ES_LPC11U6x

Errata sheet LPC11U6x

Rev. 1.4 — 7 March 2018 Errata sheet

Document information

Info	Content
Keywords	LPC11U66JBD48; LPC11U67JBD48; LPC11U67JBD64; LPC11U67JBD100; LPC11U68JBD48; LPC11U68JBD64; LPC11U68JBD100; LPC11U6x 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
1.4	20180307	• USB_ROM.4.
1.3	20170804	• USB_ROM.3.
1.2	20151022	• Added UART.1. • Added USB.1.
1.1	20140728	• Corrected USB_ROM.1 work-around. • Corrected part marking information. • Parts added: LPC11U67JBD100, LPC11U67JBD64, LPC11U66JBD48.
1	20140115	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 LPC11U6x devices typically have the following top-side marking for LQFP100 packages:

LPC11U6xJBD100

xxxxxx xx

xxxyywwxR[x]

The LPC11U6x devices typically have the following top-side marking for LQFP64 packages:

LPC11U6xJ

xxxxxx xx

xxxyywwxR[x]

The LPC11U6x devices typically have the following top-side marking for LQFP48 packages:

LPC11U6xJ

xx xx

xxxyy

wwxR[x]

Field ‘yy’ states the year the device was manufactured. Field ‘ww’ states the week the device was manufactured during that year.

Field ‘R’ identifies the device revision. This Errata Sheet covers the following revisions of the LPC11U6x:

Table 1. Device revision table

Revision identifier (R)	Revision description
‘A’	Initial device revision

2. Errata overview

Table 2. Errata summary table

Functional problems	Short description	Revision identifier	Detailed description
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 3.1
USB_ROM.2	The USBD ROM stack does not split EP0 transfer into multiple packets of 8 bytes (MAXP allowed) in low speed mode.	‘A’	Section 3.2
USB_ROM.3	FRAME_INT is cleared if new SetConfiguration or USB_RESET are received.	‘A’	Section 3.3

Table 2. Errata summary table *…continued …continued*

Functional problems	Short description	Revision identifier	Detailed description
USB_ROM.4	USB full-speed device fail in the Command/Data/Status Flow after bus reset and bus re-enumeration.	'A'	Section 3.4
USB.1	The USB controller is unable to generate STALL on EP0_OUT.	'A'	Section 3.5
UART.1	The UART controller sets the Idle status bits for receive and transmit before the transmission of the stop bit is complete.	'A'	Section 3.6

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
n/a	n/a	n/a	n/a

3. Functional problems detail

3.1 USB_ROM.1

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) {
}
```

3.2 USB_ROM.2

Introduction:

When USB device operates in low-speed mode the maximum packet length (MAXP) for control transfer and interrupt transfers is restricted to 8 bytes. Hence when more than 8 bytes needs to be transferred, the data should be split into multiple 8 byte packets. But the current ROM stack splits the control transfer into multiples of 64 bytes only.

Problem:

Device will not enumerate when used in low-speed mode.

Work-around:

The software work-around for this issue is to override the cases where the ROM stack would queue a large transfer and split them into smaller 8 byte packet transfers. Since low speed USB allows only interrupt endpoints, a workaround for HID class implementation is shown below:

```
static ErrorCode_t HID_LowSpeedPatch(USBD_HANDLE_T hUsb, void *data, uint32_t event)
{
     USB_CORE_CTRL_T *pCtrl = (USB_CORE_CTRL_T *) hUsb;
     USB_HID_CTRL_T *pHidCtrl = (USB_HID_CTRL_T *) data;
     ErrorCode_t ret = ERR_USBD_UNHANDLED;
     uint16_t cnt = 0, len = 0;

     switch (event) {
     case USB_EVT_SETUP:
          if (pCtrl->SetupPacket.bmRequestType.BM.Type == REQUEST_STANDARD) {

               switch (pCtrl->SetupPacket.bRequest) {
               case USB_REQUEST_GET_DESCRIPTOR:
                    /* handle HID descriptors first */
                    switch (pCtrl->SetupPacket.wValue.WB.H) {
                    case HID_HID_DESCRIPTOR_TYPE:
                         pCtrl->EP0Data.pData = pHidCtrl->hid_desc;
                         len = ((USB_COMMON_DESCRIPTOR *)
                         pHidCtrl->hid_desc)->bLength;
                         ret = LPC_OK;
                         break;

                    case HID_REPORT_DESCRIPTOR_TYPE:
                         ret = pHidCtrl->HID_GetReportDesc(pHidCtrl,
                         &pCtrl->SetupPacket,
                         &pCtrl->EP0Data.pData, &len);
                         break;

                    case HID_PHYSICAL_DESCRIPTOR_TYPE:
                         if (pHidCtrl->HID_GetPhysDesc == 0) {
                         ret = (ERR_USBD_STALL); /* HID Physical Descriptor is not
                         supported */
                         }
                         else {
```

```
                                ret = pHidCtrl->HID_GetPhysDesc(pHidCtrl,
                                &pCtrl->SetupPacket,&pCtrl->EP0Data.pData, &len);
                            }
                            break;

                        default:
                            ret = pCtrl->USB_ReqGetDescriptor(pCtrl);
                            break;
                        }
                        break;

                    case USB_REQUEST_GET_CONFIGURATION:
                        ret = pCtrl->USB_ReqGetConfiguration(pCtrl);
                        break;

                    case USB_REQUEST_GET_INTERFACE:
                        ret = pCtrl->USB_ReqGetInterface(pCtrl);
                        break;

                    default:
                        break;
                    }
                }
                else if ((pCtrl->SetupPacket.bmRequestType.BM.Type == REQUEST_CLASS) &&
                        (pCtrl->SetupPacket.bmRequestType.BM.Recipient ==
REQUEST_TO_INTERFACE) &&
                        pCtrl->SetupPacket.bRequest == HID_REQUEST_GET_REPORT) ) {

                    pCtrl->EP0Data.pData = pCtrl->EP0Buf; /* point to data to be sent */
                    /* allow user to copy data to EP0Buf or change the pointer to his own
                    buffer */
                    ret = pHidCtrl->HID_GetReport(pHidCtrl, &pCtrl->SetupPacket,
                        &pCtrl->EP0Data.pData, &pCtrl->EP0Data.Count);
                }
                break;

            case USB_EVT_IN:
                if (pCtrl->SetupPacket.bmRequestType.BM.Dir == REQUEST_DEVICE_TO_HOST) {
                    ret = LPC_OK;
                }
                break;

            }
            if (ret == LPC_OK) {
                if ((len != 0) && (pCtrl->EP0Data.Count > len)) {
                    pCtrl->EP0Data.Count = len;
                }
                cnt = (pCtrl->EP0Data.Count > USB_MAX_PACKET0) ? USB_MAX_PACKET0 :
                pCtrl->EP0Data.Count;
                cnt = USBD_API->hw->WriteEP(pCtrl, 0x80, pCtrl->EP0Data.pData, cnt);
                pCtrl->EP0Data.pData += cnt;
```

```
          pCtrl->EP0Data.Count -= cnt;
     }
     else if (ret == ERR_USBD_UNHANDLED) {
          ret = g_defaultHidHdlr(hUsb, data, event);
     }

     return ret;
}
```

To install this patch handler do the following:

1. declare a global variable: `static USB_EP_HANDLER_T g_defaultHidHdlr;`
2. install the override handler during initialization phase:

```
     ret = USBD_API->hid->init(hUsb, &hid_param);
     if (ret == LPC_OK) {
          g_defaultHidHdlr = pCtrl->ep0_hdlr_cb[pCtrl->num_ep0_hdlrs - 1];
          /* store the default CDC handler and replace it with ours */
          pCtrl->ep0_hdlr_cb[pCtrl->num_ep0_hdlrs - 1] = HID_LowSpeedPatch;
     ....
     }
```

3.3 USB_ROM.3: 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);
}
```

3.4 USB_ROM.4: USB full-speed device fail in the Command/Data/Status Flow after bus reset and bus re-enumeration

Introduction:

The LPC11U6x 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);
```

3.5 USB.1: USB controller is unable to generate STALL on EP0_OUT

Introduction:

The LPC11U6x have a full-speed USB device controller with support for 10 physical endpoints.

Problem:

The USB device controller is unable to return a STALL handshake on an OUT data packet to endpoint zero. An NAK handshake is returned instead.

Work-around:

Endpoint zero is the control endpoint. All requests sent to the control endpoint consist of three stages (SETUP / DATA / STATUS). When an unsupported ControlWrite request (with data phase) is sent by the host to the device, the device is unable to STALL the data phase of this request.

To solve this problem, the device firmware must accept the data transmitted during the data phase of this ControlWrite request and return a STALL handshake when the IN token for the STATUS stage is received.

3.6 UART.1

Introduction:

In receive mode, the UART controller provides a status bit (the RXIDLE bit in the UART STAT register) to check whether the receiver is currently receiving data. If RXIDLE is set, the receiver indicates it is idle and does not receive data.

In transmit mode, the UART controller provides two status bits (TXIDLE and TXDISSTAT bits in the UART STAT register) to indicate whether the transmitter is currently transmitting data. The TXIDLE bit is set by the controller after the last stop bit has been transmitted. The TXDISSTAT bit is set by the controller after the transmitter has sent the last stop bit and has become fully idle following a transmit disable executed by setting the TXDIS bit in the UART CTRL register.

The status bits can be used to implement software flow control, but their setting does not affect normal UART operation.

Problem:

The RXIDLE bit is incorrectly set for a fraction of the clock cycle between the reception of the last data bit and the reception of the start bit of the next word, that is while the stop bit is received. RXIDLE is cleared at the beginning of the start bit.

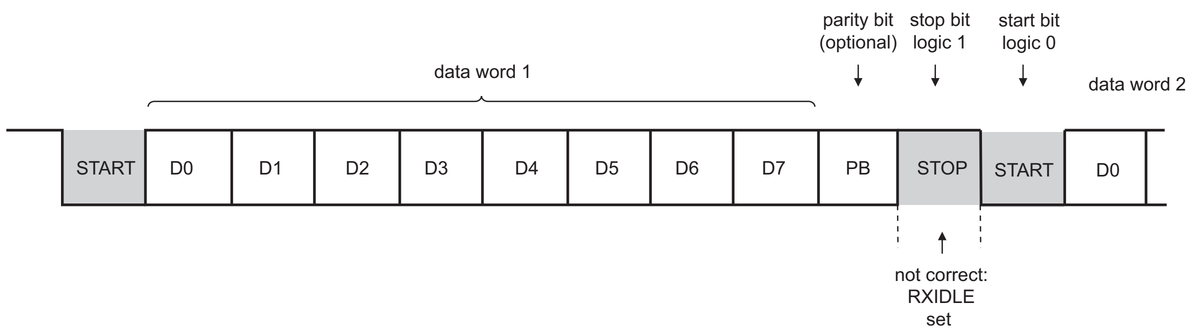

Fig 1. Incorrect setting of RXIDLE during UART receive

Both, TXIDLE and TXDISSTAT are set incorrectly between the last data bit and the stop bit while the transfer is still ongoing.

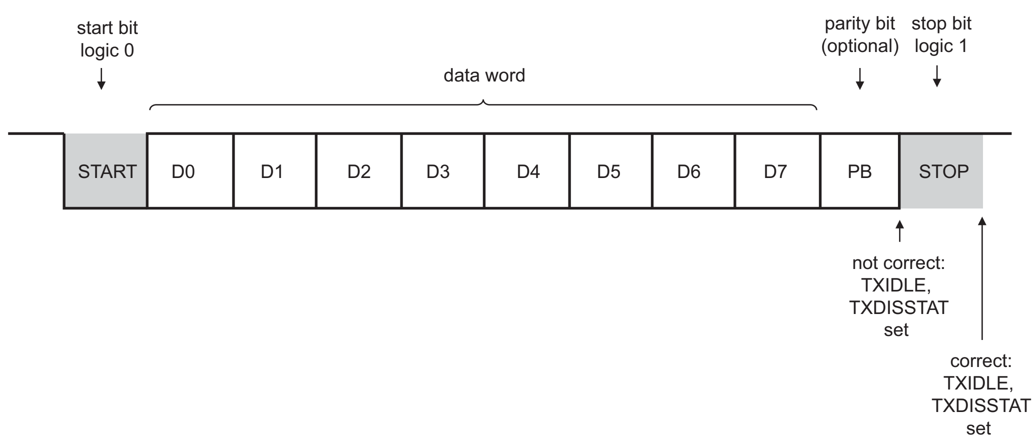

Fig 2. Incorrect setting of TXIDLE and TXDISSTAT during UART transmit

Work-around:

When writing code that checks for the setting of any of the status bits RXIDLE, TXIDLE, TXDISSTAT, check the value of the status bit in the STAT register:

- If status bit = 1, add a delay of one UART bit time (if STOPLEN = 0, one stop bit) or two bit times (if STOPLEN = 1, two stop bits) and check the value of the status bit again:
  - If status bit = 1, the receiver is idle.
  - If status bit = 0, the receiver is receiving data.
- If the status bit = 0, the receiver is receiving data.

4. Legal information

4.1 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.

4.2 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.

4.3 Trademarks

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

5. Contents

1 Product identification 3
2 Errata overview 3
3 Functional problems detail 5
3.1 USB_ROM.1 5
3.2 USB_ROM.2 8
3.3 USB_ROM.3: FRAME_INT is cleared if new SetConfiguration or USB_RESET are received. 11
3.4 USB_ROM.4: USB full-speed device fail in the Command/Data/Status Flow after bus reset and bus re-enumeration 12
3.5 USB.1: USB controller is unable to generate STALL on EP0_OUT. 14
3.6 UART.1 15
4 Legal information. 17
4.1 Definitions. 17
4.2 Disclaimers 17
4.3 Trademarks. 17
5 Contents 18

Date of release: 7 March 2018
Document identifier: ES_LPC11U6X