C28x PMBus Communications Stack

USER'S GUIDE

Copyright

Texas Instruments
13905 University Boulevard
Sugar Land, TX 77479
http://www.ti.com/c2000

Revision Information

This is version 1.03.00.00 of this document, last updated on Fri Feb 12 19:17:24 IST 2021.

Table of Contents

1 Introduction

The PMBus (Power Management Bus) Communications Software Stack supports PMBus slave operation on C2000 devices. The current version of the library supports only slave mode on the following devices:

1. F28004x
2. F2838x
3. F28002x

The PMBus slave communication stack is based on **PMBus specification (Part I, II) v1.2** and supports the following:

- PMBus Specification v1.2 Bus Speeds
  - Standard - 100kHz
  - Fast - 400kHz
- SMBus Specification v2.0 Transactions
  1. Quick Command
  2. Send Byte
  3. Receive Byte
  4. Write Byte/Word
  5. Read Byte/Word
  6. Process Call
  7. Block Write/Read (Blocks support up to 255 data bytes)
  8. Block Write/Block Read/Process Call (Blocks support up to 255 data bytes)
- SMBus Specification v2.0 Alert Response (Alert line)
- PMBus Specification v1.2 (Extentions to SMBus v2.0) Features
  - Group Command
  - Extended Read Byte/Word Command

Known deviations and/or unsupported features defined in the PMBus specification v1.2:

- Extended Write Byte/Word Command: Supported only as defined in PMBus specification v1.0
- Address Resolution Protocol (ARP): Not supported, as this is an optional feature
- Host Notify Protocol: Not supported, as this isn't required when Alert Line is supported

1.1 Chapter Overview

Chapter 2 provides resource and forum links.

Chapter 3 describes the directory structure of the library.

Chapter 4 provides step-by-step instructions on how to integrate the library into a project.

Chapter 5 details about the different aspects of the protocol that are supported by this library, including the different transaction types.

Chapter 6 describes the programming interface, structures and routines available for this stack.

Chapter 7 details the master and example setup code and test functions.

Chapter 8 describes the other PMBus examples that aren't using the communication stack code.

Chapter 9 provides a revision history of the library.

1.2 Legacy PMBus Library Compatibility

For C2000 PMBus communication library users that developed with the *F28004x* library available in *C2000Ware 2.00.00.03* and earlier, a compatibility header has been provided to map the legacy API names to the latest library API names.

Located at:

```
~\PMBus\c28\include\pmbus_stack_compatibility.h
```

Within your PMBus application, switch the legacy PMBus library for the latest PMBus library and include the header listed above within your main application code.

2 Other Resources

The user can get answers to F28002x, F28004x, or F2838x frequently asked questions(FAQ) from the TI Resource Explorer.
https://dev.ti.com/tirex

Check out the TI C2000 page: http://www.ti.com/c2000

TI community forums website: http://e2e.ti.com

Building the PMBus Communications Stack library and examples requires **Codegen Tools v20.2.1.LTS or later**.

3 Library Structure

The PMBus Communications Stack Library is distributed as part of the C2000Ware software framework. The library, source code and examples are packaged under:

```
~\C2000Ware_X_XX_XX_XX\libraries\communications\PMBus
```

Figure. 3.1 shows the directory structure, while the subsequent table 3.1 provides a description of each folder.

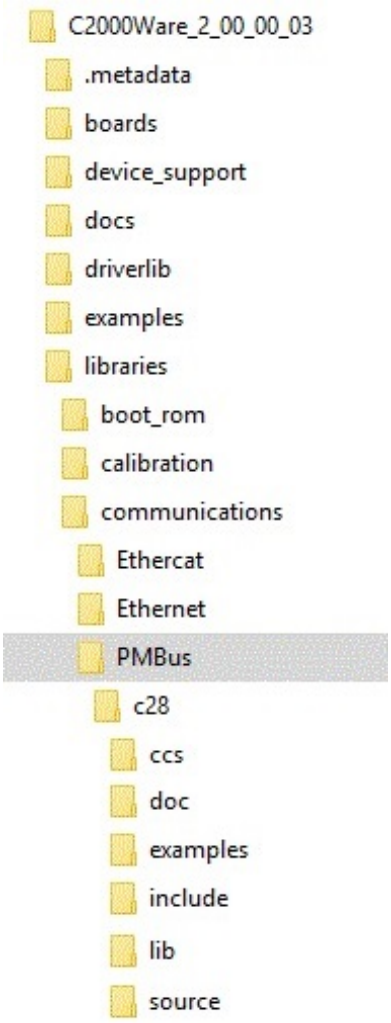

Figure 3.1: Directory Structure of the Library

Folder	Description
<base>	Base install directory. By default this is C:/ti/c2000/C2000Ware_X_XX_XX_XX/libraries/communications/PMBus For the rest of this document <base> will be omitted from the directory names.
<base>/ccs	Project files for the library. Allows the user to reconfigure, modify and re-build the library to suit their particular needs.
<base>/docs	Documentation for the current revision of the library including revision history.
<base>/examples	Examples that illustrate use of the library. These examples were built for the F28002x, F28004x, and F2838x devices using the CCS 10.1.0.00010 IDE. Additionally, non-stack PMBus examples are located here.
<base>/include	Header files for the library. These include function prototypes and structure definitions.
<base>/lib	Pre-built binaries for the library.
<base>/source	Source files for the library.

Table 3.1: Library Directory Structure Description

4 Using the PMBus Communications Stack Library

The source code and project(s) for the library are provided. The user may import the library project(s) into CCSv9 (or later) and be able to view and modify the source code for all routines and lookup tables (see Fig. 4.1)

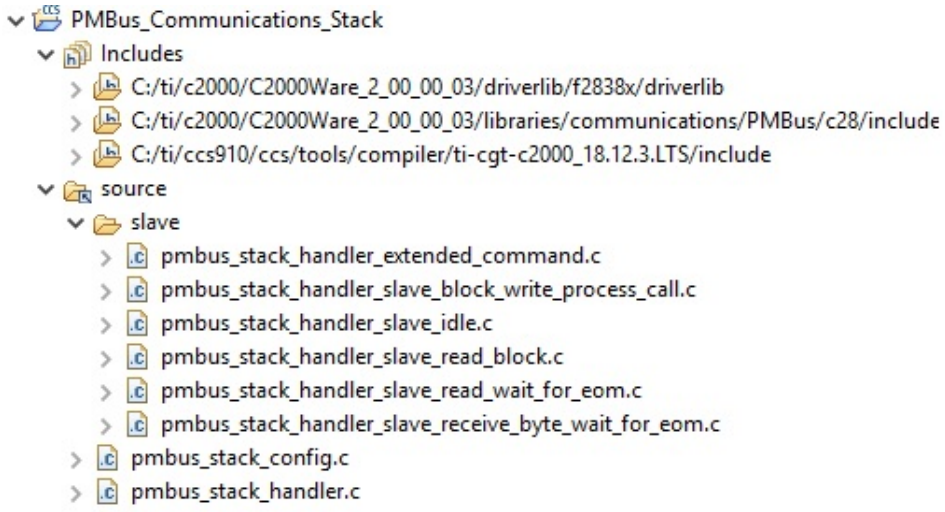

Figure 4.1: F2838x Library Project View

4.1 Integrating the Library into your Project

To begin integrating the library into your project follow these steps:

1. Go to the **Project Properties** and add a path to the device driverlib directory, to the *Include Options* section of the project properties (Fig. 4.2). This option tells the compiler where to find the device driver header files. In addition, you must add a path for the PMBus stack interface, for the compiler to find the stack header files.

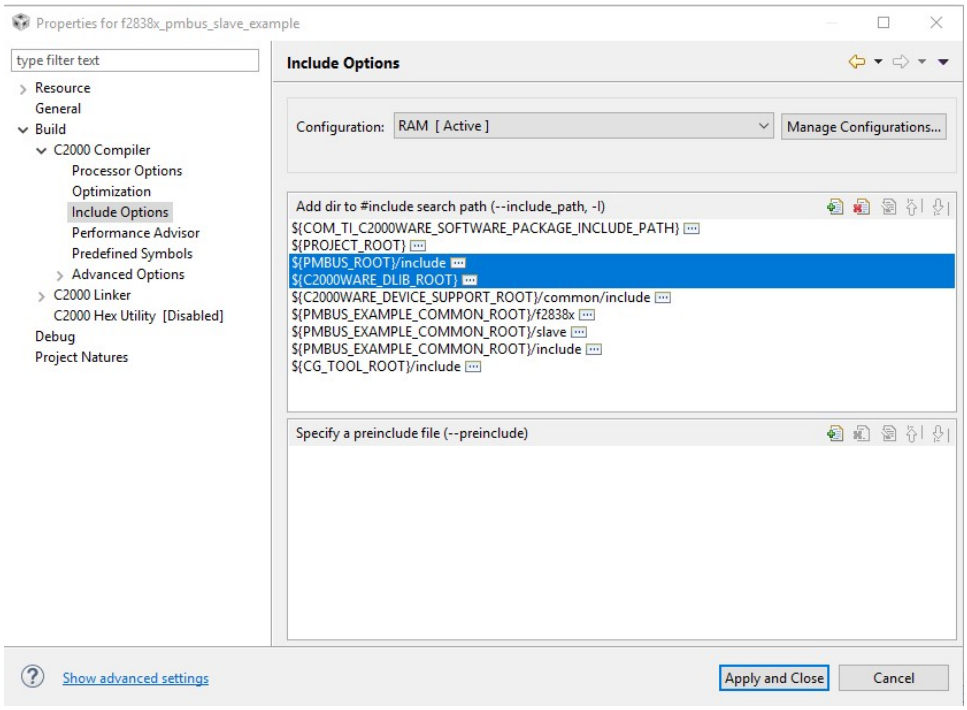

Figure 4.2: Adding the Library Header Path to the Include Options

2. Add the Communications Stack library (replace "device" with F28002x, F28004x, or F2838x), **'<device>_PMBus_Communications_Stack.lib'** to the project explorer as shown in Fig. 4.3. Also add the device driver library.

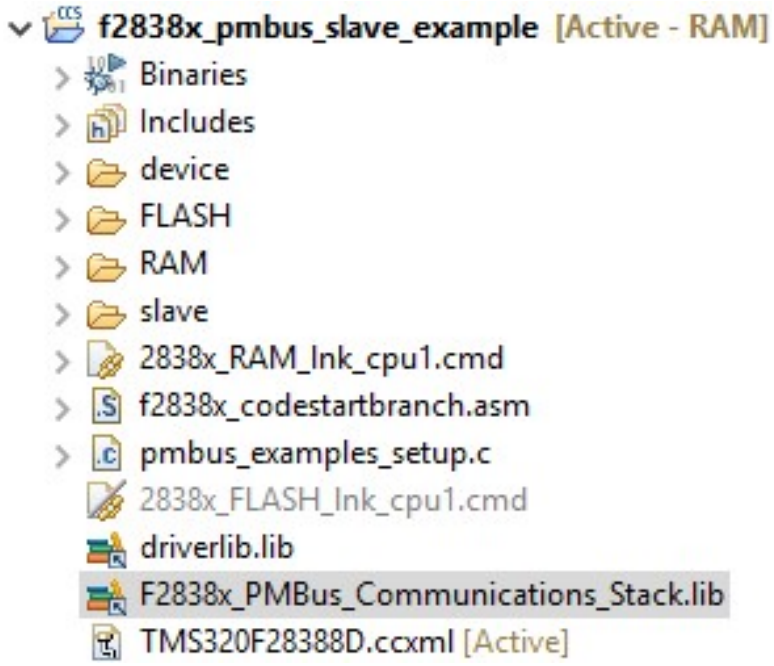

Figure 4.3: Adding the library and location to the file search path

3. For the slave device, define **_PMBUS_SLAVE** in the *Predefined Symbols* option under the C2000 Compiler menu, Fig. 4.4. If using F2838x, **CPU1** must also be defined.
   NOTE: FOR DEVICES THAT OPERATE IN MASTER MODE THE USER MUST DEFINE THE SYMBOL **_PMBUS_MASTER** INSTEAD.

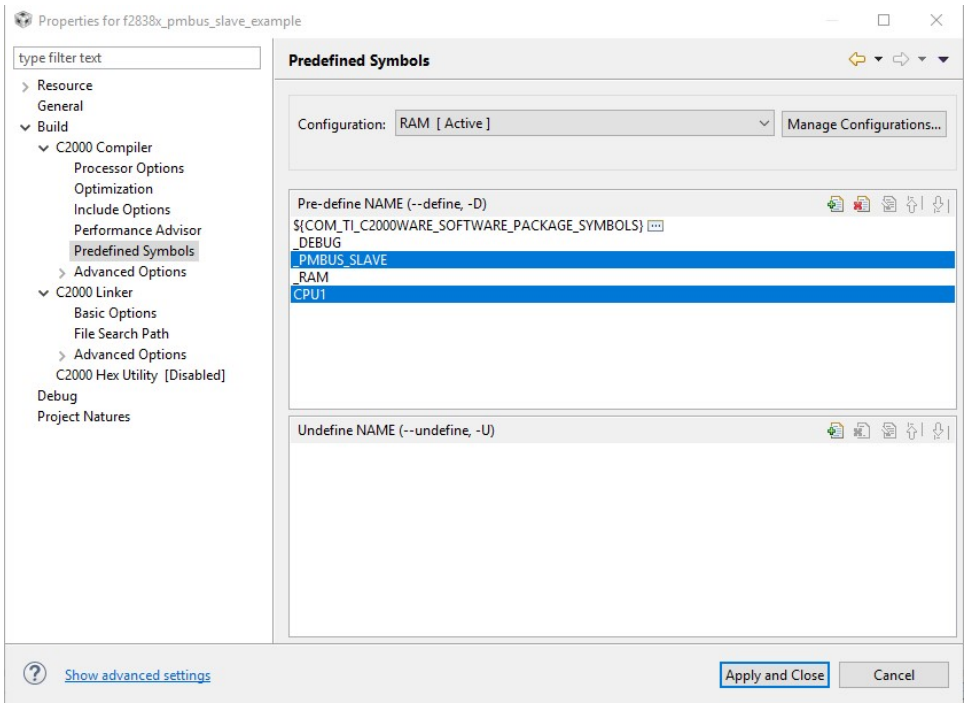

Figure 4.4: Compiler Predefined Symbols

5 The PMBus Protocol

5.1 PMBus Slave Mode

In slave mode, the device only responds to messages from the Master. The slave cannot initiate messages, with the exception of asserting the alert line to notify the master of a fault in the system.

The slave mode state machine is handled through PMBus Module interrupts. The Interrupt Service Routine invokes the state machine handler, which deciphers the transaction initiated by the master, and takes appropriate action depending on the transaction type.

Transaction Types Include:

- Quick Command
- Send Byte
- Receive Byte
- Write Byte/Word
- Read Byte/Word
- Process Call
- Block Write/Read (Blocks support up to 255 data bytes)
- Block Write/Block Read/Process Call (Blocks support up to 255 data bytes)
- Group Commands
- Extended Commands

5.1.1 Packet Error Checking

The PMBus Module has the option to work with, or without, packet error checking. While the option is available in the hardware, the software stack assumes that error checking is enabled. All transactions, with the exception of the quick command, must have a trailing **Packet Error Check (PEC)** value associated with it; this feature lends a measure of robustness to the communications.

In the event of an invalid PEC, the state machine will abort its current processing and revert to its idle state (or issue a debugging halt if the emulator is connected), while a NACK is issued on the bus; the decision to either halt or retransmit lies with the master.

5.2 Slave Mode Message Types

This section describes the different transaction (message types) that are recognized, and supported by the slave state machine handler. During initialization, the slave handler is setup to automatically acknowledge up to 4 bytes, with the final byte requiring a manual acknowledgment, and to verify the PEC received is correct.

The primary handler is always called in the interrupt service routine of the following interrupt sources

- DATA_READY
- EOM
- DATA_REQUEST

Each of these is a bit-field in the PMBus status register; in addition to these, the RD_BYTE_COUNT is also queried by the state machine handler. All state transitions occur based on the value of these bit fields at the invocation of the state machine.

The following abbreviations are used in the descriptions of the transactions,

S
: The start signal on the bus

ADDR
: The address of the slave device

Rd/R
: The read bit asserted after the slave address is put on the bus

Wr/W
: The write bit asserted after the slave address is put on the bus

A
: Acknowledgment

NA
: NACK or No Acknowledgment

P
: The stop signal on the bus

Sr
: Repeated Start

PEC
: Packet Error Check byte

Each transaction (message) description will have an image of the message format; Fig. 5.1 describes the convention used,

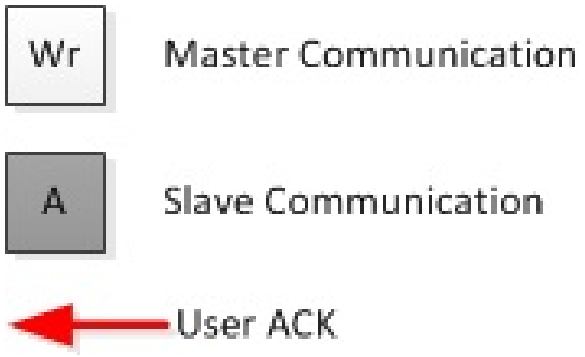

Figure 5.1: Message Format Legend

5.2.1 Quick Command

When a Quick Command is received, the **EOM (End-of-Message)** status bits is set, and the **RD_BYTE_COUNT (Received Byte Count)** field is 0.

The Slave manually ACKs the transaction by writing to the PMBACK register.

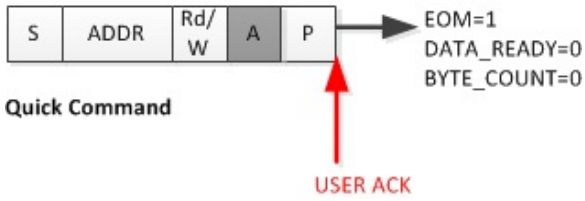

Figure 5.2: Quick Command

5.2.2 Send Byte

When a Send Byte is received, the **DATA_READY** and **EOM (End-of-Message)** status bits are set, and the **RD_BYTE_COUNT (Received Byte Count)** field is 2, indicating two bytes were received, the data byte and the PEC.

The Slave reads the data and manually ACKs the message by writing to the PMBACK register.

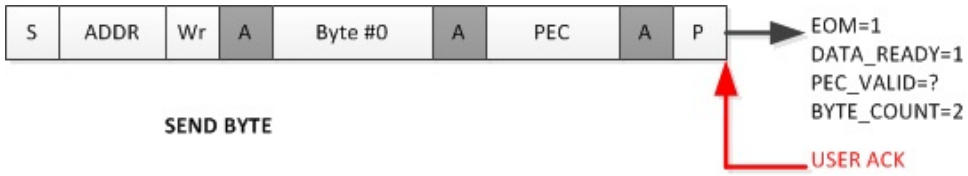

Figure 5.3: Send Byte

5.2.3 Write Byte

The Write Byte is identical to Send Byte, with the exception that **RD_BYTE_COUNT (Received Byte Count)** is now 3, that is, a command byte, a data byte and the PEC.

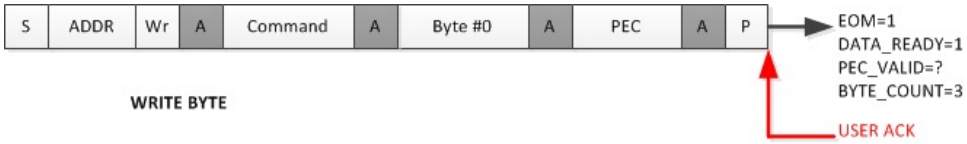

Figure 5.4: Write Byte

5.2.4 Write Word

The Write Byte is identical to Send Byte, with the exception that **RD_BYTE_COUNT (Received Byte Count)** is 4, that is, a command byte, 2 data bytes and the PEC.

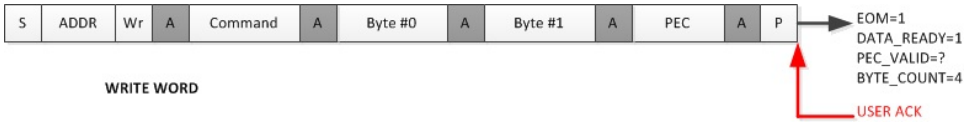

Figure 5.5: Write Word

5.2.5 Block Write

The Block Write is issued when the master has to transfer more than 2 data bytes (up to a maximum of 255 bytes). The master will transmit a command, a count (how many bytes it intends to send), followed by the bytes, ending with the PEC.

For every 4 bytes the slave receives, **DATA_READY** is asserted and **RD_BYTE_COUNT** is 4; no End-of-Message (EOM) is received at this point. The slave must read the receive buffer, and manually ACK reception of 4 bytes before the master can proceed sending the next 4 bytes. On the very last transmission **DATA_READY** and **EOM** are asserted indicating the end of transmission. The slave must read the receive buffer (which has 1 to 4 bytes depending on the initial count) and manually ACK the transaction.

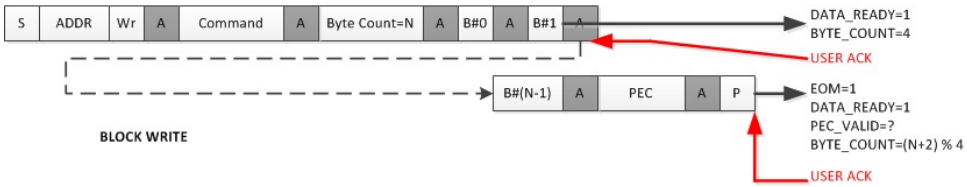

Figure 5.6: Block Write

5.2.6 Receive Byte

The master initiates a Receive Byte by putting the slave's address on the bus followed by a read bit. The slave will automatically ACK its address, load its transmit buffer, and transmit a byte and its PEC.

If there is no error in the transmission the master will **NACK** the PEC indicating the end of the transaction. Both the **NACK** and **EOM** status bits are asserted at this point.

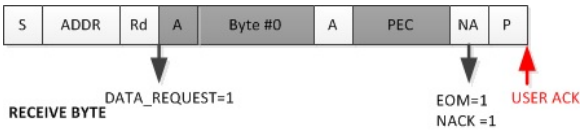

Figure 5.7: Receive Byte

5.2.7 Alert Response

A special variant of the Receive Byte is the **Alert Response** transactions where the slave device pulls the **ALERT** line low; the master must respond with the **ALERT RESPONSE ADDRESS** and a read, the alerting slave will respond by transmitting its own address as shown in Fig. 5.8.

When the master puts the Alert Response Address on the line with a read, the alerting slave hardware will automatically respond with its address, without software intervention.
NOTE: THE 7 BIT DEVICE ADDRESS PROVIDED BY THE SLAVE TRANSMIT DEVICE IS PLACED IN THE 7 MOST SIGNIFICANT BITS OF THE BYTE. THE EIGHTH BIT CAN BE A ZERO OR ONE.

Figure 5.8: Alert Responsed

5.2.8 Read Byte

The master initiates a Read Byte by putting the slave's address on the bus followed by a write bit. The master issues a command - a Read Byte command - followed by a repeated start, with the slave address and the read bit. When the repeated start is issued on the bus the **DATA_READY** bit is asserted at the slave end, with a **RD_BYTE_COUNT** of 1. At the read bit the **DATA_REQUEST** bit is asserted; the slave responds by transmitting a single byte followed by the PEC. If there is no error in the transmission the master will **NACK** the PEC indicating the end of the transaction. Both the **NACK** and **EOM** status bits are asserted at this point.

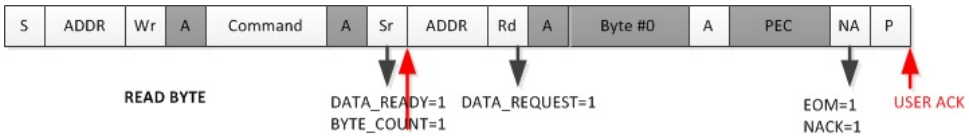

Figure 5.9: Read Byte

5.2.9 Read Word

Read Word is similar to Read Byte with the exception that the slave responds to the repeated start (read bit) by transmitting two bytes followed by the PEC.

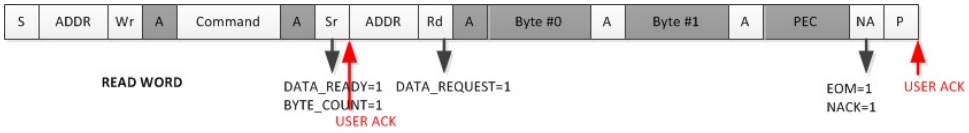

Figure 5.10: Read Word

5.2.10 Block Read

If the master transmits a Block Read command, the slave responds by sending more than 2 bytes (up to a maximum of 255 bytes). The transaction, including the status bit assertions, is similar to the read word command. The first byte sent by the slave is always the byte count, that is, the number of bytes it intends to transmit. It then follows this with the data bytes. For every 4 bytes sent by the slave (and acknowledged by the master) the **DATA_REQUEST** bit is asserted requesting the slave to send the next set of bytes. The transaction is terminated by the master by issuing a **NACK** on the bus; both the **NACK** and **EOM** status bits are asserted at the slave end at this point.

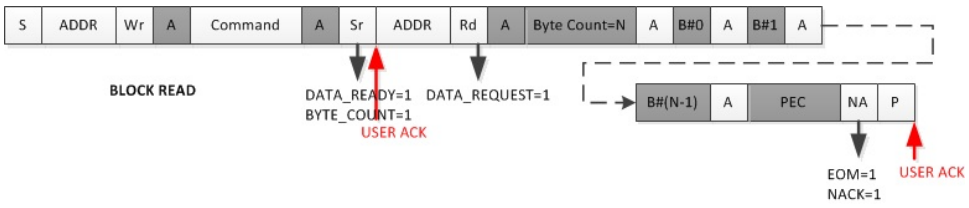

Figure 5.11: Block Read

5.2.11 Process Call

This is basically a write word followed by a read word without the read word command field and the write word STOP bit. A repeated START seperates the write and the read transactions.

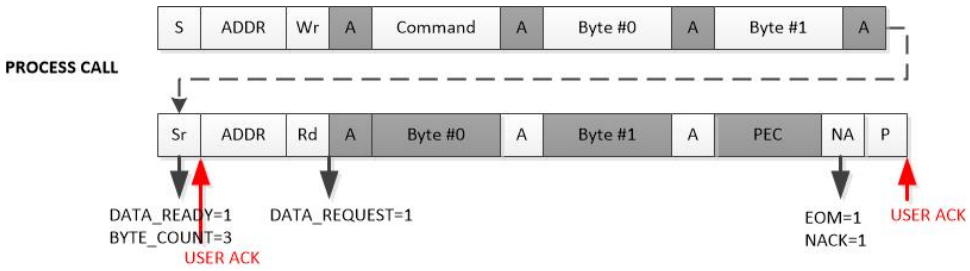

Figure 5.12: Process Call

5.2.12 Block Write/ Read/ Process Call

This is basically a block write followed by a block read. The key points to note here are the byte counts on the block write, and block read must be the same, and a single PEC is sent at the end of the block read.

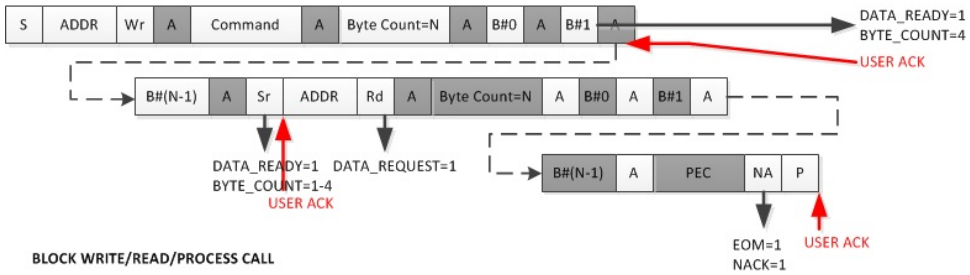

Figure 5.13: Block Write/ Read/ Process Call

5.2.13 Group Command

The Master writes to a group of slaves in a single transaction. It does this by putting each slave's address (with a write) followed by a command, two bytes, and a PEC on the bus after a repeated start (the exception is the first slave address which follows the start). A slave device will acknowledge its address on the bus, and its state machine will respond when the **DATA_READY** is asserted on the next repeated start (or on a stop, if the slave in question is the last to be addressed).

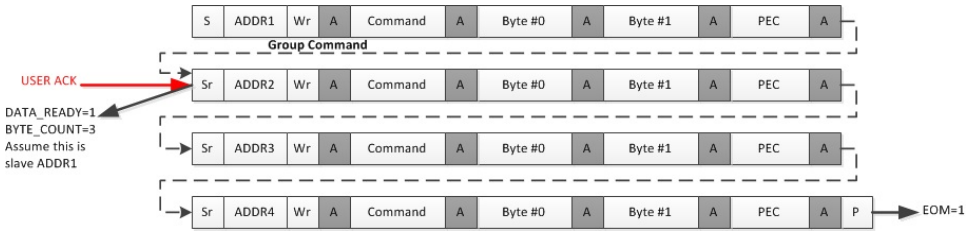

Figure 5.14: Group Command

5.2.14 Extended Command

The extended commands, following PMBus specification v1.2, is supported for two transaction types

1. Extended Read Byte
2. Extended Read Word

The extended commands, following PMBus specification v1.0, is supported for two transaction types

1. Extended Write Byte
2. Extended Write Word

NOTE: THE EXTENDED WRITES CONFORMING TO PMBUS V1.0 INCLUDE A REPEATED START (AND SLAVE ADDRESS AFTER THE EXTENSION AND COMMAND BYTES ARE SENT). THE STACK IS IMPLEMENTED FOR PMBUS V1.0 AND ISN'T COMPATIBLE WITH EXTENDED WRITES PMBUS V1.2

These commands are similar to their non-extended counterparts, with the exception that the command is preceded by the extension byte (0xFE or 0xFF). The master issues a repeated start with the slave address and the read (for a read transaction) or write (for a write transaction) bit asserted.

At this point the slave sees the **DATA_READY** signal asserted and a **RD_BYTE_COUNT** of 2 - it must check the first byte for the extension code before acknowledging. If the transaction is a write the master proceeds; an extended write byte involves 4 bytes: the extension code, the command byte, a data byte, and finally the PEC whereas a write word transaction involves an additional data byte making the total 5 bytes. If the transaction is a read, the slave must transfer 1 (read byte) or 2 (read words) bytes depending on the command received, followed by the PEC.
NOTE: THE PEC IS CALCULATED ON THE SLAVE ADDRESS (WITH WRITE BIT ASSERTED), THE EXTENSION, COMMAND BYTE, SECOND SLAVE ADDRESS (AND EITHER READ/WRITE BIT DEPENDING ON THE TRANSACTION), AND FINALLY THE DATA BYTE(S).

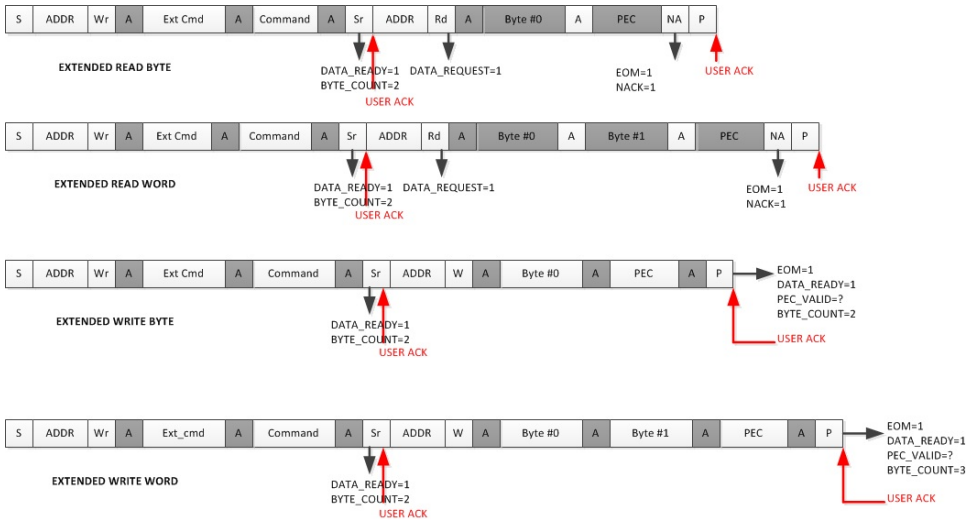

Figure 5.15: Extended Commands (Reads - v1.2, Writes - v1.0)

5.3 State Machine Description

This section describes the state machine. There are currently 6 states, each having their own function (sub-handler). They are:

PMBUS_STACK_STATE_IDLE
The idle state - the handler will enter this state on the first interrupt after power up; the handler will spend most of its time in this state. The sub-handler for this state is *PMBusStack_slaveIdleStateHandler*.

PMBUS_STACK_STATE_RECEIVE_BYTE_WAIT_FOR_EOM
This is a special state to handle a **Receive Byte** command from the master. The sub-handler for this state is *PMBusStack_slaveReceiveByteWaitForEOMStateHandler*.

PMBUS_STACK_STATE_READ_BLOCK
The handler enters this state when it establishes a read block command was issued by the master. The sub-handler for this state is *PMBusStack_slaveReadBlockStateHandler*.

PMBUS_STACK_STATE_READ_WAIT_FOR_EOM
Once the handler establishes that a read command was issued by the master, it transitions to this state awaiting an **End Of Message (EOM)** signal from the master to terminate communications. The sub-handler for this state is *PMBusStack_slaveReadWaitForEOMStateHandler*.

PMBUS_STACK_STATE_BLOCK_WRITE_OR_PROCESS_CALL
When a master issues either a **Block Write** or **Process Call** the state machine transitions to this state. The sub-handler for this state is *PMBusStack_slaveBlockWriteOrProcessCallStateHandler*.

PMBUS_STACK_STATE_EXTENDED_COMMAND
When a master issues either an **Extended Read/Write Byte/Word** the state machine transitions to this state. The sub-handler for this state is *PMBusStack_slaveExtendedCommandStateHandler*.

The transition from one state to the next is depicted in Fig. 5.16,

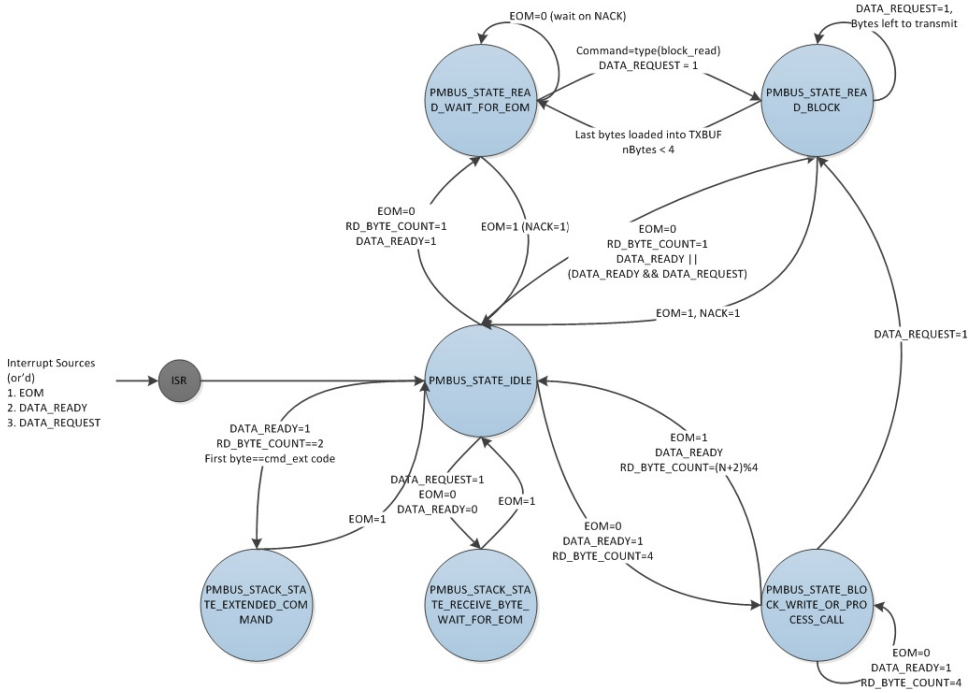

Figure 5.16: Slave Mode State Machine

5.3.1 The Idle State

This is the very first state the state machine enters after a PMBus interrupt is received (and the ISR calls the main state machine handler). The processor tries to decipher the transaction (message) type received from the master, and will either, read the contents of the receive buffer in the event of write transaction, or setup the hardware to transmit and change state accordingly

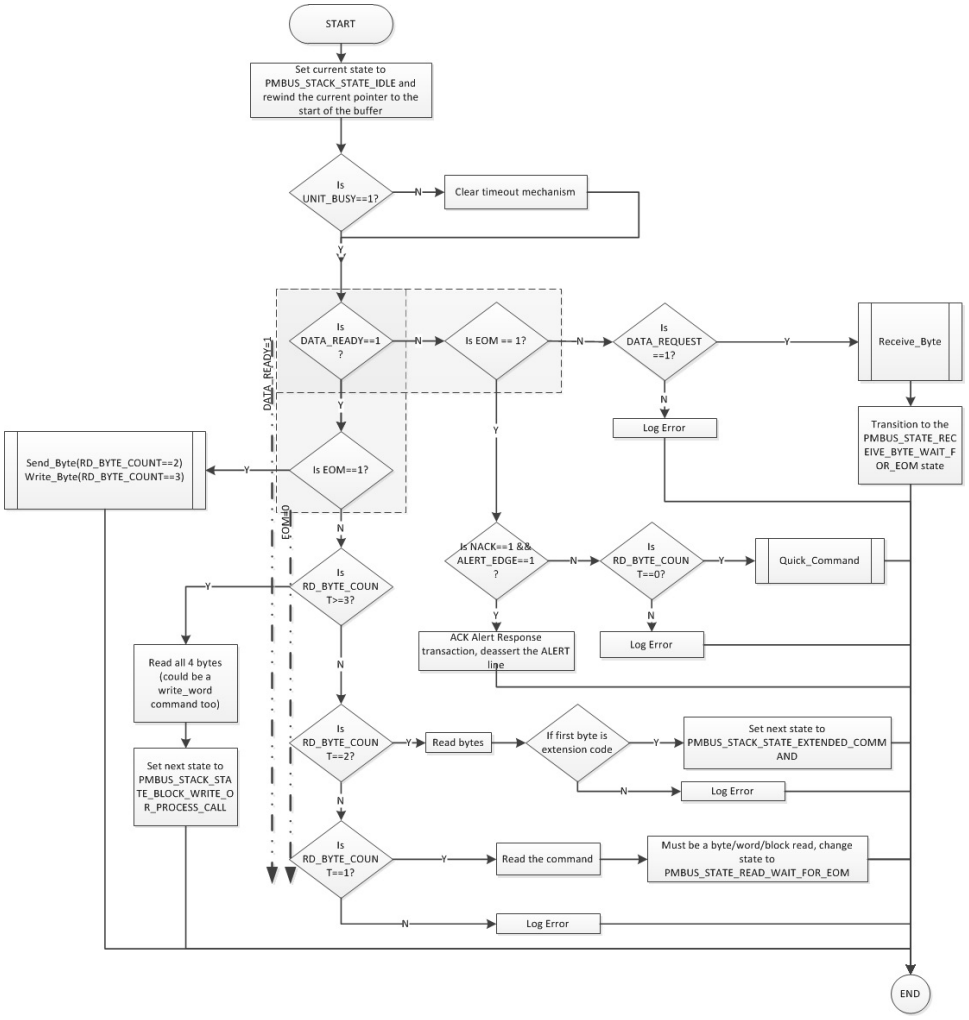

Figure 5.17: The Idle State

5.3.2 The Receive Byte and Wait for End-of-Message State

This is a special state that handles a **Receive Byte** transaction. The state machine transitions from the idle state when it sees the **DATA_REQUEST** bit asserted, with bits EOM or DATA_READY set to 0. In this state, the slave waits for the EOM signal from the master; if any other conditions are set, it is a fault condition and the handler must log the fault and revert to the idle condition

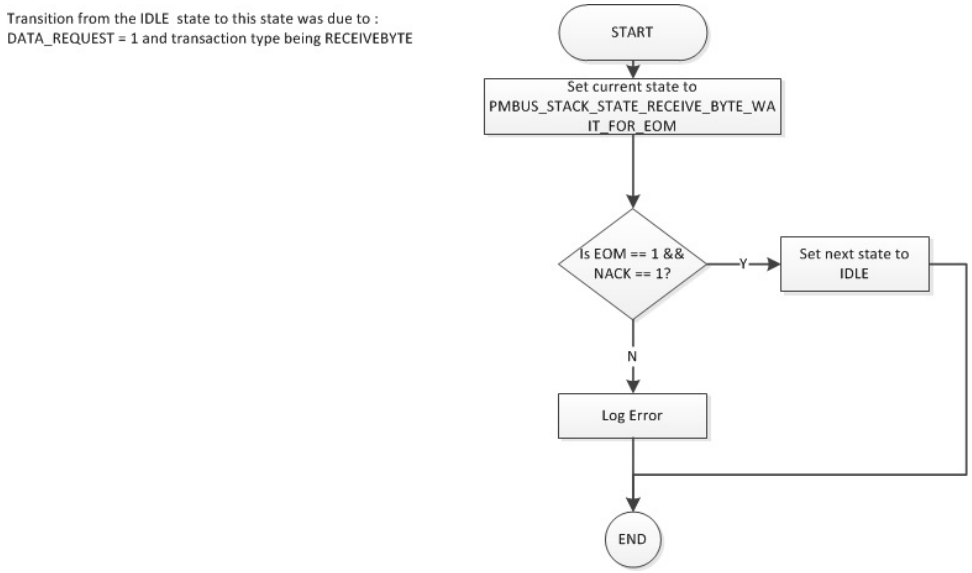

Figure 5.18: The Receive Byte and Wait for End-of-Message State

5.3.3 The Read Block State

The state machine transitions into the Read Block state (from the idle state) once it determines that the current transaction type (command) is a Read Block request from the master. The master follows up with a repeated start and the slave's address followed by the read bit; at this point the **DATA_REQUEST** status bit is asserted at the slave end, and its state machine calls the Read Block sub-handler. The slave continues to remain in this state until it transmits all its data to the master. The master terminates the transaction by issuing a NACK on the bus line.

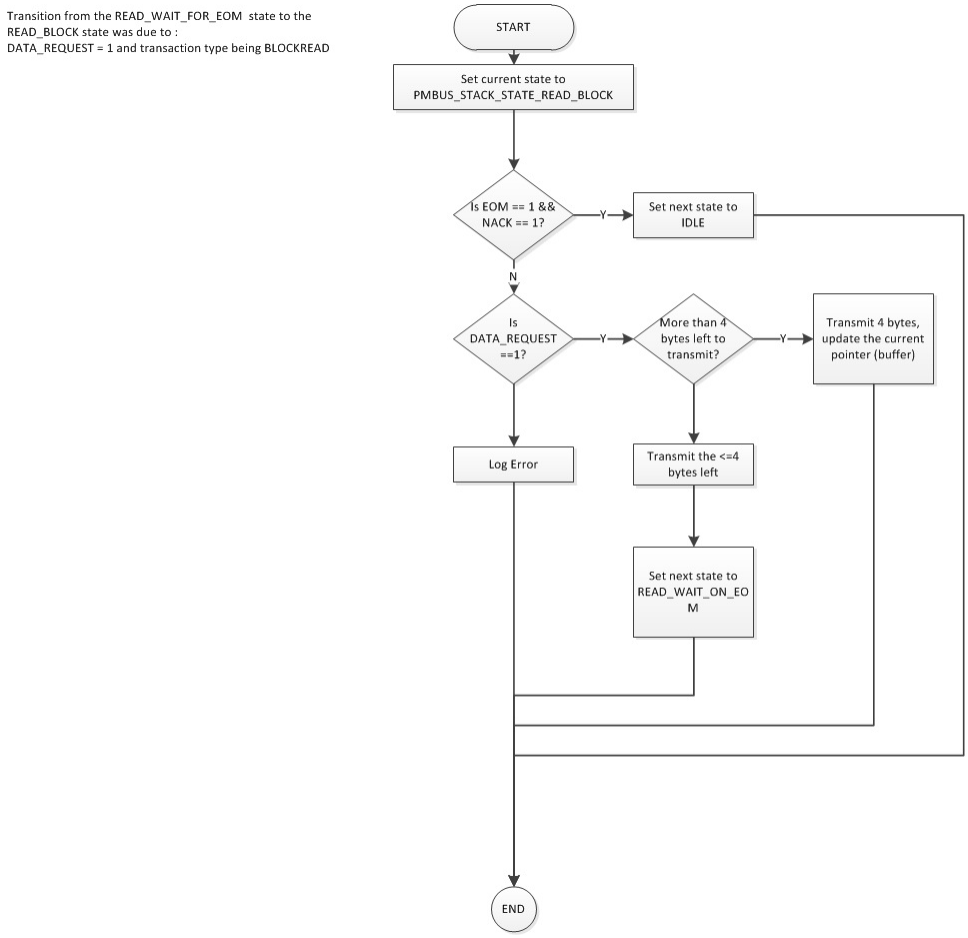

Figure 5.19: The Read Block State

5.3.4 The Read and Wait for End-of-Message State

The state machine transitions to this state from two other states, the idle state when **RD_BYTE_COUNT** and **DATA_READY** are set to 1 in the status register, or from the Read Block state when all but the last (less than or equal to 4) bytes are pending transmission. The state machine lingers in this state till the master issues a NACK on the line (**EOM = 1**) terminating the read transaction.

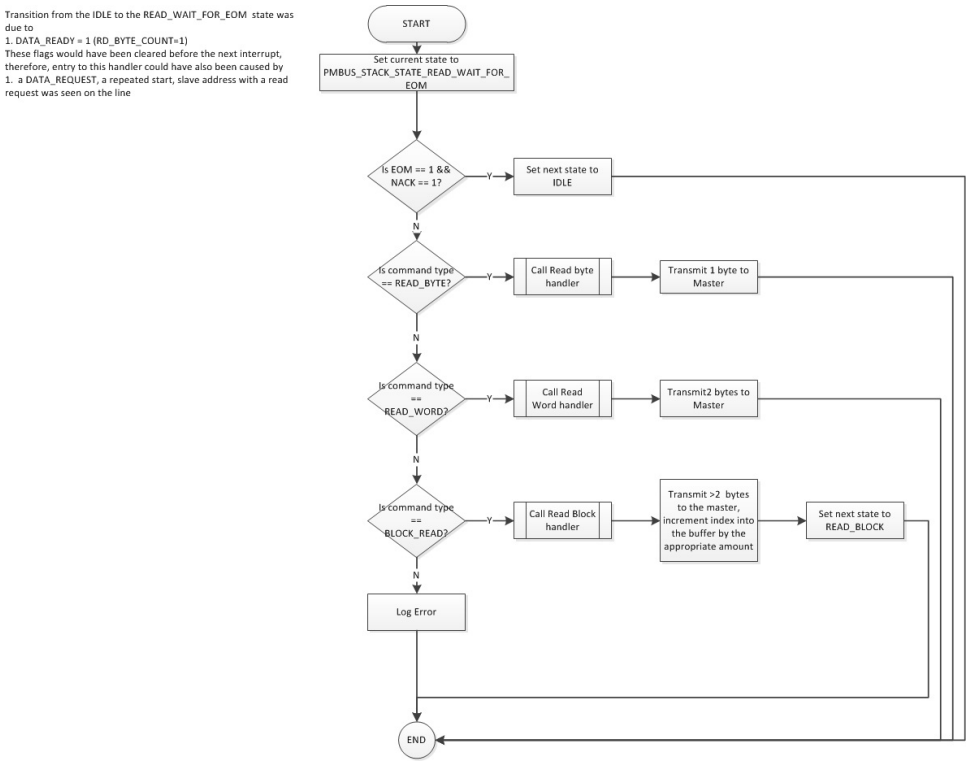

Figure 5.20: The Read and Wait for End-of-Message State

5.3.5 The Block Write or Process Call State

When the master issues a Block Write (or Block Write/ Read/ Process Call) command, the slave state machine will transition from the Idle to the Block Write state. This state handles both Block Writes and Write Word commands. The state machine remains in this state till the master completes sending all its bytes, and returns to the idle state when an End-of-Message signal (with not transmitted bytes) is received.

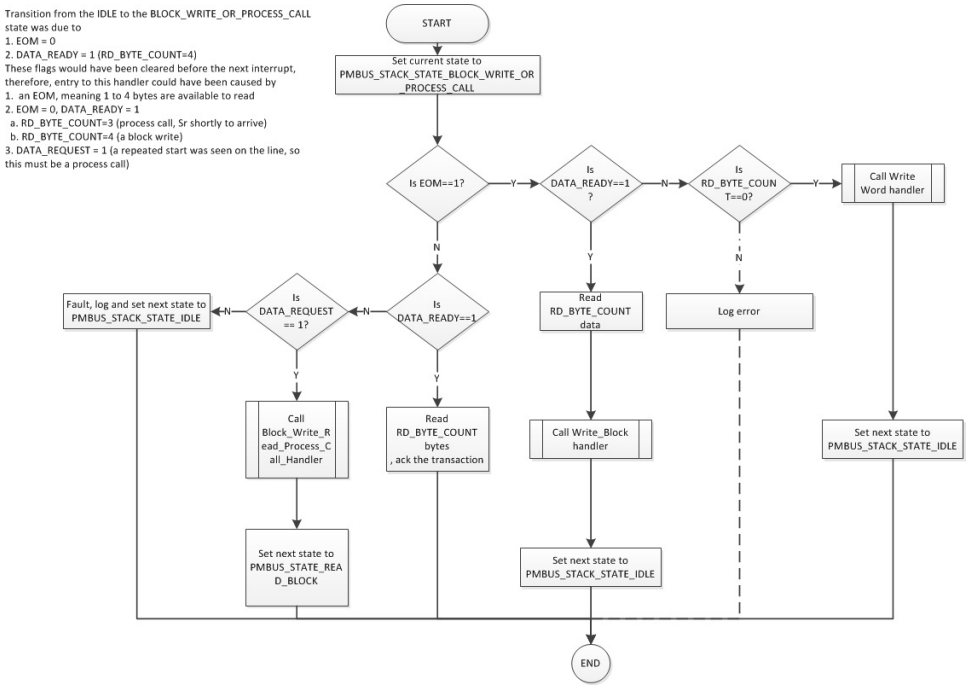

Figure 5.21: The Block Write or Process Call State

5.3.6 The Extended Command State

When the master issues an Extended Read/Write Byte/Word command, the slave state machine will transition from the Idle to the Extended Command state. The slave transition to this state, from idle, when the master issues a repeated start, and only if the first byte sent (during the write phase) was the extension command byte. In the extended command state, the slave determines if the command (the second byte sent during the write phase) was a read or write command, and accordingly proceeds to call the read byte/word handler (for an extended read transaction) or the write byte/word handler (extended write).

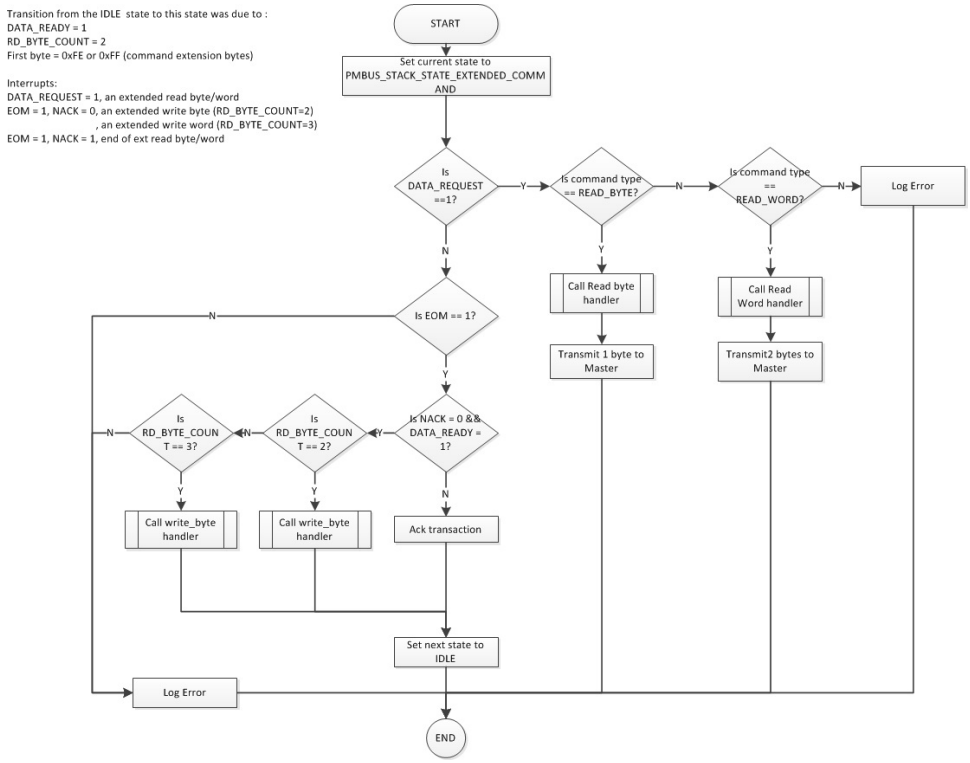

Figure 5.22: The Extended Command State

6 PMBus Communications Slave Stack APIs

6.1 Code Development with Assertion

Defines

- PMBUS_STACK_ASSERT(expr)
- PMBUS_STACK_FILENUM(number)

Functions

- static void PMBusStack_assertionFailed (int16_t file, int16_t line)

Variables

- void(∗) PMBusStack_errorHandler (void)

6.1.1 Define Documentation

6.1.1.1 PMBUS_STACK_ASSERT

The assert() for the PMBus communications stack.

Definition:

```
#define PMBUS_STACK_ASSERT(expr)
```

6.1.1.2 #define PMBUS_STACK_FILENUM(number)

Assign a "unique" number to each file, compiler error on duplicates

6.1.2 Function Documentation

6.1.2.1 static void PMBusStack_assertionFailed (int16_t *file*, int16_t *line*) [inline, static]

Handles failed assertions

Parameters:

file is the file number where the assertion failed

line is the line number where the assertion failed

Description:

This function handles any failed assertions within the stack library.

Returns:

None.

6.1.3 Variable Documentation

6.1.3.1 PMBusStack_errorHandler

Definition:

```
void(*) PMBusStack_errorHandler (void)
```

Description:

Error Handler Function Pointer

In the *Release* Mode, the user must define an error handler, and assign it to this function pointer which gets called when PMBUS_STACK_ASSERT fails in the state machine.

Note:

If the library was built in debug mode, i.e. the macro **_DEBUG** defined then it is unnecessary for the user to define this function in their project. It is only required when using the release version of the library; failure to define this will result in a linker error

Returns:

none

6.2 PMBus Configuration

Data Structures

- PMBus_StackObject
- PMBus_TransactionObject
- PMBus_TransactionObjectUnion

Enumerations

- PMBus_StackMode
- PMBus_StackState

Functions

- int32_t PMBusStack_defaultTransactionHandler (PMBus_StackHandle handle)
- bool PMBusStack_initModule (PMBus_StackHandle handle, const uint32_t moduleBase, uint16_t *buffer)
- static uint16_t * PMBusStackObject_getBufferPointer (PMBus_StackHandle handle)
- static uint16_t * PMBusStackObject_getCurrentPositionPointer (PMBus_StackHandle handle)
- static PMBus_StackState PMBusStackObject_getCurrentState (PMBus_StackHandle handle)
- static PMBus_StackMode PMBusStackObject_getMode (PMBus_StackHandle handle)
- static uint32_t PMBusStackObject_getModuleBase (PMBus_StackHandle handle)
- static uint32_t PMBusStackObject_getModuleStatus (PMBus_StackHandle handle)
- static PMBus_StackState PMBusStackObject_getNextState (PMBus_StackHandle handle)
- static uint16_t PMBusStackObject_getNumOfBytes (PMBus_StackHandle handle)
- static uint16_t PMBusStackObject_getSlaveAddress (PMBus_StackHandle handle)
- static uint16_t PMBusStackObject_getSlaveAddressMask (PMBus_StackHandle handle)
- static transactionHandler PMBusStackObject_getTransactionHandler (PMBus_StackHandle handle, const PMBus_Transaction transaction)
- static PMBus_Transaction PMBusStackObject_getTransactionType (PMBus_StackHandle handle)
- static bool PMBusStackObject_isCommandAndTransactionValid (const uint16_t command, const PMBus_Transaction transaction)
- static bool PMBusStackObject_isPECValid (PMBus_StackHandle handle)
- static void PMBusStackObject_setBufferPointer (PMBus_StackHandle handle, uint16_t *buffer)
- static void PMBusStackObject_setCurrentPositionPointer (PMBus_StackHandle handle, uint16_t *currentPointer)
- static void PMBusStackObject_setCurrentState (PMBus_StackHandle handle, const PMBus_StackState state)
- static void PMBusStackObject_setMode (PMBus_StackHandle handle, const PMBus_StackMode mode)
- static void PMBusStackObject_setModuleBase (PMBus_StackHandle handle, const uint32_t address)

- static void PMBusStackObject_setModuleStatus (PMBus_StackHandle handle, const uint32_t status)
- static void PMBusStackObject_setNextState (PMBus_StackHandle handle, const PMBus_StackState state)
- static void PMBusStackObject_setNumOfBytes (PMBus_StackHandle handle, const uint16_t numberOfBytes)
- static void PMBusStackObject_setPECValidity (PMBus_StackHandle handle, const bool validity)
- static void PMBusStackObject_setSlaveAddress (PMBus_StackHandle handle, const uint16_t address)
- static void PMBusStackObject_setSlaveAddressMask (PMBus_StackHandle handle, const uint16_t addressMask)
- static void PMBusStackObject_setTransactionHandler (PMBus_StackHandle handle, const PMBus_Transaction transaction, transactionHandler handler)
- static void PMBusStackObject_setTransactionType (PMBus_StackHandle handle, const PMBus_Transaction transaction)

Variables

- static const PMBus_TransactionObjectUnion PMBusStack_commandTransactionMap[64]
- PMBus_StackObject pmbusStackSlave
- PMBus_StackHandle pmbusStackSlaveHandle

6.2.1 Data Structure Documentation

6.2.1.1 PMBus_StackObject

Definition:

```
typedef struct
{
    uint32_t moduleBase;
    uint32_t moduleStatus;
    PMBus_StackMode mode;
    uint16_t slaveAddress;
    uint16_t slaveAddressMask;
    PMBus_StackState currentState;
    PMBus_StackState nextState;
    uint16_t *bufferPointer;
    uint16_t *currentBufferPointer;
    uint16_t numOfBytes;
    bool PECValidity;
    PMBus_Transaction transaction;
    transactionHandler transactionHandle[NTRANSACTIONS];
}
PMBus_StackObject
```

Members:

moduleBase Base address of the PMBus module.

moduleStatus Status register of the PMBus module.
mode PMBus mode of operation.
slaveAddress Slave address for the PMBus module.
slaveAddressMask Slave address mask for PMBus module.
currentState Current state of the state machine.
nextState next state of the state machine
bufferPointer pointer to a buffer of length >= 4
currentBufferPointer Current position in the buffer.
numOfBytes Number of bytes sent/received.
PECValidity Valid PEC received or sent.
transaction Current Transaction type.
transactionHandle Handler for each transaction.

Description:
PMBUS Slave Mode Object.

6.2.1.2 PMBus_TransactionObject

Definition:

```
typedef struct
{
    uint16_t transaction0;
    uint16_t transaction1;
    uint16_t transaction2;
    uint16_t transaction3;
}
PMBus_TransactionObject
```

Members:
transaction0 First Transaction field.
transaction1 Second Transaction field.
transaction2 Third Transaction field.
transaction3 Fourth Transaction field.

Description:
Structure that packs 4 transaction fields into a word.

6.2.1.3 PMBus_TransactionObjectUnion

Definition:

```
typedef struct
{
    PMBus_TransactionObject object;
    uint16_t transactionField;
}
PMBus_TransactionObjectUnion
```

Members:
object
transactionField

Description:
Union of the packed transactions struct and an unsigned word.

6.2.2 Enumeration Documentation

6.2.2.1 PMBus_StackMode

Description:
PMBus Mode of Operation.

Enumerators:
PMBUS_STACK_MODE_SLAVE PMBus operates in slave mode.
PMBUS_STACK_MODE_MASTER PMBus operates in master mode.

6.2.2.2 PMBus_StackState

Description:
Enumeration of the states in the PMBus state machine.

Enumerators:
PMBUS_STACK_STATE_IDLE PMBus in the Idle state.
PMBUS_STACK_STATE_RECEIVE_BYTE_WAIT_FOR_EOM PMBus is waiting on an end-of-message signal (NACK on last data).
PMBUS_STACK_STATE_READ_BLOCK PMBus is reading a block of data.
PMBUS_STACK_STATE_READ_WAIT_FOR_EOM PMBus is waiting on an end-of-message signal (NACK on last data).
PMBUS_STACK_STATE_BLOCK_WRITE_OR_PROCESS_CALL PMBus is either writing a block or issuing a process call.
PMBUS_STACK_STATE_EXTENDED_COMMAND PMBus is doing an extended read/write byte/word.

6.2.3 Function Documentation

6.2.3.1 PMBusStack_defaultTransactionHandler

Default Transaction Handler

Prototype:
```
int32_t
PMBusStack_defaultTransactionHandler(PMBus_StackHandle handle)
```

Parameters:
handle is the handle to the PMBus stack object

Description:
This function is the default transaction handler. Default behavior is to call PMBusStack_assertionFailed().

Returns:
If function returns, always returns value of -1.

6.2.3.2 PMBusStack_initModule

Initialize the PMBus Module

Prototype:

```
bool
PMBusStack_initModule(PMBus_StackHandle handle,
                      const uint32_t moduleBase,
                      uint16_t *buffer)
```

Parameters:

handle is the handle to the PMBus stack object

moduleBase is the base address for the PMBus peripheral instance

buffer is the buffer pointer for use by the PMBus stack object

Description:

This function initializes the PMBus peripheral (based on mode set in the PMBus stack object) for slave or master mode, assigns the buffer pointer to the PMBus stack object, and enables interrupts.

Note:

This function enables the PMBus interrupts but the user must register the necessary interrupt service routine handler and configure the ISR to call the required library handler

The buffer must point to an array of at least 4 words

Returns:

Returns **true** if initialization is successful and **false** when initialization isn't successful.

6.2.3.3 PMBusStackObject_getBufferPointer

Get PMBus Module Buffer Pointer

Prototype:

```
static uint16_t *
PMBusStackObject_getBufferPointer(PMBus_StackHandle handle) [inline,
static]
```

Parameters:

handle is the handle to the PMBus stack object

Description:

This function gets the PMBus module pointer to the buffer that stores messages from the PMBus stack object.

Returns:

Buffer pointer

6.2.3.4 PMBusStackObject_getCurrentPositionPointer

Get PMBus Module Current Buffer Position Pointer

Prototype:

```
static uint16_t *
PMBusStackObject_getCurrentPositionPointer(PMBus_StackHandle handle)
[inline, static]
```

Parameters:

handle is the handle to the PMBus stack object

Description:

This function gets the PMBus module pointer to the current position in the buffer from the PMBus stack object.

Returns:

Current buffer pointer.

6.2.3.5 PMBusStackObject_getCurrentState

Get PMBus Module Current State

Prototype:

```
static PMBus_StackState
PMBusStackObject_getCurrentState(PMBus_StackHandle handle) [inline,
static]
```

Parameters:

handle is the handle to the PMBus stack object

Description:

This function gets the PMBus module current state of the state machine from the PMBus stack object.

Returns:

Current State

- PMBUS_STACK_STATE_IDLE : Idle State
- PMBUS_STACK_STATE_RECEIVE_BYTE_WAIT_FOR_EOM : Waiting on end-of-message signal state
- PMBUS_STACK_STATE_READ_BLOCK : Reading a block of data state
- PMBUS_STACK_STATE_READ_WAIT_FOR_EOM : Waiting on end-of-message signal
- PMBUS_STACK_STATE_BLOCK_WRITE_OR_PROCESS_CALL : Writing a block or issuing a process call state
- PMBUS_STACK_STATE_EXTENDED_COMMAND : Extended read/write byte/word state

6.2.3.6 PMBusStackObject_getMode

Get PMBus Module Mode

Prototype:

```
static PMBus_StackMode
PMBusStackObject_getMode(PMBus_StackHandle handle) [inline, static]
```

Parameters:
handle is the handle to the PMBus stack object

Description:
This function gets the PMBus module instance operating mode from the PMBus stack object.

Returns:
PMBUS_STACK_MODE_SLAVE or PMBUS_STACK_MODE_MASTER

6.2.3.7 PMBusStackObject_getModuleBase

Get PMBus Module Base Address

Prototype:

```
static uint32_t
PMBusStackObject_getModuleBase(PMBus_StackHandle handle) [inline,
static]
```

Parameters:
handle is the handle to the PMBus stack object

Description:
This function gets the PMBus module instance base address from the PMBus stack object.

Returns:
PMBus module instance base address

6.2.3.8 PMBusStackObject_getModuleStatus

Get PMBus Module Status

Prototype:

```
static uint32_t
PMBusStackObject_getModuleStatus(PMBus_StackHandle handle) [inline,
static]
```

Parameters:
handle is the handle to the PMBus stack object

Description:
This function gets the PMBus module instance status value from the PMBus stack object.

Returns:
Module status

6.2.3.9 PMBusStackObject_getNextState

Get PMBus Module Next State

Prototype:

```
static PMBus_StackState
PMBusStackObject_getNextState(PMBus_StackHandle handle) [inline,
static]
```

Parameters:

handle is the handle to the PMBus stack object

Description:

This function gets the PMBus module next state of the state machine from the PMBus stack object.

Returns:

Next State

- PMBUS_STACK_STATE_IDLE : Idle State
- PMBUS_STACK_STATE_RECEIVE_BYTE_WAIT_FOR_EOM : Waiting on end-of-message signal state
- PMBUS_STACK_STATE_READ_BLOCK : Reading a block of data state
- PMBUS_STACK_STATE_READ_WAIT_FOR_EOM : Waiting on end-of-message signal
- PMBUS_STACK_STATE_BLOCK_WRITE_OR_PROCESS_CALL : Writing a block or issuing a process call state
- PMBUS_STACK_STATE_EXTENDED_COMMAND : Extended read/write byte/word state

6.2.3.10 PMBusStackObject_getNumOfBytes

Get PMBus Module Number of Bytes

Prototype:

```
static uint16_t
PMBusStackObject_getNumOfBytes(PMBus_StackHandle handle) [inline,
static]
```

Parameters:

handle is the handle to the PMBus stack object

Description:

This function gets the PMBus module number of bytes being sent or received from the PMBus stack object.

Returns:

Number of bytes

6.2.3.11 PMBusStackObject_getSlaveAddress

Get PMBus Module Slave Address

Prototype:

```
static uint16_t
PMBusStackObject_getSlaveAddress(PMBus_StackHandle handle) [inline,
static]
```

Parameters:
handle is the handle to the PMBus stack object

Description:
This functions gets the PMBus module instance slave address from the PMBus stack object

Returns:
Slave address

6.2.3.12 PMBusStackObject_getSlaveAddressMask

Get PMBus Module Slave Address Mask

Prototype:

```
static uint16_t
PMBusStackObject_getSlaveAddressMask(PMBus_StackHandle handle)
[inline, static]
```

Parameters:
handle is the handle to the PMBus stack object

Description:
This function gets the PMBus module instance slave address mask from the PMBus stack object.

Returns:
Slave address mask

6.2.3.13 PMBusStackObject_getTransactionHandler

Get PMBus Module Transaction Handler

Prototype:

```
static transactionHandler
PMBusStackObject_getTransactionHandler(PMBus_StackHandle handle,
                                       const PMBus_Transaction
transaction) [inline, static]
```

Parameters:
handle is the handle to the PMBus stack object
transaction is the PMBus transaction type

Description:
This function gets the PMBus module transaction handler function for a specific transaction type from the PMBus stack object. Transactions include:

- PMBUS_TRANSACTION_NONE
- PMBUS_TRANSACTION_QUICKCOMMAND
- PMBUS_TRANSACTION_WRITEBYTE
- PMBUS_TRANSACTION_READBYTE
- PMBUS_TRANSACTION_SENDBYTE

- PMBUS_TRANSACTION_RECEIVEBYTE
- PMBUS_TRANSACTION_BLOCKWRITE
- PMBUS_TRANSACTION_BLOCKREAD
- PMBUS_TRANSACTION_WRITEWORD
- PMBUS_TRANSACTION_READWORD
- PMBUS_TRANSACTION_BLOCKWRPC

Returns:
Pointer to transaction function handler

6.2.3.14 PMBusStackObject_getTransactionType

Get PMBus Module Transaction Type

Prototype:

```
static PMBus_Transaction
PMBusStackObject_getTransactionType(PMBus_StackHandle handle)
[inline, static]
```

Parameters:
handle is the handle to the PMBus stack object

Description:
This function gets the PMBus module transaction type from the PMBus stack object.

Returns:
Transaction type

- PMBUS_TRANSACTION_NONE
- PMBUS_TRANSACTION_QUICKCOMMAND
- PMBUS_TRANSACTION_WRITEBYTE
- PMBUS_TRANSACTION_READBYTE
- PMBUS_TRANSACTION_SENDBYTE
- PMBUS_TRANSACTION_RECEIVEBYTE
- PMBUS_TRANSACTION_BLOCKWRITE
- PMBUS_TRANSACTION_BLOCKREAD
- PMBUS_TRANSACTION_WRITEWORD
- PMBUS_TRANSACTION_READWORD
- PMBUS_TRANSACTION_BLOCKWRPC

6.2.3.15 PMBusStackObject_isCommandAndTransactionValid [static]

Check if the PMBus Command and Transaction Type are Valid

Prototype:

```
static bool
PMBusStackObject_isCommandAndTransactionValid(const uint16_t command,
                                              const PMBus_Transaction
transaction)
```

Parameters:

handle is the handle to the PMBus stack object

transaction is the PMBus transaction type

Description:

This function will query the PMBus command transaction mapping for the given command to see if it can find a match for the given transaction type.

Returns:

- **true** if the command and transaction match (and therefore valid)
- **false** if the command and transaction don't match

6.2.3.16 PMBusStackObject_isPECValid

Get PMBus Module Packet Error Checking (PEC) Validity

Prototype:

```
static bool
PMBusStackObject_isPECValid(PMBus_StackHandle handle) [inline,
static]
```

Parameters:

handle is the handle to the PMBus stack object

Description:

This function gets the PMBus module PEC validity status (either valid or invalid) from the PMBus stack object.

Returns:

Returns **true** if PEC is valid and **false** if PEC is invalid

6.2.3.17 PMBusStackObject_setBufferPointer

Set PMBus Module Buffer Pointer

Prototype:

```
static void
PMBusStackObject_setBufferPointer(PMBus_StackHandle handle,
                                  uint16_t *buffer) [inline, static]
```

Parameters:

handle is the handle to the PMBus stack object

buffer is the pointer to the buffer (must be buffer of size >= 4)

Description:

This function sets the PMBus module pointer to the buffer that stores messages in the PMBus stack object.

Returns:

None.

6.2.3.18 PMBusStackObject_setCurrentPositionPointer

Set PMBus Module Current Buffer Position Pointer

Prototype:

```
static void
PMBusStackObject_setCurrentPositionPointer(PMBus_StackHandle handle,
                                           uint16_t *currentPointer)
[inline, static]
```

Parameters:

handle is the handle to the PMBus stack object

currentPointer is the pointer to the current position in the buffer

Description:

This function sets the PMBus module pointer to the current position in the buffer in the PMBus stack object.

Returns:

None.

6.2.3.19 PMBusStackObject_setCurrentState

Set PMBus Module Current State

Prototype:

```
static void
PMBusStackObject_setCurrentState(PMBus_StackHandle handle,
                                 const PMBus_StackState state)
[inline, static]
```

Parameters:

handle is the handle to the PMBus stack object

state is the current state of the PMBus state machine

Description:

This function sets the PMBus module current state of the state machine in the PMBus stack object. States include:

- PMBUS_STACK_STATE_IDLE : Idle State
- PMBUS_STACK_STATE_RECEIVE_BYTE_WAIT_FOR_EOM : Waiting on end-of-message signal state
- PMBUS_STACK_STATE_READ_BLOCK : Reading a block of data state
- PMBUS_STACK_STATE_READ_WAIT_FOR_EOM : Waiting on end-of-message signal
- PMBUS_STACK_STATE_BLOCK_WRITE_OR_PROCESS_CALL : Writing a block or issuing a process call state
- PMBUS_STACK_STATE_EXTENDED_COMMAND : Extended read/write byte/word state

Returns:

None.

6.2.3.20 PMBusStackObject_setMode

Set PMBus Module Mode

Prototype:

```
static void
PMBusStackObject_setMode(PMBus_StackHandle handle,
                         const PMBus_StackMode mode) [inline, static]
```

Parameters:

handle is the handle to the PMBus stack object

mode is the PMBus module mode (either slave or master)

Description:

This function sets the PMBus module instance operating mode (master or slave) in the PMBus stack object.

Returns:

None.

6.2.3.21 PMBusStackObject_setModuleBase

Set PMBus Module Base Address

Prototype:

```
static void
PMBusStackObject_setModuleBase(PMBus_StackHandle handle,
                               const uint32_t address) [inline,
static]
```

Parameters:

handle is the handle to the PMBus stack object

address is the base address for the PMBus peripheral instance

Description:

This function sets the PMBus module instance base address in the PMBus stack object.

Returns:

None.

6.2.3.22 PMBusStackObject_setModuleStatus

Set PMBus Module Status

Prototype:

```
static void
PMBusStackObject_setModuleStatus(PMBus_StackHandle handle,
                                 const uint32_t status) [inline,
static]
```

Parameters:

handle is the handle to the PMBus stack object

status is the PMBus module instance register status

Description:
This function sets the PMBus module instance register status in the PMBus stack object.

Returns:
None.

6.2.3.23 PMBusStackObject_setNextState

Set PMBus Module Next State

Prototype:

```
static void
PMBusStackObject_setNextState(PMBus_StackHandle handle,
                              const PMBus_StackState state) [inline,
static]
```

Parameters:
handle is the handle to the PMBus stack object
state is the next state of the PMBus state machine

Description:
This function sets the PMBus module next state of the state machine in the PMBus stack object. States include:

- PMBUS_STACK_STATE_IDLE : Idle State
- PMBUS_STACK_STATE_RECEIVE_BYTE_WAIT_FOR_EOM : Waiting on end-of-message signal state
- PMBUS_STACK_STATE_READ_BLOCK : Reading a block of data state
- PMBUS_STACK_STATE_READ_WAIT_FOR_EOM : Waiting on end-of-message signal
- PMBUS_STACK_STATE_BLOCK_WRITE_OR_PROCESS_CALL : Writing a block or issuing a process call state
- PMBUS_STACK_STATE_EXTENDED_COMMAND : Extended read/write byte/word state

Returns:
None.

6.2.3.24 PMBusStackObject_setNumOfBytes

Set PMBus Module Number of Bytes

Prototype:

```
static void
PMBusStackObject_setNumOfBytes(PMBus_StackHandle handle,
                               const uint16_t numberOfBytes) [inline,
static]
```

Parameters:
handle is the handle to the PMBus stack object
numberOfBytes is the number of bytes sent/received

Description:
This function sets the PMBus module number of bytes being sent or received in the PMBus stack object.

Returns:
None.

6.2.3.25 PMBusStackObject_setPECValidity

Set PMBus Module Packet Error Checking (PEC) Validity

Prototype:
```
static void
PMBusStackObject_setPECValidity(PMBus_StackHandle handle,
                                const bool validity) [inline, static]
```

Parameters:
handle is the handle to the PMBus stack object
validity is the PEC validity status (true = PEC is valid, false = PEC is invalid)

Description:
This function sets the PMBus module PEC validity status (either valid or invalid) in the PMBus stack object.

Returns:
None.

6.2.3.26 PMBusStackObject_setSlaveAddress

Set PMBus Module Slave Address

Prototype:
```
static void
PMBusStackObject_setSlaveAddress(PMBus_StackHandle handle,
                                 const uint16_t address) [inline,
static]
```

Parameters:
handle is the handle to the PMBus stack object
address is the address of the PMBus module in slave mode

Description:
This function sets the PMBus module instance slave address in the PMBus stack object.

Returns:
None.

6.2.3.27 PMBusStackObject_setSlaveAddressMask

Set PMBus Module Slave Address Mask

Prototype:

```
static void
PMBusStackObject_setSlaveAddressMask(PMBus_StackHandle handle,
                                     const uint16_t addressMask)
[inline, static]
```

Parameters:

handle is the handle to the PMBus stack object

addressMask is the address mask of the PMBus module in slave mode

Description:

This function sets the PMBus module instance slave address mask in the PMBus stack object.

Returns:

None.

6.2.3.28 PMBusStackObject_setTransactionHandler

Set PMBus Module Transaction Handler

Prototype:

```
static void
PMBusStackObject_setTransactionHandler(PMBus_StackHandle handle,
                                       const PMBus_Transaction
transaction,
                                       transactionHandler handler)
[inline, static]
```

Parameters:

handle is the handle to the PMBus stack object

transaction is the PMBus transaction type

handler is the pointer to the function to handle the transaction

Description:

This function sets the PMBus module transaction handler function for a specific transaction type in the PMBus stack object. Transactions include:

- PMBUS_TRANSACTION_NONE
- PMBUS_TRANSACTION_QUICKCOMMAND
- PMBUS_TRANSACTION_WRITEBYTE
- PMBUS_TRANSACTION_READBYTE
- PMBUS_TRANSACTION_SENDBYTE
- PMBUS_TRANSACTION_RECEIVEBYTE
- PMBUS_TRANSACTION_BLOCKWRITE
- PMBUS_TRANSACTION_BLOCKREAD
- PMBUS_TRANSACTION_WRITEWORD
- PMBUS_TRANSACTION_READWORD

- PMBUS_TRANSACTION_BLOCKWRPC

Returns:
None.

6.2.3.29 PMBusStackObject_setTransactionType

Set PMBus Module Transaction Type

Prototype:

```
static void
PMBusStackObject_setTransactionType(PMBus_StackHandle handle,
                                    const PMBus_Transaction
transaction) [inline, static]
```

Parameters:
handle is the handle to the PMBus stack object
transaction is the PMBus transaction type

Description:
This function sets the PMBus module transaction type in the PMBus stack object. Transactions include:

- PMBUS_TRANSACTION_NONE
- PMBUS_TRANSACTION_QUICKCOMMAND
- PMBUS_TRANSACTION_WRITEBYTE
- PMBUS_TRANSACTION_READBYTE
- PMBUS_TRANSACTION_SENDBYTE
- PMBUS_TRANSACTION_RECEIVEBYTE
- PMBUS_TRANSACTION_BLOCKWRITE
- PMBUS_TRANSACTION_BLOCKREAD
- PMBUS_TRANSACTION_WRITEWORD
- PMBUS_TRANSACTION_READWORD
- PMBUS_TRANSACTION_BLOCKWRPC

Returns:
None.

6.2.4 Variable Documentation

6.2.4.1 PMBusStack_commandTransactionMap [static]

Definition:

```
static const PMBus_TransactionObjectUnion PMBusStack_commandTransactionMap[64]
```

Description:
PMBus Command Transaction Type Map

Each position in the map corresponds to a particular command, its entry lists the type of read transaction that is involved. It will used to distinguish between read byte, read word, and block read commands in the state machine

Any command that has both a write and read command will have the read transaction type as its entry. A command without a read command will have its write transaction type as its entry

6.2.4.2 pmbusStackSlave

Definition:

```
PMBus_StackObject pmbusStackSlave
```

Description:

PMBus Slave Object.

6.2.4.3 pmbusStackSlaveHandle

Definition:

```
PMBus_StackHandle pmbusStackSlaveHandle
```

Description:

Handle to the slave object.

6.3 PMBus State Machine Handler

Functions

- void PMBusStack_slaveBlockWriteOrProcessCallStateHandler (PMBus_StackHandle handle)
- void PMBusStack_slaveExtendedCommandStateHandler (PMBus_StackHandle handle)
- void PMBusStack_slaveIdleStateHandler (PMBus_StackHandle handle)
- void PMBusStack_slaveReadBlockStateHandler (PMBus_StackHandle handle)
- void PMBusStack_slaveReadWaitForEOMStateHandler (PMBus_StackHandle handle)
- void PMBusStack_slaveReceiveByteWaitForEOMStateHandler (PMBus_StackHandle handle)
- void PMBusStack_slaveStateHandler (PMBus_StackHandle handle)

6.3.1 Function Documentation

6.3.1.1 PMBusStack_slaveBlockWriteOrProcessCallStateHandler

PMBus Slave Block Write or Process Call State Handler

Prototype:

```
void
PMBusStack_slaveBlockWriteOrProcessCallStateHandler(PMBus_StackHandle
handle)
```

Parameters:

handle is the handle to the PMBus stack object

Description:

This function handles the state in the slave state machine that is entered when a block write or process call commands are used.

Returns:

None.

6.3.1.2 PMBusStack_slaveExtendedCommandStateHandler

PMBus Slave Extended Read/Write Byte/Word State Handler

Prototype:

```
void
PMBusStack_slaveExtendedCommandStateHandler(PMBus_StackHandle handle)
```

Parameters:

handle is the handle to the PMBus stack object

Description:

This function handles the state in the slave state machine that is entered when extended commands are used. These include extended read byte, read word, write byte, and write word transactions.

Returns:
None.

6.3.1.3 PMBusStack_slaveIdleStateHandler

PMBus Slave Idle State Handler

Prototype:

```
void
PMBusStack_slaveIdleStateHandler(PMBus_StackHandle handle)
```

Parameters:
handle is the handle to the PMBus stack object

Description:
This function handles the idle state in the slave state machine.

Returns:
None.

6.3.1.4 PMBusStack_slaveReadBlockStateHandler

PMBus Slave Read Block State Handler

Prototype:

```
void
PMBusStack_slaveReadBlockStateHandler(PMBus_StackHandle handle)
```

Parameters:
handle is the handle to the PMBus stack object

Description:
This function handles the state in the slave state machine that is entered when a read block command is used.

Returns:
None.

6.3.1.5 PMBusStack_slaveReadWaitForEOMStateHandler

PMBus Slave Read/Wait for EOM State Handler

Prototype:

```
void
PMBusStack_slaveReadWaitForEOMStateHandler(PMBus_StackHandle handle)
```

Parameters:
handle is the handle to the PMBus stack object

Description:
This function handles the state in the slave state machine that is entered when reading/waiting for the end-of-message.

Returns:
None.

6.3.1.6 PMBusStack_slaveReceiveByteWaitForEOMStateHandler

PMBus Slave Receive Byte Wait-for-EOM State Handler

Prototype:
```
void
PMBusStack_slaveReceiveByteWaitForEOMStateHandler(PMBus_StackHandle
handle)
```

Parameters:
handle is the handle to the PMBus stack object

Description:
This function handles the state in the slave state machine that is entered when a receive byte request is active and slave is waiting for end-of-message.

Returns:
None.

6.3.1.7 PMBusStack_slaveStateHandler

PMBus Slave Stack State Machine Handler

Prototype:
```
void
PMBusStack_slaveStateHandler(PMBus_StackHandle handle)
```

Parameters:
handle is the handle to the PMBus stack object

Description:
This function implements the state machine of the PMBus in slave mode. This handler is designed to operate within the PMBus interrupt service routine (ISR) triggered by the following interrupts:

- Data Ready (Read buffer is full)
- Data Request (Master has requested data)
- EOM (Master signals an end of a block message)

Note:
The handler must be called in the PMBus ISR only

Returns:
None.

7 PMBus Library Examples

The PMBus library includes a master and slave example set for each supported device. The master example tests the slave (which uses the communication stack) handling of the various PMBus commands and transactions. These examples have been implemented and tested on the device *controlCard* platform.

Refer to Figure 7.1 on the required pullups configuration to properly setup a PMBus connection.

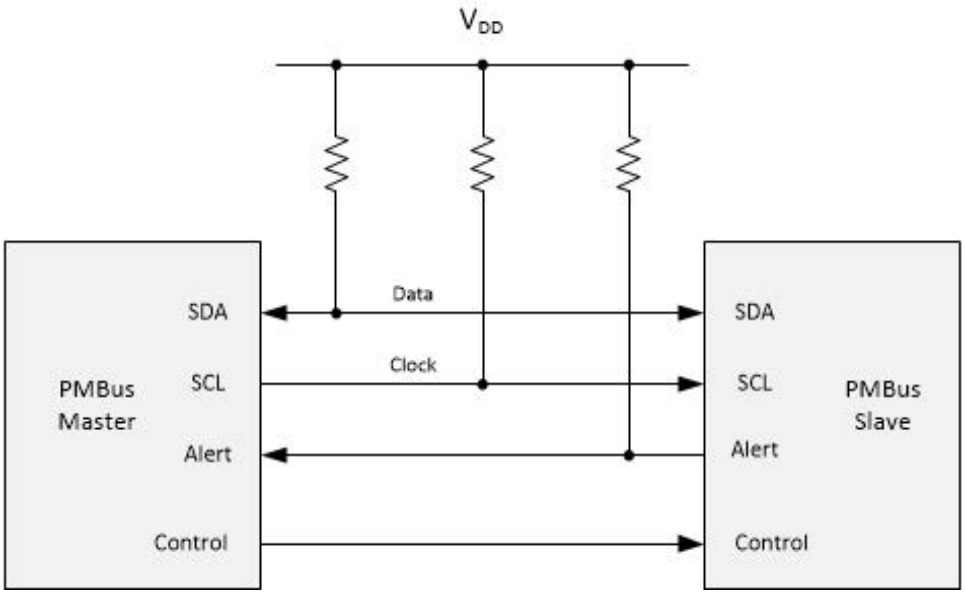

Figure 7.1: PMBus Connection Pull-Ups

7.1 PMBus Slave Mode Tests

Functions

- void PMBusSlave_alertTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_blockRead3BytesTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_blockReadTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_blockWrite2BytesTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_blockWrite3BytesTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_blockWriteReadProcessCallTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_blockWriteTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_extendedReadByteTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_extendedReadWordTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_extendedWriteByteTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_extendedWriteWordTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_groupCommandTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_noAlertTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_processCallTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_quickCommandTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_readByteTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_readWordTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_receiveByteTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_sendByteTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_writeByteTestHandler (PMBus_StackHandle handle)
- void PMBusSlave_writeWordTestHandler (PMBus_StackHandle handle)

Variables

- PMBus_StackObject pmbusStackSlave
- PMBus_StackHandle pmbusStackSlaveHandle

7.1.1 Function Documentation

7.1.1.1 PMBusSlave_alertTestHandler

Alert Test Handler

Prototype:

```
void
PMBusSlave_alertTestHandler(PMBus_StackHandle handle)
```

Parameters:

handle is the handle to the PMBus stack object

Note:

Expected Pass Value: 1

Returns:
None.

7.1.1.2 void PMBusSlave_blockRead3BytesTestHandler (PMBus_StackHandle *handle*)

Block Read (3 bytes) Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 1

Returns:
None.

7.1.1.3 void PMBusSlave_blockReadTestHandler (PMBus_StackHandle *handle*)

Block Read (255 bytes) Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 1

Returns:
None.

7.1.1.4 void PMBusSlave_blockWrite2BytesTestHandler (PMBus_StackHandle *handle*)

Block Write (2 bytes) Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
An attempted block write with 1 byte is a write byte, 2 bytes a write word - the master does not put the byte count on the line.

Make sure to run the write word test before this, as the original write word Handler overwrites the handler to point to this function.

Expected Pass Value: 6

Returns:
None.

7.1.1.5 void PMBusSlave_blockWrite3BytesTestHandler (PMBus_StackHandle *handle*)

Block Write (3 bytes) Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
An attempted block write with 1 byte is a write byte, 2 bytes a write word - the master does not put the byte count on the line.

Make sure to run the block write test before this, as the original block write handler overwrites the handler to point to this function.

Expected Pass Value: 8

Returns:
None.

7.1.1.6 void PMBusSlave_blockWriteReadProcessCallTestHandler (PMBus_StackHandle *handle*)

Block Write/Read/Process Call (255 bytes) Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 259

Returns:
None.

7.1.1.7 void PMBusSlave_blockWriteTestHandler (PMBus_StackHandle *handle*)

Block Write (255 bytes) Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 260

Returns:
None.

7.1.1.8 void PMBusSlave_extendedReadByteTestHandler (PMBus_StackHandle *handle*)

Extended Read Byte Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 1

Returns:
None.

7.1.1.9 void PMBusSlave_extendedReadWordTestHandler (PMBus_StackHandle *handle*)

ExtendedRead Word Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 1

Returns:
None.

7.1.1.10 void PMBusSlave_extendedWriteByteTestHandler (PMBus_StackHandle *handle*)

Extended Write Byte Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 6

Returns:
None.

7.1.1.11 void PMBusSlave_extendedWriteWordTestHandler (PMBus_StackHandle *handle*)

Extended Write Word Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 7

Returns:
None.

7.1.1.12 void PMBusSlave_groupCommandTestHandler (PMBus_StackHandle *handle*)

Group Command (slave 1st addressed) Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 6

Returns:
None.

7.1.1.13 void PMBusSlave_noAlertTestHandler (PMBus_StackHandle *handle*)

Alert (from 2nd slave) Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
This test requires a 2nd PMBus slave on the network asserting the alert line.

Expected Pass Value: 1

Returns:
None.

7.1.1.14 void PMBusSlave_processCallTestHandler (PMBus_StackHandle *handle*)

Process Call Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 5

Returns:
None.

7.1.1.15 void PMBusSlave_quickCommandTestHandler (PMBus_StackHandle *handle*)

Quick Command Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 4

Returns:
None.

7.1.1.16 void PMBusSlave_readByteTestHandler (PMBus_StackHandle *handle*)

Read Byte Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 1

Returns:
None.

7.1.1.17 void PMBusSlave_readWordTestHandler (PMBus_StackHandle *handle*)

Read Word Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 1

Returns:
None.

7.1.1.18 void PMBusSlave_receiveByteTestHandler (PMBus_StackHandle *handle*)

Receive Byte Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 1

Returns:
None.

7.1.1.19 void PMBusSlave_sendByteTestHandler (PMBus_StackHandle *handle*)

Send Byte Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 4

Returns:
None.

7.1.1.20 void PMBusSlave_writeByteTestHandler (PMBus_StackHandle *handle*)

Write Byte Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 5

Returns:
None.

7.1.1.21 void PMBusSlave_writeWordTestHandler (PMBus_StackHandle *handle*)

Write Word Test Handler

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected Pass Value: 6

Returns:
None.

7.1.2 Variable Documentation

7.1.2.1 PMBus_StackObject pmbusStackSlave

PMBus Slave Object.

7.1.2.2 PMBus_StackHandle pmbusStackSlaveHandle

Handle to the slave object.

7.2 PMBus Master Mode Tests

Data Structures

- PMBus_TestObject

Functions

- void PMBusMaster_initAlertTest (PMBus_TestHandle handle)
- void PMBusMaster_initBlockRead3BytesTest (PMBus_TestHandle handle)
- void PMBusMaster_initBlockReadTest (PMBus_TestHandle handle)
- void PMBusMaster_initBlockWrite2BytesTest (PMBus_TestHandle handle)
- void PMBusMaster_initBlockWrite3BytesTest (PMBus_TestHandle handle)
- void PMBusMaster_initBlockWriteReadProcessCallTest (PMBus_TestHandle handle)
- void PMBusMaster_initBlockWriteTest (PMBus_TestHandle handle)
- void PMBusMaster_initExtendedReadByteTest (PMBus_TestHandle handle)
- void PMBusMaster_initExtendedReadWordTest (PMBus_TestHandle handle)
- void PMBusMaster_initExtendedWriteByteTest (PMBus_TestHandle handle)
- void PMBusMaster_initExtendedWriteWordTest (PMBus_TestHandle handle)
- void PMBusMaster_initGroupCommandTest (PMBus_TestHandle handle)
- void PMBusMaster_initNoAlertTest (PMBus_TestHandle handle)
- void PMBusMaster_initProcessCallTest (PMBus_TestHandle handle)
- void PMBusMaster_initQuickCommandTest (PMBus_TestHandle handle)
- void PMBusMaster_initReadByteTest (PMBus_TestHandle handle)
- void PMBusMaster_initReadWordTest (PMBus_TestHandle handle)
- void PMBusMaster_initReceiveByteTest (PMBus_TestHandle handle)
- void PMBusMaster_initSendByteTest (PMBus_TestHandle handle)
- void PMBusMaster_initWriteByteTest (PMBus_TestHandle handle)
- void PMBusMaster_initWriteWordTest (PMBus_TestHandle handle)
- static void PMBusMaster_resetGlobalFlags (void)
- static void PMBusMaster_resetTestObject (PMBus_TestHandle handle)
- void PMBusMaster_runAlertTest (PMBus_TestHandle handle)
- void PMBusMaster_runBlockRead3BytesTest (PMBus_TestHandle handle)
- void PMBusMaster_runBlockReadTest (PMBus_TestHandle handle)
- void PMBusMaster_runBlockWrite2BytesTest (PMBus_TestHandle handle)
- void PMBusMaster_runBlockWrite3BytesTest (PMBus_TestHandle handle)
- void PMBusMaster_runBlockWriteReadProcessCallTest (PMBus_TestHandle handle)
- void PMBusMaster_runBlockWriteTest (PMBus_TestHandle handle)
- void PMBusMaster_runExtendedReadByteTest (PMBus_TestHandle handle)
- void PMBusMaster_runExtendedReadWordTest (PMBus_TestHandle handle)
- void PMBusMaster_runExtendedWriteByteTest (PMBus_TestHandle handle)
- void PMBusMaster_runExtendedWriteWordTest (PMBus_TestHandle handle)
- void PMBusMaster_runGroupCommandTest (PMBus_TestHandle handle)
- void PMBusMaster_runNoAlertTest (PMBus_TestHandle handle)

- void PMBusMaster_runProcessCallTest (PMBus_TestHandle handle)
- void PMBusMaster_runQuickCommandTest (PMBus_TestHandle handle)
- void PMBusMaster_runReadByteTest (PMBus_TestHandle handle)
- void PMBusMaster_runReadWordTest (PMBus_TestHandle handle)
- void PMBusMaster_runReceiveByteTest (PMBus_TestHandle handle)
- void PMBusMaster_runSendByteTest (PMBus_TestHandle handle)
- void PMBusMaster_runWriteByteTest (PMBus_TestHandle handle)
- void PMBusMaster_runWriteWordTest (PMBus_TestHandle handle)

7.2.1 Data Structure Documentation

7.2.1.1 PMBus_TestObject

Definition:

```
typedef struct
{
    uint16_t count;
    int16_t pass;
    int16_t fail;
    bool enabled;
    void (*init)(void *);
    void (*run)(void *);
}
PMBus_TestObject
```

Members:

count bytes (block transactions > 4)

pass pass metric

fail fail metric

enabled bool if this test is enabled or not

init Function pointer to test init routine.

run Function pointer to test run routine.

Description:

PMBUS_TEST structure.

7.2.2 Function Documentation

7.2.2.1 PMBusMaster_initAlertTest

Initialize the "Alert" Test

Prototype:

```
void
PMBusMaster_initAlertTest(PMBus_TestHandle handle)
```

Parameters:

handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.2 void PMBusMaster_initBlockRead3BytesTest (PMBus_TestHandle *handle*)

Initialize the "Block Read (3 Bytes)" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.3 void PMBusMaster_initBlockReadTest (PMBus_TestHandle *handle*)

Initialize the "Block Read" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.4 void PMBusMaster_initBlockWrite2BytesTest (PMBus_TestHandle *handle*)

Initialize the "Block Write (2 Bytes)" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.5 void PMBusMaster_initBlockWrite3BytesTest (PMBus_TestHandle *handle*)

Initialize the "Block Write (3 Bytes)" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.6 void PMBusMaster_initBlockWriteReadProcessCallTest (PMBus_TestHandle *handle*)

Initialize the "Block Write, Block Read, Process Call" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.7 void PMBusMaster_initBlockWriteTest (PMBus_TestHandle *handle*)

Initialize the "Block Write" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.8 void PMBusMaster_initExtendedReadByteTest (PMBus_TestHandle *handle*)

Initialize the "Extended Read Byte" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.9 void PMBusMaster_initExtendedReadWordTest (PMBus_TestHandle *handle*)

Initialize the "Extended Read Word" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.10 void PMBusMaster_initExtendedWriteByteTest (PMBus_TestHandle *handle*)

Initialize the "Extended Write Byte" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.11 void PMBusMaster_initExtendedWriteWordTest (PMBus_TestHandle *handle*)

Initialize the "Extended Write Word" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.12 void PMBusMaster_initGroupCommandTest (PMBus_TestHandle *handle*)

Initialize the "Group Command" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.13 void PMBusMaster_initNoAlertTest (PMBus_TestHandle *handle*)

Initialize the "No Alert" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.14 void PMBusMaster_initProcessCallTest (PMBus_TestHandle *handle*)

Initialize the "Process Call" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.15 void PMBusMaster_initQuickCommandTest (PMBus_TestHandle *handle*)

Initialize the "Quick Command" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.16 void PMBusMaster_initReadByteTest (PMBus_TestHandle *handle*)

Initialize the "Read Byte" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.17 void PMBusMaster_initReadWordTest (PMBus_TestHandle *handle*)

Initialize the "Read Word" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.18 void PMBusMaster_initReceiveByteTest (PMBus_TestHandle *handle*)

Initialize the "Receive Byte" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.19 void PMBusMaster_initSendByteTest (PMBus_TestHandle *handle*)

Initialize the "Send Byte" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.20 void PMBusMaster_initWriteByteTest (PMBus_TestHandle *handle*)

Initialize the "Write Byte" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.21 void PMBusMaster_initWriteWordTest (PMBus_TestHandle *handle*)

Initialize the "Write Word" Test

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.22 static void PMBusMaster_resetGlobalFlags (void) `[inline, static]`

Reset the global PMBus module flags

Returns:
None.

7.2.2.23 static void PMBusMaster_resetTestObject (PMBus_TestHandle *handle*) `[inline, static]`

Reset the PMBus Stack Object Test Statues and Counts

Parameters:
handle is the handle to the PMBus stack object

Returns:
None.

7.2.2.24 void PMBusMaster_runAlertTest (PMBus_TestHandle *handle*)

Run the "Alert" Response Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 4

Returns:
None.

7.2.2.25 void PMBusMaster_runBlockRead3BytesTest (PMBus_TestHandle *handle*)

Run the "Block Read (3 Bytes)" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 6

Returns:
None.

7.2.2.26 void PMBusMaster_runBlockReadTest (PMBus_TestHandle *handle*)

Run the "Block Read" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 258

Returns:
None.

7.2.2.27 void PMBusMaster_runBlockWrite2BytesTest (PMBus_TestHandle *handle*)

Run the "Block Write (2 Bytes)" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 1

Returns:
None.

7.2.2.28 void PMBusMaster_runBlockWrite3BytesTest (PMBus_TestHandle *handle*)

Run the "Block Write (3 Bytes)" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 1

Returns:
None.

7.2.2.29 void PMBusMaster_runBlockWriteReadProcessCallTest (PMBus_TestHandle *handle*)

Run the "Block Write, Block Read, Process Call" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 258

Returns:
None.

7.2.2.30 void PMBusMaster_runBlockWriteTest (PMBus_TestHandle *handle*)

Run the "Block Write" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 1

Returns:
None.

7.2.2.31 void PMBusMaster_runExtendedReadByteTest (PMBus_TestHandle *handle*)

Run the "Extended Read Byte" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 3

Returns:
None.

7.2.2.32 void PMBusMaster_runExtendedReadWordTest (PMBus_TestHandle *handle*)

Run the "Extended Read Word" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 4

Returns:
None.

7.2.2.33 void PMBusMaster_runExtendedWriteByteTest (PMBus_TestHandle *handle*)

Run the "Extended Write Byte" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 1

Returns:
None.

7.2.2.34 void PMBusMaster_runExtendedWriteWordTest (PMBus_TestHandle *handle*)

Run the "Extended Write Word" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 1

Returns:
None.

7.2.2.35 void PMBusMaster_runGroupCommandTest (PMBus_TestHandle *handle*)

Run the "Group Command" Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 4

Returns:
None.

7.2.2.36 void PMBusMaster_runNoAlertTest (PMBus_TestHandle *handle*)

Run the "No Alert" Response Test

Parameters:
handle is the handle to the PMBus stack object

Note:
This test requires a 2nd PMBus slave on the network asserting the alert line.

Expected pass value: 5

Returns:
None.

7.2.2.37 void PMBusMaster_runProcessCallTest (PMBus_TestHandle *handle*)

Run the "Process Call" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 4

Returns:
None.

7.2.2.38 void PMBusMaster_runQuickCommandTest (PMBus_TestHandle *handle*)

Run the "Quick Command" Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 1

Returns:
None.

7.2.2.39 void PMBusMaster_runReadByteTest (PMBus_TestHandle *handle*)

Run the "Read Byte" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 3

Returns:
None.

7.2.2.40 void PMBusMaster_runReadWordTest (PMBus_TestHandle *handle*)

Run the "Read Word" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 4

Returns:
None.

7.2.2.41 void PMBusMaster_runReceiveByteTest (PMBus_TestHandle *handle*)

Run the "Receive Byte" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 2

Returns:
None.

7.2.2.42 void PMBusMaster_runSendByteTest (PMBus_TestHandle *handle*)

Run the "Send Byte" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 1

Returns:
None.

7.2.2.43 void PMBusMaster_runWriteByteTest (PMBus_TestHandle *handle*)

Run the "Write Byte" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 1

Returns:
None.

7.2.2.44 void PMBusMaster_runWriteWordTest (PMBus_TestHandle *handle*)

Run the "Write Word" Command Test

Parameters:
handle is the handle to the PMBus stack object

Note:
Expected pass value: 1

Returns:
None.

7.3 PMBus Examples Setup Code

Functions

- void done (void)
- void PMBusExample_setupFlash (void)
- void PMBusExample_setupGPIO (void)
- void PMBusExample_setupInterrupts (void (*pmbusISR)(void))
- void PMBusExample_setupSysCtrl (void)

7.3.1 Function Documentation

7.3.1.1 done

Done Function

Prototype:

```
void
done(void)
```

Description:
This function is an infinite loop which is run at the end of testing.

Returns:
None.

7.3.1.2 PMBusExample_setupFlash

Setup Flash

Prototype:

```
void
PMBusExample_setupFlash(void)
```

Description:
This function initializes the flash module.

Returns:
None.

7.3.1.3 PMBusExample_setupGPIO

Setup GPIO pins for PMBUS mode of operation

Prototype:

```
void
PMBusExample_setupGPIO(void)
```

Description:
This function configures the GPIO muxing for PMBus.

Returns:
None.

7.3.1.4 PMBusExample_setupInterrupts

Setup Interrupts

Prototype:
```
void
PMBusExample_setupInterrupts(void (*pmbusISR)(void))
```

Parameters:
pmbusISR is the handle to the PMBus ISR function

Description:
This function enables device and PMBus interrupts. Additionally, the PMBus ISR handler is registered.

Returns:
None.

7.3.1.5 PMBusExample_setupSysCtrl

Setup System Control

Prototype:
```
void
PMBusExample_setupSysCtrl(void)
```

Description:
This function disables the watchdog, enables device PLL, and PMBus peripheral clock.

Returns:
None.

8 PMBus Other Examples

This chapter describes PMBus examples that are "peripheral only" and don’t make use of the library PMBus slave communication stack code.

8.1 Example Descriptions

PMBus (I2C mode) EEPROM Example

This example shows how to use PMBus peripheral in I2C mode to read / write to EEPROM in different read / write modes. Different modes supported:-

- EEPROM_byteWrite - Used to write a byte of data in specified EEPROM address
- EEPROM_PageWrite - Used to write a specified number of bytes from data pointer into the specified address
- EEPROM_CurrentAddress_Read - Used to read a byte referenced by current EEPROM internal address pointer
- EEPROM_SequentialRead - Used to read data from EEPROM sequentially of specified size
- EEPROM_RandomRead - Used to read upto 4 bytes of data of data sequentially from a given EEPROM address

External Connections

- Connect external I2C EEPROM at slave address 0x50
- Connect GPIO25/PMBus Data to external EEPROM SDA (serial data) pin
- Connect GPIO24/PMBus CLK to external EEPROM SCL (serial clock) pin

Watch Variables

- eepromStatus - Used as a flag variable to check whether EEPROM drivers executed correctly
- txbuf - Transmit buffer used to send the data to be programmed in EEPROM
- rxbuf - Receive buffer used to store the data read from EEPROM

Note the EEPROM used for this example is AT24C256 (http://www.ti.com/lit/an/slaa208a/slaa208a.pdf)

Note this examples doesn’t use the PMBus slave communication stack.

9 Revision History

v1.03.00.00

- New PMBus read/write to EEPROM example for F2838x and F28004x devices
- Rebuilt libraries and examples using CCSv10 and v20 compiler

v1.02.01.00

- User documentation updates to reflect PMBus specification 1.2 support
- General user documentation cleanup

v1.02.00.00

- F28002x PMBus master and slave examples added
- Library updated to include build configurations for F28002x

v1.01.00.00

- F2838x PMBus master and slave examples added
- Library API naming, formating, etc updated. (Library functionality is unchanged)
- Library compatibility header file provided for previous F28004x library users
- Examples re-structured and updated to use CCS projectspecs

v1.00.00.00

- Library stack supports slave only mode of operation
- Master and slave examples demonstrating slave stack library

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265