Freescale Semiconductor
Application Note

Document Number: AN3683
Rev. 1, 11/2009

Workaround for i.MX25 USB OTG Erratum BID2108

by *Multimedia and Applications Division*
Freescale Semiconductor, Inc.
Austin, TX

This document describes the USB OTG BID2108 errata of the i.MX25 device. It summarizes the errata and describes the workaround.

The errata occurs because CRC and long packets errors can be generated when completing a transfer that has a maximum size of one or two bytes. A software workaround is available if no more than four TX endpoints (including endpoint 0) are needed. A software workaround may be available if more than four TX endpoints (including endpoint 0) are needed depending on how many TX endpoints have the ability to send one or two byte packets.

Contents

1 Errata Description

The issue with the USB core is that in the case of one or two byte packets, one read strobe to fetch the EOP Tag of the FIFO memory is missed. Therefore, the USB logic uses whatever is on the output of the memory for the read action.

The following cases may trigger a new FIFO read cycle:

- An IN token is sent to an endpoint even if the device address is not equal to current device address
- An OUT token is sent to an endpoint even if the device address is not equal to current device address
- A SOF packet, bit 11–bit 8 of the Frame_number are treated as the endpoint number
- Software primes a new IN endpoint

As shown in Figure 1, the transmit FIFO memory consists of four physical blocks of memory (Mem_A, Mem_B, Mem_C, and Mem_D). Each block serves two endpoints:

- Mem_A for endpoints 0 and 1
- Mem_B for endpoints 2 and 3
- Mem_C for endpoints 4 and 5
- Mem_D for endpoints 6 and 7

A new FIFO read to endpoint 0 or 1 only causes the output of Mem_A to change and not the other three blocks, Mem_B, Mem_C or Mem_D.

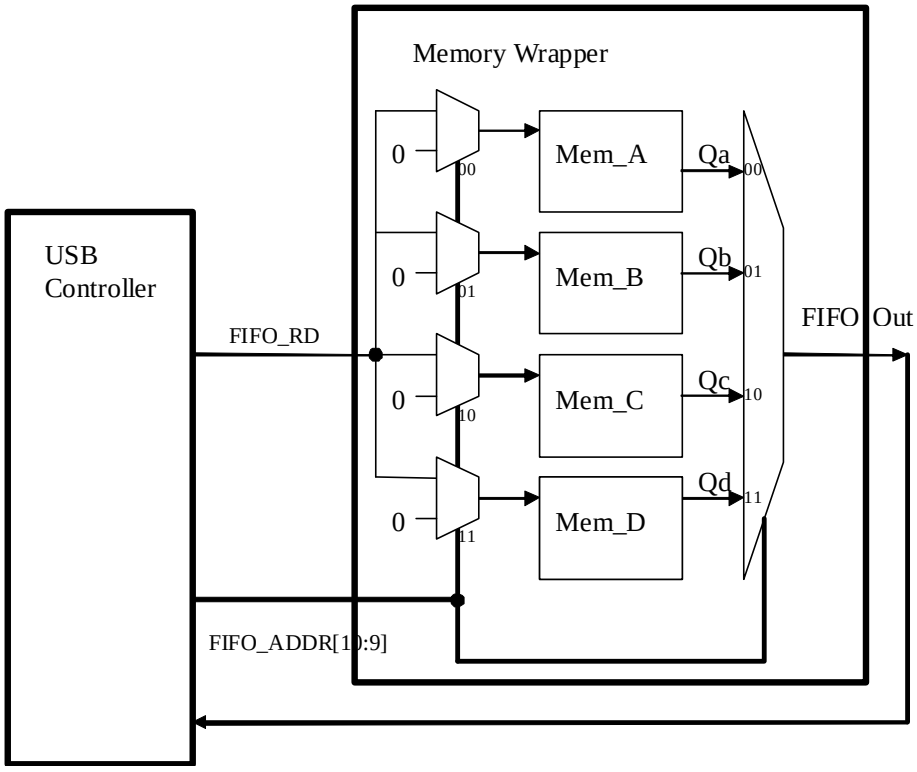

Figure 1. Transmit FIFO Memory Diagram

2 Workaround Description

This section describes the two steps of the workaround for this errata.

NOTE

The following software workaround becomes invalid after software sets either the USBCMD.RST bit or the ENDPTFLUSH bit for the corresponding TX endpoint. Therefore the following software workaround must be implemented again after software sets either the USBCMD.RST or ENDPTFLUSH bit.

2.1 Allocate Endpoints

Allocate the endpoint numbers to avoid one endpoint that may send one or two bytes packets to different memory blocks and this endpoint cannot share the memory block with any other endpoints. There are two cases:

- If no more than four TX endpoints (including endpoint 0) are needed, software allocates the endpoints to different memory blocks. To simplify the process, use even endpoints (endpoint 0, endpoint 2, endpoint 4 and endpoint 6) for IN transactions.
- If more than four TX endpoints (including endpoint 0) are needed, check how many TX endpoints have the possibility to send one or two byte packets. Each endpoint that may have one or two byte packets should occupy one full memory block. Every two of other endpoints which do not have the possibility to send one or two byte packets can share one full memory block. For example, five TX endpoints are needed, three of them have the possibility to send one or two byte packets and the other two endpoints do not. Software allocates endpoint 0, endpoint 2, endpoint 4 for the three endpoints which may have one or two byte packets, and allocates endpoint 6 and endpoint 7 for the two endpoints which do not send one or two byte packets.

There is no workaround if more than four (greater than or equal to five) TX endpoints (including endpoint 0) are needed, and more than three (greater than or equal to four) of them have the possibility to send one or two byte packets.

2.2 Prime Endpoints

To avoid the uncontrollable data output which is generated by the redundant read cycle for another endpoint which uses the same memory block with the one or two byte packet endpoint, the data output of these endpoints should always be keep to EOP Tag.

Software needs to do following to make sure the data output of these endpoints always keep to EOP tag:

- If EP0 has the possibility to send one or two bytes packet, software should prime EP1 with any two bytes of data. EP1 only needs to be primed once before the first endpoint prime command in the current software.
- If EP2 has the possibility to send one or two bytes packet, software should prime EP3 with any two bytes of data. EP3 only needs to be primed once before the first endpoint prime command in the current software.

- If EP4 has the possibility to send one or two bytes packet, software should prime EP5 with any two bytes of data. EP5 only needs to be primed once before the first endpoint prime command in the current software.
- If EP6 has the possibility to send one or two bytes packet, software should prime EP7 with any two bytes of data. EP7 only needs to be primed once before the first endpoint prime command in the current software.

Table 1 shows a summary for the different cases. When the endpoints are primed with two bytes of data, QH/dTD also needs to be allocated/initialized correctly. Software does not need to set ENDPTCTRLx. TXE bit and software can release QH/dTD after the prime cycle is complete.

Table 1. Workaround Summary

Number of TX (IN) Endpoints	Number of TX (IN) Endpoints That May Have One or Two Byte Packets	May EP0 Have One or Two Byte Packets	Work-around Available	Workaround Description		
				Use for Endpoints That May Have One or Two Byte Packets	Use for Other Endpoints	Prime with Two Bytes
≤ 4	Do not care	Do not care	Yes	EP0, EP2, EP4 EP6		EP1, EP3, EP5, EP7
5	≤ 3	Yes	Yes	EP0, EP2, EP4	EP6, EP7	EP1, EP3, EP5
		No	Yes	EP2, EP4, EP6	EP0, EP1	EP3, EP5, EP7
	> 3	Do not care	No	—		
6	≤ 2	Yes	Yes	EP0, EP2	EP4, EP5, EP6, EP7	EP1, EP3
		No	Yes	EP2, EP4	EP0, EP1, EP6, EP7	EP3, EP5
	> 2	Do not care	No	—		
7	≤ 1	Yes	Yes	EP0, EP2, EP3, EP4, EP5, EP6, EP7		EP1
		No	Yes	EP2	EP0, EP1, EP4, EP5, EP6, EP7	EP3
	> 1	Do not care	No	—		
8	0	Do not care	Yes	Use EP0, EP1, EP2, EP3, EP4, EP5, EP6, EP7 as normal		
	> 0	Do not care	No	—		

3 Example Code

Example 1 shows how to dummy prime the EP3 with any two bytes of data when EP2 has the possibility to send one or two bytes packet. This code should be run before the first endpoint prime command in the current software.

Example 1. Workaround Code

```
D_QUEUEHEAD_T           * p_otg_endpoint3_DQH_in;
DeviceQueueTransferDescriptor_T       * p_otg_endpoint3_DTD_in;

void dummy_prime_ep3(void) {
        p_otg_endpoint3_DQH_in = (D_QUEUEHEAD_T*)
(((WORD)&(*p_otg_device_queuehead_list))+64*7);

        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_InterruptOnSetup=1;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_MaximumPacketLength=64;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_ZeroLengthTerminationSelect =0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_Multiplier=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_CurrentdTD=0xDEAD;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_NextTransferElementTerminate=1;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_NextTransferElementPointer=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_Status_TransactionError=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_Status_DataBufferError=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_Status_Halted=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_Status_Active=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_InterruptOnComplete=1;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_TotalBytes=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_CurrentOffset=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_BufferPointerPage0=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_BufferPointerPage1=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_BufferPointerPage2=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_BufferPointerPage3=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_BufferPointerPage4=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_SetupBufferByte0=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_SetupBufferByte1=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_SetupBufferByte2=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_SetupBufferByte3=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_SetupBufferByte4=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_SetupBufferByte5=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_SetupBufferByte6=0;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_SetupBufferByte7=0;

        (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_NextTransferElementTerminate=1;
        (*p_otg_endpoint3_DTD_in).
                DeviceTransferDescriptor_NextTransferElementPointer=0xBAD04;
        (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_Status_TransactionError=0;
        (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_Status_DataBufferError=0;
        (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_Status_Halted=0;
        (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_Status_Active=1;
        (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_InterruptOnComplete=1;
        (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_TotalBytes=0x2;
        (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_CurrentOffset=
                ((WORD) p_bulkdata_out)& 0x00000fff;
        (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_BufferPointerPage0=
                (((WORD) p_bulkdata_out) & 0xfffff000)>>12;
        (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_FrameNumber=0;
```

```
        (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_BufferPointerPage1=
                (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_BufferPointerPage0+1;
        (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_BufferPointerPage2=
                (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_BufferPointerPage0+2;
        (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_BufferPointerPage3=
                (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_BufferPointerPage0+3;
        (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_BufferPointerPage4=
                (*p_otg_endpoint3_DTD_in).DeviceTransferDescriptor_BufferPointerPage0+4;

        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_NextTransferElementPointer=
                ((WORD)p_otg_endpoint3_DTD_in)>>5;
        (*p_otg_endpoint3_DQH_in).DeviceEndPointQueueHead_NextTransferElementTerminate=0;

        *(WORD*)USB_OTG_ENDPTPRIME |= 0x1<<(3+16); // prime In transition in endpoint3
        while(*(WORD*)USB_OTG_ENDPTPRIME & (0x1<<(3+16)));
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or
+1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

Document Number: AN3683
Rev. 1
11/2009