Power Management Bus (PMBus™) slave with XMC™

XMC1000/XMC4000

About this document

Scope and purpose

This application note gives details of the PMBus™ standard and describes the implementation of the protocol using the Infineon XMC™ microcontrollers and PMBusLib.

Applicable products

- XMC1000 and/or XMC4000 microcontroller families

References

Infineon: PMBusLib and example code, XMC™ software downloads

Infineon: XMC™ Family, http://www.infineon.com/XMC™

PMBus™ Power system management protocol specification I, PMBus™ specification Revision 1.2 Part 1

PMBus™ Power system management protocol specification II, PMBus™ specification Revision 1.2 Part 2

Intended audience

This application note is targeting enginners and developers of power applications that involve PMBus™ communication.

Table of contents

1 Overview of PMBus™ standard

The Power Management Bus (PMBus™) is a free and open standard power-management protocol with a fully defined command language that facilitates communication with power converters and other devices in a power system. In this document, only version 1.2 of the PMBus™ specification is considered. It is a variant of System Management Bus (SMBus™), and is a relatively slow speed, two-wire communications protocol based on the Inter-Integrated Circuit Bus (I2C). The PMBus™ standard defines over two hundred commands, and a substantial number of the commands are specifically tailored for power conversion processes.

The Universal Serial Interface Channel (USIC) module contained in the XMC™ microcontroller is a flexible interface module covering several serial communication protocols including I2C. This dedicated hardware plus the PMBus™ library (PMBusLib), provided free of charge, makes XMC™ an asset for power conversion applications requiring PMBus™ capabilities.

1.1 System overview

The physical layer of PMBus™ consists of a single master and multiple slaves. It is a two-line communication protocol based on I2C. The Serial Bus Data and Serial Bus Clock line perform I2C functions and are essential lines for the PMBus™ functionality. There are two additional optional signals, SMBALERT# and control signal.

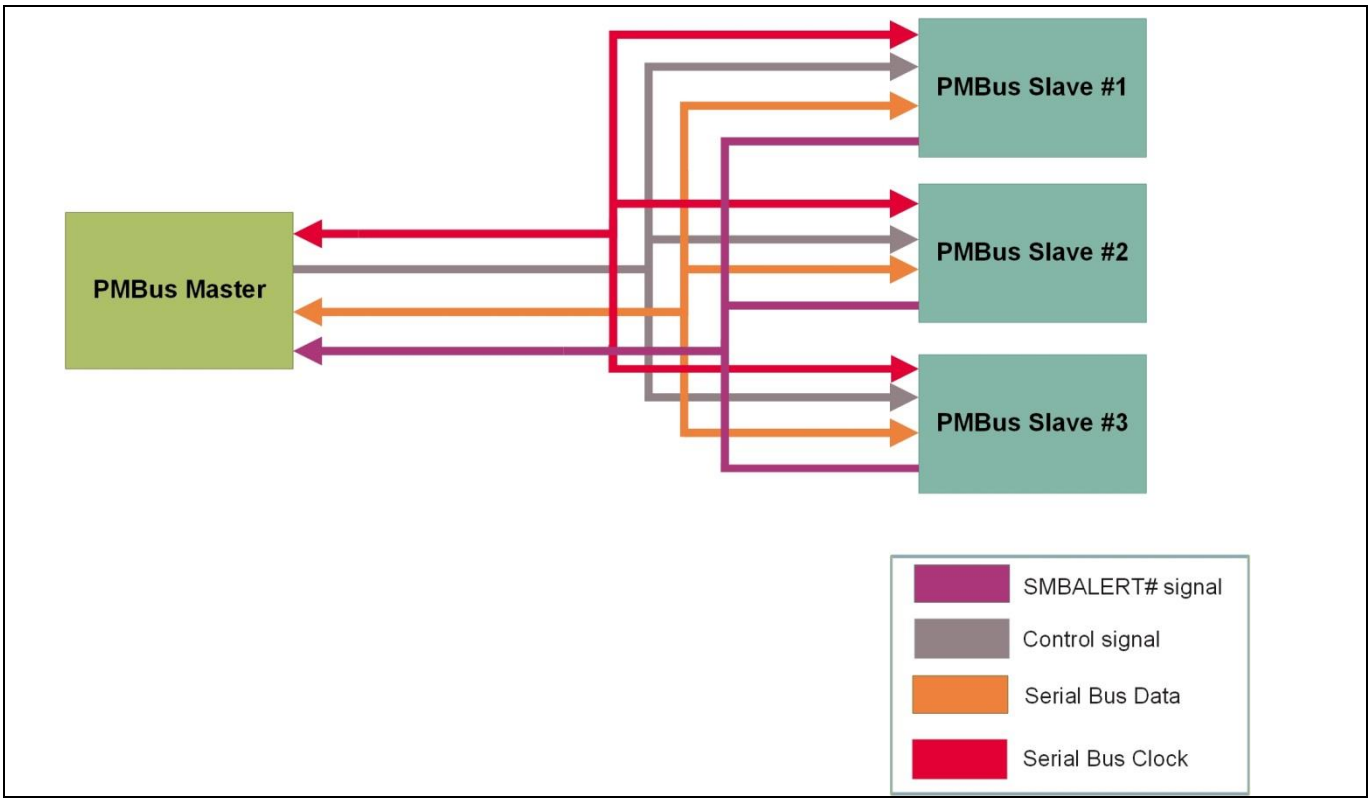

Figure 1 PMBus™ communication architecture

SMBALERT# signal

The SMBALERT# signal line is an interrupt line for slave-only devices to notify the master. The slave can signal the master through the SMBALERT# line when a fault occurs and has to be reported.

The master will process the interrupt and access all the SMBALERT# devices using alert response address and the devices that assert the signal will respond to the alert response address.

CONTROL signal

The CONTROL signal is an output signal on a PMBus™ master. The master can use this signal in conjunction with commands via the serial bus to turn the slave on or off.

1.2 PMBus™ features

The following features are recommended to improve the communication and data reliability, creating a robust PMBus™ system.

Timeout mechanism

The PMBus™ system adopts a timeout mechanism to resume the functionality if a fault condition occurs. If the I2C clock line is held low longer than a certain time (timeout) then the device must reset the communication to resume normal functionality.

Fault reporting

The slave is required to report fault conditions to the master during operation. The report can be via:-

- the SMBALERT# signal
- the host-notify protocol (section 1.3.6).

Packet Error Checking (PEC)

An optional Packet Error Checking (PEC) feature is recommended, in order to significantly increase the data reliability. PEC checks the reliability of the received data packets via a cyclic redundancy check-8 (CRC-8) algorithm.

1.3 PMBus™ protocols

In the XMC™ implementation, the following types of PMBus™ message structures are supported:

- Send byte
- Write byte
- Write word
- Read byte
- Read word
- Host notify

1.3.1 Send byte

This is used by the PMBus™ master to send a command to the slave to perform simple tasks that do not require any data bytes.

For example, the CLEAR_FAULTS command is used to clear all of the fault flags in the slave system.

The master starts the communication by generating a start condition (S) and then writes a 7 bit slave address with a write bit, followed by the 8 bit command.

The slave will acknowledge each byte received (shaded portion in the figure).

The communication is terminated by a stop condition (P) generated by the master.

The additional PEC byte ensures data reliability.

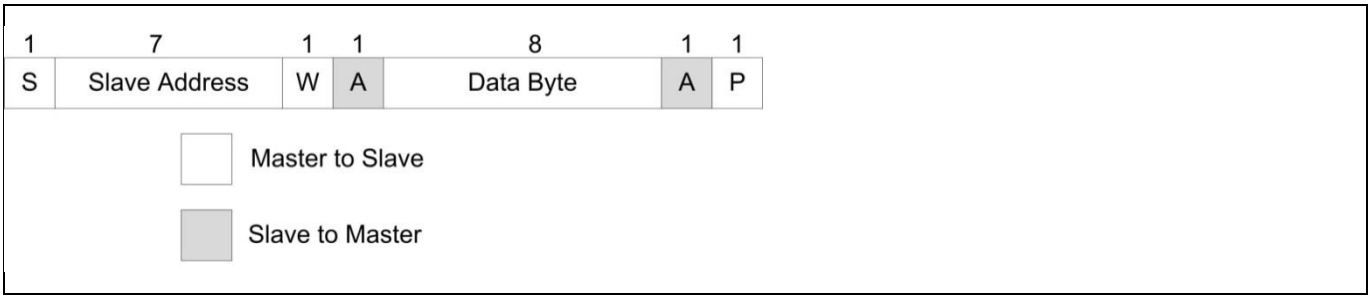

Figure 2 Send byte packet without PEC

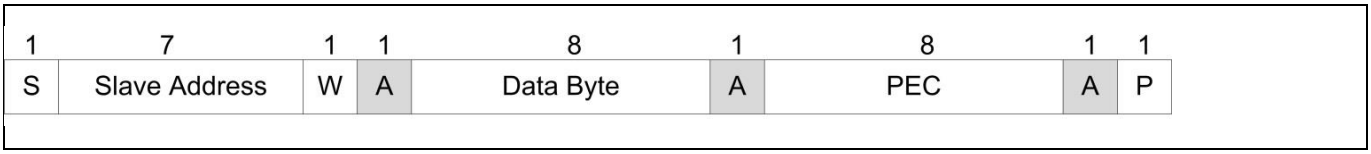

Figure 3 Send byte packet with PEC

1.3.2 Write byte

Used by the master to write a one-byte value to certain variables in slave settings.

For example, V_{OUT}_MODE command can be used to change the voltage representation by writing a new variable value.

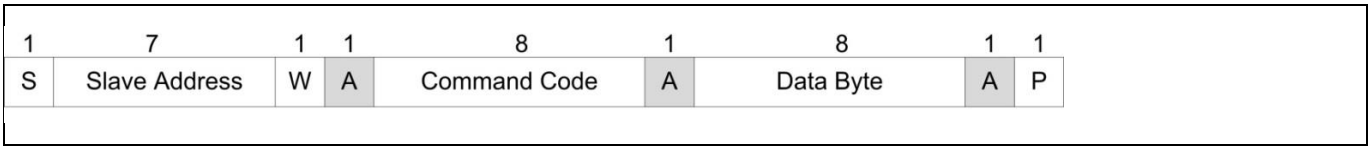

Figure 4 Write byte packet without PEC

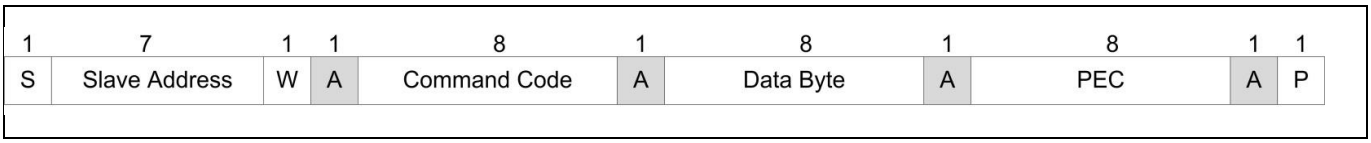

Figure 5 Write byte packet with PEC

1.3.3 Write word

Similar to write byte, this type of command is used to write one word (2 bytes) of information into the slave variable.

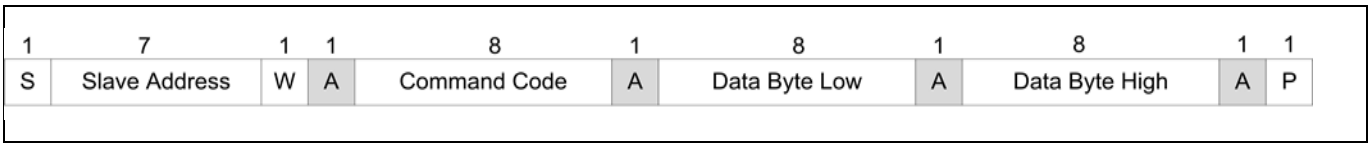

Figure 6 Write word packet without PEC

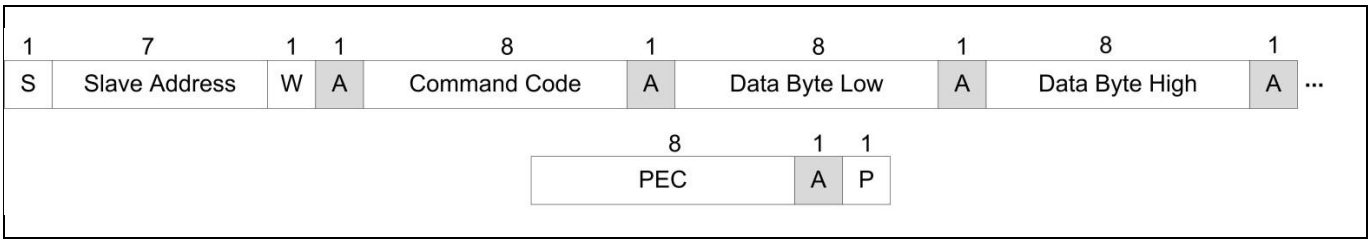

Figure 7 Write word packet with PEC

1.3.4 Read byte

This is used by the master to read one byte of information from the slave.

For example, the STATUS_BYTE command enables the master to read the error status of the slave.

As shown below, after the acknowledgement of the command code from the slave, the master sends a repeated start (Sr) condition, followed by the 7-bit address with a read bit. The slave will then acknowledge and send out one byte data.

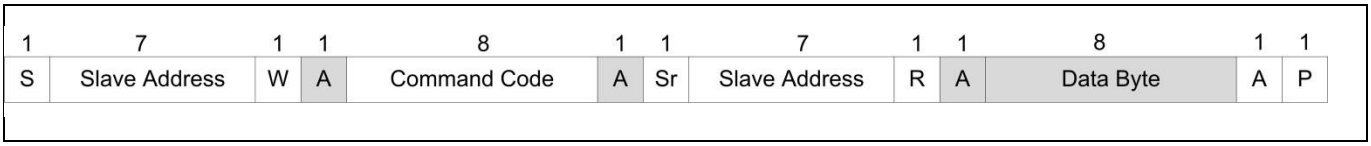

Figure 8 Read byte packet without PEC

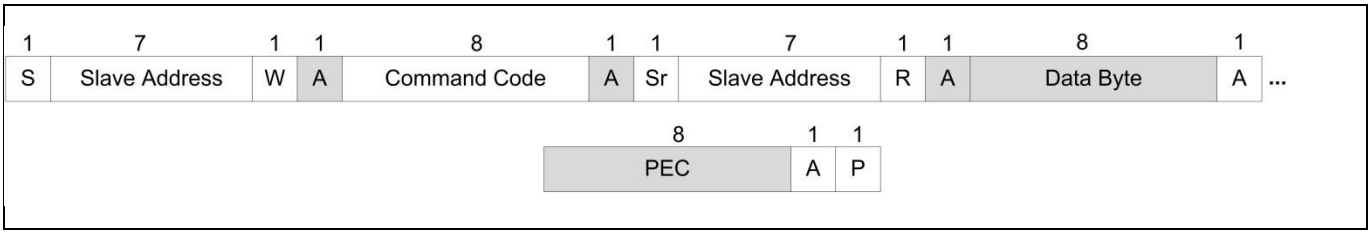

Figure 9 Read byte packet with PEC

1.3.5 Read word

Similar to read byte protocol, read word is used by the master to read one word of data from the PMBus™ slave.

For example, the READ_TEMPERATURE_1 command is used to read the temperature of the slave.

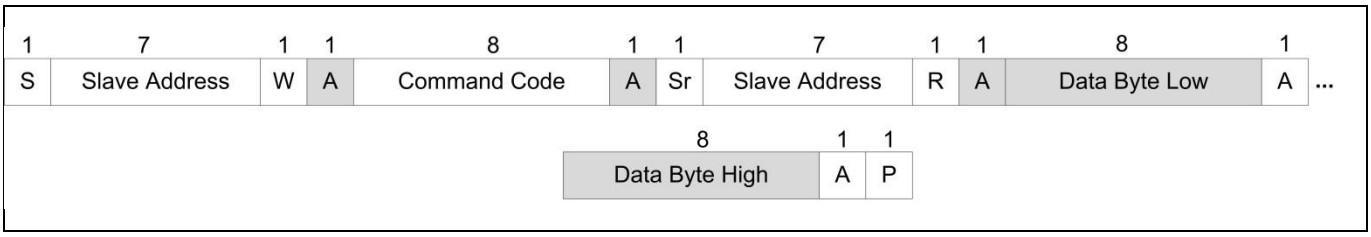

Figure 10 Read word packet without PEC

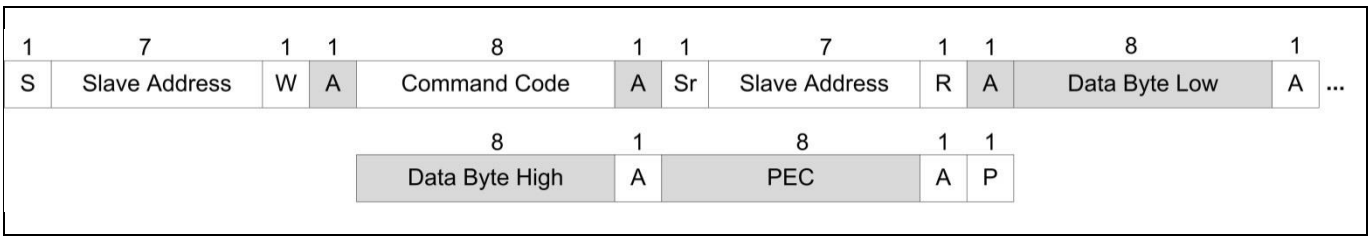

Figure 11 Read word packet with PEC

1.3.6 Host notify

When a fault condition occurs in the slave, the slave is required to report the error back to the master. In this case, the slave becomes the bus master and initiates a communication session using the host notify protocol. The slave address and two-byte status information are transmitted to the master.

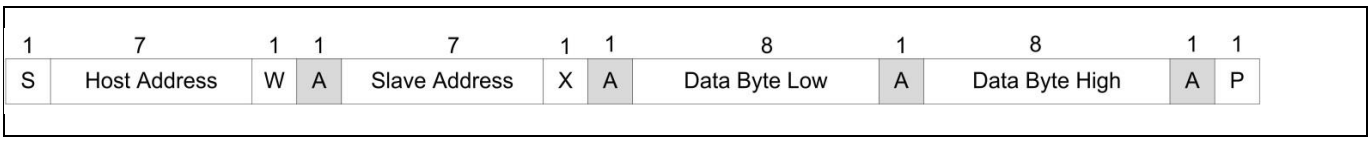

Figure 12 Host notify protocol

1.4 PMBus™ commands and data formats

The PMBus™ commands are one byte command codes including output voltage related commands, fault related commands, read status command and parametric write/read commands. With these commands the master can configure the slave peripherals and read the status.

To support the write/read of the parametric data (except for output voltage), data is represented in 2 main types of format:

- LINEAR: receive and transmit values such as volts, amperes, milliseconds or degrees Celsius. This format provides the least burden on the master at the expense of data manipulation in the PMBus™ device.
- DIRECT: receive and transmit data as a two byte two's complement binary integer. This format provides the least burden on the PMBus™ device at the expense of more complex calculations in the master.

Output voltage and output voltage related parameters are represented in three formats:

- LINEAR

Overview of PMBus™ standard

- A format that supports transmitting the VID codes of popular microprocessors via the PMBus™.
- DIRECT

Each slave is required to support one type of data format for each command supported. Not all the commands are required to be supported by PMBus™ systems, the supported command list should be determined according to each application's needs.

2 Implementation of PMBus™ slave on XMC™: PMBusLib

Based on the XMC™ low level drivers for peripherals (XMCLib), PMBusLib provides the user with all the software utilities for implementing the slave part of the PMBus™ Rev. 1.2 in Infineon XMC™ microcontrollers. Below, Figure 13 shows the layers of the library, allowing better visualization of the code.

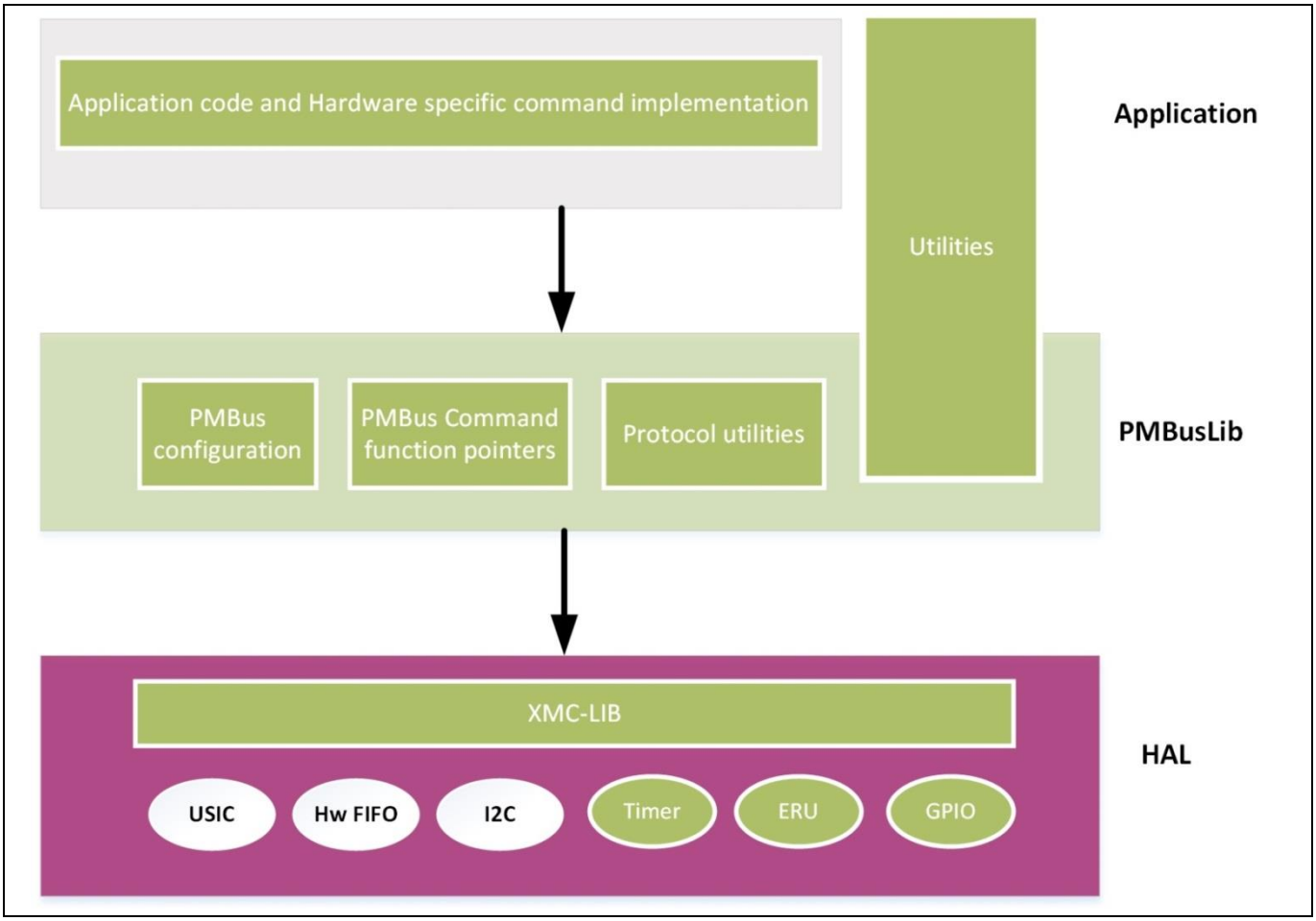

Figure 13 Different layers of the PMBusLib

HAL layer

HAL refers to the Hardware Abstraction Layer (XMCLib) provided by Infineon, which exposes all the functionality of the different peripherals in the XMC™ microcontroller. For PMBus™ slave implementation, the following components from the XMCLib are used:

- *USIC - I2C and FIFO*

  USIC is the flexible serial communication peripheral that can be configured to work as SPI, I2C or UART. For PMBusLib, USIC is configured to work in I2C mode with inbuilt FIFO's used for reception and transmission of packets over the I2C physical interface. FIFO is configured to trigger an interrupt when a byte is received in the FIFO buffer and, in a similar way FIFO is used to trigger the command transmission from the slave device when responding to a master.

  The following events are configured for I2C:

  - Start indicates that the packet transmission is being started from master-slave. This event will be triggered for every slave on the bus.

- Repeated start indicates the start of new packet transmission between a stream of bytes from the master. This event will be triggered for every slave on the bus.
- Slave read request - the master transmits an address byte before sending any command and only the slave device matching the slave address sent by the master will be triggered with this event.
- Stop condition - the end of packet transaction from master. The bus returns to the ”idle“ state once this condition occurs.

- *GPIO – SMBALERT# and control*

  Two optional lines per slave can be used for the following functions:

  - The control line is shared between multiple slaves in a PMBus™ loop. This is an input to the slave device from the master. If the master observes that the slave device is unable to meet the load power demand, then the master can request the slave to be disabled.
  - SMBALERT# is shared between multiple slaves in a PMBus™ loop. If the slave needs to notify the master of any critical error observed, then a PMBus™ slave can pull this line low. The master would poll all of the devices to read the status and, based on the status information, would take corrective action.

- *ERU - Event request unit*

  Implements the timeout that monitors the I2C data and clock line. The standard foresees a maximum of 35 ms, after this time (user configurable in PMBusLib), the bus will return to the “idle” state. If the device is not able to respond within 35 ms, the device shall re-initialize the I2C engine and return to the idle state. In this way, the ERU is configured to raise an event if the I2C lines are held low longer than 35 ms so that appropriate timeout actions can be performed from the device.

PMBusLib

PMBusLib provides an interface between the application and HAL. It configures all of the required peripherals and their respective events to implement a PMBus™ device as well as providing a skeletal implementation to handle a few generic checks that are required as part of any PMBus™ library.

PMBusLib helps in the development of the PMBus™ slave device. However, application developers should implement the supported commands and update the specific function pointer array. See section 3 “Getting started with PMBusLib”.

As part of PMBusLib, the following implementation is provided:

- *Configuration structure*

  Holds the complete configuration of the PMBus™ device. The user should modify this structure accordingly including enabling the correct macro to handle events for the peripheral. Refer to XMC_PMBUS_NODE_CONFIG_t in PMBusLib documentation to check the elements that can be configured, for example: address of device, PEC enable / disable, communication speed (100 K / 400 K).

- *PMBus™ command function pointer array*

  An array of function pointers that hold all the information related to a particular PMBus™ command. This list contains the pointer to any function that has to be executed for command received and additionally, other information such as the length of data that the command expects and command attributes (such as read / write, block read / block write supported). Note that the command number is the index of the array. Application code can make use of these command attributes to verify the type of

command received and flag related errors if not meeting the PMBus™ specification. Refer to XMC_PMBUS_NODE_CMD_INFO_t in PMBusLib documentation for more information.

- *Protocol utilities*

  PMBusLib supports the PEC (Packet error checking) capability which is optional to each manufacturer. However, if the device supports PEC, routines to evaluate the PEC of the packet are provided so that application developers can make use of this utility's functions. If PEC is enabled, every packet will have one extra PEC byte at the end of transaction that needs to be added into the protocol during command implementation. This extra byte ensures the integrity of data exchanged over the I2C bus by adding a CRC8 checksum with a polynomial x8+x2+x+1. The user can enable/disable the PEC.

2.1 PMBusLib folder structure

The following files are included in the PMBusLib package:

- xmc_pmbus_common_conf.h:
  - General configuration of the library
- xmc_pmbus_common.h:
  - Common APIs and configuration structures
- xmc_pmbus_common.c:
  - Array of commands and common APIs implementation
- xmc_pmbus_slave.h:
  - Slave APIs and configuration structures
- xmc_pmbus_slave.c:
  - Slave APIs implementation

2.2 Software flow

Unlike the PMBus™ master, the PMBus™ slave is a passive device. It does not know when the master will issue a command, and cannot predict how the master will behave. Therefore, the operation depends on interrupt events.

Figure 14 shows the software flow of the library main functionality. The so-called “slave task” will periodically move through all of the array of command callbacks (cmd_info[] in “xmc_pmbus_common.c”).

If data is received, the correct command from the array will be executed after a “stop condition” or “slave read request” has been received.

Therefore, the user is required to add the required callback names in the correct position of the array and implement the callbacks in the user application that will be executed in runtime. See section 3 for further details.

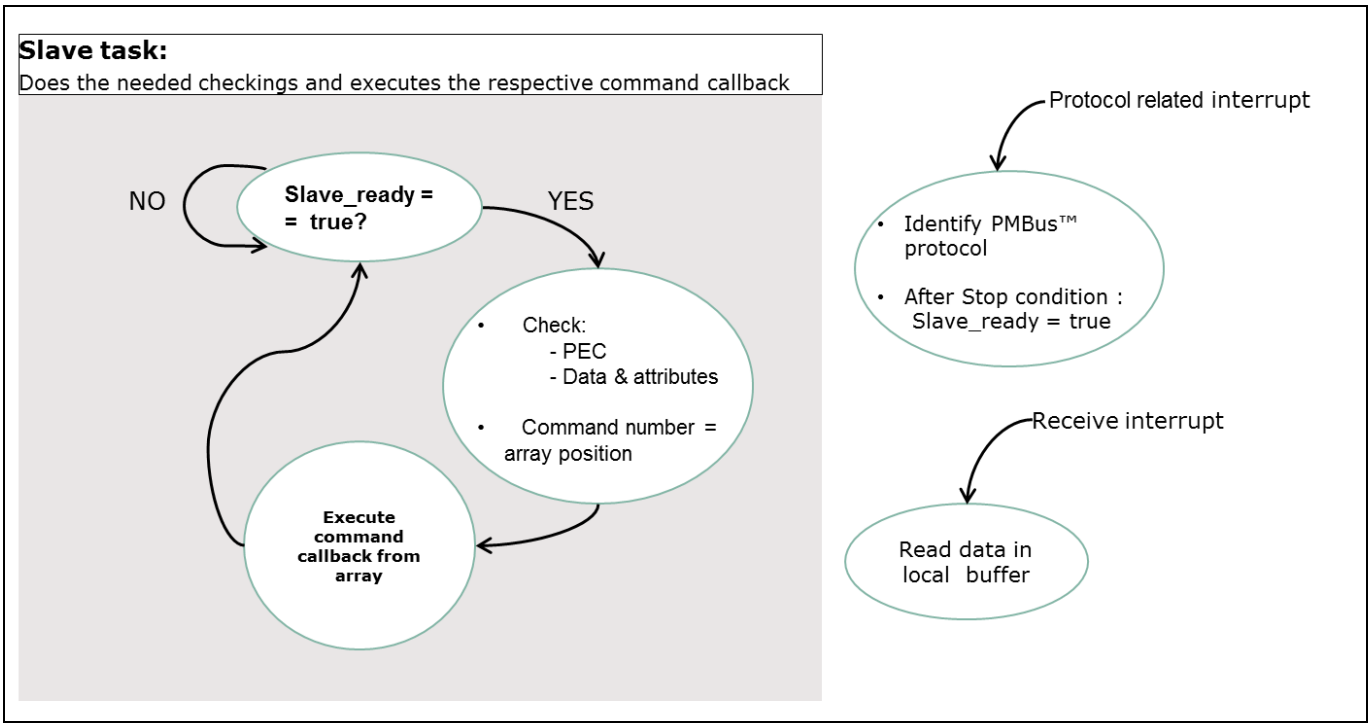

Figure 14 PMBus™ slave device, general software flow

The interrupt events utilized for the operation of the slave are:

- Start condition (SC)
- Repeated start condition (RSC)
- Slave read request (SRR)
- Stop condition (SPC)
- RX event from USIC FIFO module (RXB)

Figure 15 shows the events expected for a PMBus™ message.

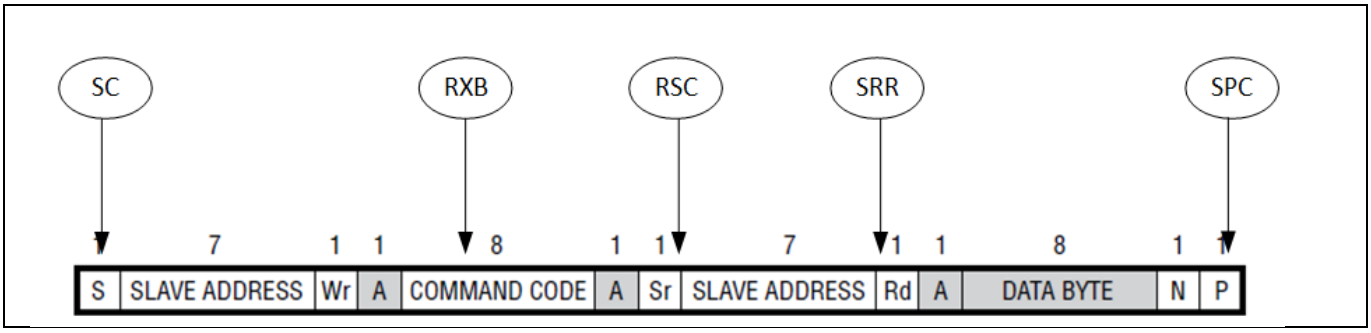

Figure 15 Events expected in PMBusLib for a PMBus™ message

Please note the following:

- Reception of I2C data bytes is performed by using a FIFO buffer and copying to a global buffer for further processing. Data is copied byte by byte to the buffer.
- Once the stop bit event/repeated start event is triggered, the received packet is processed.
- Processing of the complete packet is not part of ISR. Refer to "XMC_PMBUS_SLAVE_Task" API.

- A function pointer array is defined and the command is used as an index into the table to execute specific functional commands, thus reducing the processing time of the received command. This fast execution is crucial for power conversion applications.

Verification and validation of received commands is performed as part of the "XMC_PMBUS_SLAVE_Task" API which checks for PEC, packet size, attribute of commands (R/W), block command and then executes the command if it meets the specification of the PMBus™ command set.

Pre-requisites for any command reception include:

- All state machine variables are cleared and ready to accept a new incoming command.
- It is the responsibility of the application developer to validate the received packet for length and attribute before proceeding to execution of the command.

2.3 Software API functions

2.3.1 PMBusLib APIs

The APIs that can be used for master and slave implementation over XMC™ are defined in “xmc_pmbus_common.h”. See Table 1.

Table 1 “xmc_pmbus_common.h” APIs.

API functions name	Description
bool XMC_PMBUS_ValidateDataLength(uint32_t actual, uint32_t expected)	The function compares if both actual and expected numbers are of same value. Returns true if both are equal, else false.
void XMC_PMBUS_TransmitslaveData(XMC_USIC_CH_t *const channel, uint8_t *data_ptr, uint8_t length)	The function transmits the data bytes sequentially based on its length.
void XMC_PMBUS_TransmitmasterData(XMC_USIC_CH_t *const channel, uint8_t *data_ptr, uint8_t length)	The function transmits the data bytes sequentially based on its length.
uint8_t XMC_PMBUS_PECCalculate(uint8_t *data, uint16_t length)	This function calculates the packet error checking byte using CRC-8.
float XMC_PMBUS_Power(int8_t base, int8_t index)	This function calculates the result for base raised to the power of the index.
float XMC_PMBUS_LinearToFloat(uint16_t linear_data)	Converts linear format data to its equivalent floating point value.
uint16_t XMC_PMBUS_FloatToLinear(float float_data, int8_t exponent)	This function converts the floating-point value into its equivalent linear representation.
float XMC_PMBUS_VOutLinearToFloat(int8_t exponent, uint8_t* mantissa)	This function converts the output voltage in linear format into its equivalent floating-point value.
void XMC_PMBUS_VOutFloatToLinear(float voltage, int8_t exponent, uint8_t* mantissa)	This function converts the output voltage in floating-point to its equivalent linear format.
int16_t XMC_PMBUS_FloatToDirect(float x, int16_t m, int16_t b, int8_t r)	This function converts the floating-point value to its equivalent direct format.
float XMC_PMBUS_DirectToFloat(int16_t y, int16_t m, int16_t b, int8_t r)	This function converts the direct format value into a floating-point value.

API functions name	Description
void XMC_PMBUS_SystickStart(void)	API initializes SYSTEM TIMER. To be called only if Timeout functions are enabled. The user should exercise caution while using this API and ensure the Systick timer is not used for other functionality in the application.
void XMC_PMBUS_SystickStop(void)	API stop Systick timer. To be called only if Timeout functions are enabled. User should exercise caution while using this API and ensure the Systick timer is not used for other functionality in the application.

The APIs that can be used for master and slave implementation over XMC™ are defined in “xmc_pmbus_common.h”. See Table 2.

Table 2 “xmc_pmbus_slave.h” APIs.

API functions name	Description
void XMC_PMBUS_SLAVE_SetSMBALERTHigh(XMC_GPIO_PORT_t *const smbalert_port, const uint8_t smbalert_pin)	Drives SMB Alert port pin HIGH.
void XMC_PMBUS_SLAVE_SetSMBALERTLow(XMC_GPIO_PORT_t *const smbalert_port, const uint8_t smbalert_pin)	Drives SMB Alert port pin LOW.
void XMC_PMBUS_SLAVE_Start(const XMC_PMBUS_NODE_CONFIG_t *const config)	Sets the USIC channel to I2C mode.
void XMC_PMBUS_SLAVE_Stop(const XMC_PMBUS_NODE_CONFIG_t *const config)	Clears the USIC channel from I2C mode.
void XMC_PMBUS_SLAVE_Init(XMC_PMBUS_NODE_t *const slave);	Initializes USIC peripheral for I2C mode and configures SMB alert, Controls IO & ERU (for time out) if the features are enabled.
void XMC_PMBUS_SLAVE_Task(XMC_PMBUS_NODE_t *const slave);	This API executes the response for the command sent by the host. This API can be called in an ISR or periodically / asynchronously as and when user desires.
void XMC_PMBUS_SLAVE_ClearFlagVars(XMC_PMBUS_NODE_t *const slave);	This API clears the dynamic variables and sets the device to idle state.
void XMC_PMBUS_SLAVE_HostNotify(XMC_PMBUS_NODE_t * const slave);	Notifies an error to the PMBus™ host as defined by Host Notify Protocol (HNP).

2.3.2 Interrupt handlers

The table lists the interrupt handlers used by the PMBusLib.

Table 3 Interrupt handlers API

Handler name	Description
XMC_PMBUS_SLAVE_lRxFIFOCallBack()	Interrupt handler for FIFO standard receive buffer interrupt generated by device USIC module FIFO system. The event defined is that data is received from master system.
XMC_PMBUS_SLAVE_lProtocolCallBack()	Interrupt handler for protocol specific interrupt generated by device I^2C system. The event defined is repeated start condition, stop condition and slave read request event detected.

2.4 Packet Error Checking (PEC)

The PEC byte is used to ensure the data reliability of communications. It is calculated using all the data bytes in the communication, including the slave address and the read/write bit. The PEC uses an 8-bit cyclic redundancy check (CRC-8). The polynomial used is $C_{(x)} = x^8 + x^2 + x + 1$.

2.5 Fault reporting mechanism

When faults occur in the system, as well as handling the situation in the slave, the slave will also use a fault reporting mechanism to notify the master. Reporting is via either:

- SMBALERT#
- Host notify protocol

SMBALERT# requires an additional signal line connection between the master and slaves. In addition, the master is required to send a read status command with an ARA (Alert Response Address) to determine which slave changed the alert line and the type of fault that occurred.

Host-notify protocol is used to save on hardware connections, and also to ease reporting procedures.

3 Getting started with PMBusLib

When customizing the software stack for the dedicated application, the user must decide on appropriate adaptations to the code.

3.1.1 Configuring the library

Based on the application requirements, PMBusLib must be configured. This configuration is made by macros in the "xmc_pmbus_common_conf.h" file:

- Chose the available/s channels:
  `#define XMC_PMBUS_NODE_USICx_CHy  XMC_PMBUS_ENABLE`
- Enable the required library features using the XMC_PMBUS_ENABLE macro:
  Timeout, control signal, smbalert and host notify
- Configure the length of the slave buffer and microseconds of timeout if required:
  `#define XMC_PMBUS_MICRO_SECS (900U)`
  `#define XMC_PMBUS_SLAVE_MAX_DATA_LENGTH (0x0FU)`

3.1.2 Setting up a PMBus™ slave

For configuring each slave do the following:

Create XMC_PMBUS_NODE_CONFIG_t structure for the slave configuration.
For example:

```
const XMC_PMBUS_NODE_CONFIG_t slave1_config =
{
  /* Channel of the USIC */
  .channel = XMC_USIC1_CH1,
  /* Pointer to initialization structure I2C protocol */
  .i2c_config = &slave1_i2c_config,
  /* Protocol interrupt config*/
  .protocol_irq_sr = (uint8_t)XMC_PMBUS_USIC1_CH1_PROTOCOL_IRQ_SR,
  .protocol_irq_nvic_node = (IRQn_Type)XMC_PMBUS_USIC1_CH1_PROTOCOL_IRQ_NVIC_NODE,
  .protocol_irq_prio = (uint32_t) XMC_PMBUS_USIC1_CH1_PROTOCOL_IRQ_PRIO,
  /* RX interrupt */
  .rx_fifo_irq_sr = (uint8_t) XMC_PMBUS_USIC1_CH1_RX_FIFO_IRQ_SR,
  .rx_fifo_irq_nvic_node = (IRQn_Type) XMC_PMBUS_USIC1_CH1_RX_FIFO_IRQ_NVIC_NODE,
  .rx_fifo_irq_prio = (uint32_t) XMC_PMBUS_USIC1_CH1_RX_FIFO_IRQ_PRIO,
  /* SCL line config structure */
  .scl_pin_config = &pmbus_scl,
  /*SCL port and pin selection */
  .scl_port = XMC_GPIO_PORT0,
  .scl_pin = 10U,
  /*SCL Input multiplexer selection*/
  .scl_source = (uint8_t)USIC1_C1_DX1_P0_10,
```

```
  /* SDA line config structure */
  .sda_pin_config = &pmbus_sda,
  /*SDA port and pin selection */
  .sda_port = XMC_GPIO_PORT4,
  .sda_pin = 2U,
  /*SCL Input multiplexer selection*/
  .sda_source = (uint8_t)USIC1_C1_DX0_P4_2,
  /* CAPABILITY */
  .capability.pec = XMC_PMBUS_DISABLE,
  .capability.smb_alert = XMC_PMBUS_DISABLE,
  .capability.max_bus_speed = XMC_PMBUS_SPEED_400KHZ,
  /* Control signal and timeout disables*/
  .control_io_enable = XMC_PMBUS_DISABLE,
  .timeout_enable = XMC_PMBUS_DISABLE,
  /* Pointer to the array for received data*/
  .data_ptr = slave1_data,
};
```

Create XMC_PMBUS_NODE_t structure with a pointer to the configuration structure.

For example:

```
  XMC_PMBUS_NODE_t slave1 =
  {
     .config_ptr = &slave_config,
  };
```

3.1.3 Adding a command

In “xmc_pmbus_common.c” file, add the callbacks for the required commands to the cmd_info[] array .

The position where the callbacks are placed in the array needs to correspond with the command code. This number is described in the array comments. The extern declarations of the callback also need to be defined in “xmc_pmbus_common.c”.

For example:

```
/* Extern declarations */
extern XMC_PMBUS_STATUS_t XMC_PMBUS_NODE_CmdCapability(XMC_PMBUS_NODE_t *const node);
extern XMC_PMBUS_STATUS_t XMC_PMBUS_NODE_CmdStatusByte(XMC_PMBUS_NODE_t *const node);
extern XMC_PMBUS_STATUS_t XMC_PMBUS_NODE_CmdStatusWord(XMC_PMBUS_NODE_t *const node);
          .
          .
          .
/* Callback functions added to the callback command array */
  const XMC_PMBUS_NODE_CMD_INFO_t cmd_info[] =
  {
```

```
                .
                .
                .
    {.call_back = XMC_PMBUS_NODE_CmdCapability, .no_of_data_bytes = 1U,    .attribute
= (uint16_t) (XMC_PMBUS_PROTOCOL_RD_BYTE)}, /* Command Code - 19h; Command Name =
CAPABILITY*/
                .
                .
                .
     {.call_back = XMC_PMBUS_NODE_CmdStatusByte, .no_of_data_bytes = 1U, .attribute =
(uint16_t) (XMC_PMBUS_PROTOCOL_WR_RD_BYTE)}, /* Command Code - 78h; Command Name =
STATUS_BYTE*/
     {.call_back = XMC_PMBUS_NODE_CmdStatusWord, .no_of_data_bytes = 2U, .attribute =
(uint16_t) (XMC_PMBUS_PROTOCOL_WR_RD_WORD)}, /* Command Code - 79h; Command Name =
STATUS_WORD*/
                .
                .
                .
  }
```

Once the commands are added to the commands array and the prototypes of the callbacks are declared as extern to be visible across all of the application code, the user can implement the callback in the application code. For example, the main.c:

```
/* Capability command callback. It sends back the capabilities of the device*/
XMC_PMBUS_STATUS_t XMC_PMBUS_NODE_CmdCapability(XMC_PMBUS_NODE_t *const node)
{
  uint8_t data;
  XMC_PMBUS_STATUS_t status = XMC_PMBUS_STATUS_ERROR;
  if(node->comm_type == XMC_PMBUS_COMM_TYPE_READ)
  {
    data = node->config_ptr->capability.capability_reg;
    XMC_I2C_CH_SlaveTransmit(node->config_ptr->channel, data);
    XMC_PMBUS_STATUS_t status = XMC_PMBUS_STATUS_SUCCESS;
  }
  return (status);
}
```

3.1.4 Runtime handling

For initializing the PMBus™ slave, “XMC_PMBUS_SLAVE_Init();” needs to be called during the system initialization inside main() function.

Getting started with PMBusLib

"XMC_PMBUS_Task()" will go thru all the positions in the array of commands and will trigger the execution of the command received. Therefore, this API needs to be called periodically. For example, inside a while(1) loop in the main(), in a periodic timer interrupt or in an operative system task. See following example:

```
int main(void)
{
  /* Init API PMBUS */
  XMC_PMBUS_SLAVE_Init(&slave1);
        .
        .
        .
  while(1U)
  {
    /* slave1 task */
       XMC_PMBUS_SLAVE_Task(&slave1);
  }
 }
```

Revision history

Major changes since the last revision

Page or reference	Description of change
2016-09-09	First version

Edition 2016-09-09

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email: erratum@infineon.com

Document reference
AN_201609_PL30_029

IMPORTANT NOTICE

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (**www.infineon.com**).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.