A. Affected Silicon Revision

Errata details published in this document refer to the following silicon:

netX50, Revision A (Step A, Rom Code Rev.1, Boot loader major vers. 0x42)

B. Document Revision History

Revision	Date	Description	Author
1.0	Feb, 2008	Created, added Errata 1 - 6	J. Lipfert
1.1	August, 2008	Extended silicon description (Chapter A.)	J. Lipfert
1.2	Nov., 5th, 2009	Added Errata 7 - 9	J. Lipfert
1.3	April, 28th, 2010	Added Errata 10 - 13	J. Lipfert
1.4	Dec., 8th, 2010	Added Erratum 14	J. Lipfert M. Melzer

C. Errata Summary

Nr.	Description
1	DMA Controller: Controller can not write to SDRAM
2	SDRAM: Access to offset 0xDEAD0 – DEADF fails after Power On Reset
3	SPI: Setting the clock divider to 0 and starting a transfer causes SPI core to hang
4	SPI: Legacy Register “CR_NCPHA” Bit appears inverted when read
5	I^2C: Signal timing can cause problems with some components
6	Reset Control Register: Register not protected by netX locking mechanism
8	SPI Master: Transmit FIFO may loose data
9	UARTs: Using Transmit FIFO may result in wrong transmit data
10	Host interface: Watchdog Output / DPM_D19 signal is driven low after Reset
11	Host interface: Error in configuration register regarding DPM ISA mode
12	GPIO module: Interrupts may be lost
13	Internal PHYs: Error in 10 Mbit half duplex mode
14	Host Interface: DPM access time with Hilscher standard DPM layout is unpredictable

D. Errata Details:

1 DMA Controller: Controller can not write to SDRAM

Issue:

Due to a hardware bug, the DMA Controller has no write access to the SDRAM address range hence any DMA transfers to SDRAM will never be finished, leaving the targeted address range inaccessible for any other unit.

Solution / Workaround:

Do not set up DMA transfers to SDRAM (except dummy transfer, described in erratum 2).

2 SDRAM: Access to offset 0xDEAD0- 0xDEADF fails after Power On Reset

Issue:

When accessing the SDRAM at offset 0xDEAD0 – 0xDEADF (absolute addresses 0x800DEAD0 – 0x800DEADF) after Power On Reset, the ARM CPU hangs.

This issue is related to erratum 1: After Power On Reset, the address logic erroneously detects an active DMA transfer to SDRAM address 0x800DEAD0 which causes this address and the following 15 Byte to be inaccessible.

Solution / Workaround:

Set up a dummy DMA transfer to an unused SDRAM address (preferably 0xBFFF FFFC, which is the last Dword in the SDRAM address range), which makes the above mentioned address range accessible again. This dummy transfer is required only once after Power On Reset.

3 SPI: Setting the clock divider to ‘0’ and starting a transfer causes the SPI core to hang

Issue:

When setting the clock divider parameter (SCK_MULADD, Bits [19:8] in SPI Control register 0 and 1) to 0 and starting a transfer, the SPI core locks up.

Solution / Workaround:

Do not set the clock divider to 0 (which is not a useful setting anyway).

4 SPI: Legacy Register “CR_NCPHA” Bit appears inverted when read

Issue:

When using the Legacy Registers of the SPI interface, Bit 28 (CR_NCPHA) of the SPI0_LGY_CTRL (0x1C000D38) and SPI1_LGY_CTRL (0x1C000D78) Registers always shows the inverted value of its actual register value, while writing the Bit works correctly.

Solution / Workaround:

Generally it is strongly recommended to avoid using the Legacy Registers unless absolutely necessary (code compatibility with netX100/500). When using the Legacy Registers, software must be aware of the inverted bit and behave accordingly.

5 I^2C: Signal timing can cause problems with some components

Issue:

The I2C core of the netX50 changes signal state of the SDA line almost simultaneously with the falling edge of SCL. Though this behavior is compliant with the I2C spec, some components may erroneously detect a STOP condition at the falling edge of SDA.

Solution / Workaround:

A small capacitor (100 pF), connected between GND and the SDA signal line, should be added to designs using the I2C interface, allowing to delay the SDA signal if necessary.

6 Reset Control Register: Register not protected by netX locking mechanism

Issue:

The netX Reset Control Register, RESET_CTRL (0x1C00000C), meant to be protected against undesired accesses by the netX locking mechanism, can be read and written directly.

Solution / Workaround:

As the register and the functions it controls are working properly, no workaround is necessary. However, the following notes should be regarded:

a) Though currently not required, the unlocking procedure as described in the Program Reference Manual should still be implemented, in order to ensure software compatibility with future chip revisions that will implement the locking feature correctly.

b) Since step a) will leave the ASIC_CTRL Register area unprotected till the next access, the access to the Reset Control Register should be followed by a dummy read access to any of the other registers in that area (e.g. CLK_EN at 0x1C000024).

7 moved

8 SPI Master: Transmit FIFO may loose data.

Issue:

When using the SPI interface(s) in Master Mode and 16 Bit word length, a word that is written to the Transmit FIFO while the interface is currently at the end of the transmission of a previously written word, is discarded and not being transmitted. This problem does not occur when using word lengths < 16 Bit (4 Bit to 15 Bit).

Solution / Workaround:

When using 16 Bit word length, words may only be written to the FIFO, when the previous transmission has ended (check for inactive BSY Bit in the SPI0_STAT / SPI1_STAT Register before writing to FIFO)

9 UARTs: Using Transmit FIFO may result in wrong transmit data.

Issue:

When writing to the transmit FIFO of a UART, while the UART currently reads the next data word from the FIFO, wrong data is transmitted by the UART.

Solution / Workaround:

a) Do not use the UART FIFOs.

(Clear Bit FEN of UART0_LINE_CTRL / UART1_LINE_CTRL / UART2_LINE_CTRL registers).

OR

b) Write to the Transmit FIFOs only when they are empty (Bit TXFE of UART0_FLAG / UART1_FLAG / UART2_FLAG is set) which effectively results in using the Receive FIFOs only.

OR

c) Disable the UART before filling the Transmit FIFO (Clear Bit UARTEN of UART0_CTRL / UART1_CTRL / UART2_CTRL) and enable afterwards for transmission. As a UART of course can not receive data while disabled, two UARTs can be used, one for receiving data, which is always enabled and one for transmitting data, which is disabled while filling the FIFO.

10 Host interface: Watchdog Output / DPM_D19 signal is driven low after Reset

Issue:

After releasing the netX50 from Reset (PORn or RSTINn), the WDGACT / DPM_D19 signal on pin G17 is configured as Watchdog Output signal and is driven to a low level.

With designs using the 32 Bit mode of the netX50, this may cause contention / short circuit conditions if an external host processor or any other possible component on the host bus drives data line D19 high during system power up and/or may interfere with possible pull-up strapping options on the host processor where data line D19 is involved.

Once the netX50 host interface is switched to 32 Bit mode by writing 3'b101 to the HIF_MODE Bits (30:28) of the DPM_ARM_IF_CFG0 register (at address 0x1c003608), configuring the WDACT / DPM_D19 signal as data line D19 (which is mandatory for 32 Bit designs anyway), the signal contention risk is banned.

Solution / Workaround:

a) Design the system reset circuit in a way that either holds any other components, that may drive data line D19, in Reset state until the netX50 Firmware has been started, configuring the host interface in 32 Bit DPM mode, or that holds the netX50 in Reset state, until the rest of the system enters a state where the short period while the netX50 drives D19 low will not cause any problems.

OR

b) Use a bus switch (e.g. NC7SZ384, single gate switch) to connect the netX50 DPM_D19 to the data bus, while the switch will not be enabled (e.g. by a PIO signal) before the netX50 host interface has been switched to 32 Bit mode.

11 Host interface: Error in Configuration Register regarding DPM ISA mode

Issue:

When using the netX50 in 16 Bit ISA Mode, the BE1_MODE Bits (23:21) of the DPM_ARM_IF_CFG0 (at address 0x1c003608) register will usually be set to 3'b101 in order to activate the generation of the MEM_CS16n signal on the DPM_WRHn / PIO44 pin (C15). This setting however makes the internal Byte Enable 1 (High Byte) signal active high, while the appropriate ISA Bus Signal (SBHE, System Byte High Enable) connected to the DPM_BHEn / PIO43 pin (A18) is actually active low.

As a result, the corresponding high byte will always be written inadvertently when the Host performs a single byte write access to the low byte, while the high byte will remain untouched when the host performs 16 Bit or single high byte write accesses.

Solution / Workaround:

When creating new hardware designs, insert an inverter gate, powered by 3.3V with 5V tolerant input (e.g.NC7SZ14, single gate) between the ISA SBHE# signal and the DPM_BHEn pin.

For existing designs where a redesign is not an option, set the BE1_MODE Bits to 3'b001 (High Byte is selected, when address line A0 = 1) and the HIF_MODE Bits to 3'b010 (8 Bit DPM mode). This will configure the netX50 DPM for 8 Bit mode and disable the MEM_CS16n signal generation, which causes the Host to make 8 Bit accesses only.

12 GPIO module: Interrupts may be lost

Issue:

Whenever a pending Interrupt, generated by the GPIO module is confirmed, any further GPIO based interrupt event that occurs simultaneously with the confirmation will be cleared either and is hence lost.

Solution / Workaround:

While some GPIO module based interrupt events may be used along with a special IRQ confirmation routine, the following events must not be used with interrupts at all:

Timer modes:

- Triangle (parameter *syn_nasym* = 1'b1 in gpio_counterX_ctrl register)

External events:

- Automatic Run (parameter *event_act* = 2'b11 in gpio_counterX_ctrl register)

Input modes:

- capture cont. at rising edge (parameter *mode* = 4'b0001 in gpio_cfgX register)
- capture once at rising edge (parameter *mode* = 4'b0010 in gpio_cfgX register)
- capture once at high level (parameter *mode* = 4'b0011 in gpio_cfgX register)

  (This mode may be used along with the confirmation routine, if input signal remains at high level for at least 20 ns)

IO-Link

Please note that all mentioned GPIO functions itself may be used, if interrupt functionality is not required.

All other functions may be used for interrupt generation as long as the IRQ confirmation routine is being used, which is shown on the following pages:

```
#define NX50_lock_irqfiq_save(x)                                \
        ({                                                      \
                register unsigned int cpsr_tmp;                 \
                __asm__ __volatile__(                           \
                        "MRS %1, cpsr\n"                        \
                        "AND %0, %1, #0xC0\n"                   \
                        "ORR %1, %1, #0xC0\n"                   \
                        "MSR CPSR_c, %1"                        \
                        : "=r" (x), "=&r" (cpsr_tmp)            \
                );                                              \
        })

#define NX50_lock_irqfiq_restore(x)                             \
        ({                                                      \
                register unsigned int cpsr_tmp;                 \
                __asm__ __volatile__(                           \
                        "MRS %0, cpsr\n"                        \
                        "EOR %0, %0, #0xC0\n"                   \
                        "ORR %0, %0, %1\n"                      \
                        "MSR CPSR_c, %0"                        \
                        : "=&r" (cpsr_tmp)                      \
                        : "r" (x)                               \
                );                                              \
        })


#define MSK_GPIO_COUNTER0_2  (MSK_NX50_cnt_irq_raw_cnt0 | \
                              MSK_NX50_cnt_irq_raw_cnt1 | \
                              MSK_NX50_cnt_irq_raw_cnt2)

#define MSK_GPIO_COUNTER3_4  (MSK_NX50_cnt_irq_raw_cnt3 | \
                              MSK_NX50_cnt_irq_raw_cnt4)


void NX50_GPIO_ConfirmIrq( unsigned long ulConfirm, unsigned long ulMask );
```

```
/*****************************************************************************/
/*! GPIO Irq Reset
* \description
*   Replaces the write to Adr_NX50_gpio_cnt_irq_raw and Adr_NX50_gpio_gpio_irq_raw
*   for interrupt confirmation. \n
*    This must be called by everyone who wants to confirm an interrupt generated by the GPIO
module.
* \class
*   GPIO
* \params
*    ulConfirm   [in] Adr_NX50_gpio_cnt_irq_raw / Adr_NX50_gpio_gpio_irq_raw depending on the
IRQ to confirm
*   ulMask     [in] Interrupt mask to confirm
* \return
*                                                                              */
/*****************************************************************************/
void NX50_GPIO_ConfirmIrq( unsigned long ulConfirm, unsigned long ulMask )
{
  unsigned int  uiIrq;
  int           iIdx;

  /* Make these variables static so they can be placed in fast memory by linker.
     Sharing / Race conditions are no problem as we are only using them inside
     IRQ locked code */
  static unsigned long aulTimerValues[5] = {0};
  static unsigned long aulTimerCtrl[5]   = {0};
  static unsigned long ulSystimeNs;
  unsigned long        ulSoftIntMask     = 0;
  unsigned long        ulTimerIrqMask    = 0;

  NX50_lock_irqfiq_save(uiIrq);

  if( Adr_NX50_gpio_cnt_irq_raw == ulConfirm )
  {
    /* On Timer confirmation we need to clear software interrupts on VIC */
    unsigned    long    ulSoftIntConfirm    =    (((ulMask    &    MSK_GPIO_COUNTER0_2)    >>
SRT_NX50_cnt_irq_raw_cnt0) << SRT_NX50_vic_softint_timer0) |
                                    (((ulMask        &        MSK_GPIO_COUNTER3_4)        >>
SRT_NX50_cnt_irq_raw_cnt3) << SRT_NX50_vic_softint_timer3) |
                                    (((ulMask     &     MSK_NX50_cnt_irq_raw_sys_time)     >>
SRT_NX50_cnt_irq_raw_sys_time) << SRT_NX50_vic_softint_systime_ns);
```

```
    POKE( Adr_NX50_vic_vic_softint_clear, ulSoftIntConfirm );
  }

  /* Store all current timers */
  for( iIdx = 0; iIdx < sizeof(s_ptGpio->aulGpio_counter_cnt) / sizeof(s_ptGpio->aulGpio_counter_cnt[0]); ++iIdx)
  {
    aulTimerValues[iIdx] = s_ptGpio->aulGpio_counter_cnt[iIdx];
    aulTimerCtrl[iIdx]   = s_ptGpio->aulGpio_counter_ctrl[iIdx];
  }

  /* Save Systime value */
  PEEK(Adr_NX50_systime_systime_s);
  ulSystimeNs = PEEK(Adr_NX50_systime_systime_ns);

  POKE( ulConfirm, ulMask );

  ulTimerIrqMask = s_ptGpio->ulCnt_irq_mask_set;

  /* Verify system against gpio */
  if( ulTimerIrqMask  & MSK_NX50_cnt_irq_masked_sys_time )
  {
    unsigned long ulNewSystimeNs;
    unsigned long ulSystimeCmp   = s_ptGpio->ulGpio_systime_cmp;

    PEEK(Adr_NX50_systime_systime_s);
    ulNewSystimeNs = PEEK(Adr_NX50_systime_systime_ns);

    if( ulNewSystimeNs > ulSystimeNs )
    {
      if( (ulSystimeNs < ulSystimeCmp) && (ulNewSystimeNs > ulSystimeCmp) )
        ulSoftIntMask |= MSK_NX50_vic_softint_systime_ns;
    }
    else
    {
      if( (ulSystimeNs < ulSystimeCmp) || (ulNewSystimeNs > ulSystimeCmp) )
        ulSoftIntMask |= MSK_NX50_vic_softint_systime_ns;
    }
  }

  /* Check for timers that may have caused an IRQ during our IRQ acknowledge */
  for( iIdx = 0; iIdx < sizeof(s_ptGpio->aulGpio_counter_cnt) / sizeof(s_ptGpio->aulGpio_counter_cnt[0]); ++iIdx )
  {
    /* Timer0-2 are continues from Bit1-3, 3-4 are on Bits 29/30 */
```

```
    unsigned long ulTempIrqMsk = (iIdx < 3) ?
                                 MSK_NX50_vic_softint_timer0 << iIdx :
                                 MSK_NX50_vic_softint_timer3 << (iIdx - 3);

    /* Only process timers that are supposed to generate an IRQ and were running before */
    if( (ulTimerIrqMask & (1 << iIdx))                          &&
        (aulTimerCtrl[iIdx] & MSK_NX50_gpio_counter0_ctrl_irq_en) &&
        (aulTimerCtrl[iIdx] & MSK_NX50_gpio_counter0_ctrl_run) )
    {
      if( 0 == (aulTimerCtrl[iIdx] & MSK_NX50_gpio_counter0_ctrl_once) )
      {
        /* This is a cyclic timer, so we just need to check if the new value is
           smaller than the old one and generate an IRQ if neccessary */
        if( s_ptGpio->aulGpio_counter_cnt[iIdx] < aulTimerValues[iIdx] )
          ulSoftIntMask |= ulTempIrqMsk;

      } else
      {
        /* Single shot timer */
        if( 0 == (s_ptGpio->aulGpio_counter_ctrl[iIdx] & MSK_NX50_gpio_counter0_ctrl_run) )
        {
          /* Timer was running at start of Confirmation and has stopped now so this timer has
expired */
          ulSoftIntMask |= ulTempIrqMsk;
        }
      }
    }
  }

  /* Generate software interrupt for all counters that expired. This may result
     in "normal" AND software IRQ being set. Make sure the interrupt handlers clears both.
     Software interrupts will be cleared when calling netX50HandleTimerIrq */
  POKE( Adr_NX50_vic_vic_softint, ulSoftIntMask );

  NX50_lock_irqfiq_restore(uiIrq);
}
```

13 Internal PHYs: Error in 10 MBit half duplex mode

Issue:

When using standard Ethernet Communication with 10 MBit half duplex mode (used with legacy hubs only), the complete PHY gets stuck in case of a network collision.

As far as Real-Time-Ethernet protocols are concerned, this problem can only occur with EtherNet/IP or Modbus TCP when using hubs instead of switches (recommended), as all other protocols do not allow the use of hubs or 10 MBit mode.

For error details see attached problem description from Renesas at the end of this document.

Solution / Workaround:

Do not use legacy 10 MBit-only hubs. Use either switches or 10/100 MBit Dual Speed hubs, to make sure the netX Ethernet ports are connected with 100 Mbit or in full duplex mode only.

This erratum is fixed with all components of the 'Y' charge (9 digit charge number shows 'Y' at position 5 (nnnn**Y**nnnn).

14 Host Interface: DPM access time with Hilscher standard DPM layout is unpredictable

Issue:

When using the host interface in DPM mode along with the Hilscher standard DPM layout (or any other DPM layout providing access to the Handshake cell area), DPM access time may increase unpredictably due to a possible race condition between the host interface and other internal controllers (xPEC, ARM). As a result, all hardware designs that work with host CPUs which do not use or support the DPM RDY/BUSY signal but use DPM layouts that allow access to handshake cells, are subject to DPM access errors, resulting in wrong data read from the DPM or DPM write accesses being ignored. Such access errors occur, whenever the actual DPM access time exceeds the fixed access time used by the host CPU. Increasing the fixed access time on the host CPU is not a solution, since a maximum DPM access time can not be specified here.

Solution / Workaround:

a) Use a host CPU that supports the RDY/BUSY signal provided by the DPM to allow extending DPM access cycles if necessary.

OR

b) Do not use DPM layouts that allow access to the handshake cells or any other memory area except the internal RAM segments (maximum DPM access time to internal RAM areas is always deterministic).
   → Using standard loadable firmware provided by Hilscher along with a host CPU that does not support the RDY/BUSY signal is not possible!