FLASH Programming in Production of VS1000 Applications

Purpose

This document describes the different possibilities for programming firmware and content to VS1000-based systems in production.

Revision History			
Rev	**Date**	**Author**	**Description**
0.50	2010-03-26	POj	Initial revision.

Table of Contents

1 NAND FLASH

1.1 FLASH Organization

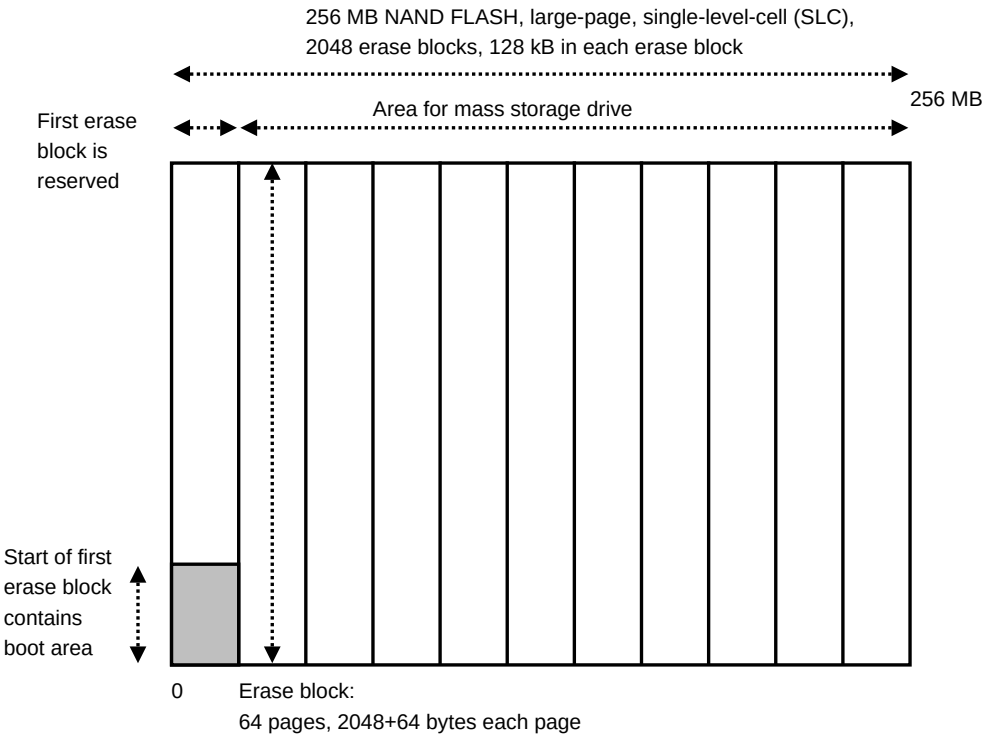

Figure 1: NAND FLASH

NAND FLASH consists of 512-byte or 2048-byte pages with 16-byte or 64-byte spare areas, respectively. These are called small-page and large-page memories. The smallest programmable area is normally 512 bytes. Single-level large-page memories allow upto 8 partial programming cycles, so four 512-byte programming operations and four 16-byte programming operations are possible.

To be able to program a page or part of a page, the page must first be erased. The smallest erasable size is one erase block. An erase block consists of a number of pages, for example 64 pages, 2048 bytes each. See the FLASH datasheet for actual numbers.

The NAND flash technology does not guarantee that all pages in an erase block will work reliably. This is why erase blocks containing such faulty pages have been marked at factory, and can not be used for storing data. Bad pages can also develop during use of the memory, so avoiding bad blocks must work dynamically. This requires mapping between logical and physical blocks.

First Erase Block

However, the first erase block of every NAND FLASH chip is guaranteed to be available. This first erase block (typically 128 kB in single-level FLASH chips) is thus ideal for boot code and other similar uses.

VS1000 uses the start of the first page to define FLASH parameters such as type, erase block size and total FLASH size. The rest of the first erase block can be used for boot code.

Other Erase Blocks

The rest of the FLASH is visible through USB mass storage mode of VS1000. The disk is used to store audio content and possibly other files as well.

But these other erase blocks are not guaranteed to work, and bad blocks can also develop during use. If the FLASH reports an erase error, that erase block must be marked bad, and a replacement block is taken into use. However, read errors are normal and should be handled using re-reads and error correction code (ECC).

Remapping and Weal Levelling

Because bad erase blocks can exist anywhere in the FLASH, data remapping is required. This also provides logical to physical translation.

Some areas in the logical disk, such as directories, are written more frequently than other areas. Because each erase block has only a limited amount of erase cycles, the lifetime of the FLASH can be increased when the writes are distributes evenly through the FLASH.

VS1000 uses a custom block remapping and wear levelling algorithm to keep data safe even when there are bad erase blocks, and to use the erase blocks uniformly.

1.2 Boot Code and Content

In production you need to program both the custom boot code, and the content. There are two main options used with NAND FLASH: the rewritable and mapperless.

	Boot Code Programming	**Content Programming**
Rewritable	UART, USB, preprogrammed	USB, can be rewritten
Mapperless	preprogrammed	preprogrammed, not changable

1.3 Rewritable NAND-FLASH Organization

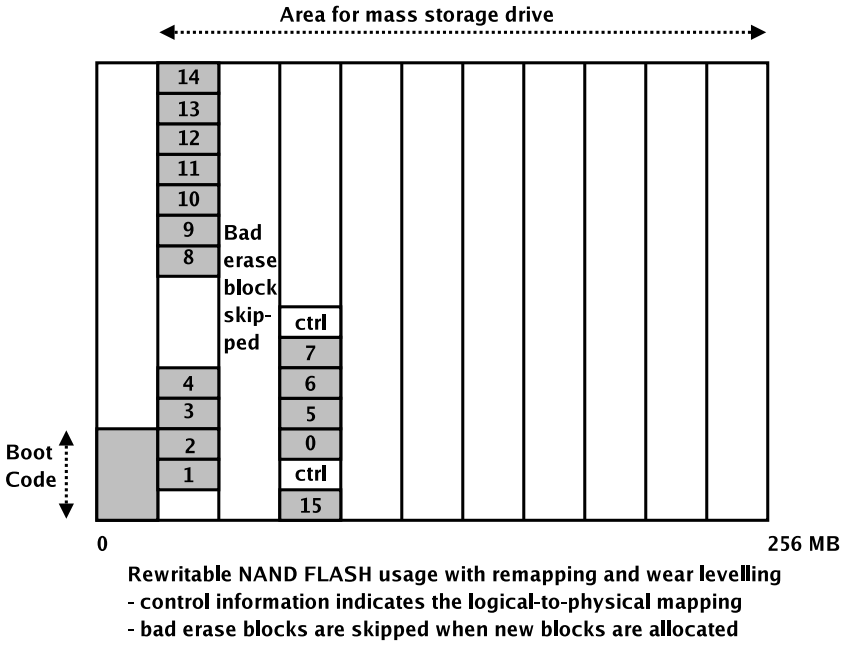

Figure 2: NAND FLASH Read-Write Operation

Normally the built-in remapping and wear-levelling algorithm is used. In this mode the NAND FLASH has the above organization: the first erase block is reserved for boot code, the rest of the FLASH holds the data for USB mass storage drive.

When logical disk sectors are written, a new version of the sector is allocated and the previous version becomes unused. A special data structure manages the book-keeping of logical-to-physical disk sector mapping, and this control information is saved in addition to the disk data to the FLASH. When allocating new pages from the FLASH bad erase blocks are skipped.

In player mode the data structure is used to find the logical-to-physical mapping, and the right data sector is quickly located.

1.3.1 Unit Programming

FLASH programming can be performed on a partly or fully assembled product. The boot code is programmed first.

The boot code can be programmed either through USB using the RAMDISK method, or through UART. Programming includes verification of the programmed boot image. Programming though UART also allows running other production tests for the device if necessary (buttons, peripherals). These tests are part of the

flash programming routine, so they do not reduce the memory available for the boot code.

The boot code can also be pre-programmed using a memory programmer tool. Because the first erase block is guaranteed to work, there are no issues with bad erase blocks.

After boot code programming the device is connected to USB and turned on. The USB disk is formatted and content copied with the normal computer tools (drag and drop). The content files can also be replaced easily later through USB.

The above can be automated by a VS1000 Linux Programmer Tool. Some code fragments are needed in the firmware (291 words with verify support) to allow reliable programming of multiple units at the same time. The copied files can also be checksummed by the Linux Programmer to verify successful content copy.

You can get the programmer package from
http://www.vlsi.fi/support_data/programmer08.zip , it includes documentation of the programmer itself. The code additions required for your firmware are included in section 3.

1.4 Mapperless NAND-FLASH Organization

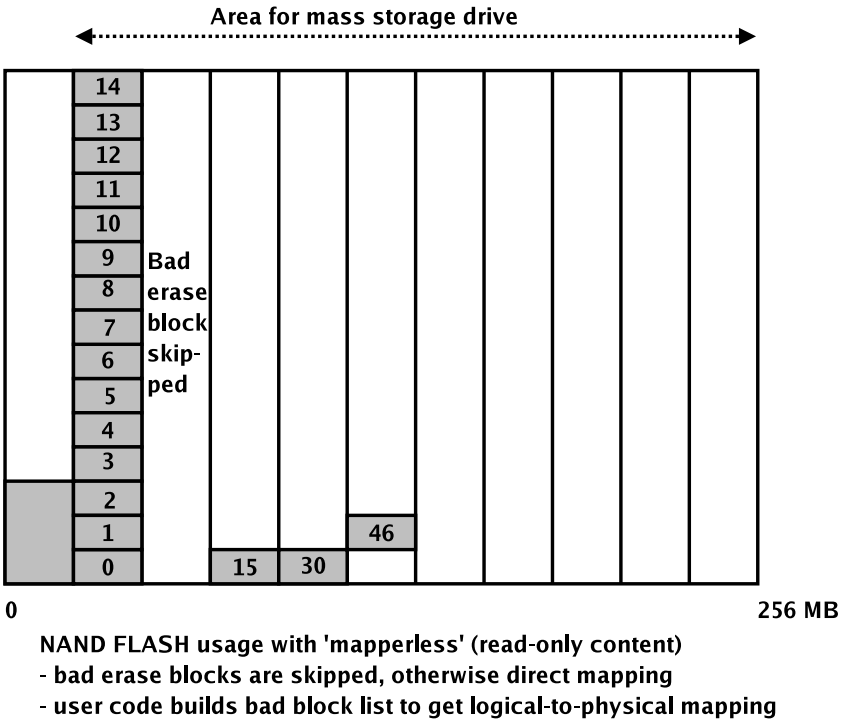

Figure 3: NAND FLASH Read-Only Operation

As an option the built-in remapping and wear-levelling algorithm can be bypassed by replacing the ReadDiskSector() function, and a different NAND-FLASH organization can be used.

In this mode the NAND FLASH has the organization shown in figure 3. Like in the rewritable case, the first erase block is reserved for boot code, the rest of the FLASH holds the data for USB mass storage drive. But in this case there is no special book-keeping data. The disk blocks are saved sequentially to all working erase blocks and bad erase blocks are left unused.

In player mode only the location of the bad erase blocks need to be known to know the logical-to-physical mapping. The bad erase blocks need to be scanned only at the startup and stored into memory.

Mapperless is read-only, so it can not be used with USB.

This organization is compatible with most NAND FLASH memory programming tools.

Pre-Programmed FLASH

The "mapperless" implementation is optimal, if the content is fixed and not changable by the end user (see also the nandfirmware package).

With "mapperless" you create a FLASH image from the firmware and the required content (imggen tool is available from VLSI's web pages). Memory programmer is then used to pre-program this image to the FLASH using the "skip bad blocks" method of the programmer.

In the firmware you need to replace the ReadDiskSector() function. During initialization you scan for the bad erase blocks, so that they can be automatically skipped by ReadDiskSector().

With "mapperless" you can't modify the content through USB. (By reprogramming the boot code you can change the unit back to rewritable mode and copy content through USB.)

The additions to your firmware needed for mapperless are included in section 4 and attached in `mapperless.c`.

2 Other Memory Types

If some other storage media than NAND FLASH is used, the whole mapper can be replaced. In this case a boot SPI EEPROM or SPI FLASH is required.

An example of the mapper replacement is the built-in RAM-DISK, which will be active if a valid NAND FLASH ident can not be found. This allows code to be loaded through USB and the code can then program the NAND FLASH or SPI memory.

SPI NOR FLASH

VS1000 Button Cell Player and VS1000 TinyPlayer are examples of systems using a SPI FLASH for both boot and audio data storage. The beginning of an SPI FLASH is reserved for boot code, and the rest is used for storage. Custom USB mass storage routine is needed because of the 4 kilobyte erasable size of the SPI FLASH.

SPI FLASH programming is often quite slow, but this allows a demo unit to be easily programmed through USB. Because there are no bad blocks in SPI FLASH chips (using NOR FLASH technology), a one-to-one copy of the resulting SPI FLASH content can be used to produce additional units.

SPI EEPROM + SD/MMC

SD/MMC players, for example VS1000+VS1003 VoIP phone, store boot code in an SPI EEPROM but use SD/MMC card for storage. Also in this case the SPI EEPROM can be either pre-programmed or programmed through USB or UART.

3 Example Code for USB Programming

These additions and patches allow more reliable and faster programming of multiple VS1000 units on the same USB bus (using USB hubs). Also, the checksumming feature allows the copied content to be verified much faster than would be possible by reading back the written data.

This code is in file `usbspeedups.c`.

```
#include <string.h>
#include <vs1000.h>
#include <audio.h>
#include <mappertiny.h>
#include <minifat.h>
#include <codec.h>
#include <vsNand.h>
#include <player.h>
#include <dev1000.h>

extern struct FsPhysical *ph;
extern struct FsMapper *map;
extern struct Codec *cod;
extern struct CodecServices cs;

#ifndef NO_USB
/* These defines enable a set of patches to allow or speed up parallel
   programming through USB using VS1000 Linux Programmer software.
   The current version is available from
   http://www.vlsi.fi/support_data/programmer08.zip .
 */
/*+22 words (+13 data words) -- use custom USB serial number */
#define USB_SERIAL_NUMBER
/*+63 words -- faster response when multiple devices in the same bus */
#define PATCH_SCSI_STATUS
/*+234 words -- 'eject' will turn off the unit, checksumming option */
#define PATCH_SCSI_COMMAND
/*-145 words -- leave out checksumming option if you need to save code space */
//#define NO_CHECKSUMMING

#ifdef USB_SERIAL_NUMBER
#include <usblowlib.h>
void MyInitUSBDescriptors(u_int16 initDescriptors) {
#define D_SIZE 26
    static u_int16 myStringDescriptor3[] = {
        (D_SIZE<<8)|03, ('2'<<8)|0, ('0'<<8)|0, ('0'<<8)|0, ('0'<<8)|0,
        ('1'<<8)|0,     ('0'<<8)|0, ('0'<<8)|0, ('0'<<8)|0,
        ('2'<<8)|0,     ('0'<<8)|0, ('0'<<8)|0, ('0'<<8)|0
    };
#undef D_SIZE
    /* Other descriptors can also be changed. */
    RealInitUSBDescriptors(initDescriptors);
    USB.descriptorTable[3] = myStringDescriptor3;
}
#endif/*USB_SERIAL_NUMBER*/

#ifdef PATCH_SCSI_STATUS
#include <usblowlib.h>
#include <scsi.h>
void MyUSBHandler(void) {
    RealUSBHandler();
    /*Patch for programming multiple devices in the same USB bus */
    if (ScsiState() == SCSI_READY_FOR_COMMAND
        && (PERIP(USB_CONFIG) & 0x7f)) {
```

```
        static int nakked = 0;
        register u_int16 stat = PERIP(USB_EP_ST0 + MSC_BULK_IN_ENDPOINT);
        register u_int16 send = PERIP(USB_EP_SEND0 + MSC_BULK_IN_ENDPOINT);
        if ( (stat & USB_STF_IN_EMPTY) ) {
            if ((stat & USB_STF_IN_NACK_SENT) &&
                send == ((MSC_BULK_IN_ENDPOINT<<10) | 13)) {
                if (++nakked > 10) {
                    /*Resend with the other DATA toggle */
                    PERIP(USB_EP_SEND0+ MSC_BULK_IN_ENDPOINT) = send | 0x8000U;
                    goto clrn;
                }
                goto clr;
            }
        } else {
        clrn:
            nakked = 0;
        clr:
            PERIP(USB_EP_ST0 + MSC_BULK_IN_ENDPOINT) =
                stat & ~USB_STF_IN_NACK_SENT;
        }
    }
}
#endif/*PATCH_SCSI_STATUS*/

#ifdef PATCH_SCSI_COMMAND
#include <usblowlib.h>
#include <scsi.h>
int startUnit = 1; /*Support eject: scsi start/stop unit.*/
u_int32 ScsiLbab(register __i0 u_int16 *res__lbab3);
auto u_int16 NewDiskProtocolCommand(register __i2 u_int16 *cmd) {
    static u_int16 checksumSave[5];
    /*Note: command length is in hi byte of cmd[0], SCSI command bytes are:
      word 0:  len scsi byte 0 (command)
      word 1:  1   2
      word 2:  3   4
      word 3:  5   6
      word 4:  7   8 */
    __y int c = (cmd[OPERATION_CODE] & 0xff);
    if (c == SCSI_START_STOP_UNIT) {
        startUnit = cmd[2]; /*both eject and start unit bits*/
#ifndef NO_CHECKSUMMING        /* Support Checksumming */
        {
            u_int32 t = (((u_int32)cmd[1] << 16) | cmd[2]);
            if ((t & 0xfc) == 0x5c) {
                memset(checksumSave, 0, sizeof(checksumSave));

                /* Note: InitFileSystem() uses minifatBuffer */
                if (InitFileSystem() == 0) {
                    if (!MscSendCsw(SCSI.Status))
                        SCSI.State = SCSI_READY_FOR_COMMAND;
                    else
                        SCSI.State = SCSI_SEND_STATUS;
                    minifatInfo.longFileName[0] = 0;
                    checksumSave[0] = t >> 8;
                    if (OpenFile(checksumSave[0]) < 0) {
                        u_int32 accu = 0;
                        register u_int32 p = 0;
                        while (p < minifatInfo.fileSize) {
                            register int l = 512;
                            ReadDiskSector(minifatBuffer, FatFindSector(p));
                            if (minifatInfo.fileSize < p + 512) {
                                l = minifatInfo.fileSize - p;
                                if (l & 1) {
                                    minifatBuffer[l/2] &= 0xff00;
                                    l++;
                                }
                            }
```

```
                            accu = CheckSumBlock(accu, minifatBuffer, l / 2);
                            *((u_int32 *)&checksumSave[3]) = accu;
                            p += 512;
                        }
                        *((u_int32 *)&checksumSave[1]) = minifatInfo.fileSize;
                    }
                    return (u_int16)-1; /* already handled */
                }
            }
        }
#endif /*!NO_CHECKSUMMING*/
    } else {
        if ((startUnit & 3) == 2) { /* previous command was EJECT MEDIA */
            /* The next command after STOP UNIT + EJECT MEDIA will not be
               serviced fully because we will turn ourselves off. */

            /* Take us 'out of the bus' */
            PERIP(SCI_STATUS) &= ~SCISTF_USB_PULLUP_ENA;
            PERIP(USB_CONFIG) = 0x8000U; /* Reset USB */
            map->Flush(map, 2); /* Flush content */
            PowerOff();
        }
        if (c == SCSI_REQUEST_SENSE) {
#ifndef NO_CHECKSUMMING            /* Support Checksumming */
            /* Read back checksum result in REQUEST_SENSE */
            if (SCSI.DataTransferLength == 63) {
                memcpy(SCSI.DataBuffer, checksumSave, sizeof(checksumSave));
                memcpyYX(SCSI.DataBuffer+5, minifatInfo.fileName, 6);
                memcpyYX(SCSI.DataBuffer+11, minifatInfo.longFileName,
                         sizeof(SCSI.DataBuffer)-11 /*21, 26 max needed..*/
                         /*sizeof(minifatInfo.longFileName)*/);
                SCSI.DataOutSize = 63;//512;
                SCSI.DataOutBuffer = SCSI.DataBuffer;
                SCSI.State = SCSI_DATA_TO_HOST;
                return (u_int16)-1;
            }
#endif /*!NO_CHECKSUMMING*/
            /* patch a bug -- sense was written to the wrong buffer */
            SCSI.DataBuffer[0] = 0x7000;
        }
    }
    return cmd[OPERATION_CODE]; /* perform the command */
}
#endif/*PATCH_SCSI_COMMAND*/

#endif/*!NO_USB*/


void main(void) {

#if 1 /*Perform some extra inits if we are started from SPI boot. */
    InitAudio(); /* goto 3.0x..4.0x */
    PERIP(INT_ENABLEL) = INTF_RX | INTF_TIM0;
    PERIP(INT_ENABLEH) = INTF_DAC;
    ph = FsPhNandCreate(0);
#endif

    /* Set the leds after nand-boot! */
    PERIP(GPIO1_ODATA) |=  LED1|LED2;      /* POWER led on */
    PERIP(GPIO1_DDR)   |=  LED1|LED2; /* SI and SO to outputs */
    PERIP(GPIO1_MODE)  &= ~(LED1|LED2); /* SI and SO to GPIO */

    if (!map) {
        map = FsMapTnCreate(ph, 0); /* tiny mapper */
    }
    PlayerVolume();
    while (1) {
```

```
#ifndef NO_USB
        if (USBIsAttached()) {
#ifdef PATCH_SCSI_STATUS
            SetHookFunction((u_int16)USBHandler, MyUSBHandler);
#endif
#ifdef PATCH_SCSI_COMMAND
            SetHookFunction((u_int16)MSCPacketFromPC,PatchDiskProtocolCommand);
#endif
#ifdef USB_SERIAL_NUMBER
            SetHookFunction((u_int16)InitUSBDescriptors, MyInitUSBDescriptors);
#endif
            MassStorage();
        }
#endif
        if (InitFileSystem() == 0) {
            minifatInfo.supportedSuffixes = supportedFiles;
            player.totalFiles = OpenFile(0xffffU);
            if (player.totalFiles == 0)
                goto noFSnorFiles;

            player.nextStep = 1;
            player.nextFile = 0;
            while (1) {
                if (player.randomOn) {
                    player.currentFile =
                        Shuffle(player.totalFiles, player.currentFile);
                } else {
                    player.currentFile = player.nextFile;
                }
                if (player.currentFile < 0)
                    player.currentFile += player.totalFiles;
                if (player.currentFile >= player.totalFiles)
                    player.currentFile -= player.totalFiles;

                if (OpenFile(player.currentFile) < 0) {
                    /* By default the next track is the next track to play */
                    player.nextFile = player.currentFile + 1;

                    /* Some inits moved from PlayCurrentFile() to here to allow
                       easier resume implementation. */
                    player.volumeOffset = -12; /* reset default volume offset*/
                    player.ffCount = 0;
                    PlayerVolume();
                    cs.cancel = 0;
                    cs.goTo = -1;
                    cs.fileSize = cs.fileLeft = minifatInfo.fileSize;
                    cs.fastForward = 1; /* reset play speed to normal */

                    PlayCurrentFile();
                } else {
                    player.nextFile = 0;
                }
#ifndef NO_USB
                if (USBIsAttached())
                    break;
#endif
            }
        } else {
            /* AudioOutputSamples does not go to low-power mode + poweroff */
        noFSnorFiles:
            LoadCheck(&cs, 32); /* decrease or increase clock */
            memset(tmpBuf, 0, sizeof(tmpBuf));
            AudioOutputSamples(tmpBuf, sizeof(tmpBuf)/2);
            /* When no samples fit, calls the user interface
               -- handle volume control and power-off */
        }
    }
}
```

4 Example Code for Mapperless

This code is in file `mapperless.c`.

```
#include <string.h>
#include <vs1000.h>
#include <audio.h>
#include <mappertiny.h>
#include <minifat.h>
#include <codec.h>
#include <vsNand.h>
#include <player.h>
#include <dev1000.h>

extern struct FsPhysical *ph;
extern struct FsMapper *map;
extern struct Codec *cod;
extern struct CodecServices cs;

#define MAPPERLESS /*+137 instruction words + MAX_BADS*2 data words. */
#ifdef MAPPERLESS
/*  Note to MAPPERLESS:
  - This option can be used when FLASH programmer is used:
    a FAT filesystem image is written by a flash programmer starting
    from the third erase block, and bad eraseblocks are skipped.
    (Use the "skip bad blocks" method of programming.)
  - This disk can not be written to, and thus USB is also disabled.
  - At startup MapperlessSetup() bad eraseblocks are scanned and
    remembered, and the normal sector read function is replaced.
  - An image creator software is available from VLSI web pages.
  - The first erase block is reserved for firmware, the second
    erase block is reserved to be used freely by the programmer.
 */
#define NO_USB /*With mapperless you can't program content through USB*/
#define MAX_BADS 256 /*Maximum number of bad erase blocks supported.*/
s_int16 numOfBads;      /* remembers the number of bad erase blocks */
u_int32 bads[MAX_BADS]; /* remembers the bad erase block start block numbers */
auto u_int16 MapperlessReadDiskSector(register __i0 u_int16 *buffer,
      register __reg_a u_int32 sector) {
    /*Leave 2 erase blocks reserved for firmware and optional settings. */
    __y u_int32 sect = sector + (ph->eraseBlockSize<<1);
    register int i = 0;
    while (i < numOfBads) { /* Find mapping between logical and physical */
        if (sect < bads[i])
            break;
        sect += ph->eraseBlockSize;
        i++;
    }    /* NOTE: currently reads without error correction code! */
    return (u_int16)ph->Read(ph, sect, 1, buffer, NULL);
}

void MapperlessSetup(void) {    /* Setup -- Scan bad erase blocks. */
    register u_int16 i;
    numOfBads = 0;
    /* reserve first two erase blocks for firmware and optional settings. */
    for (i=2; i<ph->eraseBlocks; i++) {
        /* read both spares */
        ph->Read(ph, (u_int32)i * ph->eraseBlockSize, 2, NULL, minifatBuffer);
        if ((minifatBuffer[0/2 + 2] & 0xff) != 0xff ||
            (minifatBuffer[16/2+ 2] & 0xff) != 0xff) {     /* bad */
            bads[numOfBads++] = (u_int32)i * ph->eraseBlockSize;
            if (numOfBads == MAX_BADS)
                break;
        }
    }    /* Replace normal read function -- bypass mapper. */
    SetHookFunction((u_int16)ReadDiskSector, MapperlessReadDiskSector);
}
#endif /*MAPPERLESS*/
```

```
#ifdef NO_USB
void MyUserInterfaceIdleHook(void) {
    if (uiTrigger) {
        uiTrigger = 0;
#ifdef NO_USB
        KeyScanNoUSB(0x1f); /* Does not cancel play if USB is attached.*/
#else
        KeyScan(); /* Normal key scan. */
#endif
        /* Update LEDs */
        {
            register u_int16 leds = LED1;
            if (player.pauseOn > 0 && (uiTime & 0x06) != 0)
                leds = 0; /* power led flashes when in pause mode */
            if (player.randomOn)
                leds |= LED2;
            PERIP(GPIO1_ODATA) = (PERIP(GPIO1_ODATA) & ~(LED1 | LED2)) | leds;
        }
    }
}
#endif /*NO_USB*/

void main(void) {
#if 1 /*Perform some extra inits if we are started from SPI boot. */
    InitAudio(); /* goto 3.0x..4.0x */
    PERIP(INT_ENABLEL) = INTF_RX | INTF_TIM0;
    PERIP(INT_ENABLEH) = INTF_DAC;
    ph = FsPhNandCreate(0);
#endif
    /* Set the leds after nand-boot! */
    PERIP(GPIO1_ODATA) |=  LED1|LED2;      /* POWER led on */
    PERIP(GPIO1_DDR)   |=  LED1|LED2; /* SI and SO to outputs */
    PERIP(GPIO1_MODE)  &= ~(LED1|LED2); /* SI and SO to GPIO */

#ifdef MAPPERLESS
    MapperlessSetup(); /* Scan eraseblocks, replace ReadDiskSector() */
#else
    if (!map) {
        map = FsMapTnCreate(ph, 0); /* tiny mapper */
    }
#endif/*MAPPERLESS*/

    /* Set hooks */
#ifdef NO_USB
    SetHookFunction((u_int16)IdleHook, MyUserInterfaceIdleHook);
#endif /*NO_USB*/

    PlayerVolume();
    while (1) {
#ifndef NO_USB
        if (USBIsAttached()) {
            MassStorage();
        }
#endif
        if (InitFileSystem() == 0) {
            minifatInfo.supportedSuffixes = supportedFiles;
            player.totalFiles = OpenFile(0xffffU);
            if (player.totalFiles == 0)
                goto noFSnorFiles;

            player.nextStep = 1;
            player.nextFile = 0;
```

```
            while (1) {
                if (player.randomOn) {
                    player.currentFile = /* Shuffle is from dev1000.h */
                        Shuffle(player.totalFiles, player.currentFile);
                } else {
                    player.currentFile = player.nextFile;
                }
                if (player.currentFile < 0)
                    player.currentFile += player.totalFiles;
                if (player.currentFile >= player.totalFiles)
                    player.currentFile -= player.totalFiles;

                if (OpenFile(player.currentFile) < 0) {
                    /* By default the next track is the next track to play */
                    player.nextFile = player.currentFile + 1;

                    /* Some inits moved from PlayCurrentFile() to here to allow
                       easier resume implementation (cs.goTo set to value at start). */
                    //player.pauseOn = 0; /* moved even earlier */
                    player.volumeOffset = -12; /* reset default volume offset */
                    player.ffCount = 0;
                    PlayerVolume();
                    cs.cancel = 0;
                    cs.goTo = -1;
                    cs.fileSize = cs.fileLeft = minifatInfo.fileSize;
                    cs.fastForward = 1; /* reset play speed to normal */

                    PlayCurrentFile();
                } else {
                    player.nextFile = 0;
                }
#ifndef NO_USB
                if (USBIsAttached())
                    break;
#endif
            }
        } else {
            /* AudioOutputSamples does not go to low-power mode + poweroff */
        noFSnorFiles:
            LoadCheck(&cs, 32); /* decrease or increase clock */
            memset(tmpBuf, 0, sizeof(tmpBuf));
            AudioOutputSamples(tmpBuf, sizeof(tmpBuf)/2);
            /* When no samples fit, calls the user interface
               -- handle volume control and power-off */
        }
    }
}
```

5 Document Version Changes

This chapter describes the most important changes to this document.

Version 0.50, 2010-01-27

- Initial version.

6 Contact Information

VLSI Solution Oy
Hermiankatu 8 G
FIN-33720 Tampere
FINLAND

Fax: +358-3-316 5220
Phone: +358-3-316 5230
Email: support@vlsi.fi
URL: http://www.vlsi.fi/