PC20x and PC302 NAND interfacing Applications Note 42.

Date: 17-Dec-08
Version: V1.0
Status: Release
Type: Application Note

picoChip Designs Ltd
Riverside Buildings
108 Walcot Street
Bath BA1 5BG
UK
www.picochip.com

1 Introduction.

The purpose of this document is to introduce NAND flash memory technology, describing the hardware and software requirements when interfacing to NAND Flash to the PC202 and PC302 family of WCDMA Base band processors.

1.1 Basic NAND operation.

The NAND Flash array is organized in a series of blocks which are divided in several pages. This is shown in Section 3. Data is stored either in byte (8 bits) or half-word (16 bits) format depending on the device type. Each page is constituted of a main storage area and a spare area (physically similar) typically used for data error identification and correction, wear leveling etc...

One native property of NAND Flash devices is that they may contain a percentage of invalid blocks in the memory array. Before delivering the chip, these blocks are identified and marked as “Invalid Blocks” in the first or second page of each block. The existence of bad blocks does not affect the good memory blocks because each block is independent and individually isolated from the bit lines by block select transistors.

Because NAND Flash devices have a finite lifetime (approximately 100 000 write/erase cycles), additional invalid blocks may develop while being used. Storing data requires bad-block management and data error identification and correction. Refer to Section 2. picoChip PC20x and PC302 devices do not have native hardware support for these requirements so the user is required to implement these within software.

NAND Flash operations are fully controlled through a multiplexed I/O interface and additional control signals. Commands, addresses and data are transferred through the external input/output data bus (8-bit) to the dedicated internal registers. For 16-bit devices used with the PC20x, commands, addresses and data use the lower 8 bits (7 - 0), the upper 8 bits are only used during data-transfer cycles.

Read and program operations are performed on a per page basis whereas erase operations are performed on a block basis. To read or write from NAND Flash, a command sequence is issued to select a block and a page. After this selection, the entire page can be read or written. The command sequence normally consists of a Command Latch Cycle, an Address Latch Cycle and a Data Cycle — either read or write.

The waveforms shown in Figure 1 depict the successive accesses: Command Latch, Address Latch and Data Output. Notice that no command can be sent to the NAND Flash during tR due to it’s busy-state period.

1.2 PC20x and PC302 External Bus interface (EBI)

The PC20x and PC302 processor family features the external bus interface (EBI) to enable the interfacing to external Flash devices via integrated logic circuitry. The EBI provides Native support for NOR flash devices.

When interfacing to NAND flash technology there need to be additional software and control signals generated by the ARM board support package typically driving GPIO signals. In addition to the Invalid Block Definition, Identification and Error Checking and Correction required by all NAND type Flash devices with greater than 2% invalid Blocks.

2 Invalid Block Management and Error Corrected Code

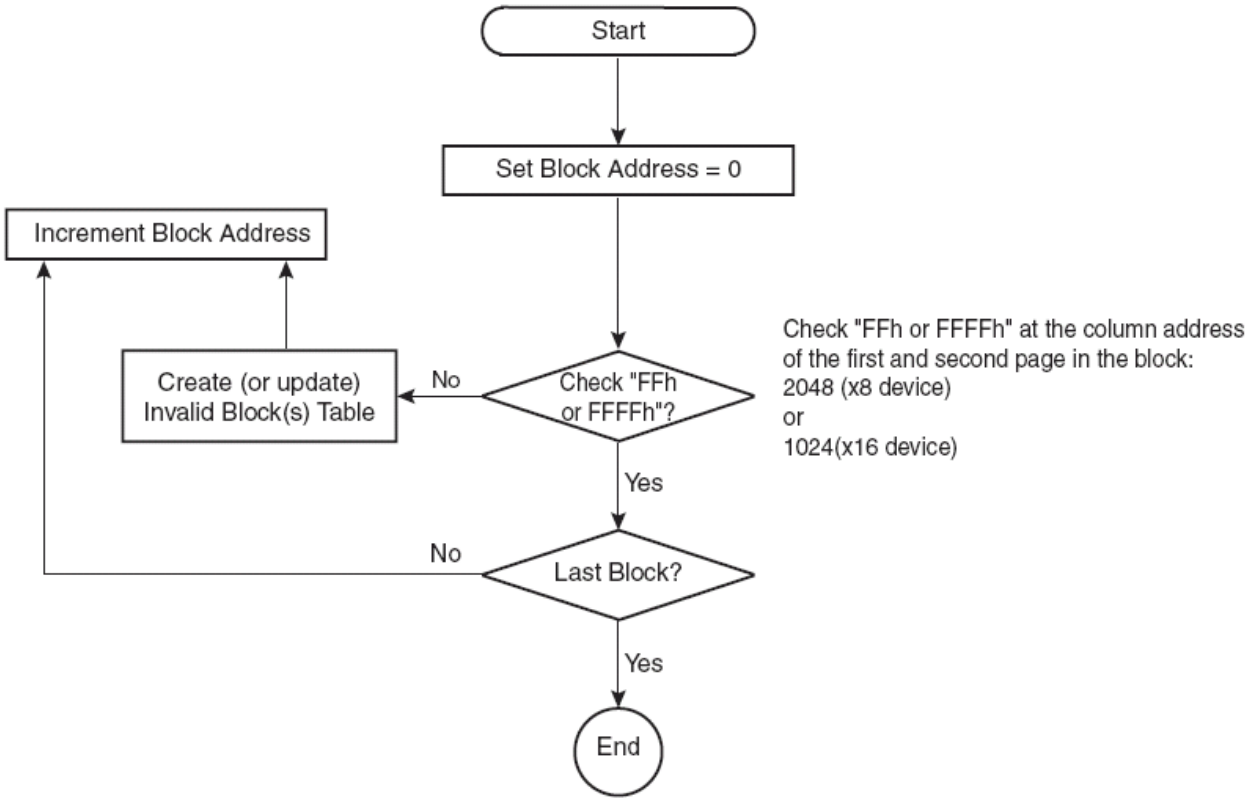
2.1 Invalid Block Definition

NAND flash devices contain a certain percentage of invalid blocks at the end of the production process. Invalid blocks are defined as blocks that contain one or more invalid bits.

2.2 Invalid Block Identification

Before shipping, every NAND flash device is tested with specific test patterns under different voltage and temperature conditions in order to identify memory locations containing errors. When errors are detected, the block to which the invalid memory location belongs is marked as an “Invalid Block”. All device locations are erased (FFh for 8-bit devices, FFFFh for 16-bit devices) except locations where the invalid block information is written. The invalid block status is defined by the first byte (8-bit devices) or first half word (16-bit devices) in the spare area. Manufacturers make sure that either the first or second page of every invalid block has non-FFh (8-bit devices) or non-FFFFh (16-bit devices) data at the column address of 2048 (8-bit devices) or 1024 (16-bit devices). Since invalid block information (located in the spare area) written by the manufacturer is not write/erase protected, it can be lost and will be almost impossible to recover. In order to prevent loosing this information, it is highly recommended to proceed to a block status mapping before any write or erase operation. Figure 1Shows how bad block identification can be done by software.

Figure 1: Bad Block Identification Flow Chart.



2.3 Error Checking and Correction

NAND devices are subject to data failures that occur during device operation. To ensure data read/write integrity, system error checking and correction (ECC) algorithms should be implemented. picoChip's Family of Processors do not provide ECC hardware support. ECC is to be implemented by the end user within there application code.

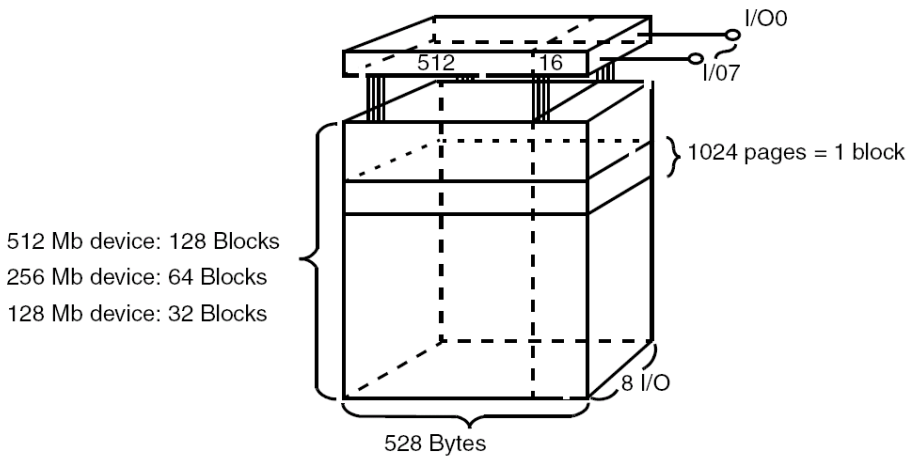
Micron ECC.c Example Code is attached for reference below.

3 NAND Flash Array Architecture

3.1 NAND Flash Example

The Spansion s30ML-p ORNAND™ Flash Family is used as an example however the implementation can be paired with any standard NAND flash devices that meet the timing specifications detailed within this application note. Figure 2 and Figure 3 show how the Spansion memory Arrays are organized for 8 bit and 16 bit devices respectively, Program operations work on page units whilst Erase operations are performed on block unites.

Figure 2: NAND x8 Array Organisation



X8 Array, Page and Block Description.

A page is made up of 528 Bytes in which 512 Bytes are used for main memory storage and 16 Bytes are reserved for redundancy or other usage. Various memory sizes are made up of a greater number of Block sizes.

1 Page = 528 Bytes

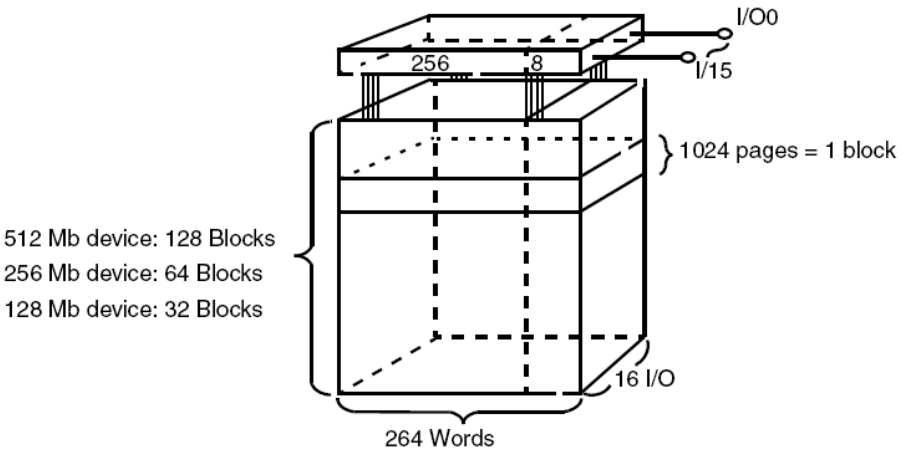
1 Block = 528 x 1024 Pages

128Mb memory – 528 Bytes x 1024 Pages x32 Blocks

256Mb memory – 528 Bytes x 1024 Pages x62 Blocks

512Mb memory – 528 Bytes x 1024 Pages x128 Blocks

Figure 3: NAND x16 Array Organisation



X16 Array, Page and Block Description.

A page is made of 264 Words in which 256 Words are used for memory storage and 8 Words are reserved for redundancy or other usage.

1 Page = 264 Words

1 Block = 264 x 1024 Pages

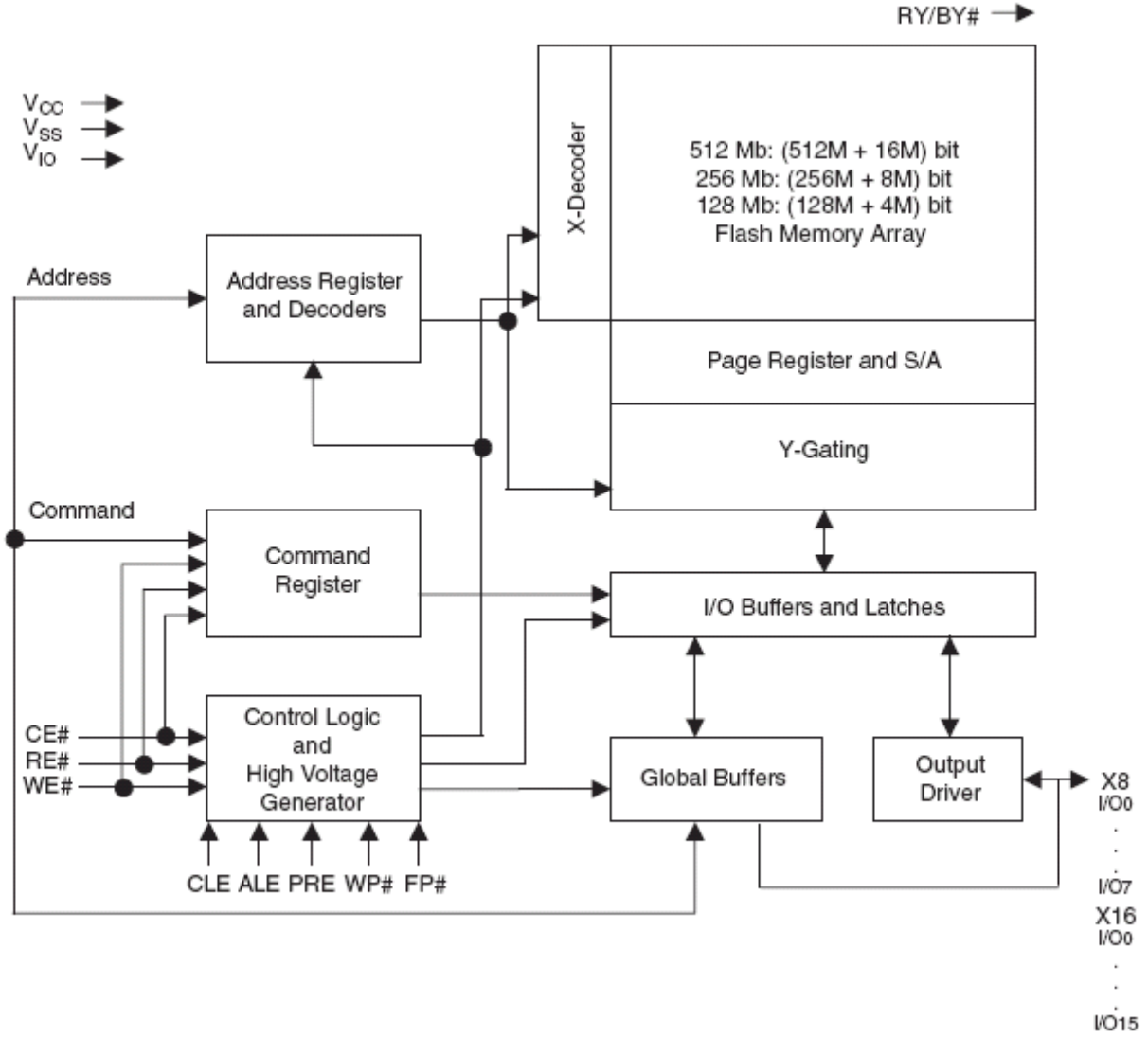
128Mb memory – 264 Words x 1024 Pages x32 Blocks

256Mb memory – 264 Words x 1024 Pages x62 Blocks

512Mb memory – 264 Words x 1024 Pages x128 Blocks

3.2 Spansion NAND Flash Block Diagram.

Figure 4: S30ML-P ORNAND™ Flash Family Block Diagram.



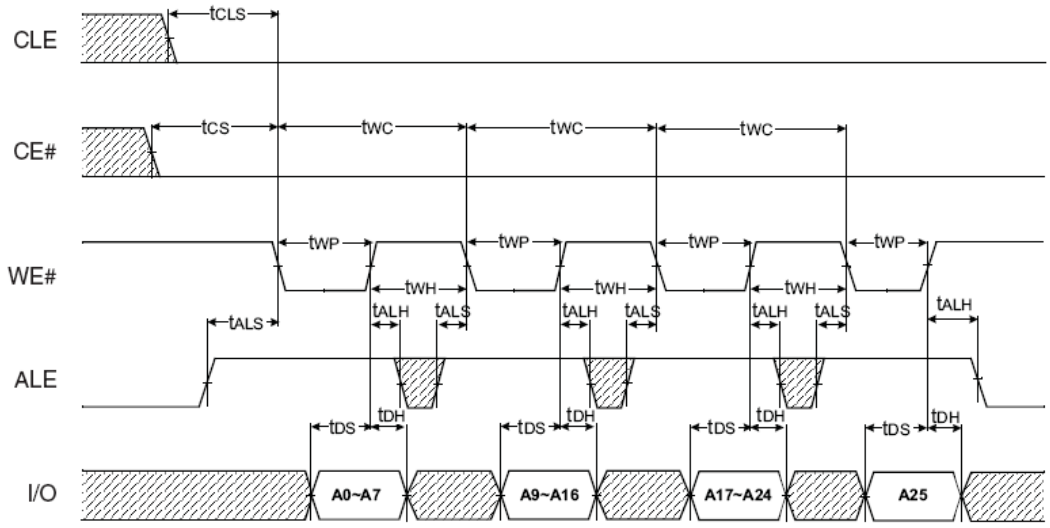
4 Comparison of NAND / EBI operations

4.1 NAND Operation

NAND Devices support various functions, all of the various operations are made up of three types of access these are the following A) Command Latch cycle, address Latch cycle and data in/out latch operation. Below details the various NAND access type that make up the various operations. (page read, block erase, status read… etc)

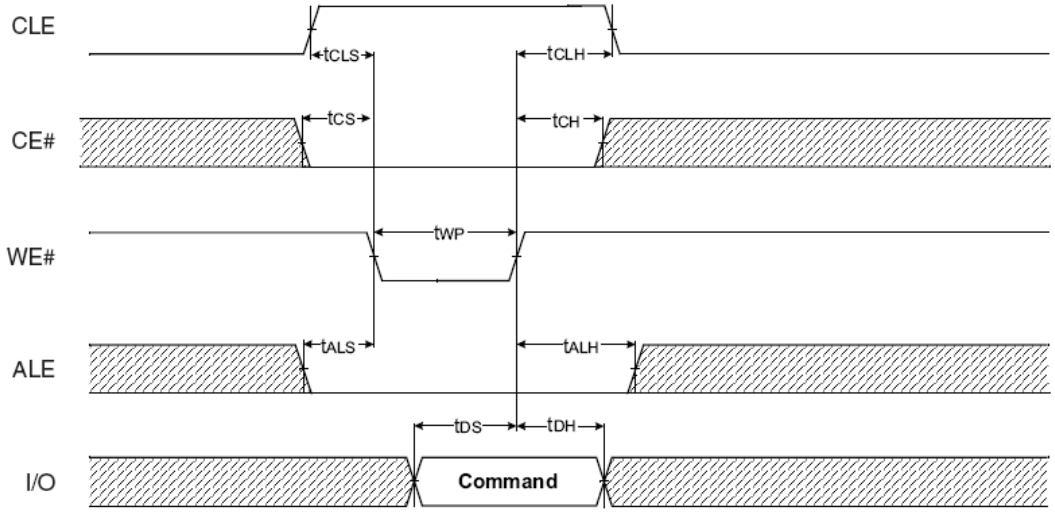
4.1.1 NAND Address Latch Cycle.

Figure 5: Address Latch Cycle.



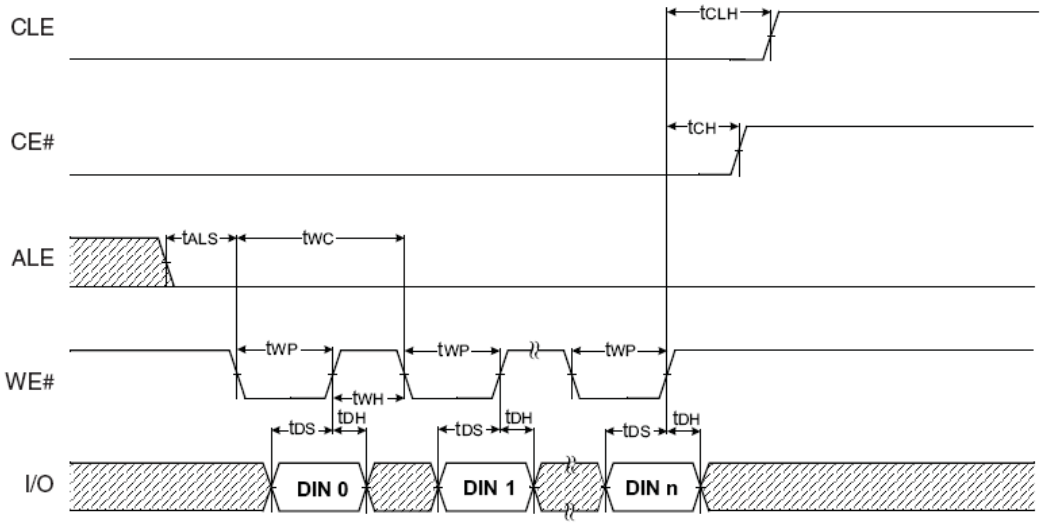
4.1.2 NAND Command Latch Cycle.

Figure 6: Command Latch Cycle.



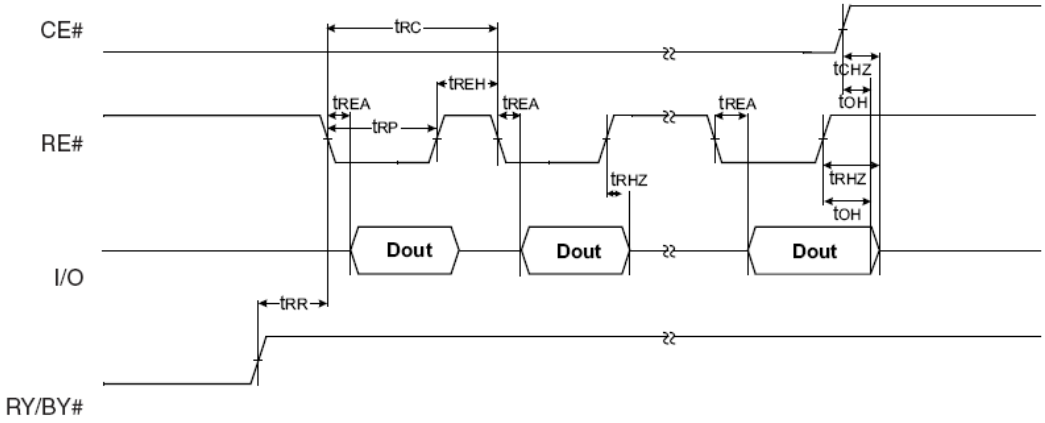
4.1.3 NAND Data input latch operation.

Figure 7: Sequential data in Timing Diagram



4.1.4 NAND Sequential Data out cycle

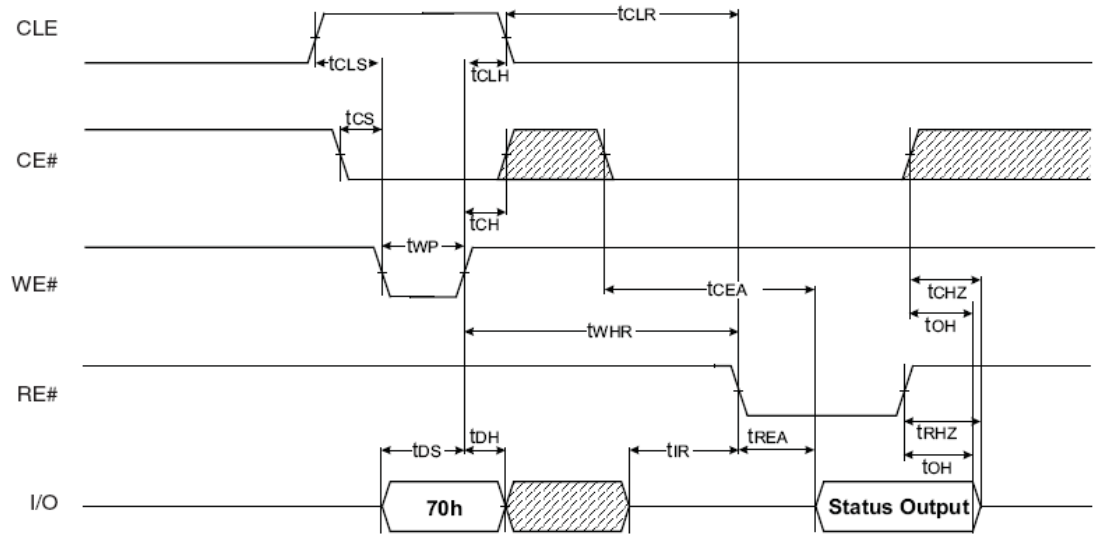
Figure 8: Sequential data out Timing Diagram



4.2 NAND operations

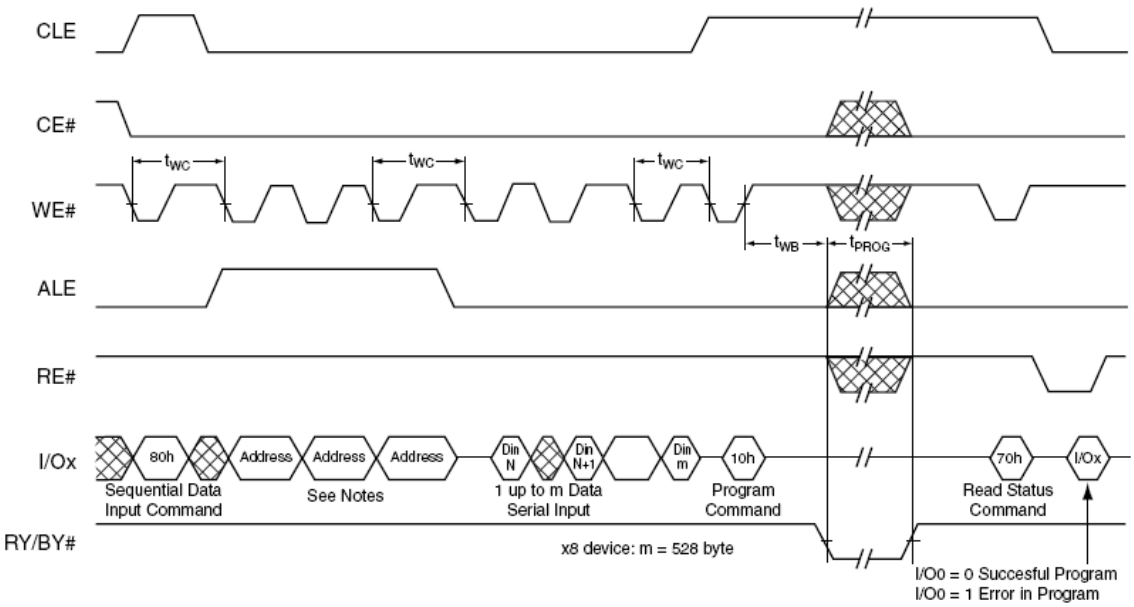
4.2.1 NAND Status Read Operation.

Figure 9: Status Read Operation Timing Diagram



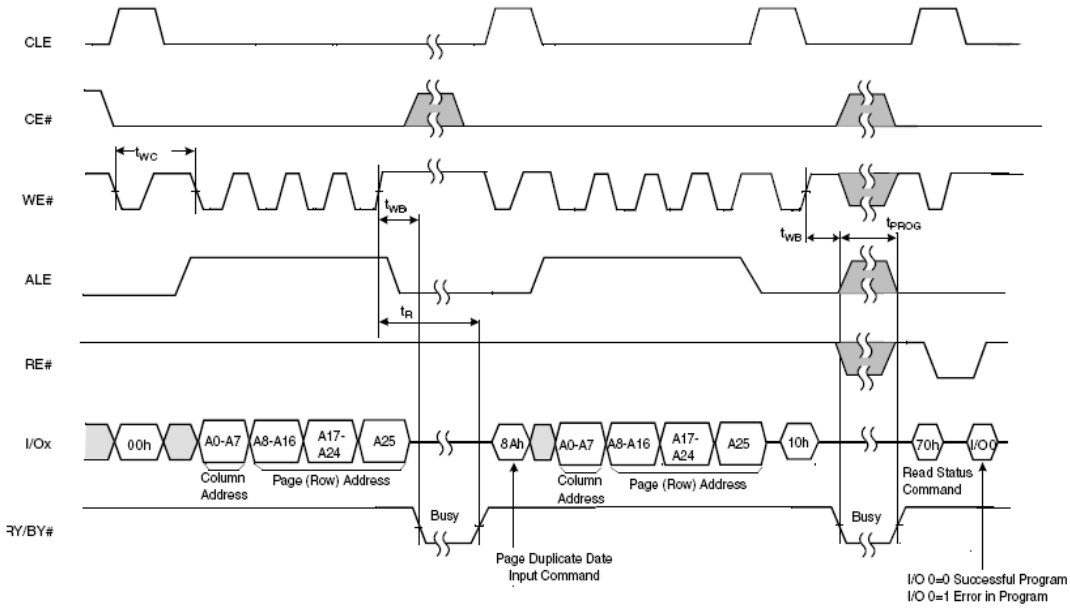
4.2.2 NAND Page Program Operation

Figure 10: Page Program operation.



4.2.3 NAND Page Duplicate Program operation.

Figure 11: Page duplicate program operation Timing Diagram

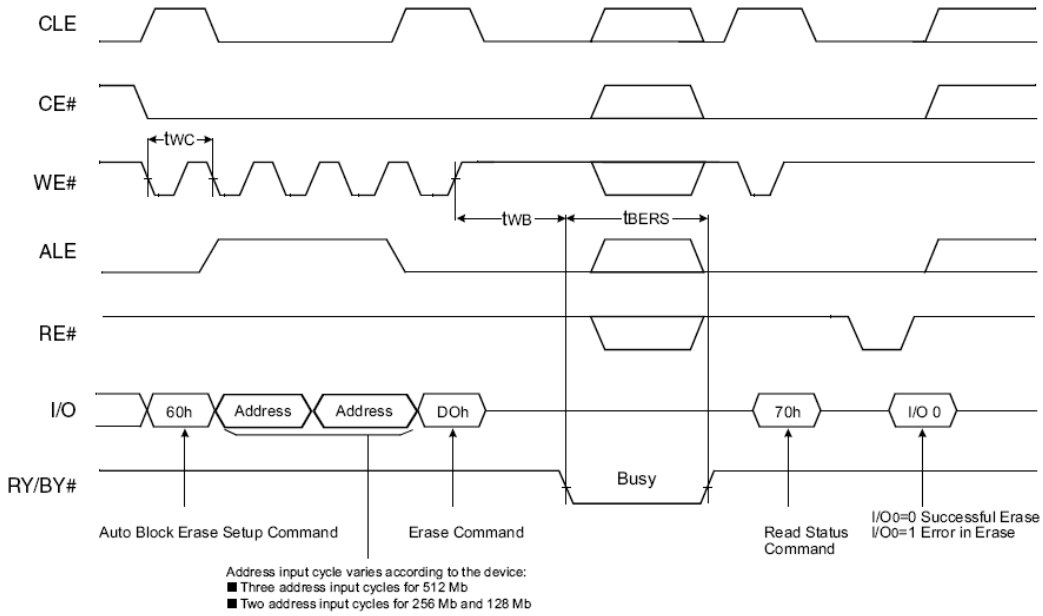


Note

1. *Address input cycle varies per device: 512 Mb devices require 4 address input cycles; 256 Mb and 128 Mb devices require 3 address input cycles.*

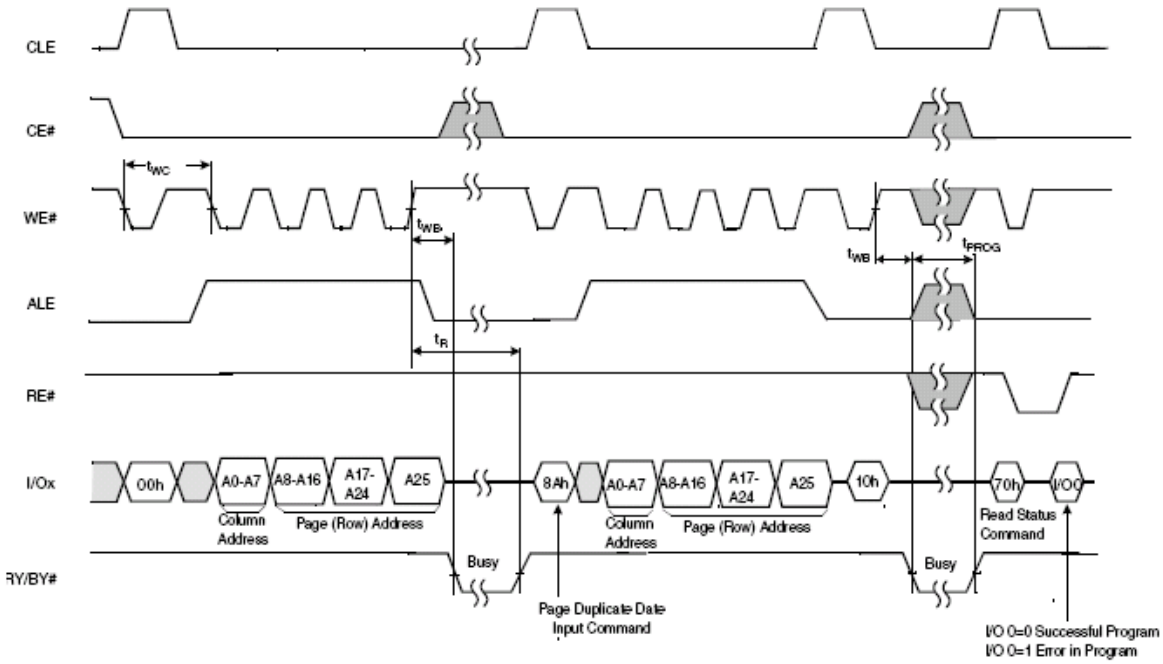
4.2.4 NAND Block Erase Operation Timing Diagram

Figure 12: Block Erase operation Timing Diagram



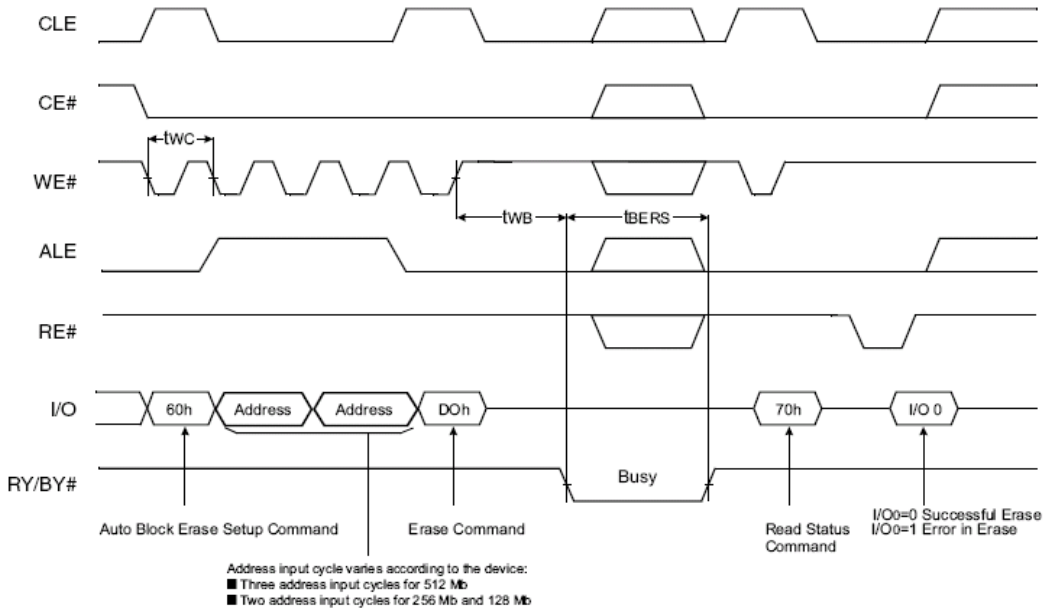
4.2.5 NAND Page Program

Figure 13: Page Program Operation Timing Diagram



4.2.6 NAND Block Erase Timing Diagram

Figure 14: Block Erase Operation Timing Diagram



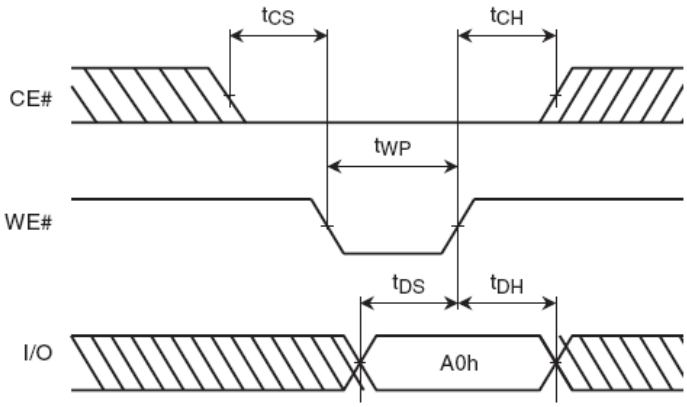
4.3 Fundamental NAND Access operations

All of the above operations need to be supported within software and will require additional GPIO management to comply with the /CS, ALE and CLE Timing requirements.

This section looks at the fundamental Read and write timing operations focusing on the EBI read and write cycles comparing these to the NAND requirements.

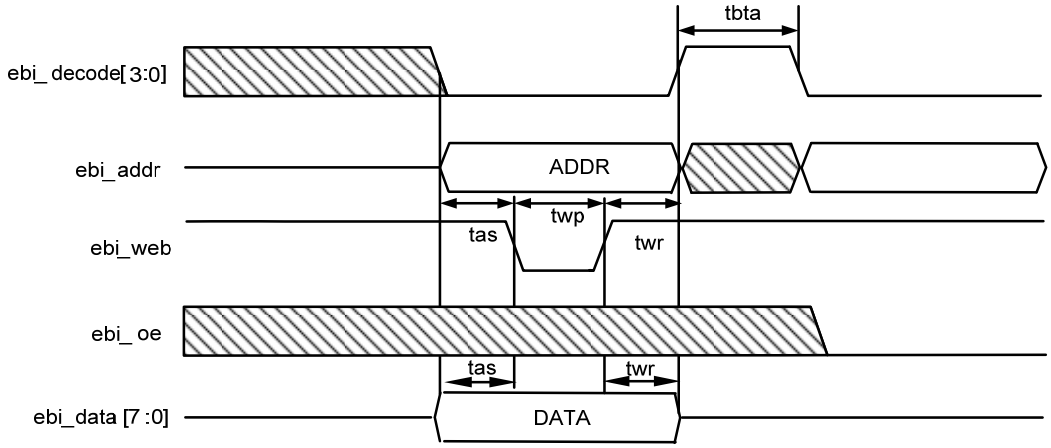
4.3.1 NAND single write Access.

Figure 15: NAND single write Timing Diagram



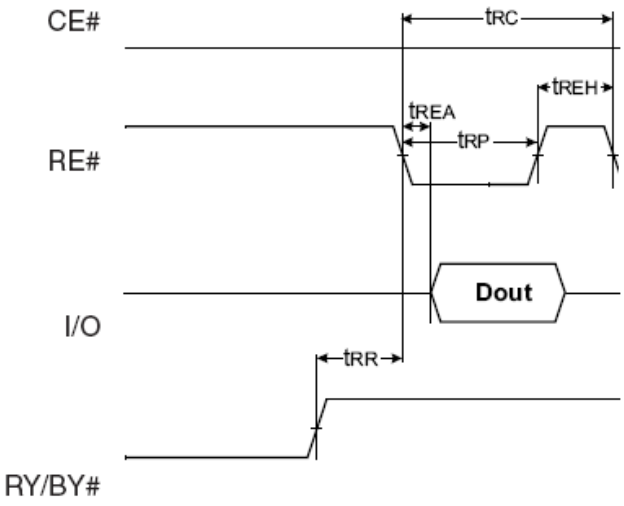
4.3.2 EBI Write Operation.

Figure 16: NAND single write Timing Diagram



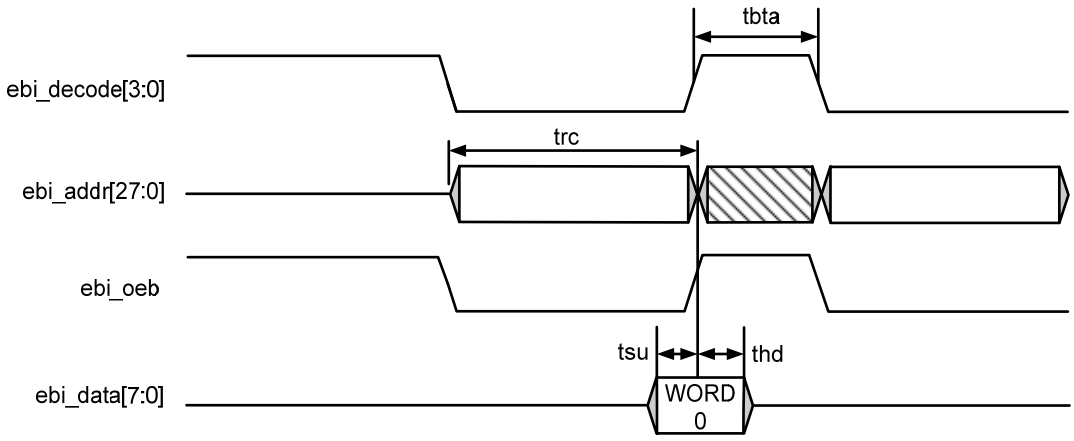
4.3.3 NAND Single Read Operation.

Figure 17: NAND single read Timing Diagram



4.3.4 EBI Read Operation.

Figure 18: EBI Read Operation Timing Diagram



4.4 Read and Write Timing parameter comparison

Table 1: NAND AC Characteristics

Spansion S30ML-PNAND Parameter	*Symbol*	*Min (ns) x8*	*Max (ns) x8*	*Min (ns) x16*	*Max (ns) x16*	*EBI Related Parameter*
Data Hold Time	t_{DH}	5				thd
CE Access Time	t_{CEA}		23		35	trc (Read) and tas+twp+twr (write) – Not Compatible Managed by S.W
RE Access Time	t_{REA}		18		30	trc
Ready to RE Low	t_{RR}	20		20		– Not Compatible Managed by S.W
	t_{CLH}	10		10		– Not Compatible Managed by S.W
ALE Hold Time	t_{ALH}	10		10		– Not Compatible Managed by S.W
CE Hold Time	t_{CH}	10		10		– Not Compatible Managed by S.W
RE High Hold Time	t_{REH}	5		15		tbta
WE Pulse Width	t_{WP}	15		25		twp
RE Pulse Width	t_{RP}	20		30		trc
Write Cycle Time	t_{WC}	30		45		tcsu + twp + tas +twr
Read Cycle Time	t_{RC}	30		50		tcsu + trc+ + tbta
						tcsu – time for Control (ALE and CLE) and /CS set up time (GPIO control configurable be software)

4.4.1 EBI Timing set Write Access

Table 2: EBI write timing AC characteristics

Parameter	*Time(ns)*	*Value At reset (EBI clks)*	*NAND Value (EBI clks)*	*Notes*
tbta	35nS	7	TBA	Bus turn around time for read/write or write/read access
twp	50nS	10	TBA	Write pulse length
twr	15nS	3	TBA	Write address/data hold
tas	15nS	3	TBA	Address set up time to write pulse driven low

4.4.2 EBI Timing set Read Access

Table 3: EBI read timing AC characteristics

Parameter	*Time(ns)*	*Value At reset (EBI clks)*	*NAND Value (EBI clks)*	*Notes*
trc	100nS	20	TBA	Read cycle time, address/select to data valid
Tbta	35nS	7	TBA	Bus turn around time for read/write or write/read access.

5 GPIO control signal requirements.

5.1 /CE ALE and CLE control signals.

To meet the specific timings detailed above as under S.W control GPIO lines are required to assist full NAND support.

Ready/Busy# (R/B) - A GPIO line is dedicated to read the Ready/Busy# (R/B) signal provided by the NAND Flash device. This signal is and input GPIO and should be monitored / interrogated prior to any given NAND access.

Chip Select /CS – There are two types of flash requirements, some flah devices do not care about the /CS signal during read accesses, other flash devices require that the /CS remains low during all access types and for the entire access period. In the case when interfacing to a Standard NAND Flash, a GPIO line is dedicated to drive the CE signal.

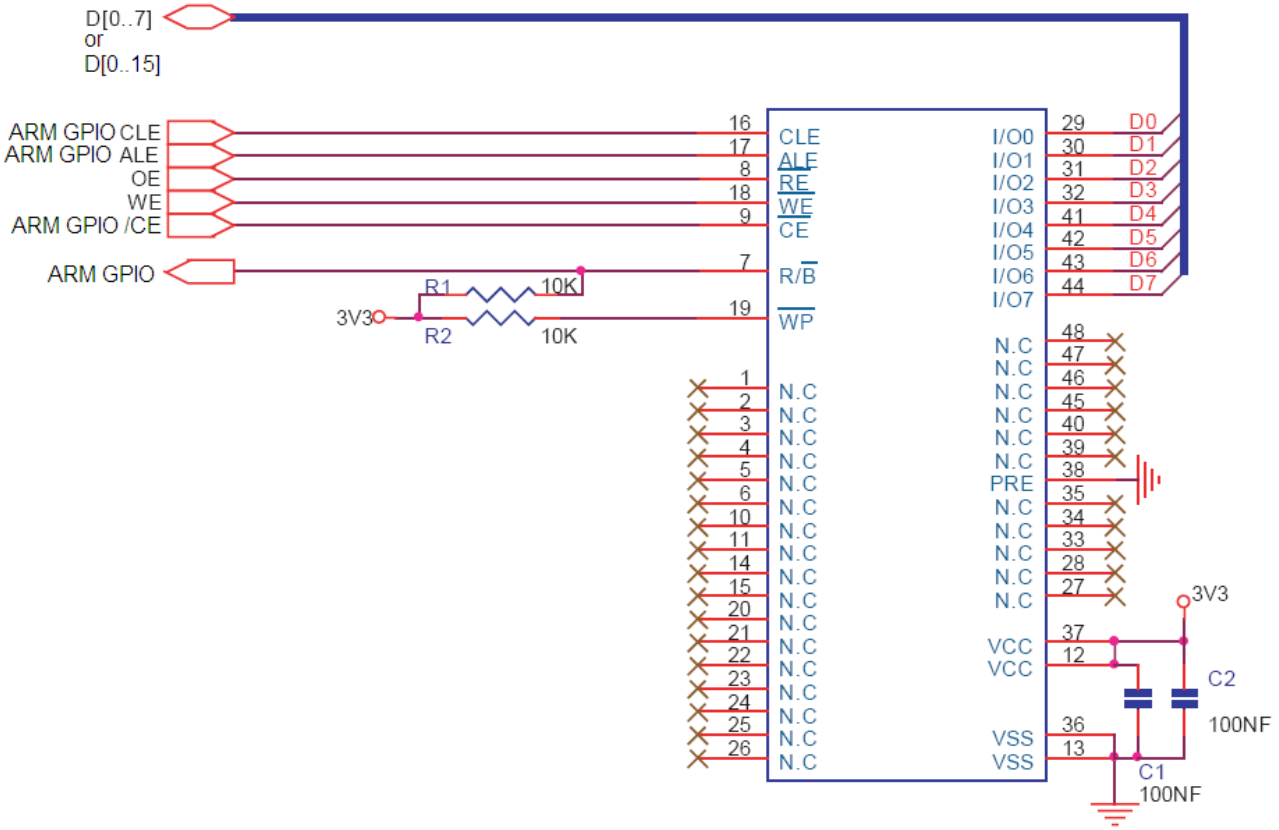
ALE and CLE Control signals – Due to the minimum timing requirements of tCLR, tAR and tALH of 10ns the signals ALE and CLE are required to be present prior to a given access type. The following table details the state of the GPIO signals for a given Access type.

Table 4: GPIO State Table

GPIO - CLE	*GPIO - ALE*	*GPIO - /CE*	*NAND Register Selected*
0	0	1	DATA register
0	1	1	ADDRESS register
1	0	1	COMMAND register

6 Hardware connectivity

Figure 19: x8 and x16 EBI to NAND conections



6.1 Software configuration

The following configuration must be carried out:

- Configure a GPIO line as an input and enable the clock of this GPIO to manage the Ready/Busy signal.
- Configure a GPIO line as an output and enable the clock of this GPIO to manage the ALE signal.
- Configure a GPIO line as an output and enable the clock of this GPIO to manage the CLE signal
- Configure a GPIO line as an output and enable the clock of this GPIO to manage the /CS. For /CS do not care devices this GPIO signal is not required Check the NAND datasheet to see if your device has the full /CS requirements.
- Configure the Static Memory Timing Register - Set 1 (SMTMGR_SET1) depending on NAND Flash device timings.
- Control GPIO signals as per Table 4: GPIO State Table

Table 5: NAND pin description

Pin	*Description*
I/O7 – I/O0	Data Inputs/Outputs. The I/O0 to I/O7 pins are used for transferring address, command, and input data to the device and output data from the device. Active for x8 and x16 ORNAND device configurations.
I/O15 – I/O8	Data Inputs/Data Outputs. The I/O8 to I/O15 pins are used for transferring address, command, and input data tothe device and output data from the device. Active only for x16 ORNAND device configuration. I/O8 to I/O15 must be low level during address and command input.
CLE	Command Latch Enable. The CLE input signal is used to control the loading of any command into the internal command register. The command is always latched into the internal command register from the data inputs on the rising edge of WE# signal while the CLE signal is active high.
ALE	Address Latch Enable. The ALE input signal is used to control the loading of any address into the internal address register. The address is always latched into the internal address register from the data inputs on the rising edge of WE# signal while the ALE signal is active high.
CE#	Chip Enable. The CE# input signal controls if the device is selected. When the device is in the Busy State (RY/BY# = L) while a read, program, or erase operation is in progress, the CE# signal going high is ignored. While in Busy State, the device will not enter Standby Mode even with the CE# signal going high.
RE#	Read Enable. The RE# input signal controls reads from the device. Output data is driven and becomes available tREA time after the falling edge of RE# signal. The internal address column counter increments up one address every RE# rising edge.
WE#	Write Enable. The WE# input signal controls writes to the device. Commands, addresses, and input data are latched on the rising edge of WE# signal.
WP#	Write Protect. The WP# input signal is used to protect the device from accidental programming or erasing. When WP# signal is asserted low, the entire memory is protected from writes and no program and erase operation can execute.
PRE	Power-On Read Enable. The PRE input signal controls the automatic read operation execution during power-on. The power-on automatic read feature is enabled when PRE signal is asserted high during power-on and allows page 0 to be automatically read out of the device so any system can automatically shadow boot code into DRAM without having to issue a read command. It is recommended that the user tie the PRE input pin to VCC.
RY/BY#	Ready/Busy. The RY/BY# output signal is used to indicate the operating condition of the device. The RY/BY# signal is always in the Busy State and asserts low when the device is executing read, program, and erase operations. When a read, program, or erase operation has completed, the RY/BY# signal stops asserting low to indicate completion and that the device is ready for a new operation. Since the RY/BY# output is open-drain, an external pull-up resistor is required.
VCC	Power Supply. The VCC input is the main power supply for this device.
VSS	Ground. The VSS input is the main supply ground for this device.
NC	No Connect. Lead is not internally connected.
VIO	VIO is the I/O power supply for device. The VIO pin exists only if the Enhanced VersatileI/O control feature is needed through a different product ordering part number.
FP#	Fast Program/Erase Mode. FP# is used to set the device in standard mode or Fast Program/Erase Mode. When FP# is held at VIL, the device operates in Fast Program/Erase Mode. When FP# is held at VIH, the device operates in standard mode. Note: FP# contains an internal pull-up. When unconnected, FP# is at VIH.

Index of Tables

7 Index of Figures

8 Version History

Version	Date	Author(s)	Changes made
1.0	5th Dec	RR	Initial release for review

9 Appendix A NAND AC Characteristics.

Parameter	Description		Min. (x8, no V_{IO})	Max (x8, no V_{IO})	Min. (x8 with V_{IO}, x16 no V_{IO}, x16 with V_{IO})	Max (x8 with V_{IO}, x16 no V_{IO}, x16 with V_{IO})	Unit
t_{CLS}	CLE Setup Time		0	-	0	-	ns
t_{CLH}	CLE Hold Time		10	-	10	-	
t_{CS}	CE Setup Time		0	-	0	-	
t_{CH}	CE Hold Time		10	-	10	-	
t_{WP}	Write Pulse Width (See Note)		15	-	25	-	
t_{ALS}	ALE Setup Time		0	-	0	-	
t_{ALH}	ALE Hold Time		10	-	10	-	
t_{DS}	Data Setup Time		15	-	20	-	
t_{DH}	Data Hold Time		5	-	10	-	
t_{WC}	Write Cycle Time		30	-	45	-	
t_{WH}	WE# High Hold Time		10	-	15	-	
t_{WW}	WP# High to WE# Low		100	-	100	-	
t_{RR}	Ready to RE# Falling Edge		20	-	20	-	
t_{RW}	Ready to WE# Falling Edge		20	-	20	-	
t_{RP}	Read Pulse Width		20	-	30	-	
t_{RC}	Read Cycle Time		30	-	50	-	
t_{REA}	RE# Access Time		-	18	-	30	
t_{CEA}	CE# Access Time		-	23	-	35	
t_{CLR}	CLE to RE# delay		10	-	10	-	
t_{ALEA}	ALE Access Time		-	28	-	40	
t_{OH}	Data Output Hold Time		10	-	10	-	
t_{RHZ}	RE# High to Output High Impedance		-	40	-	40	
t_{CHZ}	CE# High to Output High Impedance		-	40	-	40	
t_{REH}	RE# High Hold Time		5	-	15	-	
t_{IR}	Output High Impedance to RE# Falling Edge		0	-	0	-	
t_{RHW}	RE# High to WE# Low		30	-	30	-	
t_{WHC}	WE# High to CE# Low		30	-	30	-	
t_{WHR}	WE# High to RE# Low		60	-	60	-	
t_R	Data Transfer from Memory to Register		-	9	-	9	µs
t_{WB}	WE# High to Busy		-	100	-	100	ns
t_{RST}	Device Reset (Read/Program/Erase)	Read	5	-	5	-	µs
		Program	10	-	10	-	
		Erase	500	-	500	-	
t_{READ}	RE# Access Time (Read ID)			18	-	18	ns

10 Appendix B – ECC.C code example

```
/*******************************************************************************
*
*    File Name:  ECC.c
*     Revision:  1.0
*         Date:  Jan 20, 2006
*        Email:  nandsupport@micron.com
*      Company:  Micron Technology, Inc.
*
*  Description:  Micron NAND I/O Driver
**
*   Disclaimer:  This software code and all associated documentation,
*     comments or other information (collectively "Software") is provided
*     "AS IS" without warranty of any kind. MICRON TECHNOLOGY, INC.
*     ("MTI") EXPRESSLY DISCLAIMS ALL WARRANTIES EXPRESS OR IMPLIED,
*     INCLUDING BUT NOT LIMITED TO, NONINFRINGEMENT OF THIRD PARTY
*     RIGHTS, AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS
*     FOR ANY PARTICULAR PURPOSE. MTI DOES NOT WARRANT THAT THE
*     SOFTWARE WILL MEET YOUR REQUIREMENTS, OR THAT THE OPERATION OF
*     THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE. FURTHERMORE,
*     MTI DOES NOT MAKE ANY REPRESENTATIONS REGARDING THE USE OR
*     THE RESULTS OF THE USE OF THE SOFTWARE IN TERMS OF ITS CORRECTNESS,
*     ACCURACY, RELIABILITY, OR OTHERWISE. THE ENTIRE RISK ARISING OUT
*     OF USE OR PERFORMANCE OF THE SOFTWARE REMAINS WITH YOU. IN NO
*     EVENT SHALL MTI, ITS AFFILIATED COMPANIES OR THEIR SUPPLIERS BE
*     LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, INCIDENTAL,
*     OR SPECIAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS
*     OF PROFITS, BUSINESS INTERRUPTION, OR LOSS
*     OF INFORMATION) ARISING OUT OF YOUR USE OF OR INABILITY TO USE THE
SOFTWARE,
*     EVEN IF MTI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
*     Because some jurisdictions prohibit the exclusion or limitation of
liability for
*     consequential or incidental damages, the above limitation may not apply to
you.
*
*                Copyright © 2006 Micron Technology, Inc.
*                All rights researved
*
* Rev  Author                Date        Changes
* ---  --------------  ----------  ------------------------------
* 1.0  WW                      01/20/2006  Initial release
*/


#include <Windows.h>
#include <assert.h>

typedef unsigned short  u16;
typedef unsigned char*  u8p;
typedef unsigned short* u16p;
typedef unsigned*       u32p;
typedef void *                voidp;
```

```
typedef signed char s8;
typedef unsigned char u8;

typedef signed short s16;
typedef unsigned short u16;

typedef signed int s32;
typedef unsigned int u32;


u8 gSilent = 1;

/*
      [Data Format]
      ------------------------------------------------------------------------
      7           6           5           4           3           2           1
            0
      ------------------------------------------------------------------------
      ...
                        Byte0 P8e | P16e | P32e | P2048e
      ...
                        Byte1 P8o |      |      |
      ...
                        Byte2 P8e | P16o |      |
      ...
                        Byte3 P8o |      |      |
      ...

      ...
                                                | P2048o
      ...
                        Byte512     P8o               |
      ------------------------------------------------------------------------
      P1o         P1e         P1o         P1e         p1o         P1e
      P1o         P1e
      -----------     -----------     -----------     -----------
          P2o             P2e                 ....
            ------------------              ----------------
                    P4o                             P4e
      ------------------------------------------------------------------------

      [Reg Format]
      ------------------------------------------------------------------------
      31 ... 27      26     .... 16       15      14     ....  2    1    0
      ------------------------------------------------------------------------
             P2048O  P1024O      P1O      P2048E  P1024E       P4E  P2E  P1E   ?
EO or OE
      ------------------------------------------------------------------------

      [Calc Format]
      ------------------------------------------------------------------------
      7           6           5           4           3           2           1
            0
      ------------------------------------------------------------------------
      P128e   P64e    P32e    P16e    P8e     P4e     P2e     P1e
      ------------------------------------------------------------------------
```

```
      P128o  ..............................................  P10
      ------------------------------------------------------------------------
      P2048o  P1024o  P512o   P256o   P2048e  P1024e  P512e   P256e
      ------------------------------------------------------------------------

      [Fail Bit Address Offset]
      ------------------------------------------------------------------------
      <P2048 P1024 ... P4 P2 P1>                        ==> 11 bit
      ------------------------------------------------------------------------

      ------------------------------------------------------------------------
      ------------------------------------------------------------------------
      ------------------------------------------------------------------------
      ------------------------------------------------------------------------

*/

#define P1EMask         (1 << 0)
#define P2EMask         (1 << 1)
#define P4EMask         (1 << 2)
#define P8EMask         (1 << 3)
#define P16EMask  (1 << 4)
#define P32EMask  (1 << 5)
#define P64EMask  (1 << 6)
#define P128EMask (1 << 7)
#define P256EMask (1 << 8)
#define P512EMask (1 << 9)
#define P1024EMask      (1 << 10)
#define P2048EMask      (1 << 11)

#define P1OMsk          (1 << 16)
#define P2OMsk          (1 << 17)
#define P4OMsk          (1 << 18)
#define P8OMsk          (1 << 19)
#define P16OMsk         (1 << 20)
#define P32OMsk         (1 << 21)
#define P64OMsk         (1 << 22)
#define P128OMsk  (1 << 23)
#define P256OMsk  (1 << 24)
#define P512OMsk  (1 << 25)
#define P1024OMsk (1 << 26)
#define P2048OMsk (1 << 27)

#define TF(value) (value ? 1 : 0)

#define P2048e(a) (TF(a & P2048EMask)     << 0 )
#define P2048o(a) (TF(a & P2048OMsk)      << 1 )
#define P1e(a)          (TF(a & P1EMask)  << 2 )
#define P1o(a)          (TF(a & P1OMsk)         << 3 )
#define P2e(a)          (TF(a & P2EMask)  << 4 )
#define P2o(a)          (TF(a & P2OMsk)         << 5 )
#define P4e(a)          (TF(a & P4EMask)  << 6 )
#define P4o(a)          (TF(a & P4OMsk)         << 7 )

#define P8e(a)          (TF(a & P8EMask)  << 0 )
#define P8o(a)          (TF(a & P8OMsk)         << 1 )
```

```
#define P16e(a)         (TF(a & P16EMask) << 2 )
#define P16o(a)         (TF(a & P16OMsk)  << 3 )
#define P32e(a)         (TF(a & P32EMask) << 4 )
#define P32o(a)         (TF(a & P32OMsk)  << 5 )
#define P64e(a)         (TF(a & P64EMask) << 6 )
#define P64o(a)         (TF(a & P64OMsk)  << 7 )

#define P128e(a)  (TF(a & P128EMask)      << 0 )
#define P128o(a)  (TF(a & P128OMsk) << 1 )
#define P256e(a)  (TF(a & P256EMask)      << 2 )
#define P256o(a)  (TF(a & P256OMsk) << 3 )
#define P512e(a)  (TF(a & P512EMask)      << 4 )
#define P512o(a)  (TF(a & P512OMsk) << 5 )
#define P1024e(a) (TF(a & P1024EMask)     << 6 )
#define P1024o(a) (TF(a & P1024OMsk)      << 7 )

#define P8e_s(a)  (TF(a & P8EMask)  << 0 )
#define P8o_s(a)  (TF(a & P8OMsk)         << 1 )
#define P16e_s(a) (TF(a & P16EMask) << 2 )
#define P16o_s(a) (TF(a & P16OMsk)  << 3 )
#define P1e_s(a)  (TF(a & P1EMask)  << 4 )
#define P1o_s(a)  (TF(a & P1OMsk)         << 5 )
#define P2e_s(a)  (TF(a & P2EMask)  << 6 )
#define P2o_s(a)  (TF(a & P2OMsk)         << 7 )

#define P4e_s(a)  (TF(a & P4EMask)  << 0 )
#define P4o_s(a)  (TF(a & P4OMsk)         << 1 )

u8 mu_pack_ecc(u8 id, u8 *ecc_buf)
      {
      u32 l;
      l = 0; //mu_ecc_pgseq_val(id);
      ecc_buf[0] = l;          // P128e, ..., P1e
      ecc_buf[1] = l >> 16;    // P128o, ..., P1o
      ecc_buf[2] = ((l >> 8) & 0x0f) | ((l >> 20) & 0xf0); // P2048o, P1024o,
P512o, P256o, P2048e, P1024e, P512e, P256e
      printf(" [packecc(%8x -> %3X %3X %3X)]\n", l, ecc_buf[0], ecc_buf[1],
ecc_buf[2]);
      return 0;
      }

u8 mu_count_bits(u32 v)
      {
      u8 i, count;

      for(count=i=0; i<32; i++)
            count += (v>>i)&1 ? 1:0;
      return count;
      }

u8 mu_unpack_ecc(u8 id, u8 *ecc_mem, u8* ecc_reg, u32* m, u32* r)
      {
      u32 mem, reg;

      //reg = mu_ecc_pgseq_val(id);
      mem = ecc_mem[0] | (ecc_mem[1] << 16) | ((ecc_mem[2] & 0xF0) << 20) |
```

```
((ecc_mem[2] & 0x0F) << 8);
      reg = ecc_reg[0] | (ecc_reg[1] << 16) | ((ecc_reg[2] & 0xF0) << 20) |
((ecc_reg[2] & 0x0F) << 8);
      if(!gSilent) printf("[unpackecc(%3X %3X %3X) (mem:reg=%8x:%8x) ",
ecc_mem[0], ecc_mem[1], ecc_mem[2], mem, reg);
      if( mem == reg )
            {
            if(!gSilent) printf("\n");
            return 0;
            }
      else
            {
            u32 tmp;
            tmp = mem ^ reg;
            if(!gSilent) printf("mu_unpack_ecc: XOR=%x XorBits=%X Xor&memBits=%x
Xor&regBits=%x\n",
                  tmp, mu_count_bits(tmp), mu_count_bits(tmp&mem),
mu_count_bits(tmp&reg) );
            *m = mem;
            *r = reg;
            }

      return 1;
      }

u8 mu_ecc_correct(u32 m, u32 r, u8 *buf)
      {
      u16 pe, po, is_ecc_ff;

      is_ecc_ff = ((m & 0xFFFFFF) == 0xFFFFFF);
      m = m^r;

      switch (mu_count_bits(m))
            {
            case 0: // shall never get there since ecc equal, no call in
                  return 0;
            case 1: // dont correct
                  if(!gSilent) printf ( "ECC area error. No need to
correct.\n");
                  return 0;
            case 12: // Correctable ?
                  po = m & 0x0000FFFF;
                  pe = (m>>16) & 0x0000FFFF;
                  po = (pe^po);
                  if( 0x0FFF == po )
                        {
                        buf[pe>>3] ^= (1 << (pe&7));
                        if(!gSilent) printf ( "Single bit error: offset: %d,
bit: %d\n", pe>>3, pe&7);
                        return 0;
                        }
                  else
                        {
                        if(!gSilent) printf( "Uncorrectable error A !! \n");
                        return -1;
                        }
```

```
            default:
                  if(is_ecc_ff && r == 0)
                        return 0;
                  if(!gSilent) printf( "Uncorrectable error B !! \n");
                  return -1;
            }
      }

u8 mu_ecc_generate_512(u8 *eccbuf, u8 *datbuf)
      {
      u32 i, j;
      u8    byteParity[512], tmp = 0, tmp2=0;
      u8  parity_LUT[32] = {0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0, /**/
1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1};
      u8  sum=0, Pcount = 0;
      u8    Pc0 = 0, Pc1 = 0, Pc2 = 0, Pc3 = 0, Pc4 = 0, Pc5 = 0, Pc6 = 0, Pc7 =
0;
      u8    PE1, PO1, PE2, PO2, PE4, PO4;
      u8    PE8 = 0, PO8 = 0, PE16 = 0, PO16 = 0, PE32 = 0, PO32 = 0;
      u8    PE64 = 0, PO64 = 0, PE128 = 0, PO128 = 0, PE256 = 0, PO256 = 0;
      u8    PE512 = 0, PO512 = 0, PE1024 = 0, PO1024 = 0, PE2048 = 0, PO2048 =
0;

      for( i = 0; i < 512; i++)
            {
            Pcount = Pcount ^ datbuf[i];
            tmp = (datbuf[i] & 0xf0) >> 4;
            tmp2 = datbuf[i] & 0x0f;
            switch(tmp)
                  {
                  case 0:     case 3:     case 5:     case 6:     case 9:
      case 10: case 12: case 15:
                        byteParity[i] = parity_LUT[tmp2]; break;
                  default:
                        byteParity[i] = parity_LUT[tmp2+16]; break;
                  }
            }

      Pc0 = ((Pcount & 0x01) ? 1 : 0);
      Pc1 = ((Pcount & 0x02) ? 1 : 0);
      Pc2 = ((Pcount & 0x04) ? 1 : 0);
      Pc3 = ((Pcount & 0x08) ? 1 : 0);
      Pc4 = ((Pcount & 0x10) ? 1 : 0);
      Pc5 = ((Pcount & 0x20) ? 1 : 0);
      Pc6 = ((Pcount & 0x40) ? 1 : 0);
      Pc7 = ((Pcount & 0x80) ? 1 : 0);

      PO1 = Pc6 ^ Pc4 ^ Pc2 ^ Pc0;
      PE1 = Pc7 ^ Pc5 ^ Pc3 ^ Pc1;
      PO2 = Pc5 ^ Pc4 ^ Pc1 ^ Pc0;
      PE2 = Pc7 ^ Pc6 ^ Pc3 ^ Pc2;
      PO4 = Pc3 ^ Pc2 ^ Pc1 ^ Pc0;
      PE4 = Pc7 ^ Pc6 ^ Pc5 ^ Pc4;

      for( i = 0 ; i < 512;i++)
            sum=sum ^ byteParity[i];
```

```
for ( i = 0; i < 512; i = i+2 )
      PO8 = PO8 ^ byteParity[i];

for ( i = 0; i < 512; i = i+4 )
      {
      PO16 = PO16 ^ byteParity[i];
      PO16 = PO16 ^ byteParity[i + 1];
      }
for ( i = 0; i < 512; i = i+8 )
      {
      for ( j = 0; j <= 3; j++ )
            PO32 = PO32 ^ byteParity[i+j];
      }
for ( i = 0; i < 512; i = i+16 )
      {
      for ( j = 0; j <= 7; j++ )
            PO64 = PO64 ^ byteParity[i+j];
      }
for ( i = 0; i < 512; i = i+32 )
      {
      for ( j = 0; j <= 15; j++ )
            PO128 = PO128 ^ byteParity[i+j];
      }
for ( i = 0; i < 512; i = i+64 )
      {
      for ( j = 0; j <= 31; j++ )
            PO256 = PO256 ^ byteParity[i+j];
      }
for ( i = 0; i < 512; i = i+128 )
      {
      for ( j = 0; j <= 63; j++ )
            PO512 = PO512 ^ byteParity[i+j];
      }
for ( i = 0; i < 512; i = i+256 )
      {
      for ( j = 0; j <= 127; j++ )
            PO1024 = PO1024 ^ byteParity[i+j];
      }

for ( i = 0; i < 512; i = i+512 )
      {
      for ( j = 0; j <= 255; j++ )
            PO2048 = PO2048 ^ byteParity[i+j];
      }

if(sum==0)
      {
      PE2048=PO2048;
      PE1024=PO1024;
      PE512=PO512;
      PE256=PO256;
      PE128=PO128;
      PE64=PO64;
      PE32=PO32;
      PE16=PO16;
```

```
            PE8=PO8;
            }
      else
            {
            PE2048 = (PO2048 ? 0 : 1);
            PE1024 = (PO1024 ? 0 : 1);
            PE512  = (PO512  ? 0 : 1);
            PE256  = (PO256  ? 0 : 1);
            PE128  = (PO128  ? 0 : 1);
            PE64   = (PO64   ? 0 : 1);
            PE32   = (PO32   ? 0 : 1);
            PE16   = (PO16   ? 0 : 1);
            PE8    = (PO8    ? 0 : 1);
            }

      eccbuf[0]=(PO128 << 7) |(PO64 << 6) |(PO32 << 5) |(PO16 << 4) |(PO8 << 3)
|(PO4 << 2) |(PO2 << 1) |(PO1);
      eccbuf[1]=(PE128 << 7) |(PE64 << 6) |(PE32 << 5) |(PE16 << 4) |(PE8 << 3)
|(PE4 << 2) |(PE2 << 1) |(PE1);
      eccbuf[2]=(PE2048 << 7) |(PE1024 << 6) |(PE512 << 5) |(PE256 << 4)
|(PO2048 << 3) |(PO1024 << 2) |(PO512 << 1) |(PO256);

      if(0)
            {
            u8    ecctmp[3];
            u32 tmp = eccbuf[0] | (eccbuf[1] << 16) | ((eccbuf[2] & 0xF0) << 20)
| ((eccbuf[2] & 0x0F) << 8);

            ecctmp[0] = ~(P64o(tmp) | P64e(tmp) | P32o(tmp) | P32e(tmp) |
P16o(tmp) | P16e(tmp) | P8o(tmp) | P8e(tmp) );
            ecctmp[1] = ~(P1024o(tmp) | P1024e(tmp) | P512o(tmp) | P512e(tmp) |
P256o(tmp) | P256e(tmp) | P128o(tmp) | P128e(tmp));
            ecctmp[2] = ~( P4o(tmp) | P4e(tmp) | P2o(tmp) | P2e(tmp) | P1o(tmp)
| P1e(tmp) | P2048o(tmp) | P2048e(tmp));

            printf("mu_prt_ecc: %3X %3X %3X \n", ecctmp[0], ecctmp[1],
ecctmp[2]);

            printf("    P2048o.e(tmp)     = %d : %d\n", P2048o(tmp)?1:0,
P2048e(tmp)?1:0);
            printf("    P1024o.e(tmp)     = %d : %d\n", P1024o(tmp)?1:0,
P1024e(tmp)?1:0);
            printf("    P512o.e(tmp)      = %d : %d\n", P512o(tmp)?1:0,
P512e(tmp)?1:0);
            printf("    P256o.e(tmp)      = %d : %d\n", P256o(tmp)?1:0,
P256e(tmp)?1:0);
            printf("    P128o.e(tmp)      = %d : %d\n", P128o(tmp)?1:0,
P128e(tmp)?1:0);
            printf("    P64o.e(tmp) = %d : %d\n", P64o(tmp)?1:0, P64e(tmp)?1:0);
            printf("    P32o.e(tmp) = %d : %d\n", P32o(tmp)?1:0, P32e(tmp)?1:0);
            printf("    P16o.e(tmp) = %d : %d\n", P16o(tmp)?1:0, P16e(tmp)?1:0);
            printf("    P8o.e(tmp)  = %d : %d\n", P8o(tmp)?1:0, P8e(tmp)?1:0);
            printf("    P4o.e(tmp)  = %d : %d\n", P4o(tmp)?1:0, P4e(tmp)?1:0);
            printf("    P2o.e(tmp)  = %d : %d\n", P2o(tmp)?1:0, P2e(tmp)?1:0);
            printf("    P1o.e(tmp)  = %d : %d\n", P1o(tmp)?1:0, P1e(tmp)?1:0);
            }
```

```
	return 0;
	}


// Unit testing

#if 1

void  HammingInstErrSimple(u8*	buf, u16 len)
{
	u16		j, err;
	u8		mask, bf, af;
	#define MaxErr 1

	for(err=0; err<MaxErr; )
	{
		j = rand()%len;
		assert(j<len);
		mask = ~( 1<<(rand()%8) );
		if( (buf[j] & mask) != buf[j] )
		{
			bf = buf[j];
			buf[j] &= mask;
			af = buf[j];
			printf("%d %X %X\n", j, bf, af);
			err++;
		}
	}
}

extern u8 tstdat[2112];

int main(int argc, char* argv[])
{
	int i, nNG=0, nOK=0;
	u32 m,r;

	u8 eccbuf[4], eccbuf2[4];

	mu_ecc_generate_512(eccbuf, tstdat);
	printf("eccbuf: %2X %2X %2X \n", eccbuf[0], eccbuf[1], eccbuf[2]);
	mu_ecc_generate_512(eccbuf, tstdat+512);
	printf("eccbuf: %2X %2X %2X \n", eccbuf[0], eccbuf[1], eccbuf[2]);
	mu_ecc_generate_512(eccbuf, tstdat+1024);
	printf("eccbuf: %2X %2X %2X \n", eccbuf[0], eccbuf[1], eccbuf[2]);
	mu_ecc_generate_512(eccbuf, tstdat+1024+512);
	printf("eccbuf: %2X %2X %2X \n", eccbuf[0], eccbuf[1], eccbuf[2]);

	nNG=nOK=0;
	for(i=0; i<800; i++)
	{
		printf("[%d] \n", i);
		mu_ecc_generate_512(tstdat+512, tstdat);
		if(i<400)
			HammingInstErrSimple(tstdat, 512);
```

```
            else
                  HammingInstErrSimple(tstdat+512, 3);
            mu_ecc_generate_512(eccbuf, tstdat);
            printf("ecc: Before=%2X %2X %2X : After=%2X %2X %2X \n",
tstdat[512+0], tstdat[512+1], tstdat[512+2], eccbuf[0], eccbuf[1], eccbuf[2]);
            if( mu_unpack_ecc(0, tstdat+512, eccbuf, &m, &r) )
            {
                  if( mu_ecc_correct(m, r, tstdat) )
                  {
                        nNG++;
                  }
                  else
                  {
                        nOK++;
                  }
            }
            else
            {
                  printf("                                                  No
error detected!\n");
                  while(1);
            }
      }
      printf("                                                NG OK = %d
%d\n", nNG, nOK);
}

#endif
```