffsfs\_SetError(int err);
  - ◆ void \*yaffsfs\_malloc(size\_t size); --使用freertos的pvPortMalloc替代malloc，统一heap资源
  - ◆ void yaffsfs\_free(void \*ptr); --vPortFree
  - ◆ void yaffsfs\_OSInitialisation(void); --OS接口初始化函数
  - ◆ void yaffs\_bug\_fn(const char \*file\_name, int line\_no);
  - ◆ int yaffsfs\_CheckMemRegion(const void \*addr, size\_t size, int write\_request); --提供memory区域检查，检查地址合法性
- 另外，文件系统需要占据较多的RAM资源，在freertos\_config.h中设置heap最大到1MB，工程本身的stack和heap也需要相应设置的大一点。

# Yaffs and Yaffs Direct Interface- Nand Flash接口系统

➤ Nand Flash接口系统：比如接在Flexspi/SEMC接口上，需要实现基于这个硬件接口的Flash访问接口。

yaffs\_start\_up()作为整个文件系统的入口，初始化了设备接口，加载了驱动设备。原始的罗列了很多设备接口，Nor Flash,nandsim,甚至RAM disk等。这种选择nand接口即可。

```
/* Install the various devices and their device drivers */
```

```
//yflash2_install_drv("yflash2");
```

```
//yaffs_m18_install_drv("M18-1");
```

```
//yaffs_nor_install_drv("nor");
```

```
//yaffs_nandsim_install_drv("nand", "emfile-nand", 256);
```

```
yaffs_nand_port_install_drv(NXP_YAFFS_NAME, "yaffs_file-nand", BLOCKS_PER_NAND_FLASH);
```

# Yaffs and Yaffs Direct Interface- Nand Flash接口系统

Yaffs\_nand\_drv.c是为nand flash预设计的访问接口，注意同样替换该文件中的malloc函数

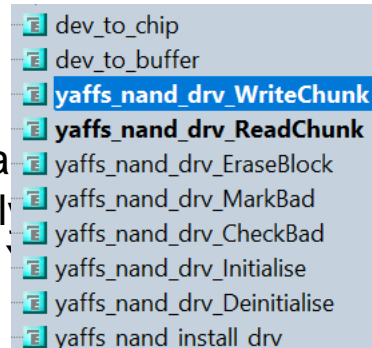
访问API中目标参数都是以chunk为单位的

- A **chunk** equates to the allocation units of flash. For Yaffs1, each chunk equates to a 512-byte or larger (that is, a 512-byte or larger data portion and a 16-byte spare area). For Yaffs2, a chunk will typically be a single 2k page devices, a chunk will typically be a single 2k page: 2kbytes of data and 64 bytes of spare). working with smaller chunk sizes by using inband tags

$\text{chunkId} = \text{block\_id} * \text{chunks\_per\_block} + \text{chunk\_offset\_in\_block}$

- 之前已经说明过了yaffs2的ECC及坏块管理需要在驱动层做，并设置use\_nand\_ecc/is\_yaffs2为1.

- yaffs\_nand\_install\_drv: 安装以下的Nand Flash访问接口
- yaffs\_nand\_drv\_WriteChunk
- yaffs\_nand\_drv\_ReadChunk
- yaffs\_nand\_drv\_EraseBlock
- yaffs\_nand\_drv\_MarkBad
- yaffs\_nand\_drv\_CheckBad
- yaffs\_nand\_drv\_Initialise
- yaffs\_nand\_drv\_Deinitialise



```
dev_to_chip
dev_to_buffer
yaffs_nand_drv_WriteChunk
yaffs_nand_drv_ReadChunk
yaffs_nand_drv_EraseBlock
yaffs_nand_drv_MarkBad
yaffs_nand_drv_CheckBad
yaffs_nand_drv_Initialise
yaffs_nand_drv_Deinitialise
yaffs_nand_install_drv
```

# Yaffs and Yaffs Direct Interface- Nand Flash接口系统

•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.

# Yaffs and Yaffs Direct Interface-注意事项

## ➤ Yaffscfg2k.c有几点注意:

- 其中直接使用malloc、free, 修改为要么使用yaffs\_malloc或者pvPortMalloc以适应freertos
- Nand chip初始化参数要么使用ONFI获取的信息要么根据数据手册手动填入
- 初始化Yaffs的参数: 对于yaffs2注意以下两个参数, 需要设置为1:
  - ① use\_nand\_ecc-使用在yaffs1中, 设置为1驱动层完成ecc校验, 否则yaffs1会自己完成ecc的计算和校验。对于Nor Flash, 设置该位为1, 但是不需要做任何ecc操作。
  - ② is\_yaffs2-使用yaffs2
- 使用宏控制debug trace信息

```
#if (defined(ENABLE_YAFFS_TRACE)) && (ENABLE_YAFFS_TRACE == 1)
unsigned yaffs_trace_mask =

    YAFFS_TRACE_SCAN |
    YAFFS_TRACE_GC |
    YAFFS_TRACE_ERASE |
    YAFFS_TRACE_ERROR |
    YAFFS_TRACE_TRACING |
    YAFFS_TRACE_ALLOCATE |
    YAFFS_TRACE_BAD_BLOCKS |
    YAFFS_TRACE_VERIFY |
    0;
#else
unsigned yaffs_trace_mask = 0; //no yaffs log
#endif
```

```
struct nand_chip *nand_chip_init()
{
    //struct nand_chip *chip = malloc(sizeof(struct nand_chip));
    struct nand_chip *chip = yaffsfs_malloc(sizeof(struct nand_chip));

    memset(chip, 0, sizeof(struct nand_chip));

    chip->private_data = 0;
    /*
    chip->set_ale = NULL;//nand_set_ale;
    chip->set_cle = NULL;//nand_set_cle;
    chip->read_cycle = NULL;//nand_read_cycle;
    chip->write_cycle = NULL;//nand_write_cycle;
    chip->check_busy = NULL;//nand_check_busy;
    chip->idle_fn = NULL;//nand_idle_fn;
    */
    chip->bus_width_shift = 0;

    chip->blocks = BLOCKS_PER_NAND_FLASH;
    chip->pages_per_block = PAGES_PER_BLOCK;
    chip->data_bytes_per_page = DATA_BYTES_PER_PAGE;
    chip->spare_bytes_per_page = SPARE_BYTES_PER_PAGE;

    return chip;
} « end nand_chip_init »
```

```
param->total_bytes_per_chunk = chip->data_bytes_per_page;
param->chunks_per_block = chip->pages_per_block;
param->n_reserved_blocks = 5;
param->start_block = 0; // First block
param->end_block = n_blocks - 1; // Last block
param->is_yaffs2 = 1;
param->use_nand_ecc = 1;
param->n_caches = 10;
param->stored_endian = 2;
```



# Yaffs and Yaffs Direct Interface-动态加载

- yaffs\_nand\_port\_install\_drv实现了驱动和配置的动态加载  
通过将初始化的yaffs\_dev送入yaffs\_add\_device()加载，即可使用该设备  
这样也可以实现多文件系统的同时挂载。

# Yaffs API-常见的使用API

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

10) `yaffs_freespace(path)`;

11) `yaffs_totalspace(path)`;

12) `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.



# Yaffs console

在实际使用Yaffs时可以先采用Console的方式验证，这样可以直观的看到效果。

```
[Press ENTER to execute the previous command again]
>mount

NAND Flash初始化!
NAND Flash初始化成功!
NAND FlashID:0xeff10095
NAND Flash  厂商: WINBOND
NAND Flash ?数据区域大小: 2048字节
NAND Flash ?备用区域大小: 64字节
NAND Flash h块包含的?数: 64?
NAND Flash h层包含的?数: 1024?
NAND Flash设备包含层数: 1层
NAND Flash columnWidth: 12 bits
NAND Flash isFeatureCommandSupport: 0
NAND Flash eccCheckType: Dev_ecc
NAND Flash statusCommandType: CMD normal
NAND Flash ctlAccessMemAddr1: 0x0
NAND Flash ctlAccessMemAddr2: 0x9e000000
mount /yaffs_nxp successful.

[Press ENTER to execute the previous command again]
>dir
/yaffs_nxp/1      inode 257      length 2048      node 4000      directory
/yaffs_nxp/2      inode 258      length 2048      node 4000      directory
/yaffs_nxp/3      inode 259      length 2048      node 4000      directory
/yaffs_nxp/test   inode 260      length 2048      node 4000      directory
/yaffs_nxp/lost+found  inode 2      length 2048      node 4186      directory

Free space in /yaffs_nxp is 132763648

dir command end

[Press ENTER to execute the previous command again]
>mkdir test2
/yaffs_nxp/test2 was created

[Press ENTER to execute the previous command again]
>dir
/yaffs_nxp/1      inode 257      length 2048      node 4000      directory
/yaffs_nxp/2      inode 258      length 2048      node 4000      directory
/yaffs_nxp/3      inode 259      length 2048      node 4000      directory
/yaffs_nxp/test   inode 260      length 2048      node 4000      directory
/yaffs_nxp/test2  inode 3      length 2048      node 4186      directory
/yaffs_nxp/lost+found  inode 2      length 2048      node 4186      directory

Free space in /yaffs_nxp is 132763648

dir command end
```

# 联系窗口/Contact Windows

如果您对阅读本文有疑问，可以通过以下邮件联系方式沟通。

➤ NXP CAS: [Dawei.You@nxp.com](mailto:Dawei.You@nxp.com)