Software Implementation of PMBus over I2C for TMS2803x

V1.2
January 2011

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

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

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of 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. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible or liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that products or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products.
www.ti.com/sc/docs/stdterms.htm

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

Software Implementation of PMBus over I2C for TMS2803x

Katie Enderle *C2000 Applications*

ABSTRACT

PMBus (Power Management Bus) is a free and open standard communications protocol for power management devices. This document provides a software implementation of the PMBus protocol over the I2C hardware on TMS320F28035 Piccolo™ MCUs. The software implementation provides functions to control the underlying I2C as well as to handle PMBus transactions as either a master or slave, allowing these layers to be abstracted so users can focus on developing the application layer of their PMBus application. The application report contains example code that demonstrates the use of the F28035 as a PMBus master device and as a PMBus slave device.

Contents

Figures

Tables

1 General Overview - PMBus

1.1 PMBus Origin

The PMBus (Power Management Bus) specification was created as a means to standardize digital bus communication with power converters. The protocol was developed by the Power Management Bus Implementers Forum (PMBus-IF), a subgroup of the System Management Interface Forum (SM-IF) – membership in these forums is free and open to all. Unlike some communications protocols which simply concern data transfer, the PMBus specification extends to a command layer by specifying a Command Language – a standard list of 256 commonly used power management commands. Devices need only support those commands required for their application.

1.2 PMBus Features

The underlying hardware protocol for PMBus is I2C, a widespread 2-wire protocol. In addition, PMBus specifies 2 more optional lines, control and alert. The alert line is used by the slave to notify the master of a fault, and the control line is used by the master as a chip select line to turn the slave on or off. These lines are optional – PMBus can operate as a 2-wire protocol if required with just clock and data lines.

A couple of features provide added robustness to the PMBus specification, something important for critical systems. PMBus, like SMBus, implements timeout functionality – if the clock is held low for longer than the timeout interval, the devices must reset communication within a specified period of time. An optional feature that is highly recommended to increase robustness is Packet Error Checking (PEC). PEC checks the validity of a received packet via a Cyclic Redundancy Check-8 (CRC-8) algorithm.

1.3 PMBus Formats

PMBus transactions follow one of six formats: Send Byte, Read Byte, Write Byte, Read/Write Byte, Read Word, and Read/Write Word. For all transactions, the MSB (most significant bit) of each byte is sent first.

1.3.1 Send Byte

Send Byte commands simply command the slave to perform an action. The master simply transmits one byte over I2C.

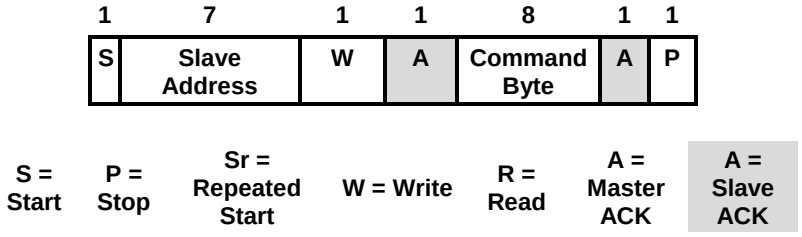

Figure 1. Send Byte Format

1.3.2 Read Byte

Read Byte commands are used to read one byte of information from the slave. The master uses I2C to transmit one byte for the command, and then receive one byte from the slave. In general, Read Byte commands are used to access read-only registers or parameters in the slave device.

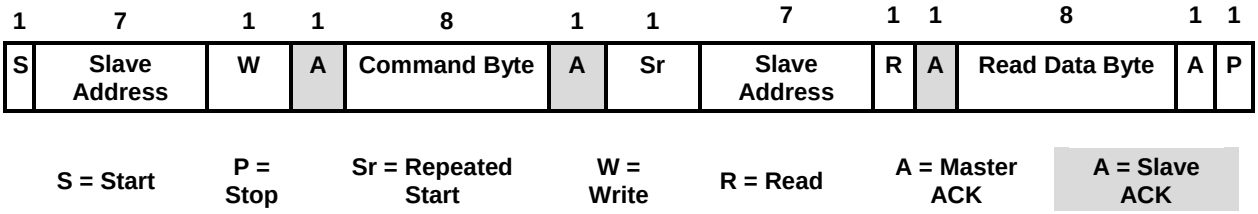

Figure 2. Read Byte Format

1.3.3 Write Byte

Write Byte commands are used to write one byte of information to a register in the slave device. The master uses I2C to transmit two data bytes – one for the command byte, which tells the slave what to do with the second byte, which contains the actual write data. In general, Write Byte commands are used to access write-only registers or parameters in the slave device. (It should be noted that there are no 'Write Word only' commands.)

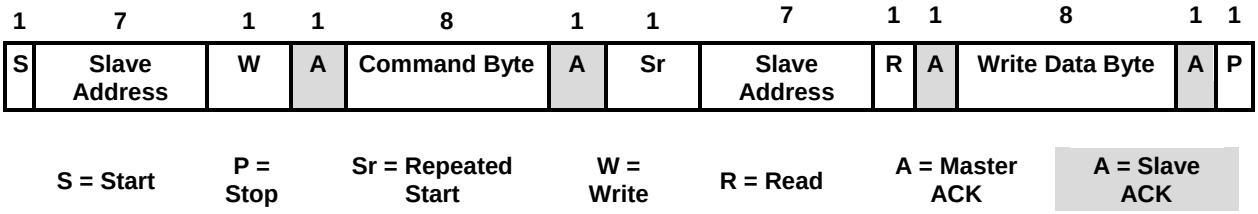

Figure 3. Write Byte Format

1.3.4 Read/Write Byte

Read/Write Byte commands are used to read or write one byte of information in the slave device. These commands follow either the Read Byte or Write Byte format, with simply the read/write bit determining the transaction direction. In general, Read/Write Byte commands are used to access registers or parameters that can be either written to or read from.

1.3.5 Read Word

Read Word commands are used to read one word (two bytes) of information from the slave device. The master uses I2C to transmit one data byte (the command byte) and then read two data bytes. The lowest-order data byte is sent first.

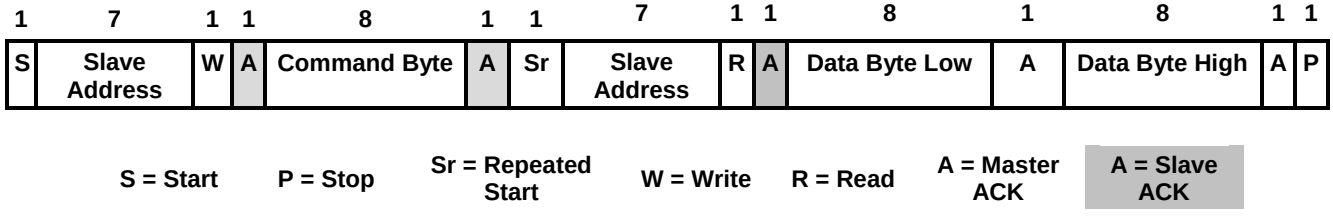

Figure 4. Read Word Format

1.3.6 Read/Write Word

Read/Write Word commands are used to read or write one word (two bytes) of information to or from the slave device. The master uses I2C to transmit the command byte, and then either reads or writes two data bytes to/from the slave device, depending on the direction of the transaction. The format is the same as Read Word for reads, and for writes it is the same except the direction of the last two bytes is from the master (transmitting) to the slave (receiving). The lowest-order data byte is sent first.

1.3.7 Packet Error Checking Protocol

When using packet error checking, an additional byte is added before the stop byte in each transaction. For reads, the PEC byte is read from the slave and the master compares it to its own PEC byte calculation. For writes, the PEC byte is sent to the slave from the master, and the slave compares it to its own PEC byte calculation.

After the comparison, if the PEC bytes differ, the slave detects a PEC error. The preferred response in the PMBus protocol is to send a NACK, which is usually done for a hardware PMBus and PEC implementation. However, the NACK is not required by the PMBus protocol. Because this implementation is done through software over I2C hardware rather than dedicated PMBus hardware, it cannot calculated and check the PEC byte in time for the NACK. Instead it takes the following actions, per the PMBus Specification:

- Does not respond to or act upon the command
- Flushes the command code and any received data
- Sets the CML bit in the register for STATUS_BYTE (variable Status_Byte in this implementation)
- Sets the PEC failed bit in the register for STATUS_CML (variable Status_Cml in this implementation) and
- Notifies the master of the presence of a fault condition by pulling the Alert line low.

2 Scope and User Implementation

2.1 Scope

This code implements the physical and transport layer for the PMBus protocol. The application layer is left up to the user. Some requirements of PMBus that are not covered by this app note and should be handled by the user include the following:

- PMBus devices must start up in a controlled manner without interaction from the serial bus – this is left up to the user.
- The code does not include an implementation of the extended command protocol or definitions of extended commands.

- Some PMBus slave devices support the Host Notify Protocol, in which slaves can temporarily act as masters to communicate with the host – this is an optional feature not implemented in this app note.

2.2 Implementation Checklist

When writing the user application, some modifications should be made to the code.

- Based on the Packet Error Checking capabilities of other devices communicating on the PMBus, the user must change the value of PEC defined in PMBus.h. PEC = 0 will build the project without Packet Error Checking, PEC = 1 will build the project with Packet Error Checking.
- Select the correct build configuration for the device – master, or slave.
- Select an appropriate master frequency for the application, somewhere between the 10kHz and 400kHz as specified in the PMBus specification. The required prescale value should be set accordingly (see the function description of I2CMaster_Init).
- The user should change the GPIOs used for the alert and control lines to match the pins used in their application. For master devices, the alert line currently triggers xint1. See the device documentation to correctly configure the desired GPIO to trigger xint1.
- For using the C2000 device as a PMBus Slave, there are some sections in the code marked “User Code”. These sections should be modified based on the PMBus commands implemented for the specific application. See the PMBus slave function descriptions in this document and the documentation in the code comments for more details.
- While the command bytes for all PMBus commands are defined in PMBus.h, the actions performed in response to the commands (defined in Part II of the PMBus Specification) are not. This is left up to the user, based on their application and their set of PMBus commands. For slave devices, the user must alter the code to provide the correct data bytes to send to the master when requested, and to take appropriate actions in response to received commands. For master devices, the user must add code to store any data received from the slave and take appropriate actions in response.

2.3 Implementation Guidelines

The following guidelines should assist in integrating the example code with the user application.

- The file PMBus.h contains symbolic definitions for the indexes for all 256 PMBus commands, matching the command names in the PMBus Specification Part II – these indexes should be used when calling the PMBusMaster_Transmit() function.
  - Example: PMBusMaster(STATUS_TEMPERATURE, …, …, …)
- The file PMBus.h also contains structures for the PMBus Status Registers with bit definitions that align with the PMBus Specification Part II. These registers are implemented in the same style as the C280x C/C++ Header files for easy, symbolic access of each bit within the register. Definition in this way also allows for CCS auto-complete functionality, making for easier code

development. This same structure can be copied to implement any other PMBus registers that are needed for the application.

- The master side of the example code is geared toward communication with one slave device, though communication with multiple slaves is possible – by calling the function PMBusMaster_Init() with the address of the next slave the master can switch which slave it is commanding.
- Selection of the appropriate pull-up resistors on the communication lines, as well as other hardware considerations, is left up to the user. It should be noted that the internal pull-up resistors on the C2000 GPIOs do not fulfill this requirement, and external pull-up resistors should be added.
- The GPIOs are configured for a Control line, but according to the PMBus specification the real implementation of the Control line functionality is its effects on certain PMBus commands. As this is part of the application layer it is left up to the user to implement this functionality as part of their command implementations. See the PMBus Specification Part II for more information about how the Control line operates with different commands.

3 Function Descriptions – PMBus

The files PMBusMaster.c and PMBusSlave.c contain code for implementing PMBus as a master or slave. By setting PEC in PMBus.h to 0 or 1, the implementation can be built with or without Packet Error Checking (PEC) functionality.

3.1 Master Functions

The following descriptions apply to functions in the file PMBusMaster.c.

3.1.1 PMBusMaster_Init(PMBusMaster_SlaveAddress, PMBusMaster_Prescale)

This function should be called once before PMBus operation begins. The device configures the underlying I2C with the I2CMaster_Init() function and configures an additional two GPIO pins for Alert and Control line functionality. The alert line pin is also configured to trigger the XINT1 interrupt – the user can modify the interrupt xint1_isr for application-specific alert line response. PMBusMaster_Prescale configures the master to communicate at the desired frequency by passing this value when initializing the underlying I2C. See I2CMaster_Init() for advice on choosing a prescale value. Before exiting the function, the master device waits for the slave to be ready by calling the I2CMaster_SlavePresent() function.

Table 1. PMBusMaster_Init() Parameter Descriptions

Name	Description
PMBusMaster_SlaveAddress	The slave device address.
PMBusMaster_Prescale	The prescale value to achieve desired master frequency.

3.1.2 PMBusMaster(PMBusMaster_CommandByte, PMBusMaster_RWFlag, PMBusMaster_Message, *PMBusMaster_ReceivedValue)

This function performs a PMBus transaction as a master over the hardware I2C. PMBusMaster_CommandByte contains the index for a PMBus command (STATUS_WORD, for example). This index is used to determine the PMBus command byte to send as well as to determine what type of command is required (read byte, write byte, etc.) and therefore the number of bytes to send and receive over I2C. For read/write commands, the parameter PMBusMaster_RWFlag determines if the master is reading from or writing to the slave. For write commands, PMBusMaster_Message contains the message to be sent – it is of type int so it can contain a one or two byte message, depending on the command. For read commands, *PMBusMaster_ReceivedValue is the pointer where the read data will be stored and passed back to the application level.

If PEC is defined as 1, Packet Error Checking Functionality is implemented in PMBusMaster(). Notice inside the main switch statement, there are sections beginning with #if !PEC and #else – these sections implement the function without and with PEC, respectively. If PEC is implemented, this function checks the PEC from the slave against its own calculation on read commands, and sends a PEC byte to be checked by the slave on write commands. This function passes the command byte, slave address, read/write flag, and any bytes transmitted or received to PMBusMaster_CRC8MakeBitwise() for PEC byte calculation. The function returns the result of the PEC comparison (1 = success, 0 = fail).

Table 2. PMBusMaster() Parameter Descriptions

Name	Description
PMBusMaster_CommandByte	The PMBus command passed by the user, from the defined list of commands in PMBus.h (for example, STATUS_WORD). This is an index to a table containing the codes of the PMBus commands, as defined in the PMBus spec.
PMBusMaster_RWFlag	The PMBus read/write flag. It is only necessary for read/write byte and read/write word commands to indicate if the master is reading from the slave or writing to the slave. Read: RW = 1, Write: RW = 0;
PMBusMaster_Message	The value to be written if this is a write command – it can contain either a byte or a word, depending on if this is a Write Byte command or a Write Word command.
*PMBusMaster_ReceivedValue	For Read functions, this is a pointer to an array to contain the byte(s) received from the slave.

3.1.3 xint1_isr

External interrupt 1 is configured to be triggered when the Alert line drops to a low voltage state. The user can alter this interrupt service routine to service the alert line according to the desired functionality of the application.

3.2 Slave Functions

The following descriptions apply to functions in the file PMBusSlave.c.

3.2.1 PMBusSlave_Init(PMBusSlave_DeviceAddress)

This function should be called once before PMBus operation. It configures the underlying I2C as a slave with the device address specified by the parameter PMBusSlave_DeviceAddress. It also sets up the GPIOs for I2C operation and for Alert and Control line functionality.

Table 3. PMBusSlave_Init() Parameter Descriptions

Name	Description
PMBusSlave_DeviceAddress	The slave device's own address.

3.2.2 PMBusSlave_DecodeCommand(PMBusSlave_RxCommand)

This function is called from the PMBusSlave() function once the command byte has been received from the master. The function looks up the command byte in a table and determines what type of command it is (read byte, write byte, etc.) It also determines if the command is one supported by the slave device and prepares information for transmission if necessary.

Table 4. PMBusSlave_DecodeCommand() Parameter Descriptions

Name	Description
PMBusSlave_RxCommand	The command byte received by the slave.

The user should alter the portion of code marked "User Code" to implement their application-specific commands.

```
////////////USER CODE////////////
#warn  "User should change code to implement their application's supported PMBus
commands."
switch (PMBusSlave_Index)         //should include all user supported commands
{
    case STATUS_TEMPERATURE:
        PMBusSlave_TransmitBuffer[0] = Temperature;
        break;

    ...

    default:
        PMBusSlave_DummyCommand = 1;           //command not supported by this slave
        break;
}
////////////END USER CODE///////
```

3.2.3 PMBusSlave()

This function waits for a PMBus command byte from the master, then performs PMBus reads and writes accordingly. The user should alter the code section labeled "User Code" to implement the application-specific slave actions in response to the master's commands, such as calling other functions, setting flags, or storing data.

```
////////////USER CODE////////////
//contains what actions to take after getting information from the master,
//usually where to store received data and what functions to call in response
#warn  "User should modify code to implement their application's supported PMBus
commands."
switch (PMBusSlave_Index)         //should include all user supported commands
{
    case STORE_DEFAULT_CODE:
        Default_Code = PMBusSlave_ReceiveBuffer[0];
        break;

    ...

    default:         //command not supported by this slave
        break;
}
////////////END USER CODE///////
```

3.3 PMBus with PEC

The files PMBusMaster.c and PMBusSlave.c contain code for implementing PMBus with Packet Error Checking (PEC) functionality, but this code is only built if PEC is defined as 1 in PMBus.h.

*3.3.1 PMBusMaster_Crc8MakeBitwise(PMBusMaster_CRC, PMBusMaster_Poly, *PMBusMaster_Pmsg, PMBusMaster_MsgSize)*

This function generates a PEC byte per the PMBus Specification based on the bytes at *PMBusMaster_Pmsg. It returns the PEC byte.

Table 5. PMBusMaster_Crc8MakeBitwise() Parameter Descriptions

Name	Description
PMBusMaster_CRC	The initial value for the CRC calculation.
PMBusMaster_Poly	The polynomial.
*PMBusMaster_Pmsg	Pointer to the data to use in the CRC calculation. This should include the slave address, read/write bit, command byte, and all data bytes that were just sent or received by the master.
PMBusMaster_MsgSize	The number of bytes of data to use in the CRC calculation.

*3.3.2 PMBusSlave_Crc8MakeBitwise(PMBusSlave_CRC, PMBusSlave_Poly, *PMBusSlave_Pmsg, PMBusSlave_MsgSize)*

This function generates a PEC byte per the PMBus Specification based on the bytes at *PMBusSlave_Pmsg. It returns the PEC byte.

Table 6. PMBusMaster_Crc8MakeBitwise() Parameter Descriptions

Name	Description
PMBusSlave_CRC	The initial value for the CRC calculation.
PMBusSlave_Poly	The polynomial.
*PMBusSlave_Pmsg	Pointer to the data to use in the CRC calculation. This should include the slave address, read/write bit, command byte, and all data bytes that were just sent or

Name	Description
	received by the slave.
PMBusSlave_MsgSize	The number of bytes of data to use in the CRC calculation.

3.3.3 Efficient PEC Calculation in F2806x Devices Using VCU

F2806x devices feature a Viterbi, Complex Math, and CRC Unit (VCU). The VCU is capable of performing a CRC-8 calculation and accumulate in one cycle. Since the PEC byte is calculated using a CRC-8 algorithm, use of the VCU can reduce the PEC calculation down to just one cycle per byte in the PMBus message, a vast speed improvement over the nested loops used in the software implementation.

To use the VCU in this project, replace all function calls to the PMBusMaster_Crc8MakeBitwise() or PMBusSlave_Crc8MakeBitwise() with get_CRC8(). This function is implemented with the following piece of assembly code:

```
;=====================================================================
; long get_CRC8(long *addr,int size)
;=====================================================================
; Input parameters:
;   *+XAR4   addr    : pointer to the block whose CRC needs to be
;                                                          calculated
;      AL                          size     : size of the block in words
;
; Output parameters:
;   ACC                                        : CRC of the block
;=====================================================================
        .sect ".text"
        .def _get_CRC8

_get_CRC8
        VCRCCLR
        ADDB        SP, #4              ; allocate 4 words for local
        MOV                         *-SP[2],ACC               ; save off AL
        ZAPA
        ; clear the ACC,P and OVERflow
        MOV                         AL,*-SP[2]                              ; restore AL
_crc_loop
        VCRC8L_1   *XAR4++
        SUBB         ACC,#1
        CMPB         AL,#0
        SBF          _crc_done,EQ
        SB           _crc_loop,UNC

_crc_done
        VMOV32       *-SP[4], VCRC        ; Store CRC
        MOV          AL, *-SP[4]          ; return AL
        VMOV32       VCRC, *-SP[2]        ; Restore VCRC
        SUBB         SP, #4               ; restore stack pointer
        LRETR
```

4 Lower-level Function Descriptions – I2C

This software implementation of PMBus relies on an underlying I2C layer that controls the hardware. The file I2CMaster.c contains functions to control the I2C layer, and PMBusSlave.c contains a function to initialize the slave side I2C layer.

4.1 I2C Master

The following descriptions apply to functions in I2CMaster.c. They control the I2C in a master device.

4.1.1 I2CMaster_Init(I2CMaster_SlaveAddress, I2CMaster_Prescale)

This function configures the I2C module as the I2C master, configures two GPIOs to be the I2C clock and data lines, and sets up the appropriate I2C interrupts. It sets up the module clock for the desired communications frequency using I2CMaster_Prescale.

The relationship between the value passed in I2CMaster_Prescale and the master communications frequency can be expressed by the following equation:

$$f_{master} = \frac{60000}{(prescale + 1) * 25} kHz$$

Table 7. I2CMaster_Init() Parameter Descriptions

Name	Description
I2CMaster_SlaveAddress	The slave device address.
I2CMaster_Prescale	The prescale value to obtain the desired communications frequency.

*4.1.2 I2CMaster_Transmit(I2CMaster_ByteCountTx, *I2CMaster_TxArray, I2CMaster_ByteCountRx, *I2CMaster_RxArray)*

This function should be called whenever the master makes an I2C transaction. The function sends the specified number of bytes from the data in the structure passed by *I2CMaster_TxArray, then receives the specified number of bytes from the slave into the structure passed by *I2CMaster_RxArray.

Table 8. I2CMaster_Transmit() Parameter Descriptions

Name	Description
I2CMaster_ByteCountTx	The number of bytes to transmit.
*I2CMaster_TxArray	Pointer to the transmit buffer.
I2CMaster_ByteCountRx	The number of bytes to receive.
*I2CMaster_RxArray	Pointer to the receive buffer.

4.1.3 I2CMaster_SlavePresent(I2CMaster_SlaveAddress)

This function checks if there is an I2C slave connected to the clock and data lines. It does this by sending a dummy byte to the slave and getting it's ACK status. If an ACK is received, the function returns 1, and if a NACK is received, the function returns 0.

Table 9. I2CMaster_SlavePresent() Parameter Descriptions

Name	Description
I2CMaster_SlaveAddress	The slave device address.

4.1.4 I2CMaster_NotReady()

This function returns the value of the busy bit (I2caCtrlRegs.I2CSTR.bit.BB).

4.1.5 I2CMaster_Wait()

This function is called from the I2CMaster_Transmit() function. It waits until the master is done with a transaction by polling the stop and busy bits (I2caRegs.I2CSTR.bit.STP and I2caRegs.I2CSTR.bit.BB).

4.1.6 i2c_master_int1a_isr

This is the I2C master interrupt service routine. It is triggered by a Read-Ready (RRDY) interrupt when the master receives data from the slave. It stores the data in a receive buffer located at *I2CMaster_ReceiveField and increments the pointer.

4.2 I2C Slave

The following descriptions apply to functions in PMBusSlave.c to initialize the I2C layer for the slave device.

4.2.1 I2CSlave_Init(I2CSlave_OwnAddress)

This function configures the I2C module as the I2C master, configures two GPIOs to be the I2C clock and data lines, and sets up the appropriate I2C interrupts. It sets up the module clock for the desired communications frequency using I2CMaster_Prescale.

Table 10. I2CSlave_Init(I2CSlave_OwnAddress) Parameter Descriptions

Name	Description
I2CSlave_OwnAddress	The slave device's own address.

5 Example Project

The example CCSv4 project demonstrates how to use the PMBus functions. It was designed to be run on two Piccolo F2803x controlCard Experimenter's Kits, one acting as the PMBus master, and one acting as the PMBus slave. The example code in master.c and slave.c demonstrate the use of the PMBus functions. One command of each type is sent from the master to the slave, and data passes back and forth accordingly. This example code, along with the code in PMBusMaster.c and PMBusSlave.c can be used as a framework for developing a custom PMBus application.

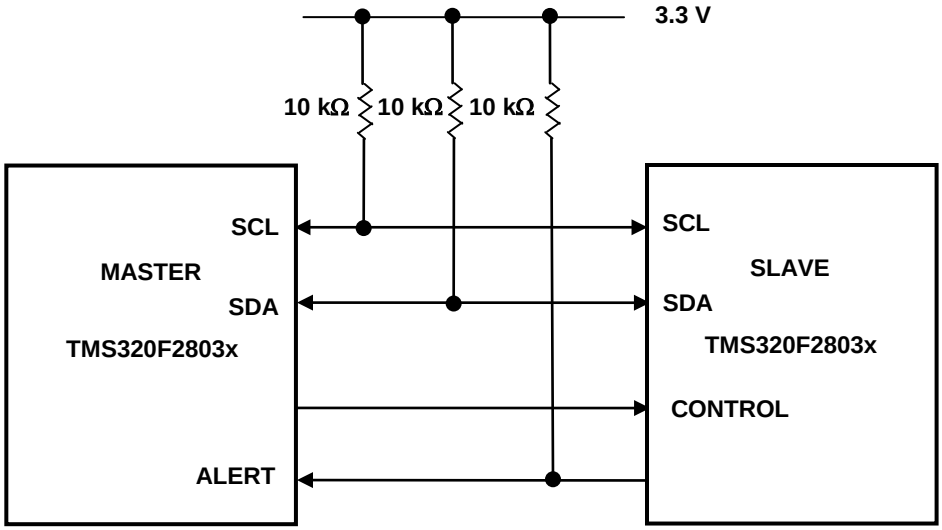

Figure 5. Example Project Setup: F2803x Master and Slave

5.1 Hardware Setup

The underlying I2C hardware requires pull-up resistors on the I2C clock and data lines. On the controlCard dock, connect resistors from the GPIO pins being used for I2C (GPIO 28 & 29 or GPIO 32 & 33) to one of the 3.3V pins available on the dock. (If the Alert line is being used, do the same to this line as well). Then use wires to connect the I2C clock and data pins on each dock. (See Figure 5).

5.2 Software Setup

After importing the project into CCSv4, set the active build configuration to “Slave,” connect the slave Piccolo device, and build and load the project. Disconnect the slave and connect the master. Set the active build configuration to “Master” and click Debug to build and load the project. Before running the code on the master, make sure the slave is connected via the clock and data lines, and is running. Now run the program – setting breakpoints and other debugging operations on the master device will work as normal, but only on the master device. If you want to debug the slave, load the code on the master device first, and then run the debugger on the slave side. In this case, start the slave running before turning on the master to ensure that the master will not think the slave has timed out while waiting for the program to begin running.

6 Naming Conventions

Many of the functions and variables in this app note duplicate for the master and slave side of communication. However, these functions and variables differ in the master and slave programs, and are not interchangeable. In order to minimize confusion between master and slave versions of the same functions and variables, a naming convention has been adopted using the name of the file preceding the variable or function name.

- Functions: FileName_FunctionName(FileName_Parameter1, FileName_Parameter2)
  - Example: PMBusMaster_Init(PMBusMaster_SlaveAddress, PMBusMaster_Prescale);
- Variables: FileName_VariableName
  - Example: PMBusSlave_CommandGroup

7 File Descriptions

Table 11. File Descriptions

File Name	Description
PMBus.h	Header file containing symbolic definitions of the indexes for all 256 PMBus commands, as well as header file style register definitions of the PMBus status registers. This model should be followed to implement the other PMBus registers needed for the user's application. PEC, defined at the top of this file, determines if Packet Error Checking is implemented or not.
PMBusMaster.h	Header file containing the function declarations for the PMBus master functions.
PMBusSlave.h	Header file containing the function declarations for PMBus slave functions.
I2CMaster.h	Header file containing the function declarations for I2C master functions.
PMBusMaster.c	File containing PMBus master function implementations.
PMBusSlave.c	File containing PMBus slave function implementations. The sections marked "User Code" should be modified to match the PMBus commands implemented in the user's application.
Master.c	An example program that shows how to use functions in PMBusMaster.c
Slave.c	An example program that shows how to use functions in PMBusSlave.c.

8 References

1. *TMS320F28030/28031/28032/28033/28034/28035 Piccolo Microcontrollers Data Manual (Rev. D)* (SPRS584)
2. *TMS320x280x, 2803x Piccolo Inter-Integrated Circuit (I2C) Module Reference Guide* (SPRUFZ9C)
3. *TMS320F2803x Piccolo System Control and Interrupts Reference Guide* (SPRUGL8B)
4. *2803x C/C++ Header Files and Peripheral Examples* (SPRC892)

For more information about PMBus and to download the latest PMBus Specification documents:

5. *PMBus-IF (Power Management Bus Implementer's Forum)* http://pmbus.org
6. *SM-IF (System Management Interface Forum)* http://smiforum.org