ES_LPC11U3x

Errata sheet LPC11U3x

Rev. 3.2 — 7 March 2018 **Errata sheet**

Document information

Info	Content
Keywords	LPC11U34FHN33; LPC11U34FBD48; LPC11U34FHN33; LPC11U35FHN33; LPC11U35FBD48; LPC11U35FBD64; LPC11U35FHI33; LPC11U35FET48; LPC11U36FBD48; LPC11U36FBD64; LPC11U37FBD48; LPC11U37FBD64; LPC11U37HFBD64; LPC11U3x errata
Abstract	This errata sheet describes both the known functional problems and any deviations from the electrical specifications known at the release date of this document. Each deviation is assigned a number and its history is tracked in a table.

Revision history

Rev	Date	Description
3.2	20180307	Added USB_ROM.3.
3.1	20170804	Added USB_ROM.2.
3	20140808	Added USB_ROM.1.
2.1	20140123	Added LPC11U37HFBD64.
2	20130117	Added I2C.1.
1	20120601	Initial version

Contact information

For more information, please visit: **http://www.nxp.com**

For sales office addresses, please send an email to: **salesaddresses@nxp.com**

1. Product identification

The LPC11U3x devices typically have the following top-side marking:

LPC11U3x

/xxx

xxxxxxx

xxYYWWxR[x]

The last letter in the last line (field 'R') will identify the device revision. This Errata Sheet covers the following revisions of the LPC11U3x:

Table 1. Device revision table

Revision identifier (R)	Revision description
'A'	Initial device revision

Field 'YY' states the year the device was manufactured. Field 'WW' states the week the device was manufactured during that year.

2. Errata overview

Table 2. Errata summary table

Functional problems	Short description	Revision identifier	Detailed description
ADC.1	A/D Global Data register should not be used with burst mode or hardware triggering.	'A'	Section 3.1
I2C.1	In the slave-transmitter mode, the device set in the monitor mode must write a dummy value of 0xFF into the DAT register.	'A'	Section 3.2
USB_ROM.1	The USB ROM driver routine hwUSB_ResetEP() accidentally corrupts the subsequent word of memory while clearing the STALL bit of the selected endpoint.	'A'	Section 3.3
USB_ROM.2	FRAME_INT is cleared if new SetConfiguration or USB_RESET are received.	'A'	Section 3.4
USB.ROM.3	USB full-speed device fail in the Command/Data/Status Flow after bus reset and bus re-enumeration.	'A'	Section 3.5

Table 3. AC/DC deviations table

AC/DC deviations	Short description	Revision identifier	Detailed description
n/a	n/a	n/a	n/a

Table 4. Errata notes table

Errata notes	Short description	Revision identifier	Detailed description
Note.1	During power-up, an unexpected glitch (low pulse) could occur on the port pins as the V_{DD} supply ramps up.	'A'	Section 5.1

3. Functional problems detail

3.1 ADC.1

A/D Global Data register should not be used with burst mode or hardware triggering.

Introduction:

On the LPC11U3x, the START field and the BURST bit in the A/D control register specify whether A/D conversions are initiated via software command, in response to some hardware trigger, or continuously in burst ("hardware-scan") mode. Results of the ADC conversions can be read in one of two ways. One is to use the A/D Global Data Register to read all data from the ADC. Another is to use the individual A/D Channel Data Registers.

Problem:

If the burst mode is enabled (BURST bit set to ‘1’) or if hardware triggering is specified, the A/D conversion results read from the A/D Global Data register could be incorrect. If conversions are only launched directly by software command (BURST bit = '0' and START = ‘001’), the results read from the A/D Global Data register will be correct provided the previous result is read prior to launching a new conversion.

Work-around:

When using either burst mode or hardware triggering, the individual A/D Channel Data registers should be used instead of the A/D Global Data register to read the A/D conversion results.

3.2 I2C.1

In the slave-transmitter mode, the device set in the monitor mode must write a dummy value of 0xFF into the DAT register.

Introduction:

The I2C monitor allows the device to monitor the I2C traffic on the I2C bus in a non-intrusive way.

Problem:

In the slave-transmitter mode, the device set in the monitor mode must write a dummy value of 0xFF into the DAT register. If this is not done, the received data from the slave device will be corrupted. To allow the monitor mode to have sufficient time to process the data on the I2C bus, the device may need to have the ability to stretch the I2C clock. Under this condition, the I2C monitor mode is not 100% non-intrusive.

Work-around:

When setting the device in monitor mode, enable the ENA_SCL bit in the MMCTRL register to allow clock stretching.

Software code example to enable the ENA_SCL bit:

```
LPC_I2C_MMCTRL |= (1<<1); //Enable ENA_SCL bit
```

In the I2C ISR routine, for the status code related to the slave-transmitter mode, write the value of 0xFF into the DAT register to prevent data corruption. In order to avoid stretching the SCL clock, the data byte can be saved in a buffer and processed in the Main loop. This ensures the SI flag is cleared as fast as possible.

Software code example for the slave-transmitter mode:

```
case 0xA8:     // Own SLA + R has been received, ACK returned
case 0xB0:
case 0xB8:     // data byte in DAT transmitted, ACK received
case 0xC0:     // (last) data byte transmitted, NACK received
case 0xC8:     // last data byte in DAT transmitted, ACK received
    DataByte = LPC_I2C->DATA_BUFFER;//Save data. Data can be process in Main loop
    LPC_I2C->DAT = 0xFF;            // Pretend to shift out 0xFF
    LPC_I2C->CONCLR = 0x08;         // clear flag SI
break;
```

3.3 USB_ROM.1

The USB ROM driver routine hwUSB_ResetEP() accidentally corrupts the subsequent word of memory while clearing the STALL bit of the selected endpoint.

Introduction:

The on-chip USB2.0 full-speed device controller uses the USB endpoint (EP) Command/Status List organized in memory to store the EPs command/status information. Bit 29 indicates the STALL status of the corresponding EP. The USB ROM driver routine hwUSB_ResetEP(), which is called during SET_CONFIGURATION and

SET_INTERFACE requests for all EPs present in the corresponding configuration/interface, clears the STALL bit of the selected EPs in Command/Status List as part of EP reset procedure.

Problem:

During the EP reset procedure executed by the USB ROM driver routine hwUSB_ResetEP(), it not only clears the STALL bit of the selected EP but also corrupts the subsequent word of memory. This issue is caused by a software bug in the hwUSB_ResetEP() routine.

Below is a summary of the runtime errors resulting from this issue:

- Case 1. When reset procedure is invoked on an EP which is at the end of the EP list, this bug will accidentally corrupt the memory area following the EP Command/Status List. In the current version of USB ROM driver this area is used for storing the receiver buffer address for control endpoint (EP0). This corruption causes erratic behavior on control OUT transaction.
- Case 2. When reset procedure is invoked on an EP which is in the beginning or middle of the EP list, this bug will accidentally clear the STALL bit of the subsequent EP in list.
  - If hwUSB_ResetEP() is called during SET_CONFIGURATION, clearing the STALL bit of the subsequent EP has no consequence since STALL condition is cleared for all EPs during SET_CONFIGURATION procedure.
  - If hwUSB_ResetEP() is called during SET_INTERFACE when selecting an ALT interface, this issue could clear STALL condition (if exists) on the subsequent EP. This condition is very rare.

Work-around:

The software work-around to address Case 1 is to specify one extra EP in the max_num_ep field of the USBD_API_INIT_PARAM_T structure passed to the ROM driver's hw->init() routine. This extra EP provides a padding buffer to avoid corruption to the subsequent word of memory. This workaround is demonstrated with the line of code highlighted in red in function usb_init() in the following example.

If your system is affected with Case 2, user should check the "ep_halt" member of USB_CORE_CTRL_T structure in the SET_INTERFACE event and set STALL bit for any EP which got cleared due to this bug. This condition is very rare. This workaround is demonstrated with the function StallWorkAround () in the following example. Notice that StallWorkAround is set to be an interface event in the usb_init() function (highlighted in bold).

```
typedef volatile struct _EP_LIST {
  uint32_t  buf_ptr;
  uint32_t  buf_length;
} EP_LIST;

ErrorCode_t StallWorkAround(USBD_HANDLE_T hUsb)
{
      ErrorCode_t ret = LPC_OK;
      USB_CORE_CTRL_T *pCtrl = (USB_CORE_CTRL_T *) hUsb;
      EP_LIST *epQueue;
```

```
      int32_t i;

      /*    WORKAROUND for Case 2:
      Code clearing STALL bits in endpoint reset routine corrupts memory area
          next to the endpoint control data.
      */
      if (pCtrl->ep_halt != 0) { /* check if STALL is set for any endpoint */
            /* get pointer to HW EP queue */
            epQueue = (EP_LIST *) LPC_USB->EPLISTSTART;
            /* check if the HW STALL bit for the endpoint is cleared due to bug. */
            for (i = 1; i < pCtrl->max_num_ep; i++) {
                  /* check OUT EPs */
                  if ( pCtrl->ep_halt & (1 << i)) {
                        /* Check if HW EP queue also has STALL bit = _BIT(29) is set */
                        if (( epQueue[i << 1].buf_ptr & _BIT(29)) == 0) {
                              /* bit not set, cleared by BUG. So set it back. */
                              epQueue[i << 1].buf.ptr |= _BIT(29);
                        }
                  }
                  /* Check IN EPs */
                  if ( pCtrl->ep_halt & (1 << (i + 16))) {
                        /* Check if HW EP queue also has STALL bit = _BIT(29) is set */
                        if (( epQueue[(i << 1) + 1].buf_ptr & _BIT(29)) == 0) {
                              /* bit not set, cleared by BUG. So set it back. */
                              epQueue[(i << 1) + 1].buf_ptr |= _BIT(29);
                        }
                  }
            }
      }

      return ret;
}

/* Initialize USB sub system */
static ErrorCode_t usbd_init(void)
{
      USBD_API_INIT_PARAM_T usb_param;
      USB_CORE_DESCS_T desc;
      ADC_INIT_PARAM_T adc_param;
      ErrorCode_t ret = LPC_OK;

      /* enable clocks and pinmux */
      usb_pin_clk_init();

      /* initialize USBD ROM API pointer. */
      g_pUsbApi = (const USBD_API_T *) LPC_ROM_API->usbdApiBase;
      /* initialize call back structures */
      memset((void *) &usb_param, 0, sizeof(USBD_API_INIT_PARAM_T));
      usb_param.usb_reg_base = LPC_USB0_BASE;
      /*  WORKAROUND for Case 1
```

```
        For example When EP0, EP1_IN, EP1_OUT and EP2_IN are used we need to specify
        usb_param.max_num_ep as 3 here. But as a workaround for this issue specify
        usb_param.max_num_ep as 4. So that extra EPs control structure acts as padding
        buffer to avoid data corruption. Corruption of padding memory doesn't affect the
        stack/program behavior.
        */
        usb_param.max_num_ep = 3 + 1;
        usb_param.USB_Interface_Event = StallWorkAround;

        usb_param.mem_base = USB_STACK_MEM_BASE;
        usb_param.mem_size = USB_STACK_MEM_SIZE;

        /* Set the USB descriptors */
        desc.device_desc = (uint8_t *) &USB_DeviceDescriptor[0];
        desc.string_desc = (uint8_t *) &USB_StringDescriptor[0];
        /* Note, to pass USBCV test full-speed only devices should have both
           descriptor arrays point to same location and device_qualifier set to 0.
        */
        desc.high_speed_desc = (uint8_t *) &USB_FsConfigDescriptor[0];
        desc.full_speed_desc = (uint8_t *) &USB_FsConfigDescriptor[0];
        desc.device_qualifier = 0;

        /* USB Initialization */
        ret = USBD_API->hw->Init(&g_hUsb, &desc, &usb_param);
        if (ret == LPC_OK) {
}
```

3.4 USB_ROM.2: FRAME_INT is cleared if new SetConfiguration or USB_RESET are received.

Introduction:

In the USB ROM API, the function call EnableEvent can be used to enable and disable FRAME_INT.

Problem:

When the FRAME_INT is enabled through the USB ROM API call:

```
ErrorCode_t(* USBD_HW_API::EnableEvent)(USBD_HANDLE_T hUsb, uint32_t EPNum, uint32_t
     event_type, uint32_t enable),
```

the FRAME_INT is cleared if new SetConfiguration or USB_RESET are received.

Work-around:

Implement the following software work-around in the ISR to ensure that the FRAME_INT is enabled:

```
void USB_IRQHandler(void)
{
USBD_API->hw->EnableEvent(g_hUsb, 0, USB_EVT_SOF, 1);
USBD_API->hw->ISR(g_hUsb);
}
```

3.5 USB_ROM.3: USB full-speed device fail in the Command/Data/Status Flow after bus reset and bus re-enumeration

Introduction:

The LPC11U3x device family includes a USB full-speed interface that can operate in device mode and also, includes USB ROM based drivers. A Bulk-Only Protocol transaction begins with the host sending a CBW to the device and attempting to make the appropriate data transfer (In, Out or none). The device receives the CBW, checks and interprets it, attempts to satisfy the request of the host, and returns status via a CSW.

Problem:

When the device fails in the Command/Data/Status Flow, and the host does a bus reset / bus re-enumeration without issuing a Bulk-Only Mass Storage Reset, the USB ROM driver does not re-initialize the MSC variables. This causes the device to fail in the Command/Data/Status Flow after the bus reset / bus re-enumeration.

Work-around:

Implement the following software work-around to re-initialize the MSC variables in the USBD stack.

```
void *g_pMscCtrl;

ErrorCode_t mwMSC_Reset_workaround(USBD_HANDLE_T hUsb)

{

((USB_MSC_CTRL_T *)g_pMscCtrl)->CSW.dSignature = 0;

     ((USB_MSC_CTRL_T *)g_pMscCtrl)->BulkStage = 0;

     return LPC_OK;

}

ErrorCode_t mscDisk_init(USBD_HANDLE_T hUsb, USB_CORE_DESCS_T *pDesc,
     USBD_API_INIT_PARAM_T *pUsbParam)

{    USBD_MSC_INIT_PARAM_T msc_param;

     ErrorCode_t ret = LPC_OK;

     memset((void *) &msc_param, 0, sizeof(USBD_MSC_INIT_PARAM_T));

     msc_param.mem_base = pUsbParam->mem_base;

     msc_param.mem_size = pUsbParam->mem_size;

     g_pMscCtrl = (void *)msc_param.mem_base;

     ret = USBD_API->msc->init(hUsb, &msc_param);

     /* update memory variables */

     pUsbParam->mem_base = msc_param.mem_base;

     pUsbParam->mem_size = msc_param.mem_size;
```

```
    return ret;
}

    usb_param.USB_Reset_Event = mwMSC_Reset_workaround;

    ret = USBD_API->hw->Init(&g_hUsb, &desc, &usb_param);
```

4. AC/DC deviations detail

4.1 n/a

5. Errata notes detail

5.1 Note.1

The General Purpose I/O (GPIO) pins have configurable pull-up/pull-down resistors where the pins are pulled up to the VDD level by default. During power-up, an unexpected glitch (low pulse) could occur on the port pins as the VDD supply ramps up.

6. Legal information

6.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

6.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the *Terms and conditions of commercial sale* of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

6.3 Trademarks

Notice: All referenced brands, product names, service names and trademarks are the property of their respective owners.

7. Contents

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 7 March 2018
Document identifier: ES_LPC11U3X