AVR32136: AVR32 UC3 NAND flash GPIO driver

32-bit AVR®
Microcontrollers

Application Note

Features

- Open NAND flash interface (ONFi).
- Fully configurable GPIO and timing settings.
- Uses the CPU local bus for high speed I/O access.
- Supports both 8-bit and 16-bit NAND flash devices.
- Supports a general NAND flash command interface.
- Optional Hamming ECC algorithm implemented in software.

1 Introduction

This application note describes how to connect a NAND flash device to an AVR®32 UC3 family device and communicate using only GPIO lines.

How to configure the driver to match the target hardware is also covered by this application note.

Most new NAND flash devices share the same electrical interface, separated between 8-bit and 16-bit devices. The command interface (open NAND flash interface) is the same for both 8-bit and 16-bit devices, which makes it feasible to have a generic driver for all NAND flash devices.

The optional Hamming ECC algorithm implemented in software is a part of the Software Framework and can be enabled seamless with the generic NAND flash interface described in this application note.

Rev. 32110A-AVR32-12/08

2 NAND flash interface

2.1 Open NAND flash interface

This example application and driver are using the open NAND flash interface (ONFi) to communicate with the NAND flash device from the UC3 microcontroller.

For more information about ONFi see the website http://www.onfi.org/

2.2 NAND flash command interface

The driver implemented for UC3 devices does not support the full command set, but the most vital part to identify, read, erase and program the flash device. See Table 2-1 for a list of supported commands.

Table 2-1. NAND flash command interface supported by driver

Name	Command	Supported by driver
Read	0x00 0x30	Yes
Copyback read	0x00 0x35	
Change read column	0x05 0xE0	Yes
Read cache enhanced	0x00 0x31	
Read cache	0x31	
Read cache end	0x3F	
Block erase	0x60 0xD0	Yes
Interleaved	0xD1	
Read status	0x70	Yes
Read status enhanced	0x78	
Page program	0x80 0x10	Yes
Interleaved	0x11	
Page cache program	0x80 0x15	
Copyback program	0x85 0x10	
Interleaved	0x85 0x11	
Change write column	0x85	
Read id	0x90	Yes
Read parameter page	0xEC	Yes
Read unique ID	0xED	
Get features	0xEE	
Set features	0xEF	
Reset	0xff	Yes

These NAND flash commands are used through the driver interface. See chapter 4 on page 5 for more details.

3 Electrical wiring

The open NAND flash interface (ONFi) also defines the hardware interface, packages and pin out. This makes it easy to swap out NAND flash devices to another size and/or brand.

This chapter will describe how the NAND flash interface should be connected to the UC3 microcontroller.

3.1 Command signals

3.1.1 Command latch enable (CLE)

The command latch enable signal (CLE) is active high and used to tell the NAND flash device that the data on the I/O lines are a command which should be latched into the command register. This signal is used in combination with the write enable (WE) signal.

3.1.2 Address latch enable (ALE)

The address latch enable signal (ALE) is active high and used to tell the NAND flash device that the data on the I/O lines are an address which should be latched into the address register. This signal is used in combination with the write enable (WE) signal.

3.1.3 Chip enable (CE)

The chip enable signal (CE) is used to select and enable the device, and is active low. This enables the design to have multiple NAND flash devices, each connected to a separate chip enable signal.

3.1.4 Read enable (RE)

The read enable signal (RE) controls the reading of data from the NAND flash device, and is active low. The internal address register is incremented by one for each read. The auto increased address enables the possibility to fast and easy transfer data from the NAND flash device.

The NAND flash device will need a read page command, or other read commands, before data transfer is activated. Most NAND flash devices starts up in read mode on the first page of data, which can be used to easy read out a bootloader.

3.1.5 Write enable (WE)

The write enable signal (WE) controls writing commands and data to the NAND flash device, and is active low. When writing data the internal address register is incremented by one for each write. The auto increased address enables the possibility to fast and easy transfer data to the NAND flash device.

NAND flash also uses the write enable signal when writing commands and addresses to the device.

3.1.6 Write protect (WP)

The write protect signal (WP) is used to enable and disable write and erase protection for the entire device, and is active low.

3.1.7 Read/busy output (R/B)

The ready/busy signal (R/B) is used to tell if the NAND flash device is ready for new tasks or busy doing a task, the signal is busy low and ready high. A task can be read, erase, program, etc.

3.2 I/O signals

The I/O signals are used to transfer data to the NAND flash device. The interface is either 8-bit or 16-bit and is a bidirectional serial bus. It is called serial due to the fact that addresses are sent to the NAND flash device in one byte at a time.

The signals must be routed to a single GPIO port on the AVR32 UC3 device, with all the signals in a successive order.

The lowest numbered GPIO line on the I/O lines chosen on the AVR32 UC3 device must be routed to I/O 0 on the NAND flash device. The other I/O lines must be routed to the following GPIO lines on that port in successive order.

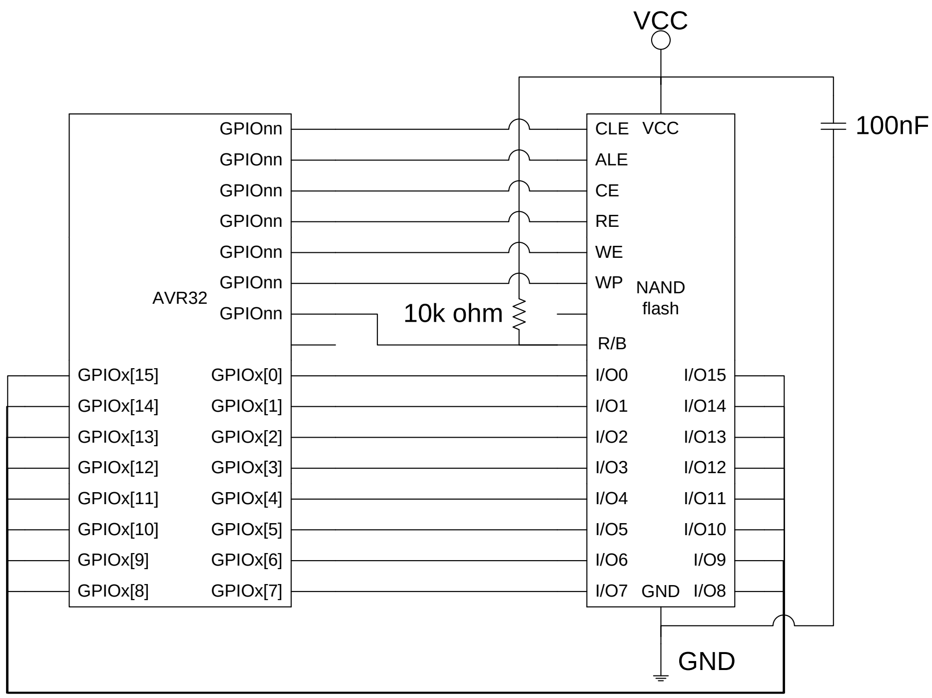
For an example of how to connect a NAND flash device see Table 3-1.

Table 3-1. GPIO lines connected to an 8-bit NAND flash I/O lines example

UC3 GPIO port	UC3 GPIO line	NAND flash I/O line	8-bit interface	16-bit interface
Port B	0	I/O 0	X	X
Port B	1	I/O 1	X	X
Port B	2	I/O 2	X	X
Port B	3	I/O 3	X	X
Port B	4	I/O 4	X	X
Port B	5	I/O 5	X	X
Port B	6	I/O 6	X	X
Port B	7	I/O 7	X	X
Port B	8	I/O 8		X
Port B	9	I/O 9		X
Port B	10	I/O 10		X
Port B	11	I/O 11		X
Port B	12	I/O 12		X
Port B	13	I/O 13		X
Port B	14	I/O 14		X
Port B	15	I/O 15		X

3.3 Block diagram of wiring

Figure 3-1. Example wiring of 16-bit NAND flash connected to an AVR32 MCU



On 8-bit NAND flash devices the upper byte of the I/O lines are not present. Hence it is neither needed nor possible to route these signals to the AVR32 device. When interfacing 8-bit NAND flash only eight GPIO lines are needed on the AVR32 device.

4 NAND flash GPIO driver interface

4.1 Setup the struct nand_driver_data

The driver needs a struct which contains the entire configuration for the NAND flash driver. This struct is called *nand_driver_data* and is defined in the *nand.h* file.

4.1.1 nand_info

This *struct nand_info* will be filled by the driver when initializing, must be left unaltered.

4.1.2 bad_table

This *struct nand_bad_table* must be initialized after the driver has been initialized, for more information see chapter 4.3 on page 7.

4.1.3 gpio_ce

This integer representing the NAND flash chip enable signal (CE) must be set to a GPIO line number.

4.1.4 gpio_rb

This integer representing the NAND flash read/busy signal (R/B) must be set to a GPIO line number.

4.1.5 gpio_we

This integer representing the NAND flash write enable signal (WE) must be set to a GPIO line number.

4.1.6 gpio_wp

This integer representing the NAND flash write protect signal (WP) can be set to a GPIO line number. If set to a negative value, the write protect feature will be assumed controlled by external pull-up.

4.1.7 gpio_ale

This integer representing the NAND flash address latch enable signal (ALE) must be set to a GPIO line number.

4.1.8 gpio_cle

This integer representing the NAND flash command latch enable signal (CLE) must be set to a GPIO line number.

4.1.9 gpio_io_port

This integer representing which port on the CPU local bus the I/O lines to the NAND flash are connected to.

4.1.10 gpio_io_mask

This bit field represents the mask the I/O lines represent on the CPU local bus. I.e. for an 8-bit interface with offset 8 on the CPU local bus this mask would be 0x0000FF00.

4.1.11 gpio_io_adress

This address must be set to the base address on the CPU local bus for the given GPIO port which is used for I/O.

This can be automatically given with the following C code:

```
nfd.gpio_io_address = (unsigned long *)
    (AVR32_GPIO_LOCAL_ADDRESS + (0x100 * nfd.gpio_io_port));
```

The variable *nand_data* is the *struct nand_driver_data*.

4.1.12 gpio_io_offset

This integer is the offset from GPIO line 0 on the GPIO port used for the I/O interface. I.e. using bit 8 to bit 15 on a GPIO port would give an offset 8.

4.1.13 gpio_io_size

This integer represents the number of GPIO lines used for the I/O interface. It must be identical to the bus width for the NAND flash device, i.e. 8 for 8-bit devices and 16 for 16-bit devices.

4.2 Initialize the GPIO lines and read out the NAND flash ID

After all vital information has been filled into the *struct nand_driver_data*, the GPIO lines and NAND flash device is ready to be initialized. This is done with the function:

```
nand_init(struct nand_driver_data *nfd);
```

The function will setup all the GPIO lines, reset the NAND flash device and read out the NAND flash ID.

4.3 Initialize the bad block table and scan for bad blocks

Since NAND flash is by nature not 100 % error free the NAND flash driver needs to have a table of which blocks are bad. Since there can be different number of blocks in NAND flash devices, the size of the bad block table must be calculated after the NAND flash ID has been read.

It is possible to get the number of blocks from the driver by looking in the *nand_info struct* within the *nand_driver_data struct* after initialization. There is an integer *num_blocks* which holds the information about number of blocks.

The *bad_table struct* within the *nand_driver_data struct* must be initialized with a memory area large enough to hold this information.

This can be done with the following C code:

```
unsigned char *block_status;

block_status = malloc(nfd.nand_info.num_blocks);
if (!block_status)
    return;
nfd.bad_table.block_status = block_status;
```

The bad block table can now be generated with the function:

```
nand_create_badblocks_table(struct nand_driver_data *nfd);
```

This function will examine the entire flash, locate the bad blocks and make a note of where they are in the bad blocks table. This will protect the user later when erasing, programming and reading data from the NAND flash device.

The user should check the return value from *nand_create_badblocks_table()* for errors.

4.4 Erase a NAND flash block

The driver can erase the entire content of a given NAND flash block; this will effectively set the value of all bytes in that block to 0xFF. This will not erase the spare area in the block. To erase a block use the following function:

```
nand_erase(struct nand_driver_data *nfd,
           const unsigned long block);
```

Example:

```
nand_erase(&nfd, 42);
```

The example above will erase block 42 in the NAND flash.

The user should check the return value from *nand_erase()* for errors.

4.5 Program a NAND flash page

The driver can program the contents of a given page within a block. The pages must usually be programmed from offset 0 and upwards. This function will not program the spare area in the block. To program data to a page use the following function:

```
int nand_write(struct nand_driver_data *nfd,
        const unsigned long block, const unsigned long offset,
        const unsigned char *buf, const unsigned short count);
```

Example:

```
unsigned char buffer[5];
buffer[0] = 'A';
buffer[1] = 'V';
buffer[2] = 'R';
buffer[3] = '3';
buffer[4] = '2';
nand_write(&nfd, 42, 0, buffer, 5);
```

The example above will program the 5 byte large array into offset 0 of block 42 in the NAND flash.

The user should check the return value from *nand_write()* for errors.

4.6 Read a NAND flash page

The driver can read the contents of a given page within a block. This function will not read the spare area in the block. To read data from a page use the following function:

```
int nand_read(struct nand_driver_data *nfd,
       const unsigned long block, const unsigned long offset,
       unsigned char *buf, const unsigned short count);
```

Example:

```
unsigned char buffer[5];
nand_read(&nfd, 42, 0, buffer, 5);
```

The example above will read 5 bytes from offset 0 of block 42 in the NAND flash and store it into the buffer array.

The user should check the return value from *nand_read()* for errors.

5 Configuration of the NAND flash driver

To configure the NAND flash driver the *nand.h* header file must be altered to match the system configuration.

The following defines must be set correctly before use:

- Define FCPU to correct CPU speed.
- Set NAND_BUS_TYPE to NAND_BUS_TYPE_GPIO to use GPIO interface.
- Set NAND_ECC_TYPE to NAND_ECC_NONE for no ECC algorithm.

5.1 Enabling Hamming ECC algorithm

To enable this interface the NAND_ECC_TYPE must be set to NAND_ECC_SW in the *nand.h* header file for the software project.

After changing the value of NAND_ECC_TYPE the *nand_write()* and *nand_read()* functions will automatically start to take advantage of the Hamming ECC algorithm. It is therefore vital that return values from these functions are handled in the software project.

Also note that all previous writes to the NAND flash without ECC will be invalid after the ECC algorithm has been turned on.

6 Implementations

6.1 NAND flash driver files

The driver consists of four files; *nand.c*, *nand.h*, *nand_gpio.c* and *nand_gpio.h*. Where *nand.h* and *nand.c* contains a generic NAND flash interface and *nand_gpio.h* declares all functions and *nand_gpio.c* contains the source code specific for using GPIO to interface NAND flash.

In the software framework the NAND flash GPIO driver is located in COMPONENTS/MEMORY/NAND_FLASH/NAND_FLASH_GPIO and an example is located in the EXAMPLE sub directory.

6.2 Hamming ECC algorithm files

The algorithm consists of three files; *ecc.h*, *ecc-sw.c* and *ecc-sw.h*. Where *ecc.h* contains defines common for the ECC algorithm and ecc-sw.c and ecc-sw.h is the implementation of the Hamming algorithm.

In the software framework the Hamming ECC algorithm is located in SERVICES/MEMORY/ECC_HAMMING.

6.3 Doxygen documentation

All source code is prepared for doxygen automatic documentation generation.

Doxygen is a tool for generating documentation from source code by analyzing the source code and using known keywords. For more details see http://www.stack.nl/~dimitri/doxygen/.

Headquarters

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: 1(408) 441-0311
Fax: 1(408) 487-2600

International

Atmel Asia
Unit 1-5 & 16, 19/F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
Hong Kong
Tel: (852) 2245-6100
Fax: (852) 2722-1369

Atmel Europe
Le Krebs
8, Rue Jean-Pierre Timbaud
BP 309
78054 Saint-Quentin-en-Yvelines Cedex
France
Tel: (33) 1-30-60-70-00
Fax: (33) 1-30-60-71-11

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
Japan
Tel: (81) 3-3523-3551
Fax: (81) 3-3523-7581

Product Contact

Web Site
www.atmel.com

Technical Support
avr32@atmel.com

Sales Contact
www.atmel.com/contacts

Literature Request
www.atmel.com/literature