National Semiconductor

# USBN9602

# Firmware Description

**By Jim Lyle, National Semiconductor**

This white paper shows a number of C procedures and code fragments which illustrate how to manipulate and use the USBN9602 Universal Serial Bus Function Controller. These examples are extracted from the firmware for a USBN9602 Evaluation Board[1], which uses a COP8 microcontroller[2]. Except where specifically noted, however, the code discussed here is generic, and does not rely on a specific microcontroller or implementation.

This paper assumes a familiarity with the USBN9602 architecture. Please refer to the specification for detailed descriptions of this device and its operation.

The complete listing for the firmware discussed here is included at the end of this paper.

## Firmware Structure

The Evaluation Board for which this firmware was written is complex, and has a number of different functions and interfaces. A block diagram is shown in Figure 1. The firmware is correspondingly complex, with "driver" sections for all the peripherals shown. Yet the basic structure of the firmware is relatively simple, with three basic elements:

1. Initialization

   This is the code that initializes the microcontroller and all peripheral circuitry.

2. "Main" Loop

   This is the code loop that keeps the microcontroller busy when there is nothing more pressing to do. These are the tasks that must be done regularly, but are not time critical and can be interrupted.

3. Interrupt Handlers

   These are the tasks that are time critical, and that must be serviced as soon as conditions warrant. These also tend to be the tasks that are asynchronous or that occur irregularly. Here is where most of the "driver" code is found.

1. A limited number of these Evaluation Boards are available from National Semiconductor. Please contact your distributor or sales representative for more information.
2. For more information, see: http://www.national.com

This firmware uses a hierarchical structure. At the lowest level are the simple, primitive routines that handle the most basic functions. These are used as building blocks to form the more complex routines at the higher levels of the hierarchy. This hierarchical structure is straightforward, easily understood, and readily adapted and tested. It is *not* necessarily the most efficient, because numerous subroutine calls and returns, and register pushes and pops are used to keep things simple.

The remaining sections this paper focus on the USBN9602 and follow this hierarchical structure upwards. First comes a discussion of the code primitives used, followed by discussions of the initialization, “main” loop, and interrupt code. Afterwards, a special section will discuss some of the USB “standard” device requests, and how they were implemented in this firmware.

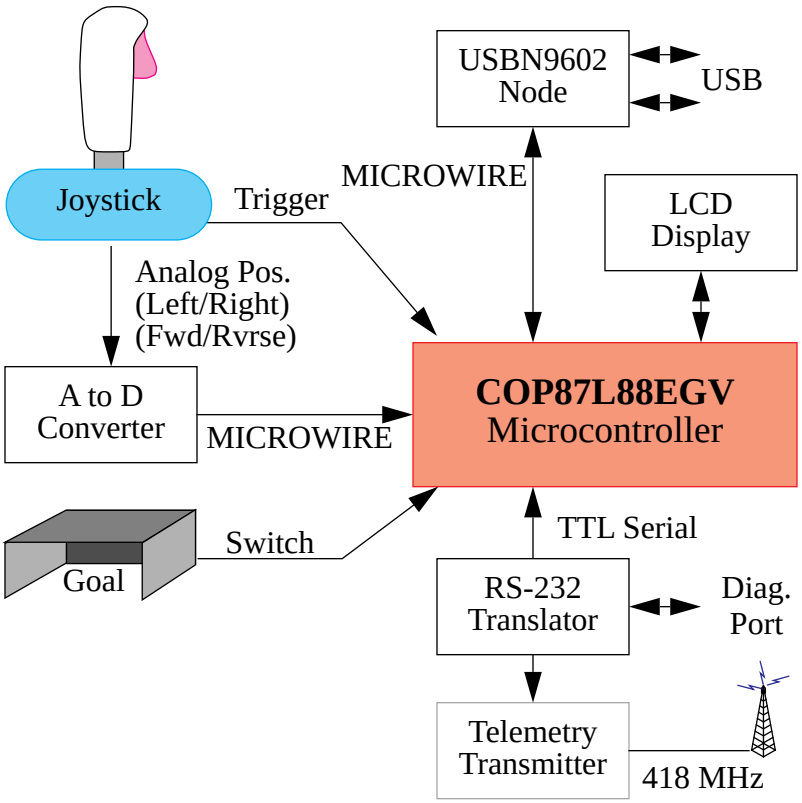


**FIGURE 1. Block Diagram of USBN9602 Evaluation Board**

## *Code Primitives*

The following code primitives are the simplest, most basic functions supported. These examples assert control signals, select Flash memory banks, strobe data in and out of the memory, and so on. By themselves, each of these primitives forms only a portion of a complete NAND Flash access.

### read_usb and write_usb

The *read_usb* and *write_usb* procedures are shown in Figure 2 and Figure 3 respectively. These procedures are the most basic and most often called, and are also the only ones that of necessity must be system specific. Therefore this paper won’t delve too deeply into the details here.

The USBN9602 supports either a MICROWIRE interface or a byte-wide interface. The latter is divided into multiplexed and non-multiplexed variants, with or without DMA. This firmware and the evaluation board for which it was written assumes the MICROWIRE interface, but that affects only these two routines and the macros that they use.

```
/*********************************************************************/
/* This subroutine reads the USB register whose address is given.    */
/*********************************************************************/
byte read_usb(byte adr)
  {
USBCSON;                          /*turn on CS*/
MWADRCMD(adr,USBREAD);            /*send cmd and addr*/
MWOUT(0);                         /*send dummy data  */
USBCSOFF;                         /*turn off CS*/
return(MWSR);                     /*return the result*/
  }
```

**FIGURE 2.** ***read_usb*** **Procedure (page 49 of listing)**

Other implementations and other interfaces will need to rewrite these procedures for their particular systems. Note that because these procedures are called so often by the rest of the code, it is essential that they are efficient. That is why macros were used instead of other subroutines.

```
/*********************************************************************/
/* This subroutine writes the USB register whose address is given.   */
/*********************************************************************/
void write_usb(byte adr, byte dta)
  {
USBCSON;                          /*turn on CS*/
MWADRCMD(adr,USBWRITE);           /*send cmd and addr*/
MWOUT(dta);                       /*send the data*/
USBCSOFF;                         /*turn off CS*/
  }
```

**FIGURE 3.** ***write_usb*** **Procedure (page 50 of listing)**

## Macros

Macros are used throughout this firmware to simplify code where redundant operations are necessary. Two representative macros are shown in Figure 4 and Figure 5. The first of these flushes a FIFO (in this case the transmitter associated with endpoint[1] 0). The second enables the transmitter associated with endpoint 0.

A brief review of the USBN9602 architecture is in order here[2]. The USBN9602 supports up to 7 total endpoints. There is a FIFO associated with each endpoint[3]. The normal usage depends on the FIFO type:

- Transmit FIFO

1. “Endpoints” are defined in the USB specification. Essentially they are sub-channels to or from which data can be transferred by the host. A node may contain up to 16 endpoints. Endpoint 0 is required, is bidirectional, and is used for system control transfers. Other endpoints are optional, and are unidirectional.
2. Refer to the USBN9602 specification for more details.
3. Note that endpoint 0 is a special case because it is bidirectional. The transmit and receive functions of that endpoint share the FIFO, and only one or the other should be enabled at any given time.

Flush FIFO (optional: necessary only if FIFO already contains data).

Load FIFO with data (by writing to the data register).

Set the associated TX_LAST[1] bit to indicate that all data has been transferred to the FIFO.

Choose the appropriate PID by setting the associated TX_TOGL bit (for DATA1) or clearing it (for DATA0).[2]

Set the associated TX_EN bit to enable the transmitter.

Wait for the associated interrupt (indicates transmission has completed).

- Receive FIFO

  Set the associated RX_EN bit to enable the receiver.

  Wait for the associated interrupt (indicates data has been received)

  Read data from FIFO (by reading the data register).

  FLUSH the FIFO to discard any remaining data.

Note that all non-zero endpoints must be enabled before any of the preceding operations will have any effect (endpoint 0 is required and so is always enabled). This is done in the setconfiguration procedure (Figure 23) and will be discussed in detail there.

```
/* Flush and disable the USB TX0 ***************************************/
#define FLUSHTX0 {write_usb(TXC0,FLUSH);}
```

**FIGURE 4. *FLUSHTX0* Macro (page 32 of listing)**

The *FLUSHTX0* Macro in Figure 4 simply sets the FLUSH bit in the transmitter 0 control register (TXC0), and clears all other bits (including TX_EN, thereby disabling the transmitter). The FLUSH bit clears itself when the operation completes and does not need to be explicitly reset.

```
/* enable TX0, using the appropriate DATA PID *************************/
#define TXEN0_PID                                                      \
  { if(dtapid.TGL0PID) write_usb(TXC0,TX_TOGL+TX_EN); /*DATA1*/\
    else write_usb(TXC0,TX_EN);                       /*DATA0*/\
    dtapid.TGL0PID=!dtapid.TGL0PID;}
```

**FIGURE 5. *TXEN0_PID* Macro (page 32 of listing)**

The *TXEN0_PID* Macro in Figure 5 sets the TX_EN bit, and sets or resets the TX_TOGL bit depending on the state of the TGL0PID memory flag (which is then updated accordingly). For other endpoints the equivalent code must also set the TX_LAST bit.

There are numerous other macros that perform similar functions for receive operations and for other endpoints. Please review these in the listing.

1. There is no corresponding bit for endpoint 0.
2. See the USB specification for a detailed description of the DATA0 and DATA1 PID mechanism. Essentially, these are tags associated with each transfer that must “toggle” (alternate) in a predefined manner.

## *Initialization*

The USBN9602 must be initialized at power-on and whenever the host sends a reset command over the USB interface. This initialization is straightforward, and is done by the *init_usb* procedure shown in Figure 6.

The first thing this procedure does is clear the *get_desc* and *usb_cfg* memory flags (more on these later). Then it issues a software reset (by setting a bit in a control register which will clear itself when the reset operation completes) and defines the type of interrupt used. The USBN9602 can provide either low- or high-true interrupts, and either open-drain or push pull in the low-true case, push-pull only in the high-true case (which is the option selected here).

After that this procedure initializes the clock output generator. The USBN9602 uses a 48 MHz clock. That frequency is fixed by requirements in the USB Specification, and it can't be altered. The clock output pin is provided for convenience though. In some cases it can eliminate the need for an additional crystal or oscillator in the system. It is produced by dividing the 48 MHz base clock by an integer, and on the USBN9602 Evaluation Board it is used to provide a 9.6 MHz clock to the COP8 microcontroller.

Next, the node address is set to 0 (the default value)[1], and the endpoint 0 control register (EPC0) is cleared (insuring that it isn't stalled and setting the endpoint sub-address to 0 as well). Note that EPC1 through EPC6 contain an Endpoint Enable (EP_EN) bit that must be set to enable the corresponding endpoint, but endpoint 0 has no such requirement because this endpoint must always be present and enabled.

Then this procedure defines the conditions that should cause an interrupt. This is done by setting bits in a series of mask registers, with each bit corresponding to a particular cause or condition. Each of the endpoints can be masked separately, and there are also separate bits for underruns, overruns, packets received, packets transmitted, system commands and conditions, errors, and so on. Note that the mapping between FIFO and endpoint is as follows:

| Endpoint # | TX FIFO # | RX FIFO # |
|---|---|---|
| 0 | FIFO 0 (half duplex) | |
| 1 | 1 | - |
| 2 | - | 1 |
| 3 | 2 | - |
| 4 | - | 2 |
| 5 | 3 | - |
| 6 | - | 3 |

**TABLE 1. Endpoint Number to FIFO Number Map**

Like-numbered FIFOs (such as TX FIFO 1 and RX FIFO 1) are physically separate and may be used completely independently. They share only the numbers.

Since the endpoint 0 is effectively half-duplex, this procedure flushes and disables the transmitter before enabling the receiver. Then the USBN9602 is made operational by writing OPR_ST to the Node Functional State Register (NFSR), and finally the Node Attach (NAT). Up until this point the USB transceiver has been forcing an SE0[2], which essentially "cloaks" the node and makes it invisible to the host system. Once the NAT bit is set, however, the host will detect that a new node (this one) is present and ready to be enumerated.

1. USB devices power up with address 0. The USB Specification defines an "enumeration" process that sequentially queries each device and then assigns it a unique address.
2. Single-Ended Zero state, as defined by the USB Specification.

```
/*********************************************************************/
/* This subroutine initializes the 9602.                             */
/*********************************************************************/
void init_usb(void)
  {
      /*toss out any previous state ********************************/
status.GETDESC=0;
usb_cfg = 0;

      /*give a software reset, then set ints to active high push pull */
write_usb(MCNTRL,SRST); write_usb(MCNTRL,INT_H_P);

      /*initialize the clock generator *****************************/
      /* prior to this point, the clock output will be 4 Mhz.  After, */
      /* it will be (48 MHz/CLKDIV)                                  */
write_usb(CCONF,CLKDIV-1);

      /*set default address, enable EP0 only ***********************/
write_usb(FAR,AD_EN+0); write_usb(EPC0, 0x00);

      /*set up interrupt masks *************************************/
write_usb(NAKMSK,NAK_O0);                        /*NAK evnts*/
write_usb(TXMSK,TXFIFO0+TXFIFO1+TXFIFO3);        /*TX events*/
write_usb(RXMSK,RXFIFO0+RXFIFO1+RXFIFO3);        /*RX events*/
write_usb(ALTMSK,SD3+RESET_A);                   /*ALT evnts*/
    /*this is modified in the */
    /*suspend-resume routines */
    /*so if any change is made*/
    /*here it needs to be ref-*/
    /*lected there too.       */
write_usb(MAMSK,(INTR_E+RX_EV+NAK+TX_EV+ALT));

      /*enable the receiver and go operational *********************/
FLUSHTX0;                          /*flush TX0 and disable   */
write_usb(RXC0,RX_EN);             /*enable the receiver     */

write_usb(NFSR,OPR_ST);            /*go operational          */
write_usb(MCNTRL,INT_H_P+NAT);     /*set NODE ATTACH         */
  }
```

**FIGURE 6.** ***init_usb*** **Procedure (initializes the USBN9602)**

## *“Main” Loop*

Typically, USBN9602 related firmware will be interrupt driven, because of response time requirements and because it is difficult to predict when or how the host will initiate a transfer. It is possible to do some things outside of interrupts however. For example, a FIFO can be either emptied or filled in the “main” (or executive) loop if the data rate is low or bursty enough.

Consider the code fragment in Figure 7, (taken from this firmware’s ‘main’ loop). The USBN9602 Evaluation Board uses endpoint 6 to write data to a small LCD display. This LCD is a slow device, and is in fact so slow that it would seriously hamper system performance if writes to the LCD were done within the interrupt routines. Instead, when the host transfers data to that endpoint, the firmware quickly notes the fact, handshakes with the host, and then puts the number of characters received into the memory variable *rcount3*. Since LCD updates are infrequent, it is possible to transfer data directly out of the FIFO into the LCD almost at leisure.

```
      if (rcount3>0)
{
  putch(read_usb(RXD3));    /*xfer RX3 FIFO --> LCD   */
  --rcount3;                /*decriment count         */
```

**FIGURE 7. Extract from main() (page 39 of listing)**

## *Interrupt Handlers*

Of course most USB operations must happen at a much faster pace than those discussed in the preceding section. USB devices must handshake with the host in a timely fashion if it is to properly enumerate and then remain configured. Therefore most USBN9602 operations will typically be interrupt driven, and that is why this component has a sophisticated interrupt structure. The numerous interrupt mask bits were already mentioned. For each of these mask bits there is also a corresponding bit in an event register that can be used by firmware to quickly determine which interrupt occurred.

### usb_isr

This firmware's primary usb interrupt service routine is called *usb_isr*, and is shown in Figure 8 through Figure 11 (it is divided up because it is long, and to focus the discussion). Though long, it is relatively straightforward. In essence, it tests the bits in the event registers in sequence (the exact sequence used corresponds to the 'priority' chosen for the events).

```
/*********************************************************************/
/* This is the interrupt service routine for USB operations          */
/*********************************************************************/
void usb_isr(void)
  {
evnt = read_usb(MAEV);              /*check the events       */

PORTDD.USB_I = 0;                   /*turn on  intr. LED     */

if (evnt & NAK)
  {
    evnt=read_usb(NAKEV);           /*check the NAK events   */
    if      (evnt&NAK_O0) nak0();   /*endpoint 0             */
    else if (evnt&NAK_O1) nak1();   /*endpoint 2             */
    else if (evnt&NAK_O2) nak2();   /*endpoint 4             */
    else if (evnt&NAK_O3) nak3();   /*endpoint 6             */
    else                            /*some other TX event    */
      {
      }
  }
```

**FIGURE 8. *usb_isr* Procedure (page 63 of listing)**

The first thing this procedure does is to read the Main Event Register (MAEV) and store it in the memory variable *evnt*. In this case it is primarily to improve performance[1], but in some other cases it can be crucial for another reason. Many of the condition and status bits in the USBN9602 are cleared when read[2]. This means there is only one chance to read them. Please keep this very important fact firmly in mind.

Next this procedure turns on a diagnostic LED, then enters a series of tests to determine what class of interrupt occurred. The first of these tests is contained within Figure 8. The NAK bit will be set in the MAEV register (and therefore now in our *evnt* variable) if an unmasked NAK event occurred. A NAK event means that the USBN9602 generated a NAK handshake on the USB bus for a particular endpoint.

If a NAK event occurred then this procedure moves deeper and reads the NAK Event Register (NAKEV). Once again it stores this into the *evnt* memory variable, and this means that the previous (MAEV) contents are lost. That is ok though because this firmware will only service one interrupt at a time.

One at a time, the NAKEV bits are checked. One or more may be active, but only the first one found will target to the appropriate dedicated procedure. If there are any others they will be taken care of next time.

```
else if (evnt & RX_EV)
  {
    evnt=read_usb(RXEV);            /*check the RX events     */

    if      (evnt&RXFIFO0) rx_0();  /*endpoint 0              */
    else if (evnt&RXFIFO1) rx_1();  /*endpoint 2              */
    else if (evnt&RXFIFO2) rx_2();  /*endpoint 4              */
    else if (evnt&RXFIFO3) rx_3();  /*endpoint 6              */
    else                            /*some other RX event     */
      {
      }
  }
```

**FIGURE 9.** ***usb_isr*** **Procedure Continued (page 63 of listing)**

If no NAK events are responsible for the interrupt, then this procedure next looks for RX events (in Figure 9) or TX events (in Figure 10) in that order. The structure of the tests in each case is identical to that used for the NAK events.

```
else if (evnt & TX_EV)
  {
    evnt=read_usb(TXEV);            /*check the TX events     */
    if      (evnt&TXFIFO0) tx_0();  /*endpoint 0              */
    else if (evnt&TXFIFO1) tx_1();  /*endpoint 1              */
    else if (evnt&TXFIFO2) tx_2();  /*endpoint 3              */
    else if (evnt&TXFIFO3) tx_3();  /*endpoint 5              */
    else                            /*some other TX event     */
      {
      }
  }
```

**FIGURE 10.** ***usb_isr*** **Procedure Continued (page 64 of listing)**

If the interrupt is not a NAK, RX, or TX event, then it must either be an alternate event or a spurious interrupt. This procedure examines the Alternate Event Register (ALTEV) to figure out which (as shown in Figure 11), and in the former case calls the *usb_alt* procedure (more on this later).

1. The COP8 microcontroller can access memory variables much faster than registers in the USBN9602 (which are connected via the MICROWIRE port). It can also use special bit test instructions on memory variables.
2. Fortunately, many of the event bits themselves only clear when the event they signal has been serviced. This helps to guarantee that all interrupts will be handled, even if multiple interrupts occur at the same time.

Finally, the *usb_isr* procedure must do some housekeeping. The USBN9602 produces an interrupt output that is a logic level, and this ordinarily remains asserted until all interrupts have been asserted. The COP8 microcontroller excepts to see edges, though: one for each interrupt. The USBN9602 can be made to simulate these interrupt edges by momentarily disabling, then re-enabling the interrupts. That is done here by first clearing and then restoring the Main Mask Register (MAMSK) contents. Then the diagnostic LED is turned off.

```
else if (evnt & ALT) usb_alt();     /*alternate event?        */

else                                /*spurious event!         */
  {
  }

     /*the 9602 produces interrupt LEVELS, the COP looks for edges.  */
     /*So we have to fool the 9602 into producing new edges for us   */
     /*when we are ready to look for them.  We do this by temporarily*/
     /*disabling the interrupts, then re-enabling them.              */
evnt=read_usb(MAMSK);               /*save old mask contents  */
write_usb(MAMSK,(0));               /*disable interrupts      */
write_usb(MAMSK,evnt);              /*re-enable interrupts    */

PORTDD.USB_I = 1;                   /*turn off intr. LED      */
  }
```

**FIGURE 11. *usb_isr* Procedure Continued (page 64 of listing)**

### usb_alt

The *usb_isr* procedure determines which interrupt to service and routes execution accordingly. It doesn't actually service the interrupt per se. That task is left to other procedures. One of these is the *usb_alt* procedure previously mentioned. This procedure examines the Alternate Event Register (ALTEV) and then acts accordingly. These the USB 'reset', 'suspend', and 'resume' events. These primarily effect the Node Functional State Register (NFSR). Refer to Figure 12 for more details.

Note that the suspend and resume code segments adjust the interrupt mask. This is necessary because the 'suspend' state is signaled by bus inactivity, and if the bus were inactive for a long time it might otherwise result in a long series of interrupts. Therefore the suspend interrupt is masked here, and unmasked again when a 'resume' event occurs.

### rx_0

The *rx_0* procedure is shown in Figure 13 through Figure 15. This one is busy, handling the receiver traffic for endpoint 0, including all the standard USB device requests. The first thing it does is read the Receive Status Register 0 (RXS0) into the *rxstat* memory variable (in this case necessary not just for performance, but because all the important bits are cleared when read).

First, the packet waiting in the FIFO must be either a 'setup' packet, or an 'out' packet[1]. If it is a 'setup' packet then the entire packet (known already to be 8 bytes long) is transferred into a secondary buffer, and then both the receiver and transmitter are flushed and disabled.

1. 'Setup', 'out', and 'in' packets are defined in the USB Specification.

```
/*********************************************************************/
/* 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     */
    write_usb(NFSR,OPR_ST);         /*go operational          */
  }

else if(evnt & SD3)                 /*suspend event           */
  {
    write_usb(ALTMSK,RESUME_A+RESET_A);  /*adjust interrupts  */
    write_usb(NFSR,SUS_ST);         /*enter suspend state     */
  }

else if(evnt & RESUME_A)            /*resume  event           */
  {
    write_usb(ALTMSK,SD3+RESET_A);       /*adjust interrupts  */
    write_usb(NFSR,OPR_ST);         /*go operational          */
  }

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

**FIGURE 12.** ***usb_alt*** **Procedure (page 51 of listing)**

```
/*********************************************************************/
/* This subroutine handles RX events for FIFO0 (endpoint 0)          */
/*********************************************************************/
void rx_0(void)
  {
rxstat=read_usb(RXS0);              /*get receiver status     */

      /*is this a setup packet? ************************************/
if(rxstat & SETUP_R)
  {
  /*read data payload into buffer then flush/disble the RX ****/
    for(desc_idx=0; desc_idx<8; desc_idx++)
      usb_buf[desc_idx] = read_usb(RXD0);

      FLUSHRX0;                     /*make sure the RX is off */
      FLUSHTX0;                     /*make sure the TX is off */
```

**FIGURE 13.** ***rx_0*** **Procedure (initializes the USBN9602)**

Decoding the 'setup' packet is shown in Figure 14[1]. The first thing checked is whether or not it is a standard or non-standard request. The latter is ignored. The former is further decoded into request type. Some of the simpler requests are

managed right here. Others require more work in additional procedures. In all cases though, 0 or more bytes are loaded into the transmitter FIFO. These standard requests will be discussed in more detail later.

```
    if ((usb_buf[0]&0x60)==0x00)    /*if a standard request   */
      switch (usb_buf[1])           /*find request target     */
{
  case CLEAR_FEATURE:
    clrfeature();
    break;

  case GET_CONFIGURATION:
    write_usb(TXD0,usb_cfg);/*load the config value   */
    break;

  case GET_DESCRIPTOR:
    getdescriptor();
    break;

  case GET_STATUS:
    getstatus();
    break;

  case SET_ADDRESS:
  /*set and enable new address for endpoint 0, but set*/
  /*DEF too, so new address doesn't take effect until */
  /*the handshake completes                           */
    write_usb(EPC0,DEF);
    write_usb(FAR,usb_buf[2] | AD_EN);
    break;

  case SET_CONFIGURATION:
    setconfiguration();
    break;

  case SET_FEATURE:
    setfeature();
    break;

  default:                  /*unsupported standard req*/
    break;
}
    else                            /*if a non-standard req.  */
      {
      }
```

**FIGURE 14.** ***rx_0*** **Procedure Continued (page 56 of listing)**

The 'setup' packet code continues in Figure 15. For a 'setup' packet the proper handshake consists of 0 or more data bytes (already loaded by the preceding code), transmitted with a DATA1 PID. Therefore for 'setup' packets the transmitter is enabled on exiting rx_0, and the receiver will not be[1].

1. The details of the packet format are beyond the scope of this paper. See the USB Specification.

1. The receiver will be re-enabled ultimately by the *tx_0* procedure that follows.

```
  /*the following is done for all setup packets.  Note that if*/
  /*no data was stuffed into the FIFO, the result of the fol- */
  /*lowing will be a zero-length response.                    */
      write_usb(TXC0,TX_TOGL+TX_EN);  /*enable the TX (DATA1) */
      dtapid.TGL0PID=0;               /*store NEXT PID state  */
  }

      /*if not a setup packet, it must be an OUT packet ***************/
else
  {
    if (status.GETDESC)             /*get_descr status stage? */
      {
      /*test for errors (zero length, correct PID)            */
if ((rxstat& 0x5F)!=0x10)   /*length error??          */
  {
  }

status.GETDESC=0;           /*exit get_descr mode     */
FLUSHTX0;                   /*flush TX0 and disable   */
      }

    write_usb(RXC0,RX_EN);          /*re-enable the receiver  */
  }

      /*we do this stuff for all rx_0 events *************************/
  }
```

**FIGURE 15. *rx_0* Procedure Continued (page 57 of listing)**

If the packet is an ‘out’ packet instead, then it is handled differently. If it is the final handshake of a get_descriptor sequence[1] (determined by the *status*.GETDESC flag) then the transmitter is flushed (and the GETDESC flag is cleared). Otherwise this firmware ignores it[2]. No response is necessary in either case, so the receiver is re-enabled.

There are similar receiver procedures for each active endpoint. The other endpoints may not need to deal with ‘setup’ packets, but the overall structure will be:

1. Check the status of the packet (and transfer it if appropriate).
2. Decode the packet and act accordingly.
3. If endpoint 0 then queue up a response and enable the transmitter (if necessary), otherwise re-enable the receiver.
4. If not endpoint 0, re-enable the receiver.

### tx_0

The tx_0 procedure is shown in Figure 16. In general, a transmitter interrupt means that a previously queued transmission has completed, or has encountered an error. Therefore this procedure first copies the Transmitter Status Register 0 (TXS0) to the *txstat* variable, then examines the status bits in the latter and flushes the FIFO if all is well.

FIFOs limit the size of USB packets. A total transfer may be much larger than the limiting FIFO, however, and so must be broken down to span multiple packets. Whenever a transmission completes successfully, *tx_0* checks whether there are any additional packets to send. In this case that corresponds to the get_descriptor sequence mentioned before. If

1. This may well happen before the entire descriptor has been sent, do the firmware must be ready for it at any time.
2. Note that it is possible to use endpoint 0 to transfer data, in which case ‘out’ packets will be expected for that purpose and the corresponding code would be placed here.

there are additional packets they are loaded into the transmitter FIFO and the transmitter is re-enabled. If there are no additional packets then the receiver is enabled.

There are similar transmitter procedures for each active endpoint. The overall structure will be:

1. Check the status and handle errors if necessary.
2. Queue up the next packet and re-enable the transmitter (if necessary), otherwise re-enable the receiver (endpoint 0) or do nothing (not endpoint 0).

```
/*********************************************************************/
/* This subroutine handles TX events for FIFO0 (endpoint 0)          */
/*********************************************************************/
void tx_0(void)
  {
    byte lim;

txstat=read_usb(TXS0);              /*get transmitter status  */

      /*if a transmission has completed successfully, check to see if */
      /*we have anything else that needs to go out, otherwise turn the*/
      /*receiver back on ***********************************************/
if ((txstat & ACK_STAT) && (txstat & TX_DONE))
  {
    FLUSHTX0;                       /*flush TX0 and disable   */

  /*the desc. is sent in pieces; queue another piece if nec.  */
    if(status.GETDESC)
      {
lim=desc_idx+8;             /*set new max limit       */

      /*move the data into the FIFO   */
for(; ((desc_idx<lim)&&(desc_idx<desc_sze)); desc_idx++)
  get_desc();
TXEN0_PID;                  /*enable TX, choose PID   */
      }
    else
      {
write_usb(RXC0,RX_EN);      /*re-enable the receiver  */
      }
  }

      /*otherwise something must have gone wrong with the previous ****/
      /*transmission, or we got here somehow we shouldn't have ********/
       else
 {
 }

      /*we do this stuff for all tx_0 events ************************/
  }
```

**FIGURE 16.** ***tx_0*** **Procedure (page 60 of listing)**

### nak0

NAKs are a normal way for a USB device to tell the host that it isn't ready yet to complete a transfer. In most cases it probably isn't necessary to even notice them, until and unless the situation that causes them won't resolve itself.

For example, suppose endpoint 0 is in the midst of transmitting a multi-packet transfer (for that now infamous get_descriptor sequence perhaps). The host may decide to abandon the rest of the transfer though, and it does this by

sending an ‘out’ packet to the endpoint 0 receiver. The receiver isn’t enabled though; the transmitter is. Therefore this ‘out’ packet gets a NAK handshake, and will continue to receive NAKs until something specific is done to break the deadlock. That is the purpose of the *nak0* procedure shown in Figure 17. It merely exits get_descriptor mode, flushes the transmitter, and then re-enables the receiver.

```
/*********************************************************************/
/* This subroutine handles OUT NAK events for FIFO0 (endpoint 0)     */
/*********************************************************************/
void nak0(void)
  {
      /*important note:  even after servicing a NAK, another NAK     */
      /*interrupt may occur if another ‘OUT’ or ‘IN’ packet comes in  */
      /*during our NAK service.                                       */

      /*if we’re currently doing something that requires multiple ‘IN’*/
      /*transactions, ‘OUT’ requests will get NAKs because the FIFO is*/
      /*busy with the TX data.  Since the ‘OUT’ here means a premature*/
      /*end to the previous transfer, just flush the FIFO, disable the*/
      /*transmitter, and re-enable the receiver.                      */
      if (status.GETDESC)                  /*get_descr status stage? */
{
  status.GETDESC=0;                /*exit get_descr mode     */
  FLUSHTX0;                        /*flush TX0 and disable   */
  write_usb(RXC0,RX_EN);           /*re-enable the receiver  */
}

      /*we do this stuff for all nak0 events ************************/
  }
```

**FIGURE 17. *nak0* Procedure (page 62 of listing)**

Such deadlocks are unlikely on other endpoints because only endpoint 0 is bidirectional (half-duplex). Still, there may be other reasons to track NAK occurrences. If so, the structure of those routines will depend on the particular purpose or need.

## *USB Standard Requests*

So far this paper has focused primarily on the nuts and bolts of transferring USB data with the USBN9602 device. This section turns to a discussion of some of the higher-level protocol issues, and in particular some of the USB “Standard” device requests. Implementing these basic functions at some level is required for basic USB compliance. Note that depending on the device “class”, additional device requests may also be required.

### getdescriptor

This paper has already touched on elements of the get_descriptor request. A descriptor is a data block contained within the device that describes what it is or how it works. The host asks for the device’s descriptors when enumerating, so that it can properly assign resources and load the associated driver(s). The descriptors contained in this firmware are defined starting on page 35 of the listing.

```
/*********************************************************************/
/* The GET_DESCRIPTOR request is done here                           */
/*********************************************************************/
void getdescriptor(void)
  {
status.GETDESC=1;           /*enter get_descr mode    */
desc_typ = usb_buf[3];      /*store the type requested*/
if(desc_typ==DEVICE) desc_sze = DEV_DESC_SIZE;

else if(desc_typ==CONFIGURATION) desc_sze = CFG_DESC_SIZE;

      /*adjust size, if the host has asked for less than we   */
      /*want to send.  Note that we only check the low order  */
      /*byte of the wlength field.  If we ever need to send   */
      /*back descriptors longer than 256 bytes, we'll need to */
      /*revisit this.                                         */
if (desc_sze > usb_buf[6]) desc_sze = usb_buf[6];

      /*send the first data chunk back                        */
for(desc_idx=0; ((desc_idx<8)&&(desc_idx<desc_sze)); desc_idx++) get_desc();
  }
```

**FIGURE 18.** ***getdescriptor*** **Procedure (page 53 of listing)**

As mentioned, descriptors are long, and require multiple packets to transfer them. So first the *getdescriptor* procedure sets the global *status*.GETDESC flag, then loads the global variables *desc_typ* and *desc_sze* with the type and length of the chosen descriptor, respectively. Finally this procedure loads the first packet of data into the transmitter FIFO (the transmitter is enabled within the *rx_0* procedure after this procedure returns).

Loading the FIFO is actually done using the get_desc procedure shown in Figure 19. This is straightforward. Note however that at one location in the device descriptor the board id (*brd_id*), which comes from EEPROM is substituted for the equivalent from ROM. This location corresponds to the device's serial number, and this mechanism allows each Evaluation Board to be uniquely identified.

```
/*********************************************************************/
/* This subroutine loads a byte from a descriptor into endpoint 0    */
/* Fifo.                                                             */
/*********************************************************************/
void get_desc(void)
  {
    byte desc_dta;

       /*select the appropriate descriptor.  compiler limits forced*/
       /*the code to be written in this (admittedly) akward way.   */
if(desc_typ==DEVICE)
  {
    if (desc_idx==12) desc_dta=brd_id;
    else              desc_dta=DEV_DESC[desc_idx];
  }
else if(desc_typ==CONFIGURATION)
  desc_dta=CFG_DESC[desc_idx];

write_usb(TXD0,desc_dta);           /*send data to the FIFO   */
  }
```

**FIGURE 19.** ***get_desc*** **Procedure (page 56 of listing)**

## setfeature and clrfeature

The *setfeature* and *clrfeature* procedures are shown in Figure 20 and Figure 21, respectively. As shown these procedures are not fully implemented, but they do show the basic structure necessary. Please refer to the USB Specification for a full description.

```
/*********************************************************************/
/* The SET_FEATURE request is done here                              */
/*********************************************************************/
void setfeature(void)
  {
switch (usb_buf[0]&0x03)            /*find request target     */
  {
    case 0:                         /*DEVICE                  */
      break;

    case 1:                         /*INTERFACE               */
      break;

    case 2:                         /*ENDPOINT                */
      switch (usb_buf[3])           /*find specific endpoint  */
{
  case 0:
    stalld.0 = 1;
    break;
.
.
.
  case 6:
    stalld.6 = 1;
    break;
  default:
    break;
}
      break;

    default:                        /*UNDEFINED               */
      break;
  }
  }
```

**FIGURE 20. *setfeature* Procedure (page 55 of listing)**

The set_feature command is used to stall an endpoint, or to enable the device to remotely wakeup the host system. The *setfeature* procedure starts by testing to the target selected. This may be either ‘device’, ‘interface’, or ‘endpoint’. Currently there are no interface features defined, and the only device feature defined is the remote wakeup capability (which is not supported by the Evaluation Board). Therefore this procedure ignores both those requests[1]. The only endpoint feature defined is the stall. This procedure determines which specific endpoint is the target and then sets a status flag (*stalld*.x) accordingly, but it does not actually stall the endpoint. For full compliance, set the STALL bit in the appropriate Endpoint Control Register (EPC0 - EPC6) here.

The clr_feature command is used to undo a preceding set_feature. The *clrfeature* procedure ignores the device and interface targets for the same reasons as before. For the endpoint targets, it clears the associated status flag. For full compliance, clear the STALL bit in the appropriate Endpoint Control Register (EPC0 - EPC6) here.

1. For full compliance with the USB Specification, an attempt to set or clear undefined features must result in a stall.

```
/*********************************************************************/
/* The CLEAR_FEATURE request is done here                            */
/*********************************************************************/
void clrfeature(void)
  {
switch (usb_buf[0]&0x03)            /*find request target     */
  {
    case 0:                         /*DEVICE                  */
      break;

    case 1:                         /*INTERFACE               */
      break;

    case 2:                         /*ENDPOINT                */
      switch (usb_buf[3])           /*find specific endpoint  */
{
  case 0:
    stalld.0 = 0;
    break;
.
.
.
  case 6:
    stalld.6 = 0;
    break;
  default:
    break;
}
      break;

    default:                        /*UNDEFINED               */
      break;
  }
  }
```

**FIGURE 21. *clrfeature* Procedure (page 52 of listing)**

### getstatus

The get_status request returns 2 status bytes to the host. Refer to the USB Specification for the exact meaning of the two bytes, which differ for device, interface and endpoint targets. For the latter the least significant bit of the first byte is set if the endpoint is stalled, and cleared if not. The *getstatus* procedure formats this using the EPSTATUS macro.

### setconfiguration and getconfiguration

USB devices are initially unconfigured when first powered or reset. This essentially means that only endpoint 0 is active, and that certain system resources (such as power consumption) are set to special (low) limits. During enumeration, the host looks at the device's requirements (through the descriptors), and determines whether or not the device can be supported, and at what level. Only then does the host configure the device and allow it to become fully active.

```
/*********************************************************************/
/* The GET_STATUS request is done here                               */
/*********************************************************************/
void getstatus(void)
  {
switch (usb_buf[0]&0x03)            /*find request target     */
  {
    case 0:                         /*DEVICE                  */
      write_usb(TXD0,0);            /*first  byte is reserved */
      break;

    case 1:                         /*INTERFACE               */
      write_usb(TXD0,0);            /*first  byte is reserved */
      break;

    case 2:                         /*ENDPOINT                */
      switch (usb_buf[3])           /*find specific endpoint  */
{
  EPSTATUS(0); EPSTATUS(1); EPSTATUS(2); EPSTATUS(3);
  EPSTATUS(4); EPSTATUS(5); EPSTATUS(6);
  default:
    break;
}
      break;

    default:                        /*UNDEFINED               */
      break;
  }

write_usb(TXD0,0);                  /*second byte is reserved */
  }
```

**FIGURE 22. *getstatus* Procedure (page 53 of listing)**

Some devices may support multiple configurations. These may represent a progression of capabilities, each of which offers more capabilities, albeit at increasing system resource costs. The purpose is to allow the device to operate at some level, even if all the resources it needs are not available.

The set_configuration request is used to turn on (or off) the device. The host passes a configuration value to the device, which corresponds to a configuration defined in the descriptors (or 0, which corresponds to the unconfigured state.

The *setconfiguration* procedure shown in Figure 23 processes this request. This firmware supports only one configuration, so this procedure first makes the on or off decision, then acts accordingly. If configuring, this procedure initializes the requisite memory variables, then flushes and enables all other endpoints used. The latter operation means setting the EP_EN bit and the endpoint sub-address in the EPCx register. If unconfiguring, the non-zero endpoints are simply disabled.

The *getconfiguration* code shown in Figure 24 is simpler; so much so that it is incorporated directly into the rx_0 procedure, rather than into a procedure of its own. All it does is return the current configuration value (stored by setconfiguration in the *usb_cfg* variable) to the host.

```
  case GET_CONFIGURATION:
    write_usb(TXD0,usb_cfg);/*load the config value   */
    break;
```

**FIGURE 23.** ***getconfiguration*** **Code Fragment (from *rx_0*, Figure 14)**

```
/*********************************************************************/
/* The SET_CONFIGURATION request is done here                        */
/*********************************************************************/
void setconfiguration(void)
  {
usb_cfg = usb_buf[2];              /*set the configuration # */
if (usb_buf[2]!=0)                 /*set the configuration   */
  {
    dtapid = 0;                    /*FIRST PID is DATA0      */
    stalld = 0;                    /*nothing stalled         */

    FLUSHTX1;                      /*flush  TX1 and disable  */
    write_usb(EPC1,EP_EN+01);      /*enable EP1 at adr 1     */

    FLUSHRX1;                      /*flush  RX1 and disable  */
    write_usb(EPC2,EP_EN+02);      /*enable EP2 at adr 2     */
    write_usb(RXC1,RX_EN);         /*enable RX1              */

    FLUSHTX3;                      /*flush  TX3 and disable  */
    write_usb(EPC5,EP_EN+05);      /*enable EP5 at adr 5     */
    queue_joy();                   /*queue up some data      */

    FLUSHRX3;                      /*flush  RX3 and disable  */
    write_usb(EPC6,EP_EN+06);      /*enable EP6 at adr 6     */
    write_usb(RXC3,RX_EN);         /*enable RX3              */
  }
else                               /*unconfigure the device  */
  {
    write_usb(EPC1,0);             /*disable EP1             */
    write_usb(EPC2,0);             /*disable EP2             */
    write_usb(EPC5,0);             /*disable EP5             */
    write_usb(EPC6,0);             /*disable EP6             */
  }
  }
```

**FIGURE 24.** ***setconfiguration*** **Procedure (page 54 of listing)**

### setaddress

The set_address request is one needed very early in enumeration. Recall that newly initialized devices are set to the default address 0, then one at a time queried and assigned a unique address. The *setaddress* code (pard of *rx_0*) handles this assignment. It does two things. First it takes the address assigned and puts it into the Function Address Register (FAR), enabling the address at the same time by setting the AD_EN bit. It also sets the DEF bit in the EPCO register. This last step is crucial. This temporarily overrides the FAR register contents and allows the USBN9602 to continue responding to the default address until the next ‘in’ packet is received. This is necessary so that the handshake for the set_address command can be transferred properly. Once it is (with the ‘in’ packet), then the DEF bit clears and the new address takes over.

```
 case SET_ADDRESS:
/*set and enable new address for endpoint 0, but set*/
/*DEF too, so new address doesn't take effect until */
/*the handshake completes                            */
  write_usb(EPC0,DEF);
  write_usb(FAR,usb_buf[2] | AD_EN);
  break;
```

**FIGURE 25.** ***setaddress*** **Code Fragment (from *rx_0*, Figure 14)**

## *Complete Listing*

The complete listing follows of the firmware discussed in this white paper (for clarity the header files are not shown). Those segments that have been discussed here are highlighted.

```
/*********************************************************************/
/* This is the source code for simple monitor program used to test    */
/* the 9602 USB device's MICROWIRE port.                              */
/*                                                                    */
/* Written by Jim Lyle, with some code borrowed from Bob Martin       */
/*                                                                    */
/* Copyright (c) 1997, National Semiconductor.  All rights reserved.  */
/*                                                                    */
/* VERSION 0X.50, Aug 28, 1997, Jim Lyle:  Revision control begins.   */
/*                                                                    */
/* VERSION 0X.51, Sep 04, 1997, Jim Lyle:  OHCI fix.                  */
/*                                                                    */
/* VERSION 0X.60, Sep 05, 1997, Jim Lyle:  Additional standard reqs.  */
/*   implemented to pass the chapter 9 compatibility tests.           */
/*                                                                    */
/* VERSION 0X.61, Sep 12, 1997, Jim Lyle:  Changed the USB standard   */
/*   request decoding structure.                                      */
/*                                                                    */
/*                                                                    */
/*********************************************************************/
#include <8788eg.h>
#include "USBN9602.h"
#include "global.h"
#include "lcd_demo.h"

#define FALSE    0
#define TRUE     -1
typedef  unsigned char byte;

/*********************************************************************/
/* Hardware Connections:                                              */
/*                                                                    */
/*   PG1 - Watchdog output.  Tie this LOW to minimize power when      */
/*         these errors occur (watchdog is ignored).                  */
/*   PG2 - 9602 CS* (lo true)                                         */
/*                                                                    */
/*   MICROWIRE- 9602 device                                           */
/*                                                                    */
/*   Uart- The Uart receiver is connected to the RS-232 port.         */
/*                                                                    */
```

```
/*   Tmr1- Used to produce the real-time clock reference (10 Hz).     */
/*                                                                    */
/**********************************************************************/
/* Bit definitions in Port D: */
#define CSA2D    0                          /*Analog to Digital CS*   */
#define LCDRS    2                          /*reg  select             */
#define LCDRW    3                          /*rd (high) or wr (low)   */
#define LCDCS    1                          /*chip select (high true) */
#define HRTBT    4                          /*Heartbeat LED (Low on)  */
#define USB_I    7                          /*USB intr. LED (Low on)  */

/* Bit definitions in Port G: */
#define CSUSB    002                        /*chip select (low  true) */
#define CSUSBM   0b00000100                 /*binary mask             */
#define CSEE     003                        /*chip select (high true) */
#define CSEEM    0b00001000                 /*binary mask             */

/**********************************************************************/
/* System Dependent Values                                            */
/**********************************************************************/
#define VID      'N'                        /*vendor ID    */
#define MAJREV   'X'                        /*major revision    */
#define MINREV   61                         /*minor revision    */
#define BAUDRTE  19200                      /*selected baudrate */

/* The system clock is the 48 MHz USB clock, divided by this value:   */
#define CLKDIV   5                          /*USB clock divisor       */
#define CLKRTE   48/CLKDIV                  /*resulting COP clock     */

#if CLKDIV==5                               /*9.6 MHz                 */
/* For oscillator frequency of 9.6 MHz, there are 48,000 ticks per    */
/* half period, (each period= 1/10 sec)                               */
#define HPDTCKS  48000                      /*half period ticks       */

/* The drive servos are neutral (off) with 1.52 msec pulses. The pulse*/
/* to pulse spacing isn't critical, and is set to 12 msec.            */
#define OFFTCKS  1459                       /*ticks in neutral pulse  */
#define PTPTCKS  11520                      /*ticks between pulses    */
#define MINTCKS  968                        /*minimum servo pulse wid.*/

#if BAUDRTE==19200                          /*19200 baud  */
#define PSRVAL   0x10                       /*Set prescaler to 1.5    */
#define BAUDVAL  0x14                       /*  and divisor to 21     */
#else                                       /*9600  baud  */
#define PSRVAL   0xC0                       /*Set prescaler to 12.5   */
#define BAUDVAL  0x04                       /*  and divisor to 5      */
#endif

#else                                       /*10 MHz                  */
/* For oscillator frequency of 10 MHz, there are 50,000 ticks per     */
/* half period, (each period= 1/10 sec)                               */
#define HPDTCKS  50000                      /*half period ticks       */

/* The drive servos are neutral (off) with 1.52 msec pulses. The pulse*/
/* to pulse spacing isn't critical, and is set to 12 msec.            */
#define OFFTCKS  1520                       /*ticks in neutral pulse  */
#define PTPTCKS  12000                      /*ticks between pulses    */
#define MINTCKS  1008                       /*minimum servo pulse wid.*/
```

```
#if BAUDRTE==19200                          /*19200 baud  */
#define PSRVAL   0x60                       /*Set prescaler to 6.5    */
#define BAUDVAL  0x04                       /*  and divisor to 5      */
#else                                       /*9600  baud  */
#define PSRVAL   0xC8                       /*Set prescaler to 13.0   */
#define BAUDVAL  0x04                       /*  and divisor to 5      */
#endif

#endif

/*********************************************************************/
/* Servo related values                                              */
/*********************************************************************/
#define MINOFFV  122                        /*These values define the */
#define MAXOFFV  134                        /* neutral servo 'window' */
#define IDLEV    128                        /*this is the idle value  */

/*********************************************************************/
/* Message related values                                            */
/*********************************************************************/
/* Command message constants  */
#define MAXMESG  6                          /*byte length of message  */
#define SYNCBYT  0xAA                       /*sync code expected      */

/*********************************************************************/
/* 9602 commands and constants                                       */
/*********************************************************************/
#define USBREAD   0x00                      /*reads data at spec. addr*/
#define USBWRITE  0x80                      /*write selected register */
#define USBBURST  0xC0                      /*burst write */

/*********************************************************************/
/* EEPROM commands and constants                                     */
/*********************************************************************/
#define  EEREAD   0b10000000                /*reads data at spec. addr*/
#define  EEWEN    0b00110000                /*enables all prog. modes */
#define  EEERASE  0b11000000                /*erase selected register */
#define  EEWRITE  0b01000000                /*write selected register */
#define  EEERAL   0b00100000                /*erase all registers     */
#define  EEWRALL  0b00010000                /*write all registers     */
#define  EEWDS    0b00000000                /*disables all prog. modes*/

/*********************************************************************/
/* Other Equates                                                     */
/*********************************************************************/

/*********************************************************************/
/* These are the macros                                              */
/*********************************************************************/
/* macros with arguments*/
#define HIBYT(x) (x)/256
#define LOBYT(x) (x)%256

/* Send data out the MICROWIRE port **********************************/
#define MWOUT(dta)                                                    \
  { MWSR = dta;                         /*put in shft reg   */ \
    PSW.BUSY = 1;                       /*start shifting    */ \
    while (PSW.BUSY == 1) ; }           /*wait until done   */
```

```
/* Turn off all MICROWIRE chip selects ********************************/
#define MWCSOFF                                                        \
  { USBCSOFF;                                                  \
    EECSOFF;                                                   \
    A2DCSOFF; }

/* send address and command out via the MICROWIRE port ***************/
#define MWADRCMD(adr,cmd) MWOUT((adr & 0x3F) | cmd)

/* store the status byte in FIFO0 for the chosen endpoint *************/
#define EPSTATUS(ep)                                                   \
    case ep:                                                   \
      if (stalld.ep) write_usb(TXD0,1); else write_usb(TXD0,0);\
      break;

/* Turn off the 9602 CS signal ****************************************/
#define USBCSOFF PORTGD.CSUSB = 1

/* Turn on the 9602 CS signal *****************************************/
#define USBCSON                                                        \
  { PORTGC.SKSEL = 1;               /*sel alternate SK mode */\
    MWCSOFF                         /*deassert all chip sels*/\
    PORTGD.CSUSB = 0; }             /*re-assert USB CS*     */

/* Turn off the EEPROM CS signal **************************************/
#define EECSOFF  PORTGD.CSEE  = 0

/* Turn off the Analog to Digital CS* signal **************************/
#define A2DCSOFF PORTDD.CSA2D = 1

/* This momentarily re-enables, then disables the interrupts**********/
#define TEST_INTS {PSW.GIE=1; NOP; PSW.GIE=0;}

/* Flush and disable the USB TX0 **************************************/
#define FLUSHTX0 {write_usb(TXC0,FLUSH);}

/* Flush and disable the USB TX1 **************************************/
#define FLUSHTX1 {write_usb(TXC1,FLUSH);}

/* Flush and disable the USB TX3 **************************************/
#define FLUSHTX3 {write_usb(TXC3,FLUSH);}

/* Flush and disable the USB RX0 **************************************/
#define FLUSHRX0 {write_usb(RXC0,FLUSH);}

/* Flush and disable the USB RX1 **************************************/
#define FLUSHRX1 {write_usb(RXC1,FLUSH);}

/* Flush and disable the USB RX3 **************************************/
#define FLUSHRX3 {write_usb(RXC3,FLUSH); rcount3=0;}

/* enable TX0, using the appropriate DATA PID *************************/
#define TXEN0_PID                                                      \
  { if(dtapid.TGL0PID) write_usb(TXC0,TX_TOGL+TX_EN); /*DATA1*/\
    else write_usb(TXC0,TX_EN);                       /*DATA0*/\
    dtapid.TGL0PID=!dtapid.TGL0PID;}

/* enable TX1, using the appropriate DATA PID, but not toggling it ****/
#define TXEN1_PID_NO_TGL                                               \
```

```
  { if(dtapid.TGL1PID) write_usb(TXC1,TX_TOGL+TX_LAST+TX_EN);  \
    else write_usb(TXC1,TX_LAST+TX_EN);              /*DATA0*/\
  }

/* enable TX1, using the appropriate DATA PID ************************/
#define TXEN1_PID                                                    \
  { TXEN1_PID_NO_TGL;                                          \
    dtapid.TGL1PID=!dtapid.TGL1PID;}

/* enable TX3, using the appropriate DATA PID, but not toggling it ****/
#define TXEN3_PID_NO_TGL                                             \
  { if(dtapid.TGL3PID) write_usb(TXC3,TX_TOGL+TX_LAST+TX_EN);  \
    else write_usb(TXC3,TX_LAST+TX_EN);              /*DATA0*/\
  }

/* enable TX3, using the appropriate DATA PID ************************/
#define TXEN3_PID                                                    \
  { TXEN3_PID_NO_TGL;                                          \
    dtapid.TGL3PID=!dtapid.TGL3PID;}

/*********************************************************************/
/* These are the global variables                                    */
/*********************************************************************/
byte usb_buf[8];                    /*buffer used for USB     */
byte desc_typ, desc_idx, desc_sze;

byte usb_cfg;                       /*usb config. setting     */


     /*This area is used to track real (elapsed) time ***************/
byte ticks;                         /*real time ticks (10/second) */

bits pswimg;                        /*the isr saves PSW here  */

byte EEbufh, EEbufl;                /*EEPROM reg buffer       */

#define OUTQLEN  8                          /*length of outq          */
char outq[8];                       /*serial transmit queue   */
byte outq_idx;                      /*outq index              */

byte brd_id;                        /*our unique ID           */
    /*must be a BCD value     */

byte evnt, rxstat, txstat;          /*USB status temp storage */

byte lcdchars;                      /*count of chars written  */

byte tmp1, tmp2;                    /*temporary data: CAREFUL */

byte joy_x, joy_y;                  /*joystick results        */

byte rcount3;                       /*cnt of bytes in RX FIFO3*/

     /*This is the cmd buffer for cmds received via the serial port **/
     /*the order here is critical, and should not change *************/
byte rsnc;                          /*[+0]sync code (AAh exp) */
byte rcmd;                          /*[+1]command op code     */
byte rdta;                          /*[+2]data for command    */
byte radh;                          /*[+3]address (msb)       */
```

```
byte radl;                          /*[+4]        (lsb)       */
byte rcks;                          /*[+5]checksum            */

      /*these are misc. status bits ********************************/
bits status;                        /*misc. status bits       */
#define DEBUG     4                         /*set when host listening */
#define GETDESC   5                         /*set when a get_descr.   */
#define USB_CMD   6                         /*set when doing usb cmd  */
    /*sequence is underway    */
#define XMIT_ON   7                         /*set when RF transmitter */
    /*is sending packets      */

bits dtapid;                        /*PID related status      */
#define TGL0PID   0                         /*tracks NEXT data PID    */
#define TGL1PID   1                         /*tracks NEXT data PID    */
#define TGL2PID   2                         /*tracks NEXT data PID    */
#define TGL3PID   3                         /*tracks NEXT data PID    */

bits stalld;                        /*bits 0-6 correspond to  */
    /*like-numbered endpoints */
    /*and are set to indicate */
    /*the endpoint is stalled.*/


      /*this is the buffer used by the RF transmitter code ************/
byte xmit_buf[16];                  /*RF transmitter buffer   */
byte xmit_sum, xmit_idx;

/*this is the startup banner                                          */
const char msg0[] =  “Waiting For USB  Host Attachment”;
/*              LINE0 00000000000000001111111111111111 LINE1         */

/*for now the sizes and offsets below need to be hand calculated,     */
/*until I can find a better way to do it                              */
/*for multiple byte values, LSB goes first                            */
#define DEV_DESC_SIZE 18
const byte DEV_DESC[] = {DEV_DESC_SIZE,     /*length of this desc.    */
 0x01,              /*DEVICE descriptor       */
 0x00,0x01,         /*spec rev level (BCD)    */
 0x00,              /*device class            */
 0x00,              /*device subclass         */
 0x00,              /*device protocol         */
 0x08,              /*max packet size         */
 0x00,0x04,         /*National’s vendor ID    */
 0x5A,0xC3,         /*National’s product ID   */
 0x12,0x48,         /*National’s revision ID  */
 0,                 /*index of manuf. string  */
 0,                 /*index of prod.  string  */
 0,                 /*index of ser. # string  */
 0x01               /*number of configs.      */
 };

const byte CFG_DESC[] = {0x09,              /*length of this desc.    */
 0x02,              /*CONFIGURATION descriptor*/
 0x2E,0x00,         /*total length returned   */
 0x01,              /*number of interfaces    */
 0x01,              /*number of this config   */
 0x00,              /*index of config. string */
 0x80,              /*attr.: bus powered      */
```

```
 50,               /*max power (100 mA)      */

 0x09,             /*length of this desc.    */
 0x04,             /*INTERFACE descriptor    */
 0x00,             /*interface number        */
 0x00,             /*alternate setting       */
 0x04,             /*# of (non 0) endpoints  */
 0x00,             /*interface class         */
 0x00,             /*interface subclass      */
 0x00,             /*interface protocol      */
 0x00,             /*index of intf. string   */

/*The WDM driver for this board references the endpoints by 'pipe'    */
/*number: 0, 1, 2 et al in order below:                               */

       /*Pipe 0                                        */
 0x07,             /*length of this desc.    */
 0x05,             /*ENDPOINT descriptor     */
 0x81,             /*address (IN)            */
 0x03,             /*attributes  (INTERRUPT) */
 0x20,0x00,        /*max packet size (32)    */
 0xFF,             /*interval (ms)           */

       /*Pipe 1                                        */
 0x07,             /*length of this desc.    */
 0x05,             /*ENDPOINT descriptor     */
 0x02,             /*address (OUT)           */
 0x02,             /*attributes  (BULK)      */
 0x20,0x00,        /*max packet size (32)    */
 0xFF,             /*interval (ms)           */

       /*Pipe 2                                        */
 0x07,             /*length of this desc.    */
 0x05,             /*ENDPOINT descriptor     */
 0x85,             /*address (IN)            */
 0x03,             /*attributes  (INTERRUPT) */
 0x40,0x00,        /*max packet size (64)    */
 0xFF,             /*interval (ms)           */

       /*Pipe 3                                        */
 0x07,             /*length of this desc.    */
 0x05,             /*ENDPOINT descriptor     */
 0x06,             /*address (OUT)           */
 0x02,             /*attributes  (BULK)      */
 0x40,0x00,        /*max packet size (64)    */
 0xFF};            /*interval (ms)           */

#define CFG_DESC_SIZE sizeof(CFG_DESC)

/*********************************************************************/
/* These are the function prototypes                                 */
/*********************************************************************/
void isr(void);
void rcveih(void);
void xmitih(void);
void init_cop(void);
void rf_xmit(void);
void gathr_joy(void);
void queue_joy(void);
```

```
void do_cmd(void);
void xmit(char c);
byte xmit_rtn(char c);
byte mw_cmd(byte dta);
byte mw_adrcmd(byte adr, byte cmd);
byte read_usb(byte adr);
void write_usb(byte adr, byte dta);
void init_usb(void);
void nak0(void);
void nak1(void);
void nak2(void);
void nak3(void);
void clrfeature(void);
void getdescriptor(void);
void getstatus(void);
void setconfiguration(void);
void setfeature(void);
void get_desc(void);
void rx_0(void);
void rx_1(void);
void rx_2(void);
void rx_3(void);
void tx_0(void);
void tx_1(void);
void tx_2(void);
void tx_3(void);
void usb_alt(void);
void usb_isr(void);
void init_lcd(void);
void set_xy(byte lcd_x, byte lcd_y);
void putch(char lcd_char);
void clear_lcd(void);
void write_lcd(char lcd_cmd);
void lcd_delay(void);
void EEcson(void);
byte EEcmd(byte dta);
byte ee_adrcmd(byte adr, byte cmd);
void EEwait(void);
void EEenbl(void);
void EEdsbl(void);
void EEerse(byte adr);
void EEbulk(void);
void EErgrd(byte adr);
void EErgwr(byte adr);
byte A2D_conv(byte chnl);

/*********************************************************************/
/* These are the interrupt vectors                                   */
/*********************************************************************/
#pragma memory ROM [32] @0x1e0;
#asm
.addrw  int_exit                    ;int 00 (VIS)
.addrw  trap1                       ;int 01 (port L edge)
.addrw  trap2                       ;int 02 (T3B)
.addrw  trap3                       ;int 03 (T3A/underflow)
.addrw  trap4                       ;int 04 (T2B)
.addrw  trap5                       ;int 05 (T2A/underflow)
.addrw  utx_ih                      ;int 06 (UART transmit)
.addrw  urx_ih                      ;int 07 (UART receive )
```

```
.addrw  trap8                   ;int 08 (reserved)
.addrw  trap9                   ;int 09 (MICROWIRE)
.addrw  trap10                  ;int 10 (T1B)
.addrw  t1__ih                  ;int 11 (T1A/underflow)
.addrw  trap12                  ;int 12 (T0  underflow)
.addrw  usb_ih                  ;int 13 (external)
.addrw  trap14                  ;int 14 (reserved)
.addrw  trap15                  ;int 15 (INTR)
#endasm

/*********************************************************************/
/* This is the code area                                             */
/*********************************************************************/
#pragma memory ROM [0x00F0] @0x0000;

/*********************************************************************/
/*********************************************************************/
/* This is the main program loop:                                   **/
/*********************************************************************/
/*********************************************************************/
main()
  {

init_cop();                     /*initialize COP proc.    */

init_usb();                     /*initialize 9602         */

init_lcd();                     /*display startup message */

/*      Read parameters from the EEPROM */
brd_id = 0x00;                  /*set default ID          */
EErgrd(0x0);                    /*read EEprom register 0  */
if (EEbufh == 0x0A5)            /*if valid data found     */
  brd_id = EEbufl;              /*  set actual  ID        */

while (TRUE)                    /*then loop forever       */
  {
    if (status.XMIT_ON) rf_xmit();  /*if transmitting...      */
    else
      {
      }

  /*The following operations require the MICROWIRE port, which*/
  /*is also heavily used by various interrupt operations.  To */
  /*prevent conflicts, we must turn off interrupts here temp- */
  /*orarily.  Notice that we re-enable them on occasion here  */
  /*to prevent them from being turned off for too long at any */
  /*given time.                                               */
    PSW.GIE = 0;                /*dsable global interrupt */
      if (rcount3>0)
{
  putch(read_usb(RXD3));    /*xfer RX3 FIFO --> LCD   */
  --rcount3;                /*decriment count         */
}
      else lcdchars = 32;           /*make sure this is synced*/
    PSW.GIE = 1;                /*enable global interrupt */

    NOP;
    NOP;
```

```
    gathr_joy();                    /*get  the joystick data  */
  }

return(0);                          /*make the compiler happy */
  }

/*********************************************************************/
/* This is the main interrupt handler:                               */
/*********************************************************************/
#pragma memory ROM [0x0100] @ 0x00FF;
void isr(void)
  {

#asm
push    A
x       A,B
push    A
x       A,X
push    A
ld      A,PSW                       /*save                    */
x       A,pswimg                    /*  PSW                   */
vis                                 /*vector to int handler   */
#endasm

bufr:   while (TRUE);                       /*if we ever miss the vis */

      /*unexpected interrupt (remember which one) ********************/
trap0:  goto trap0;
trap1:  goto trap1;
trap2:  goto trap2;
trap3:  goto trap3;
trap4:  goto trap4;
trap5:  goto trap5;
trap6:  goto trap6;
trap7:  goto trap7;
trap8:  goto trap8;
trap9:  goto trap9;
trap10: goto trap10;
trap11: goto trap11;
trap12: goto trap12;
trap13: goto trap13;
trap14: goto trap14;
trap15: goto trap15;

      /*serial char. received ***************************************/
urx_ih: rcveih();                           /*service the UART recvr. */
goto int_exit;                      /*and restore context     */

      /*UART transmitter is ready **********************************/
utx_ih: xmitih();                           /*service the UART xmttr. */
goto int_exit;                      /*and restore context     */

      /*tmr1 interrupt (10 Hz) **************************************/
t1__ih: if (++ticks>5)                      /*adjust the tick count   */
  {
    ticks=0;
    PORTDD.HRTBT=!PORTDD.HRTBT;     /*heartbeat (1 Hz flash)  */
  }
```

```
PSW.T1PND = 0;                      /*reset the int pend. bit */
goto int_exit;                      /*and restore context     */

     /*external interrupt ****************************************/
usb_ih: PSW.EXPND = 0;                    /*reset the int pend. bit */
usb_isr();                          /*service the 9602        */

     /*restore the context ***************************************/
int_exit:
PSW.C  = pswimg.C;                  /*restore carry and       */
PSW.HC = pswimg.HC;                 /*  half-carry bits       */

#asm
pop     A
x       A,X
pop     A
x       A,B
pop     A
reti
#endasm

  }

#pragma memory ROM [0x1DFF] @0x0200;
/*********************************************************************/
/* This is the handler for the RS232 port.                           */
/*********************************************************************/
void rcveih(void)
  {
#asm
ld      B,#rcks                     ;point at last element in buffer
x       A,[B-]                      ;get rcks
x       A,[B-]                      ;get radl, rcks --> radl
x       A,[B-]                      ;get radh, radl --> radh
x       A,[B-]                      ;get rdta, radh --> rdta
x       A,[B-]                      ;get rcmd, rdta --> rcmd
x       A,[B-]                      ;get rsnc, rcmd --> rsnc
#endasm

     /*will the following op clear ENUR?? if not DO it              */
if (ENUR.DOE == 1)                  /*if Data Overrun Error   */
  {
    ENUR.DOE = 0;                   /*clear the error         */
  }

rcks = RBUF;                        /*read the rx data reg    */

if (rsnc==SYNCBYT)                  /*if sync code is valid   */
  if (rcks==rsnc+rcmd+rdta+radh+radl) do_cmd();  /*do the cmd */
  }

/*********************************************************************/
/* This is the handler for the RS232 port.                           */
/*********************************************************************/
void xmitih(void)
  {
#asm
ld      B,#outq+7                   ;point at last element in buffer
```

```
x       A,[B-]                      ;get [7]
x       A,[B-]                      ;get [6], [7] --> [6]
x       A,[B-]                      ;get [5], [6] --> [5]
x       A,[B-]                      ;get [4], [5] --> [4]
x       A,[B-]                      ;get [3], [4] --> [3]
x       A,[B-]                      ;get [2], [3] --> [2]
x       A,[B-]                      ;get [1], [2] --> [1]
x       A,[B-]                      ;get [0], [1] --> [0]
x       A,TBUF                      ;load new character to send

ld      A,outq_idx                  ;get index
ifeq    A,#1                        ;will the queue be empty?
rbit    ETI,ENUI                    ;if so shut off ints
dec     A                           ;decriment the index
x       A,outq_idx                  ;and store the new value
#endasm
  }

/*********************************************************************/
/* This is the initialization code:                                  */
/*********************************************************************/
void init_cop(void)
  {
      /*At reset, TRI-STATE ports L,G, and C, and set port D high.    */
      /*however, since we might reinitialize for other reasons, we don't   */
      /*assume anything, and include specific code here (and elsewhere)    */
      /*to guarantee our initial state. */
PORTDD = 0xFF;                      /*turn all LED's off     */
PORTLC = 0;                         /*TRI-STATE and     */
PORTLD = 0;                         /* clear port l     */
PORTGC = 0;                         /*TRI-STATE and     */
PORTGD = 0;                         /* clear port g     */
PORTCC = 0;                         /*TRI-STATE and     */
PORTCD = 0;                         /* clear port c     */

      /*set all unbonded I/O pins to OUTPUTS, to prevent them from floating     */
      /*and oscillating<<<<<<<<<<<<<<<< */

      /*Set up the watchdog timer for maximum period, disabling the clock  */
      /*monitor.  It would be nice to disable to the watchdog, too, because     */
      /*it's cost (in terms of power) outweighs the negligible benefit it  */
      /*provides for this particular application.  Watchdog errors will    */
      /*occur, but they will be ignored.     */
WDSVR = 0xD8;                       /*initialize watchdog    */

      /*Set various I/O bit directions  */
PORTLC.T2B = 1;                     /*Enable T2B output (init state=0) */
PORTLC.T3B = 1;                     /*Enable T3B output (init state=0) */

USBCSOFF;                           /*turn   9602 CS* off    */
PORTGC.CSUSB = 1;                   /*enable 9602 CS* pin    */
PORTGC.CSEE  = 1;                   /*enable EEPROM CS       */

      /*T1 is used to provide 10 interrupts per second.     */
      /*second.     */
CNTRL  = 0x80;                      /*t1 in mode 1, int on underflow   */
TMRLO  = LOBYT(HPDTCKS);            /*initialize   */
TMRHI  = HIBYT(HPDTCKS);            /*  T1 counter */
T1RALO = LOBYT(HPDTCKS);            /*initialize   */
```

```
    T1RAHI = HIBYT(HPDTCKS);            /*  T1RA to half period  */
    T1RBLO = LOBYT(HPDTCKS);            /*initialize   */
    T1RBHI = HIBYT(HPDTCKS);            /*  T1RB to half period  */
    ticks = 0;                          /*clear the real time tick count   */
    CNTRL.T1C0 = 1;                     /*start T1     */

          /*Timer's 2 and 3 provide a PWM output to the drive servos that */
          /*is nominally 1.52 msecs high, and 12 msecs low.  The counters */
          /*are loaded with different initial values to stagger the pulses     */
          /*to the servos (at least initially)   */
    T2CNTRL = 0b10100000;               /*mode 1, toggle, no int */
    T3CNTRL = 0b10100000;               /*mode 1, toggle, no int */

    TMR2LO = LOBYT(PTPTCKS);            /*initialize   */
    TMR2HI = HIBYT(PTPTCKS);            /*  T2 counter */
    T2RALO = LOBYT(OFFTCKS);            /*initialize   */
    T2RAHI = HIBYT(OFFTCKS);            /*  T2RA to neutral pulse     */
    T2RBLO = LOBYT(PTPTCKS);            /*initialize   */
    T2RBHI = HIBYT(PTPTCKS);            /*  T2RB to pulse spacing     */

    TMR3LO = LOBYT(OFFTCKS);            /*initialize   */
    TMR3HI = HIBYT(OFFTCKS);            /*  T3 counter */
    T3RALO = LOBYT(OFFTCKS);            /*initialize   */
    T3RAHI = HIBYT(OFFTCKS);            /*  T3RA to neutral pulse     */
    T3RBLO = LOBYT(PTPTCKS);            /*initialize   */
    T3RBHI = HIBYT(PTPTCKS);            /*  T3RB to pulse spacing     */
    T2CNTRL.T2C0 = 0;                   /*stop T2 */
    T3CNTRL.T3C0 = 0;                   /*stop T3 */

          /*Configure the MICROWIRE port    */
    PORTGC.SO  = 1;                     /*enable SO output  */
    PORTGC.SK  = 1;                     /*enable SK output  */
    PORTGC.SKSEL=1;                     /*selects alternate SK mode*/
    CNTRL.MSEL = 1;                     /*enable MICROWIRE intf  */
    CNTRL.SL1  = 0;                     /*SK period = 2xTC  */
    CNTRL.SL0  = 0;                     /*SK period = 2xTC  */

          /*Init the UART to selected baud rate, 8 data bits, no parity, 1 stop     */
    ENU = 0;                            /*re-apply     */
    ENUR = 0;                           /* critical    */
    ENUI = 0;                           /*   reset state    */

    PSR  = PSRVAL;                      /*set baud rate*/
    BAUD = BAUDVAL;

    PORTLC.TDX = 1;                     /*Enable TDX output */
    ENUI.EDTX = 1;                      /*                  */

    PORTLC.RDX = 0;                     /*Enable RDX input, making sure that    */
    PORTLD.RDX = 0;                     /*  it is not pulled up  */
    PORTLD.RDX = 1;                     /*<<<<<<<pull it up temporarily<<<<<    */

    rcks = RBUF;                        /*clear receive buffer   */

          /*Init the A/D unit     */

          /*Misc. initialization  */
    rcks    = 0;                        /*clear receive buffer    */
    status  = 0;                        /*clear status  bits      */
```

```
rcount3 = 0;                        /*clear RX FIFO3 count    */
xmit_idx= 0;                        /*clear RF xmitter index  */
outq_idx= 0;                        /*clear the outq index    */

      /*Configure and enable the interrupts                          */
CNTRL.IEDG = 0;                     /*external ints use rising edge*/
PSW.EXEN = 1;                       /*enable the external interrupt    */
ENUI.ERI = 1;                       /*enable serial recvr interrupt    */
PSW.T1ENA= 1;                       /*enable T1 underflow interrupt    */
PSW.GIE = 1;                        /*enable global interrupt     */
  }


/*********************************************************************/
/* This subroutine transmits an RF packet.                           */
/*********************************************************************/
void rf_xmit(void)
  {
xmit(0x55);                     /*preamble                  */
xmit_sum =xmit_rtn(SYNCBYT);    /*transmit RF packet        */

xmit_sum+=xmit_rtn(xmit_buf[xmit_idx+0]);
xmit_sum+=xmit_rtn(xmit_buf[xmit_idx+1]);
xmit_sum+=xmit_rtn(xmit_buf[xmit_idx+2]);
xmit_sum+=xmit_rtn(xmit_buf[xmit_idx+3]);

xmit(xmit_sum);                 /*checksum                  */

xmit_idx = 0x0C & (xmit_idx+4); /*wrap and adjust as nec.     */
  }

/*********************************************************************/
/* This subroutine collects the joystick data and stuffs it into glbl */
/* variables set aside for the purpose.                              */
/*********************************************************************/
void gathr_joy(void)
  {
PSW.GIE = 0;                    /*dsable global interrupt     */
joy_x = A2D_conv(0);            /*convert X channel           */
TEST_INTS;                      /*check for interrupts        */
joy_y = A2D_conv(1);            /*convert Y channel           */
PSW.GIE = 1;                    /*enable global interrupt     */
  }

/*********************************************************************/
/* This subroutine queues up the joystick data into endpoint 5, which */
/* has been dedicated for this purpose.                              */
/*********************************************************************/
void queue_joy(void)
  {
write_usb(TXD3,joy_x);          /*send data to the FIFO       */
write_usb(TXD3,joy_y);
write_usb(TXD3,PORTID);
write_usb(TXD3,0x0FF);

TXEN3_PID_NO_TGL;               /*enable TX, choose PID       */
  }

/*********************************************************************/
```

```
/* This subroutine parses commands received through the serial port:  */
/*********************************************************************/
void do_cmd(void)
  {
byte *radr;                          /*pointer for memory ops  */
radr=radl;                           /*setup pointer           */

switch (rcmd)
  {
    case ‘b’:                        /*look for EE bulk erase  */
      EEbulk();                      /*bulk erase the EEPROM   */
      xmit(‘b’);                     /*send back dummy code    */
      break;

    case ‘d’:                        /*look for ‘debug’ toggle */
      status.DEBUG=!status.DEBUG;    /*flip the bit            */
      if (status.DEBUG)              /*send back conf. code    */
xmit(0xFF);
      else
xmit(0x00);
      break;

    case ‘e’:                        /*look for EE reg  erase  */
      EEerse(radl);                  /*erase the EEPROM reg    */
      xmit(‘e’);                     /*send back dummy code    */
      break;

    case ‘g’:                        /*look for 9602 get (read)*/
      xmit(read_usb(radl));          /*read  the spec. 9602 reg*/
      break;

    case ‘p’:                        /*look for 9602 prog (wr) */
      write_usb(radl,rdta);          /*write the spec. 9602 reg*/
      xmit(rdta);                    /*send it back to confirm */
      break;

    case ‘i’:                        /*look for init USB cmd   */
      init_usb();                    /*initialize the 9602     */
      xmit(‘i’);                     /*send back dummy code    */
      break;

    case ‘l’:                        /*look for EE reg  load   */
      EErgrd(radl);                  /*load  the EEPROM re     */
      xmit(EEbufl);                  /*send back low byte      */
      break;

    case ‘q’:                        /*look for Query op       */
      xmit(VID);                     /*send Vendor ID          */
      xmit(MAJREV);                  /*send Major Revision     */
      xmit(MINREV);                  /*send Minor Revision     */
      xmit(&usb_buf);                /*send usb buffer offset  */
      xmit(&EEbufh);                 /*send EE  buffer offset  */
      xmit(brd_id);                  /*send ID                 */
      break;

    case ‘c’:                        /*look for LCD command    */
      if (rdta==0xFF)                /*USB bulk data transfer  */
{                          /*pull 16 bytes from FIFO */
  for(rdta=0; rdta<=15; rdta++) putch(read_usb(RXD1));
```

```
}
      else
if (rdta) putch(rdta);      /*if ASCII, wr to display */
else clear_lcd();           /*else clear the display  */
      xmit('c');                    /*send back dummy code    */
      break;

    case 'z':                       /*look for RF Transmit cmd*/
      if (rdta)                     /*enable transmitter      */
{                           /*pull 16 bytes from FIFO */
  for(rdta=0; rdta<=15; rdta++) xmit_buf[rdta]=read_usb(RXD1);
  xmit_idx &=0x0C;          /*adj idx to pkt boundary */
  status.XMIT_ON=1;         /*set flag                */
}
      else                          /*disable transmitter     */
{
  status.XMIT_ON=0;         /*clr flag                */
}
      xmit('z');                    /*send back dummy code    */
      break;

    case 'j':                       /*look for Joystick op    */
      xmit(joy_x);                  /*send X channel          */
      xmit(joy_y);                  /*send Y channel          */
      xmit(PORTID);                 /*send switch closures    */
      break;

    case 's':                       /*look for EE reg  store  */
      EEbufl = rdta;                /*store low byte in buf   */
      EErgwr(radl);                 /*store the EEPROM reg    */
      xmit('s');                    /*send back dummy code    */
      break;

    case 't':                       /*look for toggle (CS*)   */
      PORTGD = PORTGD ^ CSUSBM;     /*toggle the CS* signal   */
      xmit(PORTGD & CSUSBM);        /*isol the bit and send it*/
      break;

    case 'r':                       /*look for Rd code        */
      xmit(*radr);                  /*get data and send it    */
      break;

    case 'w':                       /*look for Wr code        */
    //xmit(*radr=rdta);             /*doesn't compile right!!!*/
      *radr=rdta;                   /*write the data          */
      xmit(*radr);                  /*get data and send it    */
      break;

    case 'x':                       /*look for MICROWIRE exch.*/
      xmit(mw_cmd(rdta));           /*get data and exchange it*/
      break;

    default:
      xmit('?');                    /*we're confused... help!!*/
  }                                 /*end of switch           */
  }

/*********************************************************************/
/* This subroutine sends a character through the UART transmitter or  */
```

```
/* The usb interface, depending on the mode.                          */
/*********************************************************************/
void xmit(char c)
  {

     /*USB mode: move character to endpoint 1 FIFO ******************/
if (status.USB_CMD)
  {
    write_usb(TXD1,c);              /*send data to the FIFO   */
  }

     /*UART mode: move character to UART queue **********************/
else
  {
    while (outq_idx>=OUTQLEN)   /*wait for room in queue      */
      if (ENU.TBMT) xmitih();   /*or make it if we can        */
    outq[outq_idx++]=c;         /*post character              */
    ENUI.ETI=1;                 /*turn on interrupt           */
  }
  }

/*********************************************************************/
/* This subroutine is just like xmit(), except that it returns the    */
/* data passed to it                                                  */
/*********************************************************************/
byte xmit_rtn(char c)
  {
xmit(c);
return(c);
  }

/*********************************************************************/
/* This subroutine sends a command through the MICROWIRE port, then   */
/* returns the result.                                                */
/*********************************************************************/
byte mw_cmd(byte dta)
  {
MWOUT(dta);                         /*send the char     */
return(MWSR);                       /*return the result */
  }

/*********************************************************************/
/* This subroutine sends the address and command through the MICROWIRE*/
/* port.                                                              */
/*********************************************************************/
byte mw_adrcmd(byte adr, byte cmd)
  {
MWADRCMD(adr,USBREAD);              /*send cmd and addr       */
return(MWSR);                       /*return the result       */
  }

/*********************************************************************/
/* This subroutine reads the USB register whose address is given.     */
/*********************************************************************/
byte read_usb(byte adr)
  {
USBCSON;                            /*turn on CS*/
MWADRCMD(adr,USBREAD);              /*send cmd and addr*/
MWOUT(0);                           /*send dummy data  */
```

```
USBCSOFF;                          /*turn off CS*/
return(MWSR);                      /*return the result*/
  }

/*********************************************************************/
/* This subroutine writes the USB register whose address is given.   */
/*********************************************************************/
void write_usb(byte adr, byte dta)
  {
USBCSON;                           /*turn on CS*/
MWADRCMD(adr,USBWRITE);            /*send cmd and addr*/
MWOUT(dta);                        /*send the data*/
USBCSOFF;                          /*turn off CS*/
  }

/*********************************************************************/
/* This subroutine initializes the 9602.                             */
/*********************************************************************/
void init_usb(void)
  {
     /*toss out any previous state ********************************/
status.GETDESC=0;
usb_cfg = 0;

     /*give a software reset, then set ints to active high push pull */
write_usb(MCNTRL,SRST); write_usb(MCNTRL,INT_H_P);

     /*initialize the clock generator *****************************/
     /* prior to this point, the clock output will be 4 Mhz.  After, */
     /* it will be (48 MHz/CLKDIV)                                   */
write_usb(CCONF,CLKDIV-1);

     /*set default address, enable EP0 only ***********************/
write_usb(FAR,AD_EN+0); write_usb(EPC0, 0x00);

     /*set up interrupt masks *************************************/
write_usb(NAKMSK,NAK_O0);                        /*NAK evnts*/
write_usb(TXMSK,TXFIFO0+TXFIFO1+TXFIFO3);        /*TX events*/
write_usb(RXMSK,RXFIFO0+RXFIFO1+RXFIFO3);        /*RX events*/
write_usb(ALTMSK,SD3+RESET_A);                   /*ALT evnts*/
    /*this is modified in the */
    /*suspend-resume routines */
    /*so if any change is made*/
    /*here it needs to be ref-*/
    /*lected there too.       */
write_usb(MAMSK,(INTR_E+RX_EV+NAK+TX_EV+ALT));

     /*enable the receiver and go operational *********************/
FLUSHTX0;                          /*flush TX0 and disable   */
write_usb(RXC0,RX_EN);             /*enable the receiver     */

write_usb(NFSR,OPR_ST);            /*go operational          */
write_usb(MCNTRL,INT_H_P+NAT);     /*set NODE ATTACH         */
  }

/*********************************************************************/
/* 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     */
    write_usb(NFSR,OPR_ST);         /*go operational          */
  }

else if(evnt & SD3)                 /*suspend event           */
  {
    write_usb(ALTMSK,RESUME_A+RESET_A);  /*adjust interrupts  */
    write_usb(NFSR,SUS_ST);         /*enter suspend state     */
  }

else if(evnt & RESUME_A)            /*resume  event           */
  {
    write_usb(ALTMSK,SD3+RESET_A);       /*adjust interrupts  */
    write_usb(NFSR,OPR_ST);         /*go operational          */
  }

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

/*********************************************************************/
/* The CLEAR_FEATURE request is done here                            */
/*********************************************************************/
void clrfeature(void)
  {
switch (usb_buf[0]&0x03)            /*find request target     */
  {
    case 0:                         /*DEVICE                  */
      break;

    case 1:                         /*INTERFACE               */
      break;

    case 2:                         /*ENDPOINT                */
      switch (usb_buf[3])           /*find specific endpoint  */
{
  case 0:
    stalld.0 = 0;
    break;
  case 1:
    stalld.1 = 0;
    break;
  case 2:
    stalld.2 = 0;
    break;
  case 3:
    stalld.3 = 0;
    break;
  case 4:
```

```
      stalld.4 = 0;
      break;
    case 5:
      stalld.5 = 0;
      break;
    case 6:
      stalld.6 = 0;
      break;
    default:
      break;
}
      break;

    default:                        /*UNDEFINED              */
      break;
  }
  }

/*****************************************************************/
/* The GET_DESCRIPTOR request is done here                       */
/*****************************************************************/
void getdescriptor(void)
  {
status.GETDESC=1;           /*enter get_descr mode    */
desc_typ = usb_buf[3];      /*store the type requested*/
if(desc_typ==DEVICE) desc_sze = DEV_DESC_SIZE;

else if(desc_typ==CONFIGURATION) desc_sze = CFG_DESC_SIZE;

      /*adjust size, if the host has asked for less than we   */
      /*want to send.  Note that we only check the low order  */
      /*byte of the wlength field.  If we ever need to send   */
      /*back descriptors longer than 256 bytes, we'll need to */
      /*revisit this.                                         */
if (desc_sze > usb_buf[6]) desc_sze = usb_buf[6];

      /*send the first data chunk back                        */
for(desc_idx=0; ((desc_idx<8)&&(desc_idx<desc_sze)); desc_idx++) get_desc();
  }

/*****************************************************************/
/* The GET_STATUS request is done here                           */
/*****************************************************************/
void getstatus(void)
  {
switch (usb_buf[0]&0x03)            /*find request target     */
  {
    case 0:                         /*DEVICE                  */
      write_usb(TXD0,0);            /*first  byte is reserved */
      break;

    case 1:                         /*INTERFACE               */
      write_usb(TXD0,0);            /*first  byte is reserved */
      break;

    case 2:                         /*ENDPOINT                */
      switch (usb_buf[3])           /*find specific endpoint  */
{
  EPSTATUS(0); EPSTATUS(1); EPSTATUS(2); EPSTATUS(3);
```

```
    EPSTATUS(4); EPSTATUS(5); EPSTATUS(6);
    default:
      break;
  }
      break;

    default:                        /*UNDEFINED              */
      break;
  }

write_usb(TXD0,0);                  /*second byte is reserved */
  }

/*********************************************************************/
/* The SET_CONFIGURATION request is done here                        */
/*********************************************************************/
void setconfiguration(void)
  {
usb_cfg = usb_buf[2];               /*set the configuration # */
if (usb_buf[2]!=0)                  /*set the configuration   */
  {
    dtapid = 0;                     /*FIRST PID is DATA0      */
    stalld = 0;                     /*nothing stalled         */

    FLUSHTX1;                       /*flush  TX1 and disable  */
    write_usb(EPC1,EP_EN+01);       /*enable EP1 at adr 1     */

    FLUSHRX1;                       /*flush  RX1 and disable  */
    write_usb(EPC2,EP_EN+02);       /*enable EP2 at adr 2     */
    write_usb(RXC1,RX_EN);          /*enable RX1              */

    FLUSHTX3;                       /*flush  TX3 and disable  */
    write_usb(EPC5,EP_EN+05);       /*enable EP5 at adr 5     */
    queue_joy();                    /*queue up some data      */

    FLUSHRX3;                       /*flush  RX3 and disable  */
    write_usb(EPC6,EP_EN+06);       /*enable EP6 at adr 6     */
    write_usb(RXC3,RX_EN);          /*enable RX3              */
  }
else                                /*unconfigure the device  */
  {
    write_usb(EPC1,0);              /*disable EP1             */
    write_usb(EPC2,0);              /*disable EP2             */
    write_usb(EPC5,0);              /*disable EP5             */
    write_usb(EPC6,0);              /*disable EP6             */
  }
  }

/*********************************************************************/
/* The SET_FEATURE request is done here                              */
/*********************************************************************/
void setfeature(void)
  {
switch (usb_buf[0]&0x03)            /*find request target     */
  {
    case 0:                         /*DEVICE                  */
      break;

    case 1:                         /*INTERFACE               */
```

```
        break;

      case 2:                         /*ENDPOINT                */
        switch (usb_buf[3])           /*find specific endpoint  */
{
  case 0:
    stalld.0 = 1;
    break;
  case 1:
    stalld.1 = 1;
    break;
  case 2:
    stalld.2 = 1;
    break;
  case 3:
    stalld.3 = 1;
    break;
  case 4:
    stalld.4 = 1;
    break;
  case 5:
    stalld.5 = 1;
    break;
  case 6:
    stalld.6 = 1;
    break;
  default:
    break;
}
        break;

      default:                        /*UNDEFINED               */
        break;
  }
  }

/*********************************************************************/
/* This subroutine loads a byte from a descriptor into endpoint 0     */
/* Fifo.                                                              */
/*********************************************************************/
void get_desc(void)
  {
    byte desc_dta;

       /*select the appropriate descriptor.  compiler limits forced*/
       /*the code to be written in this (admittedly) akward way.   */
if(desc_typ==DEVICE)
  {
    if (desc_idx==12) desc_dta=brd_id;
    else              desc_dta=DEV_DESC[desc_idx];
  }
else if(desc_typ==CONFIGURATION)
  desc_dta=CFG_DESC[desc_idx];

write_usb(TXD0,desc_dta);           /*send data to the FIFO   */
  }

/*********************************************************************/
/* This subroutine handles RX events for FIFO0 (endpoint 0)           */
```

```
/*****************************************************************/
void rx_0(void)
  {
rxstat=read_usb(RXS0);              /*get receiver status     */

      /*is this a setup packet? ************************************/
if(rxstat & SETUP_R)
  {
  /*read data payload into buffer then flush/disble the RX ****/
    for(desc_idx=0; desc_idx<8; desc_idx++)
      usb_buf[desc_idx] = read_usb(RXD0);

      FLUSHRX0;                     /*make sure the RX is off */
      FLUSHTX0;                     /*make sure the TX is off */

    if ((usb_buf[0]&0x60)==0x00)    /*if a standard request   */
      switch (usb_buf[1])           /*find request target     */
{
  case CLEAR_FEATURE:
    clrfeature();
    break;

  case GET_CONFIGURATION:
    write_usb(TXD0,usb_cfg);/*load the config value   */
    break;

  case GET_DESCRIPTOR:
    getdescriptor();
    break;

  case GET_STATUS:
    getstatus();
    break;

  case SET_ADDRESS:
  /*set and enable new address for endpoint 0, but set*/
  /*DEF too, so new address doesn't take effect until */
  /*the handshake completes                           */
    write_usb(EPC0,DEF);
    write_usb(FAR,usb_buf[2] | AD_EN);
    break;

  case SET_CONFIGURATION:
    setconfiguration();
    break;

  case SET_FEATURE:
    setfeature();
    break;

  default:                  /*unsupported standard req*/
    break;
}
    else                            /*if a non-standard req.  */
      {
      }

  /*the following is done for all setup packets.  Note that if*/
  /*no data was stuffed into the FIFO, the result of the fol- */
```

```
    /*lowing will be a zero-length response.                    */
        write_usb(TXC0,TX_TOGL+TX_EN);  /*enable the TX (DATA1) */
        dtapid.TGL0PID=0;               /*store NEXT PID state  */
    }

        /*if not a setup packet, it must be an OUT packet **************/
  else
    {
      if (status.GETDESC)             /*get_descr status stage? */
        {
        /*test for errors (zero length, correct PID)            */
  if ((rxstat& 0x5F)!=0x10)   /*length error??          */
    {
    }

  status.GETDESC=0;           /*exit get_descr mode     */
  FLUSHTX0;                   /*flush TX0 and disable   */
        }

      write_usb(RXC0,RX_EN);          /*re-enable the receiver  */
    }

        /*we do this stuff for all rx_0 events *************************/
    }

  /*********************************************************************/
  /* This subroutine handles RX events for FIFO1 (endpoint 2)          */
  /*********************************************************************/
  void rx_1(void)
    {
  rxstat=read_usb(RXS1);              /*get receiver status     */

  if(rxstat & SETUP_R)
    {
    }
  else if (rxstat & RX_ERR)
    {
    }
  else
    {


    /*check to see if this is a valid command packet????<<<<<<<<<<*/
    /*we don't have to worry about the UART receiver stomping on  */
    /*us because it has a lower priority interrupt.  The USB might*/
    /*well stomp the UART's data, but the assumption is that any  */
    /*debug activity on the RS-232 port is stompable              */
      status.USB_CMD=1;               /*select USB command mode */
      rsnc=read_usb(RXD1);            /*move data from FIFO     */
      rcmd=read_usb(RXD1);
      rdta=read_usb(RXD1);
      radh=read_usb(RXD1);
      radl=read_usb(RXD1);
      rcks=read_usb(RXD1);


  brkpt:            if (rsnc==SYNCBYT)              /*if sync code is valid   */
        if (rcks==rsnc+rcmd+rdta+radh+radl)
  {
```

```
  FLUSHTX1;                 /*flush TX1 and disable   */
  do_cmd();                 /*do the command          */
  TXEN1_PID_NO_TGL;         /*enable TX, choose PID   */
}
    status.USB_CMD=0;               /*exit USB command mode   */
  }

FLUSHRX1;                           /*flush RX1 and disable   */
write_usb(RXC1,RX_EN);              /*re-enable the receiver  */
  }

/*********************************************************************/
/* This subroutine handles RX events for FIFO2 (endpoint 4)          */
/*********************************************************************/
void rx_2(void)
  {
rxstat=read_usb(RXS2);              /*get receiver status     */
  }

/*********************************************************************/
/* This subroutine handles RX events for FIFO3 (endpoint 6)          */
/*********************************************************************/
void rx_3(void)
  {
rxstat=read_usb(RXS3);              /*get receiver status     */

if(rxstat & SETUP_R)
  {
  }
else if (rxstat & RX_ERR)
  {
    FLUSHRX3;                       /*flush RX3 and disable   */
  }
else
  {
    rcount3+=32;                    /*update count            */
  }

write_usb(RXC3,RX_EN);              /*re-enable the receiver  */

  }

/*********************************************************************/
/* This subroutine handles TX events for FIFO0 (endpoint 0)          */
/*********************************************************************/
void tx_0(void)
  {
    byte lim;

txstat=read_usb(TXS0);              /*get transmitter status  */

      /*if a transmission has completed successfully, check to see if */
      /*we have anything else that needs to go out, otherwise turn the*/
      /*receiver back on ***********************************************/
if ((txstat & ACK_STAT) && (txstat & TX_DONE))
  {
    FLUSHTX0;                       /*flush TX0 and disable   */

  /*the desc. is sent in pieces; queue another piece if nec.  */
```

```
    if(status.GETDESC)
      {
lim=desc_idx+8;             /*set new max limit       */

      /*move the data into the FIFO   */
for(; ((desc_idx<lim)&&(desc_idx<desc_sze)); desc_idx++)
  get_desc();
TXEN0_PID;                  /*enable TX, choose PID   */
      }
    else
      {
write_usb(RXC0,RX_EN);      /*re-enable the receiver  */
      }
  }

      /*otherwise something must have gone wrong with the previous ****/
      /*transmission, or we got here somehow we shouldn't have ********/
       else
 {
 }

      /*we do this stuff for all tx_0 events ************************/
  }

/*********************************************************************/
/* This subroutine handles TX events for FIFO1 (endpoint 1)          */
/*********************************************************************/
void tx_1(void)
  {
txstat=read_usb(TXS1);              /*get transmitter status  */

      /*if a transmission has completed successfully, update the data */
      /*toggle and queue up a dummy packet ***************************/
if ((txstat & ACK_STAT) && (txstat & TX_DONE))
  {
    dtapid.TGL1PID=!dtapid.TGL1PID; /*flip the data toggle    */

    FLUSHTX1;                       /*flush the FIFO          */

    write_usb(TXD1,0x0DE);          /*send data to the FIFO   */
    write_usb(TXD1,0x0AD);          /*send data to the FIFO   */
    write_usb(TXD1,0x0BE);          /*send data to the FIFO   */
    write_usb(TXD1,0x0EF);          /*send data to the FIFO   */

    TXEN1_PID_NO_TGL;               /*enable TX, choose PID   */
  }
else
  {
  }

  }

/*********************************************************************/
/* This subroutine handles TX events for FIFO2 (endpoint 3)          */
/*********************************************************************/
void tx_2(void)
  {
txstat=read_usb(TXS2);              /*get transmitter status  */
  }
```

```
/*********************************************************************/
/* This subroutine handles TX events for FIFO3 (endpoint 5)          */
/*********************************************************************/
void tx_3(void)
  {
txstat=read_usb(TXS3);              /*get transmitter status  */

      /*if a transmission has completed successfully, update the data */
      /*toggle ******************************************************/
if ((txstat & ACK_STAT) && (txstat & TX_DONE))
  {
    dtapid.TGL3PID=!dtapid.TGL3PID; /*flip the data toggle    */
    queue_joy();                    /*queue up a new packet   */
  }
else
  {
  }

  }

/*********************************************************************/
/* This subroutine handles OUT NAK events for FIFO0 (endpoint 0)     */
/*********************************************************************/
void nak0(void)
  {
      /*important note:  even after servicing a NAK, another NAK      */
      /*interrupt may occur if another 'OUT' or 'IN' packet comes in  */
      /*during our NAK service.                                       */

      /*if we're currently doing something that requires multiple 'IN'*/
      /*transactions, 'OUT' requests will get NAKs because the FIFO is*/
      /*busy with the TX data.  Since the 'OUT' here means a premature*/
      /*end to the previous transfer, just flush the FIFO, disable the*/
      /*transmitter, and re-enable the receiver.                      */
      if (status.GETDESC)                  /*get_descr status stage? */
{
  status.GETDESC=0;                /*exit get_descr mode     */
  FLUSHTX0;                        /*flush TX0 and disable   */
  write_usb(RXC0,RX_EN);           /*re-enable the receiver  */
}

      /*we do this stuff for all nak0 events ************************/
  }

/*********************************************************************/
/* This subroutine handles OUT NAK events for FIFO1 (endpoint 2)     */
/*********************************************************************/
void nak1(void)
  {
  }

/*********************************************************************/
/* This subroutine handles OUT NAK events for FIFO2 (endpoint 4)     */
/*********************************************************************/
void nak2(void)
  {
  }
```

```
/*********************************************************************/
/* This subroutine handles OUT NAK events for FIFO3 (endpoint 6)     */
/*********************************************************************/
void nak3(void)
  {
  }

/*********************************************************************/
/* This is the interrupt service routine for USB operations          */
/*********************************************************************/
void usb_isr(void)
  {
evnt = read_usb(MAEV);              /*check the events       */

PORTDD.USB_I = 0;                   /*turn on  intr. LED     */

if (evnt & NAK)
  {
    evnt=read_usb(NAKEV);           /*check the NAK events   */
    if      (evnt&NAK_O0) nak0();   /*endpoint 0             */
    else if (evnt&NAK_O1) nak1();   /*endpoint 2             */
    else if (evnt&NAK_O2) nak2();   /*endpoint 4             */
    else if (evnt&NAK_O3) nak3();   /*endpoint 6             */
    else                            /*some other TX event    */
      {
      }
  }

else if (evnt & RX_EV)
  {
    evnt=read_usb(RXEV);            /*check the RX events    */

    if      (evnt&RXFIFO0) rx_0();  /*endpoint 0             */
    else if (evnt&RXFIFO1) rx_1();  /*endpoint 2             */
    else if (evnt&RXFIFO2) rx_2();  /*endpoint 4             */
    else if (evnt&RXFIFO3) rx_3();  /*endpoint 6             */
    else                            /*some other RX event    */
      {
      }
  }

else if (evnt & TX_EV)
  {
    evnt=read_usb(TXEV);            /*check the TX events    */
    if      (evnt&TXFIFO0) tx_0();  /*endpoint 0             */
    else if (evnt&TXFIFO1) tx_1();  /*endpoint 1             */
    else if (evnt&TXFIFO2) tx_2();  /*endpoint 3             */
    else if (evnt&TXFIFO3) tx_3();  /*endpoint 5             */
    else                            /*some other TX event    */
      {
      }
  }


else if (evnt & ALT) usb_alt();     /*alternate event?       */

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

```
      /*the 9602 produces interrupt LEVELS, the COP looks for edges.  */
      /*So we have to fool the 9602 into producing new edges for us   */
      /*when we are ready to look for them.  We do this by temporarily*/
      /*disabling the interrupts, then re-enabling them.              */
evnt=read_usb(MAMSK);               /*save old mask contents  */
write_usb(MAMSK,(0));               /*disable interrupts      */
write_usb(MAMSK,evnt);              /*re-enable interrupts    */

PORTDD.USB_I = 1;                   /*turn off intr. LED      */
  }


/*********************************************************************/
/* This initializes the LCD and displays the startup message         */
/*********************************************************************/
void init_lcd(void)
  {
PORTDD.LCDCS = 0;                   /*de-assert the lcd select  */
PORTCC = 0xFF;                      /*lcd data port is an output*/
PORTCD = 0x00;                      /*initialize lcd data       */

lcd_delay();
write_lcd(LCD_INTF + BYTE_W + TWO_LINES);     /*set mode        */

// lcd_cmd = LCD_SET + LCD_ON + LCD_CURSOR + LCD_BLINK;
write_lcd(LCD_SET + LCD_ON);

write_lcd(LCD_EMODE + LCD_INC);

clear_lcd();                        /*clear and home the lcd  */

for(tmp2=0; tmp2<=31; tmp2++) putch(msg0[tmp2]);
  }

/*********************************************************************/
/* This clears the lcd and homes the cursor                          */
/*********************************************************************/
void clear_lcd(void)
  {
write_lcd(LCD_CLR);                 /*clear the LCD           */
lcd_delay();
write_lcd(LCD_HME);                 /*send the cursor 'home'  */
lcd_delay();

lcdchars = 0;                       /*clear lcd char count    */
  }

/*********************************************************************/
/* This sets the x and y coordinates for the LCD display             */
/*********************************************************************/
void set_xy(byte lcd_x, byte lcd_y)
  {
if(lcd_y == 1) lcd_x += 0x40;
write_lcd(lcd_x+DD_ADDR);
  }

/*********************************************************************/
/* This writes a (display) character to the LCD display              */
```

```
/*********************************************************************/
void putch(char lcd_char)
  {

    if(lcdchars==16)
      {
set_xy(0,1);                        /*go to next line if nec. */
      }
    else if (lcdchars==32)                  /*or go to first line?    */
      {
set_xy(0,0);                        /*send the cursor 'home'  */
lcdchars = 0;                       /*clear lcd char count    */
      }

    ++lcdchars;                             /*update char count       */

    PORTDD.LCDRS = 1;
    PORTDD.LCDRW = 0;
    PORTCD = lcd_char;
    PORTDD.LCDCS = 1;
    NOP();
    NOP();
    NOP();
    PORTDD.LCDCS = 0;
    PORTDD.LCDRW = 1;




   }

/*********************************************************************/
/* This gives the LCD display the time it needs for various functions */
/*********************************************************************/
void lcd_delay(void)
  {
tmp1 = 24; while(tmp1--) NOP();
   }

/*********************************************************************/
/* This writes a command to the LCD display                          */
/*********************************************************************/
void write_lcd(char lcd_cmd)
  {
    PORTDD.LCDRS = 0;
    PORTDD.LCDRW = 0;
    PORTCD = lcd_cmd;
    PORTDD.LCDCS = 1;
    NOP();
    NOP();
    NOP();
    PORTDD.LCDCS = 0;
    PORTDD.LCDRW = 1;
    lcd_delay();
   }

/*********************************************************************/
/* This subroutine turns on the EEPROM CS* signal:                   */
/*********************************************************************/
```

```
void EEcson(void)
   {
PORTGC.SKSEL = 0;                  /*selects normal SK mode  */
MWCSOFF;                           /*deassert all chip sels  */
PORTGD.CSEE  = 1;                  /*assert EE CS            */
   }

/*********************************************************************/
/* This subroutine sends a command to the EEPROM through the Micro-  */
/* wire port, then returns the result.                               */
/*********************************************************************/
byte EEcmd(byte dta)
  {
MWSR = 0x0FF;                      /*put in all ones         */
PSW.BUSY = 1;                      /*shift out one bit       */
PSW.BUSY = 0;                      /*  (sk clock=2 TC cycles)*/
return(mw_cmd(dta));               /*send dta, return rslt   */
  }

/*********************************************************************/
/* This subroutine sends the address and command through the MICROWIRE*/
/* port, including the requisite start bit.                          */
/*********************************************************************/
byte ee_adrcmd(byte adr, byte cmd)
  {
return(EEcmd((adr & 0x3F) | cmd));  /*form and send the cmd   */
  }

/*********************************************************************/
/* This subroutine checks the status of the EEPROM on the MICROWIRE   */
/* port, and loops until it is not busy.  The DO line is sampled      */
/* because it follows BUSY when CS is asserted after the Tcs interval */
/*********************************************************************/
void EEwait(void)
  {
EECSOFF;                           /*turn off CS*/
EEcson();                          /*turn on  CS*/
while (PORTGP.SI != 1)             /*wait until done   */
  ;
  }

/*********************************************************************/
/* This subroutine write enables the EEPROM memory.                  */
/*********************************************************************/
void EEenbl(void)
  {
EEcson();                          /*turn on CS              */
tmp1 = EEcmd(EEWEN);               /*send enable command     */
EECSOFF;                           /*turn off CS             */
  }

/*********************************************************************/
/* This subroutine write disables the EEPROM memory.                 */
/*********************************************************************/
void EEdsbl(void)
  {
EEcson();                          /*turn on CS              */
tmp1 = EEcmd(EEWDS);               /*send disable command    */
EECSOFF;                           /*turn off CS             */
```

```
  }

/*********************************************************************/
/* This subroutine erases the specified EE register                  */
/*********************************************************************/
void EEerse(byte adr)
  {
EEenbl();                          /*turn on EE programming  */
EEcson();                          /*turn on CS              */
tmp1 = ee_adrcmd(adr,EEERASE);     /*send erase cmd and addr */
EEwait();                          /*wait until not busy     */
EECSOFF;                           /*turn off CS             */
EEdsbl();                          /*turn off EE programming */
  }

/*********************************************************************/
/* This subroutine bulk erases the EEPROM:                           */
/*********************************************************************/
void EEbulk(void)
  {
EEenbl();                          /*turn on EE programming  */
EEcson();                          /*turn on CS              */
tmp1 = EEcmd(EEERAL);              /*  erase command         */
EEwait();                          /*wait until not busy     */
EECSOFF;                           /*turn off CS*/
EEdsbl();                          /*turn off EE programming */
  }

/*********************************************************************/
/* This subroutine reads the specified EE register                   */
/*********************************************************************/
void EErgrd(byte adr)
  {
EEcson();                          /*turn on CS              */
tmp1 = ee_adrcmd(adr,EEREAD);      /*send read cmd and addr  */
PSW.BUSY = 1;                      /*shift out 1 bit         */
PSW.BUSY = 0;                      /*  (sk clock=2 TC cycles)*/
EEbufh = mw_cmd(0x00);             /*read the hi byte        */
EEbufl = mw_cmd(0x00);             /*read the lo byte        */
EECSOFF;                           /*turn off CS             */
  }

/*********************************************************************/
/* This subroutine writes the EE register whose address is in B on   */
/* entry:                                                            */
/*********************************************************************/
void EErgwr(byte adr)
  {
EEenbl();                          /*turn on EE programming  */
EEcson();                          /*turn on CS              */
tmp1 = ee_adrcmd(adr,EEWRITE);     /*send write cmd and addr */
tmp1 = mw_cmd(EEbufh);             /*send the hi byte        */
tmp1 = mw_cmd(EEbufl);             /*send the lo byte        */
EEwait();                          /*wait until not busy     */
EECSOFF;                           /*turn off CS             */
EEdsbl();                          /*turn off EE programming */
  }

/*********************************************************************/
```

```
/* This subroutine does an A-to-D conversion on the requested channel */
/* and then returns the 8-bit result.                                 */
/*********************************************************************/
byte A2D_conv(byte chnl)
   {
      /*select A-to-D converter                                       */
PORTGC.SKSEL = 0;                  /*selects normal SK mode  */
MWCSOFF;                           /*deassert all chip sels  */
PSW.BUSY = 1;                      /*give it one clock       */
PSW.BUSY = 0;                      /*  (sk clock=2 TC cycles)*/
PORTDD.CSA2D = 0;                  /*assert A2D CS*          */

      /*select appropriate start bits depending on the channel        */
if (chnl&0x01) MWSR = 0b11100000; else MWSR = 0b11000000;

      /*give it four MICROWIRE clocks (sk clock=2 TC cycles)          */
#asm
SBIT   BUSY,PSW
NOP
NOP
NOP
NOP
RBIT   BUSY,PSW
#endasm

MWOUT(0);                          /*send dummy data         */
A2DCSOFF;                          /*turn off CS*            */
return(MWSR);                      /*return result           */
```

**LIFE SUPPORT POLICY**

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component is any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

**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