ZL_PMBus API Overview

The ZL_PMBus API enables you to write applications using the Zilker Labs PMBus Interface. The Zilker Labs PMBus Interface is a USB-to-PMBus converter available on evaluation boards such as the ZL2005EV-1 Rev. 5. A block diagram showing how data flows from your computer to a PMBus device is shown in Figure 1 below. PMBus traffic tests and GUI interfaces are some of the possible applications that can benefit from the ZL_PMBus API.

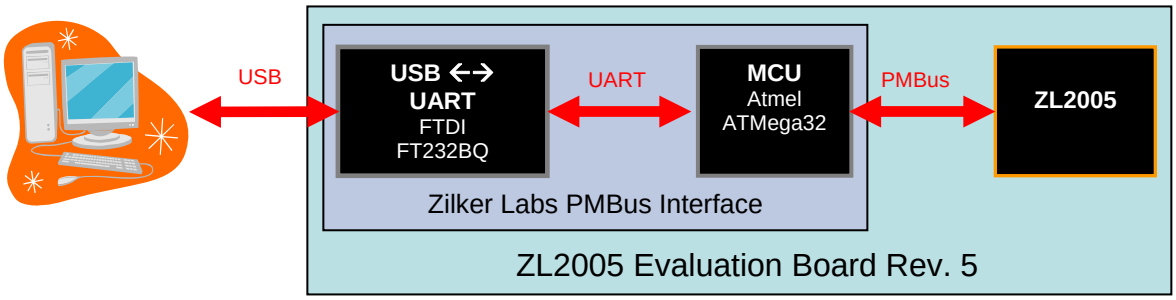

Figure 1. Data Flow Diagram of the Zilker Labs PMBus Interface

A typical application using the ZL_PMBus API is structured as follows. The top-level application will need to link ZL_PMBus.dll either internally using ZL_PMBus.lib and ZL_PMBus.h, or externally using Microsoft Dynamic-Link Library Functions. After linking, functions available in the ZL_PMBus API can be called. It should be noted that applications using the ZL_PMBus API must include the FTDI FTD2XX driver (FTD2XX.dll). This is because the Zilker Labs PMBus interface uses an FT232BQ USB-to-UART converter. We chose to do this such that the MCU responsible for performing PMBus transmissions can be re-used for standalone applications.

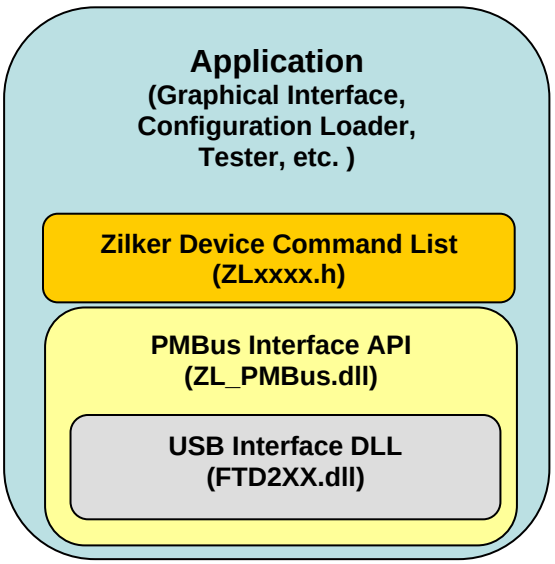

Figure 2. Hierarchy of Application and Driver Calls

Table of Contents

Function Reference

Below is an explanation of all the functions currently in the ZL_PMBus API. This includes the function parameters, return values, and usage conditions.

ZL_DLLVersion

Gets the version of the ZL_PMBUS dll you are linking to.

```
ZL_VERSION ZL_DLLVersion( void )
```

Parameters
None.

Return Values
The ZL_VERSION structure, which stores numbers for both the major and minor revision. (see "ZL_PMBus Structures, Types, and Values" on page 21 for more details)

Example

```
ZL_VERSION myVersion;
myVersion = ZL_DLLVersion();

printf("ZL_PMBus version %d.%d.\n",
       myVersion.major, myVersion.minor);
```

ZL_FWVersion

Gets the version of firmware running on the MCU.
NOTE: This command works only on firmware revisions 02 and greater.

```
ZL_STATUS ZL_FWVersion( const ZL_HANDLE deviceHandle,
                        ZL_FW_VERSION *version );
```

Parameters

deviceHandle	The handle of the device we want to retrive it's firmware version from.
*version	The firmware version, in the format of ZL_FW_VERSION, which is a structure that contains a 3-byte long version string called versionStr.

Return Values

ZL_STATUS is 0 (ZL_PMBUS_OK) if successful, otherwise a defined error code is returned.

Example

```
ZL_HANDLE myHandle; ZL_STATUS myStatus;
ZL_FW_VERSION fwversion;
int i;

myStatus = ZL_FWVersion( myHandle, &fwversion );
if( myStatus == ZL_PMBUS_OK)
{
  printf("Firmware version: ");
  for(i = 0; i < 3; i++)
    printf("%c", fwversion.versionStr[i] );
  printf("\n");
}
else {
  printf("Error in reading firmware version \n");
}
```

ZL_DeviceScan

Returns a listing of all Zilker Labs PMBus Interfaces attached to the computer. The list is composed of the serial numbers for each device, such that one can choose to open a specific device from the list using ZL_OpenDeviceBySerial.

```
ZL_STATUS ZL_DeviceScan( unsigned long *numDevices,
                         ZL_SERIAL *deviceSerials,
                         const char *deviceName)
```

Parameters

*numDevices	Pointer that returns the number of devices attached to the computer
*deviceSerials	Pointer to an array of ZL_SERIAL structures
*deviceName	C-String pointer to the name of the devices we are trying to scan

Return Value

ZL_STATUS is 0 (ZL_PMBUS_OK) if successful, otherwise a defined error code is returned.

Comments

Because ZL_DeviceScan requires a pointer to the list of serials the function will return, one must allocate enough space to include the list of serials in the first place. We recommend calling ZL_NumberOfDevices first to see how many devices are attached, then use the return data from the prior function to allocate memory for the list. This method is shown in the example below

Example

```
ZL_STATUS myStatus;
ZL_SERIAL* deviceSerials; //pointer to an array of serials
unsigned long numDevices, i;


// First, see how many devices are connected
myStatus =  ZL_NumberOfDevices( &numDevices,
                            "Zilker Labs PMBus Interface" );

if( numDevices == 0 ) {
  printf("No Devices Found.\n");
  return 0;
}


// Knowing the number of devices, create a list
// of device serials.
```

```
// Allocate space for device serials
deviceSerials = (ZL_SERIAL*)malloc(sizeof(ZL_SERIAL) *
                                                numDevices);

// Generate List of detected devices
myStatus = ZL_DeviceScan( &numDevices,
                          deviceSerials,
                          "Zilker Labs PMBus Interface" );

// Print List of devices
printf("Devices Found: \n");
for( i = 0; i < numDevices; i++ ) {
  printf("%s\n", deviceSerials[i].numStr);
}


// Open first device from list
myStatus = ZL_OpenDeviceBySerial( &myHandle,
                                  &deviceSerials[0] );

if(myStatus) { //Error in opening device
  printf("\nError in opening device \"%s\". \n",
            deviceSerials[0].numStr );
}
else {         //Device Successfully opened
  printf("\nDevice \"%s\" Successfully Opened\n",
         deviceSerials[0].numStr );
}


// Close device
myStatus = ZL_CloseDevice( myHandle );
```

ZL_DetectDevice

This function is used to see if a device handle is still open, and is typically used to report an error if an invalid handle is passed, or to realize that a device needs to be re-opened.

```
ZL_STATUS ZL_DetectDevice( const ZL_HANDLE deviceHandle,
                           const char *deviceName )
```

Parameters

deviceHandle	The device handle you are testing
*deviceName	The device name associated with the handle you are testing

Return Value

ZL_STATUS is 0 (ZL_PMBUS_OK) if the device handle still exists, otherwise a defined error code is returned.

Example

```
ZL_HANDLE myHandle;
ZL_STATUS myStatus;

//Attempt to open the device
myStatus =  ZL_OpenDeviceByName( &myHandle,
                           "Zilker Labs PMBus Interface" );

//see if device is already detected
if( !( ZL_DetectDevice( myHandle,
                        "Zilker Labs PMBus Interface" ) ) )
{
  printf("Device Detected after handle open.(expected)\n");
}
else {
  printf("Device not detected after handle open!\n");
}
```

ZL_NumberOfDevices

Returns the number of devices currently attached to the computer.

```
ZL_STATUS ZL_NumberOfDevices( unsigned long *numDevices,
                              const char *deviceName)
```

Parameters

*numDevices	The returned number of attached devices
*deviceName	C-String pointer to the name of the devices we are trying to scan

Return Value

ZL_STATUS is 0 (ZL_PMBUS_OK) if successful, otherwise a defined error code is returned.

Example

```
ZL_STATUS myStatus;
unsigned long numDevices;

// See how many devices are connected
myStatus =  ZL_NumberOfDevices( &numDevices,
                            "Zilker Labs PMBus Interface" );

printf("%d devices found.\n", numDevices);
```

ZL_OpenDeviceByName

Opens the first device found that matches the provided device name.

```
ZL_STATUS ZL_OpenDeviceByName( ZL_HANDLE *deviceHandle,
                               char *deviceName )
```

Parameters

*deviceHandle	Pointer to the opened device handle.
*deviceName	C-String pointer to the name of the device we are trying to open.

Return Value

ZL_STATUS is 0 (ZL_PMBUS_OK) if successful, otherwise a defined error code is returned.

Example

```
ZL_HANDLE myHandle;
ZL_STATUS myStatus;

//Attempt to open the device
myStatus =  ZL_OpenDeviceByName( &myHandle,
                            "Zilker Labs PMBus Interface" );
```

ZL_OpenDeviceBySerial

Opens the device found with a matching serial number. This function is typically used after calling ZL_DeviceScan.

```
ZL_STATUS ZL_OpenDeviceBySerial( ZL_HANDLE *deviceHandle,
                                 ZL_SERIAL *deviceSerial )
```

Parameters

*deviceHandle	Pointer to the opened device handle.
*deviceSerial	Pointer to the ZL_SERIAL structure containing the serial number of the device we want to open.

Return Value

ZL_STATUS is 0 (ZL_PMBUS_OK) if successful, otherwise a defined error code is returned.

Example

See Pages 4-5.

ZL_CloseDevice

Closes the device associated with the provided handle.

```
ZL_STATUS ZL_CloseDevice( const ZL_HANDLE deviceHandle )
```

Parameters

deviceHandle The device handle we are trying to close.

Return Value

ZL_STATUS is 0 (ZL_PMBUS_OK) if successful, otherwise a defined error code is returned.

Example

See Pages 4-5.

ZL_PMBUS_Write

Performs a PMBus transmission in the form of a Quick Command, Send Byte, Write Byte, Write Word, or Block Write transfer.

```
ZL_STATUS ZL_PMBUS_Write( const ZL_HANDLE deviceHandle,
                          const unsigned char numDevices,
                          PMBUS_RW_TRANSFER *pmTrans );
```

Parameters

deviceHandle	The device handle we will use to perform the transmission.
numDevices	The number of devices we will be addressing. This should always be passed 1 unless a group command is being performed.
*pmTrans	Pointer to the PMBUS_RW_TRANSFER structure, which includes the PMBus device address, transfer type, command byte(s), and data we want to send.

Return Values

ZL_STATUS is 0 (ZL_PMBUS_OK) if successful, otherwise a defined error code is returned.

Example (Quick Command)

```
ZL_STATUS myStatus;
PMBUS_RW_TRANSFER pmTrans;

//Setup PMBus transfer struct for a Quick Command Write transmission
pmTrans.address = 0x20;
pmTrans.transferType = TTYPE_PMBUS_QUICKCMD_WRITE;

myStatus = ZL_PMBUS_Write( deviceHandle,
                           1, //numDevices
                           &pmTrans );
```

Example (Send Byte)

```
//PMBus Command
const unsigned char restore_user_all = 0x16;

ZL_STATUS myStatus;
PMBUS_RW_TRANSFER pmTrans;

//Setup PMBus transfer struct for a Send Byte transmission
pmTrans.address = 0x20;
pmTrans.transferType = TTYPE_PMBUS_SEND_BYTE;
```

```
pmTrans.cmdLength = 1;
pmTrans.cmdBytes[0] = restore_user_all;

myStatus = ZL_PMBUS_Write( deviceHandle,
                           1, //numDevices
                           &pmTrans );
```

Example (Write Byte)

```
const unsigned char operation = 0x01; //PMBus Command Definition
ZL_STATUS myStatus;
PMBUS_RW_TRANSFER pmTrans;

//Setup PMBus transfer struct for a Write Byte transmission
pmTrans.address = 0x20;
pmTrans.transferType = TTYPE_PMBUS_WRITE_BYTE;
pmTrans.cmdLength = 1;
pmTrans.cmdBytes[0] = operation;
pmTrans.paramLength = 1;
pmTrans.paramBytes[0] = 0x40; //Perform a "Soft-Off"

myStatus = ZL_PMBUS_Write( deviceHandle,
                           1, //numDevices
                           &pmTrans );
```

Example (Write Word)

```
const unsigned char vout_command = 0x21; //PMBus Command

ZL_STATUS myStatus;
PMBUS_RW_TRANSFER pmTrans;

//Setup PMBus transfer struct for a Write Byte transmission
pmTrans.address = DEVICE_ADDRESS_1;
pmTrans.transferType = TTYPE_PMBUS_WRITE_WORD;
pmTrans.cmdLength = 1;
pmTrans.cmdBytes[0] = vout_command;
pmTrans.paramLength = 2;
pmTrans.paramBytes[0] = 0x3D; // NOTE: The purpose of these
pmTrans.paramBytes[1] = 0x6A; // parameter bytes are to
                              // send 3.32 Volts = 0x6A3D.
                              // They are sent in the
                              // little-endian format as
                              // required by PMBus spec.
```

```
myStatus = ZL_PMBUS_Write( deviceHandle,
                           1, //numDevices
                           &pmTrans );
```

Example (Block Write – Writing an arbitrary sequence)

```
const unsigned char ZL2005_pid_taps = 0xD5; //PMBus Command
ZL_STATUS myStatus;
PMBUS_RW_TRANSFER pmTrans;

//Setup PMBus transfer struct for a Write Byte transmission
pmTrans.address = 0x20;
pmTrans.transferType = TTYPE_PMBUS_BLOCK_WRITE;
pmTrans.cmdLength = 1;
pmTrans.cmdBytes[0] = ZL2005_pid_taps;
pmTrans.paramLength = 9;
// Write PID_TAPS   A=1634, B=-2799, C=1227
pmTrans.paramBytes[0] = 0x40; //Coefficient A –
                              // mantissa, low-byte
pmTrans.paramBytes[1] = 0xCC; //Coefficient A –
                              // mantissa, high-byte
pmTrans.paramBytes[2] = 0x7B; //Coefficient A –
                              // exponent + sign
pmTrans.paramBytes[3] = 0xF0; //Coefficient B –
                              // mantissa, low-byte
pmTrans.paramBytes[4] = 0xAE; //Coefficient B –
                              // mantissa, high-byte
pmTrans.paramBytes[5] = 0xFC; //Coefficient B –
                              // exponent + sign
pmTrans.paramBytes[6] = 0x60; //Coefficient C –
                              // mantissa, low-byte
pmTrans.paramBytes[7] = 0x99; //Coefficient C –
                              // mantissa, high-byte
pmTrans.paramBytes[8] = 0x7B; //Coefficient C –
                              // exponent + sign

myStatus = ZL_PMBUS_Write( deviceHandle,
                           1, //numDevices
                           &pmTrans );
```

Example (Block Write – Writing an ASCII string)

```
const unsigned char mfr_id = 0x99; //PMBus Command

char asciiData[] = "hello world!";
ZL_STATUS myStatus;
PMBUS_RW_TRANSFER pmTrans;

//Setup PMBus transfer struct for a Write Byte transmission
pmTrans.address = 0x20;
pmTrans.transferType = TTYPE_PMBUS_BLOCK_WRITE;
pmTrans.cmdLength = 1;
pmTrans.cmdBytes[0] = mfr_id;
strcpy( &pmTrans.paramBytes, &asciiData[0] );
pmTrans.paramLength = (unsigned char)
                      strlen( &asciiData[0] );

myStatus = ZL_PMBUS_Write( deviceHandle,
                           1, //numDevices
                           &pmTrans );
```

ZL_PMBUS_Read

Performs a PMBus transmission in the form of a Receive Byte, Read Byte, Read Word, or Block Read transfer type.

```
ZL_STATUS ZL_PMBUS_Read( const ZL_HANDLE deviceHandle,
                         PMBUS_RW_TRANSFER *pmTrans );
```

Parameters

deviceHandle	The device handle we will use to perform the transmission.
*pmTrans	Pointer to the PMBUS_RW_TRANSFER structure, which includes the PMBus device address, transfer type, command byte(s), and stores the data we will receive.

Return Value

ZL_STATUS is 0 (ZL_PMBUS_OK) if successful, otherwise a defined error code is returned.

Example (Receive Byte)

```
#define ALERT_RESPONSE_ADDRESS  0x0C
ZL_STATUS myStatus;
PMBUS_RW_TRANSFER pmTrans;

//Setup PMBus transfer struct for Receive Byte transmission
pmTrans.address = ALERT_RESPONSE_ADDRESS;
pmTrans.transferType = TTYPE_PMBUS_RECV_BYTE;

myStatus = ZL_PMBUS_Read( deviceHandle,
                          &pmTrans );

if(myStatus) { //Exit if error occured
  printf("Error in Receive Byte Example.\n");
  printf("(This is likely due to no faults\
           present on any devices)\n\n");
  return;
}

//Otherwise, Print byte contents
printf("Receive Byte Contents: %#02x,\
        meaning a device at address %#02x has a fault.\n",
        pmTrans.paramBytes[0],
        (pmTrans.paramBytes[0]>>1) & ~(0x80) );
```

Example (Read Byte)

```
const unsigned char operation = 0x01; //PMBus Command

ZL_STATUS myStatus;
PMBUS_RW_TRANSFER pmTrans;

//Setup PMBus transfer struct for a Read Byte transmission
pmTrans.address = 0x20;
pmTrans.transferType = TTYPE_PMBUS_READ_BYTE;
pmTrans.cmdLength = 1;
pmTrans.cmdBytes[0] = operation;

myStatus = ZL_PMBUS_Read( deviceHandle,
                          &pmTrans );

if(myStatus) { //Exit if error occured
  printf("Error in Read Byte Example.\n\n");
  return;
}

//Otherwise, Print byte contents
printf("Read Byte Contents: %#02x.\n",
       pmTrans.paramBytes[0]);
```

Example (Read Word)

```
const unsigned char vout_command = 0x21; //PMBus Command

ZL_STATUS myStatus;
PMBUS_RW_TRANSFER pmTrans;

//Setup PMBus transfer struct for a Read Word transmission
pmTrans.address = 0x20;
pmTrans.transferType = TTYPE_PMBUS_READ_WORD;
pmTrans.cmdLength = 1;
pmTrans.cmdBytes[0] = vout_command;

myStatus = ZL_PMBUS_Read( deviceHandle,
                          &pmTrans );

if(myStatus) { //Exit if error occured
  printf("Error in Read Word Example.\n\n");
  return;
}

//Otherwise, Print byte contents
//NOTE: I print the second byte first since
//      the data for VOUT_COMMAND is sent and received
//      in little-endian.
printf("Read Word Contents: %#02x%02x.\n",
       pmTrans.paramBytes[1], pmTrans.paramBytes[0]);
```

Example (Block Read of Arbitrary bytes)

```
const unsigned char ZL2005_pid_taps = 0xD5; //PMBus Command
const unsigned char ZL2005_pid_taps_length = 9;

ZL_STATUS myStatus;
PMBUS_RW_TRANSFER pmTrans;

//Setup PMBus transfer struct for a Read Word transmission
pmTrans.address = 0x20;
pmTrans.transferType = TTYPE_PMBUS_BLOCK_READ;
pmTrans.cmdLength = 1;
pmTrans.cmdBytes[0] = ZL2005_pid_taps;

myStatus = ZL_PMBUS_Read( deviceHandle,
                          &pmTrans );

if(myStatus) { //Exit if error occured
  printf("Error in Block Read Example.\n\n");
  return;
}
else if(pmTrans.paramLength != ZL2005_pid_taps_length) {
  printf("Invalid parameter length returned.\n\n");
  return;
}

//Print out pid_taps coefficients
printf("Block Read Demo One - PID_TAPS readout:\n");
printf("  Coefficient A: %#02x%02x%02x\n",
          pmTrans.paramBytes[6],
          pmTrans.paramBytes[7],
          pmTrans.paramBytes[8] );
printf("  Coefficient B: %#02x%02x%02x\n",
          pmTrans.paramBytes[3],
          pmTrans.paramBytes[4],
          pmTrans.paramBytes[5] );
printf("  Coefficient C: %#02x%02x%02x\n",
          pmTrans.paramBytes[0],
          pmTrans.paramBytes[1],
          pmTrans.paramBytes[2] );
```

Example (Block Read of ASCII Characters)

```
//PMBus Command
const unsigned char ZL2005_device_id = 0xE4;

ZL_STATUS myStatus;
PMBUS_RW_TRANSFER pmTrans;
unsigned char i;

//Setup PMBus transfer struct for a Read Word transmission
pmTrans.address = 0x20;
pmTrans.transferType = TTYPE_PMBUS_BLOCK_READ;
pmTrans.cmdLength = 1;
pmTrans.cmdBytes[0] = ZL2005_device_id;

myStatus = ZL_PMBUS_Read( deviceHandle,
                          &pmTrans );

if(myStatus) { //Exit if error occured
  printf("Error in Block Read Example.\n\n");
  return;
}

//print non null-terminated ASCII string
printf("Block Read Output: ");
for(i = 0; i < pmTrans.paramLength; i++) {
  printf("%c", pmTrans.paramBytes[i]);
}
```

ZL_PMBUS_SetPEC

Enables or disables Packet Error Checking (PEC) on the device.
NOTE: This command works only on firmware revisions 03 and greater.

```
ZL_STATUS ZL_PMBUS_SetPEC( const ZL_HANDLE deviceHandle,
                           const unsigned char PECFlagIn );
```

Parameters

deviceHandle	The device handle we will use to enable/disable PEC.
PECFlagIn	Flag which takes on the definitions of either PEC_ENABLE or PEC_DISABLE

Return Value

ZL_STATUS is 0 (ZL_PMBUS_OK) if successful, otherwise a defined error code is returned.

Example

```
ZL_HANDLE myHandle;
ZL_STATUS myStatus;

myStatus = ZL_PMBUS_SetPEC( myHandle, PEC_ENABLE );
if( myStatus == ZL_PMBUS_OK )
{
  printf("Set pec\n");
}
else
{
  printf("Error in setting pec.\n");
}
```

ZL_PMBUS_GetPEC

Tells whether Packet Error Checking (PEC) is enabled/disabled.
NOTE: This command works only on firmware revisions 03 and greater.

```
ZL_STATUS ZL_PMBUS_GetPEC( const ZL_HANDLE deviceHandle,
                           unsigned char * PECFlagOut );
```

Parameters

deviceHandle	The device handle we will use to enable/disable PEC.
*PECFlagOut	Pointer to unsigned character that returns with either PEC_ENABLE or PEC_DISABLE

Return Value

ZL_STATUS is 0 (ZL_PMBUS_OK) if successful, otherwise a defined error code is returned.

Example

```
ZL_HANDLE myHandle;
ZL_STATUS myStatus;
unsigned char pecEnable;

myStatus = ZL_PMBUS_GetPEC( myHandle, &pecEnable ) == 0
if( myStatus == ZL_PMBUS_OK )
{
  printf("Pec set to: %d\n", pecEnable);
}
else {
  printf("Error in reading pec.\n");
}
```

ZL_PMBus Structures, Types, and Values

The ZL_PMBus API makes use of a few special structures to make it easy to send and receive the data you need. Below is a list of the structures and a description of how they work.

PMBUS_RW_TRANSFER

The PMBUS_RW_TRANSFER is a structure used with the ZL_PMBUS_Write and ZL_PMBUS_Read commands. It contains the transfer type, address, command byte(s), and parameter byte(s) that will be used to communicate with the device.

```
typedef struct PMBusRWStruct {
  unsigned char transferType;
  unsigned char address;
  unsigned char cmdLength;
  unsigned char cmdBytes[2];
  unsigned char paramLength;
  unsigned char paramBytes[256];
} PMBUS_RW_TRANSFER;
```

The transferType variable should be set to one of the predefined transfer types found in ZL_PMBus.h. The transfer types are also listed below:

```
// Transfer Types used by ZL_PMBUS_Write
#define TTYPE_PMBUS_QUICKCMD_READ     1
#define TTYPE_PMBUS_QUICKCMD_WRITE    2
#define TTYPE_PMBUS_SEND_BYTE         4
#define TTYPE_PMBUS_WRITE_BYTE        7
#define TTYPE_PMBUS_WRITE_WORD        8
#define TTYPE_PMBUS_BLOCK_WRITE       10
// Transfer Types used by ZL_PMBUS_Read
#define TTYPE_PMBUS_RECV_BYTE         3
#define TTYPE_PMBUS_READ_BYTE         5
#define TTYPE_PMBUS_READ_WORD         6
#define TTYPE_PMBUS_BLOCK_READ        11
// Transfer Types used with ZL_PMBUS_ProcessCall
#define TTYPE_PMBUS_PROC_CALL         9
#define TTYPE_PMBUS_BLKWR_BLKRD_PROC  12
```

The address variable is passed as just the lower 7 bytes of an address byte in a PMBus transmission. This means that for an address of 0x20 in PMBUS_RW_TRANSFER, 0x40 or 0x41 will be sent in an Address+Write or Address+Read, respectively. The address is shifted left in the MCU code.

The cmdLength variable describes how many command bytes need to be sent. This value is typically 1 unless you are doing an extended command transfer, in which case it should be 2.

The cmdBytes array holds the command byte to be sent as well as an extended command byte. The bytes must be put in the array in the order that they are sent. This means that for non-extended command transmissions the command byte must be placed in cmdByte[0].

The paramLength variable is used to either describe the number of bytes to be sent, or to read the number of bytes that were received.

The paramBytes array holds the parameter bytes we want to send, but can also contain the parameter bytes we received. Parameter bytes should be put in the order they are sent.

ZL_HANDLE

The ZL_HANDLE type is a pointer that points to the instance of the FTDI USB-UART converter attached to the computer.

ZL_STATUS

ZL_STATUS is a signed long variable that is typically used to return whether a command was successful or not. DLL Versions 0.4 and greater include the following status codes:

```
API-Wide Error Codes
ZL_PMBUS_OK                          0    // No Error
ZL_PMBUS_ERR_GENERIC                -1
ZL_PMBUS_ERR_DEVHANDLE              -2
ZL_PMBUS_ERR_TRANS_DATA_INV         -3
ZL_PMBUS_ERR_TRANS_DATA_UNDERRUN    -4
ZL_PMBUS_ERR_TRANS_DATA_OVERRUN     -5
ZL_PMBUS_ERR_TRANS_TIMEOUT          -6

Error codes related to sending PMBus data
ZL_PMBUS_ERR_SEND_START            -100
ZL_PMBUS_ERR_SEND_REP_START        -101
ZL_PMBUS_ERR_SEND_ADR              -102
ZL_PMBUS_ERR_SEND_REP_ADR          -103
ZL_PMBUS_ERR_SEND_CMD              -104
ZL_PMBUS_ERR_SEND_PARAMLEN         -105
ZL_PMBUS_ERR_SEND_PARAM            -106
ZL_PMBUS_ERR_SEND_PEC              -107
ZL_PMBUS_ERR_SEND_STOP             -108
```

```
Error codes related to receiving PMBus data
ZL_PMBUS_ERR_RECV_PARAMLEN        -140
ZL_PMBUS_ERR_RECV_PARAM           -141
ZL_PMBUS_ERR_RECV_PEC             -142

PMBus-specific user input errors
ZL_PMBUS_ERR_BAD_TTYPE            -170
ZL_PMBUS_ERR_BAD_CMDLEN           -171
ZL_PMBUS_ERR_NUMDEVICES_IS_ZERO   -172
```

More information on these error codes can be found in the ZL_PMBus.h API header file.

ZL_VERSION

ZL_VERSION is a structure that contains the major and minor release numbers. The version of the dll you are linking to can be found via the ZL_DLLVersion command.

```
typedef struct revision {
  long major;
  long minor;
} ZL_VERSION;
```

ZL_FW_VERSION

ZL_FW_VERSION is a structure that contains the firmware version. The version of firmware your MCU is using can be found via the ZL_FWVersion command.

```
typedef struct fwRevision {
  char versionStr[3];
} ZL_FW_VERSION;
```

ZL_SERIAL

ZL_SERIAL contains a C-String buffer that holds a series of ASCII characters that serve as each device's serial number. The serial numbers retrieved via ZL_DeviceScan are stored in a small EEPROM used by the FTDI USB-UART converter.

Revision History

Date	Rev. #	
5/25/06	2.0	Initial Release
6/6/07	3.0	Added ZL_FWVersion, ZL_SetPEC, & ZL_GetPEC. Added ZL_STATUS Error Codes
5/01/09	AN2018.0	Assigned file number AN2018 to app note as this will be the first release with an Intersil file number. Replaced header and footer with Intersil header and footer. Updated disclaimer information to read “Intersil and it’s subsidiaries including Zilker Labs, Inc.” No changes to application note content.

Zilker Labs, Inc.
4301 Westbank Drive
Building A-100
Austin, TX 78746

Tel: 512-382-8300
Fax: 512-382-8329