USB08 Universal Serial Bus Evaluation Board
Using the MC68HC908JB8
Designer Reference Manual
USB08 Universal Serial Bus Evaluation Board Using the MC68HC908JB8

By: Dipl.-Ing. Oliver Thamm
MCT Elektronikladen GbR
Hohe Str. 9-13
04107 Leipzig
Germany

Telephone: +49 (0)341 2118354
Fax: +49 (0)341 2118355
Email: mct@elektronikladen.de
Web: http://www.elektronikladen.de/mct
Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters which may be provided in Motorola data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.
List of Sections

Section 1. USB08 Quick Start .......................17

Section 2. Hardware Description ...................27

Section 3. Software Module Descriptions .............43

Section 4. Universal Serial Bus (USB) Interface ......59

Appendix A. Supported Standard Device Requests ..........81

Appendix B. USB08 Descriptors ....................83

Appendix C. Source Code Files ....................89

Appendix D. Bill of Materials and Schematic ..........127

Appendix E. Universal USB Device Driver (USBIO) .......131
Table of Contents

Section 1. USB08 Quick Start
1.1 Contents ......................................................... 17
1.2 Introduction ..................................................... 17
1.3 Required System Configuration ................................. 17
1.4 Connecting the Demo Board to the PC ......................... 18
1.5 Driver Installation ............................................... 19
1.6 Starting the Windows Demo Application ....................... 24

Section 2. Hardware Description
2.1 Contents ......................................................... 27
2.2 Introduction ..................................................... 27
2.3 Technical Data .................................................. 28
2.3.1 MC68HC908JB8 Microcontroller .......................... 28
2.3.2 USB08 Evaluation Board .................................. 29
2.4 Circuit Description .............................................. 30
2.4.1 MCU Core Circuit and USB Interface ..................... 31
2.4.2 Input/Output Functions .................................... 32
2.4.3 Monitor Mode Interface ................................... 33
2.4.4 User RS232 Port .......................................... 35
2.4.5 Power Supply .............................................. 36
2.5 Board Layout .................................................... 36
2.6 Jumpers and Bridges ........................................... 38
Table of Contents

2.7 Connectors ................................................. 40
   2.7.1 Expansion Connector X1 ........................ 40
   2.7.2 Monitor Mode Connector X2 ....................... 40
   2.7.3 User RS232 Connector X3 ......................... 41

2.8 Memory Map .............................................. 41

Section 3. Software Module Descriptions

3.1 Contents .................................................. 43
3.2 Introduction ............................................. 43
3.3 General Structure of the M68HC08 Firmware ............. 44
3.4 How to Build the Compiler Project ....................... 45
3.5 Main Module U08MAIN.C .................................. 48
3.6 Interrupt and Reset Vector Module VECJB8.C .......... 49
3.7 C Startup Module CRTSJB8.S ......................... 50
3.8 Push Button Module U08KEY.C ......................... 50
3.9 LED Control with U08LED.H ......................... 52
3.10 Software ADC Module U08ADC.C ...................... 52
3.11 RS232 Communication Module U08232.C ................ 54
3.12 USB Communication Module U08USB.C .................. 56
3.13 Compiler Specific Adjustments .......................... 57

Section 4. Universal Serial Bus (USB) Interface

4.1 Contents .................................................. 59
4.2 Introduction ............................................. 59
4.3 Characteristics of the USB08 Reference Design .......... 60
4.4 USB Basics ............................................... 62
Table of Contents

4.5 USB Implementation in the Reference Design . . . . . . . . . . . . . . 65
4.5.1 Activation of the USB Module . . . . . . . . . . . . . . . . . . . . . . . . . 65
4.5.2 Endpoint Configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
4.5.3 USB Reset . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.6 Device Management with Endpoint 0 . . . . . . . . . . . . . . . . . . . . . 69
4.6.1 Enumeration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
4.6.2 Assignment of the Device Address . . . . . . . . . . . . . . . . . . . . . . 69
4.6.3 Requesting Descriptors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
4.6.4 Device Configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.6.5 STALL Condition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.7 Data Communication via Endpoints EP1 and EP2 . . . . . . . . . . . . . 75
4.7.1 Receiving Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
4.7.2 Transmission of Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
4.8 Host Interaction: Vendor ID and Product ID . . . . . . . . . . . . . . . . . 78
4.9 Windows Device Driver . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78

Appendix A. Supported Standard Device Requests

Supported Standard Device Requests . . . . . . . . . . . . . . . . . . . . . . . . . . . 81

Appendix B. USB08 Descriptors

B.1 Contents . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
B.2 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
B.3 Device Descriptor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
B.4 Configuration Descriptor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
B.5 Interface Descriptor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
B.6 Endpoint 1 Descriptor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
B.7 Endpoint 2 Descriptor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
B.8 String Descriptors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
Appendix C. Source Code Files

C.1 Contents ................................................................. 89
    HC908JB8.H ......................................................... 90
    U08USB.H .......................................................... 93
    U08232.H .......................................................... 96
    U08LED.H .......................................................... 96
    U08MAIN.C ......................................................... 97
    U08DESC.C ......................................................... 100
    U08USB.C .......................................................... 104
    U08232.C .......................................................... 113
    U08KEY.C .......................................................... 116
    U08ADC.C .......................................................... 117
    VECJB8.C .......................................................... 119
    CRSJB8.S .......................................................... 120
    USB08.LKF ......................................................... 121
    BUILD.BAT ......................................................... 121
    USB08.MAP ......................................................... 122
    USB08.S19 ......................................................... 125

Appendix D. Bill of Materials and Schematic

Bill of Materials and Schematic .................................... 127

Appendix E. Universal USB Device Driver (USBIO)

E.1 Contents ............................................................. 132
E.2 Introduction ......................................................... 135
E.3 Overview ........................................................... 135
    E.3.1 Platforms ..................................................... 136
    E.3.2 Features ...................................................... 136
E.4 Architecture ......................................................... 138
    E.4.1 USBIO Object Model ........................................ 140
    E.4.1.1 USBIO Device Objects ................................... 140
    E.4.1.2 USBIO Pipe Objects ...................................... 142
    E.4.2 Establishing a Connection to the Device ............... 144
    E.4.3 Power Management ......................................... 146
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>E.4.4</td>
<td>Device State Change Notifications</td>
<td>148</td>
</tr>
<tr>
<td>E.5</td>
<td>Programming Interface</td>
<td>149</td>
</tr>
<tr>
<td>E.5.1</td>
<td>Programming Interface Overview</td>
<td>149</td>
</tr>
<tr>
<td>E.5.2</td>
<td>Control Requests</td>
<td>150</td>
</tr>
<tr>
<td>E.5.3</td>
<td>Data Transfer Requests</td>
<td>182</td>
</tr>
<tr>
<td>E.5.3.1</td>
<td>Bulk and Interrupt Transfers</td>
<td>182</td>
</tr>
<tr>
<td>E.5.3.2</td>
<td>Isochronous Transfers</td>
<td>184</td>
</tr>
<tr>
<td>E.5.4</td>
<td>Input and Output Structures</td>
<td>185</td>
</tr>
<tr>
<td>E.5.5</td>
<td>Enumeration Types</td>
<td>214</td>
</tr>
<tr>
<td>E.5.6</td>
<td>Error Codes</td>
<td>218</td>
</tr>
<tr>
<td>E.6</td>
<td>USBIO Class Library</td>
<td>220</td>
</tr>
<tr>
<td>E.6.1</td>
<td>CUsbIo Class</td>
<td>220</td>
</tr>
<tr>
<td>E.6.2</td>
<td>CUsbIoPipe Class</td>
<td>221</td>
</tr>
<tr>
<td>E.6.3</td>
<td>CUsbIoThread Class</td>
<td>222</td>
</tr>
<tr>
<td>E.6.4</td>
<td>CUsbIoReaderClass</td>
<td>222</td>
</tr>
<tr>
<td>E.6.5</td>
<td>CUsbIoWriter Class</td>
<td>222</td>
</tr>
<tr>
<td>E.6.6</td>
<td>CUsbIoBufClass</td>
<td>223</td>
</tr>
<tr>
<td>E.6.7</td>
<td>CUsbIoBufPool Class</td>
<td>223</td>
</tr>
<tr>
<td>E.7</td>
<td>USBIO Demo Application</td>
<td>223</td>
</tr>
<tr>
<td>E.7.1</td>
<td>Dialog Pages for Device Operations</td>
<td>224</td>
</tr>
<tr>
<td>E.7.1.1</td>
<td>Device</td>
<td>224</td>
</tr>
<tr>
<td>E.7.1.2</td>
<td>Descriptors</td>
<td>224</td>
</tr>
<tr>
<td>E.7.1.3</td>
<td>Configuration</td>
<td>225</td>
</tr>
<tr>
<td>E.7.1.4</td>
<td>Interface</td>
<td>225</td>
</tr>
<tr>
<td>E.7.1.5</td>
<td>Pipes</td>
<td>225</td>
</tr>
<tr>
<td>E.7.1.6</td>
<td>Class or Vendor Request</td>
<td>226</td>
</tr>
<tr>
<td>E.7.1.7</td>
<td>Feature</td>
<td>226</td>
</tr>
<tr>
<td>E.7.1.8</td>
<td>Other</td>
<td>226</td>
</tr>
<tr>
<td>E.7.1.9</td>
<td>Dialog Pages for Pipe Operations</td>
<td>227</td>
</tr>
<tr>
<td>E.7.1.10</td>
<td>Pipe</td>
<td>227</td>
</tr>
<tr>
<td>E.7.1.11</td>
<td>Buffers</td>
<td>227</td>
</tr>
<tr>
<td>E.7.1.12</td>
<td>Control</td>
<td>228</td>
</tr>
<tr>
<td>E.7.1.13</td>
<td>Read from Pipe to Output Window</td>
<td>228</td>
</tr>
<tr>
<td>E.7.1.14</td>
<td>Read from Pipe to File</td>
<td>228</td>
</tr>
<tr>
<td>E.7.1.15</td>
<td>Write from File to Pipe</td>
<td>229</td>
</tr>
</tbody>
</table>
# Table of Contents

| E.8 | Installation Issues | 229 |
| E.8.1 | Automated Installation: The USBIO Installation Wizard | 229 |
| E.8.2 | Manual Installation: The USBIO Setup Information File | 232 |
| E.8.3 | Uninstalling USBIO | 236 |
| E.8.4 | Building a Customized Driver Setup | 237 |
| E.9 | Registry Entries | 239 |
| E.10 | Related Documents | 241 |
| E.11 | Light Version Limitations | 241 |
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Demo Board Connected to the USB Hub</td>
<td>18</td>
</tr>
<tr>
<td>1-2</td>
<td>Found New Hardware Screen</td>
<td>19</td>
</tr>
<tr>
<td>1-3</td>
<td>Found New Hardware Wizard Start Screen</td>
<td>20</td>
</tr>
<tr>
<td>1-4</td>
<td>Locate Driver Files Screen</td>
<td>21</td>
</tr>
<tr>
<td>1-5</td>
<td>Driver Files Search Results Screen</td>
<td>22</td>
</tr>
<tr>
<td>1-6</td>
<td>Found New Hardware Wizard Finish Screen</td>
<td>23</td>
</tr>
<tr>
<td>1-7</td>
<td>Windows Demo Application IO08USB</td>
<td>24</td>
</tr>
<tr>
<td>1-8</td>
<td>Driver Entry for USB08 in the Device Manager Window</td>
<td>25</td>
</tr>
<tr>
<td>2-1</td>
<td>USB08 Evaluation Board</td>
<td>30</td>
</tr>
<tr>
<td>2-2</td>
<td>PCB Component Side Layout Plan</td>
<td>37</td>
</tr>
<tr>
<td>2-3</td>
<td>Detailed Layout Plan</td>
<td>37</td>
</tr>
<tr>
<td>2-4</td>
<td>Solder Bridge Placement on Downside of the PCB</td>
<td>39</td>
</tr>
<tr>
<td>3-1</td>
<td>Structure and Dependencies of the Firmware Files</td>
<td>45</td>
</tr>
<tr>
<td>3-2</td>
<td>Measurement of Resistor Values Using a Digital Input</td>
<td>52</td>
</tr>
<tr>
<td>4-1</td>
<td>USB Address Register (UADDR)</td>
<td>65</td>
</tr>
<tr>
<td>4-2</td>
<td>USB Control Register 3 (UCR3)</td>
<td>66</td>
</tr>
<tr>
<td>4-3</td>
<td>USB Interrupt Register 0 (UIR0)</td>
<td>68</td>
</tr>
<tr>
<td>4-4</td>
<td>USB Control Register 0 (UCR0)</td>
<td>68</td>
</tr>
<tr>
<td>4-5</td>
<td>USB Interrupt Register 1 (UIR1)</td>
<td>69</td>
</tr>
<tr>
<td>4-6</td>
<td>USB Status Register 0 (USR0)</td>
<td>70</td>
</tr>
<tr>
<td>4-7</td>
<td>USB Control Register 0 (UCR0)</td>
<td>71</td>
</tr>
<tr>
<td>4-8</td>
<td>USB Address Register (UADDR)</td>
<td>72</td>
</tr>
<tr>
<td>4-9</td>
<td>USB Interrupt Register 1 (UIR1)</td>
<td>75</td>
</tr>
<tr>
<td>4-10</td>
<td>USB Status Register 1 (USR1)</td>
<td>76</td>
</tr>
<tr>
<td>4-11</td>
<td>USB Control Register 1 (UCR1)</td>
<td>77</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-1</td>
<td>USB08 Evaluation Board Schematic</td>
<td>129</td>
</tr>
<tr>
<td>E-1</td>
<td>USB Driver Stack</td>
<td>138</td>
</tr>
<tr>
<td>E-2</td>
<td>USBIO Device and Pipe Objects Example</td>
<td>143</td>
</tr>
<tr>
<td>E-3</td>
<td>Layout of an Isochronous Transfer Buffer</td>
<td>183</td>
</tr>
<tr>
<td>E-4</td>
<td>USBIO Class Library</td>
<td>220</td>
</tr>
<tr>
<td>Table</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>2-1</td>
<td>Port A Monitor Mode Entry Levels</td>
<td>33</td>
</tr>
<tr>
<td>2-2</td>
<td>Monitor Mode Cable Pin Configuration</td>
<td>34</td>
</tr>
<tr>
<td>2-3</td>
<td>Jumper Configuration</td>
<td>38</td>
</tr>
<tr>
<td>2-4</td>
<td>Solder Bridges Configuration</td>
<td>39</td>
</tr>
<tr>
<td>2-5</td>
<td>MC68HC908JB8 Memory Map</td>
<td>41</td>
</tr>
<tr>
<td>3-1</td>
<td>Memory Utilization</td>
<td>47</td>
</tr>
<tr>
<td>4-1</td>
<td>Low-Speed USB Packet Types</td>
<td>62</td>
</tr>
<tr>
<td>4-2</td>
<td>MC68HC908JB8 Endpoint Configuration</td>
<td>66</td>
</tr>
<tr>
<td>D-1</td>
<td>Bill of Materials for USB08 V 1.01</td>
<td>128</td>
</tr>
<tr>
<td>E-1</td>
<td>I/O Operations Supported by the USBIO Device Driver</td>
<td>149</td>
</tr>
<tr>
<td>E-2</td>
<td>Error Codes Defined by the USBIO Device Driver</td>
<td>218</td>
</tr>
<tr>
<td>E-3</td>
<td>Registry Parameters Supported by the USBIO Driver</td>
<td>239</td>
</tr>
</tbody>
</table>
Section 1. USB08 Quick Start

1.1 Contents

1.2 Introduction ........................................... 17
1.3 Required System Configuration ............................ 17
1.4 Connecting the Demo Board to the PC .................... 18
1.5 Driver Installation ........................................ 19
1.6 Starting the Windows Demo Application ................. 24

1.2 Introduction

This section describes the connection and startup of the USB08 (universal serial bus) evaluation board demo application. The main component of the USB08 is the Motorola MC68HC908JB8 8-bit microcontroller (MCU).

1.3 Required System Configuration

To connect the USB08, you will need a personal computer (PC) with one of the following Microsoft® operating systems:

- Windows® 98
- Windows ME
- Windows 2000 Professional

**NOTE:** Ensure that the PC has the necessary hardware (universal serial bus (USB) host controller and USB root hub) and that the necessary system drivers are installed.

Microsoft and Windows are registered trademarks of Microsoft Corporation in the United States and/or other countries.
1.4 Connecting the Demo Board to the PC

Since low-speed USB devices should be equipped with a captive connection, the USB cable is fixed on the USB08 board (downstream direction). In the upstream direction (PC/host side), the USB connections are always type A. Therefore, the cable of the USB08 demo board has a type A plug.

The connection of the demo board is made directly to the USB socket of the PC or, as shown in the Figure 1-1, to a USB hub.

![Figure 1-1. Demo Board Connected to the USB Hub](image)

The board supply current can be delivered by the USB connection. Therefore, the jumper JP2, which is directly beside the USB cable, has to be in the position **Bus Powered**. The jumper JP1-A (jumper block, highest position) must be opened, which corresponds to the default shipping configuration.
1.5 Driver Installation

For this example, the installation of the driver software is described using the Windows 2000 operating system. The installation using Windows 98 (second edition) looks quite similar.

After the electrical connection of the demo board, the Windows operating system recognizes the presence of a new hardware component and shows the message **Found New Hardware**.

![Found New Hardware Screen](image)
The hardware assistant, Figure 1-3, now tries to find the suitable driver information for the USB08 evaluation board. Click the Next button.

**NOTE:** The installation using the Windows 2000 operating system requires administrator rights.

![Found New Hardware Wizard Start Screen](image-url)

**Figure 1-3. Found New Hardware Wizard Start Screen**
Insert the USB08 product CD into the CD-ROM drive and mark the appropriate check box **CD-ROM drives** as shown in **Figure 1-4.** Click the **Next** button.

![Figure 1-4. Locate Driver Files Screen](image)
As shown in **Figure 1-5**, the hardware assistant will find the driver information file usbio_el.inf in the root directory of the CD ROM. Confirm this selection by clicking **Next**.

![Driver Files Search Results Screen](image)

**Figure 1-5. Driver Files Search Results Screen**
The Windows operating system now copies the INF file and the driver file usbio_el.sys to the appropriate Windows directories. After clicking Finish (Figure 1-6), the driver installation will be completed and the USB device will be ready for use.

**NOTE:** The installation does not require a restart of the computer, since this is a true Plug & Play installation.
1.6 Starting the Windows Demo Application

The windows demo application:

- Shows the measured values and push button information coming from the demo board
- Allows the controlling of the demo board light-emitting diodes (LED)

The demo application is located in the root directory of the USB08 product CD. The file name of the demo application is IO08USB.EXE. This program can be started directly from the CD.

As shown in Figure 1-7, the bottom line of the application window shows the status of the connection established to the USB08 demo board. The LED symbols on the left upper side of the application window can be switched on or off by pressing the keys of the USB08 demo board.
By setting the check boxes on the upper right side it is possible to switch on or off the LEDs of the demo board. The needle pointer instruments on the lower side of the application window indicate the measured values of the three variable resistors:

- Input 1 represents the photo sensor.
- Input 2 shows the thermistor value.
- Input 3 can be varied using the turnable regulator.

The USB08 evaluation board can be disconnected from the USB port and reconnected at any time, because the drivers are automatically activated or deactivated by the Windows operating system. The activation/deactivation of the drivers can be watched in the operating system’s device manager window (start button/settings/control panel/system/device manager). The catalog entry USB0 controlled devices and the device entry USB08 Evaluation Board are visible only if the hardware is present. See Figure 1-8.

![Figure 1-8. Driver Entry for USB08 in the Device Manager Window](image)
The Windows demo application, IO08USB.EXE, must be re-started in the case of a hardware connection interrupt. This is because an automatic resynchronization (though it would be possible) was not implemented here. The demo application is arranged as simply and as understandable as possible.
Section 2. Hardware Description

2.1 Contents

2.2 Introduction ......................................................... 27
2.3 Technical Data .......................................................... 28
  2.3.1 MC68HC908JB8 Microcontroller ................................. 28
  2.3.2 USB08 Evaluation Board ........................................ 29
2.4 Circuit Description .................................................... 30
  2.4.1 MCU Core Circuit and USB Interface ........................... 31
  2.4.2 Input/Output Functions ......................................... 32
  2.4.3 Monitor Mode Interface ......................................... 33
  2.4.4 User RS232 Port .................................................. 35
  2.4.5 Power Supply ..................................................... 36
2.5 Board Layout ........................................................... 36
2.6 Jumpers and Bridges .................................................. 38
2.7 Connectors ............................................................... 40
  2.7.1 Expansion Connector X1 ......................................... 40
  2.7.2 Monitor Mode Connector X2 .................................... 40
  2.7.3 User RS232 Connector X3 ....................................... 41
2.8 Memory Map ............................................................ 41

2.2 Introduction

The USB08 evaluation board is the hardware platform for the universal serial bus (USB) reference design. The board serves the provided demo application, which is contained in the integrated FLASH memory of the M68HC08 microcontroller (MCU).
Beyond that, the USB08 enables the implementation and testing of its own M68HC08 software for evaluation purposes. For that purpose, the board contains a monitor mode interface for reprogramming and debugging. The monitor mode interface of the USB08 is compatible with Motorola development tools such as the M68ICS08JB8 and other third-party tools.

2.3 Technical Data

This subsection provides technical data for both the MC68HC908JB8 and the USB08 evaluation board.

2.3.1 MC68HC908JB8 Microcontroller

The main component of the USB08 evaluation board is the MC68HC908JB8, a Motorola 8-bit MCU. Features of the MC68HC908JB8 include:

- Efficient M68HC08 MCU core
- 8 Kbytes of on-chip FLASH memory with security feature
- 256 bytes of random-access memory (RAM)
- 3-MHz bus clock (6-MHz quartz crystal)
- 2 × 16-bit timer with:
  - Input capture
  - Output compare
  - Pulse-width modulator (PWM)
- Low-speed USB 1.1 interface module
- Integrated 3-V voltage regulator
- Computer operating properly (COP) watchdog timer
- Low-voltage interrupt (LVI) reset controller
- Inputs for RESET and IRQ pins
- Up to 21 input/output (I/O) lines
2.3.2 USB08 Evaluation Board

Features of the USB08 evaluation board include:

- M68HC908JB8 MCU packaged in a 28-pin small-outline integrated circuit package (SOIC)
- Three light-emitting diodes (LED)
- Three input keys
- Three analog sensors:
  - Light
  - Temperature
  - Angle of rotation
- Current supplied alternatively via USB connection or on-board voltage regulator
- Monitor mode interface for in-system programming and debugging
- Additional RS232 interface for connection to PC or serial liquid crystal display (LCD)
- Push buttons for reset and IRQ
- Jumper for power-on reset (POR)
- All MCU pins are accessible via a 26-pin universal expansion connector
- Small user breadboard area reserved for customer circuit extensions

The USB08 evaluation board is shown in Figure 2-1.
2.4 Circuit Description

A schematic of the USB08 demo board is provided in Appendix D. Bill of Materials and Schematic. The MC68HC908JB8 MCU needs few external elements. A wide range of peripheral functions including the USB module and an 8-Kbyte FLASH memory are integrated on-chip. The MC68HC908JB8 is offered in several packages. For the USB08 reference design, the 28-pin SOIC version was chosen instead of the 20-pin dual in-line package (DIP) because the SOIC package has some additional I/O pins.
2.4.1 MCU Core Circuit and USB Interface

The operating voltage, \( V_{DD} \), is supported by the capacitors C6 and C3 close to the MCU. Out of this primary operating voltage of approximately 5 V, the MCU produces an internal operating voltage, \( V_{REG} \), of 3.3 V, using an integrated voltage regulator. This voltage is supported by two capacitors, C4 and C5, and continues in the circuit as \( V_{CC} \).

In particular, the internal voltage \( V_{CC}/V_{REG} \) is used as the USB interface driver voltage supply. \( V_{CC}/V_{REG} \) is accessible over the expansion plug connector X1. However, it must be noted that \( V_{CC}/V_{REG} \) can be additionally loaded only with a few milliamps.

For clock generation, the external elements Q1, C1, C2, and R18 are used. These elements form a Pierce oscillator together with the active elements integrated in the MCU. This oscillator produces a clock frequency of 6 MHz. The internal bus clock of the MCU (3 MHz) as well as the USB clock (1.5 MHz) are derived from the main clock frequency.

The USB data lines are connected to the MCU pins PTE3 (USB D+) and PTE4 (USB D–). So that the USB hub will be able to classify this equipment as a low-speed USB device, a pullup resistance of 1.5 k\( \Omega \) (R7) to the data line D– is required.

On the demo board, R7 is not installed. This is because the MC67HC908JB8 has an additional internal pullup at PTE4 which can be activated and de-activated by software.

To optimize the connection adjustment, the serial resistors in the data lines R16 and R17 and inductances (ferrite beads) in the current supply path L1 and L2 are used. However, these measures are optional.

The reset system of the M68HC08 shows clear differences from other Motorola MCUs (M68HC11 and M68HC12). For example, the capacitor C19 at the reset pin of this circuit could never be used in an M68HC11 system. This is because the MC68HC908JB8 has an integrated low-voltage inhibit (LVI) circuit. Therefore, no external reset controller is required.
2.4.2 Input/Output Functions

For demonstration purposes, the board has:

- Three push buttons
- Three light-emitting diodes (LED)
- Sensor resistors

The push buttons are connected to the three port pins PTA4, PTA5, and PTA6. By pushing the buttons, a low level is produced on the appropriate input. Since the port pins have internal pullup resistors, no external resistors are required. The buttons are bridged with capacitors, to support correct reading of the inputs by the software and to avoid noise. The occurrence of the high-low edge at the respective pin of port A is an input event which results in the generation of a keyboard interrupt by the MC68HC08JB8. This interrupt is then used by the program (see Section 3. Software Module Descriptions).

For optical signalling, three LEDs are attached to port D. These port pins have a high drive capability of up to 25 mA. Therefore, it is not necessary to use a driver. On the board, PTD0, PTD1, and PTD2 are used for LED control. All outputs generated by the port D pins have an open-drain characteristic and are 5-V tolerant.

The remaining port D pins (PTD3–PTD6) are used for controlling the software analog-to-digital converter (ADC). The ADC implementation is described in detail in Section 3. Software Module Descriptions as well as in the application note entitled Simple A/D for MCUs without Built-in A/D Converters, Motorola document order number AN477/D. This application note can be found on the World Wide Web at:

http://www.motorola.com/semiconductors/

The software ADC senses the resistance of:

- R1 (photo resistor)
- R2 (thermistor)
- R3 (potentiometer)

To determine capacitor load times, the MCU pins PTE0–PTE2 serve as trigger inputs for the software ADC.
The I/O pins of the MCU are accessible on the expansion connector X1. User specific peripheral circuits can be attached to X1.

**NOTE:** It may be that not all functions of the demo board may be used with user-specific peripheral circuits attached to X1.

### 2.4.3 Monitor Mode Interface

For FLASH programming and software debugging, the MC68HC908JB8 uses a special operating mode, monitor mode. The difference between monitor mode and normal user mode is that firmware out of the read-only memory (ROM) is executed instead of the user program. First, this firmware examines a set of I/O pins and specifies the concrete operating parameters. Finally, this firmware establishes an asynchronous serial interface function on the port pin PTA0. This interface works bidirectionally (half duplex) and corresponds to the usual RS232 conventions. The baud rate equals 9600 baud. An additional requirement, besides the quartz clock (6 MHz), is the allocation of certain logic levels to some port pins as listed in **Table 2-1**.

<table>
<thead>
<tr>
<th>Port Pin</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTA0</td>
<td>High</td>
</tr>
<tr>
<td>PTA1</td>
<td>High</td>
</tr>
<tr>
<td>PTA2</td>
<td>Low</td>
</tr>
<tr>
<td>PTA3</td>
<td>High</td>
</tr>
</tbody>
</table>

The monitor mode circuitry on the evaluation board produces the levels shown in **Table 2-1** using four pullup or pulldown resistors. These resistors are connected to the MCU using the jumpers JP1-C–JP1-F. After removing these jumpers, a previously loaded user program can access the four port A pins without restrictions.

Apart from the above requirements, to enter monitor mode it is necessary to apply a voltage of approximately 7–10 V to the IRQ pin of the MCU. This voltage is generated by the RS232 transceiver’s (IC2) charge pump and limited to 8.2 V using the breakdown diode D7. JP1-A
Hardware Description

is the first jumper of the jumper block JP1, and it must be set in order to apply high voltage to IRQ.

If the monitor mode interface is not needed or if it disturbs the investigation of certain circuit configurations, it can be uncoupled completely from the MCU core. For this purpose:

- All jumpers of jumper block JP1 have to be removed.
- RS232 receiver IC2 has to be removed from the socket.

Using the X2 plug connector, the monitor mode interface is connected to the PC. The monitor mode cable consists of:

- A flat cable with a Berg connector (2 × 5 pin, crimping connection) on the device side
- A sub-D9 connector (crimping connection) on the PC side

A one-to-one connection is implemented by this cable configuration, as shown in **Table 2-2**.

<table>
<thead>
<tr>
<th>X2 Pin</th>
<th>USB08 Monitor</th>
<th>PC RS232</th>
<th>Sub-D 9 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>T1OUT</td>
<td>RxD</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>R1IN</td>
<td>TxD</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>GND</td>
<td>GND</td>
<td>5</td>
</tr>
</tbody>
</table>

The MC68HC908JB8 logic levels are based on the operating voltage \( V_{CC} \) (3.3 V); however, the transceiver IC2 works with \( V_{DD} \) (5 V). The adjustment of the logic levels according to specification is not difficult (refer to the individual integrated circuit data sheets). The Schottky diode, D6, enables the push/pull exit R1OUT to be wired-OR capable and prevents a feeding of levels beyond the tolerance limit of the input PTA0.
2.4.4 User RS232 Port

The monitor mode interface uses only one sending/receiving channel of the RS232 transceiver IC2. The remaining channel is used for an additional user RS232 port.

In contrast to the RS232 channel for the monitor mode interface, the user RS232 port incorporates separate sending and receiving lines. The PTA7 pin of the MCU is used for receiving and the PTC0 pin is used for sending.

NOTE: The MC68HC908JB8 does not have a serial communications interface (SCI) hardware module for asynchronous serial communication. Therefore, the necessary timing has to be generated by software.

If PTA7 and/or PTC0 are to be used, the diode D5 serves for the adjustment of the logic levels between 5 V and 3 V. Otherwise, the RS232 transceiver can be uncoupled from the MCU by removing the jumpers JP1-G and JP1-H.

X3 is the user RS232 port plug connector. If this interface is attached to a PC, a line connection similar to the monitor mode interface is necessary. In this case, the bridges BR1 and BR2 on the downside of the printed circuit board (PCB) (see Figure 2-3) have to be connected in positions 1 and 2. For this configuration, the PC works as a host and the USB08 board represents the device side.

The reverse case happens, if a serial liquid crystal display (LCD) is to be operated at the user RS232 port. In this configuration, the USB08 board is the host and the LCD module represents the device side. The necessary RxD/TxD crossing is done by configuration of the bridges BR1 and BR2 in positions 2 and 3. At the same time, the serial LCD can be supplied with operating voltage by closing the bridge BR3.

NOTE: This specification deviates from standard RS232 mapping.

Serial alphanumeric LCDs are offered by several vendors. In the test configuration, the LCDs used are from the Canadian manufacturer Matrix Orbital (http://www.matrixorbital.com).
2.4.5 Power Supply

Power can be supplied to the USB08 board by using the USB or via the voltage regulator IC3. The change between these options is done by replacing the jumper JP2. If the jumper is placed in position 2–3 (Bus Powered), the operating power is supplied by $V_{Bus}$ and $V_{GND}$ from the USB.

If a dc voltage between 8 V and 20 V is fed into the power plug X4 in jumper position 1–2 (Self Powered), the voltage regulator IC3 supplies 5 V in the case. The solder bridge BR4 on the downside of the PCB (see Figure 2-3) has to be in position 1–2. Alternatively, if the bridge BR4 is in position 2–3, a stabilized 5-V power supply can be used to feed $V_{DD}$ directly.

The voltage regulator IC3 is specified with 1 ampere. Although no special cooling measures are intended, IC3 is more than sufficiently dimensioned. The input current of the board, even in the worst case, is clearly smaller than 100 mA.

A USB hub supplies at least 100 mA. Therefore, the power supply of the board via the USB is possible without any problem. The USB08 board power input specification should be registered in the device descriptor of the USB device (see Section 4. Universal Serial Bus (USB) Interface).

2.5 Board Layout

Figure 2-2 and Figure 2-3 show the components and parts layout, as well as a general picture of the board.

On the component side, Figure 2-2:

- The jumpers, plugs, push buttons, and LEDs are marked.
- The USB cable is fixed on the board with a cable strap, and the four line ends are soldered directly to the X5 connection points (without patch cord). This kind of connection is usual for low-speed USB devices.

A detailed layout plan of the USB08 board with the names of all components is shown in Figure 2-3.
Figure 2-2. PCB Component Side Layout Plan

Figure 2-3. Detailed Layout Plan
2.6 Jumpers and Bridges

The jumper configuration is shown in Table 2-3.

Table 2-3. Jumper Configuration

<table>
<thead>
<tr>
<th>Jumper</th>
<th>Position(a)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP1-A</td>
<td>Open*</td>
<td>Normal user mode</td>
</tr>
<tr>
<td></td>
<td>Closed</td>
<td>High voltage on IRQ to enter monitor mode</td>
</tr>
<tr>
<td>JP1-B</td>
<td>Open</td>
<td>RS232 is disconnected from the power supply.</td>
</tr>
<tr>
<td></td>
<td>Closed*</td>
<td>RS232 is connected to the power supply.</td>
</tr>
<tr>
<td>JP1-C</td>
<td>Open</td>
<td>PTA0 can be used without restriction.</td>
</tr>
<tr>
<td></td>
<td>Closed*</td>
<td>PTA0 is used for monitor mode communication.</td>
</tr>
<tr>
<td>JP1-D</td>
<td>Open</td>
<td>PTA1 can be used without restriction.</td>
</tr>
<tr>
<td></td>
<td>Closed*</td>
<td>PTA1 is used for monitor mode configuration.</td>
</tr>
<tr>
<td>JP1-E</td>
<td>Open</td>
<td>PTA2 can be used without restriction.</td>
</tr>
<tr>
<td></td>
<td>Closed*</td>
<td>PTA2 is used for monitor mode configuration.</td>
</tr>
<tr>
<td>JP1-F</td>
<td>Open</td>
<td>PTA3 can be used without restriction.</td>
</tr>
<tr>
<td></td>
<td>Closed*</td>
<td>PTA3 is used for monitor mode configuration.</td>
</tr>
<tr>
<td>JP1-G</td>
<td>Open</td>
<td>PTA7 can be used without restriction.</td>
</tr>
<tr>
<td></td>
<td>Closed*</td>
<td>PTA7 serves as receiving line for the user RS232.</td>
</tr>
<tr>
<td>JP1-H</td>
<td>Open</td>
<td>PTC0 can be used without restriction.</td>
</tr>
<tr>
<td></td>
<td>Closed*</td>
<td>PTC0 serves as transmission line for the user RS232.</td>
</tr>
<tr>
<td>JP2</td>
<td>1-2</td>
<td>Self-powered: power supply via voltage regulator</td>
</tr>
<tr>
<td></td>
<td>2-3*</td>
<td>Bus-powered: power supply via USB</td>
</tr>
</tbody>
</table>

Placement of the solder bridges on the downside of the PCB is shown in Figure 2-4. Table 2-4 shows the solder bridges configuration.
**Figure 2-4. Solder Bridge Placement on Downside of the PCB**

**Table 2-4. Solder Bridges Configuration**

<table>
<thead>
<tr>
<th>Solder Bridge</th>
<th>Position(1)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BR1 and BR2</td>
<td>1-2*</td>
<td>User RS232 configured in external device mode (PC)</td>
</tr>
<tr>
<td></td>
<td>2-3</td>
<td>User RS232 configured in host mode (LCD connection)</td>
</tr>
<tr>
<td>BR3</td>
<td>Open*</td>
<td>$V_{CC}$ is not present at user RS232 port (standard).</td>
</tr>
<tr>
<td></td>
<td>Closed</td>
<td>$V_{CC}$ is present at pin 9 of the user RS232.</td>
</tr>
<tr>
<td>BR4</td>
<td>1-2*</td>
<td>Power supply via voltage regulator, 8–20 V needed at X4</td>
</tr>
<tr>
<td></td>
<td>2-3</td>
<td>Power supply directly from X4, must be stabilized at 5 V</td>
</tr>
</tbody>
</table>

1. * = delivery status
2.7 Connectors

The connectors are described here.

2.7.1 Expansion Connector X1

<table>
<thead>
<tr>
<th></th>
<th>VDD</th>
<th>1</th>
<th>2</th>
<th>VCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTS</td>
<td>3</td>
<td>4</td>
<td>PTA0</td>
<td></td>
</tr>
<tr>
<td>PTD0</td>
<td>5</td>
<td>6</td>
<td>PTA1</td>
<td></td>
</tr>
<tr>
<td>PTD1</td>
<td>7</td>
<td>8</td>
<td>PTA2</td>
<td></td>
</tr>
<tr>
<td>PTD2</td>
<td>9</td>
<td>10</td>
<td>PTA3</td>
<td></td>
</tr>
<tr>
<td>PTD3</td>
<td>11</td>
<td>12</td>
<td>PTA4</td>
<td></td>
</tr>
<tr>
<td>PTD4</td>
<td>13</td>
<td>14</td>
<td>PTA5</td>
<td></td>
</tr>
<tr>
<td>PTD5</td>
<td>15</td>
<td>16</td>
<td>PTA6</td>
<td></td>
</tr>
<tr>
<td>PTD6</td>
<td>17</td>
<td>18</td>
<td>PTA7</td>
<td></td>
</tr>
<tr>
<td>PTE3</td>
<td>19</td>
<td>20</td>
<td>PTE0</td>
<td></td>
</tr>
<tr>
<td>PTE4</td>
<td>21</td>
<td>22</td>
<td>PTE1</td>
<td></td>
</tr>
<tr>
<td>PTC0</td>
<td>23</td>
<td>24</td>
<td>PTE2</td>
<td></td>
</tr>
<tr>
<td>GND</td>
<td>25</td>
<td>26</td>
<td>GND</td>
<td></td>
</tr>
</tbody>
</table>

2.7.2 Monitor Mode Connector X2

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>2</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>N.C.</td>
<td>1</td>
<td>2</td>
<td>N.C.</td>
<td></td>
</tr>
<tr>
<td>PC_RxD</td>
<td>3</td>
<td>4</td>
<td>N.C.</td>
<td></td>
</tr>
<tr>
<td>PC_TxD</td>
<td>5</td>
<td>6</td>
<td>N.C.</td>
<td></td>
</tr>
<tr>
<td>N.C.</td>
<td>7</td>
<td>8</td>
<td>N.C.</td>
<td></td>
</tr>
<tr>
<td>GND</td>
<td>9</td>
<td>10</td>
<td>N.C.</td>
<td></td>
</tr>
</tbody>
</table>
2.7.3 User RS232 Connector X3

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>N.C.</td>
<td>N.C.</td>
<td>Rx (Tx)</td>
<td>N.C.</td>
<td>N.C.</td>
<td>N.C.</td>
<td>N.C.</td>
<td>7</td>
<td>8</td>
<td>VDD</td>
<td>N.C.</td>
</tr>
<tr>
<td>GND</td>
<td>9</td>
<td>10</td>
<td>N.C.</td>
<td>N.C.</td>
<td>N.C.</td>
<td>N.C.</td>
<td>N.C.</td>
<td>N.C.</td>
<td>N.C.</td>
<td>N.C.</td>
</tr>
</tbody>
</table>

2.8 Memory Map

Table 2-5. MC68HC908JB8 Memory Map

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Size</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x003F</td>
<td>64 bytes</td>
<td>Control registers</td>
</tr>
<tr>
<td>0x0040</td>
<td>0x013F</td>
<td>256 bytes</td>
<td>RAM</td>
</tr>
<tr>
<td>0x0140</td>
<td>0xDBFF</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>0xDC00</td>
<td>0xFBFF</td>
<td>8 Kbytes</td>
<td>FLASH memory</td>
</tr>
<tr>
<td>0xFC00</td>
<td>0xFFDF</td>
<td>—</td>
<td>Reserved</td>
</tr>
<tr>
<td>0xFFFF</td>
<td>0xFFFF</td>
<td>32 bytes</td>
<td>Interrupt vector table (FLASH)</td>
</tr>
</tbody>
</table>

For a detailed description of the MC68HC908JB8 memory map, in particular the addresses of control registers and interrupt vectors, refer to the *MC68HC908JB8 Technical Data*, Motorola document order number MC68HC908JB8/D.
Section 3. Software Module Descriptions

3.1 Contents

3.2 Introduction ......................................................... 43
3.3 General Structure of the M68HC08 Firmware .................... 44
3.4 How to Build the Compiler Project ............................... 45
3.5 Main Module U08MAIN.C ........................................... 48
3.6 Interrupt and Reset Vector Module VECJB8.C ................. 49
3.7 C Startup Module CRTSJB8.S .................................... 50
3.8 Push Button Module U08KEY.C .................................. 50
3.9 LED Control with U08LED.H .................................... 52
3.10 Software ADC Module U08ADC.C ............................... 52
3.11 RS232 Communication Module U08232.C ....................... 54
3.12 USB Communication Module U08USB.C ......................... 56
3.13 Compiler Specific Adjustments ................................. 57

3.2 Introduction

This section describes the structure and interaction of the software
modules. These software modules, running on the Motorola
microcontroller MC68HC908JB8, form the firmware of the USB08
reference design.
3.3 General Structure of the M68HC08 Firmware

The firmware of the M68HC08 consists of several source code modules which are embedded into a common compiler project. The main() function is contained in the module U08MAIN.C. It controls the program sequence via an endless loop (as usual in embedded software). The module U08KEY.C settles the scanning of the input keys. Control of the light-emitting diodes (LED) on the board is done using simple C macros; no special C module is required for this purpose. The module U08ADC.C is responsible for reading of the resistive sensors.

These modules are supported by the file VECJB8.C, which contains the interrupt and reset vectors. The four C source code modules are merged into a common compiler project. In addition, the assembler module CRTSJ8B.S which contains the C startup code is required.

The control of keys, LEDs, and analog-to-digital (A/D) transmitters are support functions to the demonstration project because the main attention is paid to the communication interface. The communication functions are implemented directly by means of “#include” instructions in the main module U08MAIN.C.

Two ways of communication are implemented in the demonstration: RS232 or USB. The selection takes place by defining USE_USB_PIPE in the head of the main module. If this macro is defined, U08USB.C automatically becomes a part of the main module; otherwise, the RS232 communication module U08232.C is merged.

If the USB implementation in the file U08USB.C is used, the file U08DESC.C is also translated at the same time. It contains the static data for all necessary USB descriptors.

For each of the C files U08KEY.C, U08ADC.C, U08USB.C, and U08232.C there is a corresponding header file (* H) with the same base name. The functions for controlling the LEDs on the board are contained as macros in the file U08LED.H. In several cases, the header MC68HC08JB8.H contains the register and bit mask definitions required by the MC68HC908JB8 microcontroller (MCU).
Figure 3-1 shows the structure and interaction of the modules and files that could be included by means of "#include" instructions. To accomplish a complete compiler build, the grey modules have to be included in a compiler project.

3.4 How to Build the Compiler Project

The Cosmic C Compiler can arrange a project within the compiler IDE. The compiler project owns:

- A list of source modules to be compiled
- Translation options
- Additional tools such as S-record generation

**Figure 3-1. Structure and Dependencies of the Firmware Files**
The compiler project for the USB08 reference design covers these C modules (see Figure 3-1):

- U08MAIN.C
- U08KEY.C
- U08ADC.C
- VECJB8.C

An alternative possibility consists of controlling the translation of the project via a batch file as shown in this example (BUILD.BAT).

```bash
cx6808 -v -l u08main.c u08adc.c u08key.c vecjb8.c
clink -o usb08.h08 usb08.lkf
chex -fm -h -o usb08.s19 usb08.h08
```

This batch file can be invoked under the MS-DOS® system environment to translate and link USB08 firmware components. The result of this process is a S-record file named USB08.S19, which can be loaded into the FLASH memory of the MC68HC908JB8.

Another important file for controlling the translation is the linker file USB08.LKF:

```bash
# USB08 LINK COMMAND FILE
# COSMIC HC08 C COMPILER
#
+seg .text -b 0xdc00 -n .text # program start address
+seg .const -a .text # constants follow code
+seg .bsct -b 0x0040 -n .bsct # zero page start address
+seg .ubsct -a .bsct -n .ubsct # data start address
+seg .data -a .ubsct # data start address
+def __sbss=@.bss # start address of bss
# Put your startup file here
crtsjb8.o # startup routine
# Put your files here
u08main.o
u08key.o
u08adc.o
# "c:\programs\cosmic\cx08\Lib\libi.h08"
"c:\programs\cosmic\cx08\Lib\libm.h08"
+seg .const -b 0xffff0 # vectors start address
# Put your interrupt vectors file here if needed
vecjb8.o
+def __memory=@.bss # symbol used by library
+def __stack=0x013f # stack pointer initial value
```

MS-DOS is a registered trademark of Microsoft Corporation in the United States and/or other countries.
In the linker file:

- The starting addresses of the various segments are set.
- The text segment starts at the address 0xDC00 (for example, at the beginning of the internal FLASH memory).
- The constants immediately follow the text segment.
- The zero page starts at 0x0040 instead of the usual 0x0000.
- The MC68HC908JB8 control registers are located in the address range 0x0000–0x0040.
- The data segment follows the zero page in the random-access memory (RAM).
- The interrupt vectors start at 0xFFF0 and the stack pointer is set to 0x013F (end of the internal RAM memory).

Table 3-1 shows the approximate values for memory utilization when USB communication has been implemented.

<table>
<thead>
<tr>
<th>Starting Address</th>
<th>End Address</th>
<th>Length</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0x003F</td>
<td>64 byte</td>
<td>MC68HC908JB8 control registers</td>
</tr>
<tr>
<td>0x0040</td>
<td>0x0075</td>
<td>53 byte</td>
<td>Variables in RAM</td>
</tr>
<tr>
<td>0x0076</td>
<td>0x0117</td>
<td>163 byte</td>
<td>Free RAM</td>
</tr>
<tr>
<td>0x0118</td>
<td>0x013F</td>
<td>40 byte</td>
<td>Stack in RAM</td>
</tr>
<tr>
<td>0xDC00</td>
<td>0xE2FF</td>
<td>1.8 Kbyte</td>
<td>Code and constant values</td>
</tr>
<tr>
<td>0xE300</td>
<td>0xFBFF</td>
<td>6.2 Kbyte</td>
<td>Free FLASH memory</td>
</tr>
<tr>
<td>0xFFF0</td>
<td>0xFFFF</td>
<td>16 byte</td>
<td>Interrupt and reset vectors</td>
</tr>
</tbody>
</table>
3.5 Main Module U08MAIN.C

Which variant to be compiled is specified at the head of this module using the macro USE_USB_PIPE. If defined, the USB version will be produced; otherwise, the RS232 version will be produced.

The two versions are formed by merging different “#include” files. In addition, the names of the interface functions used in the main program are standardized. For example:

- If the RS232 version is active, the getSSCI() (for receiving a character via RS232) is renamed by a macro to getPipe().
- If the USB version is active, getPipe() is mapped to getUSB().

This enables the use of uniform function names in the main program, independent of the version selected in each case.

The function main() contains the continuous loop of the master program. As usual, it is called by the C startup module after all fundamental hardware and system initializations are finished. Also, at the beginning of these initializations, the C startup module calls the function _HC08Setup(). In this function, all register accesses and initializations, which must take place immediately after system resets, are summarized. The summarizing of these initializations within its own function keeps the C startup module static. It has the advantage that the C startup module does not have to be changed and retranslated, even if further initialization steps become necessary.

At the beginning of function main() the peripheral modules used by the program are initialized:

- initPipe() — communication module (RS232 or USB)
- initLED() — LED readouts
- initKey() — keyboard entry
- initSADC() — software analog-to-digital converter (ADC)

Subsequently, the I flag is deleted to enable global interrupts.
The main program loop uses this operational sequence:

1. An analog-to-digital (A/D) conversion is performed. One of three conversion channels is updated in each cycle run. This procedure was selected because the transformation, with the simple ADC software implemented in the module \texttt{U08ADC.C}, takes several milliseconds.

2. Subsequently, the delivery of an 8-byte data telegram by the input pipe is accomplished. This length was chosen because it corresponds to the number of bytes sent by the PC program. The necessary information for the control of the three LEDs is contained in the first three bytes. If the received byte is 0, the respective LED is switched off; otherwise, it is switched on.

3. To send an answer telegram back at the host PC, first fill the send buffer utilizing the first six bytes of the eight bytes available. In the first three bytes, the status of the input keys is coded. The next three bytes transmit the last values of the three analog converter channels.

4. Now the function \texttt{putPipe()} is called eight times to send the data telegram. Afterward, the entire cycle run is repeated.

All further program functions, in particular the communication via USB and the processing of the push button events, are processed by interrupt functions.

\textbf{3.6 Interrupt and Reset Vector Module VECJB8.C}

The file \texttt{VECJB8.C} contains the definitions of the interrupt vector table placed at the end of the M68HC08 memory map. The entries in this table are the start addresses of the respective interrupt service routines. The MC68HC908JB8 uses eight (7 + 1) vectors. The last, highest position (address 0xFFFE/0xFFFF) is used by the reset vector.

In the USB08 reference design, the key pad interrupts of the input/output (I/O) port A are used as well as the USB interrupts in case the USB implementation was activated. The other interrupt vectors refer to an empty dummy interrupt service routine (ISR). This dummy ISR is
practically without function; however, it can be used in the debugging phase for seeking out unexpected (spurious) interrupts. In addition, the allocation of all interrupt vectors is important for the implementation of the FLASH memory security feature (read-out protection). For additional information, refer to the *MC68HC908JB8 Technical Data*, Motorola document order number MC68HC908JB8/D.

The reset vector refers to the start address of the application. This point of entrance is located in the C startup module.

### 3.7 C Startup Module CRTSJB8.S

The C startup module used essentially corresponds to the standard startup module from the Cosmic C Compiler package with one exception. Immediately after a reset (and after the initialization of the stack pointer), a subroutine reference was inserted to `_HC08Setup()`. This subfunction is defined in `U08MAIN.C` and performs urgent accesses to M68HC08 control registers which should be completed immediately after the reset.

### 3.8 Push Button Module U08KEY.C

Port A (PTA) of the MC68HC908JB8 has eight port bits for keyboard connection. Each of these lines can cause an interrupt. The port bits individually can be configured for use within the MC68HC908JB8 keyboard interrupt module (KBI).

The USB08 evaluation board uses three single keys which are connected to port lines PTA[4:6]. The switch noise reduction is performed via a resistor-capacitor (RC) combination at each key. This combination is made by:

- M68HC08 internal pullup resistors and a capacitor (parallel to the key)
- Hysteresis of the port A input Schmitt triggers
The KBI module of the MC68HC908JB8 greatly simplifies the scanning of the attached buttons. The software necessary for this takes only 20 lines of C code.

The conditions are created in the initialization function initKey(). First the internal pullup resistors at the port A pins are activated. A short pulse of an active H level is driven at the port A pins which accelerates the rising of the logic levels at these pins. This prevents a false reading of the initial low level on the lines.

The initialization function ends with the resetting of the status variable, KeyState, and the enabling of the keyboard interrupt.

During the manipulation of a key, the interrupt service routine isrKey() is called. At port A, the pressed key is seen as a 0 bit. The appropriate bit location is set accordingly in the status variable KeyState. By activation of a key the key status is inverted. This implementation simulates an on/off push button.

If the main program wants to know the current status of a key (on/off), it uses the access function getKey(). The number of the key (1...) will be handed over as a function argument. The function getKey() calculates a bit mask for access to an individual bit of the (internal) status variable KeyState. The return value amounts to 0, if the key is off; otherwise, a 1 is returned.

This module provides an easy way to specify the desired number of possible keys. For this purpose, two macros are used. KEY_MASK defines the used lines of port A by setting a “one” flag at the appropriate bit location. The macro KEY_FIRST defines at which bit location the first key is attached. Some examples:

<table>
<thead>
<tr>
<th>KEY_MASK</th>
<th>KEY_FIRST</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>0</td>
<td>One key at PTA[0]</td>
</tr>
<tr>
<td>0x02</td>
<td>1</td>
<td>One key at PTA[1]</td>
</tr>
<tr>
<td>0x08</td>
<td>7</td>
<td>One key at PTA[7]</td>
</tr>
<tr>
<td>0x70</td>
<td>4</td>
<td>Three keys at PTA[4..6]</td>
</tr>
<tr>
<td>0xF0</td>
<td>4</td>
<td>Four keys at PTA[4..7]</td>
</tr>
<tr>
<td>0xFF</td>
<td>0</td>
<td>Eight keys at PTA[0..7]</td>
</tr>
</tbody>
</table>

The port bits included in the key scan have to follow one after another.
**NOTE:** It has to be pointed out for completeness, that with the KBI module of the MC68HC908JB8 not only single keys but also extensive key fields in matrix arrangement can be easily scanned.

### 3.9 LED Control with U08LED.H

Controlling the three light-emitting diodes (LED) attached to port D is easy. For initialization, the data direction of the used port pins PTD[0..2] has to be switched to an output state. The initialization as well as the switching of the LEDs is performed via four macros. Therefore, a header file is enough for the realization of these functions. A special C module is not required in this case.

The LEDs are addressed, beginning with a 1. The switching on of the first LED (at PTD[0]) takes place, for example, by means of:

```c
onLED(1);
```

### 3.10 Software ADC Module U08ADC.C

Although the MC68HC908JB8 does not have an integrated ADC, it is nevertheless possible to measure analog values (and in particular resistance values) in a simple way. For this purpose, an RC combination is attached to a conventional digital port pin and the load time of the capacitor is measured. The working principle is shown in Figure 3-2.

![Figure 3-2. Measurement of Resistor Values Using a Digital Input](image-url)
First, the switch S is closed and the capacitor C uncharged. As soon as the switch S is opened, the charging procedure begins. The voltage at the input E of the microcontroller rises according to the exponential function:

\[ U_E(t) = V_{CC} (1 - e^{-t/RC}) \]

The threshold voltage at the input pin of the MC68HC908JB8, from which a level change from low to high takes place, is approximately 50 percent of the operating voltage \( V_{CC} \). Until this threshold is reached, the loading time, \( t_x \), amounts to:

\[ t_x = K \cdot R_x C \text{ with } K \approx 0.7 \text{ for } V_{CC}/2 \]

Since \( k \) represents a constant value (in the considered short time frame), a linear connection between the measured time and the product \( R_x C \) appears. Since C is constant, one can draw a direct conclusion from the charge time to the value of the resistance \( R_x \).

To determine the absolute resistance of \( R_x \), first the value of the capacity C and the constant k have to be determined. However, that is not economical (particularly with series products having a certain distribution of its value). Instead, one can accomplish a calibration cycle before the actual measurement. This calibration cycle uses a reference resistance value \( R_0 \) and determines the time \( t_0 \). The following measure cycle uses series connection \( R_x \), consisting of the reference resistance \( R_0 \) and the variable resistance \( R_1 \) (for example, \( R_X = R_0 + R_1 \)), to determine the time \( t_x \). The result can be calculated using the relationship:

\[ R_1/R_0 = (R_x - R_0)/R_0 = (t_x - t_0)/t_0 \]

The range of values of the result is between 0 and 1, if \( R_{1\text{max}} = R_0 \).

The software ADC module of the USB08 application serves three A/D channels (see Section 2. Hardware Description). Reaching the threshold voltage is sensed via the port pins PTE[0..2]. The switch function for charging/discharging the capacitor is realized by switching the port pins as outputs. In contrast to the circuit diagram (Figure 3-1), the polarity is exchanged (for instance, the point of reference is \( V_{CC} \) instead of ground) and the charge of the capacitor is made by activating
the ground potential at the resistors via the port pins PTD[3..5] (calibration cycle) and PTD[6] (measuring cycle).

Before the software of the ADC module can be used, the initialization function initSADC() has to be called. In addition, the main timer is required to run with 3 MHz (prescaler 0). The clocking of three impulses per microsecond is the basis of the time measurement in this module.

The function responsible for the A/D conversion of one channel at a time is getSADC(). The channel number (1..3) is handled as a parameter. The A/D conversion is performed in the two mentioned steps: calibration and measurement.

The desired 8-bit range for the results (values of 0…255) is a result of scaling of the A/D output. In order to keep the run time of the necessary division and multiplication operations small, the scaling function is implemented using some in-line assembler directives.

A detailed discussion of the software A/D converter used here is contained in the application note entitled Simple A/D for MCUs without Built-in A/D Converters, Motorola document order number AN477/D. This application note can be found on the World Wide Web at:

http://www.motorola.com/semiconductors/

3.11 RS232 Communication Module U08232.C

The RS232 communication module performs the sending and the receiving of data to the host PC. The RS232 implementation is one of the two possible alternatives. By the definition of the macro USE_USB_PIPE in the main module U08MAIN.C, it is possible to switch from the RS232 version to the USB version of the communication module.

Since the MC68HC908JB8 does not have a hardware serial communications interface (SCI) peripheral module, an RS232 transceiver has to be implemented by a software-based SCI module. Two general-purpose I/O pins are used as receiving and transmission lines. The timing necessary for the desired baud rate is derived from a time loop.
The module contains these three interface functions:

- **initSSCI()** — initialization of the software SCI module
- **getSSCI()** — receiving of a character
- **putSSCI()** — transmission of a character

The module initialization function initSSCI() sets the data direction registers for the output and input port. Before this setting, a 1 is written to the data register of the transmission line so that the output value of this line is the standard high state.

The receive function getSSCI() waits until the state of the receiving line changes to low. This indicates the beginning of the start bit of an arriving byte. The following eight data bits are scanned suitably, in each case in the center of the bit time. The result of this scanning is finally returned to the calling function.

The available implementation does not examine whether the arriving stop bit shows incorrect low levels (framing error). Also, an over scanning for the purpose of noise reduction does not take place.

The production of the bit rate is controlled by the module-internal function delayHalfBit(). The function is implemented with help from some in-line assembly code to ensure an accurate time performance, which can be simply changed by the user if necessary. The possible changes necessary for the adjustment to different baud rates is documented in the source text on the basis of two examples.

Adjustments regarding the port pins used as sending or receiving lines are easily possible. The module uses five macros for the control and scanning in of these pins. These macros are defined in the head of the file **U08232.C**. Almost all port pins can be used for the software serial communications interface (SCI) module by changing the bit masks and/or the port designators in these macros.

In this demo application, the moderate baud rate of 2400 baud is selected. An increase to 9600 baud is possible, but tests first must show that the application runs without problems. It has to be taken into account that the bit rate production is determined by a certain number of execution cycles by the central processor unit (CPU), which temporarily
disables any interrupts. Also, in an application with permanent interrupt use, the software SCI should run in an interrupt-controlled way. Since the receiving of a character is performed using port pin PTA7, the use of the keyboard interrupt associated with this pin would be possible for the recognition of the start bit. The timing for the scanning of the following received data bits could be made using the available main timer (TIM) of the MC68HC908JB8 as well as the sending of characters using the PTC0 pin.

3.12 USB Communication Module U08USB.C

The USB communication module U08USB.C demonstrates how data can be exchanged between the microcontroller and host PC over an USB connection. This module can be linked (alternatively to the RS232 communication module U08232.C) into the USB08 application when the macro USE_USB_PIPE in the main module U08MAIN.C is activated.

Just like RS232, the USB uses serial streams for the data communication. The substantially more complex operational sequence of the USB can be encapsulated so that the integration into existing projects is possible without problems. Therefore, the USB implementation shown here is just as simple to manage for the firmware programmer as the classical RS232 version. Only three interface functions are needed:

- initUSB() — initialization of the USB communication module
- getUSB() — receiving a character
- putUSB() — transmitting a character

The integrated USB peripheral module of the MC68HC908JB8 is controlled using some control registers within the address range of 0x020 to 0x03F; the function initUSB() takes care of the initialization of these registers. In addition, the status of the USB equipment is set to the initial status (Powered) and the two software buffers which buffer the sending and/or the receiving of data in the application are initialized.

If data should be received, the routine getUSB() is called by the application. First, this routine stays in a waiting loop until data from the
USB peripheral module arrives. As soon as the data has arrived, the next character is taken from the buffer and the read index is incremented. Since we are dealing with a ring buffer, this index, if it overflows beyond the upper buffer border, is set back to the lower buffer border (index 0).

The size of the receiving ring buffer is specified by the macro MAX_RXBUF_SIZE. The selected value has to be a power-of-two number.

The transmission function putUSB() uses another buffer area which is independent of the receive buffer including index variables for read and write access. Again, the buffer size is specified using a macro (MAX_TXBUF_SIZE). Concerning the size, the restriction on power-of-two numbers applies here as well.

The character handed over to putUSB() is placed into the buffer. If the buffer is full, the routine waits until the send buffer again is able to store data. After placing the character into the send buffer, the access index variable is updated.

Interrupt-controlled implementation of these functions is performed in the background of the microcontroller application:

- Filling of the receive buffer
- Sending of the characters in the send buffer via the USB

For this purpose, an interrupt-controlled USB handler was implemented and can be used in many other applications without any changes. The principle of operation and the places of possible or necessary modifications are described in detail in Section 4. Universal Serial Bus (USB) Interface.

### 3.13 Compiler Specific Adjustments

The source text modules were written and translated with the M68HC08 Cosmic C Compiler. This compiler supports the complete language scope available for ANSI-C. The porting of the firmware to another M68HC08 ANSI-C compiler should be possible without any problems,
because this application does not use problematic constructions, like bit fields in the source code.

In some places, using individual assembler instructions in the form of in-line assembler directives is considered useful, for example, for the interrupt enable in the main module **U08MAIN.C**. For this purpose, the Cosmic C Compiler offers the following instruction:

```c
_asm("<assembly statement >");
```

When using other compilers, similar instructions should be available. However, some small syntactic adjustments can be necessary. Beside the main module, the modules **U08232.C** and **U08ADC.C** also contain such in-line assembler constructions.

The marking of a function as an interrupt service routine is not regulated in the ANSI-C standard. For lack of a uniform regulation, the different compilers handle this necessary marking in a different way. For example, the Cosmic C Compiler uses the modification `@interrupt`:

```c
@interrupt void interrupt_handler();
```

Other compilers use “#pragma” instructions to mark interrupt functions.
Section 4. Universal Serial Bus (USB) Interface

4.1 Contents

4.2 Introduction ................................................................. .59
4.3 Characteristics of the USB08 Reference Design ................... .60
4.4 USB Basics ................................................................. .62
4.5 USB Implementation in the Reference Design ....................... .65
  4.5.1 Activation of the USB Module ..................................... .65
  4.5.2 Endpoint Configuration ............................................. .65
  4.5.3 USB Reset ............................................................ .67
4.6 Device Management with Endpoint 0 ............................... .69
  4.6.1 Enumeration .......................................................... .69
  4.6.2 Assignment of the Device Address ............................... .69
  4.6.3 Requesting Descriptors ............................................ .72
  4.6.4 Device Configuration .............................................. .74
  4.6.5 STALL Condition ................................................... .74
4.7 Data Communication via Endpoints EP1 and EP2 .................... .75
  4.7.1 Receiving Data ....................................................... .76
  4.7.2 Transmission of Data .............................................. .76
4.8 Host Interaction: Vendor ID and Product ID ....................... .78
4.9 Windows Device Driver .................................................. .78

4.2 Introduction

The universal serial bus (USB) is an interface for the connection of peripheral devices, for example, printers, scanners, keyboards, and pointing devices to a PC or a similar host.
The USB specification(1) which can be found on the World Wide Web at:

http://www.usb.org

is an industry standard, which exactly defines this bus system beginning with the electrical interface up to the higher protocol layers to guarantee the inter-operability of all the different devices. A simple way of handling is the most important requirement for USB devices from the view of the user.

Several versions of the USB specification exist. Apart from the already established release 1.1, on which the MC68HC908JB8 and the available reference design is based, the specification 2.0 was compiled in the year 2000 by the USB Implementers Forum (USB-IF). This version ensures compatibility to the version 1.1 and contains the already known speed classifications “low speed” and “full speed”. Beyond that, release 2.0 introduces a high-speed device type. First high-speed devices are expected to be established in the market by end of the year 2001.

The basic specification of the USB is supplemented by several class specifications for certain types of device classes, which can be found frequently (for example, human interface device class for keyboards, mouse pointers, etc.). Further information regarding conditions and contents of basic and class specifications can be found on the World Wide Web at:

http://www.usb.org

4.3 Characteristics of the USB08 Reference Design

The USB08 reference design shows, via a detailed example, how the integrated USB module of the Motorola microcontroller unit (MCU) MC68HC908JB8 can be used. The MCU is used for a measuring and control application and exchanges data with a PC via the USB. This reference design shows that communication to a PC using USB can be just as simple as a normal RS232 link.

1. Universal Serial Bus Specification Revision 1.1; September 23, 1998
The integrated USB module of the MC68HC908JB8 works at a data rate of 1.5 Mbit/s, thus it is defined as a low-speed USB device. For measuring and control applications, typically only small data rates are needed, and this is already realized using low-speed USB devices. A low-speed USB device ensures an information flow rate, which can be compared roughly with an RS232 link with 9600 baud. At first glance, that doesn’t seem to be much; however, the USB variant offers a set of other advantages.

While RS232 always represents point-to-point connection, USB supports a bus structure. The PC serves as bus master and several USB devices can be attached. If the connections (ports) at the PC (host) are not sufficient, USB hubs can be inserted. Hubs can be cascaded up to five levels. Each USB device is addressed by the host via a unique address. The address range supports up to 127 addresses. Thus, a whole measurement and recording system can be arranged easily using a dozen low-speed USB devices and two or three commercial hubs.

This reference design contains USB08 evaluation board firmware; therefore, this Plug & Play demo application can be evaluated immediately. Beyond that, all source code is provided in the form of C modules for the M68HC08 Cosmic C compiler. The user can use these sources as a starting point for their own USB development.

The most important modules for the USB implementation are the source code modules **U08USB.C**, **U08DESC.C**, and the header file **U08USB.H** (see Section 3. Software Module Descriptions). The USB functional description refers to these source code modules.

Administration of the USB device takes the largest portion of the referenced implementation source code. This is done via so-called standard device service requests. These transfers take place via the control endpoint 0 and are multi-level, complex communication procedures. Since the implementation of these complex functions virtually can be transferred as a block from the reference design to any other application, the practical work for the administration of a USB device should not be overestimated. Simply, the actual data communication functions (which take place using the interrupt endpoints 1 and 2) can be adapted to the concrete user application.
4.4 USB Basics

Concerning the electrical interface, the plug and the cable, as well as questions on the bus topology we refer to the *Universal Serial Bus Specification Revision 1.1* standard reference. The volume of information contained in this specification exceeds by far this manual. Also some good introductions are offered by books, for example the book by Kelm\(^1\).

**NOTE:** Some terms and procedures from the USB specification, which are important for the implementation of the reference design, will be repeated here and described briefly.

Packets form the basic modules of the USB communication in the levels above the electrical connection. Packets are atomic, for instance, they cannot be interrupted or divided into sections. The packet types shown in Table 4-1 are relevant for low-speed USB.

### Table 4-1. Low-Speed USB Packet Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Group</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETUP</td>
<td>Token</td>
<td>Starts a control transfer</td>
</tr>
<tr>
<td>IN</td>
<td>Token</td>
<td>Starts a data transfer to the host</td>
</tr>
<tr>
<td>OUT</td>
<td>Token</td>
<td>Starts a data transfer to the device</td>
</tr>
<tr>
<td>DATA0</td>
<td>Data</td>
<td>Transfers 0 to 8 data bytes</td>
</tr>
<tr>
<td>DATA1</td>
<td>Data</td>
<td>As before (toggle Data0/1)</td>
</tr>
<tr>
<td>ACK</td>
<td>Handshake</td>
<td>Information was accepted</td>
</tr>
<tr>
<td>NAK</td>
<td>Handshake</td>
<td>Busy — send again later</td>
</tr>
<tr>
<td>STALL</td>
<td>Handshake</td>
<td>Information was incorrect</td>
</tr>
</tbody>
</table>

In addition to the packets shown in Table 4-1, the bus traffic consists of further quasi-static bus conditions (reset, suspend, resume) and the “Keep-alive-EOP” (refer to *Universal Serial Bus Specification Revision 1.1* standard for more detailed information).

\(^1\) Kelm, H.J.: USB1.1; Franzis 2000
A USB transaction is a series of packets to transmit information between a host and a device. A transaction is always initiated by sending a token packet (SETUP, IN, or OUT). This packet is always sent by the host because devices cannot initiate a USB transaction. The token packet contains the address and the desired endpoint of the device.

SETUP and OUT packets are supplemented by a DATA packet from the host, which contains up to eight bytes of data. The packets DATA0 and DATA1 are always sent in an alternating sequence. This procedure is called data toggle and serves for error protection. Following the data packet, the device answers with a handshake packet. If the device could receive the data, it sends an ACK packet. If the device was not immediately ready, it sends a NAK packet signalling to the host that the packet should be sent again at a later time. In the event of an error, the device sends a STALL packet.

**SETUP Transaction**

<table>
<thead>
<tr>
<th>Host</th>
<th>Host</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETUP (ADDR,EP)</td>
<td>DATA0 (D1..D8)</td>
<td>ACK</td>
</tr>
</tbody>
</table>

**OUT Transaction**

<table>
<thead>
<tr>
<th>Host</th>
<th>Host</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT (ADDR,EP)</td>
<td>DATA0 (D1..D8)</td>
<td>ACK</td>
</tr>
</tbody>
</table>

Regarding **IN** transactions, the data packet is sent by the device and the host closes the transaction with a handshake packet.

**IN Transaction**

<table>
<thead>
<tr>
<th>Host</th>
<th>Device</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN (ADDR,EP)</td>
<td>DATA0 (D1..D8)</td>
<td>ACK</td>
</tr>
</tbody>
</table>
If the device does not hold any data ready for sending, it sends a busy handshake NAK instead of the data packets.

**IN Transaction (Device Busy)**

<table>
<thead>
<tr>
<th>Host</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN (ADDR,EP)</td>
<td>NAK</td>
</tr>
</tbody>
</table>

While the actual data communication via the stream pipes is based on simple IN and OUT transactions, the management of the device uses more complex control transfers via the control endpoint 0. These control transfers are secured using a double handshake. Control transfers consist of two or three transaction stages, like those shown here.

**2-Stage Control Transfer (No Data)**

<table>
<thead>
<tr>
<th>Setup Stage</th>
<th>Status Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETUP</td>
<td>IN,DATA0,ACK</td>
</tr>
<tr>
<td>DATA0</td>
<td>ACK</td>
</tr>
</tbody>
</table>

**3-Stage Control Transfer (Host Read)**

<table>
<thead>
<tr>
<th>Setup Stage</th>
<th>Data Stage</th>
<th>Status Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETUP</td>
<td>IN</td>
<td>OUT</td>
</tr>
<tr>
<td>DATA0,ACK</td>
<td>DATA1,ACK</td>
<td>DATA1,ACK</td>
</tr>
</tbody>
</table>

The setup stage starts with a SETUP transaction (see above). The data stage is necessary if data has to be sent from the device to the host and consists of several IN transactions. The status stage serves for the back confirmation that the information was processed correctly. In the status stage, empty DATA1 packets are sent.
4.5 USB Implementation in the Reference Design

The following paragraphs describe the implementation of the USB into the reference design.

4.5.1 Activation of the USB Module

For initialization of the MC68HC908JB8 USB module, the user program must call the initialization routine initUSB(). This routine writes all registers with the same default values, which are present after a power-on reset. Beyond that, the USB module is activated by setting the bit USBEN in the USB address register UADDR.

In addition, the bit PULLEN in USB control register UCR3 is set resulting in the internal pullup resistor at D-/PTE4 being activated if this option was selected in the source code. For this, the macro USB_IPUE must be defined as 1. Alternatively, it is possible to add an external resistor on the USB08 evaluation board (see Section 2. Hardware Description).

4.5.2 Endpoint Configuration

The integrated USB module of the MCHC908JB8 supports three endpoints. In addition to the mandatory bidirectional control endpoint EP0, two unidirectional interrupt endpoints, EP1 and EP2, are available.

The data direction is always indicated from the view of the host. An IN endpoint serves for the data transfer from the device to the host and an OUT endpoint is used for the data transfer from the host to the device.

EP1 is always configured as an IN endpoint, since the MCU sends data via this endpoint. The endpoint EP2 of the MC68HC908JB8 can be
configured as an IN or as an OUT endpoint. The reference design uses this endpoint as an OUT endpoint, to do data transfers to the MCU.

Table 4-2 provides an overview of the endpoint configuration.

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Type</th>
<th>Direction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>EP0</td>
<td>Control</td>
<td>IN/OUT</td>
<td>Device configuration</td>
</tr>
<tr>
<td>EP1</td>
<td>Interrupt</td>
<td>IN</td>
<td>Data transfer to the host</td>
</tr>
<tr>
<td>EP2</td>
<td>Interrupt</td>
<td>OUT(^{(a)})</td>
<td>Data transfer to the device</td>
</tr>
</tbody>
</table>

a. Alternatively as IN configurable, this option is not used here.

Some adjustments in the control registers of the USB module are necessary for the configuration of the endpoints. While EP0 is always active as a control endpoint, the bit ENABLE1 in control register 3 (UCR3) has to be set to activate the endpoint EP1. Likewise, the bit ENABLE2 has to be set for activation of the endpoint EP2.

![Figure 4-2. USB Control Register 3 (UCR3)](image)

However, the activation of the endpoints EP1 and EP2 takes place not in the initialization routine initUSB(), but only after the device receives a USB reset.
4.5.3 USB Reset

An USB reset is an event which is used by the USB hub to reset the attached devices to the initial state. Electrically, the reset signal is a special bus condition (single ended zero) which is initiated by the host and then passed on by the hub(s).

The USB module of the MC68HC908JB8 reacts to this either with a hardware reset or with an interrupt, dependent on the configuration selected in the CONFIG (configuration) register. Since a hardware reset (in particular during the debugging via monitor mode) is not without side effects, the generation of an interrupt is preferred here. For this purpose, the URSTD bit in the CONFIG register has to be set. It has to be considered that a write access to the CONFIG register is possible only once after each power-on reset. Therefore, write access to this control register is done in the function \_HC08Setup() in the module U08MAIN.C.

The USB reset interrupt, together with all the other USB sources of interrupt, points to a central USB interrupt vector. The USB interrupt has, apart from the software interrupt (SWI), the highest priority in the interrupt system of the MC68HC908JB8. The USB interrupt vector is stored at the vector address 0xFFFFA/0xFFFFB (see 3.6 Interrupt and Reset Vector Module VECJB8.C). In the reference design, it points to the function isrUSB(), which is responsible for the entire interrupt-controlled USB handling.

In the interrupt service routine all applicable interrupt flags are successively examined. If it is recognized that the RSTF bit in the USB interrupt register 1 (UIR1) is set, it means that a USB reset interrupt has occurred.

In this case, first the function initUSB() is called to set all control registers of the USB module into the default condition. Afterward, the two interrupt endpoints, EP1 and EP2, will be enabled to prepare for the following transfer of data. Now, the local interrupt enable bits TXD0IE, RXD0IE, TXD1IE and RXD1IE in the USB interrupt register 0 (UIR0) are set so they can react on the information sent or received from the endpoints using interrupts. In addition, the end-of-packet interrupt will be enabled with EOPIE (suspend handling).
At the end of the USB reset interrupt, the control endpoint 0 will be enabled for receiving, thus the configuration instructions which follow after the reset (device requests) are received by the USB receiver. This option will be enabled by setting the RX0E bit in the USB control register 0 (UCR0).

The USB device is now in the DEFAULT state. That means the device is attached to the bus and is supplied with current. In addition, it has received a USB reset and reacts to instructions with the default address zero. For further information, refer to Universal Serial Bus Specification Revision 1.1, Chapter 9.1 — USB Device States.

After these fundamental initialization steps, further setup steps summarized under the term of enumeration follow.
4.6 Device Management with Endpoint 0

4.6.1 Enumeration

After the basic initialization described in the previous paragraphs, the USB module is now able to react with an interrupt on packets which are addressed to the control endpoint EP0. Now, the process of the enumeration (that is, the configuration and integration into the system) at the USB is continued.

4.6.2 Assignment of the Device Address

In the following step, the host assigns a unique USB address to the device, which is located in the range from 1 to 127. For this purpose, the host sends a SET_ADDRESS standard device request. Standard device requests are always served by the EP0. These transfers are control transfers which are implemented for the EP0 only (the other endpoints are used exclusively for the data communication by means of interrupt transfers).

USB device requests are started, by the host sending a SETUP packet. The MCU receives this information and generates an EP0 receive interrupt. In the interrupt service routine isrUSB(), the interrupt is identified by the controller on the basis of the RXD0F flag in the USB interrupt register 1 (UIR1).

![USB Interrupt Register 1 (UIR1)](image)

Since a receive interrupt could be initiated by an OUT packet for the endpoint EP0, it must be determined whether the received information
refers to a SETUP or an OUT packet. Therefore, the flag SETUP in the USB status register 0 (USR0) has to be examined. If this flag is set, the last token received by EP0 was a SETUP token and the interrupt routine is branched to the function for handling of SETUP transactions, handleSETUP().

In the USB status register 0 (USR0) the function handleSETUP() finds not only the type of the transaction (SETUP or OUT), but can also determine the number of received bytes. In the case of a SETUP transaction, the data length is always defined as eight bytes. Therefore, USR0 should contain the value 0×48.

The eight bytes received by endpoint 0 are available in the eight registers UE0D0–UE0D7 and will be transferred into the buffer variable SetupBuffer. This variable of the type setup_buffer is explained in the following excerpt from U08USB.H.

```
// Structure of Setup Packet sent during
// SETUP Stage of Standard Device Requests
// according to USB1.1 spec page 183
typedef struct {
    uchar bmRequestType; // Characteristics (Dir,Type,Recipient)
    uchar bRequest;     // Standard Request Code
    iword wValue;       // Value Field
    iword wIndex;       // Index or Offset Field
    iword wLength;      // No. of Bytes to transfer (Data Stage)
} setup_buffer;
```

The field bmRequestType must contain the value 0 in bits 5 and 6; otherwise, it is not a standard device request.
The type of standard request is coded in the field bRequest. For continuation of the enumeration, the host should send the standard device request SET_ADDRESS and the handleSETUP() routine should branch to the function setAddress().

The function first validates the contents of the fields of SetupBuffer. In the case of an error, a STALL handshake is initiated to give the host problem feedback.

Before the device address in the field wValue is finally accepted, the MCU has to prepare the transmission of a receive acknowledgment. This acknowledgement still is completed using the old device address zero. An additional safety feature is the mandatory control transfer status stage.

The handshake takes place via a telegram with a data length of zero, which is requested by the host by means of an IN transaction. For this purpose, the length TP0SIZx is defined as zero and the TX0E bit is set in the USB control register 0 (UCR0). This enables the transmitter of the endpoint 0.

This handshake transaction always uses a DATA1 packet; therefore, T0SEQ is set.

The routine setAddress() now returns to handleSETUP() where the receive interrupt for EP0 is re-enabled. Finally, the MCU terminates the interrupt service routine isrUSB().

The device address is still located in the SetupBuffer. The service routine isrUSB() is again activated by a transmit interrupt for EP0. After decoding, if it was determined that the cause of the interrupt was an IN
transaction, the function handleIN() is called. In the field bRequest of the SetupBuffer, SET_ADDRESS is still contained as the current standard request type. The function handleIN() now reacts by transferring the device address from the buffer to the USB address register (UADDR).

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read:</td>
<td>USBEN</td>
<td>UADD6</td>
<td>UADD5</td>
<td>UADD4</td>
<td>UADD3</td>
<td>UADD2</td>
<td>UADD1</td>
</tr>
<tr>
<td>Write:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Reset:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 4-8. USB Address Register (UADDR)**

The device is now turned into an ADDRESSED status. An exception would be if the transmitted device address was zero again. This case would mean an uncoupling of the device from the USB system.

With the completion of the IN transaction as a status stage, the control transfer for the treatment of the standard device request SET_ADDRESS is finished. The completion is noted in the SetupBuffer by setting the flag REQUEST_COMPLETE.

4.6.3 Requesting Descriptors

Further in the process of enumeration, the host will request the configuration of the device. For this purpose, descriptors which contain information about the status and one or more possible configurations, are made available by the device. The host loads these descriptors, selects a suitable driver, and forces the device to take a certain configuration. As a consequence, the device will be ready for use and will be able to transfer data via the interrupt endpoints EP1 and EP2.

Sending descriptor information takes place during a standard device request. The request is of the type GET_DESCRIPTOR and in principle is handled the same way as that for the standard device request, SET_ADDRESS, described above.
Within the setup stage, the function handleSETUP() is called. The request type is recognized as GET_DESCRIPTOR and branched to the subfunction getDescriptor(). There, it is determined if a device configuration or string descriptor was requested. According to this requirement, a pointer will be set on the respective data source. After examination of the data length, this data will be written into the USB endpoint 0 send registers UE0D0–UE0D7.

The number of bytes is limited (at least for low-speed USB devices) to eight bytes per DATA packet. If more data has to be transferred, the device has to divide the data into blocks of eight bytes and distribute these portions using several sequential IN transactions. The last data block is identified by the host due to a length smaller than eight bytes. That means, if the length of the transmitted data amounts to an integral multiple of eight, there has to be an additional empty IN transaction (data length zero).

The data length and the transmitter release bit TX0E are inserted into the USB control register 0 (UCR0) (see SET_ADDRESS), then the data is available for the host to pick up.

If there is only one packet (length smaller eight), the GET_DESCRIPTOR standard device request is marked as complete (REQUEST_COMPLETE).

If several packets have to be sent, the send buffer is filled up again by the routine handleIN() during the following EP0 transmit interrupt, until all data is sent.

Sequential IN transactions are served with interchanging data packets. The interchanging between DATA0 and DATA1 packets is called a data toggle and serves for error protection.

Detailed information about the descriptors used in the reference design are contained in Appendix B. USB08 Descriptors.
4.6.4 Device Configuration

After the host processes all descriptors claimed by the device, it will set up the device with a SET_CONFIGURATION standard device request. A device can have several configurations (for example, with different power options, resolutions, or speed options). The configuration characteristics supported by the device are coded in the device descriptors.

The reference design is limited to the simplest case with only one possible configuration. The SET_CONFIGURATION request is passed on by the routine isrUSB() to handleSETUP(), after which branches to setConfiguration() take place. The configuration specified by the host is coded in the field wValue.lo of the structure SetupBuffer. If this field is larger than zero, the USB08 is ready to be put into operation. For this purpose, the transmitter of endpoint 1 and the receiver of endpoint 2 will be enabled and the internal status of the device changed to CONFIGURED.

In reverse, the host is able to return the device to the status ADDRESSED by transmission of a SET_CONFIGURATION instruction with the value 0.

For the status stage of this control transfer, the routine ends by preparing to send an empty DATA1 packet. This is completed by the following IN transaction.

4.6.5 STALL Condition

If the device discovers an error during communication via the USB which requires the involvement of the host, the device sends a STALL packet in place of the usual handshake packets ACK (ready/OK) or NAK (not ready).
To force the device not to send further STALL packets after the recovery of the error, the host can use the standard device request CLEAR_FEATURE. The responsible standard request handler clearFeature():

- Hands over the code for the endpoint concerned (0x81 for EP1, 0x02 for EP2) to wIndex
- Writes ENDPOINT_HALT into wValue
- Forces the deletion of the STALL condition for the endpoint 1 or 2

A STALL condition of the EP0 resulting from an incorrect SETUP request is reset automatically by the next arriving SETUP token.

4.7 Data Communication via Endpoints EP1 and EP2

The transmission of user data from or to the USB device takes place via the endpoints 1 and 2. EP1 is an endpoint of the type IN and serves for sending of information to the host. EP2 possesses the direction OUT and is used by the device to receive data from the host.

All data traffic of the payload endpoints EP1 and EP2, as well as the administrative traffic of the endpoint EP0, leads to interrupt handling via the interrupt service routine isrUSB(). The USB interrupt register 1 (UIR1) contains information about the exact source of the USB interrupt. If the flag TXD1F is set, a transmit complete interrupt was indicated by endpoint 1. If RXD2F is set, an endpoint 2 receive interrupt is pending.

```
<table>
<thead>
<tr>
<th>Bit 7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read:</td>
<td>EOPF</td>
<td>RSTF</td>
<td>TXD2F</td>
<td>RXD2F</td>
<td>TXD1F</td>
<td>RESUMF</td>
<td>TXD0F</td>
</tr>
<tr>
<td>Write:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reset:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Figure 4-9. USB Interrupt Register 1 (UIR1)
4.7.1 Receiving Data

If data for endpoint 2 arrives, the interrupt handler calls handleOUT2(). The number of received bytes is noted in bits RPSIZ3–RPSIZ0 with the allowed values in the range of 0–8.

![Figure 4-10. USB Status Register 1 (USR1)](image)

The received data bytes are transferred from the USB endpoint data registers UE2D0–UE2D7 to the buffer RxBuffer. This is a software ring buffer, which can be filled by the interrupt service routine isrUSB() and be read out by means of getUSB().

If the ring buffer is full, handleOUT2() waits until RxBuffer is able to store data again. In this case, the USB module answers further transmission attempts of the host with a NAK handshake.

**NOTE:** In a real application, you should not leave the buffer unserviced over a longer period of time.

4.7.2 Transmission of Data

The transmit data for EP1 is placed by the user program, via the function putUSB(), into the send buffer TxBuffer. This is (just like RxBuffer) a ring buffer, which blocks the application as soon as the buffer is about to overflow.

The host polls all interrupt end points cyclically, taking into account a guaranteed maximum latency time. That polling interval can be specified in the endpoint descriptor. For low-speed USB devices with interrupt endpoints the shortest specified polling interval is 10 ms. That means,
after 10 ms the host asks whether further data has to be fetched from endpoint 1 or not.

**NOTE:**  *In practice, the host uses only intervals of $2^n$ ms, the demanded 10 ms is then rounded to 8 ms.*

The USB interrupt routine is called cyclically and branches to the handler handleIN1. From there, data is taken from the ring buffer TxBuffer (if available) and transferred into the USB endpoint 1 data registers UE1D0–UE1D7. Subsequently, the number of bytes which should be sent is registered in the fields TP1SIZ3–TP1SIZ0. If no data is in the buffer, this number is registered as 0. The T1SEQ bit is inverted to switch between DATA0 and DATA1 packets (data toggle). Finally, by setting the bit TX1E in the USB control register 1 (UCR1), the transmission of the data is enabled.

The operation mode selected here is based on a continuous data stream. If there is no transmit data in the buffer, the device will send data packets with zero byte contents. If this condition continues for a longer time, sending of empty data packets means a waste of bus bandwidth. If this turns out as critical, a change of the operation mode is recommended. Alternatively, it is possible to disable TX1E, as long as no data is present in the buffer. Then, the endpoint answers a polling only with a NAK packet and does not occupy any additional bandwidth by sending an empty data packet.
4.8 Host Interaction: Vendor ID and Product ID

Two identifiers are used to mark a USB device and make it possible for the host to assign a suitable driver: The vendor ID and the product ID. Both IDs are registered in the device descriptor of the USB equipment.

1. The vendor ID (VID) marks the manufacturer. Normally, vendor IDs are assigned by the USB Implementers Forum. The requestor is charged for this registration.

2. The product ID is (just like the VID) a 16-bit number. The PID marks a certain product. The allocation is done by the manufacturer of the device. Unlike the VID, for the PID there are no administrative restrictions from the USB Implementers Forum.

The USB08 reference design uses the registered vendor ID of the manufacturer MCT Elektronikladen, which is 0x0C70. The product ID for the demo application is 0x0000.

To avoid collisions and complications, every type of device is requested to have a unique vendor ID and/or product ID. Devices which have fantasy IDs cannot be used as that would lead to the immediate collapse of the compatibility of different devices at the USB.

Registered users of the USB08 evaluation board can receive their own PID out of the PID pool of the VID 0x0C70, which is exclusively allocated to the user. With these unique VID/PID combinations, the user can develop and sell USB equipment without having to request his own vendor ID beforehand.

Contact MCT Elektronikladen for additional information on obtaining a unique USB08 PID. Refer to http://www.hc08web.de/usb08

4.9 Windows Device Driver

Both VID and PID represent the search criteria for the suitable Windows device driver. The link between the driver and VID/PID is done by a *.inf file. To deliver to the operating system the suitable driver for the USB equipment, the manufacturer of the USB device has to provide only a data medium, on which (preferably in the root directory) the suitable *.inf
file and the driver file are specified. The Windows Hardware Assistant copies these two files into the appropriate Windows directory and updates the driver data base as necessary. Then, during the following "Plug Event," the Windows operating system finds the driver immediately in this data base.

With the help of Microsoft SDK it would be possible for a USB device manufacturer to develop the necessary kernel mode drivers for themselves. However, this kind of programming task requires a deep knowledge of the structure and working principles of the Windows driver modules. For those engineers, who occasionally do some programming work on a PC, it is urgently recommended not to try such a task.

A possible workaround could be the operation of the USB device using a Windows standard driver for human interface devices (HID). The device class HID summarizes PC input devices, for example, a keyboard attached to the PC. It is possible to camouflage the measuring data as an HID input packet and tunnel through the HID driver. The advantage of this approach is that a kernel mode driver doesn’t have to be developed. Instead you can program your own PC application on top of the existing operating system driver.

However, there are two major disadvantages of the HID method. They are:

1. First, the complexity of the USB handling increases, particularly on the firmware side. There are additional procedures, protocols, and descriptors to implement. Definitions of these additions are not a part of the original USB specification, but are in an HID specification, which can be loaded from the USB Web site:

   http://www.usb.org

2. Secondly and by far more devastating, is the circumstance that half a dozen implementation variants exist. With each version of Windows and each new service pack or release, the user risks that some details of the USB drivers have changed. In this case, the expenses for testing may be quite high.
The USB08 reference design uses a third possibility, the universal USB device driver (USBIO) from the company Thesycon. This third-party USB driver is professionally maintained and updated as soon as new operating system conditions occur. The USBIO driver is used for the USB08 reference design as a free-of-charge adapted "Light EL" version.

For detailed documentation, components, and demo programs refer to:

- Appendix E. Universal USB Device Driver (USBIO)
- World Wide Web at http://www.thesycon.de
## Appendix A. Supported Standard Device Requests

Definition of the supported standard device requests are given here.

<table>
<thead>
<tr>
<th>Standard Device Request</th>
<th>ID</th>
<th>Supported Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET_ADDRESS</td>
<td>5</td>
<td>Any</td>
</tr>
<tr>
<td>GET_DESCRIPTOR</td>
<td>6</td>
<td>DEVICE_CONFIGURATION STRING</td>
</tr>
<tr>
<td>SET_CONFIGURATION</td>
<td>9</td>
<td>0/1</td>
</tr>
<tr>
<td>CLEAR_FEATURE</td>
<td>1</td>
<td>ENDPOINT_HALT</td>
</tr>
</tbody>
</table>

Freescale Semiconductor, Inc.
For More Information On This Product, Go to: www.freescale.com
Appendix B. USB08 Descriptors

B.1 Contents

B.2 Introduction ......................................................... 83
B.3 Device Descriptor ................................................... 84
B.4 Configuration Descriptor .......................................... 84
B.5 Interface Descriptor ................................................. 85
B.6 Endpoint 1 Descriptor .............................................. 85
B.7 Endpoint 2 Descriptor .............................................. 85
B.8 String Descriptors .................................................. 86

B.2 Introduction

This appendix defines the USB08 descriptors.
B.3 Device Descriptor

```c
const device_descriptor DeviceDesc =
{                   // Size of this Descriptor in Bytes
    sizeof(device_descriptor),
    DTDEVICE,          // Descriptor Type (=1)
    {0x10, 0x01},       // USB Spec Release Number in BCD = 1.10
    0,                  // Device Class Code (none)
    0,                  // Device Subclass Code (none)
    0,                  // Device Protocol Code (none)
    8,                  // Maximum Packet Size for EP0
    {0x70, 0x0c},       // Vendor ID = MCT Elektronikladen
    {0x00, 0x00},       // Product ID = Generic Demo
    {0x00, 0x01},       // Device Release Number in BCD
    1,                  // Index of String Desc for Manufacturer
    2,                  // Index of String Desc for Product
    0,                  // Index of String Desc for SerNo
    1                   // Number of possible Configurations
}; // end of DeviceDesc
```

B.4 Configuration Descriptor

```c
const configuration_descriptor ConfigDesc =
{                   // Size of this Descriptor in Bytes
    sizeof(configuration_descriptor),
    DT_CONFIGURATION, // Descriptor Type (=2)
    {sizeof(configuration_descriptor) + sizeof(interface_descriptor) +
      sizeof(endpoint_descriptor) + sizeof(endpoint_descriptor),
    0x00},             // Total Length of Data for this Conf
    1,                  // No of Interfaces supported by this Conf
    1,                  // Designator Value for *this* Configuration
    0,                  // Index of String Desc for this Conf
    0xc0,               // Self-powered, no Remote-Wakeup
    0                   // Max. Power Consumption in this Conf (*2mA)
}; // end of ConfigDesc
```
B.5 Interface Descriptor

```c
const interface_descriptor InterfaceDesc =
{                   // Size of this Descriptor in Bytes
    sizeof(interface_descriptor),
    DT_INTERFACE,       // Descriptor Type (=4)
    0,                  // Number of *this* Interface (0..)
    0,                  // Alternative for this Interface (if any)
    2,                  // No of EPs used by this IF (excl. EP0)
    0xff,               // IF Class Code (0xff = Vendor specific)
    0x01,               // Interface Subclass Code
    0xff,               // IF Protocol Code  (0xff = Vendor specific)
    0                   // Index of String Desc for this Interface
}; // end of InterfaceDesc
```

B.6 Endpoint 1 Descriptor

```c
const endpoint_descriptor Endpoint1Desc =
{                   // Size of this Descriptor in Bytes
    sizeof(endpoint_descriptor),
    DT_ENDPOINT,        // Descriptor Type (=5)
    0x81,               // Endpoint Address (EP1, IN)
    0x03,               // Interrupt
    {0x08, 0x00},       // Max. Endpoint Packet Size
    10                  // Polling Interval (Interrupt) in ms
}; // end of Endpoint1Desc
```

B.7 Endpoint 2 Descriptor

```c
const endpoint_descriptor Endpoint2Desc =
{                   // Size of this Descriptor in Bytes
    sizeof(endpoint_descriptor),
    DT_ENDPOINT,        // Descriptor Type (=5)
    0x02,               // Endpoint Address (EP2, OUT)
    0x03,               // Interrupt
    {0x08, 0x00},       // Max. Endpoint Packet Size
    10                  // Polling Interval (Interrupt) in ms
}; // end of Endpoint2Desc
```
B.8 String Descriptors

// Language IDs
//----------------
#define SD0LEN 4
//----------------

const uchar String0Desc[SD0LEN] = {
    // Size, Type
    SD0LEN, DT_STRING,
    // LangID Codes
    0x09, 0x04
};

// Manufacturer String
//-----------------------
#define SD1LEN sizeof("MCT Elektronikladen")*2
//-----------------------
const uchar String1Desc[SD1LEN] = {
    // Size, Type
    SD1LEN, DT_STRING,
    // Unicode String
    'M', 0,
    'C', 0,
    'T', 0,
    ' ', 0,
    'E', 0,
    'l', 0,
    'e', 0,
    'k', 0,
    't', 0,
    'r', 0,
    'o', 0,
    'n', 0
};
// Product String
//-----------------------------------------------
#define SD2LEN sizeof("USB08 Evaluation Board")*2
//-----------------------------------------------
const uchar String2Desc[SD2LEN] = {
    // Size, Type
    SD2LEN, DT_STRING,
    // Unicode String
    'U', 0,
    'S', 0,
    'B', 0,
    '0', 0,
    '8', 0,
    ' ', 0,
    'E', 0,
    'v', 0,
    'a', 0,
    'l', 0,
    'u', 0,
    'a', 0,
    't', 0,
    'i', 0,
    'o', 0,
    'n', 0,
    ' ', 0,
    'B', 0,
    'o', 0,
    'a', 0,
    'r', 0,
    'd', 0
};

// Table of String Descriptors
//
uchar * const StringDescTable[] = {
    String0Desc,
    String1Desc,
    String2Desc
};
Appendix C. Source Code Files

C.1 Contents

HC908JB8.H ................................................................. .90
U08USB.H ................................................................. .93
U08232.H ................................................................. .96
U08LED.H ................................................................. .96
U08MAIN.C ............................................................... .97
U08DESC.C ............................................................... 100
U08USB.C ................................................................. 104
U08232.C ................................................................. 113
U08KEY.C ................................................................. 116
U08ADC.C ................................................................. 117
VECJB8.C ................................................................. 119
CRTSJB8.S ............................................................... 120
USB08.LKF ............................................................... 121
BUILD.BAT ............................................................... 121
USB08.MAP ............................................................... 122
USB08.S19 ................................................................. 125
HC908JB8.H

#ifndef __HC08_H
#define __HC08_H1

// Control Register Definitions for HC908JB8 ---------------------------------
#define _IO_BASE 0
#define _P(off) *(unsigned char volatile *)(_IO_BASE + off)
#define _LP(off) *(unsigned short volatile *)(_IO_BASE + off)
#define PTA _P(0x00)
#define PTB _P(0x01)
#define PTC _P(0x02)
#define PTD _P(0x03)
#define DDRA _P(0x04)
#define DDRB _P(0x05)
#define DDRC _P(0x06)
#define DDRD _P(0x07)
#define PTE _P(0x08)
#define DDRE _P(0x09)
#define TSC _P(0x0A)
//not implemented (0x0B)
#define TCNTH _P(0x0C)
#define TCNTL _P(0x0D)
#define TMODH _P(0x0E)
#define TMODL _P(0x0F)
#define TSC0 _P(0x10)
#define TCH0H _P(0x11)
#define TCH0L _P(0x12)
#define TSC1 _P(0x13)
#define TCH1H _P(0x14)
#define TCH1L _P(0x15)
#define KBSCR _P(0x16)
#define KBIER _P(0x17)
#define UIR2 _P(0x18)
#define UC02 _P(0x19)
#define UC03 _P(0x1A)
#define UC04 _P(0x1B)
#define IOC0 _P(0x1C)
#define ISCR _P(0x1D)
#define CONFIG _P(0x1E)
#define UE0D0 _P(0x20)
#define UE0D1 _P(0x21)
#define UE0D2 _P(0x22)
#define UE0D3 _P(0x23)
#define UE0D4 _P(0x24)
#define UE0D5 _P(0x25)
#define UE0D6 _P(0x26)
#define UE0D7 _P(0x27)
#define UE1D0 _P(0x28)
#define UE1D1 _P(0x29)
#define UE1D2 _P(0x2A)
#define UE1D3 _P(0x2B)
#define UE1D4 _P(0x2C)
#define UE1D5 _P(0x2D)
#define UE1D6 _P(0x2E)
#define UE1D7 _P(0x2F)
#define UE2D0 _P(0x30)
#define UE2D1 _P(0x31)
#define UE2D2 _P(0x32)
#define UE2D3 _P(0x33)
#define UE2D4 _P(0x34)
#define UE2D5 _P(0x35)
#define UE2D6 _P(0x36)
#define UE2D7 _P(0x37)
#define UADDR _P(0x38)
#define UIR0 _P(0x39)
#define UIR1 _P(0x3A)
#define UCR0 _P(0x3B)
#define UCR1 _P(0x3C)
#define USR0 _P(0x3D)
#define USR1 _P(0x3E)
//not implemented (0x3F)

// 16-Bit Registers:
#define TCNT _LP(0x0C)
#define TMOD _LP(0x0E)
#define TCH0 _LP(0x11)
#define TCH1 _LP(0x14)

//-- Bit Mask Definitions ----------------------------------------------------
// Bits in UADDR:
#define BM_USBEN 0x80 // USB Module Enable

// Bits in UIR0:
#define BM_EOLIE 0x80 // End-of-Packet Detect Interrupt Enable
#define BM_RXD2IE 0x10 // EP2 Rx Interrupt Enable
#define BM_TXD1IE 0x08 // EP1 Tx Interrupt Enable
#define BM_TXD0IE 0x02 // EP0 Tx Interrupt Enable
#define BM_RXD0IE 0x01 // EP0 Rx Interrupt Enable

// Bits in UIR1:
#define BM_EOLF 0x80 // End-of-Packet Detect Flag
#define BM_RSTF 0x40 // Clear Reset Indicator Bit
#define BM_RXD2F 0x10 // EP2 Data Receive Flag
#define BM_TXD1F 0x08 // EP1 Data Transmit complete Flag
#define BM_TXD0F 0x02 // EP0 Data Transmit complete Flag
#define BM_RXD0F 0x01 // EP0 Data Receive Flag

// Bits in UIR2:
#define BM_EOLFR 0x80 // End-of-Packet Flag Reset
#define BM_RSTFR 0x40 // Clear Reset Indicator Bit
#define BM_RXD2FR 0x10 // EP2 Receive Flag Reset
#define BM_TXD1FR 0x08 // EP1 Transmit complete Flag Reset
#define BM_TXD0FR 0x02 // EP0 Transmit complete Flag Reset
#define BM_RXD0FR 0x01 // EP0 Receive Flag Reset

// Bits in UCR0:
Source Code Files

#define BM_T0SEQ 0x80  // EP0 Tx Sequence Bit (DATA0/1)
#define BM_TX0E 0x20   // EP0 Tx Enable
#define BM_RX0E 0x10   // EP0 Rx Enable
//@define BM_TP0SIZ 0x0f // EP0 Tx Data Packet Size

// Bits in UCR1:
#define BM_T1SEQ 0x80  // EP1 Tx Sequence Bit (DATA0/1)
#define BM_STALL1 0x40 // EP1 Force Stall Bit
#define BM_TX1E 0x20   // EP1 Tx Enable
//@define BM_TP1SIZ 0x0f // EP1 Tx Data Packet Size

// Bits in UCR2:
#define BM_STALL2 0x40 // EP2 Force Stall Bit
#define BM_RX2E 0x10   // EP2 Rx Enable

// Bits in UCR3:
#define BM_TX1STR 0x40 // Clear EP0 Transmit-1st Flag
#define BM_OSTALL0 0x20 // EP0 force STALL Bit for OUT Token
#define BM_ISTALL0 0x10 // EP0 force STALL Bit for IN Token
#define BM_PULLEN 0x04  // Pull-up Enable
#define BM_ENABLE2 0x02  // EP2 Enable
#define BM_ENABLE1 0x01  // EP1 Enable

// Bits in USR0:
//@define BM_R0SEQ 0x80 // EP0 Rx Sequence Bit (DATA0/1)
#define BM_SETUP 0x40 // Setup Token Detect Bit
//@define BM_RP0SIZ 0x0f // EP0 Rx Data Packet Size

// Bits in USR1:
//@define BM_R2SEQ 0x80 // EP2 Rx Sequence Bit (DATA0/1)
#define BM_RP2SIZ 0x0f // EP2 Rx Data Packet Size

//----------------------------------------------------------------------------
#endif
U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files

U08USB.H

.isHidden = false; 

Source Code Files
---

// Standard Interface Descriptor
// according to USB1.1 spec page 202
//
typedef struct {
    uchar bLength;       // Size of this Descriptor in Bytes
    uchar bDescriptorType; // Descriptor Type (=4)
    uchar bInterfaceNumber; // Number of *this* Interface (0..)
    uchar bAlternateSetting; // Alternative for this Interface (if any)
    uchar bNumEndpoints; // No of EPs used by this IF (excl. EP0)
    uchar bInterfaceClass; // Interface Class Code
    uchar bInterfaceSubClass; // Interface Subclass Code
    uchar bInterfaceProtocol; // Interface Protocol Code
    uchar iInterface; // Index of String Desc for this Interface
} interface_descriptor;

//----------------------------------------------------------------------------

// Standard Endpoint Descriptor
// according to USB1.1 spec page 203
//
typedef struct {
    uchar bLength; // Size of this Descriptor in Bytes
    uchar bDescriptorType; // Descriptor Type (=5)
    uchar bEndpointAddress; // Endpoint Address (Number + Direction)
    uchar bmAttributes; // Endpoint Attributes (Transfer Type)
    iword wMaxPacketSize; // Max. Endpoint Packet Size
    uchar bInterval; // Polling Interval (Interrupt) in ms
} endpoint_descriptor;

//----------------------------------------------------------------------------

// Structure of Setup Packet sent during
// SETUP Stage of Standard Device Requests
// according to USB1.1 spec page 183
//
typedef struct {
    uchar bmRequestType; // Characteristics (Direction,Type,Recipient)
    uchar bRequest; // Standard Request Code
    iword wValue; // Value Field
    iword wIndex; // Index or Offset Field
    iword wLength; // Number of Bytes to transfer (Data Stage)
} setup_buffer;

//----------------------------------------------------------------------------

// USB Status Codes
//
#define US_ATTACHED 0x00 // (not used here)
#define US_POWERED 0x01
#define US_DEFAULT 0x02
#define US_ADDRESSED 0x03
#define US_CONFIGURED 0x04
#define US_SUSPENDED 0x80
---
// USB Standard Device Request Codes
// according to USB1.1 spec page 187
//
#define GET_STATUS 0x00
#define CLEAR_FEATURE 0x01
#define SET_FEATURE 0x03
#define SET_ADDRESS 0x05
#define GET_DESCRIPTOR 0x06
#define SET_DESCRIPTOR 0x07    // optional
#define GET_CONFIGURATION 0x08
#define SET_CONFIGURATION 0x09
#define GET_INTERFACE 0x0a
#define SET_INTERFACE 0x0b
#define SYNCH_FRAME 0x0c    // optional
#define REQUEST_COMPLETE 0xff  // not part of the Standard - just
                        // a Flag to indicate that the recent
                        // Request has been finished

// Descriptor Types
// according to USB1.1 spec page 187
//
#define DT_DEVICE     1
#define DT_CONFIGURATION 2
#define DT_STRING     3
#define DT_INTERFACE  4
#define DT_ENDPOINT   5

// Function Prototypes
//
void initUSB();
uchar getUSB();
void putUSB(uchar c);
@interrupt void isrUSB();

//----------------------------------------------------------------------------
// Usb Standard Device Request Codes
// according to USB1.1 spec page 187
//
#define GET_STATUS 0x00
#define CLEAR_FEATURE 0x01
#define SET_FEATURE 0x03
#define SET_ADDRESS 0x05
#define GET_DESCRIPTOR 0x06
#define SET_DESCRIPTOR 0x07    // optional
#define GET_CONFIGURATION 0x08
#define SET_CONFIGURATION 0x09
#define GET_INTERFACE 0x0a
#define SET_INTERFACE 0x0b
#define SYNCH_FRAME 0x0c    // optional
#define REQUEST_COMPLETE 0xff  // not part of the Standard - just
                        // a Flag to indicate that the recent
                        // Request has been finished

// Descriptor Types
// according to USB1.1 spec page 187
//
#define DT_DEVICE     1
#define DT_CONFIGURATION 2
#define DT_STRING     3
#define DT_INTERFACE  4
#define DT_ENDPOINT   5

// Function Prototypes
//
void initUSB();
uchar getUSB();
void putUSB(uchar c);
@interrupt void isrUSB();
U08232.H

//------------------------------------------------------------------------------
// File: U08_232.H
// Func: Header File for RS232 Module of USB08 Demo App
// Ver.: 1.00
// Auth: (C)2000,2001 by Oliver Thamm
//       MCT Elektronikladen GbR
// http://hc08web.de/usb08
// Rem.: View/Edit this File with TAB-Size=4
//------------------------------------------------------------------------------

//-- Function Prototypes

void initSSCI();
void putSSCI(char c);
char getSSCI();

//------------------------------------------------------------------------------

U08LED.H

//------------------------------------------------------------------------------
// File: U08LED.H
// Func: LED Functions for USB08
// Auth: (C)2000 by MCT Elektronikladen GbR, Oliver Thamm
// http://www.elektronikladen.de/mct
//------------------------------------------------------------------------------

// No code in this Module - just Macros!

#define initLED() (DDRD |= 0x07)
#define toggleLED(x) (PTD ^= (1 << (x-1)))
#define offLED(x) (PTD |= (1 << (x-1)))
#define onLED(x) (PTD &= ~(1 << (x-1)))

//------------------------------------------------------------------------------
U08MAIN.C

//============================================================================
// File: U08MAIN.C
// Func: Main Module for USB08 Demo Application
// Ver.: 1.00
// Auth: (C)2000,2001 by Oliver Thamm, MCT Elektronikladen GbR
// http://hc08web.de/usb08
// Make: Build the project using U08MAIN.C, U08KEY.C,
// and VECJB8.C, use CRTSJB8.S as C-Startup Module
// Rem.: View/Edit this File with TAB-Size=4
//============================================================================

//-- Select Interface! -------------------------------------------------------
#define USE_USB_PIPE // by defining or NOT defining this
// label before compiling, you can
// select the communication interface
// (if defined => USB, if not => RS232)

//-- Includes ----------------------------------------------------------------
#include "hc08jb8.h" // HC908JB8 Register and Bitmap Definitions
#include "u08key.h" // Keyboard Module Header File
#include "u08led.h" // LED Module Header File (just Macros)
#include "u08adc.h" // Soft ADC Module Header File

//-- Compiler-dependent Stuff ------------------------------------------------
#define cli() _asm("cli")
#define nop() _asm("nop")

//-- Data Type Definitions ---------------------------------------------------
typedef unsigned char uchar;

//-- Code Starts here --------------------------------------------------------
#ifdef USE_USB_PIPE
#include "u08usb.c" // use USB implementation
#define initPipe initUSB
#define getPipe getUSB
#define putPipe putUSB
#else
#include "u08232.c" // use RS232 implementation
#define initPipe initSSCI
#define getPipe getSSCI
#define putPipe putSSCI
#endif
@interrupt void isrUSB() { }
Source Code Files

// Things that should be done immediately after Reset
// (this is called by the C-Startup Module)
void _HC08Setup() {
    CONFIG = 0x21;  // USB Reset Disable, COP Disable
    TSC = 0x00;     // clear TSTOP, Prescaler=0
}

// Dummy Interrupt Handler
// Place a Breakpoint here in case you are looking for spurious Interrupts
@interrupt void isrDummy() {
    nop();  // just for Debugging
}

void main() {
    uchar n, a;
    uchar io_buffer[8];
    uchar adc[3];

    initPipe();  // init RS232 or USB Pipe
    initLED();   // init LED Output
    initKey();   // init Key Input
    initSADC();  // init Soft ADC
    cli();
    a = 0;
    while(1) {
        // update ADC results (1 out of 3 at one time)
        adc[a] = getSADC(a+1);
        if(++a==3) a=0;

        // get data from input pipe
        n=0;
        do {
            io_buffer[n++] = getPipe();
        } while(n<8);

        // process input data
        if(io_buffer[0]==0) offLED(1);
        else onLED(1);
        if(io_buffer[1]==0) offLED(2);
        else onLED(2);
        if(io_buffer[2]==0) offLED(3);
        else onLED(3);
    }
}
// send data to output pipe

io_buffer[0] = getKey(1);
io_buffer[1] = getKey(2);
io_buffer[2] = getKey(3);
io_buffer[3] = adc[0];
io_buffer[4] = adc[1];
io_buffer[5] = adc[2];

n=0;
do {
    putPipe(io_buffer[n++]);
} while(n<8);

//-----------------------------------------------------------------------------------
const device_descriptor 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
    0, // Device Class Code (none)
    0, // Device Subclass Code (none)
    0, // Device Protocol Code (none)
    8, // Maximum Packet Size for EP0
    (0x70, 0x0c), // Vendor ID = MCT Elektronikladen
    (0x00, 0x00), // Product ID = Generic Demo
    (0x00, 0x01), // Device Release Number in BCD
    1, // Index of String Desc for Manufacturer
    2, // Index of String Desc for Product
    0, // Index of String Desc for SerNo
    1 // Number of possible Configurations
}; // end of DeviceDesc

const configuration_descriptor ConfigDesc =
{
    sizeof(configuration_descriptor), // Size of this Descriptor in Bytes
    DT_CONFIGURATION, // Descriptor Type (=2)
    sizeof(configuration_descriptor) +sizeof(interface_descriptor) +
    sizeof(endpoint_descriptor) + sizeof(endpoint_descriptor),
    0x00), // Total Length of Data for this Conf
    1, // No of Interfaces supported by this Conf
    1, // Designator Value for *this* Configuration
    0, // Index of String Desc for this Conf
    0xc0, // Self-powered, no Remote-Wakeup
    0 // Max. Power Consumption in this Conf (*2mA)
}; // end of ConfigDesc
const interface_descriptor InterfaceDesc =
{
    sizeof(interface_descriptor), // Size of this Descriptor in Bytes
    DT_INTERFACE, // Descriptor Type (=4)
    0, // Number of *this* Interface (0..)
    0, // Alternative for this Interface (if any)
    2, // No of EPs used by this IF (excl. EP0)
    0xff, // IF Class Code (0xFF = Vendor specific)
    0x01, // Interface Subclass Code
    0xff, // IF Protocol Code (0xFF = Vendor specific)
    0 // Index of String Desc for this Interface
}; // end of InterfaceDesc

const endpoint_descriptor Endpoint1Desc =
{
    sizeof(endpoint_descriptor), // Size of this Descriptor in Bytes
    DT_ENDPOINT, // Descriptor Type (=5)
    0x81, // Endpoint Address (EP1, IN)
    0x03, // Interrupt
    {0x08, 0x00}, // Max. Endpoint Packet Size
    10 // Polling Interval (Interrupt) in ms
}; // end of Endpoint1Desc

const endpoint_descriptor Endpoint2Desc =
{
    sizeof(endpoint_descriptor), // Size of this Descriptor in Bytes
    DT_ENDPOINT, // Descriptor Type (=5)
    0x02, // Endpoint Address (EP2, OUT)
    0x03, // Interrupt
    {0x08, 0x00}, // Max. Endpoint Packet Size
    10 // Polling Interval (Interrupt) in ms
}; // end of Endpoint2Desc

// Language IDs
//--------------
#define SD0LEN 4
//--------------

const uchar String0Desc[SD0LEN] =
{
    // Size, Type
    SD0LEN, DT_STRING,
    // LangID Codes
    0x09, 0x04
};
Source Code Files

// Manufacturer String
//--------------------------------------------------------
#define SD1LEN sizeof("MCT Elektronikladen")*2
//--------------------------------------------------------
cost uchar String1Desc[SD1LEN] = {
    // Size, Type
    SD1LEN, DT_STRING,
    // Unicode String
    'M', 0,
    'C', 0,
    'T', 0,
    ',', 0,
    'E', 0,
    'l', 0,
    'e', 0,
    'k', 0,
    't', 0,
    'r', 0,
    'o', 0,
    'n', 0,
    'i', 0,
    'k', 0,
    'l', 0,
    'a', 0,
    'd', 0,
    'e', 0,
    'n', 0
};
// Product String
#include "USB08 Evaluation Board"

const uchar String2Desc[SD2LEN] = {
    // Size, Type
    SD2LEN, DT_STRING,
    // Unicode String
    'U', 0,
    'N', 0,
    'B', 0,
    'A', 0,
    'T', 0,
    'E', 0,
    'V', 0,
    'E', 0,
    'N', 0,
    'B', 0,
    'A', 0,
    'R', 0,
    'D', 0
};

// Table of String Descriptors
uchar * const StringDescTable[] = {
    String0Desc,
    String1Desc,
    String2Desc
};
#include "u08usb.h" // Definitions for USB Data Types & Constants
#include "u08desc.c" // const USB Descriptor Data

// Source Code Option - set as required by Hardware!
#define USB_IPUE1 // Internal Pull-up Enable
  // 1=Enable(use build-in Pull-up Resistor)
  // 0=Disable (external Pull-up required)

--- Variables -----------------------------------------------

volatile uchar USB_State;
#define MAX_TXBUF_SIZE 16 // must be 2^x!
volatile uchar TxBuffer[MAX_TXBUF_SIZE];
volatile uchar TxBuf_RdIdx;
volatile uchar TxBuf_WrIdx;
#define MAX_RXBUF_SIZE 16 // must be 2^x!
volatile uchar RxBuffer[MAX_RXBUF_SIZE];
volatile uchar RxBuf_RdIdx;
volatile uchar RxBuf_WrIdx;
volatile uchar SuspendCounter;

setup_buffer SetupBuffer;
uchar SetupSize;
uchar * SetupDataPtr;

uchar R0Sequence; // DATA0/1 Flag for EP0 Rx
uchar R2Sequence; // DATA0/1 Flag for EP2 Rx

--- Functions -----------------------------------------------

void forceSTALL() {
    UCR3 |= BM_OSTALL0 + BM_ISTALL0;
}
#define ENDPOINT_HALT 0x00
#define RT_ENDPOINT 0x02

// CLEAR_FEATURE Standard Device Request Handler
// called by handleSETUP();
//
void clearFeature() {
    if (SetupBuffer.wValue.hi ||
        SetupBuffer.wIndex.hi ||
        SetupBuffer.wLength.hi ||
        SetupBuffer.wLength.lo) // check 0-fields
        forceSTALL();
    else if ((SetupBuffer.bmRequestType == RT_ENDPOINT) &&
        (SetupBuffer.wValue.lo == ENDPOINT_HALT) &&
        ((SetupBuffer.wIndex.lo == 0x81) || (SetupBuffer.wIndex.lo == 0x02))) {
        // clear EP1/2 Force STALL Bit
        if (SetupBuffer.wIndex.lo == 0x81) {// EP1
            UCR1 &= ~(BM_T1SEQ + BM_STALL1); // clear STALL, Sequence = DATA0
        }
        else { // EP2
            UCR2 &= ~BM_STALL2; // clear STALL
            R2Sequence = 0; // Sequence = DATA0
        }
        // prepare to send empty DATA1 at next IN Transaction
        UCR0 = BM_T0SEQ + BM_TX0E + 0;
    }

else forceSTALL();
}

// SET_ADDRESS Standard Device Request Handler
// called by handleSETUP();
//
void setAddress() {
    if (SetupBuffer.wIndex.hi ||
        SetupBuffer.wIndex.lo ||
        SetupBuffer.wLength.hi ||
        SetupBuffer.wLength.lo ||
        SetupBuffer.wValue.hi ||
        (SetupBuffer.wValue.lo & 0x80))
        forceSTALL();
    else {
        // prepare to send empty DATA1 at next IN Transaction
        UCR0 = BM_T0SEQ + BM_TX0E + 0;
    }
}
// SET_CONFIGURATION Standard Device Request Handler
// called by handleSETUP();
//
void setConfiguration() {

if (SetupBuffer.wIndex.hi ||
    SetupBuffer.wIndex.lo ||
    SetupBuffer.wLength.hi ||
    SetupBuffer.wLength.lo ||
    SetupBuffer.wValue.hi ||
    (SetupBuffer.wValue.lo > 1) ||
    (USB_State == US_DEFAULT)) {
    forceSTALL();
}
else {
    if (SetupBuffer.wValue.lo > 0) {
        // no need to remember the Configuration Value
        // since we support only one Configuration anyway
        USB_State = US_CONFIGURED;
        // Activate Interrupt Endpoints, reset STALL and DATA-Toggle
        UCR1 = BM_TX1E + 0; // EP1 Tx Enable, Data Size is 0
        UCR2 = BM_RX2E; // EP2 Rx Enable
    }
    else {
        // Zero means: go back to Adressed State
        USB_State = US_ADDRESSED;
        UCR1 = 0; // deactivate EP1
        UCR2 = 0; // deactivate EP2
    }
    // prepare to send empty DATA1 at next IN Transaction
    UCR0 = BM_TOSEQ + BM_TX0E + 0;
}
}

//----------------------------------------------------------------------
// GET_DESCRIPTOR Standard Device Request Handler
// called by handleSETUP();
//
void getDescriptor() {

uchar n;
uchar *dest;

switch(SetupBuffer.wValue.hi) {

  case DTDEVICE:        // Get Device Descriptor
    SetupDataPtr = (uchar *)&DeviceDesc;
    SetupSize = DeviceDesc.bLength;
    break;

  case DT_CONFIGURATION: // Get Configuration Descriptor
    SetupDataPtr = (uchar *)&ConfigDesc;
    SetupSize = ConfigDesc.wTotalLength.lo;
    break;

  case DT_STRING:       // Get String Descriptor
    // ### Table Index Boundary should be checked
    SetupDataPtr = StringDescTable[SetupBuffer.wValue.lo];
    SetupSize = *SetupDataPtr;
    break;

  default:
    forceSTALL();
    break;
}

if( SetupBuffer.wValue.hi == DTDEVICE ||
    SetupBuffer.wValue.hi == DT_CONFIGURATION ||
    SetupBuffer.wValue.hi == DT_STRING) {

// check if requested Length is less than Descriptor Length
if((SetupBuffer.wLength.lo < SetupSize) && (SetupBuffer.wLength.hi == 0))
  SetupSize = SetupBuffer.wLength.lo;
// copy (up to) 8 Bytes to EP0 Data Registers
n = 0;
dest = (uchar *)&UE0D0;
while((SetupSize!=0 && n<8) ) {
  *dest = *SetupDataPtr;
  dest++;
  SetupDataPtr++;
  SetupSize--;
  n++;
}
// prepare to send n Bytes as DATA1 at next IN Transaction
// Rem: RX0E (currently disabled) will be re-enabled at end of handleSETUP()
UCR0 = BM_T0SEQ + BM_TX0E + n;
// check if this is the last DATA packet to send
if(n < 8) SetupBuffer.bRequest = REQUEST_COMPLETE;
}

//---------------------------------------------------------------
Source Code Files

```c
void handleSETUP() {
    UCR0 &= ~BM_RX0E;       // Deactivate EP0 Receiver
    UIR2 = BM_RXDOFR;       // Reset EP0 Receive Flag

    SetupBuffer = *(setup_buffer *)(UE0D0);

    if(USR0 != 0x48) { // SETUP Transaction must be DATA0 with Size=8
        forceSTALL();   // otherwise we have an Error Condition
    } else { // now we will check the Request Type
        if((SetupBuffer.bmRequestType & 0x60) != 0) {
            forceSTALL(); // Non-Standard Requests will not be handled!
        } else { // Standard Request Decoder:
            switch(SetupBuffer.bRequest) {
                case CLEAR_FEATURE:  // 1
                    clearFeature();
                    break;
                case SET_ADDRESS:     // 5
                    setAddress();
                    break;
                case GET_DESCRIPTOR:  // 6
                    getDescriptor();
                    break;
                case SET_CONFIGURATION: // 9
                    setConfiguration();
                    break;
                default:
                    forceSTALL();
                    break;
            }
        }
    }
    UCR0 |= BM_RX0E;        // Activate EP0 Receiver
}

//-----------------------------------------------------------------------------

void handleOUT() {
    UCR0 &= ~(BM_RX0E+BM_TX0E); // Deactivate EP0 Receiver + Transmitter
    UIR2 = BM_RXDOFR;          // Reset EP0 Receive Flag

    // OUT Transactions over EP0 appear as Status Stage
    // of a Standard Device Request only

    UCR0 |= BM_RX0E;           // Activate EP0 Receiver
}

//-----------------------------------------------------------------------------
```
void handleIN() {

    uchar n;
    uchar *dest;

    UCR0 &= ~BM_TX0E; // Deactivate EP0 Transmitter
    UIR2 = BM_TXD0FR; // Reset EP0 Transmit complete Flag

    switch(SetupBuffer.bRequest) {
        case SET_ADDRESS:
            UADDR = SetupBuffer.wValue.lo | BM_USBEN;
            if(SetupBuffer.wValue.lo != 0) USB_State = US_ADDRESSED;
            else USB_State = US_DEFAULT;
            SetupBuffer.bRequest = REQUEST_COMPLETE;
            break;

        case GET_DESCRIPTOR:
            // copy (up to) 8 Bytes to EP0 Data Registers
            n = 0;
            dest = (uchar *)&UE0D0;
            while(SetupSize!=0 && n<8) {
                *dest = *SetupDataPtr;
                dest++;
                SetupDataPtr++;
                SetupSize--;
                n++;
            }
            // prepare to send n Bytes at next IN Transaction
            // toggle DATA0/1
            UCR0 = ((UCR0^BM_T0SEQ) & BM_T0SEQ) + BM_TX0E + BM_RX0E + n;
            // check if this is the last DATA packet to send
            if(n < 8) SetupBuffer.bRequest = REQUEST_COMPLETE;
            break;

        case CLEAR_FEATURE:
        case SET_CONFIGURATION:
            // nothing to do - handshake finished
            SetupBuffer.bRequest = REQUEST_COMPLETE;
            break;

        case REQUEST_COMPLETE:
            // Request is finished - just clear the TXD0F Flag (see above)
            // and do not re-enable EP0 Transmitter, since there is no more
            // data to send
            break;
        default:
            forceSTALL();
            break;
    }
}

//________________________________________________________________________
// handle IN Packet Transmit complete over EP1
//
void handleIN1() {

    uchar n;
    uchar *dest;

    UCR1 &= ~BM_TX1E; // Deactivate EP1 Transmitter
    UIR2 = BM_TXD1FR; // Reset EP1 Transmit complete Flag

    // refill EP1 Tx Data Buffer
    n = 0;
    dest = &UE1D0;
    while((TxBuf_RdIdx != TxBuf_WrIdx) && n<8) {
        *dest = TxBuffer[TxBuf_RdIdx];
        TxBuf_RdIdx = (TxBuf_RdIdx+1) & (MAX_TXBUF_SIZE-1);
        dest++;
        n++;
    }
    // Activate EP1 Transmitter to send n Bytes
    UCR1 = ((UCR1^BM_T1SEQ) & BM_T1SEQ) + BM_TX1E + n;
}

//----------------------------------------------------------------------------
// handle OUT Packet received over EP2
//
void handleOUT2() {

    uchar n;
    uchar newIdx;
    uchar *src;

    UCR2 &= ~BM_RX2E; // Deactivate EP2 Receiver
    UIR2 = BM_RXD2FR; // Reset EP2 Receive Flag

    // ### Sender’s DATA Toggle should be checked!

    // read out EP2 Rx Data Buffer
    src = &UE2D0;
    n = USR1 & BM_RP2SIZ; // Check Transfer Size
    while(n) {
        newIdx = (RxBuf_WrIdx+1) & (MAX_RXBUF_SIZE-1);
        while(newIdx == RxBuf_RdIdx)
          ; // wait if TxBuffer is full
        RxBuf[RxBuf_WrIdx] = *src;
        RxBuf_WrIdx = newIdx;
        src++;
        n--;
    }
    UCR2 = BM_RX2E; // Activate EP2 Receiver
}

//----------------------------------------------------------------------------
void initUSB() {
    UADDR = BM_USBEN + 0; // USB enable, default address
    UCR0 = 0; // reset EP0
    UCR1 = 0; // reset EP1
    UCR2 = 0; // reset EP2
    UCR3 = BM_TX1STR + // clear TX1ST Flag
        USB_IPUE*BM_PULLEN; // enable/disable internal Pull-up
    UCR4 = 0; // USB normal operation
    UIR0 = 0; // disable Interrupts
    UIR2 = 0xff; // clear all Flags in UIR1
    R0Sequence = 0; // EP0 Rx starts with DATA0
    R2Sequence = 0; // EP2 Rx starts with DATA0
    USB_State = US_POWERED; // powered, but not yet reset
    TxBuf_RdIdx = 0; // reset Buffer Indexes
    TxBuf_WrIdx = 0;
    RxBuf_RdIdx = 0;
    RxBuf_WrIdx = 0;
}

uchar getUSB() {
    uchar c;
    while(RxBuf_RdIdx == RxBuf_WrIdx) // wait if RxBuffer is empty
        c = RxBuffer[RxBuf_RdIdx];
    RxBuf_RdIdx = (RxBuf_RdIdx+1) & (MAX_RXBUF_SIZE-1);
    return c;
}

void putUSB(uchar c) {
    uchar newIdx;
    newIdx = (TxBuf_WrIdx+1) & (MAX_TXBUF_SIZE-1);
    while(newIdx == TxBuf_RdIdx) // wait if TxBuffer is full
        TxBuffer[TxBuf_WrIdx] = c;
    TxBuf_WrIdx = newIdx;
}

//------------------------------
// USB Interrupt Handler
// All Interrupt Sources of the JB8’s integrated USB peripheral
// will be treated by this ISR
@

interrupt void isrUSB() {   // End of Packet detected?
    if(UIR1 & BM_EOPF) { // End of Packet detected?
        SuspendCounter = 0; // reset 3ms-Suspend Counter
        UIR2 = BM_EOPFR;    // reset EOP Intr Flag
    } else { // has EP0 received some data?
        if(UIR1 & BM_RXD0F) { // was it a SETUP Packet?
            if(USR0 & BM_SETUP) // was it a SETUP Packet?
                handleSETUP();
            else // or a normal OUT Packet
                handleOUT();
        } else // has EP0 sent Data?
            handleIN();
        else // has EP1 sent Data?
            handleIN1();
        else // has EP2 received Data?
            handleOUT2();
    } else { // USB Reset Signal State detected?
        initUSB();        // Soft Reset of USB Systems
        UCR3 |= BM_ENABLE1 + BM_ENABLE2; // Enable EP1 and EP2
        UIRO = BM_TXD0IE + BM_RXD0IE +    // EP0 Rx/Tx Intr Enable and
                BM_TXD1IE + BM_RXD1IE +    // EP1 Tx and EP2 Rx Intr Enable
                BM_EOPIE;                  // and End-of-Packet Intr Enable
        UCR0 |= BM_RX0E;    // EP0 Receive Enable
        USB_State = US_DEFAULT; // Device is powered and reset
    }
}

//============================================================================
U08232.C

//============================================================================
// File: U08232.C
// Func: RS232 Implementation Module for USB08 Demo Application
// Ver.: 1.00
// Auth: (C)2000,2001 by Oliver Thamm, MCT Elektronikladen GbR
// http://hc08web.de/usb08
// Rem.: View/Edit this File with TAB-Size=4
//============================================================================

#include "hc08jb8.h" // HC08JB8 Register Definitions
#include "u08232.h" // Header File for RS232 Module

//----------------------------------------------------------------------------
// Hardware Dependencies - Physical Port Usage for Software SCI Tx/Rx
// Change the following Definitions to meet your Hardware Requirements:
//----------------------------------------------------------------------------

// Transmit Line is PTC[0]
#define setTxLow() (PTC &= ~0x01)
#define setTxHigh() (PTC |= 0x01)
#define enaTxOut() (DDRC |= 0x01)

// Receive Line is PTA[7]
#define tstRxLvl() (PTA & 0x80)
#define enaRxIn() (DDRA &= ~0x80)

//----------------------------------------------------------------------------
// Hardware Dependencies - SSCI Bit Timing generated by System Timer TIM
// Change the following Definitions to meet your Hardware Requirements:
//----------------------------------------------------------------------------

// clear TSTOP Bit in TSC Register to activate Counter
// PS0..PS2 Prescaler Bits in TSC Register must be 0 (default)
// so the Counter Rate is 3 MHz (0.333μs)

// 9600 Baud -> 104.1666 us per Bit -> 312.5 Clocks per Bit @ 3MHz
// 2400 Baud -> 416.6666 us per Bit -> 1250 Clocks per Bit @ 3MHz
// Adjust Value depending on Subroutine Call Overhead

//----------------------------------------------------------------------------
Source Code Files

void delayHalfBit() {
    // subtract ~20 Clocks for Overhead!
    // 120 * 5 Clocks = 600 Clocks
    _asm("\n    lda #120 \n    __dhbl:deca \n    nop \n    bne __dhbl \n    ");
}

void delayBitTime() {
    delayHalfBit();
    delayHalfBit();
}

//----------------------------------------------------------------------------
void initSSCI() {
    setTxHigh(); // set Output Data Latch H
    enaTxOut();  // enable Output Driver for Tx
    enaRxIn();   // Rx is an Input Line
}

//----------------------------------------------------------------------------
void putSSCI(char c) {
    unsigned char n;
    unsigned char ccr_save;
    // ccr_save = getCCR(); // save current Interrupt Mask
    // disableINT();      // disable Interrupts
    setTxLow();         // send Startbit
    delayBitTime();
    n=8;
    do {
        if((c&1)==0)
            setTxLow();
        else
            setTxHigh();
        delayBitTime();
        c >>= 1;
    } while(--n);
    setTxHigh(); // send Stopbit
    delayBitTime();
    delayBitTime();
    // setCCR(ccr_save); // restore previous Interrupt Mask
}
char getSSCI() {
    char c;
    unsigned char n;
    unsigned char ccr_save;

    // ccr_save = getCCR(); // save current Interrupt Mask
    // disableINTR(); // disable Interrupts

    while(tstRxLvl()!=0) ; // wait for H-L transition
    delayHalfBit();
    n=8;
    do {
        delayBitTime();
        c >>= 1;
        if(tstRxLvl()!=0)
            c |= 0x80;
    } while(--n);
    delayBitTime();
    if(tstRxLvl()==0) { // check Rx Line during Stopbit
        // add framing error
        // handling if desired
    }
    delayHalfBit();

    // setCCR(ccr_save); // restore previous Interrupt Mask
    return c;
}

//-----------------------------------------------------------------------------------
Source Code Files

U08KEY.C

//============================================================================
// File: U08KEY.C
// Func: Key Input Functions for USB08
// Ver.: 1.00
// Auth: (C)2000,2001 by Oliver Thamm, MCT Elektronikladen GbR
// http://hc08web.de/usb08
// Rem.: View/Edit this File with TAB-Size=4
//============================================================================

#include "hc08jb8.h"
#include "u08key.h"

//-- Definitions -------------------------------------------------------------
// Specification of *active* Key Inputs:
// PTA[4,5,6] = %01110000 = 0x70
// First Key connected to Port Bit 4
#define KEY_MASK 0x70
#define KEY_FIRST 4

//-- Variables ---------------------------------------------------------------
// Var used to track the Key Status
unsigned char KeyState;

void initKey() {
  POCR |= 0x01; // enable PTA Pullups
  PTA  |= KEY_MASK; // write 1 to Output Latches
  DDRA |= KEY_MASK; // output H-Level Pulse
  DDRA &= ~KEY_MASK; // back to Input
  KBIER = KEY_MASK; // enable Interrupts
  KBSCR = 0x04; // reset ACKK (just in case)
  KeyState = 0; // reset internal Status Var
}

char getKey(unsigned char x) {
  x += KEY_FIRST-1; // calculate Bit Position
  x = 1 << x; // create Bit Mask
  if(KeyState & x) // test the relevant Status Bit
    return 1;
  return 0;
}

@interrupt void isrKey() {
  KeyState ^= ~(PTA | ~KEY_MASK);
  KBSCR = 0x04; // reset ACKK (for noise safety only)
}

//============================================================================
U08ADC.C

//----------------------------------------------------------------------------
// File: U08ADC.C
// Func: Software ADC for USB08
// Ver.: 1.00
// Auth: (C)2000,2001 by Oliver Thamm, MCT Elektronikladen GbR
// http://hc08web.de/usb08
// Rem.: View/Edit this File with TAB-Size=4
//----------------------------------------------------------------------------

#include "hc08jb8.h"
#include "u08adc.h"

//----------------------------------------------------------------------------

void initSADC() {
    // disable internal Pull-Ups on PTE
    POCR &= ~0x80; // disable PTE20P
}

//----------------------------------------------------------------------------

unsigned scaleSADC(unsigned t1, unsigned t2) {
    t1 >>= 4;
    t2 <<= 4;
    _asm("lda 5,sp");
    _asm("psha");
    _asm("pulh");
    _asm("ldx 2,sp");
    _asm("lda 6,sp");
    _asm("div"); // A = H:A/X
    _asm("clrx"); // 0:A = t2/t1
}

//----------------------------------------------------------------------------

int getSADC(char channel) {
    unsigned t0, t1, t2;
    unsigned char p;
    unsigned volatile zz;

    // convert channel # 1/2/3 to 0x01/0x02/0x04
    if(channel == 3) channel++;

// *** calibration cycle ***
PTD &= ~0x78;    // PTD[3..6] = L
DDRD |= 0x78;     // Output
PTE |= 0x07;      // PTE[0..2] = H;
DDRE |= 0x07;     // Output
for(zz=0;zz<1000;zz--);

DDRE &= ~0x07;    // PTE HiZ (Input)
t0 = TCNT;
while((PTE & channel) != 0);
t1 = TCNT;
t1 -= t0;

// *** acquisition cycle ***
DDRD &= ~0x38;    // PTD[3..5] = HiZ
DDRE |= 0x07;     // Output
for(zz=0;zz<1000;zz--);

DDRE &= ~0x07;    // PTE HiZ (Input)
t0 = TCNT;
while((PTE & channel) != 0);
t2 = TCNT;
t2 = t2 - t0 - t1 - 100;
if(t2 > 50000u) t2=0;    // underflow
if(t2 >= t1) t2 = t1-1;   // overflow

// *** calculate scaled result ***
t2 = scaleSADC(t1,t2);
return t2;
}

主办：Freescale Semiconductor, Inc. 
为Motorola的用户打造：www.freescale.com
VECJB8.C

// INTERRUPT VECTORS TABLE FOR HC908JB8
// Cosmic HC08 C Compiler

extern void _stext();  /* startup routine */
extern void isrDummy();
extern void isrUSB();
//extern void timer0ISR();
extern void isrKey();

void (* const _vectab[])() = {
    isrKey,  /* Keypad     */
    isrDummy,  /* TIMER overflow */
    isrDummy,  /* TIMER channel 1 */
    isrDummy,  /* TIMER channel 0 */
    isrDummy,  /* IRQ1        */
    isrUSB,  /* USB         */
    isrDummy,  /* SWI         */
    _stext,  /* RESET      */
};
CRTSJBJ8.S

; C STARTUP FOR HC08JB8
; Copyright (c) 1995 by COSMIC Software
;
xref    _main, __sbss, __memory, __stack, __HC08Setup
xdef    _exit, __stext
;
__stext:
  ldhx #__stack               ; initialize stack pointer
  txs
  jsr __HC08Setup

  ldhx #__sbss                ; start of bss
  bra    loop                 ; start loop
zbcl:
  clr  0,x                    ; clear byte
  aix  #1                     ; next byte
loop:
  cphx #__memory              ; up to the end
  bne  zbcl                   ; and loop
prog:
  ldhx #__stack               ; initialize stack pointer
  txs
  jsr    _main                 ; execute main
_exit:
  bra    _exit                  ; and stay here
;
end
USB08.LKF

# USB08 LINK COMMAND FILE
# COSMIC HC08 C COMPILER
#
+seg .text -b 0xdc00 -n .text           # program start address
+seg .const -a .text                    # constants follow code
+seg .bsct -b 0x0040 -n .bsct           # zero page start address
+seg .ubsct -a .bsct -n .ubsct          # data start address
+seg .data -a .ubsct                    # start address of bss
+def __sbss=@.bss

# Put your startup file here
CRTSJ8.O                               # startup routine

# Put your files here
u08main.o
u08key.o
u08adc.o
"C:\PROGRAMS\COSMIC\CX08\LIB\LIBI.H08"
"C:\PROGRAMS\COSMIC\CX08\LIB\LM.M08"

+seg .const -b 0xfff0                   # vectors start address
# Put your interrupt vectors file here if needed
VECJB8.O

+def __memory=@.bss                     # symbol used by library
+def __stack=0x013f                     # stack pointer initial value

BUILD.BAT

cx6808 -v -l u08main.c u08adc.c u08key.c vecjb8.c
clnk -m usb08.map -o usb08.h08 usb08.lkf
chex -fm -h -o usb08.s19 usb08.h08
### USB08.MAP

Map of usb08.h08 from link file usb08.lkf - Sun Jan 07 19:29:30 2001

**Segments:**

<table>
<thead>
<tr>
<th>Start Address</th>
<th>End Address</th>
<th>Length</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000dc00</td>
<td>0000e263</td>
<td>1635</td>
<td>.text</td>
</tr>
<tr>
<td>0000e263</td>
<td>0000e2f5</td>
<td>146</td>
<td>.const</td>
</tr>
<tr>
<td>00000040</td>
<td>00000040</td>
<td>0</td>
<td>.bsct</td>
</tr>
<tr>
<td>00000040</td>
<td>00000041</td>
<td>1</td>
<td>.ubsct</td>
</tr>
<tr>
<td>00000041</td>
<td>00000041</td>
<td>0</td>
<td>.data</td>
</tr>
<tr>
<td>00000041</td>
<td>00000075</td>
<td>52</td>
<td>.bss</td>
</tr>
<tr>
<td>0000fff0</td>
<td>00010000</td>
<td>16</td>
<td>.const</td>
</tr>
</tbody>
</table>

**Modules:**

- **crtsjb8.o:**
  - start 0000dc00 end 0000dc1d length 29 section .text

- **u08main.o:**
  - start 0000dc1d end 0000e0e0 length 1219 section .text
  - start 00000041 end 00000074 length 51 section .bss
  - start 0000e263 end 0000e2f5 length 146 section .const

- **u08key.o:**
  - start 0000e0e0 end 0000e133 length 83 section .text
  - start 00000074 end 00000075 length 1 section .bss

- **u08adc.o:**
  - start 0000e133 end 0000e263 length 304 section .text

- **(c:\programme\cosmic\cx08\Lib\libm.h08)ireg.o:**
  - start 00000040 end 00000041 length 1 section .ubsct

- **vecjb8.o:**
  - start 0000fff0 end 00010000 length 16 section .const

**Stack usage:**

- **u08adc.o:**
  - _getSADC : 18 (12)
  - _initSADC : 2 (2)
  - _scaleSADC : 6 (6)

- **u08key.o:**
  - _getKey : 4 (4)
  - _initKey : 2 (2)
  - _isrKey : > 7 (7)

- **u08main.o:**
  - __HC08Setup : > 2 (2)
  - _clearFeature : 4 (2)
  - _forceSTALL : 2 (2)
<table>
<thead>
<tr>
<th>Symbol Name</th>
<th>Address</th>
<th>Definition Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>_ConfigDesc</td>
<td>0000e275</td>
<td>defined in u08main.o section .const</td>
</tr>
<tr>
<td>_DeviceDesc</td>
<td>0000e263</td>
<td>defined in u08main.o section .const</td>
</tr>
<tr>
<td>_InterfaceDesc</td>
<td>0000e27e</td>
<td>defined in u08main.o section .const</td>
</tr>
<tr>
<td>_KeyState</td>
<td>00000074</td>
<td>defined in u08key.o section .bss</td>
</tr>
<tr>
<td>_R0Sequence</td>
<td>00000042</td>
<td>defined in u08main.o section .bss</td>
</tr>
<tr>
<td>_R2Sequence</td>
<td>00000041</td>
<td>defined in u08main.o section .bss</td>
</tr>
<tr>
<td>_RxBuffer_RdIdx</td>
<td>0000004f</td>
<td>defined in u08main.o section .bss</td>
</tr>
<tr>
<td>_RxBuffer_WrIdx</td>
<td>00000050</td>
<td>defined in u08main.o section .bss</td>
</tr>
<tr>
<td>_RxBuffer</td>
<td>00000051</td>
<td>defined in u08main.o section .bss</td>
</tr>
<tr>
<td>_SetupBuffer</td>
<td>00000046</td>
<td>defined in u08main.o section .bss</td>
</tr>
<tr>
<td>_SetupDataPtr</td>
<td>00000043</td>
<td>defined in u08main.o section .bss</td>
</tr>
<tr>
<td>_SetupSize</td>
<td>00000045</td>
<td>defined in u08main.o section .bss</td>
</tr>
<tr>
<td>_String0Desc</td>
<td>0000e295</td>
<td>defined in u08main.o section .const</td>
</tr>
<tr>
<td>_String1Desc</td>
<td>0000e299</td>
<td>defined in u08main.o section .const</td>
</tr>
<tr>
<td>_String2Desc</td>
<td>0000e2c1</td>
<td>defined in u08main.o section .const</td>
</tr>
<tr>
<td>_StringDescTable</td>
<td>0000e2ef</td>
<td>defined in u08main.o section .const</td>
</tr>
<tr>
<td>_SuspendCounter</td>
<td>0000004e</td>
<td>defined in u08main.o section .bss</td>
</tr>
<tr>
<td>_TxBuffer_RdIdx</td>
<td>00000062</td>
<td>defined in u08main.o section .bss</td>
</tr>
<tr>
<td>_TxBuffer_WrIdx</td>
<td>00000061</td>
<td>defined in u08main.o section .bss</td>
</tr>
<tr>
<td>_TxBuffer</td>
<td>00000063</td>
<td>defined in u08main.o section .bss</td>
</tr>
<tr>
<td>_USB_State</td>
<td>00000073</td>
<td>defined in u08main.o section .bss</td>
</tr>
<tr>
<td>__HC08Setup</td>
<td>0000dff8</td>
<td>defined in u08main.o section .text</td>
</tr>
<tr>
<td>__memory</td>
<td>00000075</td>
<td>defined in command file section .bss</td>
</tr>
<tr>
<td>__sbss</td>
<td>00000041</td>
<td>defined in command file section .bss</td>
</tr>
<tr>
<td>__stack</td>
<td>00000013f</td>
<td>defined in command file</td>
</tr>
<tr>
<td>__stext</td>
<td>0000dc00</td>
<td>defined in u08main.o section .text</td>
</tr>
<tr>
<td>__vectab</td>
<td>0000fff0</td>
<td>defined in u08main.o section .const</td>
</tr>
</tbody>
</table>

**Symbols:**

- _ConfigDesc: 0000e275 defined in u08main.o section .const
- _DeviceDesc: 0000e263 defined in u08main.o section .const
- _InterfaceDesc: 0000e27e defined in u08main.o section .const
- _KeyState: 00000074 defined in u08key.o section .bss
- _R0Sequence: 00000042 defined in u08main.o section .bss
- _R2Sequence: 00000041 defined in u08main.o section .bss
- _RxBuffer_RdIdx: 0000004f defined in u08main.o section .bss
- _RxBuffer_WrIdx: 00000050 defined in u08main.o section .bss
- _RxBuffer: 00000051 defined in u08main.o section .bss
- _SetupBuffer: 00000046 defined in u08main.o section .bss
- _SetupDataPtr: 00000043 defined in u08main.o section .bss
- _SetupSize: 00000045 defined in u08main.o section .bss
- _String0Desc: 0000e295 defined in u08main.o section .const
- _String1Desc: 0000e299 defined in u08main.o section .const
- _String2Desc: 0000e2c1 defined in u08main.o section .const
- _StringDescTable: 0000e2ef defined in u08main.o section .const
- _SuspendCounter: 0000004e defined in u08main.o section .bss
- _TxBuffer_RdIdx: 00000062 defined in u08main.o section .bss
- _TxBuffer_WrIdx: 00000061 defined in u08main.o section .bss
- _TxBuffer: 00000063 defined in u08main.o section .bss
- _USB_State: 00000073 defined in u08main.o section .bss
- __HC08Setup: 0000dff8 defined in u08main.o section .text
- __memory: 00000075 defined in command file section .bss
- __sbss: 00000041 defined in command file section .bss
- __stack: 00000013f defined in command file
- __stext: 0000dc00 defined in u08main.o section .text
- __vectab: 0000fff0 defined in vecjb8.o section .const
Source Code Files

```c
_clearFeature          0000dc25  defined in u08main.o section .text
_exit                 0000dc1b  defined in crtsjb8.o section .text
_forceSTALL          0000dc1d  defined in u08main.o section .text
_getDescriptor       0000dce1  defined in u08main.o section .text
_getKey              0000e106  defined in u08key.o section .text
used in u08main.o
_getSADC             0000e15d  defined in u08adc.o section .text
_used in u08main.o
_getUSB              0000df5e  defined in u08main.o section .text
_handleIN            0000ddfd  defined in u08main.o section .text
_handleIN1           0000de8d  defined in u08main.o section .text
_handleOUT           0000ddd0  defined in u08main.o section .text
_handleOUT2          0000dee0  defined in u08main.o section .text
_handleSETUP         0000dd94  defined in u08main.o section .text
_initKey             0000e0e0  defined in u08key.o section .text
used in u08main.o
_initSADC            0000e133  defined in u08adc.o section .text
_used in u08main.o
_initUSB             0000df2f  defined in u08main.o section .text
_isrDummy            0000dfff  defined in u08main.o section .text
_used in vecjb8.o
_isrKey              0000e122  defined in u08key.o section .text
_used in vecjb8.o
_isrUSB              0000df9d  defined in u08main.o section .text
_used in vecjb8.o
_main                0000e001  defined in u08main.o section .text
_used in crtsjb8.o
_putUSB              0000df7c  defined in u08main.o section .text
_scaleSADC           0000e13b  defined in u08adc.o section .text
_setAddress         0000dc6e  defined in u08main.o section .text
_setConfiguration   0000dc95  defined in u08main.o section .text
_c_reg               00000040  defined in (c:\programme\cosmic\cx08\Lib\libm.h08)
 Ire.o section .ubsct
_used in u08main.o
```
Source Code Files

S123E200B608E40926FAE0C9EEF03B60D9EE7049EE008879F95E20797869EE006879F9503
S123E220E2059786A0649EE7049FA20095E702E603A051E602A2C325046F026F03E603E08E
S123E24005E602E204250CE605A001E703E604A200E702E60387E60287E605EE04CDE13B26
S106E260A70C8183
S123E263120110010000008700C00000001010200010902200010100C0000904000002EE
S123E283FF01FF00070581030800A070502030800A0403090428034D0043005400200070
S123E2A345006C0065006B00740072006F006E00690067006F006E00730069006E00AD
S115E2E3200042006F00610072006400E295E299E2C188
S113FFF0E122DFFFDFFFDFFFDFFDF9DDFFDC004C
S903FFFFFE
Appendix D. Bill of Materials and Schematic

This appendix includes:

- USB08 V 1.01 bill of materials — Table D-1
- USB08 evaluation board schematic — Figure D-1
### Table D-1. Bill of Materials for USB08 V 1.01

<table>
<thead>
<tr>
<th>Part</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1, C2</td>
<td>22 pF</td>
</tr>
<tr>
<td>C3, C4, C7, C8, C9, C10, C11, C12, C13, C14, C15, C17, C18, C19</td>
<td>100 nF</td>
</tr>
<tr>
<td>C5, C6</td>
<td>10 µF</td>
</tr>
<tr>
<td>C16</td>
<td>100 µF/25 V</td>
</tr>
<tr>
<td>C20, C21, C22</td>
<td>10 nF</td>
</tr>
<tr>
<td>D1, D2, D3, D4</td>
<td>LED</td>
</tr>
<tr>
<td>D5, D6</td>
<td>BAT42</td>
</tr>
<tr>
<td>D7</td>
<td>ZD8.2V</td>
</tr>
<tr>
<td>D8</td>
<td>1N4001</td>
</tr>
<tr>
<td>IC1</td>
<td>MC68HC908JB8ADW</td>
</tr>
<tr>
<td>IC2</td>
<td>MAX232A</td>
</tr>
<tr>
<td>IC3</td>
<td>7805</td>
</tr>
<tr>
<td>JP1</td>
<td>Header 2x8</td>
</tr>
<tr>
<td>JP2</td>
<td>Header 1x3</td>
</tr>
<tr>
<td>L1, L2</td>
<td>Ferrite</td>
</tr>
<tr>
<td>Q1</td>
<td>XTAL 6 MHz</td>
</tr>
<tr>
<td>R1</td>
<td>MPY7P (photoresistor)</td>
</tr>
<tr>
<td>R2</td>
<td>K164-4.7k (NTC)</td>
</tr>
<tr>
<td>R3</td>
<td>10 k (potentiometer)</td>
</tr>
<tr>
<td>R4, R5, R6</td>
<td>10 k</td>
</tr>
<tr>
<td>R7</td>
<td>1.5 k/5 % (optional)</td>
</tr>
<tr>
<td>R8, R13, R14, R15</td>
<td>10 k</td>
</tr>
<tr>
<td>R9</td>
<td>2.2 k</td>
</tr>
<tr>
<td>R10, R11, R12, R19</td>
<td>330 R</td>
</tr>
<tr>
<td>R16, R17</td>
<td>27 R</td>
</tr>
<tr>
<td>R18</td>
<td>10 M</td>
</tr>
<tr>
<td>S1, S2, S3, S4, S5</td>
<td>Push Button</td>
</tr>
<tr>
<td>X1</td>
<td>Header 2x13</td>
</tr>
<tr>
<td>X2, X3</td>
<td>Header 2x5</td>
</tr>
<tr>
<td>X4</td>
<td>Power Jack</td>
</tr>
</tbody>
</table>
Figure D-1: USB08 Evaluation Board Schematic

Design: Freescale Semiconductor, Inc.

For More Information On This Product, Go to: www.freescale.com
Appendix E. Universal USB Device Driver (USBIO)

USBIO

Reference Manual
Version 1.41
2000, December 20

By: Thesycon® Systemsoftware & Consulting GmbH
Wetzlarer Platz 1
D-98693 Ilmenau
Germany

Telephone: +49 3677 / 8462-0
Fax: +49 3677 / 8462-18
Email: USBIO@thesycon.de
Web: http://www.thesycon.de

Copyright © 1998-2000 Thesycon Systemsoftware and Consulting GmbH
All Rights Reserved

Reprinted with permission from Thesycon Systemsoftware & Consulting GmbH. For inclusion in this document, the manual has been reformatted only.

The following trademarks are referenced throughout this manual:
Microsoft, Windows, Win32, Windows NT, and Visual C++ are either trademarks or registered trademarks of Microsoft Corporation.
Other brand and product names are trademarks or registered trademarks of their respective holders.
E.1 Contents

E.2 Introduction ................................................................. 135
E.3 Overview ........................................................................ 135
E.3.1 Platforms .................................................................... 136
E.3.2 Features ....................................................................... 136
E.4 Architecture ................................................................. 138
E.4.1 USBIO Object Model .................................................. 140
E.4.1.1 USBIO Device Objects ............................................ 140
E.4.1.2 USBIO Pipe Objects .............................................. 142
E.4.2 Establishing a Connection to the Device ....................... 144
E.4.3 Power Management .................................................... 146
E.4.4 Device State Change Notifications ............................... 148
E.5 Programming Interface .................................................. 149
E.5.1 Programming Interface Overview ................................. 149
E.5.2 Control Requests ....................................................... 150
  IOCTL_USBIO_GET_DESCRIPTOR ..................................... 151
  IOCTL_USBIO_SET_DESCRIPTOR ..................................... 152
  IOCTL_USBIO_SET_FEATURE ........................................ 153
  IOCTL_USBIO_CLEAR_FEATURE ..................................... 154
  IOCTL_USBIO_GET_STATUS .......................................... 155
  IOCTL_USBIO_GET_CONFIGURATION ................................ 156
  IOCTL_USBIO_GET_INTERFACE ....................................... 157
  IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR ....................... 158
  IOCTL_USBIO_SET_CONFIGURATION ................................ 159
  IOCTL_USBIO_UNCONFIGUREDEVICE ................................ 160
  IOCTL_USBIO_SET_INTERFACE ....................................... 161
  IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST .................. 162
  IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST ................ 163
  IOCTL_USBIO_GET_DEVICE_PARAMETERS ......................... 164
  IOCTL_USBIO_SET_DEVICE_PARAMETERS ......................... 165
  IOCTL_USBIO_GET_CONFIGURATION_INFO ......................... 166
  IOCTL_USBIO_RESET_DEVICE ........................................ 167
  IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER ..................... 168
  IOCTL_USBIO_SET_DEVICE_POWER_STATE .......................... 169
  IOCTL_USBIO_GET_DEVICE_POWER_STATE .......................... 170
  IOCTL_USBIO_GET_DRIVER_INFO ..................................... 171
  IOCTL_USBIO_CYCLE_PORT ........................................... 172
  IOCTL_USBIO_BIND_PIPE ............................................. 174
Universal USB Device Driver (USBIO)

Contents

E.5.3 Data Transfer Requests .......................... 182
E.5.3.1 Bulk and Interrupt Transfers ................. 182
E.5.3.2 Isochronous Transfers ........................ 184
E.5.4 Input and Output Structures ..................... 185
USBIO_DRIVER_INFO .................................... 186
USBIO_DESCRIPTOR_REQUEST ......................... 187
USBIO_FEATURE_REQUEST ............................. 188
USBIO_STATUS_REQUEST ............................... 189
USBIO_STATUS_REQUEST_DATA ......................... 190
USBIO_GET_CONFIGURATION_DATA ................. 191
USBIO_GET_INTERFACE ................................. 192
USBIO_GET_INTERFACE_DATA ......................... 193
USBIO_INTERFACE_SETTING ......................... 194
USBIO_SET_CONFIGURATION ......................... 195
USBIO_CLASS_OR_VENDOR_REQUEST ................. 196
USBIO_DEVICE_PARAMETERS ....................... 198
USBIO_INTERFACE_CONFIGURATION_INFO ............ 200
USBIO_PIPE_CONFIGURATION_INFO .................. 202
USBIO_CONFIGURATION_INFO ....................... 204
USBIO_FRAME_NUMBER ................................. 205
USBIO_DEVICE_POWER .................................. 206
USBIO_BIND_PIPE ...................................... 207
USBIO_PIPE_PARAMETERS ......................... 208
USBIO_PIPE_CONTROL_TRANSFER .................... 209
USBIO_ISO_TRANSFER .................................. 210
USBIO_ISO_PACKET ..................................... 212
USBIO_ISO_TRANSFER_HEADER ....................... 213
E.5.5 Enumeration Types ............................... 214
USBIO_PIPE_TYPE ...................................... 214
USBIO_REQUEST_RECIPIENT ......................... 215
USBIO_REQUEST_TYPE ................................. 216
USBIO_DEVICE_POWER_STATE ....................... 217
E.5.6 Error Codes ...................................... 218
E.6 USBIO Class Library .................................................. 220
  E.6.1 CUsblo Class ..................................................... 220
  E.6.2 CUsbloPipe Class .............................................. 221
  E.6.3 CUsbloThread Class .......................................... 222
  E.6.4 CUsbloReaderClass .......................................... 222
  E.6.5 CUsbloWriter Class .......................................... 222
  E.6.6 CUsbloBufClass .............................................. 223
  E.6.7 CUsbloBufPool Class ......................................... 223

E.7 USBIO Demo Application ........................................... 223
  E.7.1 Dialog Pages for Device Operations ....................... 224
    E.7.1.1 Device ..................................................... 224
    E.7.1.2 Descriptors .............................................. 224
    E.7.1.3 Configuration .......................................... 225
    E.7.1.4 Interface ................................................. 225
    E.7.1.5 Pipes ...................................................... 225
    E.7.1.6 Class or Vendor Request ................................. 226
    E.7.1.7 Feature .................................................... 226
    E.7.1.8 Other ....................................................... 226
    E.7.1.9 Dialog Pages for Pipe Operations .................... 227
    E.7.1.10 Pipe ....................................................... 227
    E.7.1.11 Buffers ................................................... 227
    E.7.1.12 Control ................................................... 228
    E.7.1.13 Read from Pipe to Output Window .................... 228
    E.7.1.14 Read from Pipe to File .................................. 228
    E.7.1.15 Write from File to Pipe ................................. 229

E.8 Installation Issues ................................................. 229
  E.8.1 Automated Installation: The USBIO Installation Wizard ... 229
  E.8.2 Manual Installation: The USBIO Setup Information File .... 232
  E.8.3 Uninstalling USBIO ........................................... 236
  E.8.4 Building a Customized Driver Setup ....................... 237

E.9 Registry Entries .................................................. 239

E.10 Related Documents ............................................... 241

E.11 Light Version Limitations ....................................... 241
E.2 Introduction

USBIO is a generic Universal Serial Bus (USB) device driver for Windows 98, Windows Millennium (ME), and Windows 2000. It is able to control any type of USB device and provides a convenient programming interface that can be used by Win32 applications.

This document describes the architecture, the features, and the programming interface of the USBIO device driver. Furthermore it includes instructions for installing and using the device driver.

The reader of this document is assumed to be familiar with the specification of the Universal Serial Bus and with common aspects of Win32-based application programming.

E.3 Overview

Support for the Universal Serial Bus (USB) is built into the Windows 98, Windows Millennium, and Windows 2000 operating systems. These systems include device drivers for the USB Host Controller hardware, for USB Hubs, and for some classes of USB devices. The USB device drivers provided by Microsoft support devices that conform with the appropriate USB device class definitions made by the USB Implementers Forum. USB devices that do not conform to one of the USB device class specifications, e.g. in the case of a new device class or a device under development, are not supported by device drivers included with the operating system.

In order to use devices that are not supported by the operating system itself the vendor of such a device is required to develop an USB device driver. This driver has to conform to the Win32 Driver Model (WDM) that defines a common driver architecture for Windows 98, Windows Millennium, and Windows 2000. Writing, debugging, and testing of such a driver means considerable effort and requires a lot of knowledge about development of kernel mode drivers.

By using the generic USB device driver USBIO it is possible to get any USB device up and running without spending the time and the effort of developing a device driver. Especially, this might be useful during
Universal USB Device Driver (USBIO)

development or test of a new device. But in many cases it is also suitable to include the USBIO device driver in the final product. So there is no need to develop and test a custom device driver for the USB-based product at all.

E.3.1 Platforms

The USBIO driver supports the following operating system platforms:

- Windows 98 (Gold), the first release of Windows 98
- Windows 98 Second Edition (SE), the second release of Windows 98
- Windows Millennium, the successor to Windows 98
- Windows 2000, the successor to Windows NT
- Windows 2000 Service Pack 1

**NOTE:** Windows NT 4.0 and Windows 95 are not supported by USBIO.

E.3.2 Features

The USBIO driver provides the following features:

- Compiles with the Win32 Driver Model (WDM)
- Supports Plug&Play
- Supports Power Management
- Provides an interface to USB devices that can be used by any Win32 application
- Provides an interface to USB endpoints (pipes) that is similar to files
- Fully supports asynchronous (overlapped) data transfer operations
- Supports the USB transfer types Control, Interrupt, Bulk, and Isochronous
Multiple USB devices can be controlled by USBIO at the same time.

Multiple applications can use USBIO at the same time.

The USBIO device driver can be used to control any USB device from a Win32 application running in user mode. Examples of such devices are:

- telephone and fax devices
- telephone network switches
- audio and video devices (e.g. cameras)
- measuring devices (e.g. oscilloscopes, logic analyzers)
- sensors (e.g. temperature, pressure)
- data converters (e.g. A/D converters, D/A converters)
- bus converters or adapters (e.g. RS 232, IEEE 488)
- chip card devices

If a particular kernel mode interface (e.g. WDM Kernel Mode Streaming, NDIS) has to be supported in order to integrate the device into the operating system, it is not possible to use the generic USBIO driver. However, in such a case it is possible to develop a custom device driver based on the source code of the USBIO though. Please contact Thesycon if you need support on such kind of project.

Although the USBIO device driver fully supports isochronous data pipes, there are some limitations with respect to isochronous data transfers. They result from the fact that the processing of the isochronous data streams has to be performed by the application which runs in user mode. There is no guaranteed response time for threads running in user mode. This may be critical for the implementation of some synchronization methods, for example when the data rate is controlled by loop-back packets (see the USB Specification, Chapter 5 for synchronization issues of isochronous data streams).

However, it is possible to support all kinds of isochronous data streams using the USBIO driver. But the delays that might be caused by the thread scheduler of the operating system should be taken into consideration.
E.4 Architecture

Figure E-1 shows the USB driver stack that is part of the Windows 98, Windows Millennium, and Windows 2000 operating systems. All drivers are embedded within the WDM layered architecture.

![Diagram of USB Driver Stack]

The following modules are shown in Figure E-1:

- USB Host Controller is the hardware component that controls the Universal Serial Bus. It also contains the USB Root Hub.

- OpenHCI.SYS is the host controller driver for controllers that conform with the Open Host Controller Interface specification. Optionally, it can be replaced by UHCD.SYS that is the Universal Host Controller Driver. Which driver is used depends on the main-board chip set for the PC. For instance, Intel chipsets contain an Universal Host Controller.
• USBD.SYS is the USB Bus Driver that controls and manages all devices connected to the USB. It is provided by Microsoft as part of the operating system.

• USBHUB.SYS is the USB Hub Driver. It is responsible for managing and controlling USB Hubs.

• USBIO.SYS is the generic USB device driver USBIO.

The software interface that is provided by the operating system for use by USB device drivers is called USB Driver Interface (USBDI). It is exported by the USBD at the top of the driver stack. USBDI is an IRP-based interface. This means that each individual request is packaged into an I/O request packet (IRP), a data structure that is defined by WDM. The I/O request packets are passed to the next driver in the stack for processing and returned to the caller after completion.

The USB Driver Interface is accessible for kernel mode drivers only. Normally, there is no way to use this interface directly from applications that run in user mode. The USBIO device driver was designed to overcome this limitation. It connects to the USBDI at its lower edge and provides a private interface at its upper edge that can be used by Win32 applications. Thus, the USB driver stack becomes accessible to applications. A Win32 application is able to communicate with one or more USB devices by using the programming interface exported by the USBIO device driver. Furthermore, the USBIO programming interface may be used by more than one application or by multiple instances of one application at the same time.

The main design goal for the USBIO device driver was to make available to applications all the features that the USB driver stack provides at the USBDI level. For that reason the programming interface of the USBIO device driver (USBIOI) is closely related to the USBDI. But many of the functions cannot be translated in an one-to-one relationship.
E.4.1 USBIO Object Model

The USBIO device driver provides a communication model that consists of device objects and pipe objects. The objects are created, destroyed, and managed by the USBIO driver. An application can open handles to device objects and bind these handles to pipe objects.

E.4.1.1 USBIO Device Objects

Each USBIO device object is associated with a physical USB device that is connected to the USB. A device object is created by the USBIO driver in response to an Add Device request from the Plug&Play Manager of the operating system. The USBIO driver is able to handle multiple device objects at the same time.

Each device object created by USBIO is registered with the operating system by using a unique identifier (GUID, Globally Unique Identifier). This identifier is called “Device Interface ID”. All device objects managed by USBIO are identified by the same GUID. The GUID is defined in the USBIO Setup Information (INF) file. Based on the GUID and an instance number, the operating system generates a unique name for each device object. This name should be considered as opaque by applications. It should never be used directly or stored permanently.

It is possible to enumerate all the device objects associated with a particular GUID by using functions provided by the Windows Setup API. The Functions used for this purpose are:

- SetupDiGetClassDevs()
- SetupDiEnumDeviceInterfaces()
- SetupDiGetDeviceInterfaceDetail()

The result of the enumeration process is a list of device objects currently created by USBIO. Each of the USBIO device objects corresponds to a device currently connected to the USB. For each device object an opaque device name string is returned. This string can be passed to CreateFile() to open the device object.

A default Device Interface ID (GUID) is built into the USBIO driver. This default ID is defined in USBIO_I.H. Each device object created by USBIO is registered by using this default ID. The default Device
Interface ID is used by the USBIO demo application for device enumeration. This way, it is always possible to access devices connected to the USBIO from the demo application.

In addition, an user-defined Device Interface ID is supported by USBIO. This user-defined GUID is specified in the USBIO INF file by the USBIO_UserInterfaceGuid variable. If the user-defined interface ID is present at device initialization time USBIO registers the device with this ID. Thus, two interfaces — default and user-defined — are registered for each device. The default Device Interface ID should only be used by the USBIO demo application. Custom applications should always use a private user-defined Device Interface ID. This way, device naming conflicts are avoided.

**IMPORTANT:** Every USBIO customer should generate its own private device interface GUID. This is done by using the tool GUIDGEN.EXE from the Microsoft Platform SDK or the VC++ package. This private GUID is specified as user-defined interface in USBIO_UserInterfaceGuid in the USBIO INF file. The private GUID is also used by the customer’s application for device enumeration. For that reason the generated GUID must also be included in the application. The macro DEFINE_GUID() can be used for that purpose. See the Microsoft Platform SDK documentation for further information.

As stated above, all devices connected to USBIO will be associated with the same device interface ID that is also used for device object enumeration. Because of that, the enumeration process will return a list of all USBIO device objects. In order to differentiate the devices an application should query the device descriptor or string descriptors. This way, each device instance can be identified unambiguously.

After the application has received one or more handles for the device, operations can be performed on the device by using a handle. If there is more than one handle to the same device, it makes no difference which handle is used to perform a certain operation. All handles that are associated with the same device behave the same way.

**NOTE:** Former versions of USBIO (up to V1.16) used a different device naming scheme. The device name was generated by appending an instance number to a common prefix. So the device names were static. In order
to ensure compatibility USBIO still supports the old naming scheme. This feature can be enabled by defining a device name prefix in the variable `USBIO_DeviceBaseName` in the USBIO INF file. However, it is strongly recommended to use the new naming scheme based on Device Interface IDs (GUIDs), because it conforms with current Windows 2000 guidelines. The old-style static names should only be used if backward-compatibility with former versions of USBIO is required.

E.4.1.2 USBIO Pipe Objects

The USBIO driver uses pipe objects to represent an active endpoint of the device. The pipe objects are created when the device configuration is set. The number and type of created pipe objects depend on the selected configuration. The USBIO driver does not control the default endpoint (endpoint zero) of a device. This endpoint is owned by the USB bus driver USBD. Because of that, there is no pipe object for endpoint zero and there are no pipe objects available until the device is configured.

In order to access a pipe the application has to create a handle by opening the device object as described above and attach it to a pipe. This operation is called “bind”. After a binding is successfully established the application can use the handle to communicate with the endpoint that the pipe object represents. Each pipe maybe bound only once, and a handle may be bound to one pipe only. So there is always an one-to-one relation of pipe handles and pipe objects. This means that the application has to create a separate handle for each pipe it wants to access.

The USBIO driver also supports an “unbind” operation. That is used to delete a binding between a handle and a pipe. After an unbind is performed the handle may be reused to bind another pipe object and the pipe object can be used to establish a binding with another handle.

The following example is intended to explain the relationships described above. In Figure E-2 a configuration is shown where one device object and two associated pipe objects exist within the USBIO database.
The device object is identified by a device name as described in **E.4.1.1 USBIO Device Objects**. A pipe object is identified by its endpoint address that also includes the direction flag at bit 7 (MSB). Pipe 0x81 is an IN pipe (transfer direction from device to host) and pipe 0x02 is an OUT pipe (transfer direction from host to device). The application has created three handles for the device by calling `CreateFile()`.

Handle1 is not bound to any pipe, therefore it can be used to perform device-related operations only. It is called a device handle.

Handle2 is bound to the IN pipe 0x81. By using this handle with the Win32 function `ReadFile()` the application can initiate data transfers from endpoint 0x81 to its buffers.

Handle3 is bound to the OUT pipe 0x02. By using Handle3 with the function `WriteFile()` the application can initiate data transfers from its buffers to endpoint 0x02 of the device.
Handle2 and Handle3 are called pipe handles. Note that while Handle1 cannot be used to communicate with a pipe, any operation on the device can be executed by using Handle2 or Handle3, too.

E.4.2 Establishing a Connection to the Device

The following code sample demonstrates the steps that are necessary at the USBIO API to establish a handle for a device and a pipe. The code is not complete, no error handling is included.

```c
// include the interface header file of USBIO.SYS
#include "usbio_i.h"

// device instance number
#define DEVICE_NUMBER 0

// some local variables
HANDLE FileHandle;
USBIO_SET_CONFIGURATION SetConfiguration;
USBIO_BINDPIPE BindPipe;
HDEVINFO DevInfo;
GUID g_UsbioID = USBIO_IID;
SP_DEVICE_INTERFACE_DATA DevData;
SP_INTERFACE_DEVICE_DETAIL_DATA *DevDetail = NULL;
DWORD ReqLen;
DWORD BytesReturned;

// enumerate the devices
// get a handle to the device list
DevInfo = SetupDiGetClassDevs (&g_UsbioID,
 NULL,NULL,DIGCF_DEVICEINTERFACE|DIGCF_PRESENT);
// get the device with index DEVICE_NUMBER
SetupDiEnumDeviceInterfaces (DevInfo, NULL,
 &g_usbioID, DEVICE_NUMBER, &DevData );
// get length of detailed information
SetupDiGetDeviceInterfaceDetail (DevInfo, &DevData, NULL,
 0, &ReqLen, NULL);
// allocate a buffer
DevDetail = (SP_INTERFACE DEVICE DETAIL DATA*) malloc (ReqLen);
// now get the detailed device information
DevDetail->cbSize = sizeof(SP_INTERFACE_DEVICE DETAIL DATA);
SetupDiGetDeviceInterfaceDetail (DevInfo, &DevData, DevDetail,
 ReqLen, &ReqLen, NULL);
// open the device, use OVERLAPPED flag if necessary
// use DevDetail->DevicePath as device name
FileHandle = CreateFile(
  DevDetail->DevicePath,
  GENERIC_READ|GENERIC_WRITE,
  FILE_SHARE_WRITE|FILE_SHARE_READ,
  NULL,
  OPEN_EXISTING,
  0 /* or FILE_FLAG_OVERLAPPED */,
  NULL);
```
// setup the data structure for configuration
// use the configuration descriptor with index 0
SetConfiguration.ConfigurationIndex = 0;
// device has 1 interface
SetConfiguration.NbOfInterfaces = 1;
// first interface is 0
SetConfiguration.InterfaceList[0].InterfaceIndex = 0;
// alternate setting for first interface is 0
SetConfiguration.InterfaceList[0].AlternateSettingIndex = 0;
// maximum buffer size for read/write operation is 4069 bytes
SetConfiguration.InterfaceList[0].MaximumTransferSize = 4096;

// configure the device
DeviceIoControl(FileHandle,
    IOCTL_USBIO_SET_CONFIGURATION,
    &SetConfiguration, sizeof(SetConfiguration),
    NULL, 0,
    &BytesReturned,
    NULL
);

// setup the data structure to bind the file handle
BindPipe.EndpointAddress = 0x81; // the device has an endpoint 0x81
// bind the file handle
DeviceIoControl(FileHandle,
    IOCTL_USBIO_BIND_PIPE
    &BindPipe, sizeof(BindPipe),
    NULL, 0,
    &BytesReturned,
    NULL
);

// read (or write) data from (to) the device
// use OVERLAPPED structure if necessary
ReadFile(FileHandle, ...);

// close file handle
CloseHandle(FileHandle);

Refer to the Win32 API documentation for the syntax and the parameters of the functions SetupDiXxx, CreateFile, DeviceIoControl, ReadFile, WriteFile, CloseHandle. The file handle can be opened with the FILE_FLAG_OVERLAPPED flag if asynchronous behaviour is required.

More code samples that show the usage of the USBIO programming interface can be found in the USBIO Class Library (USBIOILIB), the USBIO demo application (USBIOAPP), and the simple console applications ReaderCpp and ReadPipe.
E.4.3 Power Management

Windows 98, Windows Millennium, and Windows 2000 support system power management. That means that if the computer is idle for a given time, some parts of the computer can go into a sleeping mode. A system power change can be initiated by the user or by the operating system itself, on a low battery condition for example. An USB device driver has to support the system power management. Each device which supports power switching has to have a device power policy owner. It is responsible for managing the device power states in response to system power state changes. The USBIO driver is the power policy owner of the USB devices that it controls. In addition to the system power changes the device power policy owner can initiate device power state changes.

Before the system goes into a sleep state the operating system asks every driver if its device can go into the sleep state. If all active drivers return success the system goes down. Otherwise, a message box appears on the screen and informs the user that the system is not able to go into the sleeping mode.

Before the system goes into a sleeping state the driver has to save all the information that it needs to reinitialize the device (device context) if the system is resumed. Furthermore, all pending requests have to be completed and further requests have to be queued. In the device power states D1 or D2 (USB Suspend) the device context stored in the USB device will be lost. Therefore, a device sleeping state D1 or D2 is handled transparent for the application. In the state D3 (USB Off) the device context is lost. Because the information stored in the device is known to the application only (e.g. the current volume level of an audio device), the generic USBIO driver cannot restore the device context in a general way. This has to be done by the application. Note that Windows 2000 restores the USB configuration of the device (SET_CONFIGURATION request) after the system is resumed.

The behaviour with respect to power management can be customized by registry parameters. For example, if a long time measurement should be performed the computer has to be prevented from going power down. For a description of the supported registry parameters, see E.9 Registry Entries.
All registry entries describing device power states are DWORD parameters where the value 0 corresponds to `DevicePowerD0`, 1 to `DevicePowerD1`, and so on.

The parameter `PowerStateOnOpen` specifies the power state to which the device is set if the first file handle is opened. If the last file handle is closed the USB device is set to the power state specified in the entry `PowerStateOnClose`.

If at least one file handle is open for the device the key `MinPowerStateUsed` describes the minimal device power state that is required. If the value is set to 0 the computer will never go into a sleep state. If this key is set to 2 the device can go into a suspend state but not into D3 (Off). A power-down request caused by a low battery condition cannot be suppressed by using this parameter.

If no file handle is currently open for the device, the key `MinPowerStateUnused` defines the minimal power state the device can go into. Thus, its meaning is similar to that of the parameter `MinPowerStateUsed`.

If the parameter `AbortPipesOnPowerDown` is set to 1 all pending requests submitted by the application are returned before the device enters a sleeping state. This switch should be set to 1 if the parameter `MinPowerStateUsed` is different from D0. The pending I/O requests are returned with the error code `USBIO_ERR_POWER_DOWN`. This signals to the application that the error was caused by a power down event. The application may ignore this error and repeat the request. The re-submitted requests will be queued by the USBIO driver. They will be executed after the device is back in state D0.
E.4.4 Device State Change Notifications

The application is able to receive notifications when the state of an USB device changes. The Win32 API provides the function `RegisterDeviceNotification` for this purpose. This way, an application will be notified if an USB device is plugged in or removed.

Please refer to the Microsoft Platform SDK documentation for detailed information on the functions `RegisterDeviceNotification` and `UnregisteredDeviceNotification`. In addition, the source code of the USBIO demo application USBIOAPP provides an usage example.

The device notification mechanism is only available if the USBIO device naming scheme is based on Device Interface IDs (GUIDs). See E.4.1.1 USBIO Device Objects for details. We strongly recommend to use this new naming scheme.

**NOTE:** The function `UnregisteredDeviceNotification` should not be used on Windows 98. There is a bug in the implementation that causes the system to become unstable. So it may crash at some later point in time. The bug seems to be “well known”, it was discussed in some Usenet groups.
E.5 Programming Interface

E.5.1 Programming Interface Overview

<table>
<thead>
<tr>
<th>Operation</th>
<th>Used On</th>
<th>Bus Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOCTL_USBIO_GET_DRIVER_INFO</td>
<td>device —</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_GET_DESCRIPTOR</td>
<td>device request on default pipe</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_SET_DESCRIPTOR</td>
<td>device request on default pipe</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_SET_FEATURE</td>
<td>device request on default pipe</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_CLEAR_FEATURE</td>
<td>device request on default pipe</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_GET_STATUS</td>
<td>device request on default pipe</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_GET_CONFIGURATION</td>
<td>device request on default pipe</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_GET_INTERFACE</td>
<td>device request on default pipe</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR</td>
<td>device —</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_SET_CONFIGURATION</td>
<td>device request on default pipe</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_UNCONFIGURE_DEVICE</td>
<td>device request on default pipe</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_SET_INTERFACE</td>
<td>device request on default pipe</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST</td>
<td>device request on default pipe</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST</td>
<td>device request on default pipe</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_GET_DEVICE_PARAMETERS</td>
<td>device —</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_SET_DEVICE_PARAMETERS</td>
<td>device —</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_GET_CONFIGURATION_INFO</td>
<td>device —</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_RESET_DEVICE</td>
<td>device reset on hub port, USBD assigns USB address</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER</td>
<td>device —</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_GET_DEVICE_POWER_STATE</td>
<td>device —</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_SET_DEVICE_POWER_STATE</td>
<td>device set properties on hub port</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_BIND_PIPE</td>
<td>device —</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_UNBIND_PIPE</td>
<td>pipe —</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_RESET_PIPE</td>
<td>pipe request on default pipe</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_ABORT_PIPE</td>
<td>pipe —</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_GET_PIPE_PARAMETERS</td>
<td>pipe —</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_SET_PIPE_PARAMETERS</td>
<td>pipe —</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN</td>
<td>pipe request on pipe</td>
<td></td>
</tr>
<tr>
<td>IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT</td>
<td>pipe request on pipe</td>
<td></td>
</tr>
<tr>
<td>ReadFile</td>
<td>pipe data transfer from pipe (IN)</td>
<td></td>
</tr>
<tr>
<td>WriteFile</td>
<td>pipe data transfer to pipe (OUT)</td>
<td></td>
</tr>
</tbody>
</table>
E.5.2 Control Requests

This section provides a detailed description of the I/O Control operations the USBIO driver supports through its programming interface. The I/O Control requests are submitted to the driver using the Win32 function DeviceIoControl (see E.4 Architecture). The DeviceIoControl function is defined as follows:

```c
BOOL DeviceIoControl ( 
    HANDLE hDevice, // handle to device of interest
    DWORD dwIoControlCode, // control code of operation to perform
    LPVOID lpInBuffer, // pointer to buffer to supply input data
    DWORD nInBufferSize, // size of input buffer
    LPVOID lpOutBuffer, // pointer to buffer to receive output data
    DWORD nOutBufferSize, // size of output buffer
    LPDWORD lpBytesReturned, // pointer to variable to receive
    // output byte count
    LPOVERLAPPED lpOverlapped // pointer to overlapped structure
    // for asynchronous operation
); 
```

Refer to the Microsoft Platform SDK documentation for more information.

The following sections describe the I/O Control codes that may be passed to the DeviceIoControl function as dwIoControlCode and the parameters required for lpInBuffer, nInBufferSize, lpOutBuffer, nOutBufferSize.
**IOCTL_USBIO_GET_DESCRIPTOR**

The IOCTL_USBIO_GET_DESCRIPTOR operation requests a specific descriptor from the device.

- **lpInBuffer** Pointer to a buffer that contains an **USBIO_DESCRIPTOR_REQUEST** (page 187) data structure. This data structure has to be filled completely by the caller.

- **nInBufferSize** Specifies the size, in bytes, of the buffer point to by lpInBuffer, which has to be sizeof(USBIO_DESCRIPTOR_REQUEST) for this operation.

- **lpOutBuffer** Pointer to a buffer that will receive the descriptor data.

- **nOutBufferSize** Specifies the size in bytes, of the buffer pointed to be lpOutBuffer.

**Comments**

The buffer that is passed to this function in lpOutBuffer should be large enough to hold the requested descriptor, otherwise only a part of the descriptor will be returned. The size of the output buffer should be a multiple of the packet size of the default pipe (endpoint zero).
The IOCTL_USBIO_SET_DESCRIPTOR operation sets a specific descriptor of the device.

- **lpInBuffer** Pointer to a buffer that contains an USBIO_DESCRIPTOR_REQUEST (page 187) data structure. This data structure has to be filled completely by the caller.
- **nInBufferSize** Specifies the size, in bytes, of the buffer pointed to by lpInBuffer, which has to be sizeof(USBIO_DESCRIPTORREQUEST) for this operation.
- **lpOutBuffer** Pointer to a buffer that contains the descriptor data to be set.
- **nOutBufferSize** Specifies the size, in bytes, of the buffer pointed to by lpOutBuffer.

**Comments** USB devices do not have to support this operation.
IOCTL_USBIO_SET_FEATURE

The IOCTL_USBIO_SET_FEATURE operation is used to set or enable a specific feature.

- **lpInBuffer**: Pointer to a buffer that contains an **USBIO_FEATURE_REQUEST** (page 188) data structure. This data structure has to be filled completely by the caller.

- **nInBufferSize**: Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`, which has to be `sizeof(USBIO_FEATURE_REQUEST)` for this operation.

- **lpOutBuffer**: Not used with this operation. Set to NULL.

- **nOutBufferSize**: Not used with this operation. Set to zero.

**Comments**: The SET_FEATURE request appears on the bus with the parameters specified in the IOCTL_USBIO_SET_FEATURE structure.
IOCTL_USBIO_CLEAR_FEATURE

The IOCTL_USBIO_CLEAR_FEATURE operation is used to clear or disable a specific feature.

- **lpInBuffer**: Pointer to a buffer that contains an **USBIO_FEATURE_REQUEST** (page 188) data structure. This data structure has to be filled completely by the caller.
- **nInBufferSize**: Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`, which has to be sizeof(USBIO_FEATURE_REQUEST) for this operation.
- **lpOutBuffer**: Not used with this operation. Set to NULL.
- **nOutBufferSize**: Not used with this operation. Set to zero.

**Comments**: The CLEAR_FEATURE request appears on the bus with the parameters specified in the IOCTL_USBIO_CLEAR_FEATURE structure.
The IOCTL_USBIO_GET_STATUS operation requests status for a specific recipient.

- **lpInBuffer**: Pointer to a buffer that contains an `USBIO_STATUS_REQUEST` (page 189) data structure. This data structure has to be filled completely by the caller.

- **nInBufferSize**: Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`, which has to be `sizeof(USBIO_STATUS_REQUEST)` for this operation.

- **lpOutBuffer**: Pointer to a buffer that will receive an `USBIO_STATUS_REQUEST_DATA` (page 190) data structure.

- **nOutBufferSize**: Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`, which has to be at least `sizeof(USBIO_STATUS_REQUEST_DATA)` for this operation.

**Comments**
The GET_STATUS request appears on the bus with the parameters specified in the `USBIO_STATUS_REQUEST` (page 189) structure. The function returns the structure `USBIO_STATUS_REQUEST_DATA` (page 190) which contains two bytes of data.
**IOCTL_USBIO_GET_CONFIGURATION**

The IOCTL_USBIO_GET_CONFIGURATION operation returns the current configuration of the device.

- **lpInBuffer** Not used with this operation. Set to NULL.
- **nInBufferSize** Not used with this operation. Set to zero.
- **lpOutBuffer** Pointer to a buffer that will receive an **USBIO_GET_CONFIGURATION_DATA** (page 191) data structure.
- **nOutBufferSize** Specifies the size, in bytes, of the buffer pointed to by **lpOutBuffer**, which has to be at least sizeof(USBIO_GET_CONFIGURATION_DATA) for this operation.

**Comments**

A GET_CONFIGURATION request appears on the bus. The structure **USBIO_GET_CONFIGURATION_DATA** (page 191) returns the configuration value. A value of zero means “not configured”.
IOCTL_USBIO_GET_INTERFACE

The IOCTL_USBIO_GET_INTERFACE operation returns the current alternate setting of a specific interface.

**lpInBuffer** Pointer to a buffer that contains an **USBIO_GET_INTERFACE** (page 192) data structure. This data structure has to be filled completely by the caller.

**nInBufferSize** Specifies the size, in bytes, of the buffer pointed to by **lpInBuffer**, which has to be sizeof(USBIO_GET_INTERFACE) for this operation.

**lpOutBuffer** Pointer to a buffer that will receive an **USBIO_GET_INTERFACE_DATA** (page 193) data structure.

**nOutBufferSize** Specifies the size, in bytes, of the buffer pointed to by **lpOutBuffer**, which has to be at least sizeof(USBIO_GET_INTERFACE_DATA) for this operation.

**Comments** A GET_INTERFACE request appears on the bus. The structure **USBIO_GET_INTERFACE_DATA** (page 193) returns the current alternate setting of the interface specified in USBIO_GET_INTERFACE.
The `IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR` operation stores the configuration descriptor to be used for set configuration requests within the USBIO device driver.

- **lpInBuffer**: Pointer to a buffer that contains the configuration descriptor data.
- **nInBufferSize**: Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`.
- **lpOutBuffer**: Not used with this operation. Set to NULL.
- **nOutBufferSize**: Not used with this operation. Set to zero.

### Comments

This request may be used to store an user-defined configuration descriptor within the USBIO driver. The stored descriptor is used by the USBIO driver in subsequent `IOCTL_USBIO_SET_CONFIGURATION` operations. The usage of `IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR` is optional. If no user-defined configuration descriptor is stored, USBIO uses the descriptor from the device.

There may be cases where the USBD driver provided by Microsoft with Windows does not process correctly the configuration descriptor that is reported by the device. This means it would not be possible to configure the device. In this situation the `IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR` request may be used to work around the problem. This request enables the application to use a modified configuration descriptor. The application can get the configuration descriptor using `ISOCTL_USBIO_GET_DESCRIPTOR` (page 151), modify it appropriately and store it in the USBIO driver using the `IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR` request. Thus, the modified configuration descriptor will be passed to USBD when the device is configured.

The following is an example for the problem described above:

In the endpoint descriptor of an audio device the `bmAttributes` field contains two additional bits of information as defined by the audio class specification. The USBD does not recognize the pipe correctly and returns an invalid pipe type, when the additional bits in `bmAttributes` are not masked off. This has to be done by the application.
The IOCTL_USBIO_SET_CONFIGURATION operation is used to set the device configuration.

**lpInBuffer**
- Pointer to a buffer that contains an USBIO_SET_CONFIGURATION (page 195) data structure. This data structure has to be filled completely by the caller.

**nInBufferSize**
- Specifies the size, in bytes, of the buffer pointed to by **lpInBuffer**, which has to be sizeof(USBIO_SET_CONFIGURATION) for this operation.

**lpOutBuffer**
- Not used with this operation. Set to NULL.

**nOutBufferSize**
- Not used with this operation. Set to zero.

**Comments**
- A SET_CONFIGURATION request appears on the bus. The USBD generates additional SET_INTERFACE requests on the bus if necessary.

All available interfaces have to be configured, or the request will fail. The number of interfaces and the alternate setting for each interface have to be specified in the structure USBIO_SET_CONFIGURATION (page 195).

All pipe handles associated with the device will be unbound and all pending requests will be cancelled. If this request returns with success, new pipe objects are available. The operation IOCTL_USBIO_GET_CONFIGURATION_INFO (page 166) may be used to query all available pipes and interfaces.

By default, the configuration descriptor that is reported by the device is passed to the USBD. If an user-defined configuration descriptor is stored with IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR (page 158), this descriptor is used.
IOCTL_USBIO_UNCONFIGURE_DEVICE

The IOCTL_USBIO_UNCONFIGUREDEVICE operation is used to set the device to its unconfigured state.

lpInBuffer Not used with this operation. Set to NULL.
nInBufferSize Not used with this operation. Set to zero.
lpOutBuffer Not used with this operation. Set to NULL.
nOutBufferSize Not used with this operation. Set to zero.

Comments A SET_CONFIGURATION request with the configuration value 0 appears on the bus. All pipe handles associated with the device will be unbound and all pending requests will be cancelled.
IOCTL_USBIO_SET_INTERFACE

The IOCTL_USBIO_SET_INTERFACE operation sets the alternate setting of a specific interface.

- **lpInBuffer**
  - Pointer to a buffer that contains an USBIO_INTERFACE_SETTING (page 194) data structure. This data structure has to be filled completely by the caller.

- **nInBufferSize**
  - Specifies the size, in bytes, of the buffer pointed to by lpInBuffer, which has to be sizeof(USBIO_INTERFACE_SETTING) for this operation.

- **lpOutBuffer**
  - Not used with this operation. Set to NULL.

- **nOutBufferSize**
  - Not used with this operation. Set to zero.

**Comments**

A SET_INTERFACE request appears on the bus.

All pipe handles associated with the interface will be unbound and all pending requests will be cancelled. If this request returns with success, new pipe objects are available. The operation IOCTL_USBIO_GET_CONFIGURATION_INFO (page 166) may be used to query all available pipes and interfaces.
The IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST operation is used to generate a class or vendor specific device request with a data transfer direction from device to host.

- **lpInBuffer**
  Pointer to a buffer that contains an USBIO_CLASS_OR_VENDOR_REQUEST (page 196) data structure. This data structure has to be filled completely by the caller.

- **nInBufferSize**
  Specifies the size, in bytes, of the buffer pointed to by lpInBuffer, which has to be sizeof(USBIO_CLASS_OR_VENDOR_REQUEST) for this operation.

- **lpOutBuffer**
  Pointer to a buffer that receives the data transferred from the device during the data phase of the control transfer. If the request does not return any data, this value can be NULL.

- **nOutBufferSize**
  Specifies the size, in bytes, of the buffer pointed to by lpOutBuffer. If this value is set to zero then there is no data transfer phase.

**Comments**
A SETUP request appears on the default pipe (endpoint zero) of the USB device with the given parameters. If a data phase is required an IN token appears on the bus and the successful transfer is acknowledged by an OUT token with a zero length data packet. If no data phase is required an IN token appears on the bus with a zero length data packet from the USB device for acknowledge.
The IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST operation is used to generate a class or vendor specific device request with a data transfer direction from host to device.

**lpInBuffer**
Pointer to a buffer that contains an USBIO_CLASS_OR_VENDOR_REQUEST (page 196) data structure.
This data structure has to be filled completely by the caller.

**nInBufferSize**
Specifies the size, in bytes, of the buffer pointed to by lpInBuffer, which has to be sizeof(USBIO_CLASS_OR_VENDOR_REQUEST) for this operation.

**lpOutBuffer**
Pointer to a buffer that contains the data to be transferred to the device during the data phase of the control transfer. If the request has no data phase this value can be NULL.

**nOutBufferSize**
Specifies the size, in bytes, of the buffer pointed to by lpOutBuffer. If this value is set to zero then there is no data transfer phase.

**Comments**
A SETUP request appears on the default pipe (endpoint zero) of the USB device with the given parameters. If a data phase is required an OUT token appears on the bus and the successful transfer is acknowledged by an IN token with a zero length data packet from the device. If no data phase is required an IN token appears on the bus and the device acknowledges with a zero length data packet.
**IOCTL_USBIO_GETDEVICE_PARAMETERS**

The IOCTL_USBIO_GETDEVICE_PARAMETERS operation returns USBIO settings related to a device.

- **lpInBuffer**: Not used with this operation. Set to NULL.
- **nInBufferSize**: Not used with this operation. Set to zero.
- **lpOutBuffer**: Pointer to a buffer that will receive an **USBIO_DEVICE_PARAMETERS** (page 198) data structure.
- **nOutBufferSize**: Specifies the size, in bytes, of the buffer pointed to by **lpOutBuffer**, which has to be at least sizeof(USBIO_DEVICE_PARAMETERS) for this operation.

**Comments**: The default state of the device parameters is defined by a set of registry parameters which are read by the USBIO driver at startup. The current state may be queried using this request.
IOCTL_USBIO_SET_DEVICE_PARAMETERS

The IOCTL_USBIO_SET_DEVICE_PARAMETERS operation is used to set USBIO parameters related to a device.

**lpInBuffer**
- Pointer to a buffer that contains an **USBIO_DEVICE_PARAMETERS** (page 198) data structure. This data structure has to be filled completely by the caller.

**nInBufferSize**
- Specifies the size, in bytes, of the buffer pointed to be *lpInBuffer*, which has to be sizeof(USBIO_DEVICE_PARAMETERS) for this operation.

**lpOutBuffer**
- Not used with this operation. Set to NULL.

**nOutBufferSize**
- Not used with this operation. Set to zero.

**Comments**
- The default state of the device parameters is defined by a set of registry parameters which are read by the USBIO driver at startup. The current state may be modified using this request.
IOCTL_USBIO_GET.Configuration_INFO

The IOCTL_USBIO_GET_CONFIGURATION_INFO operation returns information about the pipes and interfaces that are available after the device is configured.

lpInBuffer  Not used with this operation. Set to NULL.

nInBufferSize Not used with this operation. Set to zero.

lpOutBuffer Pointer to a buffer that will receive an USBIO_CONFIGURATION_INFO data structure.

nOutBufferSize Specifies the size, in bytes, of the buffer pointed to by lpOutBuffer, which has to be at least sizeof(USBIO_CONFIGURATION_INFO) for this operation.

Comments This operation returns information about all active pipes and interfaces that are available in the current configuration.
IOCTL_USBIO_RESET_DEVICE

The IOCTL_USBIO_RESET_DEVICE operation causes a reset at the hub port in which the device is plugged in.

lpInBuffer Not used with this operation. Set to NULL.
nInBufferSize Not used with this operation. Set to zero.
lpOutBuffer Not used with this operation. Set to NULL.
nOutBufferSize Not used with this operation. Set to zero.

Comments The following events occur on the bus if this request is issued:

USB Reset
GET_DEVICE_DESCRIPTOR
USB Reset
SET_ADDRESS
GET_DEVICE_DESCRIPTOR
GET_CONFIGURATION_DESCRIPTOR

All pipes associated with the device will be unbound and all pending requests will be cancelled. Note that the device receives two USB Resets and a new USB address is assigned by USBD. After this operation the device is in the unconfigured state.

The USBIO driver allows an USB reset request only if the device is configured. That means IOCTL_USBIO_SET_CONFIGURATION (page 159) was successfully executed. If the device is in the unconfigured state this request returns with an error status. This limitation is caused by the behaviour of Windows 2000. A system crash would occur on Windows 2000 if an USB Reset would be issued for an unconfigured device. Therefore, USBIO does not allow to issue an USB Reset while the device is configured.

If the device changes its USB descriptor set during an USB Reset the IOCTL_USBIO_CYCLE_PORT (page 172) request should be used instead of IOCTL_USBIO_RESET_DEVICE.

This request does not work if the system-provided multi-interface driver is used.
IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER

The IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER operation returns the current value of the frame number counter that is maintained by the USBD.

*lpInBuffer*: Not used with this operation. Set to Null.

*nInBufferSize*: Not used with this operation. Set to zero.

*lpOutBuffer*: Pointer to a buffer that will receive an USBIO_FRAME_NUMBER (page 205) data structure.

*nOutBufferSize*: Specifies the size, in bytes, of the buffer pointed to by *lpOutBuffer*, which has to be at least sizeof(USBIO_FRAME_NUMBER) for this operation.

*Comments*: The returned frame number is a 32 bit value. The lower 11 bits of this value correspond to the frame number value in the Start Of Frame token on the bus.
IOCTL_USBIO_SET_DEVICE_POWER_STATE

The IOCTL_USBIO_SET_DEVICE_POWER_STATE operation sets the power state of the device.

- **lpInBuffer**: Pointer to a buffer that contains an **USBIODEVICE_POWER** (page 206) data structure.

- **nInBufferSize**: Specifies the size, in bytes, of the buffer pointed to by **lpInBuffer**, which has to be at least sizeof(USBIODEVICE_POWER) for this operation.

- **lpOutBuffer**: Not used with this operation. Set to NULL.

- **nOutBufferSize**: Not used with this operation. Set to zero.

**Comments**

The device power state is maintained internally by the USBIO driver. This request may be used to change the current power state. If the device is set to a power state different from D0 all pending requests should be cancelled before.

**See Also**

See E.4.3 Power Management (page 146) and the description of the data structure **USBIODEVICE_POWER** (page 206) for details.
IOCTL_USBIO_GET_DEVICE_POWER_STATE

The IOCTL_USBIO_GETDEVICE_POWER_STATE operation returns the current power state of the device.

lpInBuffer Not used with this operation. Set to NULL.

nInBufferSize Not used with this operation. Set to zero.

lpOutBuffer Pointer to a buffer that will receive an USBIO_DEVICE_POWER (page 206) data structure.

nOutBufferSize Specifies the size, in bytes, of the buffer pointed to by lpOutBuffer, which has to be at least sizeof(USBIO_DEVICE_POWER) for this operation.

Comments The device power state is maintained internally by the USBIO driver. This request may be used to query the current power state.
**IOCTL_USBIO_GET_DRIVER_INFO**

The IOCTL_USBIO_GET_DRIVER_INFO operation returns version information about the USBIO API and the driver binary that is currently running.

- *lpInBuffer* Not used with this operation. Set to NULL.
- *nInBufferSize* Not used with this operation. Set to zero.
- *lpOutBuffer* Pointer to a buffer that will receive an `USBIO_DRIVER_INFO` (page 186) data structure.
- *nOutBufferSize* Specifies the size, in bytes, of the buffer pointed to by *lpOutBuffer*, which has to be at least sizeof(USBIO_DRIVER_INFO) for this operation.

*Comments* An application should check if the API version of the driver that is currently running matches with the version it expects.
The IOCTL_USBIO_CYCLE_PORT operation causes a new enumeration of the device.

- **lpInBuffer**: Not used with this operation. Set to NULL.
- **nInBufferSize**: Not used with this operation. Set to zero.
- **lpOutBuffer**: Not used with this operation. Set to NULL.
- **nOutBufferSize**: Not used with this operation. Set to zero.

**Comments**

The IOCTL_USBIO_CYCLE_PORT request is similar to the IOCTL_USBIO_RESET_DEVICE (page 167) request, except that from the software point of view a disconnect/connect is simulated. This request causes the following events to occur:

- The USBIO device instance that is associated with the USB device will be removed. The corresponding device handles become invalid and should be closed by the application.
- The operating system starts a new enumeration of the device. The following events occur on the bus:
  - USB Reset
  - GET_DEVICE_DESCRIPTOR
  - USB Reset
  - SET_ADDRESS
  - GET_DEVICE_DESCRIPTOR
  - GET_CONFIGURATION_DESCRIPTOR
- A new device instance is created by the USBIO driver.
- The application receives a PnP notification that informs it about the new device instance.

After an application issued this request it should close all handles for the current device. It can open the newly created device instance after it receives the appropriate PnP notification.
This request should be used instead of 
IOCTL_USBIO_RESET_DEVICE (page 167) if the USB device modifies 
its descriptors during an USB Reset. Particularly, this is required to 
implement the Device Firmware Upgrade (DFU) device class 
specification. Note that the USB device receives two USB Resets after 
this call. This does not conform to the DFU specification. However, this 
is the standard device enumeration method used by the Windows USB 
business driver (USBD).

This request does not work if the system-provided multi-interface driver 
is used. This driver expects that all function device drivers send a 
CYCLE_PORT request within 5 seconds.
IOCTL_USBIO_BINDPIPE

The IOCTL_USBIO_BIND_PIPE operation is used to establish a binding between a file handle and a pipe object.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lpInBuffer</td>
<td>Pointer to a buffer that contains an USBIO_BIND_PIPE (page 207) data structure. This data structure has to be filled completely by the caller.</td>
</tr>
<tr>
<td>nInBufferSize</td>
<td>Specifies the size, in bytes, of the buffer pointed to by lpInBuffer, which has to be sizeof(USBIO__BIND_PIPE) for this operation.</td>
</tr>
<tr>
<td>lpOutBuffer</td>
<td>Not used with this operation. Set to NULL.</td>
</tr>
<tr>
<td>nOutBufferSize</td>
<td>Not used with this operation. Set to zero.</td>
</tr>
</tbody>
</table>

Comments

This pipe is identified by its endpoint address. Only endpoints from the current configuration can be bound. After this operation is successfully completed the pipe can be accessed using pipe related requests (e.g. read or write).

See Also

IOCTL_USBIO_SET_CONFIGURATION (page 159) and IOCTL_USBIO_GET_CONFIGURATION_INFO (page 166)
IOCTL_USBIO_UNBIND_PIPE

The IOCTL_USBIO_UNBIND_PIPE operation deletes the binding between a file handle and a pipe object.

- *lpInBuffer*: Not used with this operation. Set to NULL.
- *nInBufferSize*: Not used with this operation. Set to zero.
- *lpOutBuffer*: Not used with this operation. Set to NULL.
- *nOutBufferSize*: Not used with this operation. Set to zero.

*Comments*: After this operation is successfully completed the handle is unbound and may be used to bind another pipe. It is not necessary to unbind a pipe handle before it is closed. Closing a handle unbinds it implicitly.
ICOTL_USBIO_RESETPIPE

The IOCTL_USBIO_RESET_PIPE operation clears an error condition on a pipe.

lpInBuffer: Not used with this operation. Set to NULL.
nInBufferSize: Not used with this operation. Set to zero.
lpOutBuffer: Not used with this operation. Set to NULL.
nOutBufferSize: Not used with this operation. Set to zero.

Comments: If an error occurs while transferring data from or to a pipe the USBD halts the pipe and returns an error code. No further transfers can be performed while the pipe is halted. To recover from this error condition and to restart the pipe an IOCTL_USBIO_RESET_PIPE has to be sent to the pipe.

This request causes that a CLEAR_FEATURE(ENDPOINTSTALL) request appears on the bus. In addition, the endpoint handling in the USB host controller will be reinitialized.

Isochronous pipes will never be halted by the USBD. This is because on isochronous pipes no handshake is used to detect errors in data transmission.
The IOCTL_USBIO_ABORT_PIPE operation causes that all outstanding requests for the pipe are cancelled.

- **lpInBuffer**: Not used with this operation. Set to NULL.
- **nInBufferSize**: Not used with this operation. Set to zero.
- **lpOutBuffer**: Not used with this operation. Set to NULL.
- **nOutBufferSize**: Not used with this operation. Set to zero.

**Comments**: All outstanding read or write requests on the pipe are aborted and returned with an error status of **USBIO_ERR_CANCELLED**.
IOCTL_USBIO_GET_PIPE_PARAMETERS

The IOCTL_USBIO_GET_PIPE_PARAMETERS operation returns USBIO settings related to a pipe.

*lpInBuffer* Not used with this operation. Set to NULL.

*lpOutBuffer* Pointer to a buffer that will receive an *USBIO_PIPE_PARAMETERS* (page 208) data structure.

*nOutBufferSize* Specifies the size, in bytes, of the buffer pointed to by *lpOutBuffer*, which has to be at least sizeof(*USBIO_PIPE_PARAMETERS*) for this operation.

*Comments* The default state of the pipe parameters is defined by a set of registry parameters which are read by the USBIO driver at startup. The current state can be queried by using this request.
**IOCTL_USBIO_SET_PIPE_PARAMETERS**

The IOCTL_USBIO_SET_PIPE_PARAMETERS operation is used to set USBIO parameters related to a pipe.

- **lpInBuffer**: Pointer to a buffer that contains an USBIO_PIPE_PARAMETERS (page 208) data structure. This data structure has to be filled completely by the caller.

- **nInBufferSize**: Specifies the size, in bytes, of the buffer pointed to by lpInBuffer, which has to be sizeof(USBIO_PIPE_PARAMETERS) for this operation.

- **lpOutBuffer**: Not used with this operation. Set to NULL.

- **nOutBufferSize**: Not used with this operation. Set to zero.

**Comments**

The default state of the pipe parameters is defined by a set of registry parameters which are read by the USBIO driver at startup. The current state can be modified by using this request.
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN

The IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN operation is used to generate a specific request (setup packet) for a control pipe with a data transfer direction from device to host.

**lpInBuffer**  
Pointer to a buffer that contains an  
**USBIO_PIPE_CONTROL_TRANSFER** (page 209) data structure. This data structure has to be filled completely by the caller.

**nInBufferSize**  
Specifies the size, in bytes, of the buffer pointed to by lpInBuffer, which has to be sizeof(USBIO_PIPE_CONTROL_TRANSFER) for this operation.

**lpOutBuffer**  
Pointer to a buffer that receives the data transferred from the device during the data phase of the control transfer. If no data transfer is required the pointer may be NULL.

**nOutBufferSize**  
Specifies the size, in bytes, of the buffer pointed to by lpOutBuffer. If this value is set to zero then there is no data transfer phase.

**Comments**  
This request is intended to be used with additional control pipes a device might provide. It is not possible to generate a control transfer for the default endpoint zero with this operation.
IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT

The IOCTL_USBIO_PIPECONTROL_TRANSFER_OUT operation is used to generate a specific request (setup packet) for a control pipe with a data transfer direction from host to device.

**lpInBuffer** Pointer to a buffer that contains an USBIO_PIPE_CONTROL_TRANSFER (page 209) data structure. This data structure has to be filled completely by the caller.

**nInBufferSize** Specifies the size, in bytes, of the buffer pointed to by `lpInBuffer`, which has to be sizeof(USBIO_PIPE_CONTROL_TRANSFER) for this operation.

**lpOutBuffer** Pointer to a buffer that contains the data transferred to the device during the data phase of the control transfer. If no data transfer is required the pointer may be NULL.

**nOutBufferSize** Specifies the size, in bytes, of the buffer pointed to by `lpOutBuffer`. If this value is set to zero then there is no data transfer phase.

**Comments** This request is intended to be used with additional control pipes a device might provide. It is not possible to generate a control transfer for the default endpoint zero with this operation.
E.5.3 Data Transfer Requests

The USBIO device driver exports an interface to USB pipes that is similar to files. For that reason the Win32 API functions `ReadFile` and `WriteFile` are used to transfer data from or to a pipe. The handle that is associated with the USB pipe is passed as `hFile` to this function.

The `ReadFile` function is defined as follows:

```c
BOOL ReadFile ( 
    HANDLE hFile, // handle of file to read 
    LPVOID lpBuffer, // pointer to buffer that receives data 
    DWORD nNumberOfBytesToRead, // number of bytes to read 
    LPDWORD lpNumberOfBytesRead, // pointer to number of bytes read 
    LPOVERLAPPED lpOverlapped // pointer to OVERLAPPED structure 
); 
```

The `WriteFile` function is defined as follows:

```c
BOOL WriteFile ( 
    HANDLE hFile, // handle of file to write 
    LPVOID lpBuffer, // pointer to data to write to file 
    DWORD nNumberOfBytesToWrite, // number of bytes to write 
    LPDWORD lpNumberOfBytesWritten, // pointer to number of bytes written 
    LPOVERLAPPED lpOverlapped // pointer to OVERLAPPED structure 
); 
```

By using these functions it is possible to implement both synchronous and asynchronous data transfer operations. Both methods are fully supported by the USBIO driver. Refer to the Microsoft Platform SDK documentation for more information on using the `ReadFile` and `WriteFile` functions.

E.5.3.1 Bulk and Interrupt Transfers

For interrupt and bulk transfers the buffer size can be larger than the maximum packet size of the endpoint (physical FIFO size) as reported in the endpoint descriptor. But the buffer size has to be equal or smaller than the value specified in the MaximumTransferSize field of the `USBIO_INTERFACE_SETTING` (page 194) structure on the Set Configuration call.
**Bulk or Interrupt Write Transfers**

The write operation is used to transfer data from the host (PC) to the USB device. The buffer is divided into data pieces (packets) of the FIFO size of the endpoint. These packets are sent to the USB device. If the last packet of the buffer is smaller than the FIFO size a smaller data packet is transferred. If the size of the last packet of the buffer is equal to the FIFO size this packet is sent. No additional zero packet is sent automatically. To send a data packet with length zero, set the buffer length to zero and use a NULL buffer pointer.

**Bulk or Interrupt Read Transfers**

The read operation is used to transfer data from the USB device to the host (PC). The buffer is divided into data pieces (packets) of the FIFO size of the endpoint. The buffer size should be a multiple of the FIFO size. Otherwise the last transaction can cause a buffer overflow error.

![Figure E-3. Layout of an Isochronous Transfer Buffer](image-url)
A read operation will be completed if the whole buffer is filled or a short packet is transmitted. A short packet is a packet that is shorter than the FIFO size of the endpoint. For more information on receiving short packets see below. To read a data packet with a length of zero, the buffer size has to be at least one byte. A read operation with a NULL buffer will be completed with success by the system without performing a read operation of the USB.

The behavior of the read operation depends on the state of the flag USBIO_SHORT_TRANSFER_OK of the related pipe. This setting may be changed by using the IOCTL_USBIO_SETPIPE_PARAMETERS (page 179) operation. The default state is defined by the registry parameter ShortTransferOk. If the flag USBIO_SHORT_TRANSFER_OK is set a read operation that returns a data packet that is shorter than the FIFO size of the endpoint is completed with success. Otherwise, every data packet from the endpoint that is smaller than the FIFO size causes an error.

### E.5.3.2 Isochronous Transfers

For isochronous transfers the data buffer that is passed to the ReadFile or WriteFile function has to contain a header that describes the location and the size of the data packets to be transferred. Therefore, the buffer that follows the header is divided into packets. Each packet is transmitted within an USB frame (normally 1 ms). The size of the packet can be different in each frame. This allows to support any data rate of the isochronous data stream.

The structure of the data buffer is shown in Figure E-3. The buffer contains an USBIO_ISO_TRANSFER_HEADER (page 213) structure of variable size at offset zero and the data packets. The header contains an USBIO_ISO_TRANSFER (page 210) structure that provides general information about the transfer buffer. An important member of this structure is the NumberOfPackets parameter. This parameter specifies the number of data packets contained in the transfer buffer. The maximum number of packets that can be used in a single transfer is limited by the registry parameter MaxIsoPackets. Each data packet has to be described by an USBIO_ISO_PACKET (page 212) structure.
in the header. Because of that, the header contains a variable size array of \texttt{USBIO\_ISO\_PACKET} (page 212) elements.

The Offset member of the \texttt{USBIO\_ISO\_PACKET} structure specifies the byte offset of the corresponding packet relative to the beginning of the whole buffer and has to be filled by the application for write and for read transfers. The Length member defines the length, in bytes, of the packet. The packet length has to be specified by the application for write transfers and is returned by the USBIO on read transfers. The Status member is used to return the completion status of the transfer of the packet.

\textbf{Isochronous Write Transfers}

The sizes of the packets have to be less than or equal to the FIFO size of the endpoint. There must not be gaps between the data packets in the transfer buffer. The Offset and Length member of the \texttt{USBIO\_ISO\_PACKET} structures have to be initialized correctly before the transfer is started.

\textbf{Isochronous Read Transfers}

The size of each packet should be equal to the FIFO size. Otherwise a data overrun error can occur. The Offset member of the \texttt{USBIO\_ISO\_PACKET} structures has to be initialized correctly before the transfer is started. There must not be gaps between the data packets in the transfer buffer. The length of each received data packet is returned in the Length member of the corresponding \texttt{USBIO\_ISO\_PACKET} structure when the transfer completes.

\textbf{NOTE:} Because the size of the received packets may be less than the FIFO size the data packets are not arranged continuously within the transfer buffer.

\textbf{E.5.4 Input and Output Structures}

This section provides a detailed description of the data structures that are used with the various input and output requests.
USBIO_DRIVER_INFO

The USBIO_DRIVER_INFO structure contains version information about the driver binary and the programming interface.

**Definition**

typedef struct _USBIO_DRIVER_INFO{
    USHORT APIVersion;
    USHORT DriverVersion;
    ULONG DriverBuildNumber;
    ULONG Flags;
} USBIO_DRIVER_INFO;

**Members**

**APIVersion**
Contains the version number of the application programming interface (API) the driver supports.

The format is as follows: upper 8 bit = major version, lower 8 bit = minor version. The numbers are encoded in BCD format.

**DriverVersion**
Contains the version number of the driver executable.

The format is as follows: upper 8 bit = major version, lower 8-bit = minor version.

**DriverBuildNumber**
Contains the build number of the driver executable.

**Flags**
This field contains zero or any combination (bit-wise or) of the following values.

**USBIO_INFOFLAG_CHECKED_BUILD**
If this flag is set, the driver that is currently running is a checked (debug) build.

**USBIO_INFOFLAG_DEMO_VERSION**
If this flag is set, the driver that is currently running is a DEMO version that has some restrictions. Refer to ReadMe.txt for a description of the restrictions.

**USBIO_INFOFLAG_LIGHT_VERSION**
If this flag is set, the driver that is currently running is a LIGHT version that has some restrictions. Refer to ReadMe.txt for a description of the restrictions.

**Comments**
This structure is an output of the IOCTL_USBIO_GET_DRIVER_INFO (page 171) operation.
**USBIO_DESCRIPTOR_REQUEST**

The USBIO_DESCRIPTOR_REQUEST structure provides information used to get or set a descriptor.

**Definition**

```c
typedef struct _USBIO_DESCRIPTOR_REQUEST{
    USBIO_REQUEST_RECIPIENT Recipient;
    UCHAR DescriptorType;
    UCHAR DescriptorIndex;
    USHORT LanguageId;
} USBIO_DESCRIPTORgetRequest;
```

**Members**

**Recipient**

Specifies the recipient of the get or set descriptor request. The values are defined by the enumeration type `USBIO_REQUEST_RECIPIENT` (page 215).

**DescriptorType**

Specifies the type of descriptor to get or set. The values are defined by the Universal Serial Bus Specification 1.1, Chapter 9 and additional device class specifications.

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Device Descriptor</td>
</tr>
<tr>
<td>2</td>
<td>Configuration Descriptor</td>
</tr>
<tr>
<td>3</td>
<td>String Descriptor</td>
</tr>
<tr>
<td>4</td>
<td>Interface Descriptor</td>
</tr>
<tr>
<td>5</td>
<td>Endpoint Descriptor</td>
</tr>
<tr>
<td>21</td>
<td>HID Descriptor</td>
</tr>
</tbody>
</table>

**DescriptorIndex**

Specifies the index of the descriptor to get or set.

**LanguageId**

Specifies the Language ID for string descriptors. Set to zero for other descriptors.

**Comments**

This structure has to be used as an input for

- IOCTL_USBIO_GET_DESCRIPTOR (page 151) and
- IOCTL_USB_SET_DESCRIPTOR (page 152) requests.
USBIOFEATUREREQUEST

The USBIO_FEATURE_REQUEST structure provides information used to set or clear a specific feature.

**Definition**

```c
typedef struct _USBIO_FEATURE_REQUEST{
    USBIO_REQUEST_RECIPIENT Recipient;
    USHORT FeatureSelector;
    USHORT Index;
} USBIO_FEATURE_REQUEST;
```

**Members**

- **Recipient**
  Specifies the recipient of the set feature or clear feature request. The values are defined by the enumeration type `USBIO_REQUEST_RECIPIENT` (page 215).

- **FeatureSelector**
  Specifies the feature selector value for the set feature or clear feature request. The values are defined by the recipient. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

- **Index**
  Specifies the index value for the set feature or clear feature request. The values are defined by the device. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

**Comments**

This structure has to be used as an input for

- `IOCTL_USBIO_SET_FEATURE` (page 153)
- `IOCTL_USBIO_CLEAR_FEATURE` (page 154) requests.
USBIO_STATUS_REQUEST

The USBIO_STATUS_REQUEST structure provides information used to request status for a specified recipient.

**Definition**

typedef struct _USBIO_STATUS_REQUEST{
    USBIO_REQUEST_RECIPIENT Recipient;
    USHORT Index;
} USBIO_STATUS_REQUEST;

**Members**

- **Recipient**
  Specifies the recipient of the get status request. The values are defined by the enumeration type USBIO_REQUEST_RECIPIENT (page 215).

- **Index**
  Specifies the index value for the get status request. The values are defined by the device. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

**Comments**

This structure has to be used as an input for IOCTL_USBIO_GET_STATUS (page 155) requests.
**USBIO_STATUS_REQUEST_DATA**

The USBIO_STATUS_REQUEST_DATA structure contains information returned by a get status operation.

**Definition**

```c
typedef struct _USBIO_STATUS_REQUEST_DATA{
    USHORT Status;
} USBIO_STATUS_REQUEST_DATA;
```

**Member** Status

Contains the 16-bit value that is returned by the recipient in response to the get status request. The interpretation of the value is specific to the recipient. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

**Comments**

This structure is an output of IOCTL_USBIO_GET_STATUS (page 155) requests.
USBIO_GET_CONFIGURATION_DATA

The USBIO_GET_CONFIGURATION_DATA structure contains information returned by a get configuration operation.

**Definition**

```c
typedef struct _USBIO_GET_CONFIGURATION_DATA{
    UCHAR ConfigurationValue;
} USBIO_GET_CONFIGURATION_DATA;
```

**Member** ConfigurationValue

Contains the 8-bit value that is returned by the device in response to the get configuration request. The meaning of the value is defined by the device. A value of zero means the device is not configured. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

**Comments**

This structure is an output of IOCTL_USBIO_GET_CONFIGURATION (page 156) requests.
USBIO_GET_INTERFACE

The USBIO_GET_INTERFACE structure provides information used to request the current alternate setting of an interface.

**Definition**

typedef struct _USBIO_GET_INTERFACE{
    USHORT Interface;
} USBIO_GET_INTERFACE;

**Member** Interface

Specifies the interface number. The meaning is device-specific. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

**Comments**

This structure has to be used as an input for IOCTL_USBIO_GET_INTERFACE (page 157) requests.
USBIO_GET_INTERFACE_DATA

The USBIO_GET_INTERFACE_DATA structure contains information returned by a get interface operation.

**Definition**
```c
typedef struct _USBIO_GET_INTERFACE_DATA {
    UCHAR AlternateSetting;
} USBIO_GET_INTERFACE_DATA;
```

**Member** *AlternateSetting*
Contains the 8-bit value that is returned by the device in response to a get interface request. The interpretation of the value is specific to the device. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

**Comments**
This structure is an output of IOCTL_USBIO_GET_INTERFACE (page 157) requests.
**USBIO_INTERFACE_SETTING**

The USBIO_INTERFACE_SETTING structure provides information used to configure an interface and its endpoints.

**Definition**

```
typedef struct _USBIO_INTERFACE_SETTING{
    USHORT InterfaceIndex;
    USHORT AlternateSettingIndex;
    ULONG MaximumTransferSize;
} USBIO_INTERFACE_SETTING;
```

**Members**

- **InterfaceIndex**
  Specifies the interface. The value is defined by the device. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

- **AlternateSettingIndex**
  Specifies the alternate setting to be set for this interface. The value is defined by the device. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

- **MaximumTransferSize**
  Specifies the maximum size, in bytes, of data transfers to or from endpoints of this interface. The value is user-defined and is valid for all endpoints of this interface. If no special requirement exists a value of 4096 (4K) should be used.

**Comments**

This structure has to be used as an input for `IOCTL_USBIO_SET_INTERFACE` (page 161) and `IOCTL_USBIO_SET_CONFIGURATION` (page 159) requests.
USBIO_SET_CONFIGURATION

The USBIO_SET_CONFIGURATION structure provides information used to set the device configuration.

**Definition**

```c
typedef struct _USBIO_SET_CONFIGURATION{
    USHORT ConfigurationIndex;
    USHORT NbOfInterfaces;
    USBIO_INTERFACE_SETTING InterfaceList[USBIO_MAX_INTERFACES];
} USBIO_SET_CONFIGURATION;
```

**Members**

- **ConfigurationIndex**
  Specifies the configuration to be set. The meaning of the value is defined by the device. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

- **NbOfInterface**
  Specifies the number of interfaces in this configuration. This is the number of valid entries in InterfaceList.

- **InterfaceList [USBIO_MAX_INTERFACES]**
  An array of `USBIO_INTERFACE_SETTING` (page 194) structures that describes each interface in the configuration. There have to be NbOfInterfaces valid entries in this array.

**Comments**

This structure has to be used as an input for `IOCTL_USBIO_SET_CONFIGURATION` (page 159) requests.
**USBIO_CLASS_OR_VENDOR_REQUEST**

The USBIO_CLASS_OR_VENDOR_REQUEST structure provides information used to generate a class or vendor specific device request.

**Definition**

typedef struct _USBIO_CLASS_OR_VENDOR_REQUEST{
    ULONG Flags;
    USBIO_REQUEST_TYPE Type;
    USBIO_REQUEST_RECIPIENT Recipient;
    UCHAR RequestTypeReservedBits;
    UCHAR Request;
    USHORT Value;
    USHORT Index;
} USBIO_CLASS_OR_VENDOR_REQUEST;

**Members**

**Flags**
This field contains zero or the following value.

**USBIO_SHORT_TRANSFER_OK**
If this flag is set, the USBIO driver does not return an error if a data packet received from the device is shorter than the maximum packet size of the endpoint. Otherwise, a short packet causes an error condition.

**Type**
Specifies the type of the device request. The values are defined by the enumeration type **USBIO_REQUEST_TYPE** (page 216).

**Recipient**
Specifies the recipient of the device request. The values are defined by the enumeration type **USBIO_REQUEST_RECIPIENT** (page 215).

**RequestTypeReservedBits**
Specifies the reserved bits of the **bmRequestType** field of the setup packet.

**Request**
Specifies the value of the **bRequest** field of the setup packet.

**Value**
Specifies the value of the **wValue** field of the setup packet.

**Index**
Specifies the value of the **wIndex** field of the setup packet.
**Comments**

The values defined by this structure are used to generate an eight byte setup packet for the control endpoint of the device. The format of the setup packet is defined by the Universal Serial Bus Specification 1.1, Chapter 9. The meanings of the values are device dependent.

This structure has to be used as an input for

- `IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST` (page 162)
- `IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST` (page 163)

operations.
**USBIO_DEVICE_PARAMETERS**

The USBIO_DEVICE_PARAMETERS structure contains device-specific parameter settings of the USBIO driver.

**Definition**

```c
typedef struct _USBIO_DEVICE_PARAMETERS{
    ULONG Options;
    ULONG RequestTimeout;
} USBIODEVICEPARAMETERS;
```

**Members**

**Options**

This field contains zero or any combination (bit-wise or) of the following values.

**USBIO_RESET_DEVICE_ON_CLOSE**

If this option is set, the USBIO driver generates an USB device reset after the last handle to the device was closed by the application. When this option is active the **USBIO_UNCONFIGURE_ON_CLOSE** flag will be ignored.

The default state of this option is defined by the registry parameter **ResetDeviceOnClose**.

**USBIO_UNCONFIGURE_ON_CLOSE**

If this option is set, the USBIO driver sets the USB device to its unconfigured state after the last handle to the device was closed by the application.

The default state of this option is defined by the registry parameter **UnconfigureOnClose**.

**USBIO_ENABLE_REMOTE_WAKEUP**

If this option is set and the USB device supports the Remote Wakeup feature the USBIO driver will support Remote Wakeup for the operating system. The USB device is able to wake the system from a sleep state. The Remote Wakeup feature is defined by the USB 1.1 specification.

The Remote Wakeup feature requires that the device is opened by an application and an USB configuration is set (device is configured).

The default state of this option is defined by the registry parameter **EnableRemoteWakeups**.
RequestTimeout

Specifies the time-out interval, in milliseconds, to be used for synchronous operations. A value of zero means an infinite interval (time-out disabled).

The default time-out value is defined by the registry parameter RequestTimeout.

Comments

This structure is intended to be used with IOCTL_USBIO_GETDEVICE_PARAMETERS (page 164) and IOCTL_USBIO_SETDEVICE_PARAMETERS (page 165) operations.
USBIO_INTERFACE_CONFIGURATION_INFO

The USBIO_INTERFACE_CONFIGURATION_INFO structure provides information about an interface.

Definition

typedef struct _USBIO_INTERFACE_CONFIGURATION_INFO{
    UCHAR InterfaceNumber;
    UCHAR AlternateSetting;
    UCHAR Class;
    UCHAR SubClass;
    UCHAR Protocol;
    UCHAR NumberOfPipes;
    UCHAR reserved1;
    UCHAR reserved2;
} USBIO_INTERFACE_CONFIGURATION_INFO;

Members

InterfaceNumber
    Specifies the index of the interface as reported by the device in the configuration descriptor.

AlternateSetting
    Specifies the index of the alternate setting as reported by the device in the configuration descriptor. The default alternate setting of an interface is zero.

Class
    Specifies the class code as reported by the device in the configuration descriptor. The meaning of this value is defined by the USB class specifications.

SubClass
    Specifies the subclass code as reported by the device in the configuration descriptor. The meaning of this value is defined by the USB class specifications.

Protocol
    Specifies the protocol code as reported by the device in the configuration descriptor. The meaning of this value is defined by the USB class specifications.

NumberOfPipes
    Specifies the number of pipes that belong to this interface and alternate setting.
reserved1
  Reserved field, set to zero.

reserved2
  Reserved field, set to zero.

Comments
This structure is an output of
IOCTL_USBIO_GET_CONFIGURATION_INFO (page 166) operations.
The USBIO_PIPE_CONFIGURATION_INFO structure provides information about a pipe.

**Definition**

typedef struct _USBIO_PIPE_CONFIGURATION_INFO{
    USBIO_PIPE_TYPE PipeType;
    ULONG MaximumTransferSize;
    USHORT MaximumPacketSize;
    UCHAR EndpointAddress;
    UCHAR Interval;
    UCHAR InterfaceNumber;
    UCHAR reserved1;
    UCHAR reserved2;
    UCHAR reserved3;
} USBIO_PIPE_CONFIGURATION_INFO;

**Members**

PipeType

Specifies the type of the pipe. The values are defined by the enumeration type USBIO_PIPE_TYPE (page 214).

MaximumTransferSize

Specifies the maximum size, in bytes, of data transfers the USBD supports on this pipe. This is the maximum size of buffers that can be used with read or write operations on this pipe.

MaximumPacketSize

Specifies the maximum packet size of USB data transfers the endpoint is capable of sending or receiving as reported by the device in the corresponding endpoint descriptor. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

EndpointAddress

Specifies the address of the endpoint on the USB device as reported in the corresponding endpoint descriptor. The endpoint address includes the direction flag at bit position 7 (MSB)

- Bit 7 = 0: OUT endpoint
- Bit 7 = 1: IN endpoint

Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.
Interval
Specifies the interval, in milliseconds, for polling the endpoint for data as reported in the corresponding endpoint descriptor. The value is meaningful for interrupt endpoints only. Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

InterfaceNumber
Specifies the index of the interface the pipe belongs to. The value corresponds to the field InterfaceNumber of an USBIO_INTERFACE_CONFIGURATION_INFO (page 200) structure.

reserved1
Reserved field set to zero.

reserved2
Reserved field, set to zero.

reserved3
Reserved field, set to zero.

Comments
This structure is an output of IOCTL_USBIO_GET_CONFIGURATION_INFO (page 166) operations. Only active pipes from the current configuration are returned.
**USBIO_CONFIGURATION_INFO**

The USBIO_CONFIGURATION_INFO structure provides information about all interfaces and all pipes available in the current configuration.

**Definition**

typedef struct _USBIO_CONFIGURATION_INFO{
    ULONG NbOfInterfaces;
    ULONG NbOfPipes;
    USBIO_INTERFACE_CONFIGURATION_INFO InterfaceInfo[USBIO_MAX_INTERFACES];
    USBIO_PIPE_CONFIGURATION_INFO PipeInfo[USBIO_MAX_PIPES];
} USBIO_CONFIGURATION_INFO;

**Members**

- **NbOfInterface**
  Contains the number of interfaces. This is the number of valid entries in the InterfaceInfo structure.

- **NbOfPipes**
  Contains the number of pipes. This is the number of valid entries in the PipeInfo structure.

- **InterfaceInfo[USBIO_MAX_INTERFACES]**
  An array of USBIO_INTERFACE_CONFIGURATION_INFO (page 200) structures that describes the interfaces. There are NbOfInterfaces valid entries in this array.

- **PipeInfo[USBIO_MAX_PIPES]**
  An array of USBIO_PIPE_CONFIGURATION_INFO (page 202) structures that describes the pipes. There are NbOfPipes valid entries in this array.

**Comments**

This structure is an output of **IOCTL_USBIO_GET_CONFIGURATION_INFO** (page 166) operations. Only active pipes from the current configuration are returned.
**USBIO_FRAME_NUMBER**

The USBIO_FRAME_NUMBER structure contains information about the USB frame counter value.

*Definition*

```c
typedef struct _USBIO_FRAME_NUMBER{
    ULONG FrameNumber;
} USBIO_FRAME_NUMBER;
```

*Members*

**FrameNumber**

Contains the current value of the frame counter maintained by the USBD.

*Comments*

This structure is an output of **IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER** (page 168) requests.
USBIO_DEVICE_POWER

The USBIO_DEVICE_POWER structure contains information about the USB device power states.

**Definition**

typedef struct _USBIO_DEVICE_POWER{
    USBIO_DEVICE_POWER_STATE DevicePowerState;
} USBIO_DEVICE_POWER;

**Member DevicePowerState**

Contains the power state of the USB device. The values are defined by the USBIO_DEVICE_POWER_STATE (page 217) enumeration type.

**Comments**

This structure is used with IOCTL_USBIO_GET_DEVICE_POWER_STATE (page 170) and IOCTL_USBIO_SETDEVICE_POWER_STATE (page 169) requests.
**USBIO_BINDPIPE**

The USBIO_BINDPIPE structure provides information about the pipe to bind to.

**Definition**

typedef struct _USBIO_BIND_PIPE{
    UCHAR EndpointAddress;
} USBIO_BIND_PIPE;

**Member**

**EndpointAddress**

Specifies the address of the endpoint on the USB device that shall be associated with the pipe. The endpoint address is specified as reported in the corresponding endpoint descriptor.

The endpoint address includes the direction flag at bit position 7 (MSB).

- Bit 7 = 1: OUT endpoint
- Bit 7 = 0: IN endpoint

Refer to the Universal Serial Bus Specification 1.1, Chapter 9 for more information.

**Comments**

This structure has to be used as an input for IOCTL_USBIO_BIND_PIPE (page 174) operations. Only active endpoints from the current configuration can be bound.
USBIO_PIPE_PARAMETERS

The USBIO_PIPE_PARAMETERS structure contains pipe specific parameter settings of the USBIO driver.

**Definition**

```c
typedef struct _USBIO_PIPE_PARAMETERS{
    ULONG Flags;
} USBIO_PIPE_PARAMETERS;
```

**Member Flags**

This field contains zero or the following value.

**USBIO_SHORT_TRANSFER_OK**

If this flag is set, the USBIO driver does not return an error during read operations from a Bulk or Interrupt pipe if a packet received from the device is shorter than the maximum packet size of the endpoint. Otherwise, a short packet causes an error condition. This option is meaningful for IN pipes only.

**Comments**

This structure is intended to be used with

- `IOCTL_USBIO_GET_PIPE_PARAMETERS` (page 178) and
- `IOCTL_USBIO_SET_PIPE_PARAMETERS` (page 179) operations.

The default setting of this parameter can be changed by means of the registry parameter `ShortTransferOk`. This parameter has an effect only for read operations from Bulk or Interrupt pipes. For Isochronous pipes the flags in the appropriate ISO data structures are used (see `USBO_ISO_TRANSFER` (page 210)).
The USBIO_PIPE_CONTROL_TRANSFER structure provides information used to generate a specific control request.

**Definition**
```c
typedef struct _USBIO_PIPE_CONTROL_TRANSFER{
    DWORD Flags;
    UCHAR SetupPacket[8];
} USBIO_PIPE_CONTROL_TRANSFER;
```

**Members**
- **Flags**
  - This field contains zero or the following value:
    - USBIO_SHORT_TRANSFER_OK
      - If this flag is set, the USBIO driver does not return an error if a data packet received from the device is shorter than the maximum packet size of the endpoint. Otherwise, a short packet causes an error condition.

- **SetupPacket[8]**
  - Specifies the setup packet to be sent to the device. The format of the eight byte setup packet is defined by the Universal Serial Bus Specification 1.1, Chapter 9.

**Comments**
- This structure has to be used as an input for IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN (page 180) and IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT (page 181) operations.
**USBIO_ISO_TRANSFER**

The USBIO_ISO_TRANSFER structure provides information used for isochronous data transfers.

**Definition**

typedef struct _USBIO_ISO_TRANSFER{
    ULONG NumberOfPackets;
    ULONG Flags;
    ULONG StartFrame;
    ULONG ErrorCount;
} USBIO_ISO_TRANSFER;

**Members**

- **NumberOfPackets**
  Specifies the number of packets to be sent to or received from the device. Each packet corresponds to an USB frame. The maximum number of packets in a read or write operation is limited by the registry parameter `MaxIsoPackets`.

- **Flags**
  This field contains zero or any combination (bit-wise or) of the following values.

  **USBIO_SHORT_TRANSFER_OK**
  If this flag is set, the USBIO driver does not return an error if a data packet received from the device is shorter than the maximum packet size of the endpoint. Otherwise, a short packet causes an error condition.

  **USBIO_START_TRANSFER_ASAP**
  If this flag is set, the transfer will be started as soon as possible and the `StartFrame` parameter is ignored. This flag has to be used if a continuous data stream shall be sent to the isochronous endpoint of the USB device.

- **StartFrame**
  Specifies the frame number the transfer shall start with. The value has to be within a system-defined range relative to the current frame. The range is normally set to 1024 frames.

  If `USBIO_START_TRANSFER_ASAP` is specified in `Flags`, this member has not to be set by the caller. It contains the frame number that the transfer started with, when the request is returned by the USBIO.
If \texttt{USBIO\_START\_TRANSFER\_ASAP} is not specified in \texttt{Flags}, this member has to be set by the caller to the frame number this transfer shall start with. An error occurs if the frame number is outside of the valid range.

\textbf{ErrorCount}

Contains the total number of errors occurred during this transaction when the request is returned by the USBIO.

\textit{Comments}

This structure is the fixed size part of the \texttt{USBIO\_ISO\_TRANSFER\_HEADER} (page 213) that has to be used as an input for \texttt{ReadFile} and \texttt{WriteFile} operations with an isochronous pipe. the transfer buffer has to contain an \texttt{USBIO\_ISO\_TRANSFER\_HEADER} (page 213) structure at offset zero.
USBIO_ISO_PACKET

The USBIO_ISO_PACKET structure defines the size and location of a single isochronous data packet within the transfer buffer that is used for isochronous data transfers.

**Definition**

typedef struct _USBIO_ISO_PACKET{
    ULONG Offset;
    ULONG Length;
    ULONG Status;
} USBIO_ISO_PACKET;

**Members**

- **Offset**
  Specifies the offset, in bytes, of the packet relative to the start of the data buffer. This parameter has to be specified by the caller for read and write operations.

- **Length**
  Specifies the size, in bytes, of the packet. This parameter has to be set by the caller for write operations. On read operations this field is set by the USBIO when the request is returned.

- **Status**
  Contains the final status code for the transfer of this packet when the request is returned by the USBIO.

**Comments**

A variable size array of USBIO_ISO_PACKET structures is part of the USBIO_ISO_TRANSFER_HEADER (page 213) that has to be used as an input for ReadFile and WriteFile operations with an isochronous pipe. An USBIO_ISO_PACKET structure is required for each data packet to be transferred. The maximum number of data packets is limited by the registry parameter MaxIsoPackets.
The USBIO_ISO_TRANSFER_HEADER structure defines the header that has to be contained in the data buffers that are used for isochronous transfers.

**Definition**

typedef struct _USBIO_ISO_TRANSFER_HEADER{
    USBIO_ISO_TRANSFER IsoTransfer;
    USBIO_ISO_PACKET IsoPacket[1];
} USBIO_ISO_TRANSFER_HEADER;

**Members**

*IsoTransfer*

This is the fixed size part of the header. See the description of the USBIO_ISO_TRANSFER (page 210) structure for more information.

*IsoPacket[1]*

This is a variable length array of USBIO_ISO_PACKET (page 212) structures. Each member defines an isochronous packet to be transferred. The number of valid entries in this array is defined by the NumberOfPackets field of IsoTransfer. The maximum number of data packets is limited by the registry parameter MaxIsoPackets.

**Comments**

The data buffer passed to ReadFile or WriteFile operations with an isochronous pipe has to contain a valid USBIO_ISO_TRANSFER_HEADER (page 213) structure at offset zero. After this header the buffer contains the isochronous data which is divided into packets. The IsoPacket array describes the location and the size of the data packets. Each data packet is transferred in a separate USB frame.

There must not be gaps between the data packets in the transfer buffer.
E.5.5 Enumeration Types

**USBIO_PIPE_TYPE**

The USBIO_PIPE_TYPE enumeration type contains values that identify the type of an USB pipe or an USB endpoint respectively.

**Definition**

```c
typedef enum __USBIO_PIPE_TYPE {
    PipeTypeControl = 0,
    PipeTypeIsochronous,
    PipeTypeBulk,
    PipeTypeInterrupt
} USBIO_PIPE_TYPE;
```

**Comments**

The meaning of the values is defined by the Universal Serial Bus Specification 1.1, Chapter 9.
**USBIO_REQUEST_RECIPIENT**

The USBIO_REQUEST_RECIPIENT enumeration type contains values that identify the recipient of an USB device request.

**Definition**

typedef enum __USBIO_REQUEST_RECIPIENT {
    RecipientDevice = 0,
    RecipientInterface,
    RecipientEndpoint,
    RecipientOther
} USBIO_REQUEST_RECIPIENT;

**Comments**
The meaning of the values is defined by the Universal Serial Bus Specification 1.1, Chapter 9.
USBIO_REQUESTTYPE

The USBIO_REQUESTTYPE enumeration type contains values that identify the type of an USB device request.

**Definition**

define enum _USBIO_REQUESTTYPE{
    RequestTypeClass = 1,
    RequestTypeVendor
}

**Comments**
The meaning of the values is defined by the Universal Serial Bus Specification 1.1, Chapter 9.

The enumeration does not contain the Standard request type defined by the USB Specification. This is because only Class and Vendor requests are supported by the USBD interface. Standard requests are generated internally by the USBD.
USBIO_DEVICE_POWER_STATE

The USBIO_DEVICE_POWER_STATE enumeration type contains values that identify the power state of a device.

Definition
typedef enum _USBIO_DEVICE_POWER_STATE{
    DevicePowerStateD0 = 0,
    DevicePowerStateD1,
    DevicePowerStateD2,
    DevicePowerStateD3
} USBIO_DEVICE_POWER_STATE;

Entries

DevicePowerStateD0
    Device fully on, normal operation
DevicePowerStateD1
    Suspend
DevicePowerStateD2
    Suspend
DevicePowerStateD3
    Device off

Comments
The meaning of the values is defined by the Power Management specification.
### E.5.6 Error Codes

#### Table E-2. Error Codes Defined by the USBIO Device Driver

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBIO_ERR_SUCCESS</td>
<td>(0x00000000L)</td>
</tr>
<tr>
<td>USBIO_ERR_CRC</td>
<td>(0xE0000001L)</td>
</tr>
<tr>
<td>USBIO_ERR_BTSTUFF</td>
<td>(0xE0000002L)</td>
</tr>
<tr>
<td>USBIO_ERR_DATA_TOGGLE_MISMATCH</td>
<td>(0xE0000003L)</td>
</tr>
<tr>
<td>USBIO_ERRSTALL_PID</td>
<td>(0xE0000004L)</td>
</tr>
<tr>
<td>USBIO_ERR_DEV_NOT_RESPONDING</td>
<td>(0xE0000005L)</td>
</tr>
<tr>
<td>USBIO_ERR_PID_CHECK_FAILURE</td>
<td>(0xE0000006L)</td>
</tr>
<tr>
<td>USBIO_ERR_UNEXPECTED_PID</td>
<td>(0xE0000007L)</td>
</tr>
<tr>
<td>USBIO_ERR_DATA_OVERRUN</td>
<td>(0xE0000008L)</td>
</tr>
<tr>
<td>USBIO_ERR_DATA_UNDERRUN</td>
<td>(0xE0000009L)</td>
</tr>
<tr>
<td>USBIO_ERR_RESERVED1</td>
<td>(0xE0000010L)</td>
</tr>
<tr>
<td>USBIO_ERR_RESERVED2</td>
<td>(0xE0000011L)</td>
</tr>
<tr>
<td>USBIO_ERR_BUFFER_OVERRUN</td>
<td>(0xE0000012L)</td>
</tr>
<tr>
<td>USBIO_ERR_BUFFER_UNDERRUN</td>
<td>(0xE0000013L)</td>
</tr>
<tr>
<td>USBIO_ERR_NOT_ACCESSSED</td>
<td>(0xE0000014L)</td>
</tr>
<tr>
<td>USBIO_ERR_FIFO</td>
<td>(0xE0000015L)</td>
</tr>
<tr>
<td>USBIO_ERR_ENDPOINT_HALTED</td>
<td>(0xE0000016L)</td>
</tr>
<tr>
<td>USBIO_ERR_NO_MEMORY</td>
<td>(0xE0000017L)</td>
</tr>
<tr>
<td>USBIO_ERR_INVALID_URB_FUNCTION</td>
<td>(0xE0000018L)</td>
</tr>
<tr>
<td>USBIO_ERR_INVALID_PARAMETER</td>
<td>(0xE0000019L)</td>
</tr>
<tr>
<td>USBIO_ERR_ERROR_BUSY</td>
<td>(0xE000001AL)</td>
</tr>
<tr>
<td>USBIO_ERR_REQUEST_FAILED</td>
<td>(0xE000001BL)</td>
</tr>
<tr>
<td>USBIO_ERR_INVALID_PIPE_HANDLE</td>
<td>(0xE000001CL)</td>
</tr>
<tr>
<td>USBIO_ERR_NO_BANDWIDTH</td>
<td>(0xE000001DL)</td>
</tr>
<tr>
<td>USBIO_ERR_INTERNAL_HC_ERROR</td>
<td>(0xE000001EL)</td>
</tr>
<tr>
<td>USBIO_ERR_ERROR_SHORT_TRANSFER</td>
<td>(0xE000001FL)</td>
</tr>
<tr>
<td>USBIO_ERR_BAD_START_FRAME</td>
<td>(0xE0000020L)</td>
</tr>
<tr>
<td>USBIO_ERR_ISOCH_REQUEST_FAILED</td>
<td>(0xE0000021L)</td>
</tr>
<tr>
<td>USBIO_ERR_FRAME_CONTROL_OWNED</td>
<td>(0xE0000022L)</td>
</tr>
<tr>
<td>USBIO_ERR_FRAME_CONTROL_NOT_OWNED</td>
<td>(0xE0000023L)</td>
</tr>
<tr>
<td>USBIO_ERR_CANCELED</td>
<td>(0xE0000024L)</td>
</tr>
<tr>
<td>USBIO_ERR_CANCELING</td>
<td>(0xE0000025L)</td>
</tr>
<tr>
<td>USBIO_ERR_FAILED</td>
<td>(0xE0000026L)</td>
</tr>
<tr>
<td>USBIO_ERR_INVALID_INBUFFER</td>
<td>(0xE0000027L)</td>
</tr>
<tr>
<td>USBIO_ERR_INVALID_OUTBUFFER</td>
<td>(0xE0000028L)</td>
</tr>
<tr>
<td>USBIO_ERR_OUT_OF_MEMORY</td>
<td>(0xE0000029L)</td>
</tr>
<tr>
<td>USBIO_ERR_PENDING_REQUESTS</td>
<td>(0xE000002AL)</td>
</tr>
<tr>
<td>USBIO_ERR_ALREADY_CONFIGURED</td>
<td>(0xE000002BL)</td>
</tr>
<tr>
<td>USBIO_ERR_NOT_CONFIGURED</td>
<td>(0xE000002CL)</td>
</tr>
<tr>
<td>USBIO_ERR_OPEN_PIPES</td>
<td>(0xE000002DL)</td>
</tr>
<tr>
<td>USBIO_ERR_ALREADY_BOUND</td>
<td>(0xE000002EL)</td>
</tr>
</tbody>
</table>
Table E-2. Error Codes Defined by the USBIO Device Driver (Continued)

<table>
<thead>
<tr>
<th>Code Description</th>
<th>Error Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>USBIO_ERR_NOT_BOUND</td>
<td>(0xE0001009L)</td>
</tr>
<tr>
<td>USBIO_ERR_DEVICE_NOT_PRESENT</td>
<td>(0xE000100AL)</td>
</tr>
<tr>
<td>USBIO_ERR_CONTROL_NOT_SUPPORTED</td>
<td>(0xE000100BL)</td>
</tr>
<tr>
<td>USBIO_ERR_TIMEOUT</td>
<td>(0xE000100CL)</td>
</tr>
<tr>
<td>USBIO_ERR_INVALID_RECIPIENT</td>
<td>(0xE000100DL)</td>
</tr>
<tr>
<td>USBIO_ERR_INVALID_TYPE</td>
<td>(0xE000100EL)</td>
</tr>
<tr>
<td>USBIO_ERR_INVALID_IOCTL</td>
<td>(0xE000100FL)</td>
</tr>
<tr>
<td>USBIO_ERR_INVALID_DIRECTION</td>
<td>(0xE0001010L)</td>
</tr>
<tr>
<td>USBIO_ERR_TOO_MUCH_ISO_PACKETS</td>
<td>(0xE0001011L)</td>
</tr>
<tr>
<td>USBIO_ERR_POOL_EMPTY</td>
<td>(0xE0001012L)</td>
</tr>
<tr>
<td>USBIO_ERR_PIPE_NOT_FOUND</td>
<td>(0xE0001013L)</td>
</tr>
<tr>
<td>USBIO_ERR_INVALID_ISO_PACKET</td>
<td>(0xE0001014L)</td>
</tr>
<tr>
<td>USBIO_ERR_OUT_OF_ADDRESS_SPACE</td>
<td>(0xE0001015L)</td>
</tr>
<tr>
<td>USBIO_ERR_INTERFACE_NOT_FOUND</td>
<td>(0xE0001016L)</td>
</tr>
<tr>
<td>USBIO_ERR_INVALID_DEVICE_STATE</td>
<td>(0xE0001017L)</td>
</tr>
<tr>
<td>USBIO_ERR_INVALID_PARAM</td>
<td>(0xE0001018L)</td>
</tr>
<tr>
<td>USBIO_ERR_DEMO_EXPIRED</td>
<td>(0xE0001019L)</td>
</tr>
<tr>
<td>USBIO_ERR_INVALID_POWER_STATE</td>
<td>(0xE000101AL)</td>
</tr>
<tr>
<td>USBIO_ERR_POWER_DOWN</td>
<td>(0xE000101BL)</td>
</tr>
<tr>
<td>USBIO_ERR_VERSION_MISMATCH</td>
<td>(0xE000101CL)</td>
</tr>
<tr>
<td>USBIO_ERR_SET_CONFIGURATION_FAILED</td>
<td>(0xE000101DL)</td>
</tr>
<tr>
<td>USBIO_ERR_VID_RESTRICTION</td>
<td>(0xE0001080L)</td>
</tr>
<tr>
<td>USBIO_ERR_ISO_RESTRICTION</td>
<td>(0xE0001081L)</td>
</tr>
<tr>
<td>USBIO_ERR_BULK_RESTRICTION</td>
<td>(0xE0001082L)</td>
</tr>
<tr>
<td>USBIO_ERR_EP0_RESTRICTION</td>
<td>(0xE0001083L)</td>
</tr>
<tr>
<td>USBIO_ERR_PIPE_RESTRICTION</td>
<td>(0xE0001084L)</td>
</tr>
<tr>
<td>USBIO_ERR_PIPE_SIZE_RESTRICTION</td>
<td>(0xE0001085L)</td>
</tr>
<tr>
<td>USBIO_ERR_Device_Size_RESTRICTION</td>
<td>(0xE0001100L)</td>
</tr>
<tr>
<td>USBIO_ERR_Device_NOT_OPEN</td>
<td>(0xE0001102L)</td>
</tr>
<tr>
<td>USBIO_ERR_NO_SUCH_DEVICE_INSTANCE</td>
<td>(0xE0001104L)</td>
</tr>
<tr>
<td>USBIO_ERR_INVALID_FUNCTION_PARAM</td>
<td>(0xE0001105L)</td>
</tr>
</tbody>
</table>
E.6 USBIO Class Library

The USBIO Class Library (USBIOLIB) contains classes which provide wrapper functions for all of the features supported by the USBIO programming interface. Using these classes in an application is more convenient than using the USBIO interface directly. The classes are designed to be capable of being extended. In order to meet the requirements of a particular application new classes may be derived from the existing ones. The class library is provided fully in source code.

Figure E-4 shows the classes included in the USBIOLIB and their relations.

![Figure E-4. USBIO Class Library](image)

E.6.1 CUsbIo Class

The class CUsbIo implements the basic interface to the USBIO device driver. It includes all functions that are related to an USBIO device object. Thus, by using an instance of the CUsbIo class all operations which do not require a pipe context can be performed.

The CUsbIo class supports device enumeration and an Open function that is used to connect an instance of the class to an USBIO device object. The handle that represents the connection is stored inside the class instance. It is used for all subsequent requests to the device.
For each device-related operation the USBIO driver supports, a member function exists in the `CUsbIo` class. The function takes the parameters that are required for the operation and returns the status that is reported by the USBIO driver.

### E.6.2 CUsbIoPipe Class

The class `CUsbIoPipe` extends the `CUsbIo` class by functions that are related to an USBIO pipe object. An instance of the `CUsbIoPipe` class is associated directly with an USBIO pipe object. In order to establish the connection to the pipe the class provides a `Bind` function. After a `CUsbIoPipe` instance is bound, pipe-related functions can be performed by using member functions of the class.

For each pipe-related operation that the USBIO driver supports a member function exists in the `CUsbIoPipe` class. The function takes the parameters that are required for the operation and returns the status that is reported by the USBIO driver.

The `CUsbIoPipe` class supports an asynchronous communication model for data transfers from or to the pipe. The `Read` or `Write` function is used to submit a data buffer to the USBIO driver. The function returns immediately indicating success if the buffer was sent to the driver successfully. There is no blocking within the `Read` or `Write` function. Therefore, it is possible to send multiple buffers to the pipe. The buffers are processed sequentially in the same order as they were submitted. The `WaitForCompletion` member function is used to wait until the data transfer from or to a particular buffer is finished. This function blocks the calling thread until the USBIO driver has completed the I/O operation with the buffer.

In order to use a data buffer with the `Read`, `Write`, and `WaitForCompletion` functions of the `CUsbIoPipe` class the buffer has to be described by a `CUsbIoBuf` object. The `CUsbIoBuf` helper class stores context information while the read or write operation is pending.
E.6.3 CUsbIoThread Class

The class CUsbIoThread provides basic functions needed to implement a worker thread that performs input or output operations on a pipe. It includes functions that are used to start and stop the worker thread.

The CUsbIoThread class does not implement the thread’s main routine. This has to be done in a derived class. Thus, CUsbIoThread is an universal base class that simplifies the implementation of a worker thread that performs I/O operations on a pipe.

NOTE: The worker thread created by CUsbIoThread is a native system thread. That means it cannot be used to call MFC (Microsoft Foundation Classes) functions. It is necessary to use PostMessage, SendMessage or some other communication mechanism to switch over to MFC-aware threads.

E.6.4 CUsbIoReader Class

The class CUsbIoReader extends the CUsbIoThread class by a specific worker thread routine that continuously sends Read requests to the pipe. The thread’s main routine gets buffers from an internal buffer pool and submits them to the pipe using the Read function of the CUsbIoPipe class. After all buffers are submitted the routine waits for the first pending buffer to complete. If a buffer is completed by the USBIO driver the virtual member function ProcessData is called with this buffer. Within this function the data received from the pipe should be processed. The ProcessData function has to be implemented by a class that is derived from CUsbIoReader. After that, the buffer is put back to the pool and the main loop is started from the beginning.

E.6.5 CUsbIoWriter Class

The class CUsbIoWriter extends the CUsbIoThread class by a specific worker thread routine that continuously sends Write requests to the pipe. The thread’s main routine gets a buffer from an internal buffer pool and calls the virtual member function ProcessBuffer to fill the
buffer with data. After that, the buffer is sent to the pipe using the `Write` function of the `CUsbIoPipe` class. After all buffers are submitted the routine waits for the first pending buffer to complete. If a buffer is completed by the USBIO driver the buffer is put back to the pool and the main loop is started from the beginning.

### E.6.6 CUsbIoBufClass

The helper class `CUsbIoBuf` is used as a descriptor for buffers that are processed by the class `CUsbIoPipe` and derived classes. One instance of the `CUsbIoBuf` class has to be created for each buffer. The `CUsbIoBuf` object stores context and status information that is needed to process the buffer asynchronously.

The `CUsbIoBuf` class contains a link element (Next pointer). This may be used to build a chain of linked buffer objects to hold them in a list. This way, the management of buffers can be simplified.

### E.6.7 CUsbIoBufPool Class

The class `CUsbIoBufPool` is used to manage a pool of free buffers. It provides functions used to allocate an initial number of buffers, to get a buffer from the pool, and to put a buffer back to the pool.

### E.7 USBIO Demo Application

The USBIO Demo Application demonstrates the usage of the USBIO driver interface. It is based on the USBIO Class Library which covers the native API calls. The Application is designed to handle one USB driver that can contain multiple pipes. It is possible to run multiple instances of the application, each connected to another USB device.

The USBIO Demo Application is a dialog based MFC (Microsoft Foundation Classes) application. The main dialog contains a button that allows to open an output window. All output data and all error messages are directed to this window. The button “Clear Output Window” discards the actual contents of the window.
The main dialog contains several dialog pages which allow to access the device-related driver operations. From the dialog page “Pipes” a separate dialog can be started for each configured pipe. The pipe dialogs are non-modal. More than one pipe dialog can be opened at a given point in time.

E.7.1 Dialog Pages for Device Operations

E.7.1.1 Device

This page allows to scan for available devices. The application enumerates the USBIO device objects currently available. It opens each device object and queries the USB device descriptor. The USB devices currently attached to USBIO are listed in the output window. A device can be opened and closed, and the device parameters can be requested or set.

Related driver interfaces:

- `CreateFile();`
- `CloseHandle();`
- `IOCTL_USBIO_GET_DEVICE_PARAMETERS (page 164)`
- `IOCTL_USBIO_SET_DEVICE_PARAMETERS (page 165)`

E.7.1.2 Descriptors

This page allows to query standard descriptors from the device. The index of the configuration and the strong descriptors can be specified. The descriptors are dumped to the output window. Some descriptors are interpreted. Unknown descriptors are presented as HEX dump.

Related driver interfaces:

- `IOCTL_USBIO_GET_DESCRIPTOR (page 151)`
E.7.1.3 Configuration

This page is used to set a configuration, to unconfigure the device, or to request the current configuration.

Related driver interfaces:

- IOCTL_USBIO_GET_DESCRIPTOR (page 151)
- IOCTL_USBIO_GET_CONFIGURATION (page 156)
- IOCTL_USBIO_STORE_CONFIG_DESCRIPTOR (page 158)
- IOCTL_USBIO_SET_CONFIGURATION (page 159)
- IOCTL_USBIO_UNCONFIGURE_DEVICE (page 160)

E.7.1.4 Interface

By using this page the alternate setting of a configured interface can be changed.

Related driver interfaces:

- IOCTL_USBIO_SET_INTERFACE (page 161)
- IOCTL_USBIO_GET_INTERFACE (page 157)

E.7.1.5 Pipes

This page allows to show all configured endpoints and interfaces by using the button “Get Configuration Info”. A new non-modal dialog for each configured pipe can be opened as well.

Related driver interfaces:

- IOCTL_USBIO_GET_CONFIGURATION_INFO (page 166)
- IOCTL_USBIO_BIND_PIPE (page 174)
- IOCTL_USBIO_UNBIND_PIPE (page 175)
E.7.1.6  Class or Vendor Request

By using this page a class or vendor specific request can be send to the USB device.

Related driver interfaces:

- IOCTL_USBIO_CLASS_OR_VENDOR_IN_REQUEST (page 162)
- IOCTL_USBIO_CLASS_OR_VENDOR_OUT_REQUEST (page 163)

E.7.1.7  Feature

This page can be used to send set or clear feature requests.

Related driver interfaces:

- IOCTL_USBIO_SET_FEATURE (page 153)
- IOCTL_USBIO_CLEAR_FEATURE (page 154)

E.7.1.8  Other

This page allows to query the device state, to reset the USB device, to get the current frame number, and to query or set the device power state.

Related driver interfaces:

- IOCTL_USBIO_GET_STATUS (page 155)
- IOCTL_USBIO_RESET_DEVICE (page 167)
- IOCTL_USBIO_GET_CURRENT_FRAME_NUMBER (page 168)
- IOCTL_USBIO_SET_DEVICE_POWER_STATE (page 169)
- IOCTL_USBIO_GET_DEVICE_POWER_STATE (page 170)
E.7.1.9 Dialog Pages for Pipe Operations

Three different types of pipe dialogs can be selected. For IN pipes a Read from pipe to file dialog and a Read from pipe to output window dialog can be activated. For OUT pipes a Write from file to pipe dialog can be started. The pipe dialog Read from pipe to output window cannot be used with isochronous pipes.

When a new pipe dialog is opened it is bound to a pipe. If the dialog is closed the pipe is unbound. Each pipe dialog contains pipe-related and transfer-related functions. The first three dialog pages are the same in all pipe dialogs. The last page has a special meaning.

E.7.1.10 Pipe

By using this page it is possible to access functions Reset Pipe, Abort Pipe, Get Pipe Parameters, and Set Pipe Parameters.

Related driver interfaces:

- IOCTL_USBIO_RESETPIPE (page 176)
- IOCTL_USBIO_ABORT_PIPE (page 177)
- IOCTL_USBIO_GET_PIPE_PARAMETERS (page 178)
- IOCTL_USBIO_SET_PIPE_PARAMETERS (page 179)

E.7.1.11 Buffers

By means of this page the size and the number of buffers can be selected. For Interrupt and Bulk pipes the “Size of Buffer” field is relevant. For Isochronous pipes the “Number of Packets” field is relevant and the required buffer size is calculated internally. In the “Max Error Count” field a maximum number of errors can be specified. When this number is exceeded, the data transfer is aborted. Each successful transfer resets the error counter to zero.
E.7.1.12  Control

This dialog page allows to access user-defined control pipes. It cannot be used to access the default pipe (endpoint zero) of an USB device.

Related driver interfaces:

- IOCTL_USBIO_PIPE_CONTROL_TRANSFER_IN (page 180)
- IOCTL_USBIO_PIPE_CONTROL_TRANSFER_OUT (page 181)

E.7.1.13  Read from Pipe to Output Window

This dialog page allows to read data from an Interrupt or Bulk pipe and to dump it to the output window. For large amounts of data the transfer may be slowed down because of the overhead involved with printing to the output window. The printing of the data can be enabled/disabled by the switch Print to Output Window.

Related driver interfaces:

- ReadFile();
- IOCTL_USBIO_ABORT_PIPE (page 177)

E.7.1.14  Read from Pipe to File

This dialog page allows to read data from the pipe to a file. This transfer type can be used for Isochronous pipes as well. The synchronization type of the Isochronous pipe has to be “asynchronous”. The application does not support data rate feedback.

Related driver interfaces:

- ReadFile();
- IOCTL_USBIO_ABORT_PIPE (page 177)
E.7.1.15 Write from File to Pipe

This dialog page allows to write data from a file to the pipe. This transfer type can be used for Isochronous pipes as well. The synchronization type of the isochronous pipe has to be “asynchronous”. The application does not support data rate feedback.

Related driver interfaces:

- `WriteFile();`
- `IOCTL_USBIO_ABORT_PIPE` (page 177)

E.8 Installation Issues

This section discusses the topics related to installation of the USBIO device driver. Included is a description of how a customized driver setup can be built.

**IMPORTANT:** On Windows 2000 administrator rights are required to install a device driver. Because the USBIO driver is installed in the same way as any other Plug&Play device driver the installation requires administrator rights. Once the USBIO driver is installed standard user rights are sufficient to load the driver and to use the driver by accessing its programming interface.

E.8.1 Automated Installation: The USBIO Installation Wizard

Using the USBIO Installation Wizard is the quickest and easiest way for installing the USBIO device driver. This wizard performs the driver installation automatically in a step-by-step procedure. The device the USBIO driver will be installed for can be selected from a list. It is not necessary to manually edit or copy any files. After installation is complete the wizard allows to save the specific setup files that has been generated for the selected device. These files can be used at a later time to manually install the USBIO driver for the same device, without using the Installation Wizard.
The steps required to install the USBIO driver by using the Installation Wizard are described below.

- **On Windows 2000** make sure you are logged on as an administrator or have enough privileges to install device drivers on the system. In general, special privileges are required to install device drivers on Windows 2000.

- **Connect your USB device** to the system. After plugging in the device Windows launches the New Hardware Wizard and prompts you for a device driver. Complete the New Hardware Wizard by clicking Next on each page and Finish on the last page. Windows either installs a system-provided driver or registers the device as “Unknown”.

  **Do not** abort the New Hardware Wizard by clicking the Cancel button. This will prevent Windows from enumerating the device and storing enumeration information in the registry. As a result of this, the device is not visible in the system and USBIO Installation Wizard is not able to install the driver for it.

For some kinds of devices the system does not launch the New Hardware Wizard. A system-provided device driver will be installed silently. This will happen if the device belongs to a predefined device class, Human Interface Devices (HID), Audio Devices, or Printer Devices for example. The USBIO Installation Wizard is able to install the USBIO driver for such devices but this will disable any system-provided driver.

- **Start the USBIO Installation Wizard** by selecting the appropriate shortcut from the Start menu. It is also possible to start the wizard directly by executing USBIOwiz.exe.

- The first page shows some hints concerning the installation process. Click the Next button to continue. Note that you can abort the Installation Wizard at any time by clicking the Cancel button.

- On the next page the wizard shows a list containing all USB devices currently connected to the system. Select the device the USBIO driver shall be installed for. The Hardware ID will be shown for the selected device. A Hardware ID is a string that is used internally by the operating system to unambiguously identify the
device. It is built from a bus identifier (USB), the 16-bit vendor ID (VID), the 16-bit product ID (PID), and optionally the revision code (REV). The IDs and the revision code are reported by the device in the USB Device Descriptor.

If your device is not shown in the list make sure it is plugged in properly and you have finished the New Hardware Wizard as described above. You may use the Device Manager to check if the device was enumerated by the system. The Device Manager can be accessed by right-clicking the “My Computer” icon on the desktop and then choosing Properties.

Use the Refresh button to rescan for active devices and to rebuild the list.

To continue, click the Next button.

- The next page shows detailed information about the selected USB device. If a driver is already installed for the device information about the driver is also shown. Verify that you selected the correct device. If not, use the Back button to return to the device list and select another device.

To install the USBIO driver for the selected device, click the Next button.

**WARNING:** If you install the USBIO driver for a device that is currently controlled by another device driver the existing driver will be disabled. This will happen immediately. As a result, the device may no longer be used by the operating system and by applications. If the device belongs to the HID class, a mouse or a keyboard for example, this can cause problems.

- On the last page the Installation Wizard shows the completion status of driver installation. If the installation was successful the USBIO driver is running. It has been dynamically loaded by the operating system.

The USBIO Installation Wizard allows you to save the specific driver installation file (INF) that it generated for the device. The INF file is specific for the selected device because it contains the Hardware ID of that device. You can use the button labeled “Save INF file” to save the generated INF file with a name of your choice.
and in a location of your choice. The Installation Wizard copies also the USBIO driver binary file usbio.sys to the same location as the INF file. You can use these files at a later time to install the USBIO driver manually.

You can use the button labeled “Run USBIO Application” to start the demo application that is included in the USBIO package. The application allows you to test several USB operations manually. Please refer to E.7 USBIO Demo Application for further information.

To quit the USBIO Installation Wizard, click Finish.

E.8.2 Manual Installation: The USBIO Setup Information File

A Setup Information File (INF) is required for proper installation of the USBIO device driver. This file describes the driver to be installed and defines the operations to be performed during the installation process.

An INF file is in ASCII text format. It can be viewed and modified with any text editor, Notepad for example. The contents and the syntax of an INF file are documented in the Microsoft Windows 2000 DDK.

The INF file is loaded and interpreted by a software component that is built into the operating system, called Device Installer. The Device Installer is closely related to the Plug&Play Manager that handles hot plugging and removal of USB devices. After the Plug&Play Manager has detected a new USB device the system searches its internal INF file data base, located in %WINDIR%\INF\, for a matching driver. If no driver can be found the New Hardware Wizard pops up and the user will be asked for a driver.

The association of device and driver is based on a string that is called Hardware ID. The Plug&Play Manager builds the Hardware ID string from the 16-bit vendor ID (VID), the 16-bit product ID (PID), and optionally the revision code (REV). The string is prefixed by the bus identifier USB. Examples for Hardware ID strings are:

USB/VID_046D&PID_0100
USB/VID_046D&PID_C001&REV_0401
USB/CLASS_09&SUBCLASS_01&PROT_00
As shown in the last example a Hardware ID can also describe a device class and subclass. This makes it possible to provide a driver that will be used whenever the system detects a device that belongs to a specific device class. An example for such a kind of driver is the system-provided HID mouse driver. This driver is installed for any type of USB mouse, regardless of the vendor, the USB Vendor ID, and the USB Product ID. The driver selection is based on the class, subclass, and protocol identifiers. Please refer to the Microsoft Windows 2000 DDK for detailed information on Hardware IDs and driver selection algorithms. Another good source of information are the INF files that ship with the operating system. They are located in a subdirectory of the Windows system directory, named “INF”. Note that on Windows 2000 this subdirectory has a Hidden attribute by default.

In order to prepare an installation disk that can be used to install the USBIO driver for your device the following steps are required.

- Copy the USBIO driver binary usbio.sys to a floppy disk or to a directory location of your choice. Copy the INF file usbio.inf provided with the USBIO package to the same location. Note that you can choose any name for the INF file, based on your company name or your product name for example. But the file name extension has to be .inf. In the following discussion it is assumed the INF file is named usbio.inf.

- Open the usbio.inf file using a text editor, Notepad for example. Edit the [Devices] section. There are various examples of Hardware ID strings prepared in this section. Select one of the examples that matches your needs. Usually, the very first example is appropriate. It associates the USBIO driver with your device by using the USB Vendor ID and Product ID. Remove the semi-colon at the start of the line and replace the VID_XXXX and PID_XXXX placeholders in the Hardware ID string by your USB Vendor ID and Product ID as shown in the examples above. Note that the IDs are given as 4-digit hexadecimal numbers.

- Edit the [Strings] section at the end of the usbio.inf file to modify the device description string for your device, defined by the value of $DeviceDesc1. The device description text will be displayed in the Device Manager next to the icon that represents your device.

- Save the INF file to accommodate your changes.
Now you are prepared to start the driver installation. The required steps are described below.

- Connect your USB device to the system. After plugging in the device Windows launches the New Hardware Wizard and prompts you for a device driver. Provide the New Hardware Wizard with the location of your installation files (usbio.inf and usbio.sys). Complete the wizard by following the instructions shown on screen. If the INF file matches with your device the driver should be installed successfully.

Note that on Windows 2000 and Windows Millennium the New Hardware Wizard shows a warning message that complains about the fact that the driver is not certified and digitally signed. You may ignore this warning and continue with driver installation. The USBIO driver is not certified because it is not an end-user product. When the USBIO driver is integrated into such a product it is possible to get a certification and a digital signature from the Windows Hardware Quality Labs (WHQL).

- If the device belongs to a predefined device class that is supported by the operating system, the system does not launch the New Hardware Wizard after the device is plugged in. Instead of that a system-provided device driver will be installed silently. Human Interface Devices (HID) like mice and keyboards, Audio Devices, or Printer Devices are examples for such devices. The operating system does not ask for a driver because it finds a matching entry for the device’s class and subclass ID in its internal INF file database, as mentioned above.

Use the Device Manager to install the USBIO driver for a device for that a driver is already running. To start the Device Manager choose Properties on the “My Computer” icon on the desktop. In the Device Manager locate your device and choose Properties on the entry. On the property page that pops up choose Driver and click the button labeled “Update Driver”. The Upgrade Device Driver Wizard is started which is similar to the New Hardware Wizard mentioned above. Provide the wizard with the location of your installation files (usbio.inf and usbio.sys) and complete the driver installation by following the instructions shown on screen.
For some device classes, especially HID devices like mice and keyboards, Windows does not allow you to install a driver with a different device class. That means you have to modify the device class entry in the [Version] section of the usbio.inf file to match with the device’s class. The device class is specified by the keywords Class and ClassGUID in the [Version] section.

For example, if you want to use a keyboard or a mouse to test the USBIO driver the new entries should be

```plaintext
Class=HIDClass
ClassGUID={745a17a0-74d3-11d0-b6fe-00a0c90f57da}
```

The ClassGUID value that is associated with a device class can be found in system-provided INF files in %WINDIR%\INF\ or in the Windows 2000 DDK documentation.

**NOTE:** At least two drivers are used for USB keyboard and mouse devices. One belongs to the USB HID class and the other one belongs to the keyboard or mouse class. The keyboard or mouse driver runs on top of the USB HID driver. The USBIO driver can replace the USB HID driver only. In the Device Manager the HID driver is shown in a section labeled “Human Interface Devices”. To be sure to replace the correct driver refer to the “Driver File Details” dialog in the Properties page of the entry. If the driver stack contains the file HIDUSB.SYS then you have selected the correct entry in the Device Manager.

In the Device manager the section “Universal Serial Bus controllers” contains an item labeled “USB Root Hub”.

**Do not install USBIO for the USB Root HUB!**

The USB Root Hub is not an USB device. It is built into the USB host controller and is controlled by a special device driver provided by the operating system.

After the driver installation was successfully completed your device should be shown in the Device Manager in the section that corresponds to the device class you specified in the usbio.inf file. You may use the Properties dialog box of that entry to verify that the USBIO driver is installed and running.

In order to verify that the USBIO driver is working properly with your device you should use the USBIO Demo Application USBIOAPP.EXE. Please refer to E.7 USBIO Demo Application for detailed information on the Demo Application.
E.8.3 Uninstalling USBIO

In order to uninstall USBIO for a given device the Device Manager has to be used. The Device Manager can be accessed by right-clicking on “My Computer” icon on the desktop and then choosing Properties. In the Device Manager double-click on the entry of the device and choose the property page that is labeled “Driver”. There are two options:

- Remove the device from the system by clicking the button “Uninstall”. The operating system will reinstall a driver the next time the device is connected or the system is rebooted.

- Install a new driver for the device by clicking the button “Update Driver”. The operating system launches the Upgrade Device Driver Wizard which searches for driver files or lets you select a driver.

In order to avoid that USBIO is reinstalled automatically and silently by the operating system it is necessary to manually remove the INF file that was used to install the USBIO driver.

During driver installation Windows stores a copy of the INF file in its internal INF file data base that is located in %WINDIR%\INF\. The original INF file is renamed and stored as oemX.inf for example, where X is a decimal number. The exact INF naming scheme depends on the operating system (Windows 2000 uses a slightly different scheme than Windows 98). The best way to find the correct INF file is to do a search for some significant string in all the INF files in the directory %WINDIR%\INF\ and its subdirectories.

Note that on Windows 98 and Windows ME the INF file may also be stored in a directory named %WINDIR%\INF\OTHER\. Another naming scheme based on the provider name is used in that case.

Note also that on Windows 2000 the %WINDIR%\INF\ directory has a Hidden attribute by default. Therefore, the directory is not shown in Windows Explorer by default.

Once you have located the INF file, delete it. This will prevent Windows from reinstalling the USBIO driver. Instead of that the New Hardware Wizard will be launched and you will be asked for a driver.
E.8.4 Building a Customized Driver Setup

When the USBIO driver is included and shipped with a retail product some setup parameters should be customized. This is necessary because the USBIO device driver might be used by several vendors and it is possible that an user has two products and that both of them use the USBIO driver. This can cause conflicts with respect to the file name of the driver executable, the location of registry parameters, the device names, and the driver interface GUIDs used. To avoid such problems a vendor who redistributes the USBIO driver for use with a hardware product should choose a new file name for the driver binary, generate a private interface GUID, and select a private location in the registry to be used to store startup parameters. In order to do that the usbio.inf file has to be customized as well.

The following list shows the steps required to build a customized USBIO setup:

- Choose a new name for the driver binary file usbio.sys. The name should not cause conflicts with drivers provided by Windows. Rename the file usbio.sys to your new name.

- Rename the Setup Information file usbio.inf. You can choose any name you want. For instance, the name may be based on your company’s or your product’s name. Note that the file extension should not be changed. It has to be “.inf”.

- Edit the [_CopyFiles_sys] section in the INF file to include the new name of the driver binary.

- Edit the value S_DriverName in the [Strings] section to match with the new name you defined for the driver binary.

- Edit the [Strings] section in the INF file to modify text strings that are shown at the user interface level. You may change the following parameters:

  - S_Provider
  - S_Mfg
  - S_DeviceClassDisplayName
  - S_DeviceDesc1
  - S_DiskName
  - S_ServiceDisplayName
• Edit the following values in the [Strings] section to specify a location in the Registry that is used to store the USBIO driver’s configuration parameters:
  
  S_ConfigPath  
  S_DeviceConfigPath1

  Note the S_ConfigPath should specify a location that is a subkey of  
  HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services.  
  The name of the subkey should be the same as the name you  
  choosed for the driver binary.

• Generate a private Globally Unique Identifier (GUID) to  
  unambiguously identify the device instances that will be created  
  by USBIO for your device. Use GUIDGEN.EXE from the Microsoft  
  Platform SDK or from the Visual C++ package for this purpose.  
  Copy the text representation of the GUID to the line in the INF file  
  that defines the registry value USBIO_UserInterfaceGuid.  
  Activate this line by removing the “;” at the beginning. Use the  
  private GUID in your application to search for available devices.  
  GUIDGEN.EXE allows you to export a  
  static const struct GUID = {...} statement that can be  
  included in the source code of an application. For an example,  
  refer to the source code of USBIOAPP or ReaderCpp.

• Edit the driver parameter settings in the sections  
  _Parameters1_98 and _Parameters1_NT. The parameters in  
  _Parameters1_98 define the default behaviour of the USBIO  
  driver on Windows 98. The parameters in _Parameters1_NT  
  define the default behaviour of the USBIO driver on  
  Windows 2000. For a detailed description of the supported  
  settings, refer to E.9 Registry Entries.

• After you finished testing your INF file remove any lines and  
  comments that are not needed. Especially, make sure that the  
  word USBIO does not occur in the files you ship with your product.  
  This is a requirement that is defined by the USBIO licensing  
  conditions. See also the License Agreement you received with the  
  USBIO package.
E.9 Registry Entries

The behaviour of the driver can be customized by startup parameters stored in the registry. The parameters are stored under a path that is specified in the INF file. This registry path is

```plaintext
\HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\USBIO\Parameters
```

by default.

The location can be customized by changing the `S_ConfigPath` and `S_DeviceConfigPath1` variables in the `[Strings]` section of the INF file.

The driver reads the parameters when a new device object is added. If a parameter does not exist when the driver attempts to read it, the driver creates the entry using an internal default value.

Table E-3 lists all registry parameters.

<table>
<thead>
<tr>
<th>Value</th>
<th>Min</th>
<th>Default</th>
<th>Max</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RequestTimeout</td>
<td>0</td>
<td>1000</td>
<td></td>
<td>Time-out interval for synchronous I/O requests, in milliseconds. Zero means infinite (no time-out).</td>
</tr>
<tr>
<td>ShortTransferOk</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>If set to 1 short packets in read transfers are allowed. If set to 0 short packets in read transfers cause errors.</td>
</tr>
<tr>
<td>UnconfigureOnClose</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>If set to 1 the device will be unconfigured when the last file handle is closed. If set to 0 the device state is not changed.</td>
</tr>
<tr>
<td>ResetDeviceOnClose</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>If set to 1 the device receives an USB reset if the last file handle is closed. If set to 0 the device state is not changed.</td>
</tr>
<tr>
<td>EnableRemoteWakeup</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>If set to 1 Remote Wakeup is enabled. If set to 0 Remote Wakeup is disabled.</td>
</tr>
<tr>
<td>MaxIsoPackets</td>
<td>16</td>
<td>64</td>
<td>512</td>
<td>Maximum number of packets allowed is an isochronous data transfer.</td>
</tr>
<tr>
<td>PowerStateOnOpen</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>Device power state that will be set when the device is opened (first handle is opened). 0...3 correspond to D0...D3</td>
</tr>
</tbody>
</table>
### Table E-3. Registry Parameters Supported by the USBIO Driver (Continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Min</th>
<th>Default</th>
<th>Max</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerStateOnClose</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>Device power state that will be set when the device is closed (last handle is closed). 0...3 correspond to D0...D3</td>
</tr>
<tr>
<td>MinPowerStateUsed</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>The minimum power state of the device while it is used (open handles exist). On system suspend the device is not allowed to go into states higher than this value. 0...3 correspond to D0...D3 The value 0 (D0) means: no suspend allowed if the device is in use. The value 3 (D3) means: full suspend (off) allowed if the device is in use.</td>
</tr>
<tr>
<td>MinPowerStateUnused</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>The minimum power state of the device while it is not used (no open handles exist). On system suspend the device is not allowed to go into states higher than this value. 0...3 correspond to D0...D3 The value 0 (D0) means: no suspend allowed if the device is not in use. The value 3 (D3) means: full suspend (off) allowed if the device is not in use.</td>
</tr>
<tr>
<td>AbortPipesOnPowerDown</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Handling of outstanding read or write requests when the device goes into a suspend state (leaves D0): 1 = abort pending requests 0 = do not abort pending requests</td>
</tr>
<tr>
<td>SuppressPnPRemoveDlg</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>If this flag is set, Windows 2000 does not show a warning dialog if the device is removed.</td>
</tr>
<tr>
<td>DebugPort</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>Destination of trace messages for debugging purposes: 0 = kernel debugger or debug monitor 1...3 = COM1...COM3 This parameter is available only if the debug (checked) build of the USBIO driver is used.</td>
</tr>
<tr>
<td>DebugMask</td>
<td>0</td>
<td>3</td>
<td></td>
<td>Control of message output for debugging. This parameter is available only if the debug (checked) build of the USBIO driver is used.</td>
</tr>
<tr>
<td>DebugBaud</td>
<td>2.400</td>
<td>57.600</td>
<td>115.200</td>
<td>Baudrate selection for debug output to COM port. This parameter is available only if the debug (checked) build of the USBIO driver is used.</td>
</tr>
</tbody>
</table>
E.10 Related Documents

- Universal Serial Bus Specification 1.0, 1.1
- USB device class specifications (Audio, HID, Printer, etc.)
- Windows 2000 DDK Documentation
- Windows 98 DDK Documentation
- Microsoft Platform SDK Documentation

E.11 Light Version Limitations

The light version for MCT Elektronikladen of the USBIO driver has the following limitations:

1. The Vendor ID of your device has to match with the Vendor ID 0x0C70 of MCT Elektronikladen. You can contact MCT Elektronikladen to receive a unique product ID.

   Possible error codes when this restriction is offended:
   USBIO_ERR_VID_RESTRICTION

2. Only one Interrupt IN endpoint and one Interrupt OUT endpoint is supported. If your device has more endpoints or other endpoint types configuring the device (SET_CONFIGURATION) will fail.

   Possible error codes when this restriction is offended:
   USBIO_ERR_PIPE_RESTRICTION
   USBIO_ERR_BULK_RESTRICTION
   USBIO_ERR_ISO_RESTRICTION

3. The maximum FIFO size of an endpoint is limited to eight bytes.

   Possible error codes when this restriction is offended:
   USBIO_ERR_PIPE_SIZE_RESTRICTION

4. The maximum size of the data stage for a Class or Vendor Request is limited to eight bytes.

   Possible error codes when this restriction is offended:
   USBIO_ERR_EP0_RESTRICTION
How to Reach Us:

USA/EUROPE/LOCATIONS NOT LISTED:
Motorola Literature Distribution
P.O. Box 5405
Denver, Colorado 80217
1-303-675-2140
1-800-441-2447

TECHNICAL INFORMATION CENTER:
1-800-521-6274

JAPAN:
Motorola Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu, Minato-ku
Tokyo 106-8573 Japan
81-3-3440-3569

ASIA/PACIFIC:
Motorola Semiconductors H.K. Ltd.
Silicon Harbour Centre
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
852-26668334

HOME PAGE:
http://www.motorola.com/semiconductors/