Keywords: USB,host,MAX3421,MAX3421E,embedded

APPLICATION NOTE 4000

MAX3421E-Revisions 1 and 2 Host OUT Transfers

Feb 15, 2007

Abstract: This application note discusses the MAX3421E-Revisions 1 and 2 acting as a host USB controller. The MAX3421E sends data to a USB peripheral using a double-buffered Send FIFO. In Revs 1 and 2 of the part, special consideration is required for programming OUT transfers. This application note describes the sending mechanism. Example code for programming single-buffered and double-buffered OUT transfers is given.

Introduction

The MAX3421E is a USB host/peripheral controller that, when operating as a host, uses a double-buffered Send FIFO to send data to a USB peripheral. This FIFO pair is controlled by two registers:

R2: SNDFIFO, the FIFO data register
R7: SNDBC, the byte count register

The microcontroller repeatedly writes the SNDFIFO register R2 to load the FIFO with up to 64 data bytes. Then the microcontroller writes the SNDBC register, which this does three things:

1. Tells the MAX3421E SIE (Serial Interface Engine) how many bytes in the FIFO to send.
2. Connects the SNDFIFO and SNDBC register to the USB logic for USB transmission.
3. Clears the SNDBAVIRQ interrupt flag. If the second FIFO is available for µC loading, the SNDBAVIRQ immediately re-asserts.

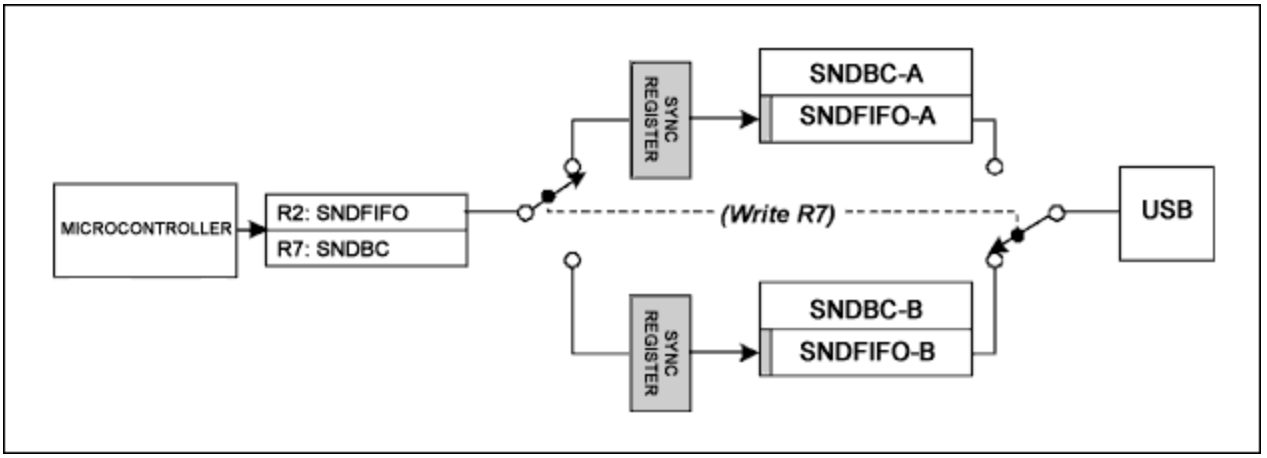

Figure 1. The SNDFIFO and SNDBC registers load a "ping-pong" pair of FIFOs and byte count registers.

As illustrated in **Figure 1**, the first FIFO byte does not come from the physical FIFO. Instead, the first FIFO

byte comes from a synchronization register used to reconcile two internal clock domains, one for the microcontroller and the other for the USB logic.

Programming Single-Buffered Transfers

For simple transfers that are not bandwidth-critical, the firmware to send a packet is relatively straightforward. The steps are:

1. Load the SNDFIFO.
2. Load the SNDBC register.
3. Launch the transfer by writing the HXFR register with the OUT PID and end-point number.
4. Wait for the HXFRDNIRQ (Host Transfer Done Interrupt Request).
5. Read the transfer result code from the HRSL register.
    1. If ACK, you are done.
    2. If NAK, go to the next step.
6. Reload the first FIFO byte, and relaunch the transfer:
    1. Write SNDBC = 0, a dummy value to switch the FIFO containing the OUT data back under microcontroller control.
    2. Rewrite only the first FIFO byte to the SNDFIFO register. This byte goes into the SYNC REGISTER in Figure 1.
    3. Rewrite the correct byte count for the resent packet to the SNDBC register. This switches the FIFO back to the USB side for USB retransmission.
    4. Relaunch the packet by going to step 3.

Step 4 makes this procedure a single-buffered transfer. Because this sequence waits for step 4 to complete, the microcontroller does not load the second FIFO while the first one moves from the MAX3421E SNDFIFO to the USB peripheral.

Step 6 is required because in MAX3421E-Revision 1, the synchronization flip-flop must be re-initialized for every repeated USB OUT transfer.

Programming Double-Buffered Transfers

Double buffering improves performance when long data records, consisting of multiple 64-byte packets, are transmitted from the USB host to a USB peripheral. Performance improves because, while one SNDFIFO is connected to USB to transmit a packet, the microcontroller can concurrently load the other SNDFIFO with the next 64-byte data packet. The program steps are, however, a bit more complex than for a single-buffered transfer. At any given time the program must track two buffers of data because of the need to flip the FIFOs (i.e., when a NAK is encountered), reload the first byte, and reload the byte count register to flip them back. **Figure 2** illustrates one possible sequence of steps. An example function, **Send_OUT_Record()**, at the end of this note implements the Figure 2 flowchart.

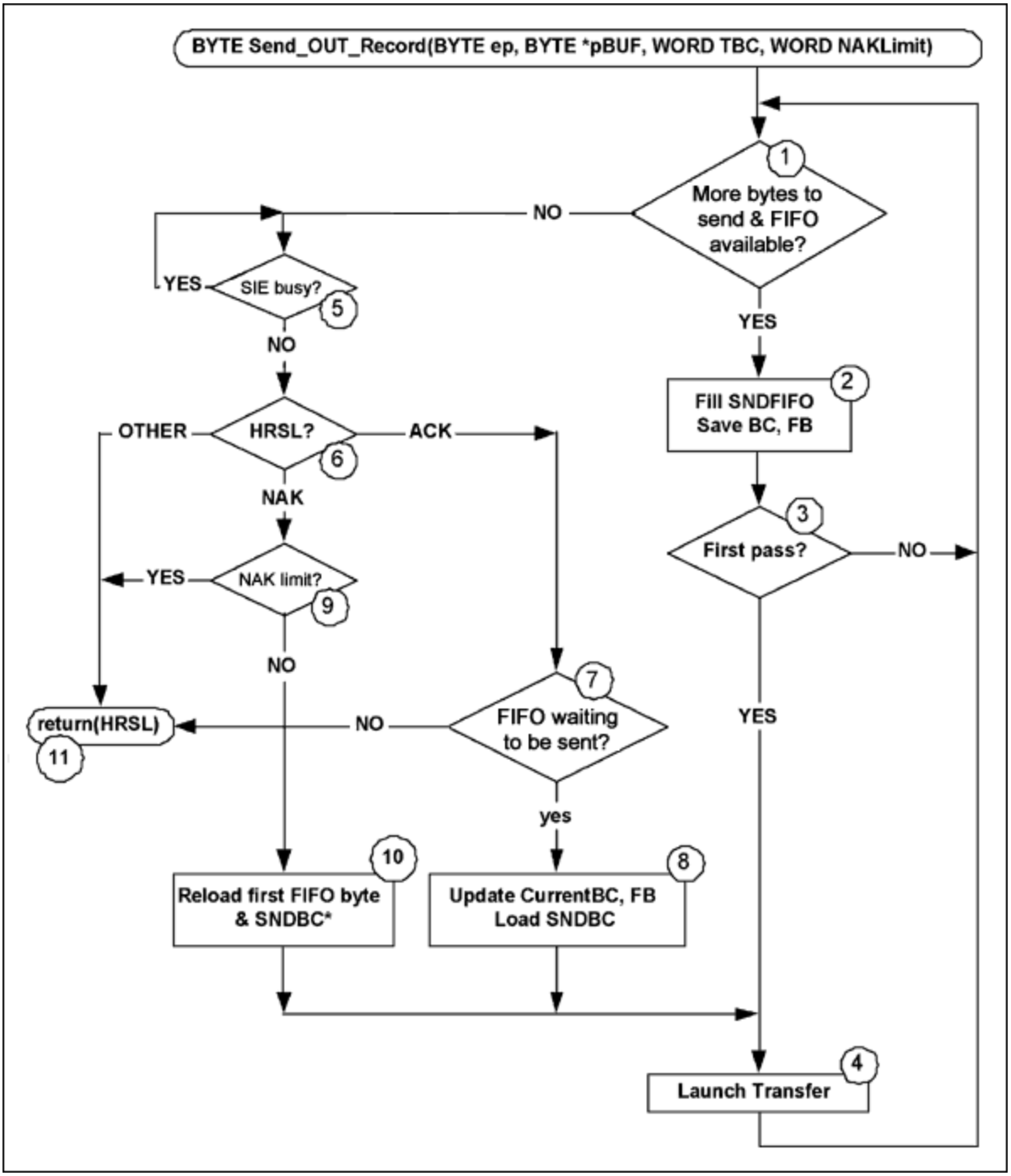

Figure 2. Flowchart illustrates double buffered OUT packets.

The loop on the right (steps 1 through 4) loads FIFOs with USB data; the loop on the left, starting with step 5, dispatches FIFOs over USB and handles the NAK retries. The "First pass?" check (step 3) ensures that a USB transfer is launched immediately after the host controller loads the first SNDFIFO. The flowchart is arranged to give FIFO loading priority over sending, thereby maintaining the double-buffered performance.

The flowchart is best understood by considering three cases:

1. Send one packet (1 to 64 byte total payload).

2. Send two packets (65 to 128 byte total payload).
3. Send three or more packets (129 or more byte total payload).

Referring to the **Send_OUT_Record()** function at the end of this note, the calling program passes four parameters to this function:

1. **ep**, end-point number to which the OUT packets should be dispatched
2. ***pBUF**, a pointer to a buffer full of byte data
3. **TBC**, the total byte count to send (the number of bytes in the buffer)
4. **NAKLimit**, the number of consecutive NAK responses to accept from the peripheral before giving up and returning

The function returns the value of the MAX3421E Host Result Code Register (HRSL) that was read on the last packet sent. This value will indicate "ACK" if the transfer was successful, "NAK" if the NAK limit was exceeded, or an error code for other problem terminations.

Send One OUT Packet

The loop enters at step 1 with 64 or fewer bytes to send. The step 1 result is true, so the SPI master fills the SNDFIFO. The function saves the byte count (BC) and SNDFIFO first byte (FB) in case BC and FB are needed in step 10—if, that is, the peripheral "NAKs" the OUT transfer. Since this is the first transfer of a record, the function launches the OUT transfer at step 4. Back at step 1 there are no more bytes to send, so the function waits for transfer completion in step 5 and tests the device response at step 6. At this point, if there is an ACK, step 7 finds no more data to send and the function returns at step 11. If there is a NAK, however, the NAK limit is tested at step 9 and, if not exceeded, the packet is resent in steps 10 and 4.

Send Two OUT Packets

The function again enters at step 1 with between 65 and 128 bytes to send. As before, the function fills the first SNDFIFO and immediately launches the transfer in steps 2 through 4. Back at step 1, the function finds more bytes to send, so it again fills the SNDFIFO at step 2. Since this is not the first transfer, the function does not yet launch the next packet at step 4. Back at step 1 once again, the first SNDFIFO is moving the first packet over USB while the second packet remains in the second SNDFIFO, ready for transmission. Note that at step 2 the function actually saved two sets of BC/FB data (byte count and FIFO first byte): one set for the currently transferring SNDFIFO, and another set for the pending data waiting in the second SNDFIFO. Since there are no more data to send (and also both SNDFIFOs are full of data), control branches to step 5.

The function waits in step 5 for the first packet to transfer, and then checks the transfer result at step 6. As before, if the packet is "NAK'd," the function reloads the FIFO and byte count register with the pending BC/FB values in step 10, resends the packet at step 4, and branches back to step 5 (by going through step 1) to wait for completion. The steps 5-6-9-10-4-5 loop continues for every NAK until an ACK is received or the NAK limit is reached.

If the peripheral ACKs the OUT transfer, the function tests for termination at step 7. If not complete, the function branches to step 8 to send the next packet, which is waiting in the second SNDFIFO. At step 8 the function performs several tasks: copies the "pending" BC/FC values into the "current" BC/FC variables in case these values are needed at step 10; switches the second SNDFIFO to USB by loading the current byte count; and launches the OUT packet at step 4. The loop then proceeds as before for every NAK. When an ACK response is received, the function exits at steps 7 through 11.

Send Three or More OUT Packets

This function basically follows the previous procedure for two packets until the point when one packet is underway and another is pending (waiting in the second SNDFIFO). Then, every time the function launches another packet at step 4, it checks for more data to load into a SNDFIFO at step 1. There must be more data to send and a FIFO must be available for the program flow to reach step 2. Because the "load SNDFIFO" loop in steps 1 through 3 takes priority over the "send FIFO" loop in steps 5...4, data are always loaded into a SNDFIFO while data concurrently move over USB, thereby accomplishing the double buffering.

Performance

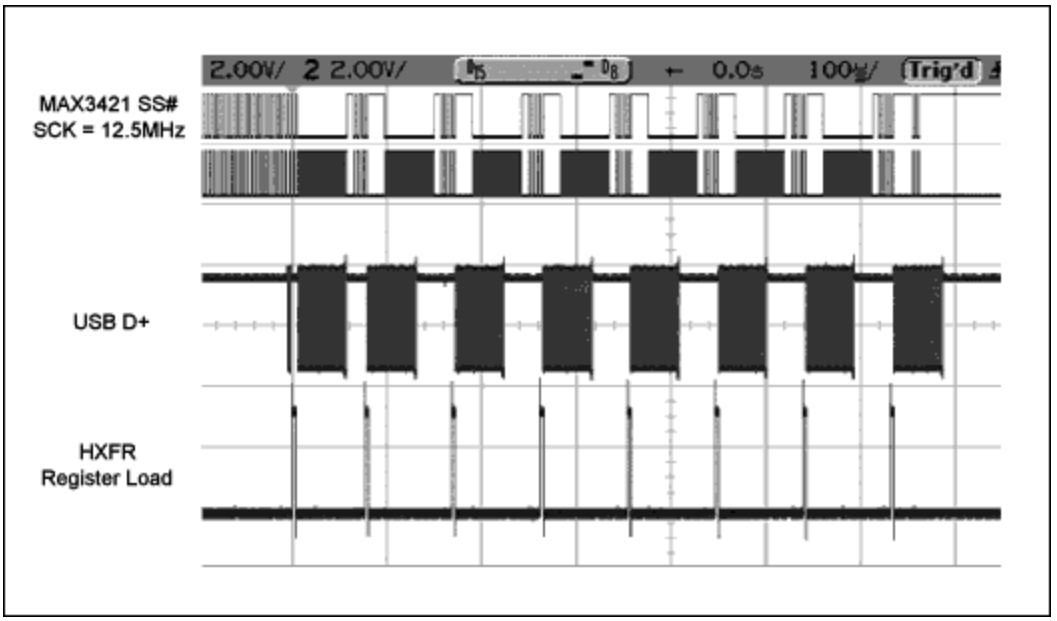

Figure 3. Scope traces show the MAX3421E loading and sending the USB packets.

Figure 3 is a scope capture illustrating the overlap between loading and sending 64 byte OUT packets using the **Send_OUT_Record()** function. The record size is 512 bytes, comprising eight 64-byte packets. In this trace, the peripheral device attached to the MAX3421E generates no NAKs, so the maximum transfer bandwidth can be measured and so the entire transfer fits onto one scope capture.

The top two traces show SPI port activity as the SPI master (an ARM7 with SPI hardware) loads 64 byte data into the MAX3421E SNDFIFO. For every SNDFIFO load, the SS# (Slave Select) line goes low and the SCK (Serial Clock) pulses 65 x 8 times, once to load a command byte and then eight times for each of the 64 data bytes. The first SNDFIFO load is done before this trace starts, so Figure 3 shows seven SNDFIFO loads.

The third trace is the USB D+ signal, showing the 64-byte OUT packets moving over USB from the MAX3421E host to the peripheral. The bottom trace is a pulse indicating the ARM7 loading the MAX3421E HXFR register to initiate a transfer.

Note that soon after a USB packet starts, the ARM7 starts loading the second SNDFIFO concurrent with the USB packet moving over the bus. This double buffering improves transfer bandwidth.

Bandwidth Measurement

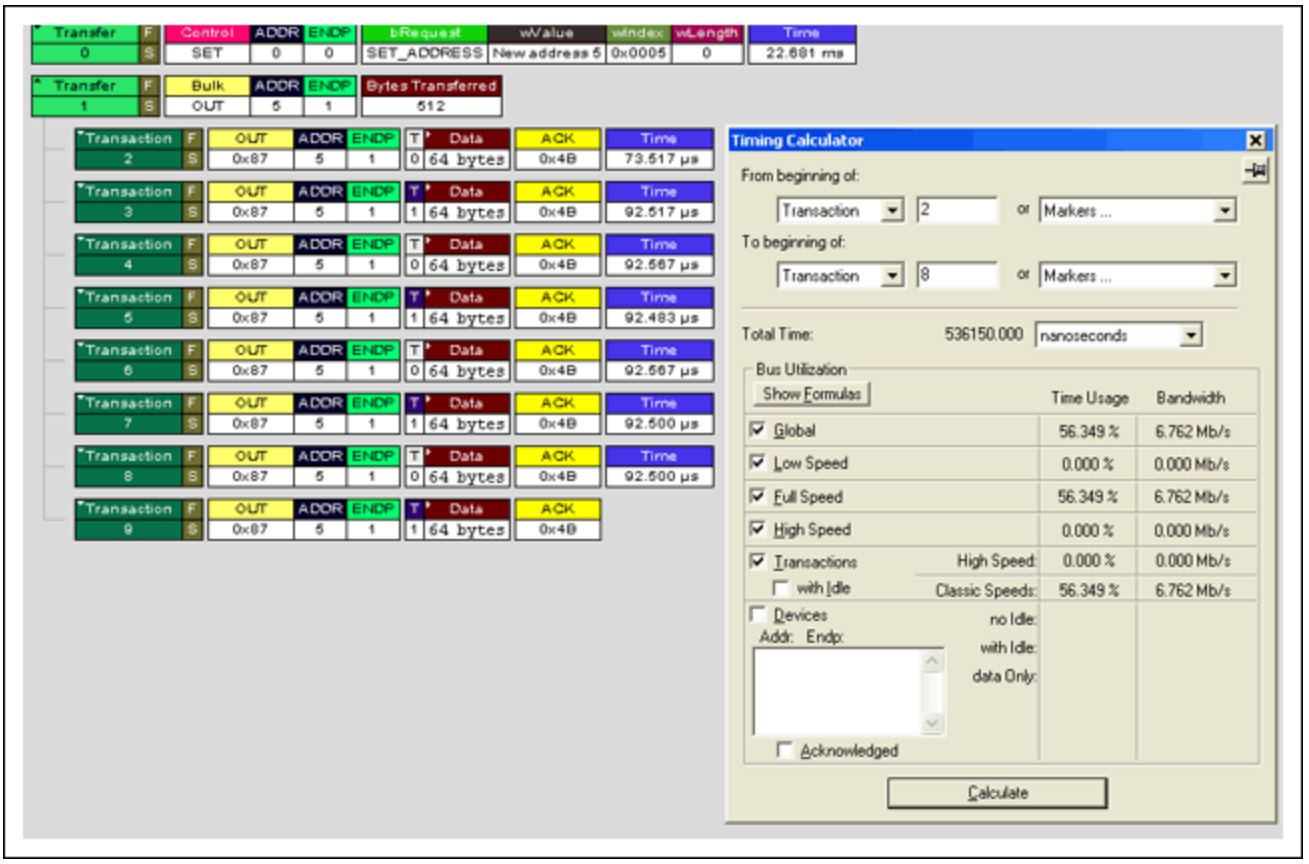

Figure 4. CATC (LeCroy) trace and bus analysis for the Figure 3 transfer.

As **Figure 4** illustrates, the **Send_OUT_Record()** function transferred a 512-byte record at 6.76Mbps. For reference, a USB full-speed USB thumb drive attached to a PC using a UHCI USB controller (the only USB attachment) transferred 512 byte records at 5.94Mbps.

Implementation Notes

The code example provided below was written and tested using the Maxim USB Laboratory, whose details can be found in application note 3936, "The Maxim USB Laboratory." This laboratory kit contains both a MAX3421E and a MAX3420E USB peripheral controller. To test the firmware, the MAX3421E host was connected to the MAX3420E peripheral with a USB cable, and the MAX3420E was commanded to accept every OUT transfer (no NAKs) by the commented-out statement labeled "*1*".

Note: *Future revisions of the MAX3421E will not have the FIFO reload issue. The MAX3421E will relaunch an OUT packet simply by reloading the HXFR register. This change will make the* ***Send_OUT_Record()*** *function unnecessary.*

The Send_OUT_Record() Example Function

```
// ******************************************************************************
// Send an OUT record to end point 'ep'.
// pBuf points to the byte buffer; TBC is total byte count.
// NAKLimit is the number of NAKs to accept before returning.
```

```
//
// Returns HRSL code (0 for success, 4 for NAK limit exceeded, HRSL for problems)
// ***************************************************************************
//
BYTE Send_OUT_Record(BYTE ep, BYTE *pBuf, WORD TBC, WORD NAKLimit)
{
static WORD NAKct,rb;           // Buf index, NAK counter, remaining bytes to
send
WORD bytes2send;                // temp
BYTE Available_Buffers;         // Remaining buffers count (0-2)
BYTE FI_FB;                     // Temporary FIFO first byte
static BYTE CurrentBC;          // Byte count for currently-sending FIFO
static BYTE CurrentFB;          // First FIFO byte for currently-sending FIFO
static BYTE PendingBC;          // Byte count for next 64 byte packet scheduled
for sending
static BYTE PendingFB;          // First FIFO byte for next 64 byte packet
scheduled for sending
BYTE dum;
BYTE Transfer_In_Progress,FirstPass;    // flags
//
NAKct=0;
Available_Buffers = 2;
rb = TBC;                       // initial remaining bytes = total byte count
FirstPass = 1;
//
do
        {
        while((rb!=0)&&(Available_Buffers!=0))
        // WHILE there are more bytes to load and a buffer is available
                {
                // Pwreg(rEPIRQ,bmOUT1DAVIRQ);// *1* enable the 3420 for another
OUT transfer
                FI_FB = *pBuf;                // Save the first byte of the 64
byte packet
                bytes2send = (rb >= 64) ? 64: rb;         // Lower of 64 bytes
and remaining bytes
                rb -= bytes2send;                         // Adjust 'remaining
bytes'
                Hwritebytes(rSNDFIFO,64,pBuf);
                pBuf += 64;                   // Advance the buffer pointer to
the next 64-byte chunk
                Available_Buffers -= 1        // One fewer buffer is now
available
//
                if(Available_Buffers==1)      // Only one has been loaded
                        {
                        CurrentBC = bytes2send;
                        CurrentFB = FI_FB;
                        }
                else                           // Available_Buffers must be 0,
both loaded.
                        {
                        PendingBC = bytes2send;
                        PendingFB = FI_FB;
                        }
//
                if(FirstPass)
                        {
                        FirstPass = 0;
                        Hwreg(rSNDBC,CurrentBC);        // Load the byte count
                        L7_ON                            // Light 7 is used as
scope pulse
                        Hwreg(rHIRQ,bmHXFRDNIRQ);       // Clear the IRQ
                        Hwreg(rHXFR,(tokOUT | ep));     // Launch an OUT1
transfer
                        L7_OFF
                        }
                }       // While there are bytes to load and there is space for
them
```

```
//
        do // While a transfer is in progress (not yet ACK'd)
        {
//              while((Hrreg(rHRSL) & 0x0F) == hrBUSY) ;        // Hang here
until current packet completes
                while((Hrreg(rHIRQ) & bmHXFRDNIRQ) != bmHXFRDNIRQ) ;
                dum = Hrreg(rHRSL) & 0x0F;                      // Get transfer
result
                if (dum == hrNAK)
                        {
                        Transfer_In_Progress = 1;
                        NAKct += 1;
                                if (NAKct == NAKLimit)
                                        return(hrNAK);
                                else
                                        {
                                        Hwreg(rSNDBC,0);                // Flip
FIFOs
                                        Hwreg(rSNDFIFO,CurrentFB);
                                        Hwreg(rSNDBC,CurrentBC);        // Flip
FIFOs back
                                        L7_ON                            // Scope
pulse
                                        Hwreg(rHIRQ,bmHXFRDNIRQ);       // Clear
the IRQ
                                        Hwreg(rHXFR,(tokOUT | ep));     // Launch
an OUT1 transfer
                                        L7_OFF
                                        }
                        }
                else if (dum == hrACK)
                        {
                        Available_Buffers += 1;
                        NAKct = 0;
                        Transfer_In_Progress = 0;               // Finished this
transfer
                        if (Available_Buffers != 2)             // Still some
data to send
                                {
                                CurrentBC = PendingBC;
                                CurrentFB = PendingFB;
                                Hwreg(rSNDBC,CurrentBC);
                                L7_ON                            // Scope pulse
                                Hwreg(rHIRQ,bmHXFRDNIRQ);       // Clear the IRQ
                                Hwreg(rHXFR,(tokOUT | ep));     // Launch an OUT1
transfer
                                L7_OFF
                                }
                        }
                else return(dum);
        }
        while(Transfer_In_Progress);
}
while(Available_Buffers!=2);            // Go until both buffers are available
(have been sent)
return(0);
}
```

Related Parts		
MAX3421E	USB Peripheral/Host Controller with SPI Interface	Free Samples

More Information
For Technical Support: http://www.maximintegrated.com/support
For Samples: http://www.maximintegrated.com/samples
Other Questions and Comments: http://www.maximintegrated.com/contact

---