Flash File Systems

Technical issues and implementation details for a family of successful embedded flash file systems

Tim Stoutamore, Principal Engineer
Blunk Microsystems
stout@blunkmicro.com
408-323-1758

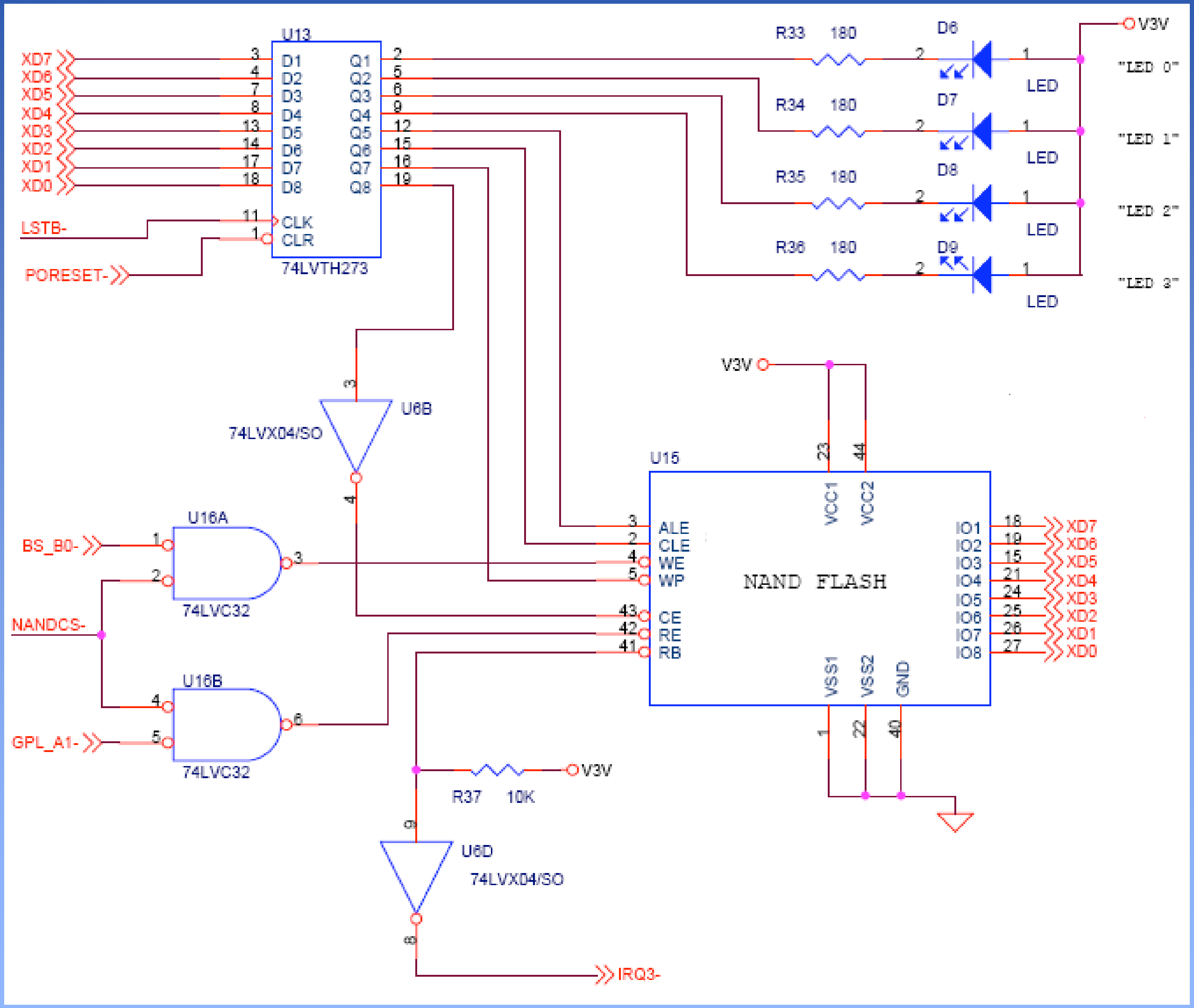
Santa Clara, CA USA
August 2007

Blunk has been providing embedded flash file system products since 2000. Our products have been used in routers, dashboard navigation systems, satellites, set top boxes, and approximately 10% of the cell phone market.

Application Program Interface

POSIX	<stdio.h>
open()	fopen()
close()	fclose()
read()	fread()
write()	fwrite()
mkdir()	fprintf()
lseek()	remove()
unlink()	rename()
chmod()	rewind()
mkdir()	fseek()
truncate()	fgetpos()
...	. . .

Simple NAND Interface



Driver Fragment

```
/*--------------------------------------------------------------------*/
/* Send address as three separate bytes.                              */
/*--------------------------------------------------------------------*/
raiseALE();
NandPort = (ui8)(addr >> 9);
NandPort = (ui8)(addr >> 17);
NandPort = (ui8)(addr >> 25);
lowerALE();

/*--------------------------------------------------------------------*/
/* Send Erase Start command.                                          */
/*--------------------------------------------------------------------*/
raiseCLE();
NandPort = 0xD0;  /* erase start command */
lowerCLE();
```

Flash Memory
SUMMIT

File System Requirements

Need program that will:

- Behave like a traditional file system
- Use flash memory as backing store
- Not violate the requirements/restrictions of flash memory

BLUNK

Obstacles

Single Level Cell NOR

- Erase (up to 3 sec) sets every bit in a large (ex. 64KB) block
- Program allows you to clear individual bits
- "Wear Fatigue": must program/erase all blocks evenly

Multi Level Cell NOR

- Each cell holds one of four voltage levels and represents two bits. Comparators map voltages to the bit assignments: 11, 10, 01, and 00.
- Can't rely on clearing a single bit without affecting adjacent bits.

Obstacles Continued

Single Level Cell NAND

- Partial programming limit (typ. 3-4 per page)
- Bad Blocks: both as shipped and failures during operation
- Bit Errors: requires error detection and correction algorithms

Multi Level Cell NAND

- No partial programming allowed
- Pages must be written in numerical order
- More bit errors: requires correction for 4 or more bit errors per 512 bytes.
- Power-failure while programming a page can corrupt previously written page.

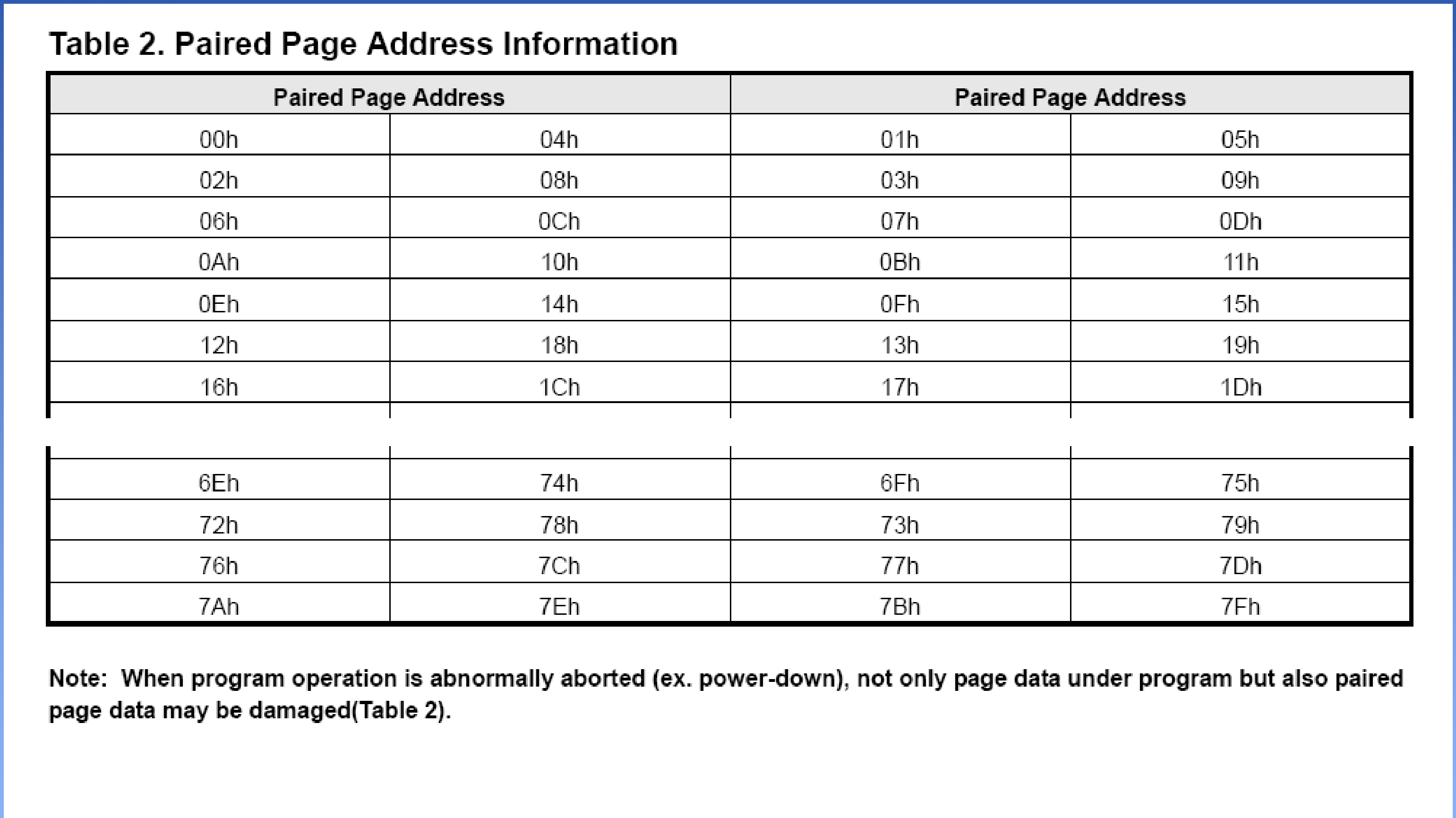


Table 2. Paired Page Address Information

Paired Page Address		Paired Page Address	
00h	04h	01h	05h
02h	08h	03h	09h
06h	0Ch	07h	0Dh
0Ah	10h	0Bh	11h
0Eh	14h	0Fh	15h
12h	18h	13h	19h
16h	1Ch	17h	1Dh
6Eh	74h	6Fh	75h
72h	78h	73h	79h
76h	7Ch	77h	7Dh
7Ah	7Eh	7Bh	7Fh

Note: When program operation is abnormally aborted (ex. power-down), not only page data under program but also paired page data may be damaged(Table 2).

More Obstacles

Multi-Bit Cell NOR

- Single cell holds two bits: program each 'side' separately.
- Some MBC devices limit how many times a word can be partial programmed: can't use bit map algorithms that may repeatedly update the same word.

90nm NOR

- Divided into 1024 byte pages
- No limit on partial programming, but if you do any partial programming you can only use half the page (512 bytes)

Implementation - Control Information Storage

- When control information in flash is updated, it is written to a new location and atomically marked as valid.

  NOR SLC, M18: clear flag in bit array

  NAND, NOR MLC: write control info with CRC

- Control information is given a unique "out-of-band" mark on flash.

  NOR SLC, M18: bit array in block header

  NAND, NOR MLC: per-page 'type' tag

- Serial numbers are used to mark the most recent copy.

Implementation - Reclaiming 'dirty' flash

- Select set of blocks to be erased.
- Copy used data on block(s) to be erased to another block
- Write a new copy of control information
- Erase the selected block(s)

Blunk's Power-Fail Guarantee

Directory structures, closed files, and files open for reading are never at risk.

Data written prior to the previous sync() or fflush() call is not at risk.

Only data written since the last synchronizing operation can be lost.

Implementation - Power Fail Safety

No portion of flash memory that the most recently saved control information 'thinks' contains valid data can be modified. Before a block is erased, its data is copied elsewhere, and a new copy of control information is written that has that block's data marked as unused.

Result

TargetFFS is widely used and relied upon to be power-fail safe. We perform our own extensive automated power-fail testing and customers routinely subject TargetFFS to their own automated power-fail tests. TargetFFS has beat out flash file system products from Microsoft and other competitors, by being the only system that passed the customer's automated power-fail testing.

Technique:
Background Garbage Collection

Recycle operations, which convert dirty sectors to free sectors, may be performed in the background by calling vclean() from the idle task.

Technique: Reserved Memory

A configurable number of flash sectors can be reserved, producing early volume full indication. The file system immediately exchanges reserved free sectors for dirty application sectors. When combined with background recycling, ensures a pool of free sectors is always available, boosting file system responsiveness for user interface applications, even when the volume is full or nearly full.

Technique:
Erase Suspend Support

Erasing a block of NOR flash can take ~3 seconds and a TargetFFS recycle operation may entail erasing multiple blocks. This may cause application read requests to be locked out for an unacceptable amount of time.

TargetFFS uses separate semaphores for read and write access, and supports the NOR flash erase-suspend command. An in progress block erase command will be suspended by a read request from a higher priority task. The erase is resumed after the read completes.

Technique:
NVRAM for Super-Fast Mounts

Some embedded systems contain NVRAM. When available, NVRAM can be used to eliminate the most time consuming aspect of the mount operation: searching for the most recent control information.

- void FsSaveMeta(ui32 vol_id, ui32 location);   called after control write
- int FsReadMeta(ui32 vol_id, ui32 *location);   called during mount

If FsReadMeta() returns zero, its output is used as the location of the most recent copy of control information.

Technique:
Seek Acceleration

```
int fseekmark(FILE *stream, int disable_update);
```

Bookmarks the current file offset. Closest starting point is used when searching for a new file offset.

Technique:
Per-Task CWDs

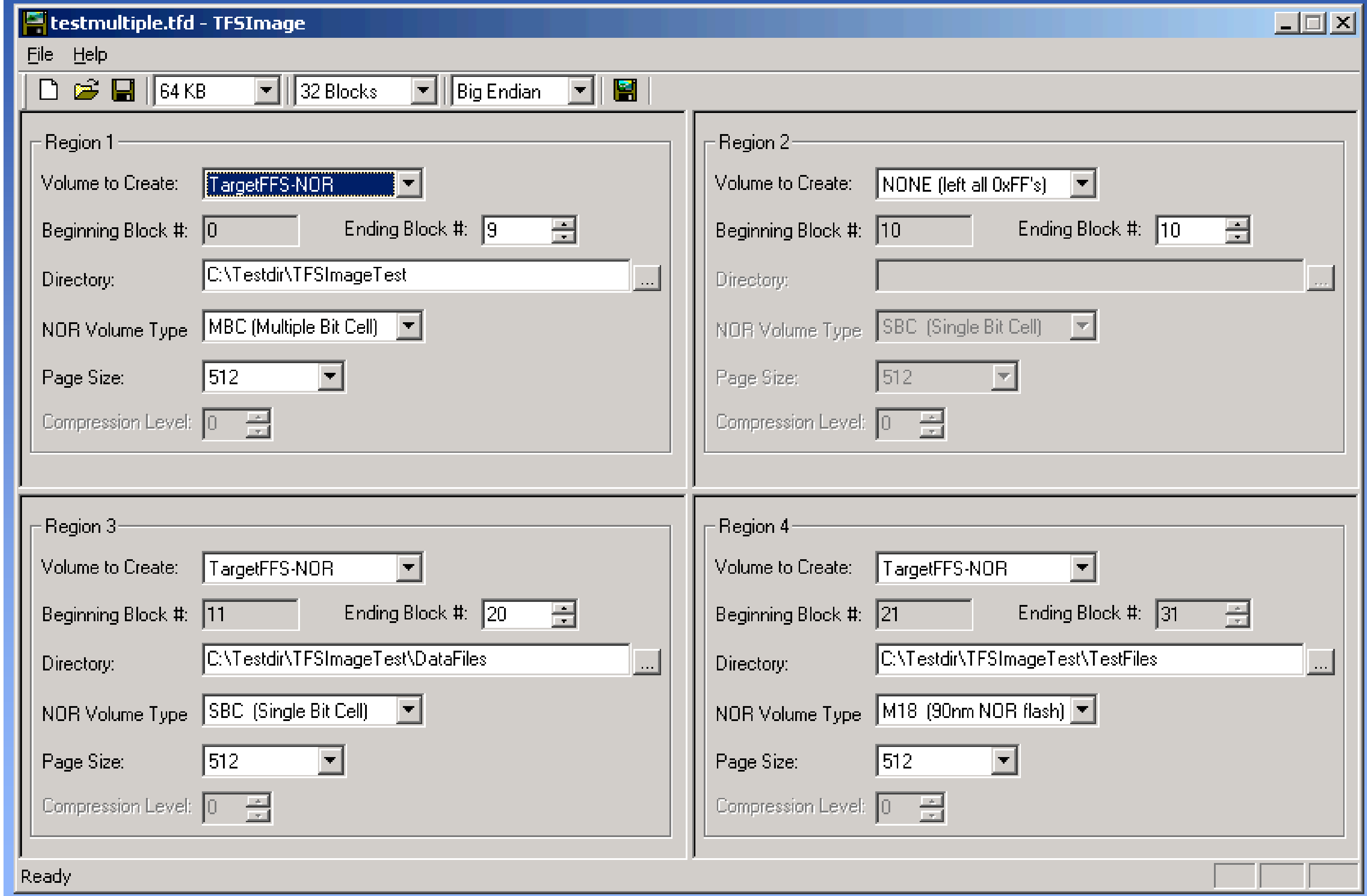
The current working directory (CWD) is specified by two 32-bit variables. The file system calls application functions to save new CWD state variables when 'chdir()' is executed. Another application function is called to read the CWD state variables when resolving relative paths.

Technique:
Access Protections

Supports the “self”, “group”, and “other” file access protections, allowing applications to restrict some operations to privileged tasks. TargetFFS calls FsGetId(), implemented by the application, to get the running task’s user and group IDs. These can be saved in a task-specific RTOS register.

Flash Memory SUMMIT

Tools: Binary Image Tool



64 KB | 32 Blocks | Big Endian

Region 1
Volume to Create: TargetFFS-NOR
Beginning Block #: 0
Ending Block #: 9
Directory: C:\Testdir\TFSImageTest
NOR Volume Type: MBC (Multiple Bit Cell)
Page Size: 512
Compression Level: 0

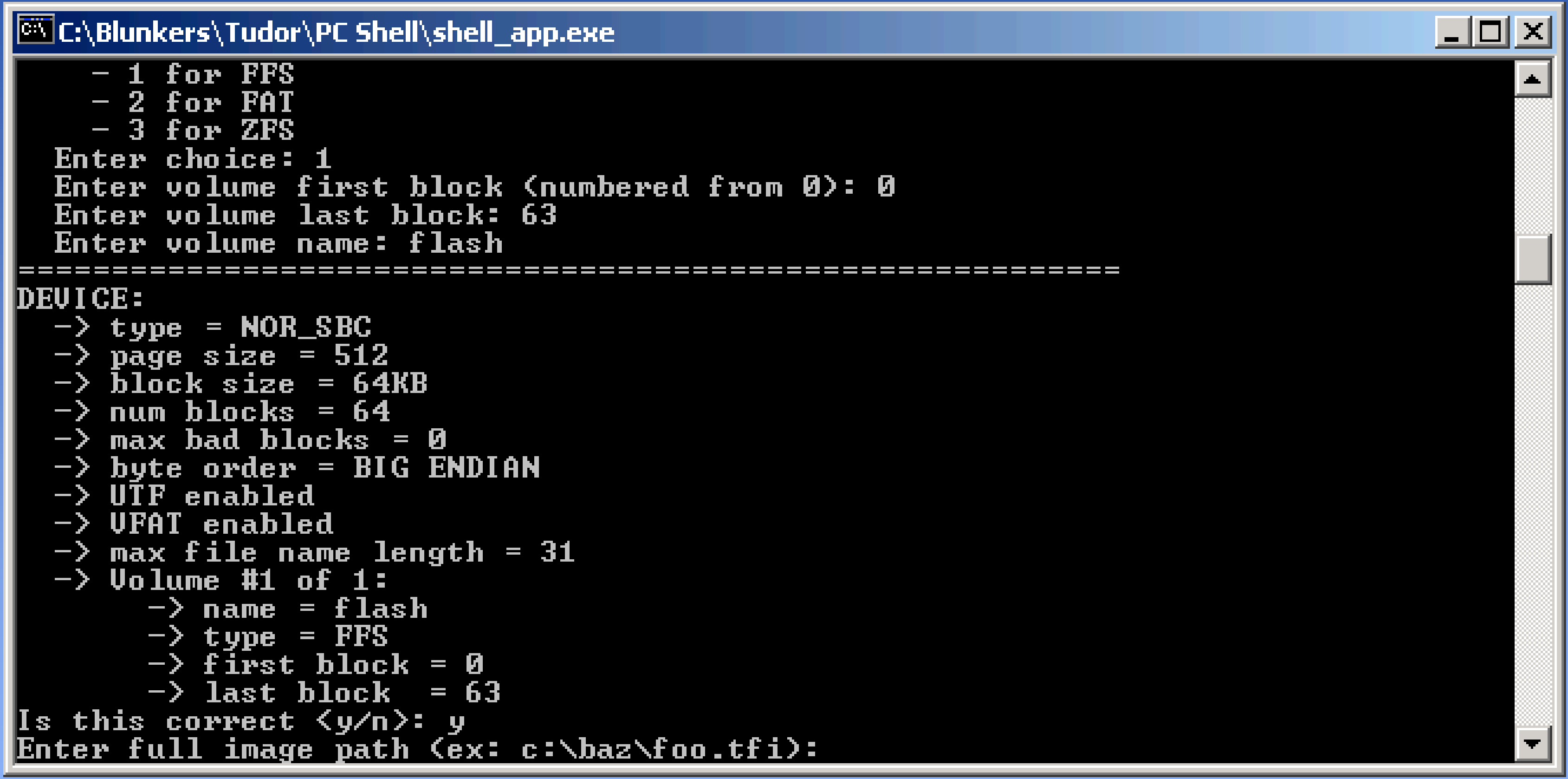
Region 2
Volume to Create: NONE (left all 0xFF's)
Beginning Block #: 10
Ending Block #: 10
Directory:
NOR Volume Type: SBC (Single Bit Cell)
Page Size: 512
Compression Level: 0

Region 3
Volume to Create: TargetFFS-NOR
Beginning Block #: 11
Ending Block #: 20
Directory: C:\Testdir\TFSImageTest\DataFiles
NOR Volume Type: SBC (Single Bit Cell)
Page Size: 512
Compression Level: 0

Region 4
Volume to Create: TargetFFS-NOR
Beginning Block #: 21
Ending Block #: 31
Directory: C:\Testdir\TFSImageTest\TestFiles
NOR Volume Type: M18 (90nm NOR flash)
Page Size: 512
Compression Level: 0

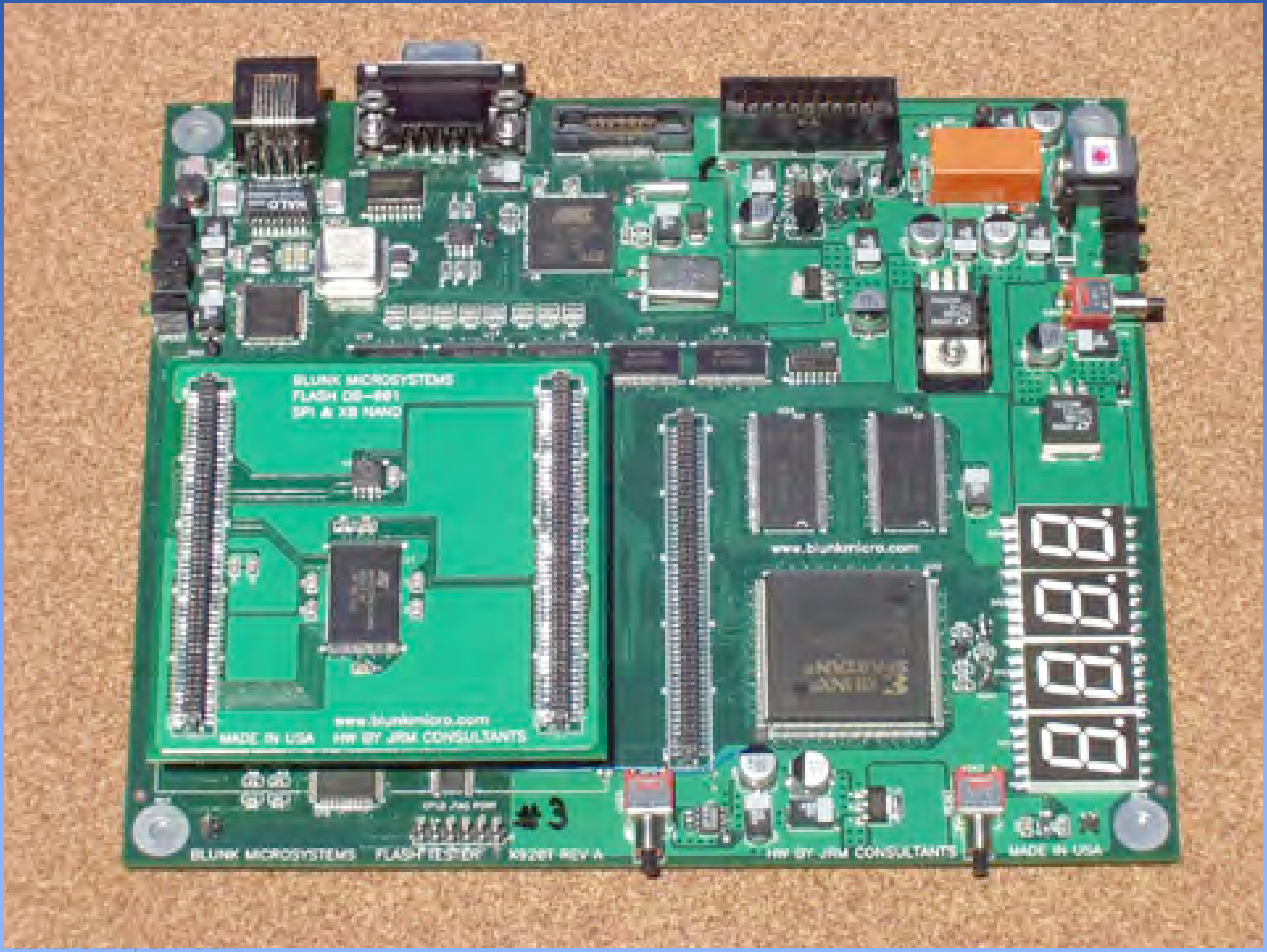
BLUNK

Tools: PC Shell Tool



```
      - 1 for FFS
      - 2 for FAT
      - 3 for ZFS
    Enter choice: 1
    Enter volume first block (numbered from 0): 0
    Enter volume last block: 63
    Enter volume name: flash
============================================================
DEVICE:
  -> type = NOR_SBC
  -> page size = 512
  -> block size = 64KB
  -> num blocks = 64
  -> max bad blocks = 0
  -> byte order = BIG ENDIAN
  -> UTF enabled
  -> VFAT enabled
  -> max file name length = 31
  -> Volume #1 of 1:
       -> name = flash
       -> type = FFS
       -> first block = 0
       -> last block  = 63
Is this correct (y/n): y
Enter full image path (ex: c:\baz\foo.tfi):
```

Tools: Flash Test System

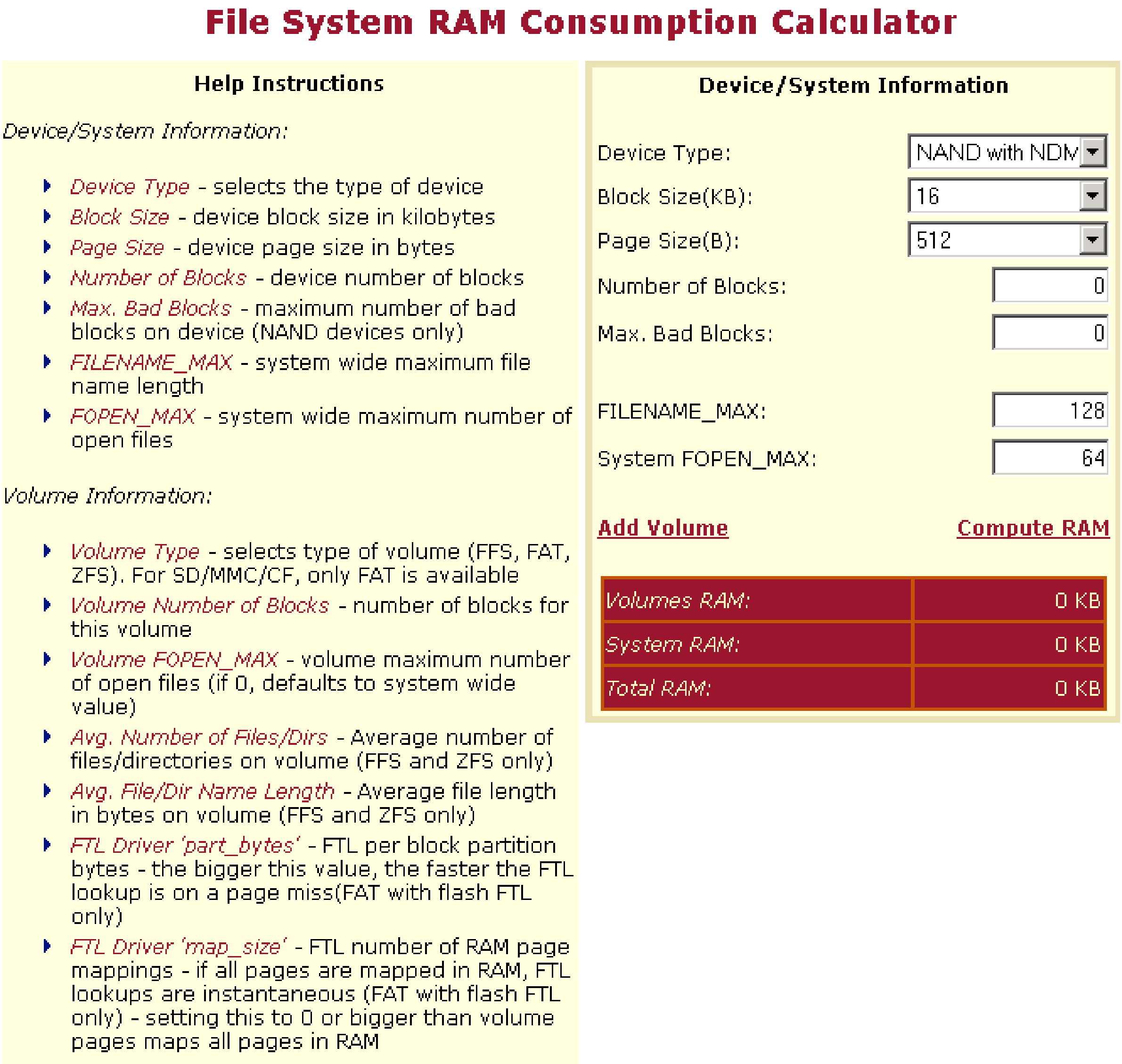


Tools: Driver Test Program

```
Writing page 514, verifying data pattern - finished
Writing page 771, verifying data pattern - finished
Writing page 1028, verifying data pattern - finished
Writing page 1285, verifying data pattern - finished
Writing page 1542, verifying data pattern - finished
Writing page 1799, verifying data pattern - finished
Writing page 2056, verifying data pattern - finished
Writing page 2313, verifying data pattern - finished
Writing page 2570, verifying data pattern - finished
Writing page 2827, verifying data pattern - finished
Writing page 3084, verifying data pattern - finished
Writing page 3341, verifying data pattern - finished
Writing page 3598, verifying data pattern - finished
```

TargetFFS includes a test program that thoroughly exercises the layer below the file system. This verifies that the target is ready to run the flash file system, and is useful for detecting both hardware and driver software errors.

Tools: RAM Footprint Calculator



File System RAM Consumption Calculator

Help Instructions

Device/System Information:

- *Device Type* - selects the type of device
- *Block Size* - device block size in kilobytes
- *Page Size* - device page size in bytes
- *Number of Blocks* - device number of blocks
- *Max. Bad Blocks* - maximum number of bad blocks on device (NAND devices only)
- *FILENAME_MAX* - system wide maximum file name length
- *FOPEN_MAX* - system wide maximum number of open files

Volume Information:

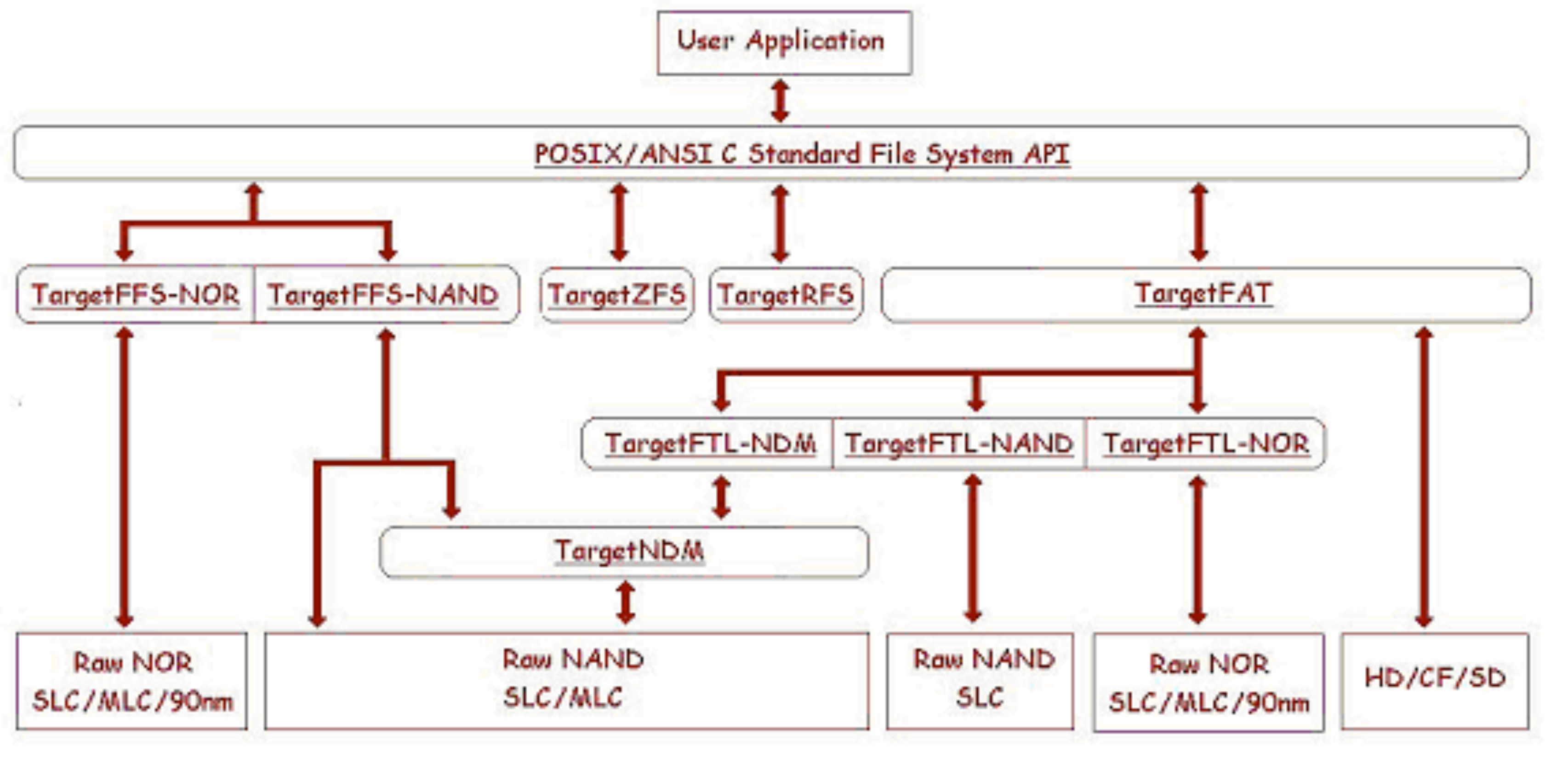
- *Volume Type* - selects type of volume (FFS, FAT, ZFS). For SD/MMC/CF, only FAT is available
- *Volume Number of Blocks* - number of blocks for this volume
- *Volume FOPEN_MAX* - volume maximum number of open files (if 0, defaults to system wide value)
- *Avg. Number of Files/Dirs* - Average number of files/directories on volume (FFS and ZFS only)
- *Avg. File/Dir Name Length* - Average file length in bytes on volume (FFS and ZFS only)
- *FTL Driver 'part_bytes'* - FTL per block partition bytes - the bigger this value, the faster the FTL lookup is on a page miss(FAT with flash FTL only)
- *FTL Driver 'map_size'* - FTL number of RAM page mappings - if all pages are mapped in RAM, FTL lookups are instantaneous (FAT with flash FTL only) - setting this to 0 or bigger than volume pages maps all pages in RAM

Device/System Information

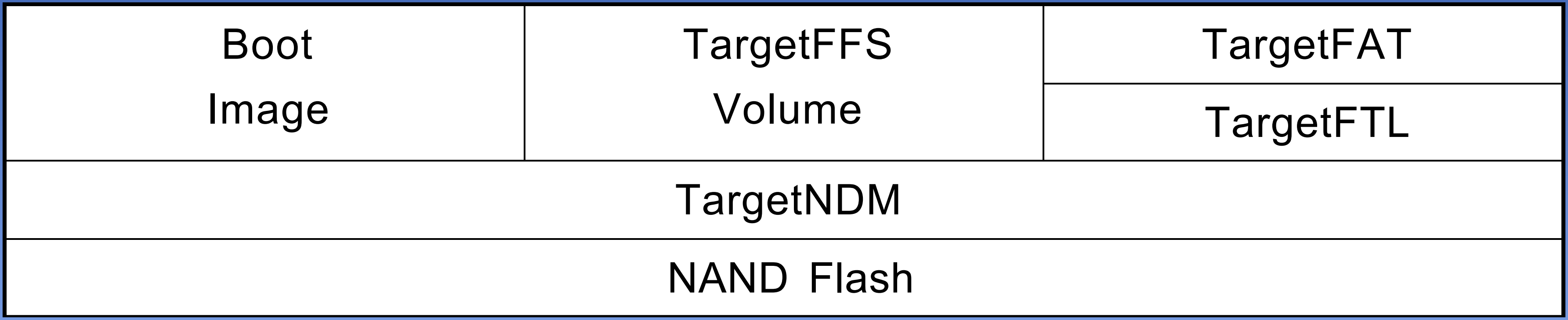
Device Type:	NAND with NDM
Block Size(KB):	16
Page Size(B):	512
Number of Blocks:	0
Max. Bad Blocks:	0
FILENAME_MAX:	128
System FOPEN_MAX:	64

Volumes RAM:	0 KB
System RAM:	0 KB
Total RAM:	0 KB

Full Spectrum of Embedded File Systems



TargetNDM



Boot Image

TargetFFS Volume

TargetFAT

TargetFTL

TargetNDM

NAND Flash

Take-Away

With the right software support, implementing embedded flash file systems on raw NAND and NOR chips is a low-cost, reliable approach.

More information at: www.blunkmicro.com