USBN9603/4 - Increased Data Transfer Rate using Ping-Pong Buffering

National Semiconductor
Application Note 1222
March 2002
Revision 1.0

The National Semiconductor® USBN9603/4 has multiple endpoints to exchange data with the host. Normally, FIFO data transmission or reception is set up using one endpoint for each pipe. However, allocating two, same-direction, endpoints to one pipe is an effective way to increase the data transfer rate because the second FIFO is filled while the contents of the first are being transmitted.

This data buffering method is called ping-pong buffering.

1.0 Using two Endpoints for One Pipe

This Application Note describes the two-endpoint case, using endpoints EP1 (TXFIFO-1) and EP3 (TXFIFO-2) for transmitting, and EP2 (RXFIFO-1) and EP4 (RXFIFO-2) for receiving data to/from the host. For simplicity, the packet data size is the same as the FIFO length (64 bytes each).

During the enumeration process, when the node receives a SET_CONFIGURATION device request, the firmware enables the related endpoint. At this time the EP1/EP3 and EP2/EP4 pairs are each assigned the identical endpoint address by the EPCx.EP[3:0] control bits.

The USBN9603/4 has a built-in priority scheme, where the endpoint with the lower number gets the response and the data, if there are multiple endpoints of the same type programmed to the same endpoint number.

It is assumed that another firmware task prepares the transmission data for Buffer-1 (for EP1) and Buffer-2 (for EP3) data buffers. Similarly, it is assumed that another program task handles the data retrieved from Buffer-1 (for EP2) and Buffer-2 (for EP4).

Figure 1-1 shows a two-pipe connection between a host and a USB device.

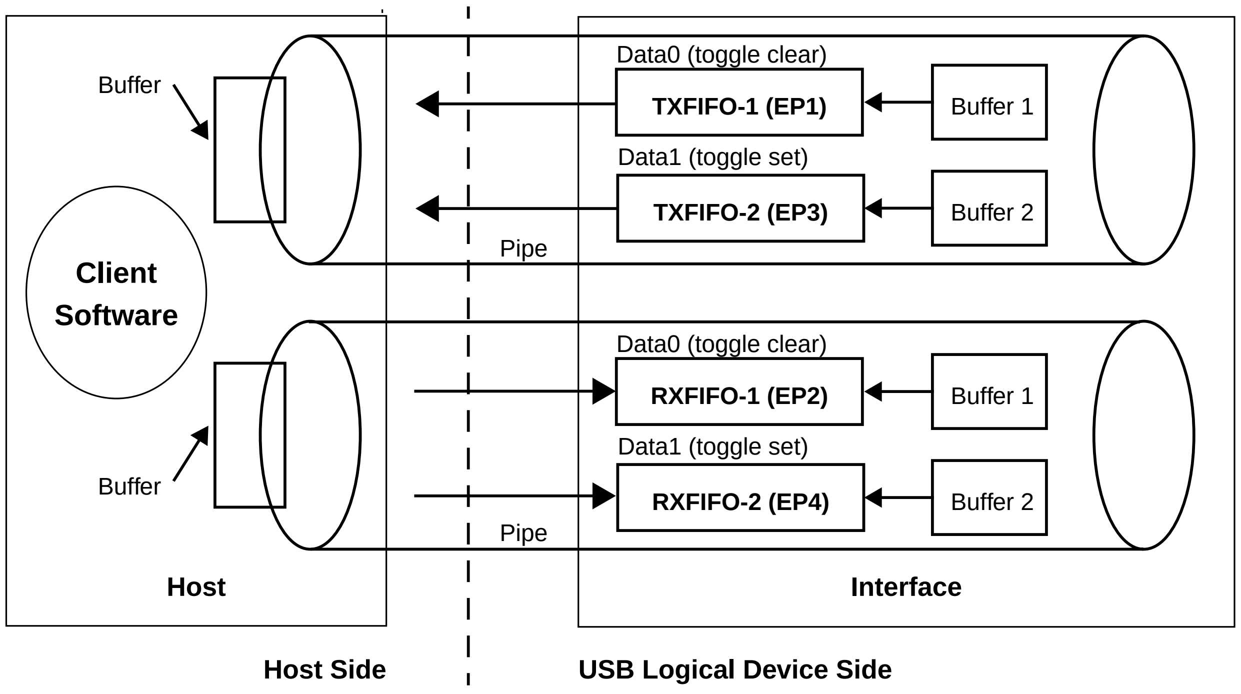

Figure 1-1. Two-pipe Connection

2.0 Data Transmission

The data packet request from the host uses IN-token with a combined node and endpoint address. There are various ways to move the transmission data from the local area into the USBN9603/4 FIFO, depending on the user's firmware. In this Application Note this data filling is performed in the main flow of the firmware, and the ping-pong buffering control is processed in each Tx event interrupt service.

2.1 MAIN FLOW

Figure 2-1 shows the data transmission process in the firmware main flow. Two flags, UPDATE1 and UPDATE2, communicate between the main flow and the Tx event interrupt service. When UPDATE1 is cleared (0) the transmission data is already set in the endpoint, when set (1) the FIFO needs to be filled with the next data.

UPDATE2 behaves in a similar manner.

The USB packet transaction format is assumed to be BULK and INTERRUPT. (The TOGGLE bit has a different meaning for the ISOCHRONOUS format of USBN9603/4. See Section 7.2.22 of the *USBN9603/USBN9604 Universal Serial Bus Full Speed Node Controller with Enhanced DMA Support Datasheet*.)

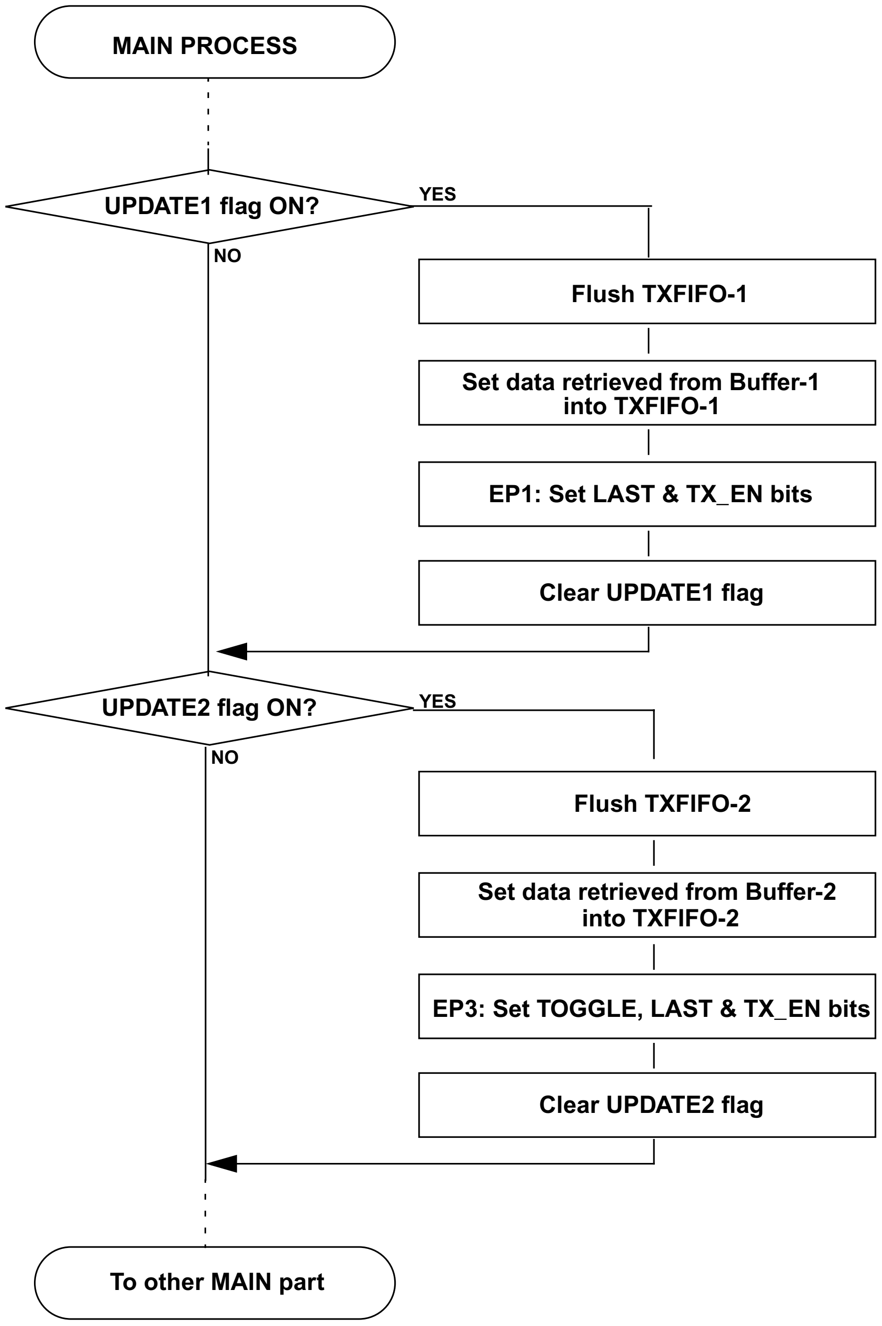
MAIN PROCESS
UPDATE1 flag ON?
YES
NO
Flush TXFIFO-1
Set data retrieved from Buffer-1 into TXFIFO-1
EP1: Set LAST & TX_EN bits
Clear UPDATE1 flag
UPDATE2 flag ON?
YES
NO
Flush TXFIFO-2
Set data retrieved from Buffer-2 into TXFIFO-2
EP3: Set TOGGLE, LAST & TX_EN bits
Clear UPDATE2 flag
To other MAIN part

Figure 2-1. FIFO Data Filling in Main Flow

2.1.1 FIFO data filling in the Main Flow.

First, EP1 is flushed and filled with data, and the LAST and TX_EN bits are set. The process is then repeated for EP3, the TOGGLE bit is also set.

For the USBN9603/4, the firmware must use the TOGGLE bit to detect, and recover from, transmitting errors at the host side.

However, when using ping-pong buffering, the firmware does not need to consider the toggle process because each endpoint uses its own, dedicated, toggle value.

(In this case, EP1 is for data0 and EP3 is for data1 but of course this is user application dependent.)

EP1 and EP3 are assigned the identical endpoint address for transmitting so the USBN9603/4 responds from the lower address number (EP1) at the first received IN-token. After EP1 transmits the contents of TXFIFO-1, the USBN9603/4 generates a transmission (Tx) event interrupt.

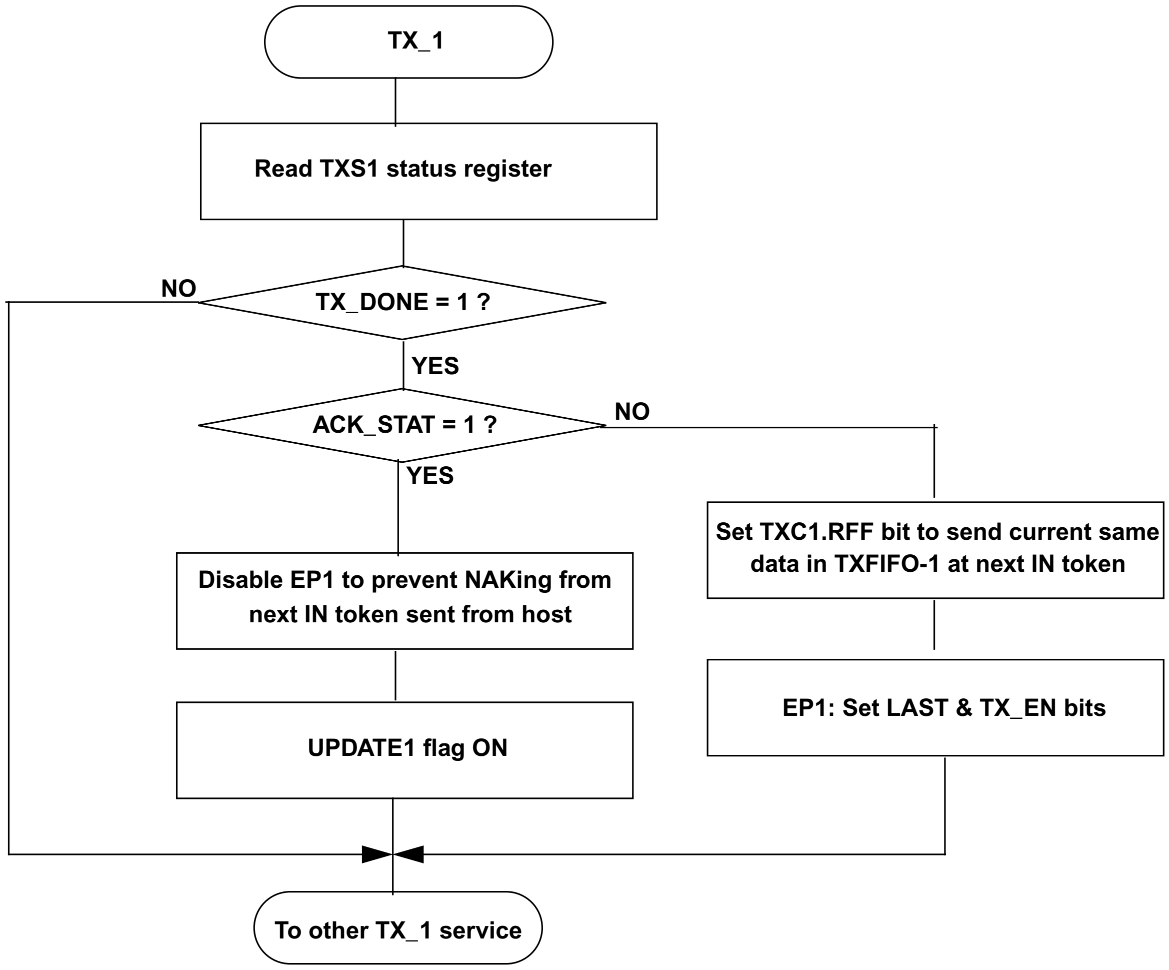
TX_1
Read TXS1 status register
TX_DONE = 1 ?
NO
YES
ACK_STAT = 1 ?
NO
YES
Set TXC1.RFF bit to send current same data in TXFIFO-1 at next IN token
Disable EP1 to prevent NAKing from next IN token sent from host
EP1: Set LAST & TX_EN bits
UPDATE1 flag ON
To other TX_1 service

Figure 2-2. EP1 Tx Event Interrupt Service

2.2 TX EVENT INTERRUPT FLOW

Figure 2-2 shows the EP1 Tx event interrupt service flow. First, firmware reads the TXS1 status register and confirms the condition of TX_DONE and ACK_STAT bits. After the USBN9603/4 has sent all FIFO content to the USB bus, the host returns the ACK signal if the data has been received without error. The ACK_STAT bit reflects the ACK signal, and the host responds within 16 bit times (Full Speed device) as defined in the Universal Serial Bus Specification 1.1. The USBN9603/4 takes care of this 16 bit time internally so firmware only reads the status register and checks both TX_DONE and ACK_STAT bit conditions.

If ACK_STAT is still clear, it can not determine whether the previously sent packet was correctly received at the host or was received with errors. This is an error transaction so the node firmware should prepare, in TXFIFO-1, the same data content as was sent before. A useful feature of the USBN9603/4 enables re-filling the same data into the FIFO. The last data exists in the FIFO so setting TXCx.RFF bit readies to send the data with adjusting internal FIFO data pointer.

When ACK_STAT is set the last transaction completes with no error, so firmware disables EP1 to prevent NAKing the next IN-token from the host. EP1 can not recognize the host endpoint address, as it is disabled at this stage, so the reply to the next IN-token is from EP3, which has the same endpoint address. Firmware sets the UPDATE1 flag for the next data filling into TXFIFO-1 in the main flow.

Figure 2-3 shows the EP3 Tx event interrupt service flow. As for EP1, the firmware first reads the TXS2 status register and checks the condition of the TX_DONE and ACK_STAT bits. If the last transaction completed with errors, firmware must re-fill TXFIFO-2 with the same contents. This process is the same as for EP1 with one difference, the firmware must set the TOGGLE bit. (As described in 2.1, EP3 is the endpoint with TOGGLE set (data1)).

If the previous transaction ends without error, the firmware enables EP1 again, so the reply to the next IN-token is again from EP1. EP3 is now empty, so setting the UPDATE2 flag fills it with the next data from the main flow.

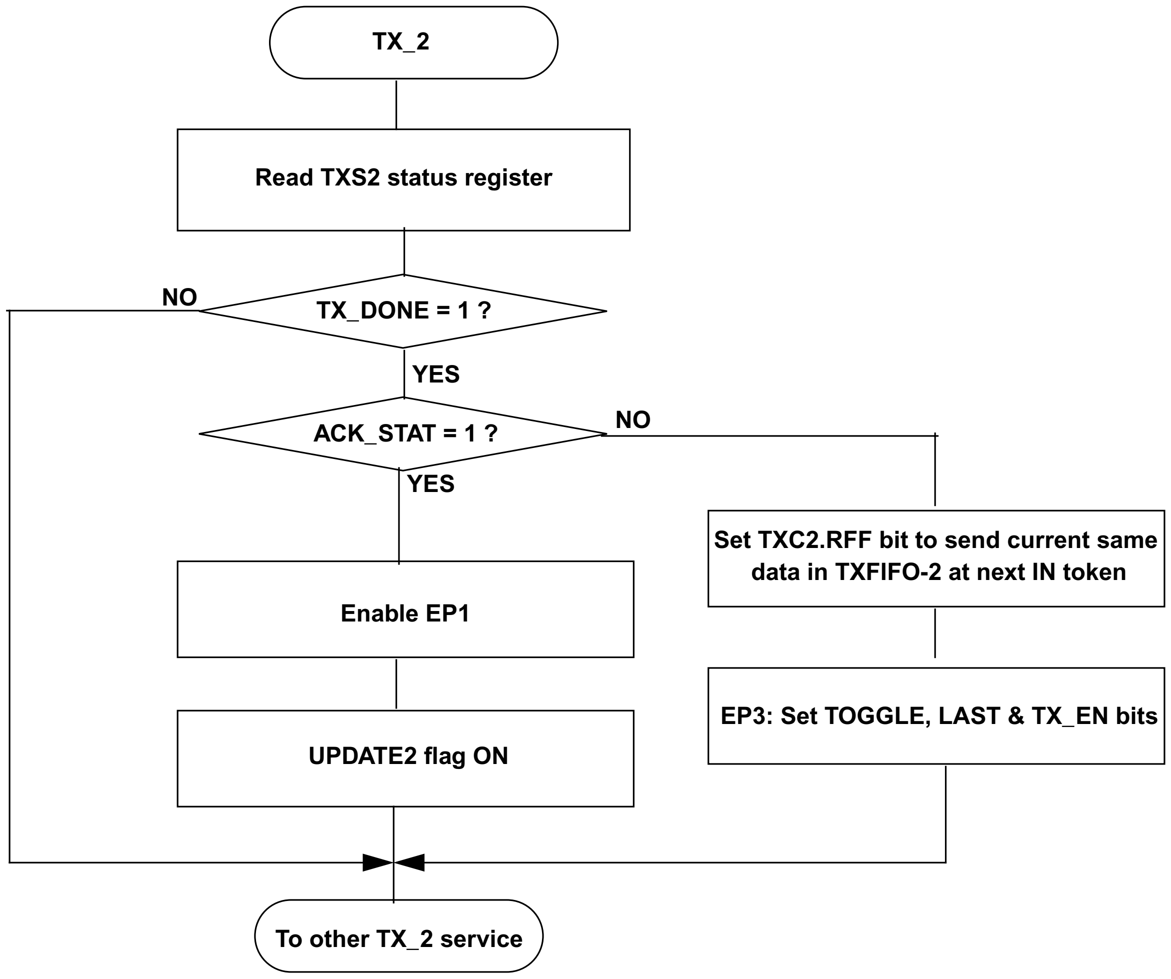
TX_2
Read TXS2 status register
NO
TX_DONE = 1 ?
YES
ACK_STAT = 1 ?
NO
YES
Set TXC2.RFF bit to send current same data in TXFIFO-2 at next IN token
Enable EP1
EP3: Set TOGGLE, LAST & TX_EN bits
UPDATE2 flag ON
To other TX_2 service

Figure 2-3. EP3 Tx Event Interrupt Service

3.0 Data Reception

The data packet transfer from the host uses OUT-token with node address and endpoint address combination. For the purposes of this Application Note, we assume that data retrieval from the FIFO to local memory, and related flow control, is executed in the USBN9603/4 data reception (Rx) event interrupt service routine.

3.1 RX EVENT INTERRUPT FLOW

Figure 3-1 shows the data reception process in EP2 Rx event interrupt service flow. First, firmware reads the RXS1 status register and confirms that there are no errors and that all the data sent by the host has been received.

If the RXS1 status register indicates an error, or incomplete data reception, the USBN9603/4 does not return the handshake status (ACK or NAK) and the host repeats the last procedure. Firmware prepares to receive a packet from the host by flushing RXFIFO-1 and setting TX_EN for the next OUT-token and data.

As described in Section 2.1, this Application Note uses two endpoints and assigns a separate TOGGLE value to each.

In this case, EP2 is assumed TOGGLE clear (data0) and EP4 is assumed TOGGLE set (data1).

When the status indicates no error, the firmware checks the TOGGLE bit:

- If TOGGLE is *not* zero, this is an error transaction and the firmware must ignore the current contents of RXFIFO-1.
- If TOGGLE is zero, this transaction completes without error and the firmware retrieves the contents of RXFIFO-1 and saves it into local memory Buffer-1.

After flushing RXFIFO-1 to prepare for the next data reception, the firmware disables EP2. As a result, EP2 never reacts with host designated endpoint address. EP4, therefore, responds alternately with the next OUT-token and data.

Figure 3-2 shows the data reception process in the EP4 Rx event interrupt service flow. The firmware reads the RXS2 status register, and checks for error conditions. The process is the same as for EP2, described above.

If no errors are detected, EP4 is assigned TOGGLE set (data1) so firmware confirms the TOGGLE setting. According to the result of the TOGGLE condition check, the current received data in RXFIFO-2 is either discarded, or retrieved into the local memory Buffer-2.

The firmware then prepares for the next reception of EP4 to flush RXFIFO-2 and set RX_EN bit. Moreover, at this stage EP2 is enabled again and sets the RX_EN bit so it is able to respond and thus receive the next OUT-token and data from the host.

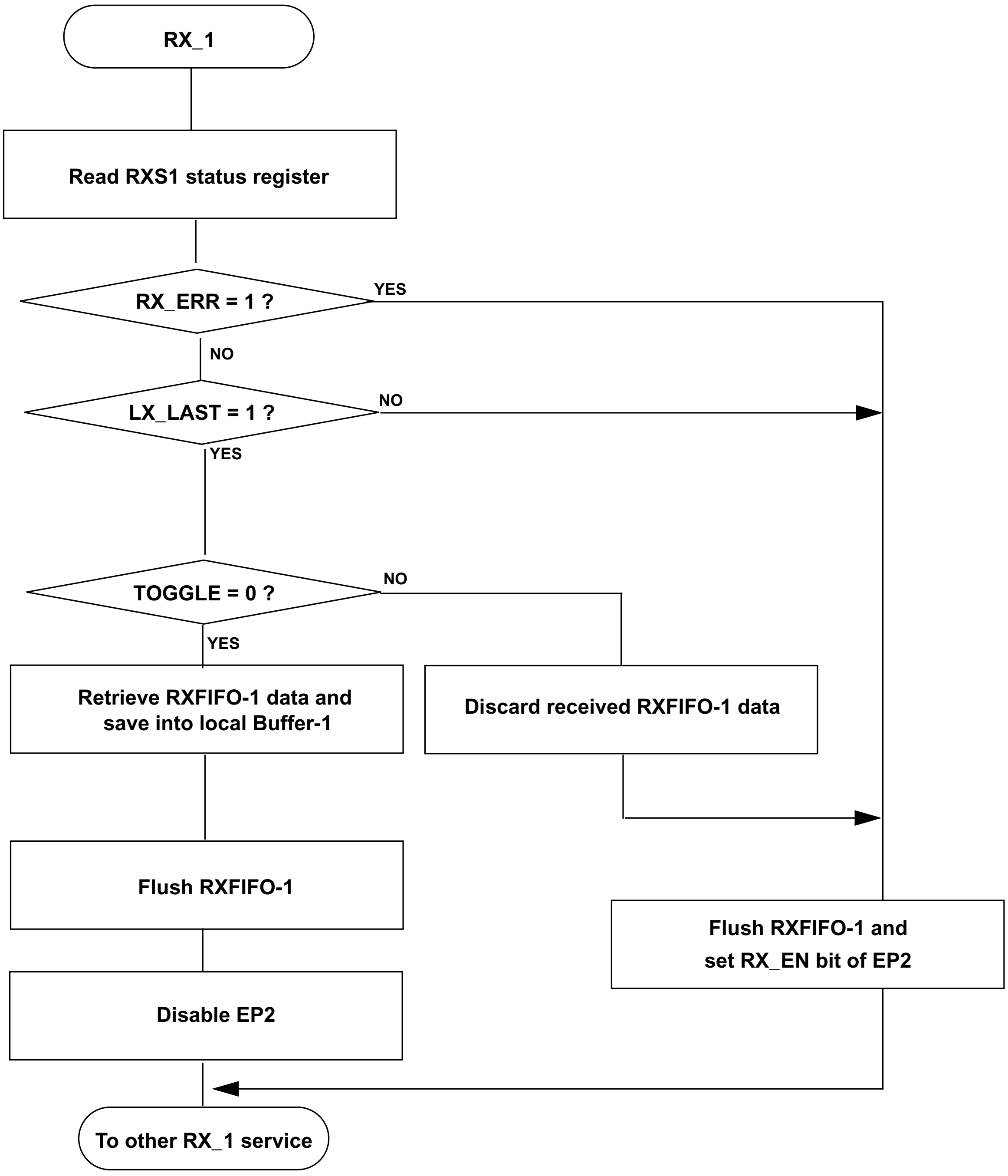
RX_1
Read RXS1 status register
RX_ERR = 1 ?
YES
NO
LX_LAST = 1 ?
NO
YES
TOGGLE = 0 ?
NO
YES
Retrieve RXFIFO-1 data and save into local Buffer-1
Discard received RXFIFO-1 data
Flush RXFIFO-1
Flush RXFIFO-1 and set RX_EN bit of EP2
Disable EP2
To other RX_1 service

Figure 3-1. EP2 Rx Event Interrupt Service

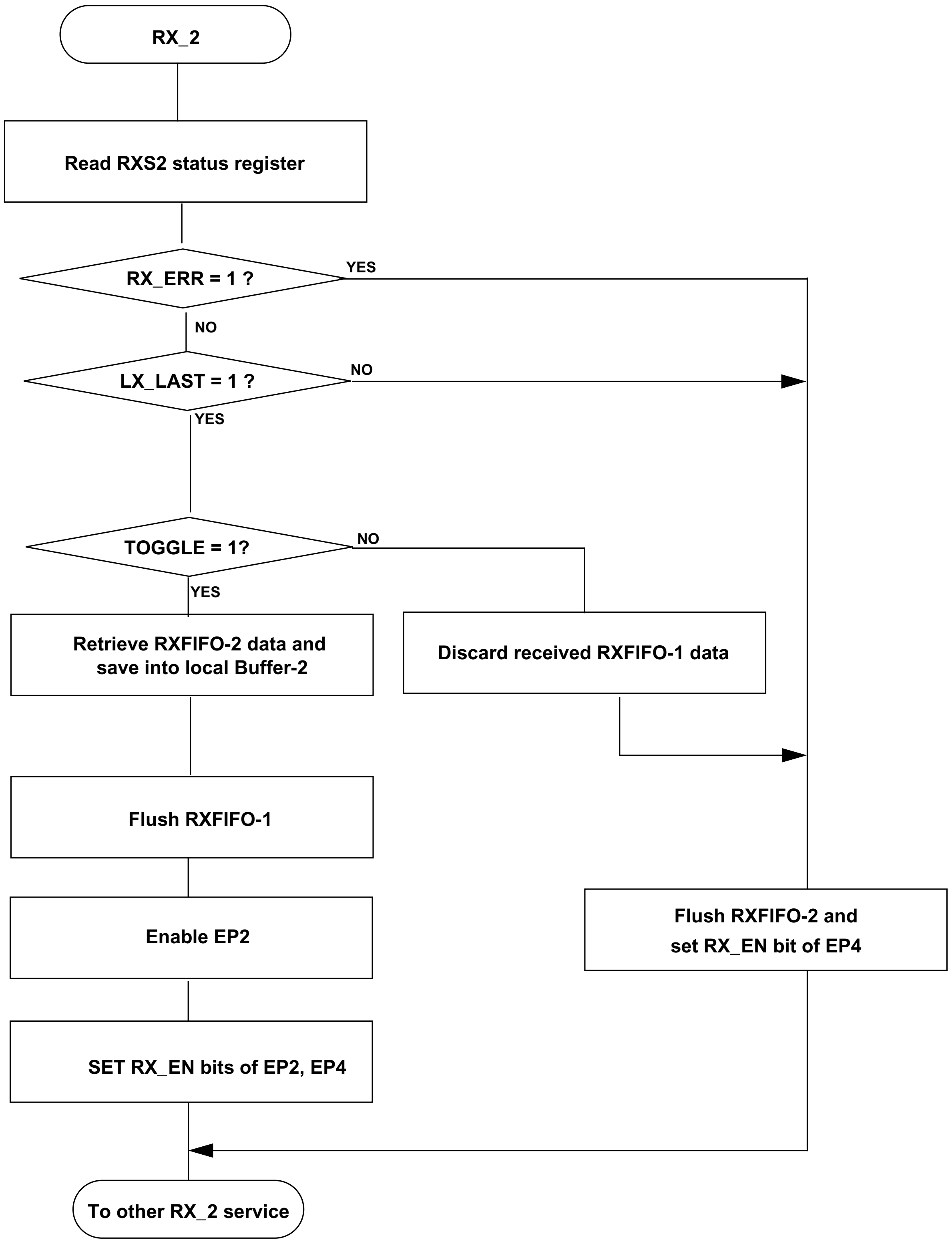
RX_2
Read RXS2 status register
RX_ERR = 1 ?
YES
NO
LX_LAST = 1 ?
NO
YES
TOGGLE = 1?
NO
YES
Retrieve RXFIFO-2 data and save into local Buffer-2
Discard received RXFIFO-1 data
Flush RXFIFO-1
Enable EP2
Flush RXFIFO-2 and set RX_EN bit of EP4
SET RX_EN bits of EP2, EP4
To other RX_2 service

Figure 3-2. EP4 Rx Event Interrupt Service

4.0 Conclusions

The USBN9603/4 can assign the same endpoint address to multiple endpoints. In this case, data is received or transmitted to/from the endpoint with the lowest number. This ping-pong buffering is an effective way to increase the data transfer rate because the second FIFO is filled while the contents of the first are being transmitted. An additional advantage is the ability to reduce the firmware effort required to alternate the TOGGLE bit on transmitting. This buffering can also be adapted for the ISOCHRONOUS transaction format.

5.0 Program Code Example

```
//-----------------------------------------------------
// SET_CONFIGURATION request
//-----------------------------------------------------
void  setconfiguration(void)
{
   usb_cfg = usb_buf[2];//set the configuration number
   if(usb_buf[2] != 0)//set the configuration
      {
         dpapid =0;    //first PID is DATA0
         stalld = 0;   //nothing stalled

   //Ping-Pong buffering: transmitting endpoints are EP1 and EP3 so these are assigned
   //the same endpoint address, in this case address 5
         FLUSHTX1   //flush TXFIFO-1
         write_usb(EPC1,EP_EN+5);//enable EP1 and address 5

         FLUSHTX2   //flush TXFIFO-2
         write_usb(EPC3,EP_EN+5);//enable EP3 and address 5

   //Ping-Pong buffering: receiving endpoints are EP2 and EP4 so these are assigned the
   //same endpoint address, in this case address 2
         FLUSHRX1   //flush RXFIFO-1
         write_usb(EPC2,EP_EN+2);//enable EP2 and address 2
         write_usb(RXC1,RX_EN);//RXFIFO-1 receive enable

         FLUSHRX2   //flush RXFIFO-2
         write_usb(EPC4,EP_EN+2);//enable EP4 and address 4
         write_usb(RXC2,RX_EN);//RXFIFO-2 receive enable
      }
   else
      {
         write_usb(EPC1,0);//disable EP1
         write_usb(EPC3,0);//disable EP3
         write_usb(EPC2,0);//disable EP2
         write_usb(EPC4,0);//disable EP4
      }
}

//-----------------------------------------------------
// MAIN PROCESS
//-----------------------------------------------------
    |
    |
    |

   //As the following process handles the FIFO data filling, it should not be disturbed
   //by other interrupts.  All interrupts are disabled from here.
global_int_off();

   //According to the EP1 condition, skip or fill the TXFIFO-1
if(update1 = 1)//check to need filling TXFIFO-1
   {
      FLUSHTX1      //flush TXFIFO-1
      for(byte_count=0; byte_count <64; byte_count++)
   //fill the data from local memory area(Buffer_1) to TXFIFO-1, fixed 64 byte

      {
         write_usb(TXD1,Buffer_1[byte_count]);
      }
   write_usb(TXC1,TX_LAST+TX_EN);     //TXFIFO-1 transmit enable and set DATA0 (TOGGLE=0)
   update1 = 0;                       //complete filling the data into TXFIFO-1
   }

   //According to the EP3 condition , skip or fill the TXFIFO-2
if(update2 = 1)      //check to need filling TXFIFO-2
```

```
{
    FLUSHTX2          //flush TXFIFO-2
    for(byte_count=0; byte_count <64; byte_count++)//fill the data from local memory area
                                                  //(Buffer_2) to TXFIFO-2, fixed 64 byte
        {
         write_usb(TXD2,Buffer_2[byte_count]);
        }
    write_usb(TXC2,TX_TOGL+TX_LAST+TX_EN);     //TXFIFO-2 transmit enable and set DATA1 (TOGGLE=1)
    update2 = 0;                              //complete filling the data into TXFIFO-2
    }
    //return to the normal condition, all interrupts are enabled from here
global_int_on();
    |
    |
    |

//----------------------------------------
// TX_1 PROCESS
//----------------------------------------
void   tx_1(void)
{
  txstat = read_usb(TXS1);//read the transmit status register
  if((txstat & TX_DONE) && (txstat & ACK_STAT))
    {
          write_usb(EPC1,0);//when last transmitted data is received
              //completely at host, disable EP1
          update1 = 1;//set flag which means TXFIFO-1 empty
     }
  else
    {
      if((txstat & TX_DONE) && !(txstat & ACK_STAT))
        {
          write_usb(TXC1,RFF+TX_LAST+TX_EN);//no ACK from host so send TXFIFO-1
              //content again ,TOGGLE=0
        }
      else
        {
         }
    }
}
//-----------------------------------------------------
// TX_2 PROCESS
//-----------------------------------------------------
void   tx_2(void)
{
  txstat = read_usb(TXS2);//read the transmit status register
  if((txstat & TX_DONE) && (txstat & ACK_STAT))
    {
          write_usb(EPC1,EP_EN+ 5);//when last transmitted data is received
              //completely at host, enable EP1
              //in this case set endpoint address 5
          update2 = 1;//set flag which means TXFIFO-2 empty
     }
  else
    {
      if((txstat & TX_DONE) && !(txstat & ACK_STAT))
        {
          write_usb(TXC2,RFF+TX_TOGL+TX_LAST+TX_EN);//no ACK from host so send TXFIFO-2
              //content again , TOGGLE=1
        }
      else
        {
         }
    }
}
```

```
//--------------------------------------------------
// RX_1 PROCESS
//--------------------------------------------------
void rx_1(void)
{
   rxstat = read_usb(RXS1); //read the receive status register
   if((rxstat & RX_ERR) || !(rxstat & RX_LAST) || (rxstat & RX_TOGL))
      {

      //if this reception is error transaction, discard current RXFIFO-1 data, prepare
      //next receive by flushing RXFIFO-1 and setting receive enable
         FLUSHRX1
         write_usb(RXC1,RX_EN);
      }
   else
      {
      //received data is error-free so retrieve RXFIFO-1 content into local memory area
      //(Buffer_1) and flush RXFIFO-1 for next new packet reception and disable this
      //endpoint to prevent the NAKing from next OUT-token
         for(byte_count = 0; byte_count <64; byte_count++)
            {
               Buffer_1[byte_count] = read_usb(RXD1);
            }
         FLUSHRX1
         write_usb(EPC2,0);
      }
}


//--------------------------------------------------
// RX_2 PROCESS
//--------------------------------------------------
void rx_2(void)
{
   rxstat = read_usb(RXS2); //read the receive status register
   if((rxstat & RX_ERR) || !(rxstat & RX_LAST) || !(rxstat & RX_TOGL))
      {

      //if this reception is error transaction, discard current RXFIFO-2 data and prepare
      //next receive by flushing RXFIFO-2 and setting receive enable
         FLUSHRX2
         write_usb(RXC2,RX_EN);
      }
   else
      {
      //received data is error-free so retrieve the RXFIFO-2 content into local
      //memory area (Buffer_2)
      for(byte_count = 0; byte_count <64; byte_count++)
         {
            Buffer_2[byte_count] = read_usb(RXD2);
         }

      //flush RXFIFO-2 for next new packet receive and enable it
      FLUSHRX2
      write_usb(RXC2,RX_EN);
      write_usb(EPC2,EP_EN+2); //enable EP2 & RXFIFO-1 again
      write_usb(RXC1,RX_EN);
   }
}
```

National Semiconductor Corporation Americas
Email: new.feedback@nsc.com

National Semiconductor Europe

Fax:	+49 (0) 180-530 85 86
Email:	europe.support@nsc.com
Deutsch Tel:	+49 (0) 69 9508 6208
English Tel:	+44 (0) 870 24 0 2171
Français Tel:	+33 (0) 1 41 91 87 90

National Semiconductor Asia Pacific Customer Response Group
Tel: 65-2544466
Fax: 65-2504466
Email: ap.support@nsc.com

National Semiconductor Japan Ltd.
Tel: 81-3-5639-7560
Fax: 81-3-5639-7507
Email: nsj.crc@jksmtp.nsc.com

www.national.com