National Semiconductor

# USBN9602 Bus Powered Applications

**By Jim Lyle, National Semiconductor**

National Semiconductor's USBN9602 *Universal Serial Bus Function Controller* is a flexible product that was designed primarily for use in a wide variety of "self powered" USB node applications. In its standard configuration it is not suitable for "bus powered" applications because it's suspend current too high.

Nonetheless, the USBN9602 can be used effectively even in bus powered applications with only a few minor modifications to its external circuitry. With these changes the suspend current drops well below the 500 uA limit established by the USB specification.

This paper discusses the hardware modifications necessary, shows examples of the associated firmware, and then discusses the results obtained in actual testing. The code fragments shown are available on the world wide web.[1]

## *Hardware Modifications*

The recommended external USBN9602 circuitry for bus powered operation is shown in Figure 1. This circuit is drawn assuming the MICROWIRE interface[2]. Note that there are only minor differences between this circuit and the one necessary for self powered operation. The only additional components are D1-D3, and R4-R5. In addition, this circuit requires a general purpose output bit, and an input that can either interrupt the microcontroller[3] or "wake" it up out of its own standby mode.

### Clock Stopping

The USBN9602 does not contain any provision for stopping it's clock oscillator. Yet this oscillator alone consumes several milliamps when active. It must be stopped for bus powered operation.

1. See http://www.national.com/sw/USB/ for this and other USBN9602 designer resources.
2. The parallel mode connections will be similar, with the primary differences in the pullup/pulldown details.
3. The USBN9602 requires an external microcontroller of some type to manage the node application and the high-level USB protocol interactions.

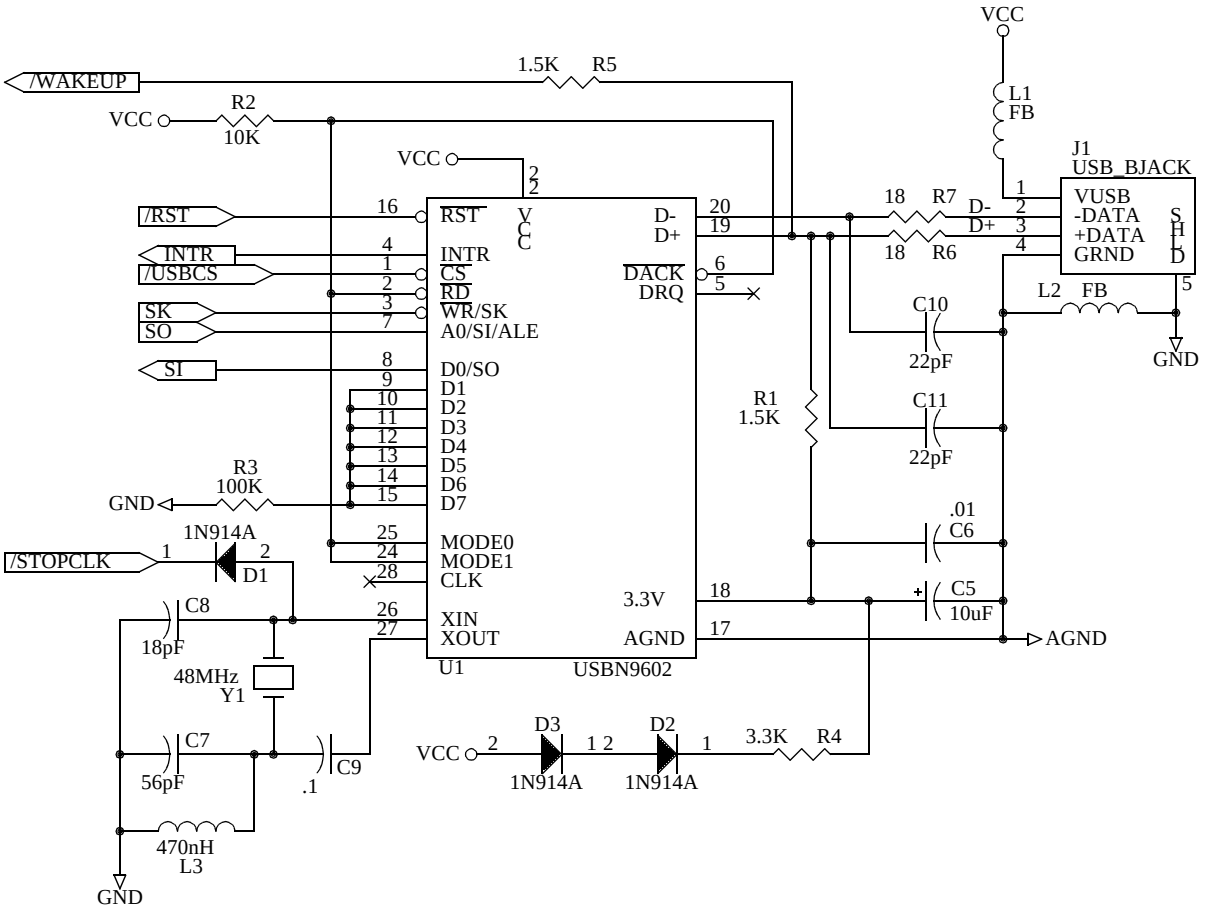


**FIGURE 1. USBN9602 Connections**

Fortunately, a simple modification to the oscillator circuit (Figure 2) stops and starts and stops it reliably. Note the Diode D1, whose anode is attached to the **XIN** pin. This diode is oriented so that it conducts current when the **/STOPCLK** signal is low, clamping the oscillator. When the **/STOPCLK** signal is high, the diode is reverse biased and effectively removed from the circuit.

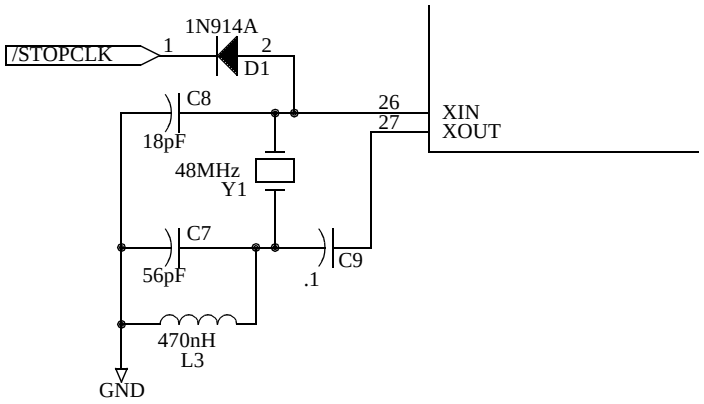


**FIGURE 2. Oscillator Circuit**

## 2 USBN9602 Firmware Description

The diode used is a small-signal type with very low capacitance. This is crucial to minimize the load on the oscillator circuit[1]. Note that the load capacitor's value (C8) has been adjusted downward so that the parallel sum of this value and the diode's capacitance equals the desired result[2].

There are no particular requirements on the **/STOPCLK** signal, except that for best operation it should swing very near VSS on the low side, and VCC on the high side.

## Microcontroller Clock

The USBN9602 has a programmable **CLK** (output) pin which was designed to provide the microcontroller it's clock and eliminate the need for an additional crystal or oscillator. Since the **/STOPCLK** signal is under firmware control, however, this necessarily implies that the microcontroller cannot use the **CLK**. That is because once **/STOPCLK** was serted, the microcontroller would also stop dead in its tracks, and there is no clean way to restart it.

Therefore the microcontroller must have its own clock of some kind[3,4].

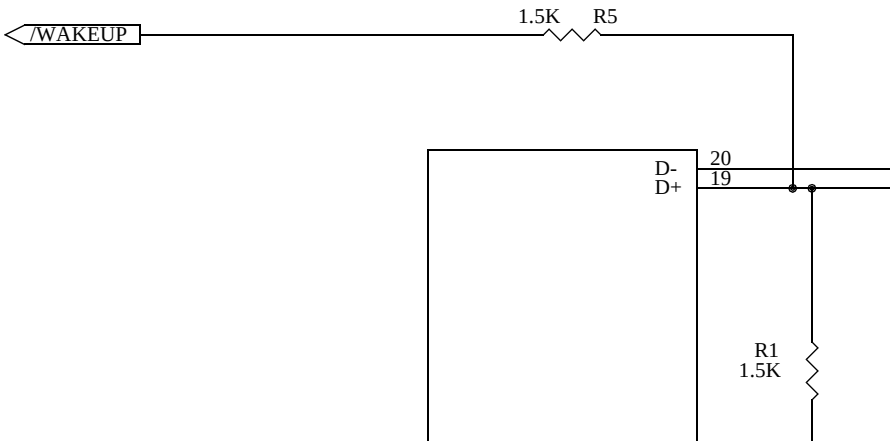


**FIGURE 3. Wake Up Circuit**

## Wake Up

With it's clock stopped, the USBN9602 is deaf and blind. It cannot detect the *resume* or *reset*[5] sequences on the USB lines when it is time to wake up. Some other mechanism is necessary, such as that shown in Figure 3. In suspend, the USB signals are normally held in the "J" state (**D+** high, and **D-** low). The microcontroller must assume responsibility

1. The total capacitance of series-connected capacitors is less than the smallest capacitor in the chain. A 1N914A diode has a capacitance of only about 2 pF, and therefore it limits the total loading to no more than this value.
2. The total load capacitance should be approximately 20 pF.
3. The microcontroller's private clock could be a very slow one, however. This would keep the microcontroller active, but at very low power levels. Then, when the node is signalled to *resume*, the USBN9602 **CLK** pin could be used to provide the higher speed clock necessary for normal operations.
4. There is another alternative. The **/STOPCLK** signal could be generated by an asynchronous flip-flop that was set by the microcontroller, and cleared by either reset or by the **/WAKEUP** signal. In this case the microcontroller could potentially still use the **CLK** pin.
5. Upon "restarting" (warm-booting) the host, the bus will first go idle, and this will cause the node to enter the suspend state. Then the host will issue a USB reset, usually without any other prior activity. Therefore the node must be able to go directly from suspend to reset.

for waking the node up upon seeing either the “K” (**D+** low and **D-** high) or “SE0” (both **D+** and **D-**) states[1]. Since USB signalling occurs at essentially 3.3V CMOS levels, most microcontrollers can simply monitor the **D+** line and look for a high-to-low transition. A large-value isolation resistor is used to limit loading and transmission line stub effects that could degrade the signal quality.

The buffer used to receive the **/WAKEUP** signal should be a Schmitt trigger variety with low capacitance, with input thresholds similar to those specified for the USB.

## Transceiver Supply

The USBN9602 contains an internal regulator for the USB transceiver. The regulator consumes some power itself, but it also sources current into the USB cable[2] during suspend. That is because the upstream hub contains 15 KOhm pulldown resistors on each of the USB lines. During suspend the hub floats it’s output drivers, so there is a current path from the USBN9602’s regulator, through R1 and the USB cable, into the **D+** pulldown.

It is not possible to simply turn the regulator off to reduce this current consumption. That would allow the **D+** line to fall, which the hub might interpret as a remote wake up signal. Also, there would be a risk of latchup in the transceiver circuitry.

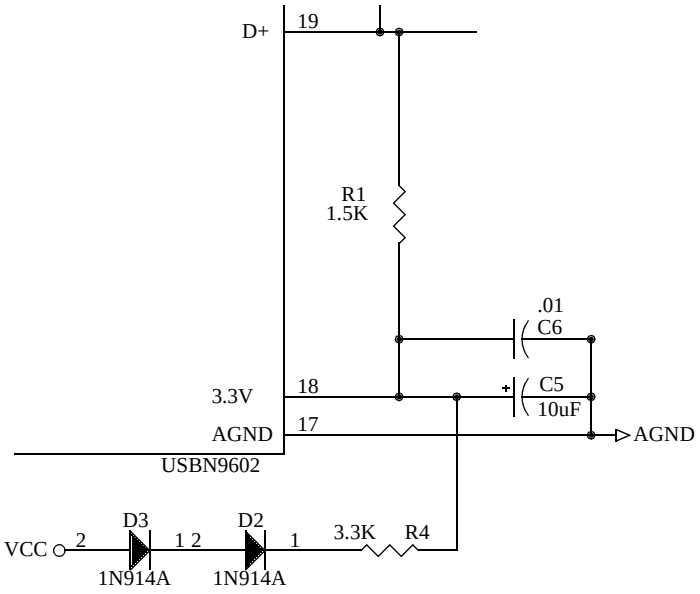


**FIGURE 4. Transceiver “Keep-Alive” Circuit**

However, during suspend the regulator can be turned off if there is an external path that provides sufficient current to maintain **D+** in a valid high state[3]. The circuit shown in Figure 4 is suitable for this purpose. The circuit draws no additional current of its own, the series diodes limit the $V_{OH}$[4] level, and the resistor limits the maximum current. It also supplies the transceiver with just enough current to stay alive. The value of the bulk bypass capacitor (C5) has been

1. These definitions refer to high-speed USB devices. Low-speed devices see the opposite polarities.
2. Of necessity, the cable and the upstream hub are always connected to a bus powered device during suspend.
3. Per the USB Specification, **D+** is defined high when greater than 2.0 Volts.
4. The diodes provide a total drop of about 1.4 Volts.

increased to 10 uF to provide a suitably low impedance (short term) current source when **D+** is driven low (to signal the *resume* or *reset* events).

This circuit is not sufficient for normal operation. The internal regulator must be turned on to ensure reliable device detection and to transmit or receive data properly.

### Miscellaneous Power Savings

Unused inputs should not be allowed to float under any circumstances. Figure 5 shows how the unused inputs should be pulled for the MICROWIRE interface mode. Do not connect these inputs directly to the power rails. Use resistors instead because that will ensure the lowest possible power consumption and will guard against damage if the USBN9602 output buffers should somehow become enabled.

If the **CLK** pin is not used (see "Microcontroller Clock" on page 3), then disable the output by setting the **CODIS** bit in the Clock Configuration (**CCONF)** register. Also, the **INTR** pin has an internal pulldown and should be kept in the low state during suspend. One easy way to accomplish this is to disable the pin by setting the **INTOC[1:0]** field in the Main Control (**MCNTRL)** register to 0.

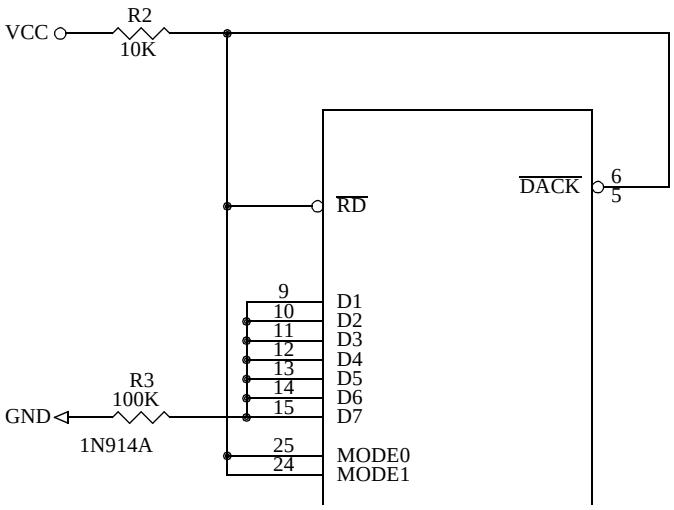


**FIGURE 5. Pullups and Pulldowns**

## *Firmware Requirements*

The firmware is responsible for managing the USBN9602 hardware at all times, including in suspend mode. The firmware must ensure an orderly shutdown, save necessary state, and then reverse these operations when waking.

### Powerup

*Suspend* is signalled on the USB by detecting 3 milliseconds of idle time on the bus. When connecting the cable, the power lines will connect before the signal lines[1]. Also, it may take the host some time to recognize a device and start communicating with it. Therefore it's likely that a false *suspend* event will occur very early in the power-up sequence of

1. This is because the power pins in the connectors are longer than the signal pins, and is necessary because the USB allows hot-plugging devices.

the USBN9602. The node should avoid entering the suspend state at this time, however, because some of the modifications made here might interfere with node detection and enumeration. A suspend timeout counter (See “Suspend Event” on page 7.) is one easy way to accomplish this.

## ALT Event Interrupt Handler

The *suspend* and *resume* events are signalled by the **SD3** and **RESUME** bits in the Alternate Event Register (**ALTEV**). An example of a suitable interrupt service routine for these bits is shown in Figure 6. It is possible that both of these bits might be set, if a resume sequence followed very closely on the suspend condition. For that reason this interrupt service routine gives the *resume* event a higher priority than the *suspend* event. Further, “else if” constructs are used to ensure that only one of the *reset*, *resume*, or *suspend* events is acted upon, in that order of succession.

```
/*********************************************************************/
/* This subroutine handles USB 'alternate' events.                   */
/*********************************************************************/
void usb_alt(void)
  {
        evnt = read_usb(ALTEV);             /*check the events        */

        if(evnt & RESET_A)                  /*reset event             */
          {
            write_usb(NFSR,RST_ST);         /*enter reset state       */
            write_usb(FAR,AD_EN+0);         /*set default address     */
            write_usb(EPC0, 0x00);          /*enable EP0 only         */
            FLUSHTX0;                       /*flush TX0 and disable   */
            write_usb(RXC0,RX_EN);          /*enable the receiver     */

          /*adjusting ints is nec. here in case we were in suspend    */
            write_usb(ALTMSK,NORMAL_ALTMSK);/*adjust interrupts       */
            write_usb(NFSR,OPR_ST);         /*go operational          */
          }

        else if(evnt & RESUME_A)            /*resume  event           */
          {
            write_usb(ALTMSK,NORMAL_ALTMSK);/*adjust interrupts       */
            write_usb(NFSR,OPR_ST);         /*go operational          */
          }

        else if((evnt & SD3)&&(suscntr==0)) /*suspend event           */
          {
            write_usb(ALTMSK,SUSPND_ALTMSK);/*adj interrupts          */
            write_usb(NFSR,SUS_ST);         /*enter suspend state     */
            deep_sleep();                   /*reduce power cons.      */
            suscntr=SUSPND_TO;              /*start suspend cntr      */
          }

        else                                /*spurious alt. event!    */
          {
          }
  }
```

**FIGURE 6. ALT Event Interrupt Handler**

### *Suspend* Event

The *suspend* event handler code is isolated in Figure 7. This code modifies the Alternate Event Mask (**ALTMSK**)[1] and Node Functional State (**NFSR**) registers to reflect their suspend state values. Then the **deep_sleep()** procedure shuts down the node and waits for the *resume* event.

The final step here is an interesting and important one. The USBN9602 asserts **SD3** whenever the bus appears idle for three milliseconds or more. This may happen after a bus reset, which might mean entering suspend just as the enumeration process begins. It might also happen during the suspend process itself[2], which might mean inadvertently re-entering suspend just after exiting. Both of these conditions are undesirable. To avoid them, a suspend time-out counter (suscntr) is initialized to some maximum value here. This counter is decremented during a periodic real-time interrupt until it reaches 0, and the microcontroller will not recognize a new *suspend* event until then. This has the effect of setting a minimum[3] time limit between successive *suspend* events. A value of two to four seconds seems to work well.

```
else if((evnt & SD3)&&(suscntr==0)) /*suspend event          */
  {
    write_usb(ALTMSK,SUSPND_ALTMSK);/*adj interrupts         */
    write_usb(NFSR,SUS_ST);         /*enter suspend state    */
    deep_sleep();                   /*reduce power cons.     */
    suscntr=SUSPND_TO;              /*start suspend cntr     */
  }
```

**FIGURE 7. *Suspend* Event Handler**

### deep_sleep() Procedure

One example of a **deep_sleep()** procedure is shown in Figure 8. First, USBN9602 state is saved, then altered to reduce power as much as possible (the **INTR** output is disabled and the regulator is turned off). A delay guarantees the USBN9602 enough time to complete it's internal operations, then it's clock is stopped.

Now the microcontroller turns its attention elsewhere. The exact sequence of events here will be application dependent, but should include things like turning off LEDs and extraneous circuitry, then preparing the microcontroller to "halt" or "sleep".

The test circuit that this white paper is based on uses a COP8 microcontroller, which stops all activity in halt mode. The remainder of this procedure only occurs after the **/WAKEUP** signal occurs. First the microcontroller restarts the USB clock and gives it adequate time to stabilize. Then the USBN9602 state is restored. Finally the **ALTMSK** and **NFSR** registers are restored to their normal configurations. The latter steps duplicate those done in the *resume* event handler because the USBN9602 may or may not detect the *resume* event as its clock restarts, and so we cannot count on a *resume* event interrupt.

1. Ordinarily suspend interrupts are enabled and resume interrupts are disabled. Upon entering suspend this state is reversed, so that the USBN9602 will not generate further suspend interrupts, but will interrupt when a *resume* event occurs.
2. The bus is idle during suspend. If the process of shutting down and waking up allows the USBN9602 to add up three milliseconds of "idle" time, then it will flag another **SD3** event.
3. This is not the same as a delay before each and every *suspend* event. The firmware waits only if it has just exited suspend. as soon as that minimum time limit expires, it is free to instantly respond to a new *suspend* event. Since *suspend* events rarely occur close together, this timeout interval will usually be hidden.

```
/*********************************************************************/
/* This subroutine puts the board to sleep, to reduce the overall    */
/* current consumption in the suspend mode.                          */
/*********************************************************************/
void deep_sleep(void)
  {
#ifdef  BUSPOWER
        usb_buf[2]=read_usb(MAMSK);         /*save old mask contents  */
        write_usb(MAMSK,0);                 /*disable interrupts      */
        usb_buf[3]=read_usb(MCNTRL);        /*save old MCNTRL         */
        write_usb(MCNTRL,(usb_buf[3]&NAT)); /*prsv NAT, clr others    */

        long_delay();                       /*wait for writes to end  */
        USBCKOFF;                           /*stop  the 9602 clock    */

      /*Turn everything extraneous on the board off.  The analog sub- */
      /*section will be turned off, so all signals going into it must */
      /*be driven low, including the CS*.                             */
        PORTDD.BIT4= 1; PORTDD.BIT7= 1;     /*turn LEDs OFF           */
        MWSR = 0;                           /*drive SO  low           */
        A2DCSON;                            /*drive CS* low           */
        ANALGOFF;                           /*turn off analog power   */

      /*Set up the wake input                                         */
        WKEN.UWK   = 0;                     /*disable the wake input  */
        WKEDG.UWK  = 0;                     /*wake on falling edge    */
        WKPND.UWK  = 0;                     /*clear any pending wake  */
        WKEN.UWK   = 1;                     /*enable the wake input   */

      /*Stop everything, including the processor clock                */
halt:   PORTGD.BIT7= 1;                     /*halt processor          */

      /*We get here only when awakened.  Start everything back up.    */
      /*Note that the "keep-alive" circuit is the only 3.3V source    */
      /*available until the MCNTRL register is restored.  The time    */
      /*constant in that circuit must be sufficiently long.           */
wake:   ANALGON;                            /*turn on analog power    */
        A2DCSOFF;                           /*restore analog CS*      */
        USBCKENB;                           /*start the 9602 clock    */
        long_delay();                       /*wait for the 9602 clk   */
        write_usb(MCNTRL,usb_buf[3]);       /*restore MCNTRL          */
        write_usb(MAMSK, usb_buf[2]);       /*re-enable interrupts    */
        write_usb(ALTMSK,NORMAL_ALTMSK);    /*adjust interrupts       */
        write_usb(NFSR,OPR_ST);             /*go operational          */
#endif
  }
```

**FIGURE 8. deep_sleep() Procedure**

### *Resume* Event

The corresponding *resume* event handler is shown in Figure 9. This handler simply restores the **ALTMSK** and **NFSR** registers to their normal operational values. Note that since the **deep_sleep()** procedure already does this, the *resume* event handler is often redundant. A separate resume handler is included for completeness, and in case the **deep_sleep()** procedure is not included or is empty (in self powered applications, for example).

### *Reset* Event

The *reset* event handler appears in Figure 10. This is relevant to the suspend discussion because in the case of a warm-boot, a bus reset will signal the exit from suspend and there may or may not be a corresponding *resume* event. Just as in the resume case, therefore, the **ALTMSK** and **NFSR** registers must be modified to their normal operation values[1].

```
else if(evnt & RESUME_A)            /*resume  event          */
  {
    write_usb(ALTMSK,NORMAL_ALTMSK);/*adjust interrupts      */
    write_usb(NFSR,OPR_ST);         /*go operational         */
  }
```

**FIGURE 9.** ***Resume*** **Event Handler**

```
if(evnt & RESET_A)                  /*reset event            */
  {
    write_usb(NFSR,RST_ST);         /*enter reset state      */
    write_usb(FAR,AD_EN+0);         /*set default address    */
    write_usb(EPC0, 0x00);          /*enable EP0 only        */
    FLUSHTX0;                       /*flush TX0 and disable  */
    write_usb(RXC0,RX_EN);          /*enable the receiver    */

  /*adjusting ints is nec. here in case we were in suspend   */
    write_usb(ALTMSK,NORMAL_ALTMSK);/*adjust interrupts      */
    write_usb(NFSR,OPR_ST);         /*go operational         */
  }
```

**FIGURE 10.** ***Reset*** **Event Handler**

## *Results*

Several devices from different lots were tested for this paper at different voltages and temperatures, using the following conditions (which represent the desired USBN9602 suspend state:

1. **INTR** low.
2. **XIN** low.
3. **XOUT** high.
4. Internal VREG ON[2] (this means the VGE bit in the **MCNTRL** register should be set).
5. The NAT bit in the **MCNTRL** register set
6. **D+** high (at about 3.3V)
7. **D-** low
8. All other inputs stable high or low.

1. Strictly speaking, if the **deep_sleep()** procedure is always included and if it does not return until a *resume* or *reset* event has been detected, then the redundant code in the resume and reset handlers can be eliminated. This firmware was not written that way though because it's intent is to support a variety of microcontrollers in both bus powered and self powered applications.
2. For this test the voltage regulator was turned on because the tester used could not simulate the "keep-alive" circuit or the current path into the hub's pulldown resistor. The regulator consumes a similar level of current, and so was enabled to simulate cable load.

The worst-case current draw found under these conditions was 210 uA, with most devices close to 130 uA.

One USBN9602 Evaluation Board was modified per the suggestions in this paper, and that board draws less than 400 uA total, including USBN9602, microcontroller, and all other circuitry combined[1].

## *Summary*

While the USBN9602 was not designed for bus powered operations, minor modifications to its external circuitry allow the suspend current to be reduced to levels well within the requirements of the USB Specification.

1. Of this total, approximately 180 uA is drawn by the hub's pulldown resistor alone.



**National Semiconductor Corporation**
Tel: 1-800-272-9959
Fax: 1-800-737-7018
Email: support@nsc.com

**National Semiconductor Europe**
Fax: (+49) 0-180-530 85 86
Email: europe.support@nsc.com
Deutsch Tel: (+49) 0-180-530 85 85
English Tel: (+49) 0-180-532 78 32

**National Semiconductor Asia Pacific Customer Response Group**
Tel: 65-254-4466
Fax: 65-250-4466
Email: sea.support@nsc.com

**National Semiconductor Japan Ltd.**
Tel: 81-3-5620-6175
Fax: 81-3-5620-6179