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 V1.00.00.00 of this document, last updated on Oct 18, 2018.

Table of Contents

Copyright 2
Revision Information 2
1 Introduction 5
2 Other Resources 7
3 Library Structure 8
3.1 Build Options used to build the library 9
4 Using the PMBus Communications Stack Library 10
4.1 Integrating the Library into your Project 10
5 The PMBus Protocol 15
5.1 PMBus Slave Mode 15
5.1.1 Packet Error Checking 15
5.2 Slave Mode Message Types 16
5.2.1 Quick Command 17
5.2.2 Send Byte 17
5.2.3 Write Byte 18
5.2.4 Write Word 18
5.2.5 Block Write 18
5.2.6 Receive Byte 19
5.2.7 Alert Response 19
5.2.8 Read Byte 19
5.2.9 Read Word 20
5.2.10 Block Read 20
5.2.11 Block Write/ Read/ Process Call 20
5.2.12 Group Command 21
5.2.13 Extended Command 21
5.3 State Machine Description 23
5.3.1 The Idle State 25
5.3.2 The Receive Byte and Wait for End-of-Message State 26
5.3.3 The Read Block State 27
5.3.4 The Read and Wait for End-of-Message State 28
5.3.5 The Block Write or Process Call State 29
5.3.6 The Extended Command State 30
6 Application Programming Interface (PMBus Communications Stack) 31
6.1 Code Development with Assertion 32
6.1.1 Define Documentation 32
6.1.2 Function Documentation 32
6.1.3 Variable Documentation 33
6.2 PMBus Configuration 34
6.2.1 Data Structure Documentation 35
6.2.2 Enumeration Documentation 37
6.2.3 Function Documentation 37
6.2.4 Variable Documentation 44
6.3 PMBus State Machine Handler 45
6.3.1 Function Documentation 45
7 Revision History 47
IMPORTANT NOTICE 48

1 Introduction

The PMBus Communications Software Stack is designed to operate on the PMBus Type-0 hardware peripheral module on the F28004x. In its current state it works in slave mode only, with support for the following transaction types:

1. Quick Command
2. Send Byte
3. Write Byte
4. Write Word
5. Block Write
6. Receive Byte
7. Read Byte
8. Read Word
9. Block Read
10. Block Write/ Read/ Process Call

Each transaction type has an associated callback function that must be defined by the end user. The **Alert Response** feature is a special variant of the Receive Byte transaction and, therefore, has the same callback. The software stack also has the ability to respond to group commands and the extended read/write byte/word.

This document describes the interface provided to the end-user and the way in which it is to be used. The current version of this library works on the following device(s):

1. F28004x

Chapter 2 provides a host of resources on the PMBus in general, as well as device specific training material.

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 talks 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 provides a revision history of the library.

There are examples that show both the slave and master implementations for the PMBus module. The example for the slave side implementation uses the communications stack, while the master side example issues a series of different transactions to test the communications channel. They can be found in the *examples* directory. For the current revision of the library, all examples have been written for the *F28004x* device and tested on a *controlCard* platform. Each example has a script **“SetupDebugEnv.js”** that can be launched from the *Scripting Console* in CCS. These scripts will set-up the watch variables for the example.

2 Other Resources

The user can get answers to F28004x frequently asked questions(FAQ) from the processors wiki page. Links to other references such as training videos will be posted here as well.
http://processors.wiki.ti.com/index.php/Main_Page.

Also check out the TI Piccolo page: http://www.ti.com/piccolo

And don't forget the TI community website: http://e2e.ti.com

Building the PMBus Communications Stack library and examples requires **Codegen Tools v6.4.4 or later**.

3 Library Structure

Build Options used to build the library 9

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

```
C:\ti\c2000\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.

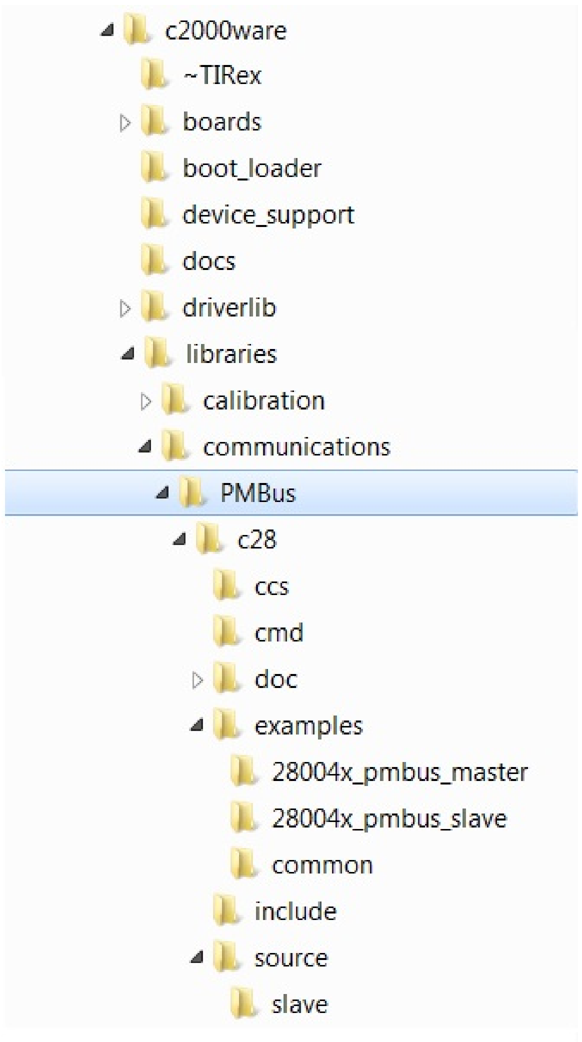


Figure 3.1: Directory Structure of the Library

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>/cmd	Linker command files used in the examples.
<base>/doc	Documentation for the current revision of the library including revision history.
<base>/examples	Examples that illustrate use of the library. At the time of writing these examples were built for the F28004x device using the CCS 6.0.0.00190 IDE.
<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

3.1 Build Options used to build the library

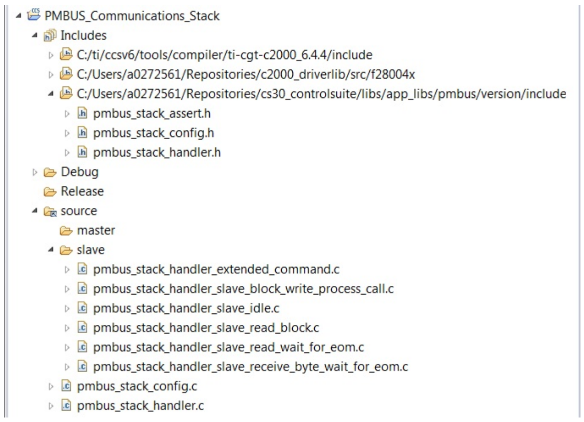
The current version of the library was built with C28x Codegen Tools v6.4.4 with the following options:

```
TODO
-v28 -ml -mt -O2 --diag_warning=225 --display_error_number --diag_wrap=off
```

4 Using the PMBus Communications Stack Library

Integrating the Library into your Project 10

The source code and project(s) for the library are provided. The user may import the library project(s) into CCSv6 (or later) and be able to view and modify the source code for all routines and lookup tables (see Fig. 4.1)



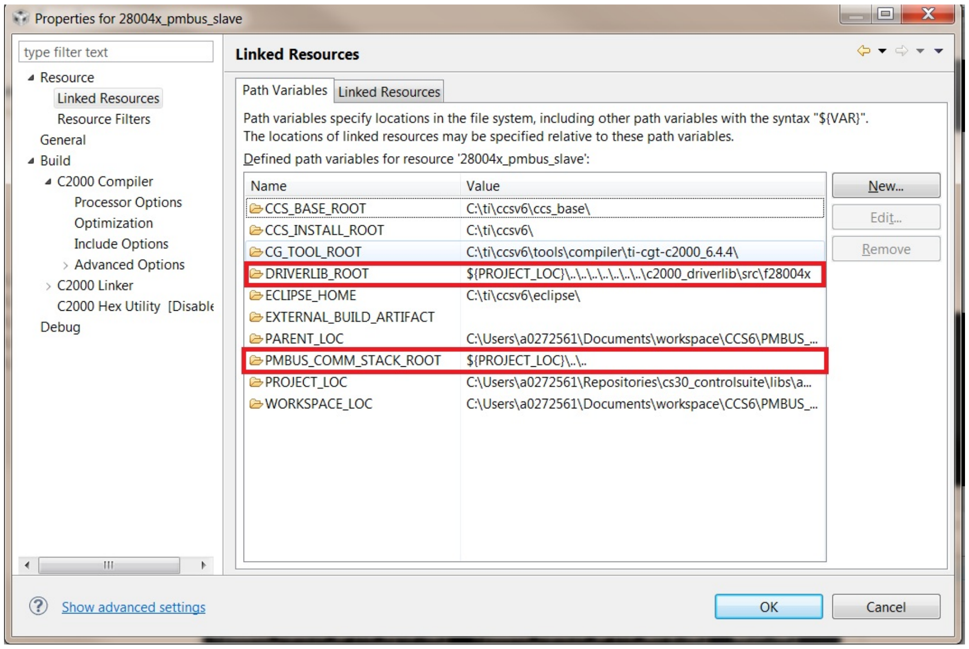
```
PMBUS_Communications_Stack
  Includes
    C:/ti/ccsv6/tools/compiler/ti-cgt-c2000_6.4.4/include
    C:/Users/a0272561/Repositories/c2000_driverlib/src/f28004x
    C:/Users/a0272561/Repositories/cs30_controlsuite/libs/app_libs/pmbus/version/include
      pmbus_stack_assert.h
      pmbus_stack_config.h
      pmbus_stack_handler.h
  Debug
  Release
  source
    master
    slave
      pmbus_stack_handler_extended_command.c
      pmbus_stack_handler_slave_block_write_process_call.c
      pmbus_stack_handler_slave_idle.c
      pmbus_stack_handler_slave_read_block.c
      pmbus_stack_handler_slave_read_wait_for_eom.c
      pmbus_stack_handler_slave_receive_byte_wait_for_eom.c
    pmbus_stack_config.c
    pmbus_stack_handler.c
```

Figure 4.1: 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->Build->Variables(Tab)** and add a new variable (see Fig. 4.2), **DRIVERLIB_ROOT**, and set its path to the root directory of the F28004x driver library. Set up another variable, **PMBUS_COMM_STACK_ROOT** to point to the v1_00_00_00_B folder of the PMBus Communications Stack Library.



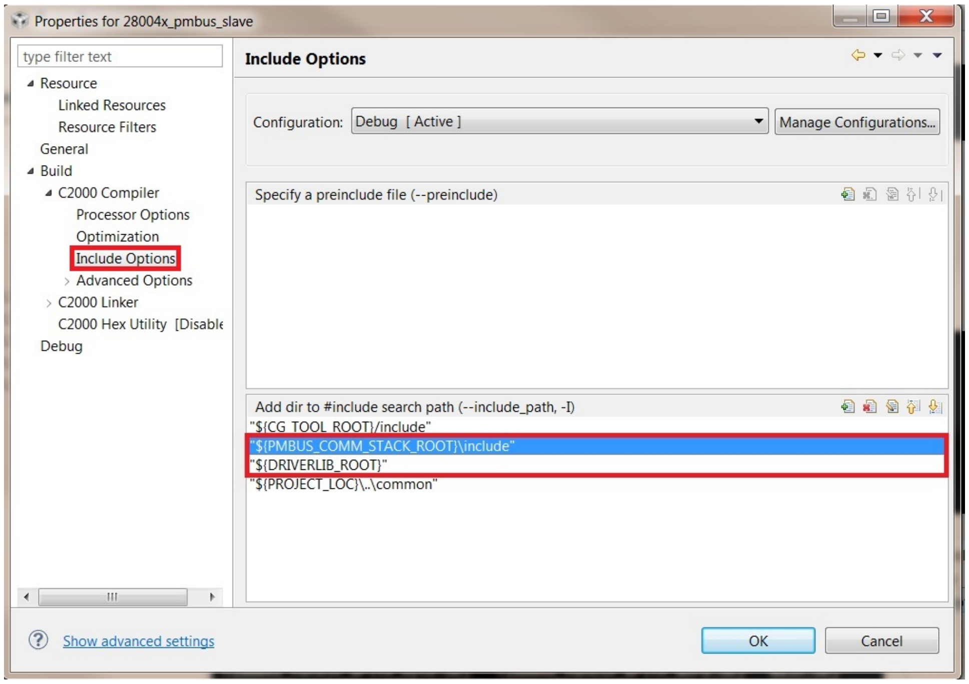
Path variables specify locations in the file system, including other path variables with the syntax "${VAR}".
The locations of linked resources may be specified relative to these path variables.

Defined path variables for resource '28004x_pmbus_slave':

Name	Value
CCS_BASE_ROOT	C:\ti\ccsv6\ccs_base\
CCS_INSTALL_ROOT	C:\ti\ccsv6\
CG_TOOL_ROOT	C:\ti\ccsv6\tools\compiler\ti-cgt-c2000_6.4.4\
DRIVERLIB_ROOT	${PROJECT_LOC}\..\..\..\..\..\..\..\c2000_driverlib\src\f28004x
ECLIPSE_HOME	C:\ti\ccsv6\eclipse\
EXTERNAL_BUILD_ARTIFACT	
PARENT_LOC	C:\Users\a0272561\Documents\workspace\CCS6\PMBUS_...
PMBUS_COMM_STACK_ROOT	${PROJECT_LOC}\..\..
PROJECT_LOC	C:\Users\a0272561\Repositories\cs30_controlsuite\libs\a...
WORKSPACE_LOC	C:\Users\a0272561\Documents\workspace\CCS6\PMBUS_...

Figure 4.2: Creating a new build variable

Add the new path, **DRIVERLIB_ROOT**, to the *Include Options* section of the project properties (Fig. 4.3). This option tells the compiler where to find the device driver header files. In addition, you must add the **PMBUS_COMM_STACK_ROOT** path for the PMBus stack interface, for the compiler to find the stack header files.



Configuration: Debug [Active]

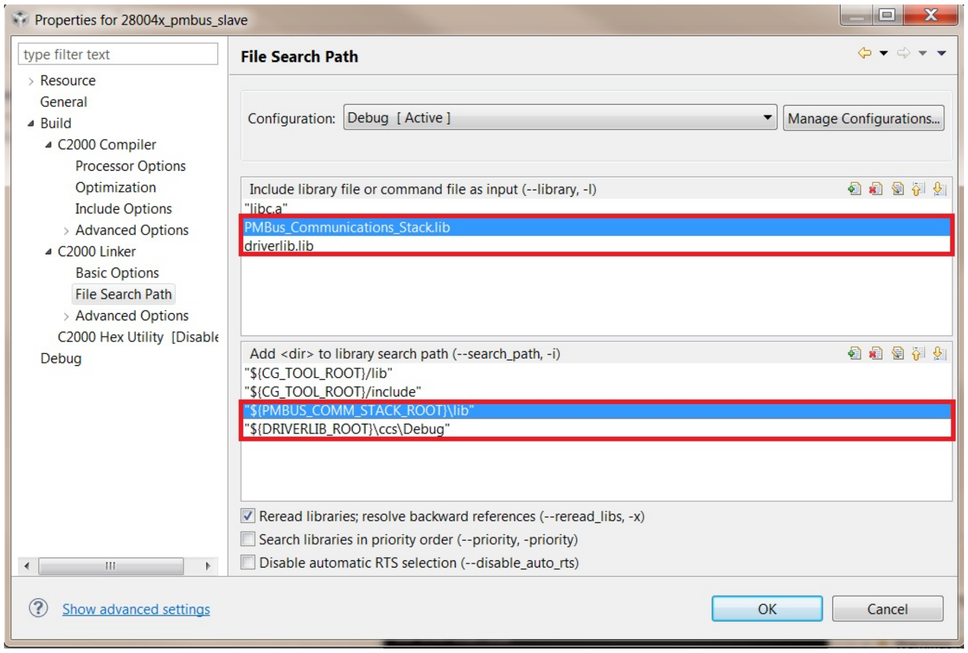
Specify a preinclude file (--preinclude)

Add dir to #include search path (--include_path, -I)

```
"${CG_TOOL_ROOT}/include"
"${PMBUS_COMM_STACK_ROOT}\include"
"${DRIVERLIB_ROOT}"
"${PROJECT_LOC}\..\common"
```

Figure 4.3: Adding the Library Header Path to the Include Options

2. Add the Communications Stack library, **'PMBus_Communications_Stack.lib', and its location to the File Search Path as shown in Fig. 4.4. Also add the device driver library.**



Configuration: Debug [Active]

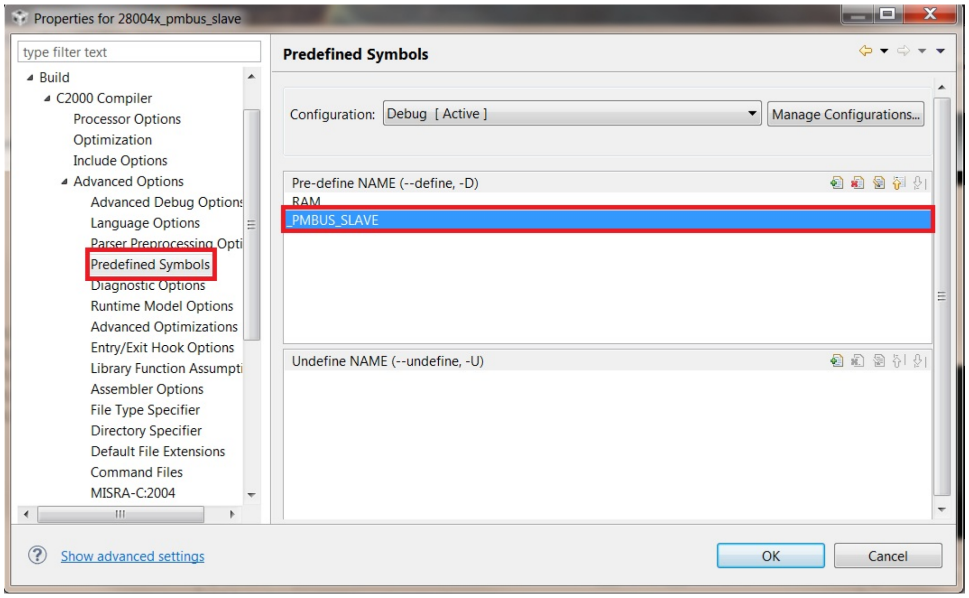
Include library file or command file as input (--library, -l)
"libc.a"
PMBus_Communications_Stack.lib
driverlib.lib

Add <dir> to library search path (--search_path, -i)
"${CG_TOOL_ROOT}/lib"
"${CG_TOOL_ROOT}/include"
"${PMBUS_COMM_STACK_ROOT}\lib"
"${DRIVERLIB_ROOT}\ccs\Debug"

☑ Reread libraries; resolve backward references (--reread_libs, -x)
☐ Search libraries in priority order (--priority, -priority)
☐ Disable automatic RTS selection (--disable_auto_rts)

Figure 4.4: 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. 4.5.**



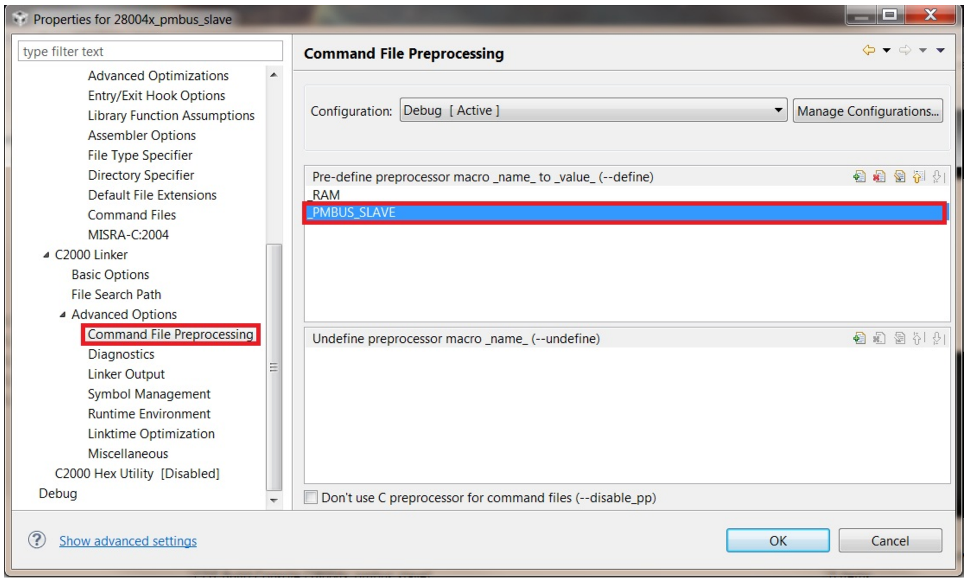
Configuration: Debug [Active]

Pre-define NAME (--define, -D)
RAM
_PMBUS_SLAVE

Undefine NAME (--undefine, -U)

Figure 4.5: Compiler Predefined Symbols

Additionally, define _PMBUS_SLAVE in the *Command File Preprocessing* option under the C2000 Linker menu, Fig. 4.6.



Configuration: Debug [Active]

Pre-define preprocessor macro _name_ to _value_ (--define)
RAM
_PMBUS_SLAVE

Undefine preprocessor macro _name_ (--undefine)

☐ Don't use C preprocessor for command files (--disable_pp)

Figure 4.6: Linker Preprocessor Macros

NOTE:FOR DEVICES THAT OPERATE IN MASTER MODE THE USER MUST DEFINE THE SYMBOL _PMBUS_MASTER INSTEAD.

These symbols are used in the stack initialization code to configure the hardware module to operate in slave (or master) mode. The preprocessor macro is used in the linker command file to conditionally allocate memory and sections for use in slave or

master mode; this allows a single linker command file to be defined for use in both modes, and across master/slave CCS projects.

NOTE: THE PREPROCESSOR MACRO IS NOT USED IN THE LINKER COMMAND FILE AS OF THIS RELEASE. ITS USE IS RESERVED FOR FUTURE RELEASES, AND MUST BE DEFINED IN THE PROJECT SETTINGS, UNDER THE C2000 LINKER FIELD AS DESCRIBED ABOVE.

5 The PMBus Protocol

PMBus Slave Mode 15
Slave Mode Message Types 16
State Machine Description 23

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.

The PMBus module supports **Standard, Fast**, and **Fast Plus** modes of operation, and conforms to v1.0 of the PMBus Specification (Part I) and v1.1 of the PMBus Specification (Part II). The software also conforms to these standards.

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,

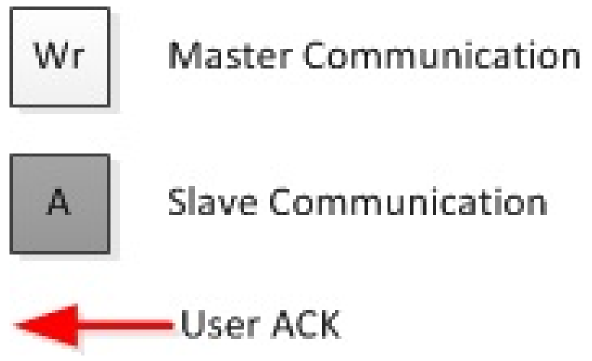


Figure 5.1: Message Format Legend

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.

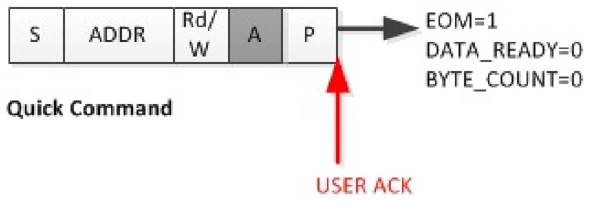


Figure 5.2: Quick Command

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.

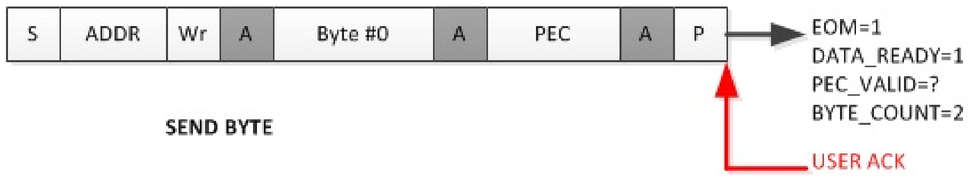


Figure 5.3: Send Byte

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.

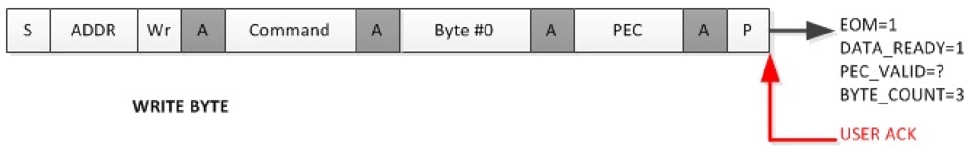


Figure 5.4: Write Byte

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.

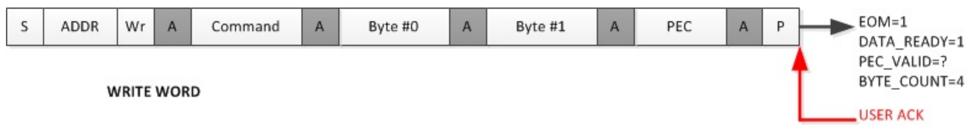


Figure 5.5: Write Word

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.

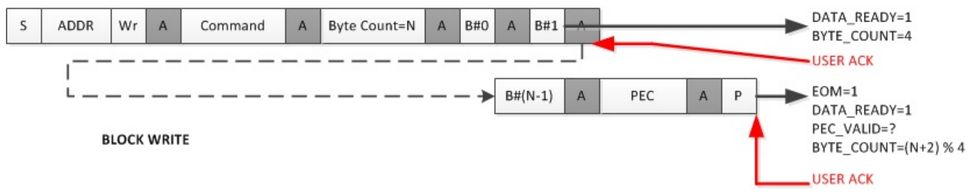


Figure 5.6: Block Write

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.

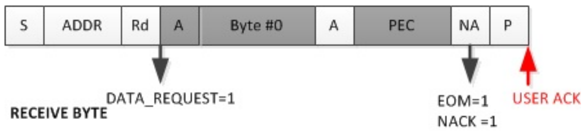


Figure 5.7: Receive Byte

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.

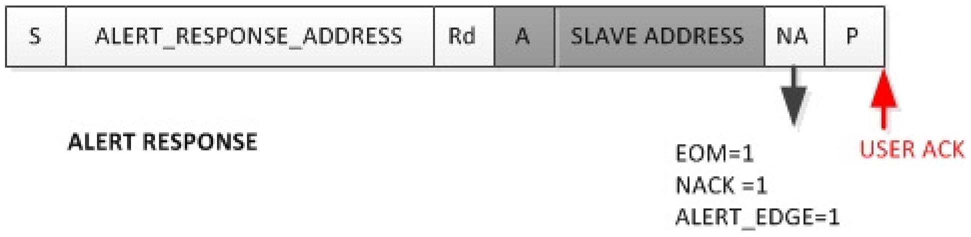


Figure 5.8: Alert Responsed

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.

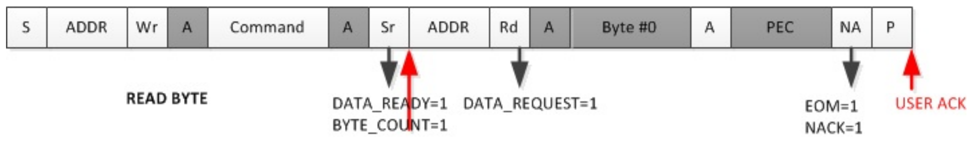


Figure 5.9: Read Byte

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.

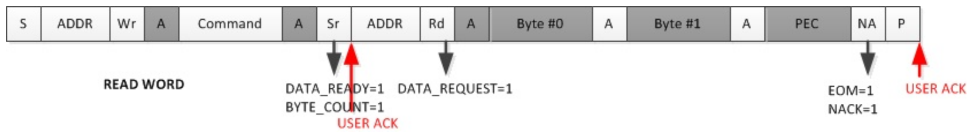


Figure 5.10: Read Word

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.

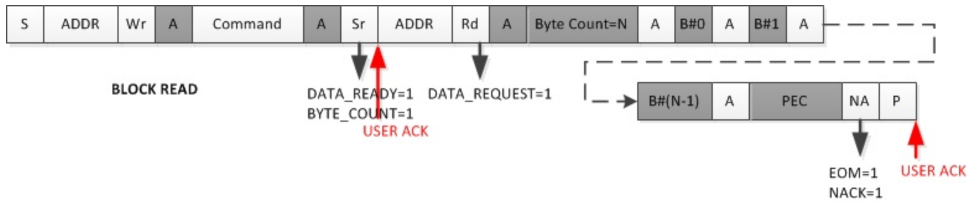


Figure 5.11: Block Read

5.2.11 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.

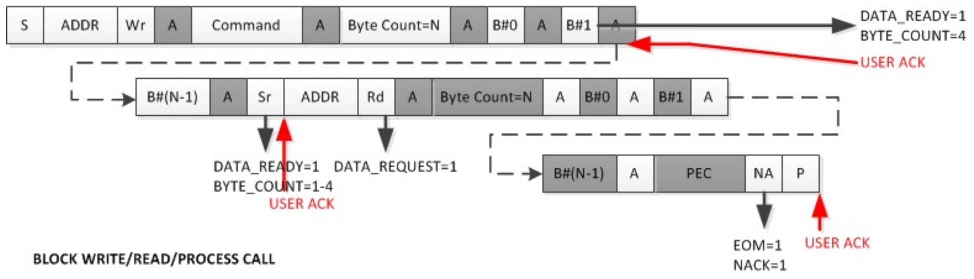


Figure 5.12: Block Write/ Read/ Process Call

NOTE: A PROCESS CALL, A WRITE WORD FOLLOWED BY A READ WORD, FALLS UNDER THE PURVIEW OF THIS TRANSACTION IN THE STATE MACHINE HANDLER

5.2.12 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).

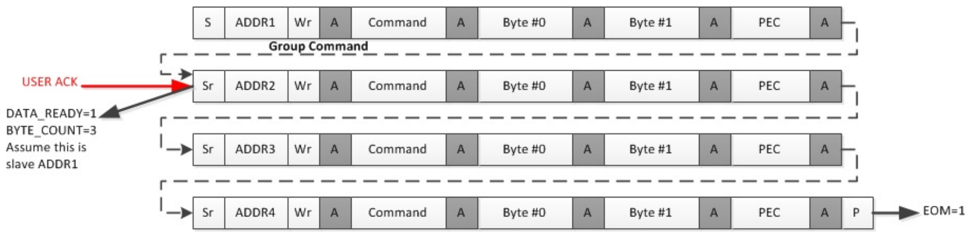


Figure 5.13: Group Command

5.2.13 Extended Command

The extended command is supported for four transaction types

1. Extended Read Byte
2. Extended Read Word
3. Extended Write Byte
4. Extended Write Word

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.

Note: This behavior conforms to v1.0 of the PMBus Protocol; v1.2 of the protocol does not require a repeated start (and slave address) after the extension and command bytes are sent

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).

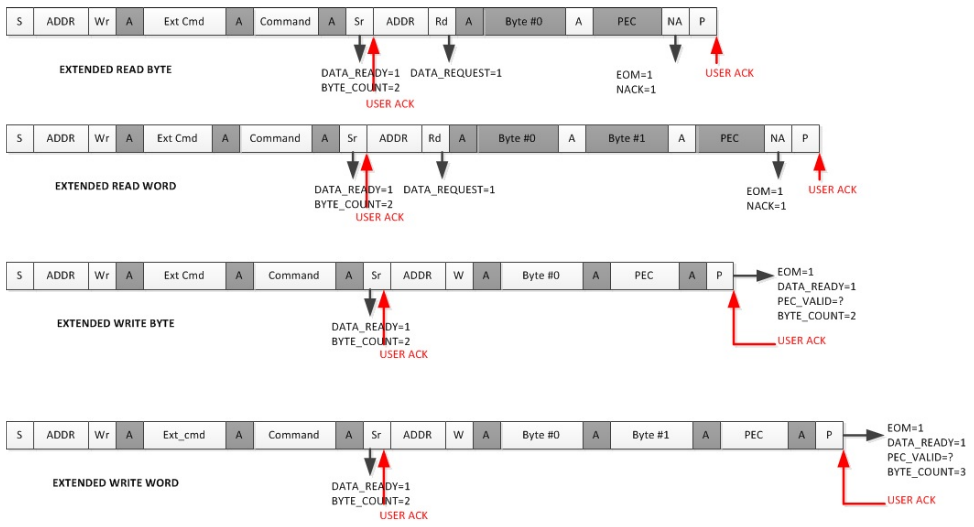


Figure 5.14: Extended Commands

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 *PMBus_Stack_slaveIdleHandler*.

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 *PMBus_Stack_slaveReceiveByteWaitForEomHandler*.

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 *PMBus_Stack_slaveReadBlockHandler*.

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 *PMBus_Stack_slaveReceiveByteWaitForEomHandler*.

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 *PMBus_Stack_slaveBlockWriteOrProcessCallHandler*.

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 *PMBus_Stack_extendedCommandHandler*.

The transition from one state to the next is depicted in Fig. 5.15,

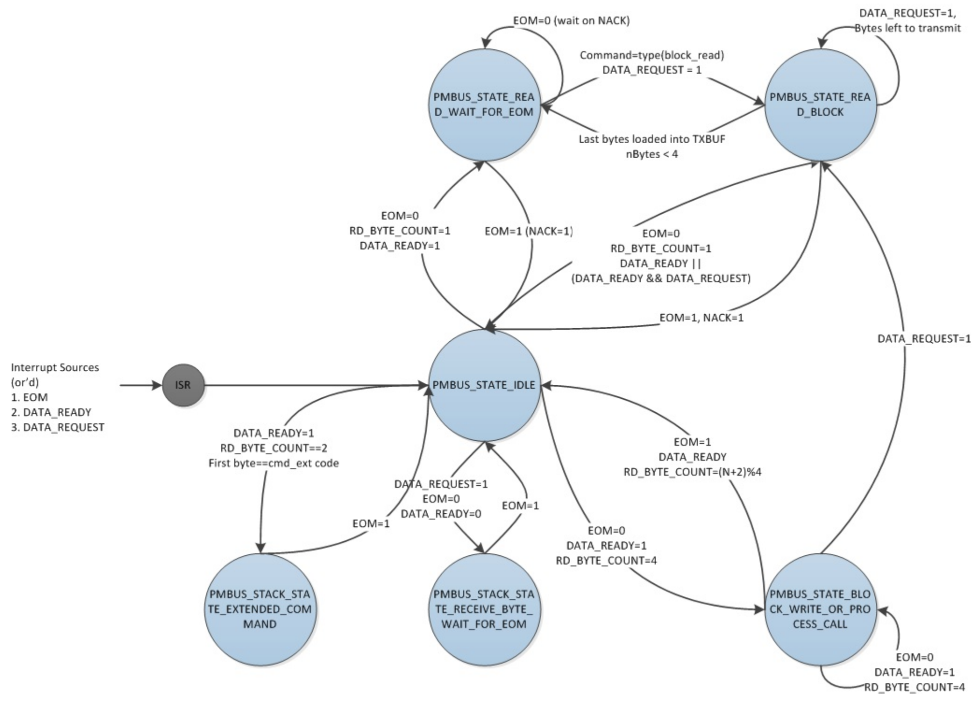

EOM=0 (wait on NACK)
DATA_REQUEST=1, Bytes left to transmit
Command=type(block_read) DATA_REQUEST = 1
PMBUS_STATE_READ_WAIT_FOR_EOM
PMBUS_STATE_READ_BLOCK
Last bytes loaded into TXBUF nBytes < 4
EOM=0 RD_BYTE_COUNT=1 DATA_READY=1
EOM=1 (NACK=1)
EOM=0 RD_BYTE_COUNT=1 DATA_READY || (DATA_READY && DATA_REQUEST)
EOM=1, NACK=1
DATA_REQUEST=1
Interrupt Sources (or'd) 1. EOM 2. DATA_READY 3. DATA_REQUEST
ISR
PMBUS_STATE_IDLE
DATA_READY=1 RD_BYTE_COUNT==2 First byte==cmd_ext code
DATA_REQUEST=1 EOM=0 DATA_READY=0
EOM=1
EOM=1
EOM=1 DATA_READY RD_BYTE_COUNT=(N+2)%4
EOM=0 DATA_READY=1 RD_BYTE_COUNT=4
PMBUS_STACK_STATE_EXTENDED_COMMAND
PMBUS_STACK_STATE_RECEIVE_BYTE_WAIT_FOR_EOM
PMBUS_STATE_BLOCK_WRITE_OR_PROCESS_CALL
EOM=0 DATA_READY=1 RD_BYTE_COUNT=4

Figure 5.15: 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

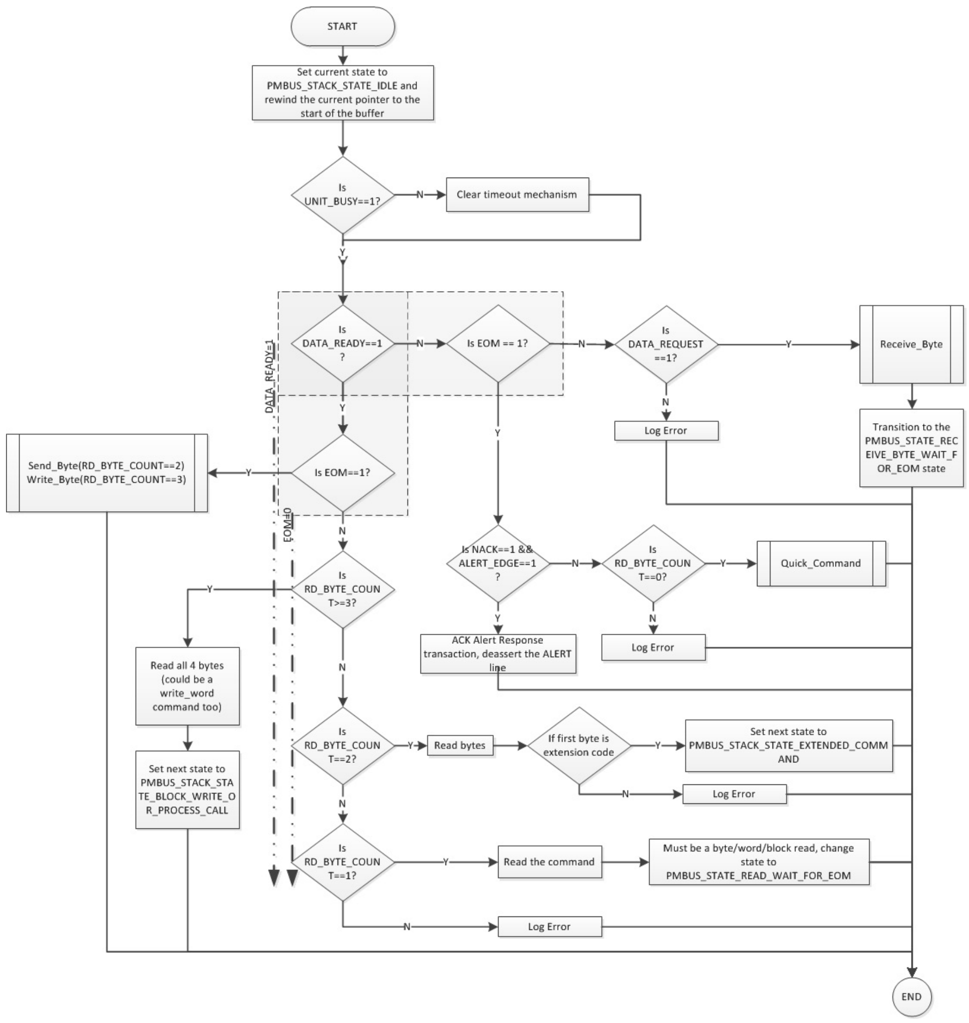

START
Set current state to PMBUS_STACK_STATE_IDLE and rewind the current pointer to the start of the buffer
Is UNIT_BUSY==1?
Clear timeout mechanism
Is DATA_READY==1 ?
Is EOM == 1?
Is DATA_REQUEST ==1?
Receive_Byte
Transition to the PMBUS_STATE_RECEIVE_BYTE_WAIT_FOR_EOM state
Log Error
DATA_READY=1
Is EOM==1?
Send_Byte(RD_BYTE_COUNT==2) Write_Byte(RD_BYTE_COUNT==3)
EOM=0
Is NACK==1 && ALERT_EDGE==1 ?
Is RD_BYTE_COUNT==0?
Quick_Command
ACK Alert Response transaction, deassert the ALERT line
Log Error
Is RD_BYTE_COUNT>=3?
Read all 4 bytes (could be a write_word command too)
Set next state to PMBUS_STACK_STATE_BLOCK_WRITE_OR_PROCESS_CALL
Is RD_BYTE_COUNT==2?
Read bytes
If first byte is extension code
Set next state to PMBUS_STACK_STATE_EXTENDED_COMMAND
Log Error
Is RD_BYTE_COUNT==1?
Read the command
Must be a byte/word/block read, change state to PMBUS_STATE_READ_WAIT_FOR_EOM
Log Error
END

Figure 5.16: 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

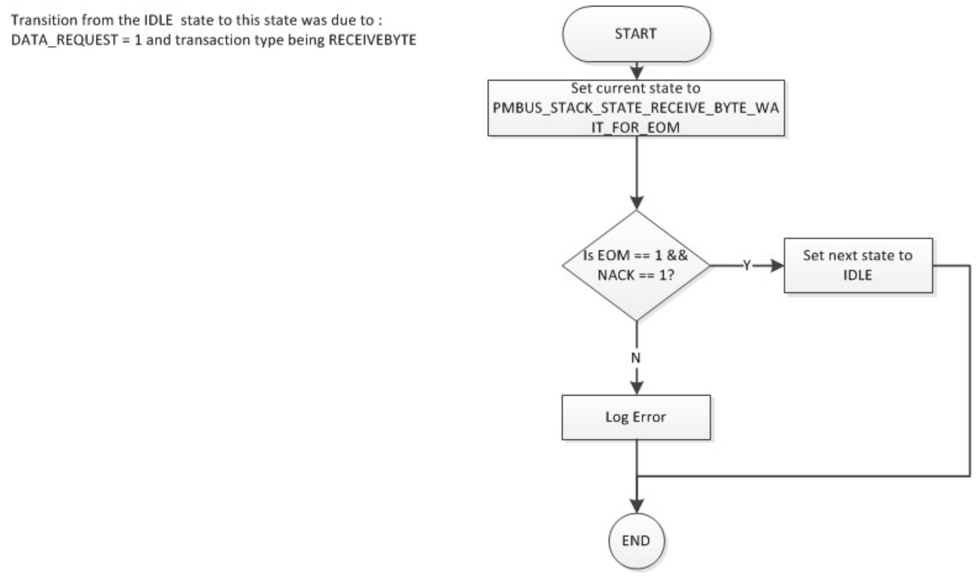

Transition from the IDLE state to this state was due to :
DATA_REQUEST = 1 and transaction type being RECEIVEBYTE
START
Set current state to PMBUS_STACK_STATE_RECEIVE_BYTE_WAIT_FOR_EOM
Is EOM == 1 && NACK == 1?
Y
Set next state to IDLE
N
Log Error
END

Figure 5.17: 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.

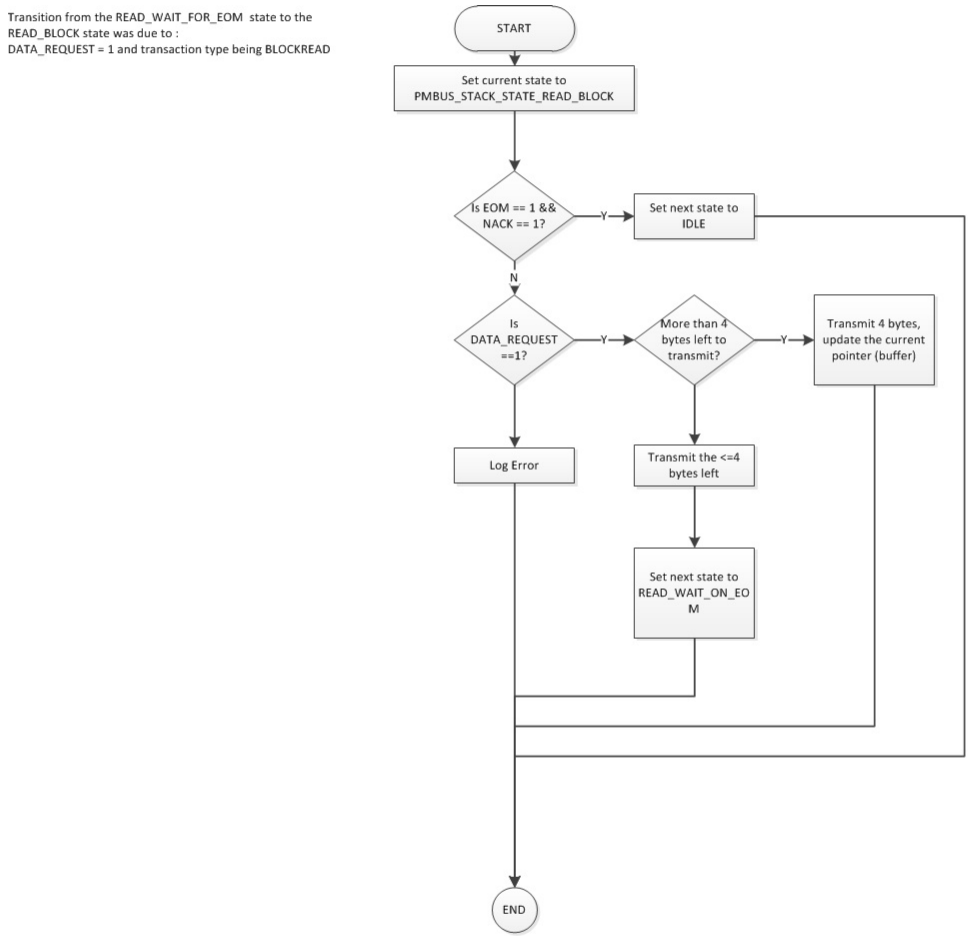
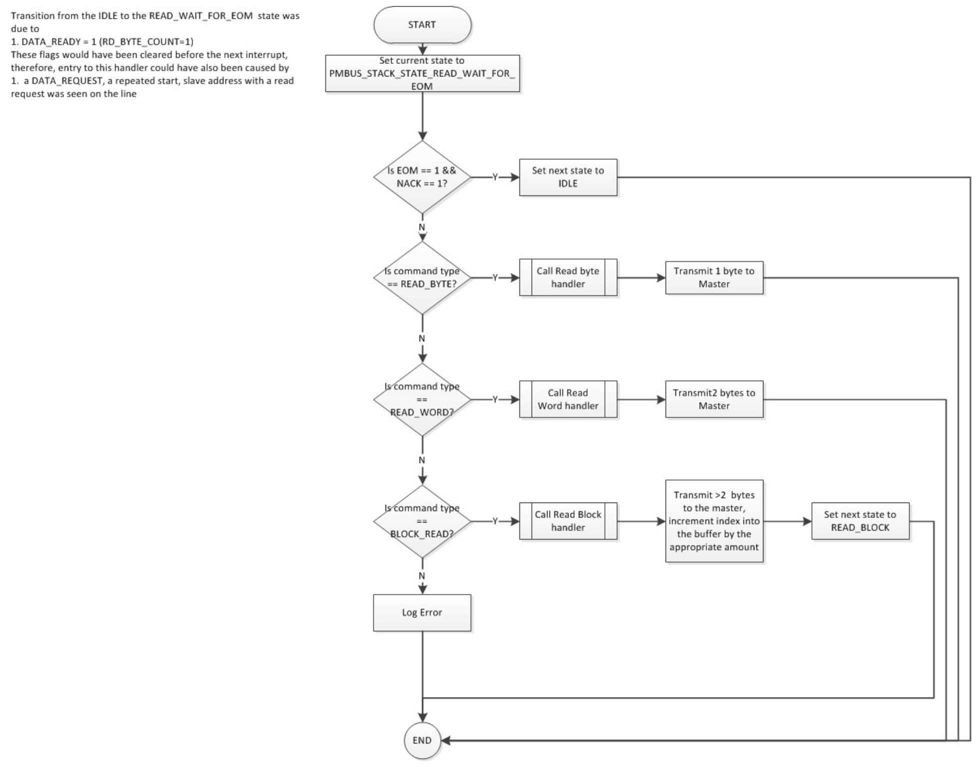

Transition from the READ_WAIT_FOR_EOM state to the READ_BLOCK state was due to : DATA_REQUEST = 1 and transaction type being BLOCKREAD
START
Set current state to PMBUS_STACK_STATE_READ_BLOCK
Is EOM == 1 && NACK == 1?
Y
Set next state to IDLE
N
Is DATA_REQUEST ==1?
Y
More than 4 bytes left to transmit?
Y
Transmit 4 bytes, update the current pointer (buffer)
Log Error
Transmit the <=4 bytes left
Set next state to READ_WAIT_ON_EOM
END

Figure 5.18: 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.

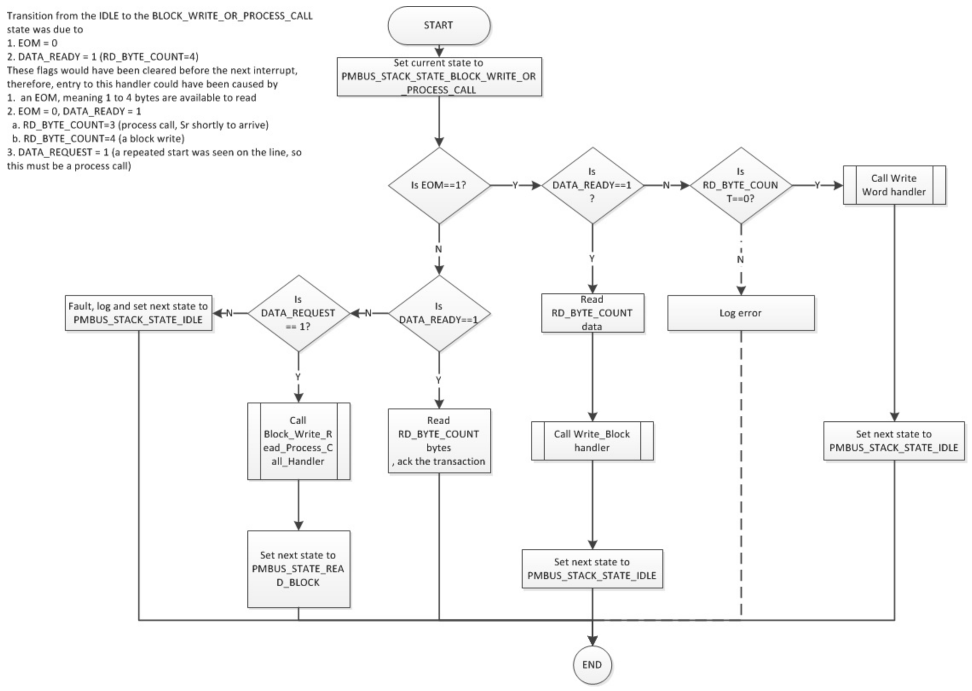

Transition from the IDLE to the READ_WAIT_FOR_EOM state was due to
1. DATA_READY = 1 (RD_BYTE_COUNT=1)
These flags would have been cleared before the next interrupt, therefore, entry to this handler could have also been caused by
1. a DATA_REQUEST, a repeated start, slave address with a read request was seen on the line

START
Set current state to PMBUS_STACK_STATE_READ_WAIT_FOR_EOM
Is EOM == 1 && NACK == 1?
Y
Set next state to IDLE
N
Is command type == READ_BYTE?
Y
Call Read byte handler
Transmit 1 byte to Master
N
Is command type == READ_WORD?
Y
Call Read Word handler
Transmit2 bytes to Master
N
Is command type == BLOCK_READ?
Y
Call Read Block handler
Transmit >2 bytes to the master, increment index into the buffer by the appropriate amount
Set next state to READ_BLOCK
N
Log Error
END

Figure 5.19: 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.



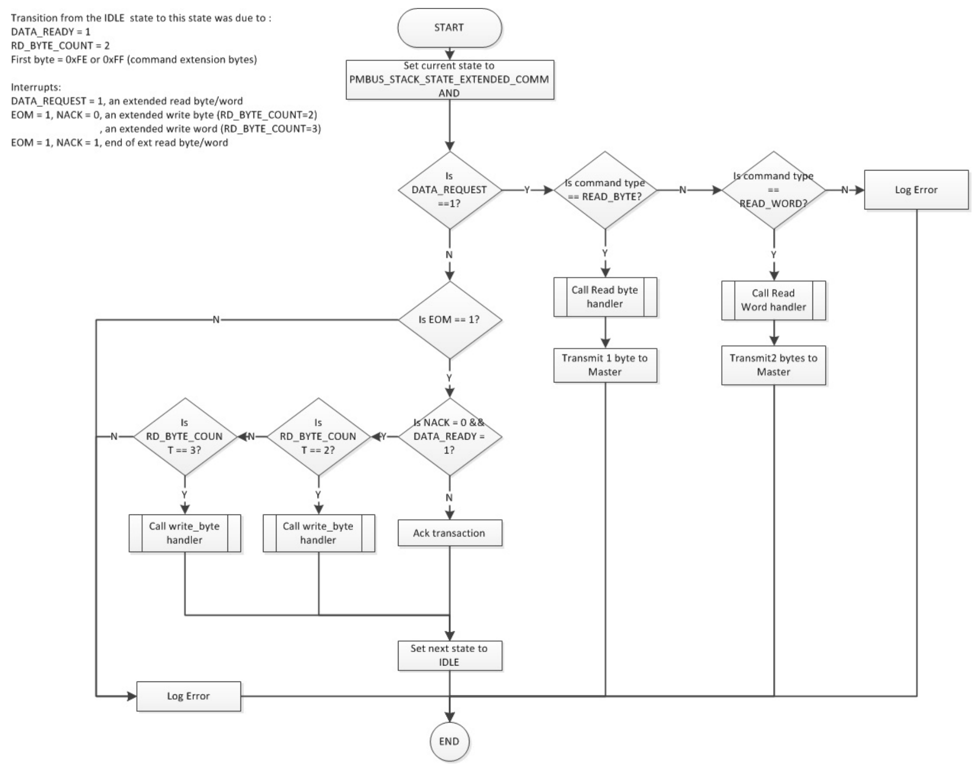
Transition from the IDLE to the BLOCK_WRITE_OR_PROCESS_CALL state was due to
1. EOM = 0
2. DATA_READY = 1 (RD_BYTE_COUNT=4)
These flags would have been cleared before the next interrupt, therefore, entry to this handler could have been caused by
1. an EOM, meaning 1 to 4 bytes are available to read
2. EOM = 0, DATA_READY = 1
  a. RD_BYTE_COUNT=3 (process call, Sr shortly to arrive)
  b. RD_BYTE_COUNT=4 (a block write)
3. DATA_REQUEST = 1 (a repeated start was seen on the line, so this must be a process call)

START
Set current state to PMBUS_STACK_STATE_BLOCK_WRITE_OR_PROCESS_CALL
Is EOM==1?
Y
Is DATA_READY==1?
N
Is RD_BYTE_COUNT==0?
Y
Call Write Word handler
N
Y
N
Fault, log and set next state to PMBUS_STACK_STATE_IDLE
N
Is DATA_REQUEST == 1?
N
Is DATA_READY==1
Read RD_BYTE_COUNT data
Log error
Y
Y
Call Block_Write_Read_Process_Call_Handler
Read RD_BYTE_COUNT bytes , ack the transaction
Call Write_Block handler
Set next state to PMBUS_STACK_STATE_IDLE
Set next state to PMBUS_STATE_READ_BLOCK
Set next state to PMBUS_STACK_STATE_IDLE
END

Figure 5.20: 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).



```
Transition from the IDLE  state to this state was due to :
DATA_READY = 1
RD_BYTE_COUNT = 2
First byte = 0xFE or 0xFF (command extension bytes)

Interrupts:
DATA_REQUEST = 1, an extended read byte/word
EOM = 1, NACK = 0, an extended write byte (RD_BYTE_COUNT=2)
                   , an extended write word (RD_BYTE_COUNT=3)
EOM = 1, NACK = 1, end of ext read byte/word
```

Figure 5.21: The Extended Command State

6 Application Programming Interface (PMBus Communications Stack)

Code Development with Assertion 32
PMBus Configuration 34
PMBus State Machine Handler 45

Each example has a script **SetupDebugEnv.js** that can be used with the scripting console in CCS to setup the watch windows and graphs automatically in the debug session. Please see CCS4:Scripting Console for more information

6.1 Code Development with Assertion

Defines

- PMBUS_STACK_ASSERT(expr)
- PMBUS_STACK_FILENUM(number)

Functions

- static void PMBus_Stack_assertionFailed (int16_t file, int16_t line)

Variables

- void(∗) PMBus_Stack_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 PMBus_Stack_assertionFailed (int16_t *file*, int16_t *line*) `[inline, static]`

Handles failed assertions.

Parameters:

← ***file*** File number where the assertion failed
← ***line*** Line number where the assertion failed

6.1.3 Variable Documentation

6.1.3.1 void(∗) PMBus_Stack_errorHandler(void)

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.

Description:
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_Stack_Obj_
- _PMBus_Transaction_Obj_
- PMBus_Transaction_Obj_u

Enumerations

- PMBus_Stack_mode
- PMBus_Stack_State

Functions

- int32_t PMBus_Stack_defaultTransactionHandler (PMBus_Stack_Handle hnd)
- static bool PMBus_Stack_doesCommandMatchTransaction (const uint16_t command, const PMBus_Transaction transaction)
- bool PMBus_Stack_initModule (PMBus_Stack_Handle hnd, const uint32_t moduleBase, uint16_t ∗buffer)
- static uint16_t PMBus_Stack_Obj_getAddress (PMBus_Stack_Handle hnd)
- static uint16_t PMBus_Stack_Obj_getAddressMask (PMBus_Stack_Handle hnd)
- static PMBus_Stack_State PMBus_Stack_Obj_getCurrentState (PMBus_Stack_Handle hnd)
- static uint16_t ∗ PMBus_Stack_Obj_getCurrPtr (PMBus_Stack_Handle hnd)
- static PMBus_Stack_mode PMBus_Stack_Obj_getMode (PMBus_Stack_Handle hnd)
- static uint32_t PMBus_Stack_Obj_getModuleBase (PMBus_Stack_Handle hnd)
- static uint32_t PMBus_Stack_Obj_getModuleStatus (PMBus_Stack_Handle hnd)
- static uint16_t PMBus_Stack_Obj_getNBytes (PMBus_Stack_Handle hnd)
- static PMBus_Stack_State PMBus_Stack_Obj_getNextState (PMBus_Stack_Handle hnd)
- static bool PMBus_Stack_Obj_getPECValid (PMBus_Stack_Handle hnd)
- static uint16_t ∗ PMBus_Stack_Obj_getPtrBuffer (PMBus_Stack_Handle hnd)
- static PMBus_Transaction PMBus_Stack_Obj_getTransaction (PMBus_Stack_Handle hnd)
- static transactionHandler PMBus_Stack_Obj_getTransactionHandler (PMBus_Stack_Handle hnd, const PMBus_Transaction transaction)
- static void PMBus_Stack_Obj_setAddress (PMBus_Stack_Handle hnd, const uint16_t address)
- static void PMBus_Stack_Obj_setAddressMask (PMBus_Stack_Handle hnd, const uint16_t addressMask)
- static void PMBus_Stack_Obj_setCurrentState (PMBus_Stack_Handle hnd, const PMBus_Stack_State state)
- static void PMBus_Stack_Obj_setCurrPtr (PMBus_Stack_Handle hnd, uint16_t ∗currPtr)
- static void PMBus_Stack_Obj_setMode (PMBus_Stack_Handle hnd, const PMBus_Stack_mode mode)
- static void PMBus_Stack_Obj_setModuleBase (PMBus_Stack_Handle hnd, const uint32_t address)

- static void PMBus_Stack_Obj_setModuleStatus (PMBus_Stack_Handle hnd, const uint32_t status)
- static void PMBus_Stack_Obj_setNBytes (PMBus_Stack_Handle hnd, const uint16_t nBytes)
- static void PMBus_Stack_Obj_setNextState (PMBus_Stack_Handle hnd, const PMBus_Stack_State state)
- static void PMBus_Stack_Obj_setPECValid (PMBus_Stack_Handle hnd, const bool PECValid)
- static void PMBus_Stack_Obj_setPtrBuffer (PMBus_Stack_Handle hnd, uint16_t *buffer)
- static void PMBus_Stack_Obj_setTransaction (PMBus_Stack_Handle hnd, const PMBus_Transaction transaction)
- static void PMBus_Stack_Obj_setTransactionHandler (PMBus_Stack_Handle hnd, const PMBus_Transaction transaction, transactionHandler tH)

Variables

- PMBus_Stack_Handle hndPmbusStackSlave
- static const PMBus_Transaction_Obj_u PMBus_Stack_commandTransactionMap[64]
- PMBus_Stack_Obj pmbusStackSlave

6.2.1 Data Structure Documentation

6.2.1.1 _PMBus_Stack_Obj_

Definition:

```
typedef struct
{
    uint32_t moduleBase;
    uint32_t moduleStatus;
    PMBus_Stack_mode mode;
    uint16_t slaveAddress;
    uint16_t slaveAddressMask;
    PMBus_Stack_State currentState;
    PMBus_Stack_State nextState;
    uint16_t *ptrBuffer;
    uint16_t *currPtr;
    uint16_t nBytes;
    bool PECValid;
    PMBus_Transaction transaction;
    transactionHandler trnHnd[NTRANSACTIONS];
}
_PMBus_Stack_Obj_
```

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
ptrBuffer pointer to a buffer of length >= 4
currPtr Current position in the buffer.
nBytes Number of bytes sent/received.
PECValid Valid PEC received or sent.
transaction Current Transaction type
trnHnd Handler for each transaction.

Description:
PMBUS Slave Mode Object.

6.2.1.2 _PMBus_Transaction_Obj_

Definition:
```
typedef struct
{
    uint16_t transaction0;
    uint16_t transaction1;
    uint16_t transaction2;
    uint16_t transaction3;
}
_PMBus_Transaction_Obj_
```

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_Transaction_Obj_u

Definition:
```
typedef struct
{
    PMBus_Transaction_Obj obj;
    uint16_t ui16;
}
PMBus_Transaction_Obj_u
```

Members:
obj
ui16

Description:
a union of the packed transactions struct and an unsigned word

6.2.2 Enumeration Documentation

6.2.2.1 PMBus_Stack_mode

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_Stack_State

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 PMBus_Stack_defaultTransactionHandler

Default Transaction Handler.

Prototype:
```
int32_t
PMBus_Stack_defaultTransactionHandler(PMBus_Stack_Handle hnd)
```

Parameters:
← ***hnd*** Handle to the PMBus_Stack_Obj object

6.2.3.2 static bool PMBus_Stack_doesCommandMatchTransaction (const uint16_t *command*, const PMBus_Transaction *transaction*) `[static]`

Checks if the command matches a given transaction type.

Parameters:
← ***command*** command to be checked against a transaction

← ***transaction*** one of the PMBus_Transaction enumerations

Description:
This function will query PMBus_Stack_commandTransactionMap for the given command to see if it can find a match for the given transaction.

Returns:
true if the command does match the given transaction type, **false** otherwise

6.2.3.3 PMBus_Stack_initModule

Initialize PMBus module.

Prototype:
```
bool
PMBus_Stack_initModule(PMBus_Stack_Handle hnd,
                       const uint32_t moduleBase,
                       uint16_t *buffer)
```

Parameters:
← ***hnd*** Handle to the PMBus_Stack_Obj object
← ***address*** Address of the PMBus module in slave mode
← ***buffer*** Address of a buffer for the PMBus_Stack_Obj object

Returns:
true if PMBus initialization was successful, else false

Note:
while this function does enable PMBus interrupts, the user must register the interrupt service routine to handle all interrupts, and call the appropriate handler within that ISR

buffer must point to an array of at least 4 words

6.2.3.4 static uint16_t PMBus_Stack_Obj_getAddress (PMBus_Stack_Handle *hnd*) `[inline, static]`

Get Address member of the PMBus_Stack_Obj object.

Parameters:
← ***hnd*** Handle to the PMBus_Stack_Obj object

Returns:
address Address of the PMBus module in slave mode

6.2.3.5 static uint16_t PMBus_Stack_Obj_getAddressMask (PMBus_Stack_Handle *hnd*) `[inline, static]`

Get Address member of the PMBus_Stack_Obj object.

Parameters:
← ***hnd*** Handle to the PMBus_Stack_Obj object

Returns:
addressMask Address Mask of the PMBus module in slave mode

6.2.3.6 static PMBus_Stack_State PMBus_Stack_Obj_getCurrentState (PMBus_Stack_Handle *hnd*) `[inline, static]`

Get Current State of the PMBus_Stack_Obj object.

Parameters:
← ***hnd*** Handle to the PMBus_Stack_Obj object

Returns:
current state of the PMBus state machine

6.2.3.7 static uint16_t∗ PMBus_Stack_Obj_getCurrPtr (PMBus_Stack_Handle *hnd*) `[inline, static]`

Get Current pointer member of the PMBus_Stack_Obj object.

Parameters:
← ***hnd*** Handle to the PMBus_Stack_Obj object

Returns:
currPtr Current position in the buffer

6.2.3.8 static PMBus_Stack_mode PMBus_Stack_Obj_getMode (PMBus_Stack_Handle *hnd*) `[inline, static]`

Get Mode member of the PMBus_Stack_Obj object.

Parameters:
← ***hnd*** Handle to the PMBus_Stack_Obj object

Returns:
mode mode of the PMBus module

6.2.3.9 static uint32_t PMBus_Stack_Obj_getModuleBase (PMBus_Stack_Handle *hnd*) `[inline, static]`

Get Module Base Address member of the PMBus_Stack_Obj object.

Parameters:
← ***hnd*** Handle to the PMBus_Stack_Obj object

Returns:
Module Base Address of the PMBus module

6.2.3.10 static uint32_t PMBus_Stack_Obj_getModuleStatus (PMBus_Stack_Handle *hnd*) `[inline, static]`

Get Module Status member of the PMBus_Stack_Obj object.

Parameters:
← ***hnd*** Handle to the PMBus_Stack_Obj object

Returns:
Status of the PMBus module

6.2.3.11 static uint16_t PMBus_Stack_Obj_getNBytes (PMBus_Stack_Handle *hnd*) `[inline, static]`

Get nBytes member of the PMBus_Stack_Obj object.

Parameters:
← ***hnd*** Handle to the PMBus_Stack_Obj object

Returns:
Number of bytes sent/received

6.2.3.12 static PMBus_Stack_State PMBus_Stack_Obj_getNextState (PMBus_Stack_Handle *hnd*) `[inline, static]`

Get Next State of the PMBus_Stack_Obj object.

Parameters:
← ***hnd*** Handle to the PMBus_Stack_Obj object

Returns:
next state of the PMBus state machine

6.2.3.13 static bool PMBus_Stack_Obj_getPECValid (PMBus_Stack_Handle *hnd*) `[inline, static]`

Get PECValid member of the PMBus_Stack_Obj object.

Parameters:
← ***hnd*** Handle to the PMBus_Stack_Obj object

Returns:
boolean, 1 - PEC is valid, 0 - PEC is invalid

6.2.3.14 static uint16_t∗ PMBus_Stack_Obj_getPtrBuffer (PMBus_Stack_Handle *hnd*) `[inline, static]`

Get buffer pointer member of the PMBus_Stack_Obj object.

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object

Returns:

Buffer Pointer to a buffer of size >= 4

6.2.3.15 static PMBus_Transaction PMBus_Stack_Obj_getTransaction (PMBus_Stack_Handle *hnd*) `[inline, static]`

Get current transaction member of the PMBus_Stack_Obj object.

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object

Returns:

transaction one of the PMBus_Transaction enumerations

6.2.3.16 static transactionHandler PMBus_Stack_Obj_getTransactionHandler (PMBus_Stack_Handle *hnd*, const PMBus_Transaction *transaction*) `[inline, static]`

Get transaction handler member of the PMBus_Stack_Obj object.

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object

← ***transaction*** one of the PMBus_Transaction enumerations

Returns:

trnHandler pointer to the function to handle the transaction

6.2.3.17 static void PMBus_Stack_Obj_setAddress (PMBus_Stack_Handle *hnd*, const uint16_t *address*) `[inline, static]`

Set Address member of the PMBus_Stack_Obj object.

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object

← ***address*** Address of the PMBus module in slave mode

6.2.3.18 static void PMBus_Stack_Obj_setAddressMask (PMBus_Stack_Handle *hnd*, const uint16_t *addressMask*) `[inline, static]`

Set Address member of the PMBus_Stack_Obj object.

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object

← ***addressMask*** Address Mask of the PMBus module in slave mode

6.2.3.19 static void PMBus_Stack_Obj_setCurrentState (PMBus_Stack_Handle *hnd*, const PMBus_Stack_State *state*) `[inline, static]`

Set Current State of the PMBus_Stack_Obj object.

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object

← ***state*** current state of the PMBus state machine

6.2.3.20 static void PMBus_Stack_Obj_setCurrPtr (PMBus_Stack_Handle *hnd*, uint16_t ∗ *currPtr*) `[inline, static]`

Set Current pointer member of the PMBus_Stack_Obj object.

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object

← ***currPtr*** Current position in the buffer

6.2.3.21 static void PMBus_Stack_Obj_setMode (PMBus_Stack_Handle *hnd*, const PMBus_Stack_mode *mode*) `[inline, static]`

Set Mode member of the PMBus_Stack_Obj object.

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object

← ***mode*** mode of the PMBus module

6.2.3.22 static void PMBus_Stack_Obj_setModuleBase (PMBus_Stack_Handle *hnd*, const uint32_t *address*) `[inline, static]`

Set Module Base Address member of the PMBus_Stack_Obj object.

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object

← ***address*** Address of the PMBus module in slave mode

6.2.3.23 static void PMBus_Stack_Obj_setModuleStatus (PMBus_Stack_Handle *hnd*, const uint32_t *status*) `[inline, static]`

Set Module Status member of the PMBus_Stack_Obj object.

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object
← ***status*** Status of the PMBus module

6.2.3.24 static void PMBus_Stack_Obj_setNBytes (PMBus_Stack_Handle *hnd*, const uint16_t *nBytes*) `[inline, static]`

Set nBytes member of the PMBus_Stack_Obj object.

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object
← ***nBytes*** Number of bytes sent/received

6.2.3.25 static void PMBus_Stack_Obj_setNextState (PMBus_Stack_Handle *hnd*, const PMBus_Stack_State *state*) `[inline, static]`

Set Next State of the PMBus_Stack_Obj object.

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object
← ***state*** next state of the PMBus state machine

6.2.3.26 static void PMBus_Stack_Obj_setPECValid (PMBus_Stack_Handle *hnd*, const bool *PECValid*) `[inline, static]`

Set PECValid member of the PMBus_Stack_Obj object.

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object
← ***PECValid*** boolean, 1 - PEC is valid, 0 - PEC is invalid

6.2.3.27 static void PMBus_Stack_Obj_setPtrBuffer (PMBus_Stack_Handle *hnd*, uint16_t ∗ *buffer*) `[inline, static]`

Set buffer pointer member of the PMBus_Stack_Obj object.

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object
← ***Buffer*** Pointer to a buffer of size >= 4

6.2.3.28 static void PMBus_Stack_Obj_setTransaction (PMBus_Stack_Handle *hnd*, const PMBus_Transaction *transaction*) `[inline, static]`

Set current transaction member of the PMBus_Stack_Obj object.

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object

← ***transaction*** one of the PMBus_Transaction enumerations

6.2.3.29 static void PMBus_Stack_Obj_setTransactionHandler (PMBus_Stack_Handle *hnd*, const PMBus_Transaction *transaction*, transactionHandler *tH*) `[inline, static]`

Set transaction handler member of the PMBus_Stack_Obj object.

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object

← ***transaction*** one of the PMBus_Transaction enumerations

← ***trnHandler*** pointer to the function to handle the transaction

6.2.4 Variable Documentation

6.2.4.1 PMBus_Stack_Handle hndPmbusStackSlave

Handle to the slave object.

6.2.4.2 const PMBus_Transaction_Obj_u PMBus_Stack_commandTransactionMap[64] `[static]`

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

6.2.4.3 PMBus_Stack_Obj pmbusStackSlave

PMBus Slave Object.

6.3 PMBus State Machine Handler

Functions

Description:

- void PMBUS_STACK_extendedCommandHandler (PMBus_Stack_Handle hnd)
- void PMBus_Stack_slaveBlockWriteOrProcessCallHandler (PMBus_Stack_Handle hnd)
- void PMBus_Stack_slaveHandler (PMBus_Stack_Handle hnd)
- void PMBus_Stack_slaveIdleHandler (PMBus_Stack_Handle hnd)
- void PMBus_Stack_slaveReadBlockHandler (PMBus_Stack_Handle hnd)
- void PMBus_Stack_slaveReadWaitForEOMHandler (PMBus_Stack_Handle hnd)
- void PMBus_Stack_slaveReceiveByteWaitForEomHandler (PMBus_Stack_Handle hnd)

6.3.1 Function Documentation

6.3.1.1 PMBUS_STACK_extendedCommandHandler

Extended Read/Write Byte/Word Handler (Slave Mode).

Prototype:

```
void
PMBUS_STACK_extendedCommandHandler(PMBus_Stack_Handle hnd)
```

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object

6.3.1.2 void PMBus_Stack_slaveBlockWriteOrProcessCallHandler (PMBus_Stack_Handle *hnd*)

Block Write or Process Call State Handler (Slave Mode).

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object

6.3.1.3 void PMBus_Stack_slaveHandler (PMBus_Stack_Handle *hnd*)

The PMBus State Machine handler (Slave Mode).

Parameters:

← ***hnd*** Handle to the PMBus_Stack_Obj object This function implements the state machine of the PMBus in slave mode, it is designed to operate in the interrupt service routine (ISR) triggered by the following interrupts

- DATA_READY (Read buffer is full)
- DATA_REQEUST (Master has requested data)
- EOM (Master signals an end of a block message)

Note:

The handler must be called in the PMBus ISR only

6.3.1.4 void PMBus_Stack_slaveIdleHandler (PMBus_Stack_Handle *hnd*)

Idle State Handler (Slave Mode).

Parameters:
← ***hnd*** Handle to the PMBus_Stack_Obj object

6.3.1.5 void PMBus_Stack_slaveReadBlockHandler (PMBus_Stack_Handle *hnd*)

Read Block State Handler (Slave Mode).

Parameters:
← ***hnd*** Handle to the PMBus_Stack_Obj object

6.3.1.6 void PMBus_Stack_slaveReadWaitForEOMHandler (PMBus_Stack_Handle *hnd*)

Read/Wait-for-EOM State Handler (Slave Mode).

Parameters:
← ***hnd*** Handle to the PMBus_Stack_Obj object

6.3.1.7 void PMBus_Stack_slaveReceiveByteWaitForEomHandler (PMBus_Stack_Handle *hnd*)

Receive Byte Wait-for-EOM State Handler (Slave Mode).

Parameters:
← ***hnd*** Handle to the PMBus_Stack_Obj object

7 Revision History

v1.00.00.00: First Release

- Slave only mode of operation
- Master mode initialization is supported

IMPORTANT NOTICE

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