Low Speed USB Design Using µPSD

This application note describes the design of a low-speed USB function for the µPSD32xxA MCU series (see Table 2.), and how to use the µPSD USB simple demonstration program. It gives answers to frequently asked questions, and to design issues (such as those to do with the newer USB 2.0 host controllers, see Table 1.) and gives designers the basic elements to design applications using the low-speed USB port of the µPSD devices.

Table 1. Current Types of USB

<table>
<thead>
<tr>
<th>Type</th>
<th>Latest USB Specifications</th>
<th>Transmission Speed</th>
<th>Maximum Data Rate</th>
<th>µPSD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Theoretical</td>
<td>Typical</td>
</tr>
<tr>
<td>Low-speed</td>
<td>Revision 2.0</td>
<td>1.5 Mbit/s</td>
<td>8 KByte/s</td>
<td>6 KByte/s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>~ 24 KByte/s</td>
<td>11 KByte/s</td>
</tr>
<tr>
<td>Full-speed</td>
<td>Revision 2.0</td>
<td>12 Mbit/s</td>
<td>1.4 MByte/s</td>
<td>1.1 MByte/s</td>
</tr>
<tr>
<td>High-speed</td>
<td>Revision 2.0</td>
<td>480 Mbit/s</td>
<td>60 MByte/s</td>
<td>&lt; 60 MByte/s</td>
</tr>
</tbody>
</table>

Note: 1. The theoretical maximum data rates cannot be reached by most USB devices, including the µPSD, whereas the typical maximum data rates can, under certain conditions. Actual values can vary due to USB and PC differences, etc. The µPSD32xxA data rates were tested with the DK3200 development kit and the USB demonstration program.
2. Interrupt transfers on USB1.1
3. On some (USB 2.0) fast hosts

Table 2. µPSD USB Family (February 2004)

<table>
<thead>
<tr>
<th>Part Number</th>
<th>Max. Clock</th>
<th>I/O</th>
<th>TQFP</th>
<th>Supply</th>
<th>Main Flash</th>
<th>2nd Flash</th>
<th>SRAM</th>
<th>USB</th>
</tr>
</thead>
<tbody>
<tr>
<td>µPSD3234A-40T6</td>
<td>40 MHz</td>
<td>37</td>
<td>52</td>
<td>5 V</td>
<td>256 KBytes</td>
<td>32 KBytes</td>
<td>8 KBytes</td>
<td>√</td>
</tr>
<tr>
<td>µPSD3234A-40U6</td>
<td>40 MHz</td>
<td>46</td>
<td>80</td>
<td>5 V</td>
<td>256 KBytes</td>
<td>32 KBytes</td>
<td>8 KBytes</td>
<td>√</td>
</tr>
<tr>
<td>µPSD3254A-40T6</td>
<td>40 MHz</td>
<td>37</td>
<td>52</td>
<td>5 V</td>
<td>256 KBytes</td>
<td>32 KBytes</td>
<td>32 KBytes</td>
<td>√</td>
</tr>
<tr>
<td>µPSD3254A-40U6</td>
<td>40 MHz</td>
<td>46</td>
<td>80</td>
<td>5 V</td>
<td>256 KBytes</td>
<td>32 KBytes</td>
<td>32 KBytes</td>
<td>√</td>
</tr>
</tbody>
</table>

This application note also features a simple demonstration example application, including firmware for running on the DK3200 development kit, and a Windows application running on a standard PC.
# AN1886 - APPLICATION NOTE

## TABLE OF CONTENTS

1. Current Types of USB .................................................. 1
2. µPSD USB Family (February 2004) .................................. 1

### IAP USB Demonstration Program .................................. 5

3. USB Demonstration Program Revision History ...................... 5

### Hardware Design .................................................... 6

4. USB Input/Output Design inside the µPSD32xxA ................... 6
5. Typical USB Connection Circuit for 5V (µPSD32xx) Systems .... 6
6. Possible Frequencies for µPSD with Low-speed USB ................ 7
7. µPSD32xx USB SFR Register Map .................................... 7

### Software Design .................................................... 8

8. List of Header Files .................................................. 8
9. List of C Source Files ................................................ 8
10. USB Communication Flow ............................................ 8
11. USB Host/Device Detailed Communication Flow View .......... 9
12. Supported Endpoints ................................................ 9
13. USB Demonstration Program Data Structures ..................... 10
14. USB Bus Level Interface ............................................ 10
15. Bus Enumeration ..................................................... 11
16. Endpoint0 .......................................................... 12
17. Endpoint1 (LCD display mirror) .................................. 12
18. Endpoint Packet Byte Function Assignment .................... 12
19. USB Setup Packet Handling – Standard Device Requests ....... 12

### USB Requests in the USB Demonstration Program ............... 13

20. Descriptor Type (the high byte of wValue in the Get_Descriptor request) .... 13
21. Valid Types of Class Descriptor .................................. 13
22. Get_Descriptor Request ............................................. 13
23. Components of a Get Descriptor Request ......................... 14
24. Get_Report Request .................................................. 14
25. Components of a Get_Report Request ............................ 14
26. SET_Report Request ................................................ 14
27. Components of a SET_Report Request ............................ 14
28. USB Descriptors ..................................................... 15
29. USB and HID Descriptor Structure ................................ 15
30. GET_Descriptor Request Servicing Routine ..................... 16
31. Descriptors ......................................................... 16
32. Example of USB Descriptors for HID Class Devices ............ 17
33. Device Descriptor ................................................... 18
34. Configuration Descriptor ........................................... 19
Table 17. Interface Descriptor \hspace{1cm} 19
Table 18. HID Descriptor \hspace{1cm} 20
Table 19. Endpoint Descriptor \hspace{1cm} 20
Table 20. Report Descriptor \hspace{1cm} 21

High Level Interface \hspace{1cm} 22
HID Level Interface \hspace{1cm} 22
SUSPEND Mode and RESUME Mode Handling Mechanism \hspace{1cm} 23
USB Disconnect on Demand Feature \hspace{1cm} 24

Important Rules when Starting to Design USB Applications with µPSD32xxA \hspace{1cm} 25

Differences between USB 1.1 Hosts and Some USB 2.0 High-speed Hosts \hspace{1cm} 25
Figure 5. Traffic on USB (µPSD32xxA connected to USB1.1 Host, or to an Intel 82801DBx Type USB 2.0 Host) \hspace{1cm} 25
Figure 6. Traffic on USB (µPSD32xxA connected to an NEC 720100A Type USB 2.0 Host) \hspace{1cm} 26

USB Interrupt must have the Highest Priority Interrupt Level \hspace{1cm} 26
Table 21. Description of the IP Bits \hspace{1cm} 27
Table 22. Description of the IPA Bits \hspace{1cm} 27

Length of USB Interrupt Service Routine \hspace{1cm} 28
Table 23. Oscillator Frequencies Suitable for Applications with USB \hspace{1cm} 28
Figure 7. USB Demonstration Program USB Drivers Immunity versus Length of USB ISR (with respect to Oscillator Frequency) \hspace{1cm} 28

Difference between Relative and Absolute Register Usage \hspace{1cm} 28
Table 24. Suggested Assignment of Register Banks \hspace{1cm} 30

USB may Generate Spurious Interrupts when the Device is Plugged in \hspace{1cm} 30
µPSD USB IP sometimes Triggers an Interrupt but no Bit of UISTA is Set \hspace{1cm} 30
µPSD USB IP Generates Interrupts with an Undocumented Flag \hspace{1cm} 30
USB State Machine does not Clear the STALL0 Bit when a SETUP Packet is Received \hspace{1cm} 30
The Number of Data Bytes Received in a Data Packet can be Greater than Eight \hspace{1cm} 31

C Source and Assembler Output can have Different Functionality \hspace{1cm} 31

Question: So How Must I Write My Drivers? \hspace{1cm} 32
Question: Must I Enable and Service End-of-Packet Interrupt Requests? \hspace{1cm} 32
Question: Is the Length of USB ISR Limited? \hspace{1cm} 32

Limits Set by the Specification: \hspace{1cm} 32
Table 25. Examples of Limits set by USB Standards \hspace{1cm} 32

Known limits: \hspace{1cm} 33
Conclusion: \hspace{1cm} 33

Question: My Device with µPSD is not Properly Detected by Windows \hspace{1cm} 33
Figure 8. USB ISR Response Time \hspace{1cm} 33
Figure 9. USB ISR Response Time - Detail \hspace{1cm} 34

Problems with Windows98 and Multiple-function USB Devices \hspace{1cm} 34
Symptoms \hspace{1cm} 34
Cause \hspace{1cm} 34
Conclusion \hspace{1cm} 34
Status \hspace{1cm} 34

More Information \hspace{1cm} 35
IAP USB DEMONSTRATION PROGRAM

The USB demonstration program was developed to provide a basic example of a customer application using a µPSD device. This application note forms part of the documentation of this product, and traces the changes and important steps at which the USB demonstration was modified to meet customer requirements (Table 3.).

Table 3. USB Demonstration Program Revision History

<table>
<thead>
<tr>
<th>Version</th>
<th>Release Date</th>
<th>Description, Features, Known Problems</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.07</td>
<td>March 2003</td>
<td>Basic version</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not functional on most of USB2.0 hosts</td>
</tr>
<tr>
<td>1.07E</td>
<td>April 2003</td>
<td>Uses (services) also EOP interrupts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Very high CPU load</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Functional on many USB2.0 hosts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Problems with implementation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Very low immunity to length of USB ISR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>STALL0 bit problem solved in EOP service routine</td>
</tr>
<tr>
<td>2.0_RP</td>
<td>December 2003</td>
<td>Functional on many of USB2.0 hosts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Relative registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses short tests of USB INT flags (for example RXD0F bit instead of UISTA &amp; uRXD0F)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Safe usage of USB registers (assembler instruction added)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Immunity to length of USB ISR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Problems with Win98</td>
</tr>
<tr>
<td>2.0_FP</td>
<td>January 2004</td>
<td>Functional with all tested USB2.0 hosts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Successfully tested on many OS platforms (Win98SE up to WinXP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Relative registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses short tests of USB INT flags (for example RXD0F bit instead of UISTA &amp; uRXD0F)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Safe usage of USB registers (assembler instruction added)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USTA packet length problem fixed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Improved immunity to length of USB ISR</td>
</tr>
<tr>
<td>2.1</td>
<td>February 2004</td>
<td>Functional with all tested USB2.0 hosts</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Successfully tested on many OS platforms (Win98SE up to WinXP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Relative registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Uses short tests of USB INT flags (for example RXD0F bit instead of UISTA &amp; uRXD0F)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Safe usage of USB registers (assembler instruction added)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Simple handling and modification of USB descriptors</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Physical descriptor supported</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Improved routines</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Improved immunity to length of USB ISR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>USTA packet length problem fixed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reduced number of headers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The first official version</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Today (February 2004) are not known any uncovered problems with USB2.0 hosts</td>
</tr>
</tbody>
</table>
HARDWARE DESIGN

Figure 1. shows a simplified view of the USB input/output stage inside the µPSD. According to the USB specification for a low-speed USB device, the minimum requirement is for one external signaling resistor (signal D– to Vcc) to be added.

Figure 1. USB Input/Output Design inside the µPSD32xxA

Figure 2. shows the typical and generally used USB circuit in µPSD32xxA systems (5V power supply only). Typically three resistors (one low-speed device pull-up signaling resistor and two serial resistors (of 50 to 100 Ω) are added.

Figure 2. Typical USB Connection Circuit for 5V (µPSD32xx) Systems

Note: 1. The 7.5kΩ resistor is used for low speed identification
Possible Frequencies for µPSD with Low-speed USB

Any suitable external clock or crystal design can be used, as described in AN1843. It is strongly recommended to use the 36MHz oscillator frequency for the USB, due to performance requirements when supporting fast USB chip sets. (24MHz can also be used, but with more limitations.)

The following formula can be used to determine other possible oscillator frequencies. In all cases, the base frequency for the USB block must be 6MHz, and oscillator frequency must not exceed 40MHz. It can be obtained using the on-chip programmable prescaler. The USCL 8-bit Prescaler Register for the USB is located in the SFR area at address E1h (Table 4). The USCL should be loaded with a value that results in a clock rate of 6MHz for the USB, using the following formula (where \( f_{\text{OSC}} \) is the MCU clock input frequency):

\[
USB_{\text{clock in}} = \frac{f_{\text{OSC}}/2}{\text{prescaler} + 1}
\]

Note: The USB works only with the MCU clock frequencies of 12, 24, or 36MHz. The prescaler values for these frequencies are 0, 1, and 2. Do not forget to change the OSC constant in the uPSD_HARDWARE.h file of USB DEMO (36MHz is the default value).

Table 4. µPSD32xx USB SFR Register Map

<table>
<thead>
<tr>
<th>SFR Add</th>
<th>Reg Name</th>
<th>Bit Register Name</th>
<th>Reset Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
<tr>
<td>E1</td>
<td>USCL</td>
<td>8-bit rescaler for USB logic</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>E6</td>
<td>UDT1</td>
<td>UDT1.7 UDT1.6 UDT1.5 UDT1.4 UDT1.3 UDT1.2 UDT1.1 UDT1.0</td>
<td>00</td>
<td>USB Endpt1 Data Xmit</td>
</tr>
<tr>
<td>E7</td>
<td>UDT0</td>
<td>UDT0.7 UDT0.6 UDT0.5 UDT0.4 UDT0.3 UDT0.2 UDT0.1 UDT0.0</td>
<td>00</td>
<td>USB Endpt0 Data Xmit</td>
</tr>
<tr>
<td>E8</td>
<td>UISTA</td>
<td>SUSPND RSTF TXD0F RXD0F RXD1F EOPF RESUMF</td>
<td>00</td>
<td>USB Interrupt Status</td>
</tr>
<tr>
<td>E9</td>
<td>UIEN</td>
<td>SUSPNDIE RSTE RSTFIE TXD0IE RXD0IE TXD1IE EOPIE RESUMIE</td>
<td>00</td>
<td>USB Interrupt Enable</td>
</tr>
<tr>
<td>EA</td>
<td>UCON0</td>
<td>TSEQ0 STALL0 TX0E RX0E TP0SIZ3 TP0SIZ2 TP0SIZ1 TP0SIZ0</td>
<td>00</td>
<td>USB Endpt0 Xmit Control</td>
</tr>
<tr>
<td>EB</td>
<td>UCON1</td>
<td>TSEQ1 EP12SEL FRESUM TP1SIZ3 TP1SIZ2 TP1SIZ1 TP1SIZ0</td>
<td>00</td>
<td>USB Endpt1 Xmit control</td>
</tr>
<tr>
<td>EC</td>
<td>UCON2</td>
<td>— — — — SOUT EP2E EP1E STALL2 STALL1</td>
<td>00</td>
<td>USB Control Register</td>
</tr>
<tr>
<td>ED</td>
<td>USTA</td>
<td>RSEQ SETUP IN OUT RP0SIZ3 RP0SIZ2 RP0SIZ1 RP0SIZ0</td>
<td>00</td>
<td>USB Endpt0 Status</td>
</tr>
<tr>
<td>EE</td>
<td>UADR</td>
<td>USBEN UADD6 UADD5 UADD4 UADD3 UADD2 UADD1 UADD0</td>
<td>00</td>
<td>USB Address Register</td>
</tr>
<tr>
<td>EF</td>
<td>UDR0</td>
<td>UDR0.7 UDR0.6 UDR0.5 UDR0.4 UDR0.3 UDR0.2 UDR0.1 UDR0.0</td>
<td>00</td>
<td>USB Endpt0 Data Recv</td>
</tr>
</tbody>
</table>
SOFTWARE DESIGN

The USB demonstration program consists of the following files (Table 5. and Table 6.).

Table 5. List of Header Files

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP_INTR.H</td>
<td>Header shared with Windows USB demo application, definition of message formats and USB demo command codes.</td>
</tr>
<tr>
<td>uPSD3200.H</td>
<td>The main header with definition of all μPSD registers, areas and basic types</td>
</tr>
<tr>
<td>LCD_3200.H</td>
<td>Header of LCD driver</td>
</tr>
<tr>
<td>uPSD_HARDWARE.H</td>
<td>Short header with definition of PSD and LCD block start address definitions as well as μPSD’s oscillator and USB input frequencies</td>
</tr>
<tr>
<td>uPSD_USB.H</td>
<td>Short header with external definitions of uPSD_USB.c file</td>
</tr>
<tr>
<td>USB.H</td>
<td>The main header with USB descriptor and packet structures, USB and HID codes</td>
</tr>
</tbody>
</table>

Table 6. List of C Source Files

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAIN.C</td>
<td>The main routine(s), USB ISR, command execution</td>
</tr>
<tr>
<td>LCD_3200.C</td>
<td>LCD driver including copy buffer</td>
</tr>
<tr>
<td>FLASHCODE1.C</td>
<td>FLASH routines for μPSD</td>
</tr>
<tr>
<td>uPSD_USB.C</td>
<td>The main USB driver</td>
</tr>
<tr>
<td>uPSD_CFG.C</td>
<td>Short USB configuration file, USB prescaler</td>
</tr>
<tr>
<td>uPSD_DEC.C</td>
<td>All USB descriptors and strings</td>
</tr>
</tbody>
</table>

When developing USB code the for μPSD32xx family, there are software design issues or considerations that must be taken into account. These are discussed in this section.

USB Communication Flow

The USB provides a communication service between software on the host and its USB function. Functions can have different communication flow requirements for different client-to-function interactions. The USB provides better overall bus utilization by allowing the separation of the different communication flows to a USB function. Each communication flow makes use of some bus access to accomplish communication between client and function. Each communication flow is terminated at an endpoint on a device. Device endpoints are used to identify aspects of each communication flow.

Figure 3. shows a more detailed view of communication flow between the device and the host. It supports the logical device and function layer communication flows. These actual communications flows cross several interface boundaries.
Chapters 6 to 8 of the USB specification describe the mechanical, electrical, and protocol interface definitions of the USB “wire”. Chapter 9 describes the USB device programming interface that allows a USB device to be manipulated from the host side of the wire.

Chapter 10 describes two host side software interfaces:

- **Host Controller Driver (HCD):** The software interface between the USB Host Controller and USB System Software. This interface allows a range of Host Controller implementations without requiring all host software to be dependent on any particular implementation. One USB Driver can support different Host Controllers without requiring specific knowledge of a Host Controller implementation. A Host Controller implementer provides an HCD implementation that supports the Host Controller.

- **USB Driver (USBD):** The interface between the USB System Software and the client software. This interface provides clients with convenient functions for manipulating USB devices.

### Endpoints Supported by µPSD32xx

A USB logical device appears to the USB system as a collection of endpoints. Endpoints are grouped into endpoint sets that implement an interface. Interfaces are views to the function. The USB System Software manages the device using the Default Control Pipe. Client software manages an interface using pipe bundles (associated with an endpoint set). Client software requests that data be moved across the USB between a buffer on the host and an endpoint on the USB device. The Host Controller (or USB device, depending on transfer direction) converts the data to packets to move it over the USB. The Host Controller also coordinates when bus access is used to move the packet of data over the USB.
Table 7. Supported Endpoints

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Function</th>
<th>Maximal packet size</th>
<th>Maximal traffic</th>
<th>Supported directions</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Endpoint 0</td>
<td>Control</td>
<td>8 Bytes</td>
<td>Any time</td>
<td>Both IN and OUT</td>
<td>UCON0, UDR0, UDT0, USTA</td>
</tr>
<tr>
<td>Endpoint 1</td>
<td>Interrupt endpoint</td>
<td>8 Bytes</td>
<td>1 per 10-255ms</td>
<td>IN only, device → host (PC)</td>
<td>UCON1, UDT1, UCON2</td>
</tr>
<tr>
<td>Endpoint 2</td>
<td>Interrupt endpoint</td>
<td>8 Bytes</td>
<td>1 per 10-255ms</td>
<td>IN only, device → host (PC)</td>
<td>UCON1, UDT1, UCON2</td>
</tr>
</tbody>
</table>

The appropriate endpoint is selected by bit-6 (EP12SEL) in the UCON1 register before the USB Endpoint cache is filled by writing data to UDT1 register. The appropriate endpoint must be also enabled in UCON2 register (bits EP1E and EP2E).

**Data Structures and Data-flow**

In between the application and the top-level firmware routines, data structures are broken up into HID reports and USB packets. Table 8. illustrates which application and firmware routines conceptually communicate with each other using the same structures.

Table 8. USB Demonstration Program Data Structures

<table>
<thead>
<tr>
<th>Host Routine(s)</th>
<th>Data Structure(s)</th>
<th>Firmware Routine(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>High level button handlers such as OnReadFlash, OnWriteFlash, OnErase, etc.</td>
<td>MCU_CMD</td>
<td>OnReportReceived, PrepareTransmitSegment</td>
</tr>
<tr>
<td>HidD_GetFeature</td>
<td>Feature Report</td>
<td>PrepareTransmitSegment</td>
</tr>
<tr>
<td>WriteFile</td>
<td>Output Report</td>
<td>OnReportReceived</td>
</tr>
<tr>
<td>ReadFile</td>
<td>Input Report</td>
<td>OnTransmitEP1</td>
</tr>
<tr>
<td>USB Host Controller</td>
<td>Max. 8 bytes length control and interrupt endpoint packets.</td>
<td>OnTransmitEP1, PrepareTransmitSegment, OnReportSegmentReceived</td>
</tr>
</tbody>
</table>

**USB Bus Level Interface**

Low speed HID devices, such as the µPSD32xxA, can pass a maximum of eight bytes per bus transaction. The USB host controller driver, on the Windows side, and the firmware, on the µPSD32xxA side, must break up HID reports, when necessary, into 8-byte USB packets.

Since the input report for the example was defined to be eight bytes in length, there are no complications in passing the LCD mirror data to the application. Mirror data is passed in the OnTransmitEP1 routine, in Firmware\main.c. Whenever one packet is successfully transmitted on EP1 IN, OnTransmitEP1 prepares the next packet with the next segment of the LCD mirror buffer.

Since the Output and Feature reports are used to pass command data, they are larger than eight bytes. The firmware must deal with the complication of assembling and disassembling reports into 8-byte setup packets to or from the control endpoint. The firmware OnReportSegmentReceived routine handles the process of accumulating the multiple segments of an Output report. When the complete report has been assembled in the buffer, the OnReportReceived routine is called to handle the MCU_CMD command.
structure or associated data contained within the report. The firmware PrepareTransmitSegment and On-
ReportTransmitted routines perform similar functions for Input reports going to the host.

**Bus Enumeration**

When a USB device is attached to, or removed from, the USB, the host uses a process known as bus enumeration to identify and manage the necessary device state changes. When a USB device is attached to a powered port, the following actions are taken:

1. The hub, to which the USB device is now attached, informs the host of the event via a reply on its status change pipe (refer to Section 11.13.3 of USB specification for more information). At this point, the USB device is in the Powered state, and the port to which it is attached is disabled.

2. The host determines the exact nature of the change by querying the hub.

3. Now that the host knows the port to which the new device has been attached, the host waits for at least 100ms to allow completion of an insertion process, and for power at the device to become stable. The host then issues a port enable and reset command to that port.

4. The hub maintains the reset signal to that port for 10ms. When the reset signal is released, the port has been enabled. The USB device is now in the Default (unenumerated) state, and can draw no more than 100mA from VBUS. All of its registers and state have been reset, and it answers to the default address.

5. The host assigns a unique address to the USB device, moving the device to the Address state.

6. Before the USB device receives a unique address, its Default Control Pipe is still accessible via the default address. The host reads the device descriptor to determine what actual maximum data payload size this USB device’s default pipe can use.

7. The host reads the configuration information from the device by reading each configuration (0 to n-1, where n is the number of configurations). This process may take several milliseconds to complete.

8. Based on the configuration information and how the USB device will be used, the host assigns a configuration value to the device. The device is now in the Configured state, and all of the endpoints in this configuration have taken on their described characteristics. The USB device may now draw the amount of VBUS power described in its descriptor for the selected configuration. From the device’s point of view it is now ready for use.

When the USB device is removed, the hub again sends a notification to the host. Detaching a device disables the port to which it had been attached. Upon receiving the detach notification, the host will update its local topological information.

The USB reset is independent of the chip reset. A USB reset signal resets the USB interface peripheral, but not the µPSD core and other peripherals. But USB reset can also be used to wake-up or reset the µPSD.

When a USB reset signal is detected on the bus, the RSTF bit in the UISTA register is set, and a USB interrupt is generated.

During the enumeration phase, the host performs a bus enumeration to identify the attached device and to assign a unique address to it. The device responds to the requests sent by the host during the enumeration process on its default pipe (Endpoint0).

The enumeration steps are as follows (with each of the following commands being sent):

1. **Get Device Descriptor.** The host sends a get device descriptor request. The device replies with its device descriptor to report its attributes (Device Class, maximum packet size for endpoint zero).

2. **Set Address.** A USB device uses the default address after reset until the host assigns a unique address using the set address request. The firmware writes the device address assigned by the host in the UADR register. The firmware must write the device address only after completion of the set address operation because the status stage that concludes the control transfer still uses the default address.
3. **Get Configuration.** The host sends a get configuration. The device replies with its configuration descriptor, interface descriptor and endpoint descriptor. The configuration descriptor describes the number of interfaces provided by the configuration, the power source (Bus or Self powered) and the maximum power consumption of the USB device from the bus. The Interface descriptor describes the number of endpoints used by this interface. The Endpoint descriptor describes the transfer type supported, and the bandwidth requirements. Depending on the function implemented, other class-specific descriptors may be returned by the device.

4. **Set Configuration.** The host assigns a configuration value to the device, based on the configuration information. The device is then in the Configured state, and can draw the amount of power described in the configuration descriptor. The device is now configured and ready to be used.

**Endpoint0**

Endpoint0 is used, in this example, for all enumeration traffic: sending data to, and receiving data from, the µPSD unit, sending commands to the µPSD.

The main USB interrupt service routine services the main and most important tasks connected with Endpoint0. This routine is shown in APPENDIX A.

**Endpoint1 (LCD display mirror)**

Periodically and automatically sent data packets from Endpoint1 implement the LCD mirror. Each data packet consists of eight bytes, and has the format shown in Table 9.

**Table 9. Endpoint Packet Byte Function Assignment**

<table>
<thead>
<tr>
<th>Byte</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>Start Position</td>
<td>Character @ POS+0</td>
<td>Character @ POS+1</td>
<td>Character @ POS+2</td>
<td>Character @ POS+3</td>
<td>Character @ POS+4</td>
<td>Character @ POS+5</td>
<td>Character @ POS+6</td>
</tr>
</tbody>
</table>

Carriage Return (CR, 0x0D) and Line Feed (LF, 0x0A) characters are inserted at the end of every display line, and a terminating null character (NUL, 0x00) is inserted at the end. For example, if the display has the following content:

USB DEMO V2.0_FP
199 04 04M 12566

It will be transmitted as:

Packet: 0x00,'U','S','B',' ','D','E','M'
Packet: 0x07,'O',' ','V','2','.','0','_'
Packet: 0x0E,'F','P',0x0D,0x0A,'1',',','9',',','9'
Packet: 0x15,' ','0','4',' ','0','4','M'
Packet: 0x1C,' ','1','2',' ','5','6','0x00

The code that is responsible for doing this can be found in main.c. This routine is shown in APPENDIX B.

**USB Setup Packet Handling – Standard Device Requests**

Most of the enumeration steps, mentioned earlier in this document, are processed in the uPSD_USB.c/OnSetupPacket routine. It is responsible for servicing the standard device requests that have been received from the host. The routine is shown in APPENDIX C.
USB REQUESTS IN THE USB DEMONSTRATION PROGRAM
The HID class uses the standard request Get_Descriptor, as described in the USB specification. When a Get_Descriptor(Configuration) request is issued, it returns the Configuration descriptor, all Interface descriptors, all Endpoint descriptors, and the HID descriptor for each interface. It shall not return the String descriptor, HID Report descriptor or any of the optional HID class descriptors. The HID descriptor shall be interleaved between the Interface and Endpoint descriptors for HID Interfaces. That is, the order shall be:

- Configuration descriptor
- Interface descriptor (specifying HID Class)
- HID descriptor (associated with above Interface)
- Endpoint descriptor (for HID Interrupt In Endpoint)
- Optional Endpoint descriptor (for HID Interrupt Out Endpoint)

Table 10. Descriptor Type (the high byte of \textit{wValue} in the \textit{Get_Descriptor} request)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Reserved should always be 0</td>
</tr>
<tr>
<td>6 to 5</td>
<td>Type</td>
</tr>
<tr>
<td></td>
<td>0 = Standard</td>
</tr>
<tr>
<td></td>
<td>1 = Class</td>
</tr>
<tr>
<td></td>
<td>2 = Vendor</td>
</tr>
<tr>
<td></td>
<td>3 = Reserved</td>
</tr>
<tr>
<td>4 to 0</td>
<td>Descriptor</td>
</tr>
</tbody>
</table>

Table 11. Valid Types of Class Descriptor

<table>
<thead>
<tr>
<th>Value</th>
<th>Class Descriptor Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x21</td>
<td>HID</td>
</tr>
<tr>
<td>0x22</td>
<td>Report</td>
</tr>
<tr>
<td>0x23</td>
<td>Physical descriptor $^1$</td>
</tr>
<tr>
<td>0x24 to 0x2F</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Note: Physical descriptors are supported by the USB demonstration since version 2.1(February 2004).

Get_Descriptor Request
The Get_Descriptor request returns a descriptor for the device. The fields of this are described in Table 12.

For standard USB descriptors, bits 0-4 of \textit{bmRequestType} indicate whether the requested descriptor is associated with the device, interface, endpoint, or other.

The \textit{wValue} field specifies the Descriptor Type in the high byte and the Descriptor Index in the low byte. The low byte is the Descriptor Index used to specify the set for Physical Descriptors, and is reset to zero for other HID class descriptors.

If a HID class descriptor is being requested then the \textit{wIndex} field indicates the number of the HID Interface. If a standard descriptor is being requested then the \textit{wIndex} field specifies the Language ID for string descriptors, and is reset to zero for other standard descriptors.

Requesting Physical Descriptor 0 returns a special descriptor identifying the number of descriptor sets and their sizes.

A Get_Descriptor request with the Physical Index equal to 1 will request the first Physical Descriptor set. A device could possibly have alternate uses for its items. These can be enumerated by issuing subsequent
Get_Descriptor requests while incrementing the Descriptor Index. A device will return the last descriptor set to requests with an index greater than the last number defined in the HID descriptor.

Table 12. Components of a Get_Descriptor Request

<table>
<thead>
<tr>
<th>Part</th>
<th>Standard USB Descriptor</th>
<th>HID Class Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>bmRequestType</td>
<td>100 xxxx</td>
<td>10000001</td>
</tr>
<tr>
<td>bRequest</td>
<td>GET_DESCRIPTOR (0x06)</td>
<td>GET_DESCRIPTOR (0x06)</td>
</tr>
<tr>
<td>wValue</td>
<td>Descriptor Type and Descriptor Index</td>
<td>Descriptor Type and Descriptor Index</td>
</tr>
<tr>
<td>wIndex</td>
<td>0 (zero) or Language</td>
<td>Interface Number</td>
</tr>
<tr>
<td>wLength</td>
<td>Descriptor Length</td>
<td>Descriptor Length</td>
</tr>
<tr>
<td>Data</td>
<td>Descriptor</td>
<td>Descriptor</td>
</tr>
</tbody>
</table>

Get_Report Request

The Get_Report request allows the host to receive a report via the Control pipe. This request is mandatory and must be supported by all devices. The fields of this are described in Table 13.

This request is useful at initialization time for absolute items, and for determining the state of feature items. This request is not intended to be used for polling the device state on a regular basis. The Interrupt In pipe should be used for recurring Input reports. The Input report reply has the same format as the reports from Interrupt pipe. An Interrupt Out pipe may optionally be used for low latency Output reports. Output reports over the Interrupt Out pipe have a format that is identical to output reports that are sent over the Control pipe, if an Interrupt Out endpoint is not declared.

Table 13. Components of a Get_Report Request

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bmRequestType</td>
<td>1010 0001 (0xA1) - CLASS_INTERFACE_TO_HOST</td>
</tr>
<tr>
<td>bRequest</td>
<td>GET_REPORT (0x01)</td>
</tr>
<tr>
<td>wValue</td>
<td>high byte Report Type</td>
</tr>
<tr>
<td></td>
<td>01 - Input</td>
</tr>
<tr>
<td></td>
<td>02 - Output</td>
</tr>
<tr>
<td></td>
<td>03 - Feature</td>
</tr>
<tr>
<td></td>
<td>04 to FF - Reserved</td>
</tr>
<tr>
<td>low byte</td>
<td>Report ID</td>
</tr>
<tr>
<td></td>
<td>Set to 0 (zero) if Report IDs are not used</td>
</tr>
<tr>
<td>wIndex</td>
<td>Interface</td>
</tr>
<tr>
<td>wLength</td>
<td>Report Length</td>
</tr>
<tr>
<td>Data</td>
<td>Report</td>
</tr>
</tbody>
</table>

SET_Report Request

The Set_Report request allows the host to send a report to the device, possibly setting the state of input, output, or feature controls. The fields of this are described in Table 14.

The meaning of the request fields for the Set_Report request is the same as that for the Get_Report request, however the data direction is reversed, and the Report Data is sent from host to device. A device might choose to ignore input Set_Report requests. Alternatively these reports could be used to reset the origin of a control (that is, current position should report zero). The effect of sent reports will also depend on whether the recipient controls are absolute or relative.
USB Descriptors

USB devices report their attributes using descriptors. A descriptor is a data structure with a specific format. Each descriptor begins with a byte-wide field that contains the total number of bytes in the descriptor, followed by a byte-wide field that identifies the descriptor type. Using descriptors allows concise storage of the attributes of individual configurations because each configuration may reuse descriptors or portions of descriptors from other configurations that have the same characteristics. In this manner, the descriptors resemble individual data records in a relational database. Where appropriate, descriptors contain references to string descriptors that provide displayable information describing a descriptor in human-readable form. The inclusion of string descriptors is optional. However, the reference fields within descriptors are mandatory. If a device does not support string descriptors, string reference fields must be reset to zero to indicate that no string descriptor is available. If a descriptor returns with a value in its length field that is less than that defined by this specification, the descriptor is invalid, and should be rejected by the host. If the descriptor returns with a value in its length field that is greater than that defined by this specification, the host ignores the extra bytes, but the next descriptor is located using the length returned rather than the length expected.

Table 14. Components of a SET_Report Request

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bmRequestType</td>
<td>0010 0001 (0x21) - CLASS_INTERFACE_TO_DEVICE</td>
</tr>
<tr>
<td>bRequest</td>
<td>SET_REPORT (0x09)</td>
</tr>
<tr>
<td>wValue</td>
<td>Report Type and Report ID</td>
</tr>
<tr>
<td>wIndex</td>
<td>Interface</td>
</tr>
<tr>
<td>wLength</td>
<td>Report Length</td>
</tr>
<tr>
<td>Data</td>
<td>Report</td>
</tr>
</tbody>
</table>

Figure 4. USB and HID Descriptor Structure
A device may return class- or vendor-specific descriptors in two ways.

- If the class or vendor specific descriptors use the same format as standard descriptors (for example, they start with a length byte, followed by a type byte), they may be returned interleaved with standard descriptors in the configuration information returned by a GetDescriptor(Configuration) request. In this case, the class or vendor-specific descriptors typically follow a related standard descriptor that they modify or extend.

- If the class or vendor specific descriptors are independent of configuration information, or use a non-standard format, a GetDescriptor() request specifying the class or vendor specific descriptor type and index may be used to retrieve the descriptor from the device. A class or vendor specification will define the appropriate way to retrieve these descriptors.

**GET_Descriptor Request Servicing Routine**

The Get_descriptor request is serviced in uPSD_USB.c/OnGetDescriptor routine. This routine is shown in APPENDIX D.

**Descriptors**

The format of all USB descriptors is given by the USB specification standards, and all the descriptors can be found in file USB_DESC.C. This can have the following content:

```c
const uchar code string1Desc[];
const uchar code string2Desc[];
const uchar code string3Desc[];
const uchar code stringXDesc[];
```

Next, the *Report* descriptor contains the HID report information that is shown in APPENDIX E.

The descriptor area can also contain physical descriptors. A physical descriptor is a data structure that provides information about the specific part, or parts, of the human body that are activating a control or controls. For example, a physical descriptor might indicate that the right hand thumb is used to activate button 5. An application can use this information to assign functionality to the controls of a device. Note that physical descriptors are entirely optional. They add complexity, and offer very little in return for most devices. However, some devices, particularly those with a large number of identical controls (for example, buttons) will find that physical descriptors help different applications assign functionality to these controls in a more consistent manner. See the HID standard for more information. In the USB demonstration program, it is defined as an empty descriptor:

```c
const uchar code PhysicalReportDesc[] =
{ 0,0,0                                       // no physical descriptor defined now
};
const uchar code PhysicalReportDescSize = sizeof(PhysicalReportDesc);
```

Next, the *Device* descriptor contains detailed information about the USB device, and supported endpoints. This routine is shown in APPENDIX F.

Next, each of the String descriptors contains important strings defining the USB device. String descriptors are optional. As noted previously, if a device does not support string descriptors, all references to string descriptors within device, configuration, and interface descriptors must be reset to zero. String descriptors use *Unicode* encoding. The strings in a USB device may support multiple languages. When requesting a string descriptor, the requester specifies the desired language using a sixteen-bit language *ID (LANGID)* defined by *Microsoft for Windows* as described in *Developing International Software for Windows 95 and Windows NT*. 

String index zero for all languages returns a string descriptor that contains an array of the two-byte LANG-ID codes that are supported by the device.

A USB device may omit all string descriptors. USB devices that omit all string descriptors shall not return an array of LANGID codes. The array of LANGID codes is not NULL-terminated. The size of the array (in bytes) is computed by subtracting two from the value of the first byte of the descriptor.

The most important and used strings are: Language ID string, Manufacturer, Product and Serial number. These are shown in Appendix G.

StringX was created as default string for all not supported or not implemented strings:

```c
const uchar code stringXDesc[2] =
{
    0, 0,                                      // Size, Type
}
```

At the end, mainly for easy access, all the strings are collected in the following structure:

```c
// Table of String Descriptors
const uchar * const code stringDescTable[] =
{
    string0Desc,
    string1Desc,
    string2Desc,
    string3Desc,
    stringXDesc                   //no string
}
```

Example of USB Descriptors for HID Class Devices

The sample device is a low-speed 105-key keyboard with an integrated pointing device. This device could be built using just one interface. However, two are used in this example so the device can support the boot protocol. As a result there are two Interface, Endpoint, HID and Report descriptors for this device.
### Table 15. Device Descriptor

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
<th>Example Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bLength</td>
<td>0</td>
<td>1</td>
<td>Size of this descriptor (in Bytes)</td>
<td>12h</td>
</tr>
<tr>
<td>bDescriptorType</td>
<td>1</td>
<td>1</td>
<td>Device descriptor type (assigned by USB)</td>
<td>01h</td>
</tr>
<tr>
<td>bcdUSB</td>
<td>2</td>
<td>2</td>
<td>USB HID Specification Release 1.0</td>
<td>0100h</td>
</tr>
<tr>
<td>bDeviceClass</td>
<td>4</td>
<td>1</td>
<td>Class code (assigned by USB). Note that the HID class is defined in the Interface descriptor</td>
<td>00h</td>
</tr>
<tr>
<td>bDeviceSubClass</td>
<td>5</td>
<td>1</td>
<td>Subclass code (assigned by USB). These codes are qualified by the value of the bDeviceClass field</td>
<td>00h</td>
</tr>
<tr>
<td>bDeviceProtocol</td>
<td>6</td>
<td>1</td>
<td>Protocol code. These codes are qualified by the value of the bDeviceProtocol field</td>
<td>00h</td>
</tr>
<tr>
<td>bMaxPacketSize0</td>
<td>7</td>
<td>1</td>
<td>Maximum packet size for Endpoint0 (the only valid values are: 8, 16, 32 or 64)</td>
<td>08h</td>
</tr>
<tr>
<td>idVendor</td>
<td>8</td>
<td>2</td>
<td>Vendor ID (assigned by USB). For this example, we will work with the value 0xFFFF</td>
<td>FFFFh</td>
</tr>
<tr>
<td>idProduct</td>
<td>10</td>
<td>2</td>
<td>Product ID (assigned by the manufacturer)</td>
<td>0001h</td>
</tr>
<tr>
<td>bcdDevice</td>
<td>12</td>
<td>2</td>
<td>Device release number (assigned by the manufacturer)</td>
<td>0100h</td>
</tr>
<tr>
<td>iManufacturer</td>
<td>14</td>
<td>1</td>
<td>Index of string descriptor describing the manufacturer</td>
<td>04h</td>
</tr>
<tr>
<td>iProduct</td>
<td>15</td>
<td>1</td>
<td>Index of string descriptor describing the product</td>
<td>0Eh</td>
</tr>
<tr>
<td>iSerialNumber</td>
<td>16</td>
<td>1</td>
<td>Index of string descriptor describing the device’s serial number</td>
<td>30h</td>
</tr>
<tr>
<td>bNumConfigurations</td>
<td>17</td>
<td>1</td>
<td>Number of possible configurations</td>
<td>01h</td>
</tr>
</tbody>
</table>
### Table 16. Configuration Descriptor

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
<th>Example Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bLength</td>
<td>0</td>
<td>1</td>
<td>Size of this descriptor (in Bytes)</td>
<td>09h</td>
</tr>
<tr>
<td>bDescriptorType</td>
<td>1</td>
<td>1</td>
<td>Configuration descriptor type (assigned by USB)</td>
<td>02h</td>
</tr>
<tr>
<td>wTotalLength</td>
<td>2</td>
<td>2</td>
<td>Total length of data returned of this configuration. Includes the combined length of all returned descriptors (configuration, interface, endpoint, and HID) returned for this configuration. This value includes the HID descriptor but none of the other HID class descriptors (report or descriptor)</td>
<td>003Bh</td>
</tr>
<tr>
<td>bNumInterfaces</td>
<td>4</td>
<td>1</td>
<td>Number of interfaces supported by this configuration</td>
<td>02h</td>
</tr>
<tr>
<td>bConfigurationValue</td>
<td>5</td>
<td>1</td>
<td>Value to use as an argument to Set Configuration</td>
<td>01h</td>
</tr>
<tr>
<td>iConfiguration</td>
<td>6</td>
<td>1</td>
<td>Index of string descriptor describing this configuration. In this case there is none</td>
<td>00h</td>
</tr>
<tr>
<td>bmAttributes</td>
<td>7</td>
<td>1</td>
<td>Configuration characteristics 7: bus powered 6: self powered 5: Remote wake-up 4..0: reserved (always 0)</td>
<td>10100000b</td>
</tr>
<tr>
<td>MaxPower</td>
<td>8</td>
<td>1</td>
<td>Maximum power consumption of USB device from bus in this specific configuration when the device is fully operations. Expressed in 2mA units (the arbitrarily chosen number in this example is 32h, which is 50 in decimal, and hence represents 100mA)</td>
<td>32h</td>
</tr>
</tbody>
</table>

### Table 17. Interface Descriptor

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
<th>Example Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bLength</td>
<td>0</td>
<td>1</td>
<td>Size of this descriptor (in Bytes)</td>
<td>09h</td>
</tr>
<tr>
<td>bDescriptorType</td>
<td>1</td>
<td>1</td>
<td>Interface descriptor type (assigned by USB)</td>
<td>04h</td>
</tr>
<tr>
<td>bInterfaceNumber</td>
<td>2</td>
<td>1</td>
<td>Number of the interface. Zero-based value identifying the index in the array of concurrent interfaces supported by this configuration</td>
<td>00h</td>
</tr>
<tr>
<td>bAlternateSetting</td>
<td>3</td>
<td>1</td>
<td>Value used to select alternate setting for the interface identified in the previous field</td>
<td>00h</td>
</tr>
<tr>
<td>bNumEndpoints</td>
<td>4</td>
<td>1</td>
<td>Number of endpoints used by this interface (excluding Endpoint0). If this value is zero, this interface only uses Endpoint0</td>
<td>01h</td>
</tr>
<tr>
<td>bInterfaceClass</td>
<td>5</td>
<td>1</td>
<td>Class code (HID code assigned by USB)</td>
<td>03h</td>
</tr>
<tr>
<td>bInterfaceSubClass</td>
<td>6</td>
<td>1</td>
<td>Subclass code 0: No subclass 1: Boot interface subclass</td>
<td>01h</td>
</tr>
<tr>
<td>bInterfaceProtocol</td>
<td>7</td>
<td>1</td>
<td>Protocol code 0: None 1: Keyboard 2: Mouse</td>
<td>01h</td>
</tr>
<tr>
<td>iInterface</td>
<td>8</td>
<td>1</td>
<td>Index of string descriptor describing this interface</td>
<td>00h</td>
</tr>
</tbody>
</table>
### Table 18. HID Descriptor

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
<th>Example Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bLength</td>
<td>0</td>
<td>1</td>
<td>Size of this descriptor (in Bytes)</td>
<td>09h</td>
</tr>
<tr>
<td>bDescriptorType</td>
<td>1</td>
<td>1</td>
<td>HID descriptor type (assigned by USB)</td>
<td>21h</td>
</tr>
<tr>
<td>bcdHID</td>
<td>2</td>
<td>2</td>
<td>HID Class Specification release number, in binary-coded decimal, for example 2.10 is 210h</td>
<td>0210h</td>
</tr>
<tr>
<td>bCountryCode</td>
<td>4</td>
<td>1</td>
<td>Hardware target country</td>
<td>00h</td>
</tr>
<tr>
<td>bNumDescriptors</td>
<td>5</td>
<td>1</td>
<td>Number of HID class descriptors to follow</td>
<td>01h</td>
</tr>
<tr>
<td>bDescriptorType</td>
<td>6</td>
<td>1</td>
<td>Report descriptor type</td>
<td>22h</td>
</tr>
<tr>
<td>bDescriptorLength</td>
<td>7</td>
<td>2</td>
<td>Total length of Report descriptor</td>
<td>3Fh</td>
</tr>
</tbody>
</table>

### Table 19. Endpoint Descriptor

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset (Bytes)</th>
<th>Size (Bytes)</th>
<th>Description</th>
<th>Example Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bLength</td>
<td>0</td>
<td>1</td>
<td>Size of this descriptor (in Bytes)</td>
<td>07h</td>
</tr>
<tr>
<td>bDescriptorType</td>
<td>1</td>
<td>1</td>
<td>Endpoint descriptor type (assigned by USB)</td>
<td>05h</td>
</tr>
<tr>
<td>bEndpointAddress</td>
<td>2</td>
<td>1</td>
<td>The address of the endpoint on the USB device described by this descriptor. The address is encoded as: Direction (ignored for Control Endpoints) 0=Out Endpoint 1=In Endpoint 6..4: reserved (always 0) 3..0: Endpoint Number</td>
<td>10000001b</td>
</tr>
<tr>
<td>bmAttributes</td>
<td>3</td>
<td>1</td>
<td>This field describes the endpoint's attributes when it is configured using the bConfigurationValue 1..0: Transfer type 00=Control 01=Bulk 10=Isochronous 11=Interrupt</td>
<td>00000011b</td>
</tr>
<tr>
<td>wMaxPacketSize</td>
<td>4</td>
<td>2</td>
<td>Maximum packet size that this endpoint is capable of sending or receiving when this configuration is selected. For interrupt endpoints, this value is used to reserve the bus time in the schedule, required for the per frame data payloads. Smaller data payloads may be sent, but will terminate the transfer and thus require intervention to restart.</td>
<td>0008h</td>
</tr>
<tr>
<td>bInterval</td>
<td>6</td>
<td>1</td>
<td>Interval for polling endpoint for data transfers (expressed in milliseconds)</td>
<td>0Ah</td>
</tr>
</tbody>
</table>
Table 20. Report Descriptor

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage Page (Generic Desktop)</td>
<td>0501h</td>
</tr>
<tr>
<td>Usage (Keyboard)</td>
<td>0906h</td>
</tr>
<tr>
<td>Collection (Application)</td>
<td>A101h</td>
</tr>
<tr>
<td>Usage Page (Key Codes)</td>
<td>0507h</td>
</tr>
<tr>
<td>Usage Minimum (224)</td>
<td>19E0h</td>
</tr>
<tr>
<td>Usage Maximum (231)</td>
<td>29E7h</td>
</tr>
<tr>
<td>Logical Minimum (0)</td>
<td>1500h</td>
</tr>
<tr>
<td>Logical Maximum (1)</td>
<td>2501h</td>
</tr>
<tr>
<td>Report Size (1)</td>
<td>7501h</td>
</tr>
<tr>
<td>Report Count (8)</td>
<td>9508h</td>
</tr>
<tr>
<td>Input (Data, Variable, Absolute)</td>
<td>8102h</td>
</tr>
<tr>
<td>Report Count (1)</td>
<td>9501h</td>
</tr>
<tr>
<td>Report Size (8)</td>
<td>7508h</td>
</tr>
<tr>
<td>Input (Constant)</td>
<td>8101h</td>
</tr>
<tr>
<td>Report Count (5)</td>
<td>9505h</td>
</tr>
<tr>
<td>Report Size (1)</td>
<td>7501h</td>
</tr>
<tr>
<td>Usage Page (Page# for LEDs)</td>
<td>0508h</td>
</tr>
<tr>
<td>Usage Minimum (1)</td>
<td>1901h</td>
</tr>
<tr>
<td>Usage Maximum (5)</td>
<td>2905h</td>
</tr>
<tr>
<td>Output (Data, Variable, Absolute)</td>
<td>9102h</td>
</tr>
<tr>
<td>Report Count (1)</td>
<td>9501h</td>
</tr>
<tr>
<td>Report Size (3)</td>
<td>7503h</td>
</tr>
<tr>
<td>Output (Constant)</td>
<td>9101h</td>
</tr>
<tr>
<td>Report Count (6)</td>
<td>9506h</td>
</tr>
<tr>
<td>Report Size (8)</td>
<td>7508h</td>
</tr>
<tr>
<td>Logical Minimum (0)</td>
<td>1500h</td>
</tr>
<tr>
<td>Logical Maximum (101)</td>
<td>2565h</td>
</tr>
<tr>
<td>Usage Page (Key Codes)</td>
<td>0507h</td>
</tr>
<tr>
<td>Logical Minimum (0)</td>
<td>1500h</td>
</tr>
<tr>
<td>Logical Maximum (101)</td>
<td>2565h</td>
</tr>
<tr>
<td>Input (Data, Array)</td>
<td>8100h</td>
</tr>
<tr>
<td>End Collection</td>
<td>00h</td>
</tr>
</tbody>
</table>
High Level Interface
At the highest level, the Windows application and the µPSD32xxA firmware communicates using MCU_CMD structures. This is shown in APPENDIX H. The only exception is LCD display mirror data, which is discussed in the next section.

This structure is a union of sub-structures, each of which specifies the parameters for a particular command.

The first byte of the structure specifies a command code. For instance, the application sends a Flash memory erase command to the firmware by setting up a MCU_CMD structure as follows:

```
reportBuf.report.u.erase.cmd     = CMD_ERASE;
reportBuf.report.u.erase.address = SWAP_UINT16(address);
reportBuf.report.u.erase.flash   = (uchar) flash;
```

Note the SWAP_UINT16 macro used to convert from little-endian to big-endian format for the 8032 processor core of the µPSD32xx.

See the Application Interface, Firmware\app\intr.h, header file for the complete declaration of the MCU_CMD structure along with the CMD_xxx codes for all the commands that can be passed between the application and the firmware. This header file is included in both the application and firmware projects, and has the following content:

```
#define CMD_RESET           0x01
#define CMD_ERASE           0x02
#define CMD_WRITE           0x03
#define CMD_READ            0x04
#define CMD_STATUS          0x05
#define CMD_SET_REGS        0x06
#define CMD_SET_PAGE        0x07
#define CMD_SET_VM          0x08
```

The physical execution of the commands is initiated by the OnReportReceived() routine, which is shown in APPENDIX I.

The only exceptions are:

- the Erase command, which takes a long execution time and thus must be executed in the background, not as a part of USB interrupt service routine
- commands that are executed for more packets (Read and Write commands). These commands are triggered by a software interrupt process.

HID Level Interface
Communication with HID devices is achieved through the structures, called “HID Reports”, that are defined in the descriptor data for each HID device. For more information on HID, see the appropriate sections of this document, or directly in the HID standards.

The USB example firmware defines three standard HID reports in the Report Descriptor, reportDesc, structure in Firmware\usb_desc.c:

- Feature Report 64 bytes of command data, passed through EP0 control endpoint.
- Input Report 8 bytes of raw LCD buffer data, passed through EP1 IN interrupt endpoint
- Output Report 64 bytes of command data, passed through EP0 control endpoint.

The Report Descriptor is read by the Microsoft HID driver during device enumeration.
At the Windows API layer, the application must break up large commands into one or more report structures. HID reports are passed to a HID device using the Win32 WriteFile routine. Reports are read from a
HID device using Feature reports, HidD_GetFeature, and Input reports, ReadFile. All of these routines expect data buffers to include an extra header byte that is used to specify what type of report is being transferred. The application defines a helper structure called REPORT_BUF that puts this extra report ID byte in front of an MCU_CMD structure. REPORT_BUF structures are then passed to and from WriteFile and HidD_GetFeature calls.

The size chosen for the Feature and Output reports (64 bytes) is arbitrary. A larger size improves efficiency for large transfers: there is less overhead per block of command data. However, with large reports, there are more wasted bytes for smaller command structures.

Note: LCD mirror data is passed to the application from the firmware as 8-byte Input reports with no higher level structure defined on top.

**SUSPEND Mode and RESUME Mode Handling Mechanism**

Since version 2.0_XP (December 2003), the USB demo supports SUSPEND mode. The suspend mode handling mechanism uses the following code for service code, found in the USB interrupt service routine, and requires the SUSPND interrupt to have been enabled.

```c
if (SUSPND)                         // Handle suspend interrupt
{                                    // keep USB logic functional
    SUSPND = 0;                    // set global flag
    GoOnSuspend = 1;               // disable INT
    UIEN &= ~uSUSPNDIE;            // disable INT
    TF2 = 1;                       // Activate execution of the rest of USB ISR
}
```

If USBSuspend is detected (a constant idle state on the USB lines for more than 3ms), then the SUSPND flag is set. Setting this bit stops the clock to USB, and causes the USB module to enter Suspend mode. Because this event can be serviced at some other place in the code, with the auxiliary bit (GoOnSuspend) set and the SUSPND flag cleared, another mechanism to reach the still functional USB part is used (albeit not in the next interrupt after this source has been enabled). Afterwards, when the application has more time, the physical suspension of the device is possible.

The GoOnSuspend event is checked in the subsequent auxiliary USB service routine when it is possible:

```c
// The following 3 lines can be deleted if you don’t plan to use Suspend
if (GoOnSuspend)                   //Service USB INT Suspend
{                                  
    OnUSBSuspend();                
}
```

The default handler, for this event, performs the physical suspend task, and is located in main.c, and is shown in APPENDIX J.

This default handler, in the USB demonstration program:
- writes information to the display, announcing the start of the Suspend mode
- disables all interrupts, except USB
- enables the USB Resume interrupt source
- enters the Idle mode of the µPSD (PCON != 1).

In this state, no instructions are executed, and the µPSD waits for any enabled interrupt. When a USB interrupt is detected, the µPSD continues execution, by restoration of the USB part, by clearing the SUSPND and RESUMF bits, restoring and enabling other previous interrupts, and clearing the display.
USB Disconnect on Demand Feature

Since version 2.0_XP (December 2003), the USB demonstration program also supports the USB Disconnect-on-Demand feature. This feature was especially developed for the µPSD32xxA (with USB), and allows the disconnecting, or reconnecting, of the USB device without physically unplugging it. This USB Disconnect-on-Demand feature is very useful in many applications, and also greatly helps while debugging your USB applications. There is no need to unplug the USB connector or cables. It quickly saves a great deal of development time, and avoids having to make any special, major modifications to your application. For more information, please see AN1815.

This feature is disabled by default, though. If you wish to use it, it needs to be enabled, in main.c, by removing the comment characters in front of the following #define line:

```
#define USBDisconnetOnDemand
```

Note: Be sure that your hardware is capable of supporting it. In the case of the DK3200 kit and the USB demonstration program, do not forgot to insert jumpers PB0-PB3 (located at the left lower part of the kit).

The responsible part of code for the USB reconnect is as shown in APPENDIX K..

The test and activation can be found in the main() routine (the feature is invoked by pressing Switch One on the DK3200 kit):

The following initialization causes the PB0,1 pins to be set as inputs:

```c
// The following two lines can be deleted if no USB Disconnect feature is implemented
00812   UPSD_xreg.DIRECTION_B |= 0x03;          // set PSD-PB0,1 DDR high
00813   UPSD_xreg.DATAOUT_B = 0x03;             // disable LED2, low level active /01/02
```

and the following lines achieve the testing and invocation:

```c
if ((UPSD_xreg.DATAIN_B & 0x04)==0)                  // check Switch ONE
    {                                                  // Disconnect USB when pressed
        ReConnectUSB();
    }
```


IMPORTANT RULES WHEN STARTING TO DESIGN USB APPLICATIONS WITH µPSD32XXA

Differences between USB 1.1 Hosts and Some USB 2.0 High-speed Hosts

Figure 5. shows the maximum traffic on the USB with USB1.1, like that for the µPSD32xxA, or some USB2.0 hosts. All the requests are transmitted only at begin of every 1ms frame. The next request can be received only at the begin of the next frame, and after the last completion or acknowledge.

Figure 5. Traffic on USB (µPSD32xxA connected to USB1.1 Host, or to an Intel 82801DBx Type USB 2.0 Host)

Figure 6. shows nearly maximum traffic on the USB, with a USB 2.0 High-speed host (such as the NEC720100). The requests are not transmitted in 1ms frames, but on demand, as soon as the software driver is ready, and the host waits and generates requests, until the last request is completed. The result is that, with some types of USB host, the throughput of the bus is as much as doubled (more than 10KByte/second).
It follows that the time constraints on the USB device are much tighter for High-speed USB 2.0 hosts, as opposed to those for old USB 1.1 hosts (where the USB device has plenty of time to complete the previous USB interrupt service routine, and all the tasks connected with the previous request, and the internal USB state machine inside the USB device is not influenced by changes or communication on USB bus).

USB Interrupt must have the Highest Priority Interrupt Level

It is strongly recommended to assign the highest priority level only to the USB, to minimize the latency to USB interrupt requests. In particular, the USB block of the μPSD32xxA has a small FIFO and no overflow flags, so all the USB servicing routines must be executed as soon as possible, for proper message handling.

In the μPSD32xxA, each interrupt source can be assigned one of two priority levels. Interrupt priority levels are defined by the appropriate bit in the interrupt priority special function register, IP and IPA:

- 0 = low priority
- 1 = high priority

A low priority interrupt may be interrupted by a high priority one. A high priority interrupt routine cannot be interrupted by any other. If two interrupts of different priority occur simultaneously, the high priority level request is serviced. If requests of the same priority are received simultaneously, an internal polling sequence determines which request is serviced. Thus, within each priority level, there is a second priority structure determined by the polling sequence.

The optimal arrangement can be achieved by the following initialization:

```
IP  = 0; // USB must have the highest priority!
IPA = 1; // USB must have the highest priority!
```
### Table 21. Description of the IP Bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Symbol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>6</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>5</td>
<td>PT2</td>
<td>Timer 2 Interrupt priority level</td>
</tr>
<tr>
<td>4</td>
<td>PS</td>
<td>USART Interrupt priority level</td>
</tr>
<tr>
<td>3</td>
<td>PT1</td>
<td>Timer 1 Interrupt priority level</td>
</tr>
<tr>
<td>2</td>
<td>PX1</td>
<td>External Interrupt (Int1) priority level</td>
</tr>
<tr>
<td>1</td>
<td>PT0</td>
<td>Timer 0 Interrupt priority level</td>
</tr>
<tr>
<td>0</td>
<td>PX0</td>
<td>External Interrupt (Int0) priority level</td>
</tr>
</tbody>
</table>

### Table 22. Description of the IPA Bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Symbol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>PDDC</td>
<td>DDC Interrupt priority level</td>
</tr>
<tr>
<td>6</td>
<td>—</td>
<td>Not used</td>
</tr>
<tr>
<td>5</td>
<td>—</td>
<td>Not used</td>
</tr>
<tr>
<td>4</td>
<td>PS2</td>
<td>2nd USART Interrupt priority level</td>
</tr>
<tr>
<td>3</td>
<td>—</td>
<td>Not used</td>
</tr>
<tr>
<td>2</td>
<td>—</td>
<td>Not used</td>
</tr>
<tr>
<td>1</td>
<td>PI2C</td>
<td>I²C Interrupt priority level</td>
</tr>
<tr>
<td>0</td>
<td>PUSB</td>
<td>USB Interrupt priority level</td>
</tr>
</tbody>
</table>
Length of USB Interrupt Service Routine

Based on performed tests and measurements, the worst-case time spent in the USB ISR, over and above the time spent in its own USB drivers, and thus normally available for additional user service routines, is as listed in Table 23., and plotted in Figure 7.. Based on these results, the only possible values of oscillator frequency are 24 and 36MHz.

**Table 23. Oscillator Frequencies Suitable for Applications with USB**

<table>
<thead>
<tr>
<th>Prescaler</th>
<th>Fosc (MHz)</th>
<th>Available Time (µs)</th>
<th>Number of 1T Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>12.0</td>
<td>Not reliable</td>
<td>Not reliable</td>
</tr>
<tr>
<td>1</td>
<td>24.0</td>
<td>140</td>
<td>280</td>
</tr>
<tr>
<td>2</td>
<td>36.0</td>
<td>264</td>
<td>792</td>
</tr>
</tbody>
</table>

**Figure 7. USB Demonstration Program USB Drivers Immunity versus Length of USB ISR (with respect to Oscillator Frequency)**

Note: USB 2.0, NEC720100A

**Difference between Relative and Absolute Register Usage**

It is strongly recommended to use relative registers in USB applications. It can save a lot of machine cycles, which can be used for real USB tasks. But some steps must be followed.

The difference occurs at the beginning and end of the USB ISR. Using absolute registers (for example using the `#pragma AREGS` directive) in the following code would be inserted:
static void UsbIsr() interrupt 6
{
    PUSH ACC
    PUSH B
    PUSH DPH
    PUSH DPL
    PUSH PSW
    MOV PSW,#00H
    PUSH AR0
    PUSH AR1
    PUSH AR2
    PUSH AR3
    PUSH AR4
    PUSH AR5
    PUSH AR6
    PUSH AR7

    Followed by the USB ISR code, and then the following code at the end:

    POP AR7
    POP AR6
    POP AR5
    POP AR4
    POP AR3
    POP AR2
    POP AR1
    POP AR0
    POP PSW
    POP DPL
    POP DPH
    POP B
    POP ACC
    RETI
}

Using relative registers, and invoking assembler directives (for example using the #pragma NOAREGS directive), we get:

static void UsbIsr() interrupt 6 using 3
{
    PUSH ACC
    PUSH B
    PUSH DPH
    PUSH DPL
    PUSH PSW
    MOV PSW,#18H

    followed by the USB ISR code, and then the following code at the end:

    POP PSW
    POP DPL
    POP DPH
    POP B
    POP ACC
    RETI
}
Evidently, by using relative registers, we can save eight PUSH and eight POP instructions, each taking two machine cycles to execute – a total saving of 32 machine instructions. Assuming a 36MHz oscillator frequency, this represents about 10.7µs, which is a long time for the µPSD32xxA. The main drawback is that we must keep in mind, when programming USB applications, that all the parts of the entire program must use different register banks, to avoid rewriting the register contents.

Since the 8051 supports four register banks, and has two levels of interrupts, any part of µPSD32xxA code can be interrupted to level-1, and this interrupt service routine can subsequently be interrupted to level-2. So we suggest using the structure, shown in Table 24., for register bank allocation.

Table 24. Suggested Assignment of Register Banks

<table>
<thead>
<tr>
<th>Register Bank 3</th>
<th>Interrupt level-2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Main USB Interrupt Service Routine only</td>
</tr>
<tr>
<td>Register Bank 2</td>
<td>Interrupt level-1</td>
</tr>
<tr>
<td></td>
<td>Remaining USB tasks, other interrupts like Timer0, 1, 2, serial line and so forth.</td>
</tr>
<tr>
<td>Register Bank 1</td>
<td>Reserved for alternative registers in main program</td>
</tr>
<tr>
<td>Register Bank 0</td>
<td>Main program, all routines (starting from main()), default state</td>
</tr>
</tbody>
</table>

USB may Generate Spurious Interrupts when the Device is Plugged in

The main worry is that the CPU load is very high during the time till the USB RESET and configuration state. The problem can be exacerbated when the USB is not the only source of interrupts to the µPSD, and during this period some of these other interrupts are missed.

Ideally, when USB driver is in the Reset state, or other non-operational state, and both of the USB lines (D– and D+) has low level (or near the same) for longer time, the internal logic must logically disable any other processing of data originating from the analog differential stage of the USB. This is because both input signals have a near zero differential voltage, and the differential stage is prone to amplifying noise, thereby generating random data bits, and misinterpreting them as communication.

The main concern is that CPU load increases dramatically for a short time (about 40ms) when the USB cable is plugged.

µPSD USB IP sometimes Triggers an Interrupt but no Bit of UISTA is Set

The µPSD32xxA can enter the USB interrupt service routine, but with none of the UISTA bits being set to identify the source of the interrupt. This situation can occur when the USB device or cable is unplugged.

µPSD USB IP Generates Interrupts with an Undocumented Flag

The µPSD32xxA can enter the USB interrupt service routine, with the reserved flag, bit-6, of UISTA set. This situation, too, can occur when the USB device or cable is unplugged, nearly always after 5 to 20µs have elapsed, when both signals (D+ and D–) go down to logic zero (BUS reset).

This state is not stable, and persists only for a short time. If this bit is set, the UISTA register contains 0xFE in most cases, so it can be misinterpreted as a request to service nearly all the interrupt sources.

USB State Machine does not Clear the STALL0 Bit when a SETUP Packet is Received

The internal USB state machine does not clear STALL0 flag, previously set to STALL Endpoint0. This causes problems with USB 2.0 and Win98.

The USB demo 2.0_XP, and later versions, fix this problem by a software clear of the STALL0 bit when it is set and a SETUP packet has been received.
An example of the solution would be as follows:

```c
if (USTA & uSETUP)                        // If it’s a SETUP packet ...
{
    if (UCON0 & uSTALL0)                   // previously STALLed
    {
        #pragma asm
        anl    UCON0,#255-uSTALL0        ;clear STALL0 bit when it hangs
        #pragma endasm
    }
}
```

**Remark:** This STALL0 bit problem is important only during the first enumeration or setup phase. It is important to be mentioned, that during regular USB usage (normal communication, usage of the device, in usual applications), there is no reason to STALL EP0, therefore this problem is not visible and has no influence on the USB communication and data exchange. In some (real-time) operating systems, all sufficient system resources must be available to USB only during the first phase to assure fast response and clearing of STALL0 bit as soon as possible to ensure the subsequent correct functioning of the USB device. Thanks to this, the STALL0 bit problem is not as significant as it might at first appear.

**The Number of Data Bytes Received in a Data Packet can be Greater than Eight**

It is possible that the content of the USTA register (bits RP0SIZ0,1,2,3) can indicate a longer packet length than the maximum value of 8 that is specified in the USB standard. A typical number is 15, and tends to occur when a USB device is connected or disconnected to or from USB host, or when there is some communication error caused by electrical disturbance. No problems have been observed during the subsequent USB operation and communication.

If the buffers are too short, and insufficient tests and conditions are made in the software, buffer overflow errors can occur, with the overwriting of some variables and incorrect system functionality, or even crash. The problem has been fixed since version 2.0_XP (January 2004).

**C Source and Assembler Output can have Different Functionality**

All the previous versions of the USB demonstration program were written in the C language, and sometimes highly optimized. Unfortunately optimization is not always appropriate, and the C language compilers and linkers have absolutely no idea about µPSD hardware functionality. Therefore some parts are wrongly compiled, and the result causes problems.

Unfortunately, µPSD has all EP0 basic service and control bits in one register, used bidirectionally. For example, in routine TransmitBufferEP0() in uPSD_USB.c the following service might have been written:

```c
UCON0 = ((UCON0 ^ uTSEQ0) & uTSEQ0) | (UCON0 & uRX0E) | nBytes;
```

It looks like a perfectly conventional line of C, that might be optimized very nicely. But the Keil compiler (and also others) converts the line as:

```c
MOV     A,UCON0
XRL     A,#080H
ANL     A,#080H
MOV     R7,A
MOV     A,UCON0
ANL     A,#010H
CRL     A,R7
CRL     A,nBytes
MOV     UCON0,A
```

The problem is that the UCON0 register is not a mathematical constant, as the optimizer might assume, but can change its state (in particular, the STALL0 bit) between the first and the last occurrence.
One correct solution is:

```
XRL UCON0, #0x80 ; toggle TSEQ bit
ANL UCON0, # (uTSEQ0 + uRX0E) ; mask
MOV A, nBytes
ORL UCON0, A
```

Besides the fact that this solution is shorter, all the operational steps are executed independently in one instruction. The µPSD hardware and software can change the contents of the register with the correct result.

All the versions of the USB demonstration program, since version 2.0_RP (December 2003), uses the new construction directly written in the assembler language. The final code will have its proper content without dependence on compiler settings, etc.

*Note: Use any other constructions and C language only once you are sure about the real assembler output.*

**Question: So How Must I Write My Drivers?**

One of the most important points is that all program steps that influence the USB part must be done in one step, by *only* one instruction if possible. Keep in mind that the hardware can change some of the “bidirectional” bits in USB register between the steps when a register is read, updated and written back. The last step overwrites the new state of the register changed by the hardware.

In assembler, only use MOV, ORL, ANL or XRL instructions. In the C language, use constructions like:

```
UCON0 &= uRX0;
UCON0 |= uSTALL0;
... etc.
```

Do not use more than one instruction (step) on one program line, because the compiler might optimize it with uncertain results.

*Note: Use any other constructions and C language only once you are sure about the real assembler output.*

**Question: Must I Enable and Service End-of-Packet Interrupt Requests?**

No, End of Packet (EOP) interrupts can be disabled. EOP interrupts are used only for some very special purposes. They can be generated as frequently as every tens of µs, result in increasing the CPU load.

*Note: Do not use USB drivers or a USB demonstration program fully based on version 1.0.7E. This version causes a very high CPU load because the USB interrupt is serviced on every EOP interrupt request even where the previous invoking packet is not significant for processing by the software driver.*

**Question: Is the Length of USB ISR Limited?**

Yes, the length of USB interrupt service is limited. The limits are especially given by the USB specifications on drivers used in hosts.

**Limits Set by the Specification:** Generally, the USB standards talk about very long time limits. Table 25. gives some examples.
**Known limits:** If a device does not respond in less than 4ms from the time that a SETUP packet is sent, which might be a real constraint during the configuration stage, Windows XP will repeat the command, but Windows98/SE will not. When a device does not respond, it is typically disconnected and marked as not functional. Also, in some specific states, it can hang (as observed on Windows98 and Windows XP, and the subsequent disconnection of the problematic device.

**Conclusion:** Write short interrupt service routines, no more than 450µs in duration (about 1350 short instructions, which is more than sufficient for most of applications). If your USB ISR can be longer, only set some your user defined flag in the USB ISR and execute the respective long service at your main() routine or other (like Timer2ISR() in the last USB demonstration Program, version 2.0_XP), executed outside the USB ISR.

**Question: My Device with µPSD is not Properly Detected by Windows**

In the last stage of configuration of the device, the GET_DESCRIPTOR command is sent after the GET_INTERFACE command. Unfortunately, because of a problem with the STALL0-bit, the USB part of the µPSD does not respond. Windows98/SE, and some other versions of Windows, have USB drivers waiting for 4ms for the completion of the request, after which they “disconnect” the device and mark it as “Problematic HID device”. Windows XP Professional version 2002, and some others send the request again after the initial 4ms. This second request is then properly processed by the USB part of µPSD, and so the device stays fully functional.

This problem was fixed since version 2.0_XP (January 2004). This version is also functional on Windows 98, and with a USB 2.0 NEC type host.

**Figure 8.** shows an example how the USB interrupts are generated. Please subtract about 6 µs (delay of PUSH and initialization part of USB ISR).

**Figure 8. USB ISR Response Time**
Problems with Windows 98 and Multiple-function USB Devices

This has already been treated at the Microsoft web site (Q249635).

**Symptoms.** Multiple function Universal Serial Bus (USB) devices that support multiple packet sizes send $DATA_0$ and $DATA_1$ packets using the largest supported data field size rather than the smallest required. Each supported function of the USB device should send $DATA_0$ and $DATA_1$ packets using the smallest required data field size. The overall performance of a multiple-function device is decreased because more data is sent over the USB device.

**Cause.** This behavior occurs because the HIDclass driver in Windows 98 Second Edition always uses the largest data field the USB device supports to transmit packets. This behavior causes the USB device to transmit unneeded data during an IN or OUT transaction.

**Conclusion.** A supported fix is now available from Microsoft. Apply it only to computers that are experiencing this specific problem. It is only intended to correct the problem that is described in this article, and for the following edition of Windows 98 Second Edition:

Microsoft Windows 98 Second Edition  
Last Reviewed: 5/8/2002 (1.0)  
Keywords: kbug KB249635

To resolve the problem, contact Microsoft Product Support Services to obtain the fix. For a complete list of Microsoft Product Support Services ‘phone numbers and information about support costs, visit the Microsoft web site.

**NOTE:** In special cases, charges that are ordinarily incurred for support calls may be canceled if a Microsoft Support Professional determines that a specific update will resolve your problem. The usual support costs will apply to additional support questions and issues that do not qualify for the specific update in question.

The English-language version of this fix should have the following file attributes, or later:

<table>
<thead>
<tr>
<th>Date</th>
<th>Time</th>
<th>Version</th>
<th>Size</th>
<th>File name</th>
<th>Platform</th>
</tr>
</thead>
</table>

**Status.** Microsoft has confirmed that this is a problem in Windows 98 Second Edition.
**More Information.** The overall lengths of the \textit{DATA0} and \textit{DATA1} packets between each function on a multiple-function device are equal. The data field of these packets is the longest length supported by the functions. For example, with a multifunction device that supports data retrieval from a piece of test-and-measurement equipment and keyboard input, the test-and-measurement function requires a 512-byte buffer, but the keyboard requires only 16 bytes of data. The following examples show the resulting DATA packets that may appear: Incoming keyboard data from the USB keyboard function.

```
[SYNC]
[DATA (16 bytes of keyboard data) (496 bytes zero-filled) ]
[EOP]
```

Incoming test-and-measurement data from the USB test-and-measurement function.

```
[SYNC]
[DATA (512 bytes test/measurement data) ]
[EOP]
```

\textit{NOTE: To see the problem, a USB analyzer is required.}

This fix is recommended when you are using multiple-function USB devices. This fix does not affect single-purpose USB devices. For additional information about Windows 98 and Windows 98 Second Edition hotfixes, go to the Microsoft Knowledge Base, and request article 206071, \textit{General Information About Windows 98 and Windows 98 Second Edition Hotfixes.}
CONCLUSION
This application note discusses important issues in USB design using μPSD32xxA, and describes all the currently known issues. It shows how to design USB applications with μPSD32xxA, and which steps and rules must be met to avoid possible issues.

The USB block in the μPSD32xxA is very simple, and is typically used in keyboards, mouse and other simple devices. The latest version of these drivers reduces the possible issues to a minimum, but many conditions must be applied when programming the USB application.

In case there is a possibility of changing the driver at the host, or to use a non-standard driver, it is recommended to use the latest version of the USB drivers in the μPSD32xxA, but to disable:

- all of the new features
  - including Enhanced Host Controller specification features, EHCI
- other modifications, especially those aimed at reaching higher throughput of USB bus
  - thereby still leaving the system capable of reaching the same traffic on the USB as with old USB1.1 controllers connected to low-speed devices

REFERENCES
10. μPSD323x, datasheet, STMicroelectronics.
11. μPSD325x, datasheet, STMicroelectronics.
12. USB Device Disconnect-On-Demand with μPSD32xx, AN1815, STMicroelectronics.
14. Designing the Oscillator for Use with μPSD, AN1843, STMicroelectronics.
APPENDIX A. USBISR

static void UsbIsr() interrupt 6 using 2
/*****************************************************************************/
Function   : static void UsbIsr()
Parameters : none
Description: USB interrupt service routine.
Note:        Do not modify this routine !!!
****************************************************************************/
{
data uchar cb;
data uchar i;

if (RXD0F) // If data received on EP0 OUT ...
{
    // do not change this SETUP packet processing part up to else !!!
    if (USTA & uSETUP) // If it’s a SETUP packet ...
    {
        if (UCON0 & uSTALL0)
        {
            #pragma asm
            anl    UCON0,#255-uSTALL0       ;clear STALL0 bit
            #pragma endasm
        }
        if  (ReadSetupPacket()) // Read the SETUP packet
        {
            if (!HandleReport()) // Test and handle HID type packets
            {
                // If this is not a HID report ... pass it on to the basic SETUP packet handler
                OnSetupPacket();
            }
        }
    }          // No Setup packet with DATA length = 8, STALLed automatically in ReadSetupPacket()
    RXD0F = 0;                      // Clear interrupt flag so next packet can come in now
}
else
    /*======== No SETUP packet, normal data packets ========*/
{
    // If in the middle of receiving a report ...
    if ((USTA & 0x0F) && (rcvIndex < OUTPUT_REPORT_SIZE))
    {
        cb = (USTA & 0x0F); // Read the next segment
        if (cb > EP0_SIZE)
        {
            cb = EP0_SIZE;
        }
        for (i = 0; i < cb; i++)
        {
            rcvReport.u.buffer[rcvIndex + i] = UDR0; // read received data
        }
        #pragma asm
        xrl    USTA,#uRSEQ                        // Toggle data toggle bit
        #pragma endasm
        // Handle report as it comes in and/or when all done
        OnReportSegmentReceived(cb);
    }
if ((rcvIndex += cb) >= OUTPUT_REPORT_SIZE)
{    // send a zero length packet
    OnReportReceived();
    //mask
    anl UCON0,#uTSEQ0+uRX0E                 ;mask
    orl UCON0,#uTX0E                        ;enable transmit
    #pragma endasm

    RXD0F = 0;            // Clear interrupt flag so next packet can come in now
} else
{
    // Got 0 length ACK data packet; expecting a SETUP packet now.
    USTA ^= uRSEQ;                // Toggle data toggle bit
    RXD0F = 0;                    // Clear interrupt flag so next packet can come in now
}
}

if (TXD0F)
{    // Clear the interrupt flag
    TXD0F = 0;
    // Do not change this part up to call of BaseEp0Handler !!!
    // If in the middle of transmitting a report ...
    if (txIndex < FEATURE_REPORT_SIZE)
    {
        // Transmit the next segment of outgoing report
        cb = min(FEATURE_REPORT_SIZE - txIndex, EP0_SIZE);
        TransmitDataEP0(txReport.u.buffer + txIndex, cb);
        if ((txIndex += cb) >= FEATURE_REPORT_SIZE)
        {
            OnReportTransmitted();
        } else
        {
            // Prepare the next segment while that last one is going out
            PrepareTransmitSegment(txIndex);
        }
    } else
    {
        // This part can changed (or BaseEp0Handler directly)
        BaseEp0TxHandler();             // Handle standard control requests
    }
}

if (SUSPND)                                // Handle suspend interrupt
{    // keep USB logic functional
    SUSPND = 0;
    GoOnSuspend = 1;                     // set global flag
    UIEN &= ~uSUSPNDIE;                  // disable INT
    TF2 = 1;                             // Activate execution of the rest of USB ISR
}

if (TXD1F)                                 // If data successfully transmitted on EP1 ...
{    // disable EP1,2 INT
    UIEN &= ~uTXD1IE;
    TF2 = 1;                             // Activate execution of the rest of USB ISR
}
if (RSTF)                                  // Handle USB bus reset, it must be at the
end of USB ISR
{
    USB_ISR_FLAGS = 0;
    OnUSBReset();                            // resets all the flags automatically
    TF2 = 1;                                 // Activate execution of the rest of USB ISR
}
g_debugUSB_INT_CNT++;// Increment USB ISR counter

APPENDIX B. ONTRANSMITEP1

static void OnTransmitEP1()
{
    data unsigned char i,nBytes;               // Num bytes of LCD data
    // Store current index into LCD buffer in first byte of tx buffer
    if (bufIndex >= LCD_BUFFER_SIZE - 1)       // one zero only, save packet size
        bufIndex = 0;                 // Wrap around to start of LCD buffer for next transmission
    nBytes = LCD_BUFFER_SIZE - bufIndex - 1;
    if (nBytes>7)                      // max. 1 length info byte + 7 characters
        nBytes = 7;
    txBuf[0] = bufIndex;                        // save position at the first byte
    i = 1;
    while (i<=nBytes)
    {
        txBuf[i++] = LCD_buffer[bufIndex++];    // fill buffer
    }
    TransmitDataEPx(1, txBuf, nBytes+1);        // Transmit input report to host
}
APPENDIX C. ONSETUPPACKET

Void OnSetupPacket()
/*****************************************************************************/
Function : void OnSetupPacket()
Parameters : none
Description: Basic handler for SETUP packets received on EP0.
****************************************************************************/
{
pTransmitBufferEP0 = NULL;
bytesToTransmitEP0 = 0;
// If it’s a standard request...
if ((setupPacket.bmRequestType & 0x60) == 0)
{
    switch (setupPacket.bRequest) // Handle the request
    {
        case GET_STATUS:        OnGetStatus();        return;
        case CLEAR_FEATURE:     OnClearFeature();     return;
        case SET_FEATURE:       OnSetFeature();       return;
        case SET_ADDRESS:       OnSetAddress();       return;
        case GET_DESCRIPTOR:    OnGetDescriptor();    return;
        case SET_DESCRIPTOR:    OnSetDescriptor();    return;
        case GET_CONFIGURATION: OnGetConfiguration(); return;
        case SET_CONFIGURATION: OnSetConfiguration(); return;
        case GET_INTERFACE:     OnGetInterface();     return;
        case SET_INTERFACE:     OnSetInterface();     return;
        default:
            break;
    }
}
STALL_EP0(); // It’s not a request we handle, so stall endpoint
}
APPENDIX D. ONGETDESCRIPTOR

static void OnGetDescriptor()
.vaadinetti
Function : static void OnGetDescriptor()
Parameters : none
Description: Handler for GET_DESCRIPTOR() control requests
******************************************************************************/
{
    unsigned int bytesRequested;
    switch (setupPacket.wValue.hi)
    {
    case DT_DEVICE:
        pTransmitBufferEP0 = (uchar*) &deviceDesc;
        bytesReadToTransmitEP0 = sizeof(deviceDesc);
        break;

    case DT_CONFIGURATION:
        pTransmitBufferEP0 = (uchar*) &configDesc;
        bytesReadToTransmitEP0 = configDesc.wTotalLength.lo;
        break;

    case DT_STRING:
        pTransmitBufferEP0 = (uchar*)
            stringDescTable[setupPacket.wValue.lo<4?setupPacket.wValue.lo:4];
        bytesReadToTransmitEP0 = *pTransmitBufferEP0;          // choose requested string
        break;

    #if HID_DEVICE
    case DT_HID_CLASS:
        pTransmitBufferEP0 = (uchar*) &hidClassDesc;
        bytesReadToTransmitEP0 = hidClassDescSize;
        break;

    case DT_HID_REPORT:
        pTransmitBufferEP0 = (uchar*) &reportDesc;
        bytesReadToTransmitEP0 = reportDescSize;
        break;

    case DT_HID_PHYSICALD:
        pTransmitBufferEP0 = (uchar*) &PhysicalReportDesc;
        bytesReadToTransmitEP0 = PhysicalReportDescSize;
        break;
    #endif

    default:
        STALL_EP0();                          // Unrecognized descriptor, so stall EP0
        return;
    }

    bytesReadRequested = (setupPacket.wLength.hi << 8) | setupPacket.wLength.lo;
shortTransfer = (bytesToTransmitEP0 < bytesRequested);
if (bytesToTransmitEP0 > bytesRequested)
{
    bytesToTransmitEP0 = bytesRequested;
}
UCON0 &= ~uTSEQ0;           // TransmitDataEP0 will toggle sequence bit to DATA1
TransmitBufferEP0();

APPENDIX E. REPORTDESC

#define REPORT_DESC_LEN sizeof(reportDesc)

const uchar code reportDesc[] = {
    0x06, 0xA0, 0xFF,              // Usage page (vendor defined)
    0x09, 0xA5,                    // Usage (vendor defined)
    0xA1, 0x01,                    // Collection (application)
    0x09, 0xA6,                    // Usage (vendor defined)

    // Feature report
    0x09, 0xA5,                    // Usage (vendor defined)
    0x15, 0x80,                    // Logical min (-127)
    0x25, 0x7F,                    // Logical max (128)
    0x75, 0x08,                    // Report size (8 bits)
    0x95, 0x40,                    // Report count (64 bytes)
    0xB1, 0x02,                    // Feature (data, variable, absolute)

    // Input report
    0x09, 0xA7,                    // Usage (vendor defined)
    0x15, 0x80,                    // Logical min (-127)
    0x25, 0x7F,                    // Logical max (128)
    0x75, 0x08,                    // Report size (8 bits)
    0x95, 0x08,                    // Report count (8 bytes)
    0x81, 0x02,                    // Input (data, variable, absolute)

    // Output report
    0x09, 0xA9,                    // Usage (vendor defined)
    0x15, 0x80,                    // Logical min (-127)
    0x25, 0x7F,                    // Logical max (128)
    0x75, 0x08,                    // Report size (8 bits)
    0x95, 0x40,                    // Report count (64 bytes)
    0x91, 0x02,                    // Output (data, variable, absolute)
    0xC0                              // End Collection (Application)
};

const uchar code reportDescSize = REPORT_DESC_LEN;
APPENDIX F. DEVICEDESC

const device_descriptor code deviceDesc = {
    sizeof(device_descriptor),                 // Size of this Descriptor in Bytes
    DT_DEVICE,                                 // Descriptor Type (=1)
    {0x10, 0x01},                              // USB Spec Release Number in BCD = 1.10 0x10,0x01
    0,                                         // Device Class Code (none)
    0,                                         // Device Subclass Code (none)
    0,                                         // Device Protocol Code (none)
    8,                                         // Maximum Packet Size for EP0
    {0x83, 0x04},                              // Vendor ID
    {0x00, 0x00},                              // Product ID
    {0x00, 0x01},                              // Device Release Number in BCD
    1,                                         // Index of String Desc for Manufacturer
    2,                                         // Index of String Desc for Product
    3,                                         // Index of String Desc for SerNo
    1,                                         // Number of possible Configurations
};

// Copy of HID class descriptor embedded in configDesc below
const uchar code hidClassDesc[] = {
    9,                                         // Descriptor length
    0x21,                                      // Descriptor type (HID)
    0, 1,                                      // HID release (1.0)
    0,                                         // Country code (none)
    1,                                         // Number of HID class descriptors to follow
    0x22,                                      // Report descriptor type (HID)
    REPORT_DESC_LEN, 0                         // Byte length of report descriptor
};

const uchar code hidClassDescSize = sizeof(hidClassDesc);

const uchar code configDesc[] = {
    9,                                         // Configuration descriptor length
    2,                                         // Descriptor type (configuration)
    34, 0,                                     // Total length of this descriptor
    1,                                         // Number of interfaces
    1,                                         // Configuration value
    0,                                         // Index of string descriptor (none)
    0x40,                                      // Self powered, no remote wakeup
    25,                                        // 500 mA max power consumption, 2mA steps
/*===== Interface =====*/
9,  // Descriptor length
4,  // Descriptor type (interface)
0,  // Number of interface
0,  // Alternate setting
1,  // Number of endpoints (except EP0)

// if you would like to disable EP1, easily write 0 (and 1 back for EP1/display copy refresh enable)
3,  // Class code (HID)
0,  // Subclass (none)
0,  // Protocol (none)
0,  // Index of string descriptor (none)

/*===== HID class descriptor (copy of hidClassDesc) =====*/
9,  // Descriptor length
0x21,  // Descriptor type (HID)
1, 1,  // HID release (1.1)
0,  // Country code (none)
1,  // Number of HID class descriptors to follow
0x22,  // Report descriptor type (HID)
REPORT_DESC_LEN, 0,  // Byte length of report descriptor

/*===== Endpoint descriptor =====*/
7,  // Descriptor length (7 bytes)
5,  // Descriptor type (endpoint)
0x81,  // Address (IN1)
3,  // Attributes (interrupt)
8, 0,  // Maximum packet size (8 bytes)
100  // Polling interval of EP1 (in msec)
);
APPENDIX G. LANGUAGE, MANUFACTURE AND PRODUCT STRINGS

// Language IDs
const uchar code string0Desc[] =
{
    &string1Desc-&string0Desc, DT_STRING,       // Size, Type
    0x09, 0x04                                 // LangID Codes
};

// Manufacturer String
const uchar code string1Desc[] =
{
    &string2Desc-&string1Desc, DT_STRING,      // Size, Type
    'S', 0,                                    // Unicode String
    'T', 0,                                    // Unicode String
    'M', 0,                                    // Unicode String
    'i', 0,                                    // Unicode String
    'c', 0,                                    // Unicode String
    't', 0,                                    // Unicode String
    'r', 0,                                    // Unicode String
    'o', 0,                                    // Unicode String
    'e', 0,                                    // Unicode String
    'l', 0,                                    // Unicode String
    'e', 0,                                    // Unicode String
    'c', 0,                                    // Unicode String
    't', 0,                                    // Unicode String
    'e', 0,                                    // Unicode String
    'l', 0,                                    // Unicode String
    'e', 0,                                    // Unicode String
    'c', 0,                                    // Unicode String
    't', 0
};

// Product String
const uchar code string2Desc[] =
{
    &string3Desc-&string2Desc, DT_STRING,      // Size, Type
    'D', 0,                                    // Unicode String
    'K', 0,                                    // Unicode String
    '3', 0,                                    // Unicode String
    '2', 0,                                    // Unicode String
    '0', 0,                                    // Unicode String
    '0', 0,                                    // Unicode String
    'E', 0,                                    // Unicode String
    'v', 0,                                    // Unicode String
    'a', 0,                                    // Unicode String
    'l', 0,                                    // Unicode String
    'u', 0,                                    // Unicode String
    'a', 0,                                    // Unicode String
    't', 0,                                    // Unicode String
    'i', 0,                                    // Unicode String
    'o', 0,                                    // Unicode String
    'n', 0,                                    // Unicode String
    'B', 0,
};
'c', 0,
'a', 0,
'o', 0,
'd', 0
}
;

// Serial Number String
const uchar code string3Desc[] =
{
    &stringXDesc-&string3Desc, DT_STRING, // Size, Type
    'S', 0,                               // Unicode String
    '/', 0,
    'N', 0,
    ':', 0,
    '0', 0,
    '0', 0,
    '1', 0,
    '/', 0,
    '2', 0,
    '0', 0,
    '0', 0,
    '4', 0,
};
typedef struct
{
    union
    {
        uchar cmd; // CMD_xxx code
        struct
        {
            uchar cmd; // CMD_ERASE
            uchar flash; // PRIMARY_FLASH or SECONDARY_FLASH
            uint16 address; // Any address in any sector
        } erase;
        struct
        {
            uchar cmd; // CMD_READ or CMD_WRITE
            uchar flash; // PRIMARY_FLASH or SECONDARY_FLASH
            uint16 address; // Target xdata address
            uint16 nBytes; // Num bytes to read/write
        } rw;
        struct
        {
            uchar cmd; // CMD_SET_xxx
            uchar page; // Desired page register value
            uchar vm; // Desired VM register value
        } setRegs;
        struct
        {
            uchar cmd; // CMD_GET_STATUS
            uchar currentCmd; // Returns current command being processed
            uchar page; // Returns page register value
            uchar vm; // Returns VM register value
            uchar ret; // Return value from flash routine
            uchar checkSum; // Check sum for write commands
        } status;
        uchar buffer[CMD_SIZE];
    } u;
} MCU_CMD, *PMCU_CMD;
APPENDIX I. ONREPORTRECEIVED

Static void OnReportReceived()

/**********************************************************
 Function : static void OnReportReceived()
 Parameters : none
 Description: Called after all segments of a report have been received.
 **********************************************************/
{
    if (rcvReport.u.cmd)                   // If this is a new command coming in ...
    {
        if (rcvReport.u.cmd == CMD_STATUS)
        {
            // For CMD_GET_STATUS, don’t overwrite current cmd we’re working on
            returnStatus = TRUE;
        }
        else
        {
            // Copy into ‘current command’ global variable
            memcpy(currentCmd.u.buffer, rcvReport.u.buffer, sizeof (rcvReport));
            memset((uchar*)&status, 0, sizeof (status));
            // g_debug1 = currentCmd.u.cmd;
        }
    }
    // Some commands are processed at this point
    switch (rcvReport.u.cmd)
    {
    case CMD_RESET:
        WDKEY=0;                          // watchdog will trigger reset in a second
        currentCmd.u.cmd = 0;
        break;
    case CMD_SET_PAGE:
        UPSD_xreg.PAGE = rcvReport.u.setRegs.page;
        currentCmd.u.cmd = 0;
        break;
    case CMD_SET_VM:
        UPSD_xreg.VM = rcvReport.u.setRegs.vm;
        currentCmd.u.cmd = 0;
        break;
    case CMD_SET_REGS:
        UPSD_xreg.PAGE = rcvReport.u.setRegs.page;
        UPSD_xreg.VM = rcvReport.u.setRegs.vm;
        currentCmd.u.cmd = 0;
        break;
    default:
        // Prepare first segment of any response to go back to host
        PrepareTransmitSegment(0);
        break;
    }
}
APPENDIX J. ONUSBSUSPEND

static void OnUSBSuspend()
/*****************************************************************************/
Function : static void OnUSBSuspend()
Parameters : none
Description: service routine for USB Suspend event
*****************************************************************************/
{

data char bie, biea;
printfLCD("\n* SUSPEND MODE *
");  
printfLCD("\rCPU in Idle mode");
bie = IE;
biea = IEA;
IEA = 1;                                   //disable all INTs except USB
IE  = 128;                                 //disable all INTs except USB
SUSPND = 1;                                //Enter suspend mode for uPSD
GoOnSuspend = 0;                           //clear global Suspend flag
// entering SUSPEND or setting SUSPNDF causes to top the clocks to the USB
// and causes the USB module to enter Suspend Mode.
UIEN  |= uRESUMIE;                        //enable resume INT
PCON  |= 1;                                //Enter Idle mode
      // here the uPSD sleeps and waits for the next INT
SUSPND = 0;                                //clear the flag
IE  = bie;
IEA = biea;
UIEN  &= ~uRESUMIE;                     //disable resume INT
RESUMF = 0;                               //clear the flag
printfLCD("\r \n"); //clear display
printfLCD("\r \n"); //clear display
printfLCD(DEMO_TITLE_STR);
}
APPENDIX K. RECONNECTUSB

static void ReConnectUSB()
{  
    data int w;

    UIEN = 0;                                  // Disable all USB interrupts
    UADR = 0;                                  // Disable USB hardware
    LCDBufTXIndex = 0;                         // LCD disp position for USB

    UPSD_xreg.DATACUT_C &= 0xEF;               // PC4=0
    UPSD_xreg.DRIVE_C  |= 0x10;                // set as OpenDrain
    UPSD_xreg.DIRECTION_C |= 0x10;             // set as output

    printfLCD("USB Disconnected");

    while ((UPSD_xreg.DATAIN_B & 0x04)==0)
    {  
        for (w=0;w<600*50;w++);                // wait 50 ms
    }

    UPSD_xreg.DRIVE_C  &= 0xEF;                // set PC4 as input
    UPSD_xreg.DIRECTION_C &= 0xEF;             // set PC4 as input

    UsbInitialize();
    printfLCD("USB Reconnected.");
}

# APPENDIX L. GLOSSARY

## Table 26. Terms and Abbreviations

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ACK</strong></td>
<td>Handshake packet indicating a positive acknowledgment.</td>
</tr>
<tr>
<td><strong>Buffer</strong></td>
<td>Storage used to compensate for a difference in data rates or time of occurrence of events, when transmitting data from one device to another.</td>
</tr>
<tr>
<td><strong>Bus Enumeration</strong></td>
<td>Detecting and identifying USB devices.</td>
</tr>
<tr>
<td><strong>Class</strong></td>
<td>A USB device is organized into classifications such as HID, audio, or other-based on the device’s features, supported requests, and data protocol.</td>
</tr>
<tr>
<td><strong>Collection</strong></td>
<td>A collection is a meaningful grouping of Input, Output, and Feature items—for example, mouse, keyboard, joystick, and pointer. A pointer Collection contains items for x and y position data and button data. The Collection and End Collection items are used to delineate collections.</td>
</tr>
<tr>
<td><strong>Control</strong></td>
<td>A sink or source of a data field—for example, an LED is a sink or destination for data. A button is an example of a source of data.</td>
</tr>
<tr>
<td><strong>Control pipe</strong></td>
<td>The default pipe used for bi-directional communication of data as well as for device requests. Also message pipe.</td>
</tr>
<tr>
<td><strong>Data phase</strong></td>
<td>Part of a device’s response to a request.</td>
</tr>
<tr>
<td><strong>Descriptor</strong></td>
<td>Information about a USB device is stored in segments of its ROM (read-only memory). These segments are called descriptors.</td>
</tr>
<tr>
<td><strong>Device</strong></td>
<td>A logical or physical entity that performs a function. The actual entity described depends on the context of the reference. At the lowest level, device may refer to a single hardware component, as in a memory device. At a higher level, it may refer to a collection of hardware components that perform a particular function, such as a USB interface device. At an even higher level, device may refer to the function performed by an entity attached to the USB; for example, a data/FAX modem device. Devices may be physical, electrical, addressable, and logical. When used as a non-specific reference, a USB device is either a hub or a function.</td>
</tr>
<tr>
<td><strong>Device class</strong></td>
<td>A method of organizing common functions and protocols for devices that serve similar functions—for example, communication, audio, display, and so on.</td>
</tr>
<tr>
<td><strong>Device descriptor</strong></td>
<td>Packet of information that describes the device—for example, the vendor, product ID, firmware version, and so on.</td>
</tr>
<tr>
<td><strong>Endpoint descriptor</strong></td>
<td>Standard USB descriptor describing the type and capabilities of a USB communication channel, or pipe.</td>
</tr>
<tr>
<td><strong>Feature control</strong></td>
<td>Feature controls affect the behavior of the device or report the state of the device. Unlike input or output data, feature data is intended for use by device configuration utilities and not applications. For example, the value for the repeat rate of a particular key could be a feature control. HID feature controls are unrelated to features discussed in Chapter 9 of the USB Specification.</td>
</tr>
<tr>
<td><strong>Feature item</strong></td>
<td>Adds data fields to a Feature report.</td>
</tr>
<tr>
<td><strong>Field</strong></td>
<td>A discrete section of data within a report.</td>
</tr>
<tr>
<td><strong>Frame</strong></td>
<td>The smallest unit of time on the Universal Serial Bus (USB); equal to 1 millisecond.</td>
</tr>
<tr>
<td><strong>HID (Human Interface Device)</strong></td>
<td>Acronym specifying either a specific class of devices or the type of device known as Human Interface Devices (HID) or HID class devices—for example, a data glove. In this document, “HID class” is synonymous with a device of type: human interface.</td>
</tr>
<tr>
<td><strong>HID class</strong></td>
<td>The classification of USB devices associated with human interface devices (HID).</td>
</tr>
<tr>
<td><strong>HID class device</strong></td>
<td>A device of type: human interface and classified as such.</td>
</tr>
<tr>
<td><strong>HID descriptor</strong></td>
<td>Information about a USB device is stored in segments of its ROM (read-only memory). These segments are called descriptors.</td>
</tr>
<tr>
<td>--------------------</td>
<td>------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Host</strong></td>
<td>A computer with a USB port, as opposed to a device plugged into it. The host computer system where the USB Host Controller is installed. This includes the host hardware platform (CPU, bus, etc.) and the operating system in use.</td>
</tr>
<tr>
<td><strong>Host Controller</strong></td>
<td>The host's USB interface.</td>
</tr>
<tr>
<td><strong>Hub</strong></td>
<td>A USB device containing one or more USB ports. A USB device that provides additional connections to the USB.</td>
</tr>
<tr>
<td><strong>Interface descriptor</strong></td>
<td>The class field of this descriptor defines this device as a HID class device.</td>
</tr>
<tr>
<td><strong>Interrupt In pipe</strong></td>
<td>The pipe used to transfer unrequested data from the device to the host.</td>
</tr>
<tr>
<td><strong>Interrupt Out pipe</strong></td>
<td>The pipe used to transfer low latency data from the host to the device.</td>
</tr>
<tr>
<td><strong>Interrupt Request (IRQ)</strong></td>
<td>A hardware signal that allows a device to request attention from a host. The host typically invokes an interrupt service routine to handle the condition that caused the request.</td>
</tr>
<tr>
<td><strong>Item</strong></td>
<td>A component of a Report descriptor that represents a piece of information about the device. The first part of an item, called the item tag, identifies the kind of information an item provides. Also, referred to generically as Report items. Included are three categories of items: Main, Global, and Local. Its tag defines each type of item. Also referred to as Main item tag, Global item tag, and Local item tag.</td>
</tr>
<tr>
<td><strong>Item parser</strong></td>
<td>The part of the HID class driver that reads and interprets the items in the Report descriptor.</td>
</tr>
<tr>
<td><strong>Logical units</strong></td>
<td>The value the device returns for Logical Minimum and Logical Maximum.</td>
</tr>
<tr>
<td><strong>LSB</strong></td>
<td>Least Significant Byte</td>
</tr>
<tr>
<td><strong>Main item</strong></td>
<td>An item that adds fields to a report. For example, Input, Output, and Feature items are all data.</td>
</tr>
<tr>
<td><strong>Message pipe</strong></td>
<td>Another name for the Control pipe.</td>
</tr>
<tr>
<td><strong>NAK</strong></td>
<td>The value returned when a request has been sent to the device and the device is not prepared to respond.</td>
</tr>
<tr>
<td><strong>Output item</strong></td>
<td>Adds one or more data fields to an output report. Output controls are a destination for data from applications - for example, LEDs.</td>
</tr>
<tr>
<td><strong>Packets</strong></td>
<td>A USB unit of information: Multiple packets make up a transaction, multiple transactions make up a transfer report.</td>
</tr>
<tr>
<td><strong>Physical Descriptor</strong></td>
<td>Determines which body part is used for a control or collection. Each Physical descriptor consists of the following three fields: Designator, Qualifier and Effort.</td>
</tr>
<tr>
<td><strong>Physical units</strong></td>
<td>The logical value with a unit parameter applied to it.</td>
</tr>
<tr>
<td><strong>Pipes</strong></td>
<td>Pipes are different ways of transmitting data between a driver and a device. There are different types of pipes depending on the type of encoding or requesting that you want to do. For example, all devices have Control pipe by default. The Control pipe is used for message-type data. A device may have one or more Interrupt pipes. An Interrupt In pipe is used for stream-type data from the device and an optional Interrupt Out pipe may be used for low latency data to the device. Other types of pipes include Bulk and Isochronous.</td>
</tr>
<tr>
<td><strong>Protocol</strong></td>
<td>A report structure other than the structure defined by the report descriptor. Protocols are used by keyboards and mouse to insure BIOS support.</td>
</tr>
<tr>
<td><strong>Report</strong></td>
<td>A data structure returned by the device to the host (or vice versa). Some devices may have multiple report structures, each representing only a few items. For example, a keyboard with an integrated pointing device could report key data independently of pointing data on the same endpoint.</td>
</tr>
</tbody>
</table>
**Table 27. Document Revision History**

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>13-Apr-2004</td>
<td>1.0</td>
<td>First Issue</td>
</tr>
</tbody>
</table>
If you have any questions or suggestions concerning the matters raised in this document, please send them to the following electronic mail addresses:

\textit{ask.memory@st.com} \textit{(for general enquiries)}

Please remember to include your name, company, location, telephone number and fax number.