Embedded Power for
Business-Critical Continuity

Technical Reference Note

Rev. 0.0

APPLICATION OVERVIEW:

iMP USB-to-I^{2}C DLL User's Manual

PRODUCT:

iMP SERIES

DESCRIPTION:

ABOUT THE DLL

There are two available DLL files available for use with the USB-to-I2C adapter, each using a different function calling convention. Aside from the function calling convention, there are no other differences between the two DLL. Similar functions are available for both DLL. The two DLL files are:

iMpDll.dll – Uses C Calling; Calling function cleans the stack
ail_HID_std.dll – Uses Standard Calling; Called function cleans the stack

To be able to use applications using the DLL, the DLL should be copied in the System32 folder located at the Windows directory (Default: Windows\System32).

EMERSON
Network Power

I²C PROTOCOL

GENERAL CHARACTERISTICS

The I²C protocol allows data to be transferred between devices using two open-drain (or open-collector) bi-directional lines. One line is the serial clock (SCL) and the other is the serial data (SDA). The bus master generates the Start conditions, the clock signals on SCL, as well as the Stop condition. An acknowledge (ACK) is transmitted by the receiving device on the bus after each byte is sent.

BIT TRANSFER

Data on SDA must be stable while SCL is high. The state of SDA when SCL is high determines the logic level of the transmitted data bit.

START AND STOP CONDITIONS

Within the procedure of the I²C bus, unique situations arise which are defined as START and STOP conditions. A HIGH to LOW transition on the SDA line while SCL is HIGH is one such unique case. This situation indicates a START condition. A LOW to HIGH transition on the SDA line while SCL is HIGH defines a STOP condition. The master always generates START and STOP conditions. The bus is considered to be busy after the START condition. The bus is considered to be free again a certain time after the STOP condition.

I²C ADDRESS

The first seven bits of an I²C transmission, after a Start condition, make up the slave address. The eighth bit (or the least significant bit) is the R/W bit that determines the direction of the message.
A '0' in the least significant position of the first byte means that the master will WRITE information
to the selected slave. A '1' in this position means that the master will READ information from the slave.
When an I²C address is sent, each device in a system compares the first seven bits after the START condition with its own address. If they match, the device considers itself addressed by the master as a slave-receiver or slave-transmitter, depending on the R/W bit.

SUBADDRESS

When an I²C device contains more than one register, the various registers are generally accessed using a subaddress that is sent following the device address (see the I2CWriteArray and I2CReadArray sections below). The subaddress acts like a pointer to the register that needs to be accessed.

DATA TRANSFER

Every byte on the SDA line must be 8-bits long. The number of bytes that can be transmitted per transfer is unrestricted. Each byte must also be followed by an acknowledge bit. Data is transferred with the most significant bit first. If a receiver can't receive another complete byte of data until it has performed some other function, it can hold the clock line SCL low to force the transmitter into a wait state.

ACKNOWLEDGE

The Acknowledge related clock pulse is generated by the master. The transmitter releases the SDA line during the acknowledge clock pulse. The receiver must pull down the SDA line during the acknowledge clock pulse so that it remains stable low during the high period of the clock pulse.
The master-receiver signals the end of a read by not acknowledging the last byte it requires.

I²C BUS DOCUMENTATION

The complete I²C Bus specification can be found at http://www.semiconductors.philips.com/buses/i2c/.

SYSTEM REQUIREMENTS

- PC-compatible system
- USB port (1.1 or 2.0 compatible)
- Microsoft Windows 98(SE), ME, 2000 or XP

INSTALLATION

The USB-to-I2C adapter is an HID class device and requires no driver installation to use. The built in HID driver of the operating is used for the driver. Once the adapter is plugged on the USB port, detection can be confirmed in the Device Manager. Upon successful detection of the adapter, an HID-compliant device will be added as seen in Figure 1.

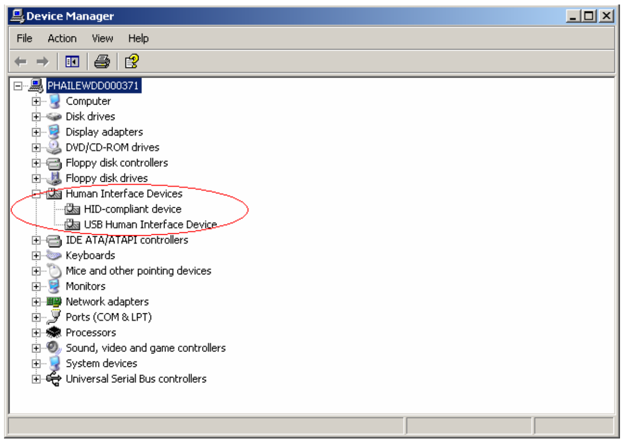

Figure 1. Device Manager when Adapter is detected

AVAILABLE FUNCTIONS

1. MPUReset

Resets the USB-to-I2C Adapter.

Prototype:
```
unsigned char MPUReset(void)
```

Input Parameters:	None
Output Parameter:	Returns 0 on a successful reset Returns 0xFF if an error occurred

2. MPUSBGetDLLVersion

Returns current DLL version.

Prototype:
```
unsigned short MPUSBGetDLLVersion (void)
```

Input Parameters:	None
Output Parameter:	Returns unsigned int interpreted as follows: MSB – Major version LSB – Minor Version

3. MPUSBGetDeviceVersion

Returns USB-to-I2C Adapter firmware version.

Prototype:
```
unsigned short MPUSBGetDeviceVersion (void)
```

Input Parameters:	None
Output Parameter:	Returns unsigned int interpreted as follows: MSB – Major version LSB – Minor Version

4. MPUSBGetI2CFrequency

Returns I2C frequency of USB-to-I2C Adapter is currently operating at.

Prototype:
```
unsigned integer MPUSBGetI2CFrequency (void)
```

Input Parameters:	None
Output Parameter:	Returns I2C frequency

5. MPUSBSetI2CFrequency

Sets current I2C frequency of USB-to-I2C Adapter to desired frequency. Allowed frequency is from 10 kHz to 400 kHz. Attempting to set I2C frequency below/above allowable frequencies will set frequency to minimum/maximum allowable frequency.

Prototype:

```
unsigned integer MPUSBSetI2CFrequency (unsigned int frequency)
```

Input Parameters:	frequency - desired I2C frequency
Output Parameter:	Returns actual I2C frequency set

6. MPUI2CWrite

Performs I2C write to a destination address. The number of bytes to be sent must be stated, as well as the pointer to the memory location of the first byte to be sent. It must also be stated whether to include a stop bit will be sent at the end of transmition or not.

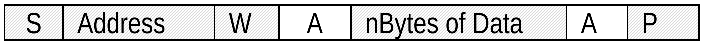

Figure 2. I2C Write

Prototype:

```
unsigned char MPUI2CWrite (unsigned char address,
                           unsigned short nBytes,
                          *unsigned char WriteData,
                           unsigned short SendStop)
```

Input Parameters:	address – Destination I2C address (7-bit)
	nBytes – Number of bytes to send (maximum of 60 bytes)
	WriteData – Memory address of location of 1st byte to be sent
	SendStop – Set to ‘1’ to transmit stop bit
	Set to ‘0’ to not transmit stop bit
Output Parameter:	Returns error code:
	0x00 – No error
	0x01 – Address NACK
	0x02 – Data NACK
	0x07 – Arbitration lost
	0x81 – Size overflow
	0xFF – USB-to-I2C not detected

7. MPUI2CRead

Performs I2C read to a destination address. The number of bytes to be received must be stated, as well as the pointer to the memory location where the first byte received will be stored. It must also be stated whether to include a stop bit will be sent at the end of transmition or not.

S	Address	R	A	nBytes of Data	A	P

Figure 3. I2C Read

Prototype:
```
unsigned char MPUI2CRead (unsigned char address,
                           unsigned short nBytes,
                          *unsigned char ReadData,
                           unsigned short SendStop)
```

Input Parameters:	address – Source I2C address (7-bit) nBytes – Number of bytes to receive (maximum of 60 bytes) Readata – Memory address of location where 1st byte to be received will be stored SendStop – Set to '1' to transmit stop bit Set to '0' to not transmit stop bit
Output Parameter:	Returns error code: 0x00 – No error 0x01 – Address NACK 0x02 – Data NACK 0x07 – Arbitration lost 0x81 – Size overflow 0xFF – USB-to-I2C not detected

8. MPUI2CWriteArray

Performs I2C write array to a destination address, is an I2C write wherein the 1st byte sent is the sub-address (ie in PMBus, the subaddress will be the PMBus command), followed a certain number of bytes, with a stop bit at the end.

S	Address	W	A	Sub-Address	A	nBytes of Data	A	P

Figure 4. I2C Write Array

Prototype:
```
unsigned char MPUI2CWriteArray (unsigned char address,
                                 unsigned char subaddress,
                                 unsigned short nBytes,
                                *unsigned char WriteData)
```

Input Parameters:	address – Destination I2C address subaddress – Destination sub-address nBytes – Number of bytes to send (maximum of 60 bytes) WriteData – Memory address of location of 1st byte to be sent
Output Parameter:	Returns error code: 0x00 – No error 0x01 – Address NACK 0x02 – Data NACK 0x07 – Arbitration lost 0x81 – Size overflow 0xFF – USB-to-I2C not detected

9. MPUI2CReadArray

Performs I2C read array to a destination address, wherein an I2C write sending one byte of sub-address (ie in PMBus, the subaddress will be the PMBus command) is performed (without stop bit), followed by an I2C read of a certain number of bytes. A stop bit is then sent at the end.

S	Address	W	A	Sub-Address	A	S	Address	A	nBytes of Data	A	Stop?

Figure 5. I2C Read Array

Prototype:

```
unsigned char MPUI2CReadArray (unsigned char address,
                               unsigned char subaddress,
                               unsigned short nBytes,
                              *unsigned char ReadData)
```

Input Parameters:	address – Source I2C address subaddress – Source sub-address nBytes – Number of bytes to receive (maximum of 60 bytes) ReadData – Memory address of location where 1st byte to be received will be stored
Output Parameter:	Returns error code: 0x00 – No error 0x01 – Address NACK 0x02 – Data NACK 0x07 – Arbitration lost 0x81 – Size overflow 0xFF – USB-to-I2C not detected