

Software Implementation of PMBus Over I2C for TMS320F2803x

Katie Enderle

ABSTRACT

Power Management Bus (PMBus) is a free and open standard communications protocol for power management devices. This application report provides a software implementation of the PMBus protocol over the inter-integrated circuit (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 you can focus on developing the application layer of your PMBus application.

Project collateral and source code discussed in this application report can be downloaded from <http://www.ti.com/lit/zip/SPRABJ6>. The example code demonstrates the use of the F28035 as a PMBus master and slave device.

Contents

1	General Overview - PMBus	2
2	Scope and User Implementation	4
3	Function Descriptions – PMBus	5
4	Lower-Level Function Descriptions – I2C	8
5	Example Project	10
6	Naming Conventions	11
7	File Descriptions	11
8	References	12

List of Figures

1	Send Byte Format	2
2	Read Byte Format	2
3	Write Byte Format	3
4	Read Word Format	3
5	Example Project Setup: F2803x Master and Slave	10

List of Tables

1	PMBusMaster_Init() Parameter Descriptions	5
2	PMBusMaster() Parameter Descriptions	6
3	PMBusSlave_Init() Parameter Descriptions	6
4	PMBusSlave_DecodeCommand() Parameter Descriptions	6
5	PMBusMaster_Crc8MakeBitwise() Parameter Descriptions	7
6	PMBusMaster_Crc8MakeBitwise() Parameter Descriptions	8
7	I2CMaster_Init() Parameter Descriptions	9
8	I2CMaster_Transmit() Parameter Descriptions	9
9	I2CMaster_SlavePresent() Parameter Descriptions	9
10	I2CSlave_Init(I2CSlave_OwnAddress) Parameter Descriptions	10
11	I2CSlave_Init(I2CSlave_OwnAddress) Parameter Descriptions	11

1 General Overview - PMBus

1.1 PMBus Origin

The PMBus 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 two 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.

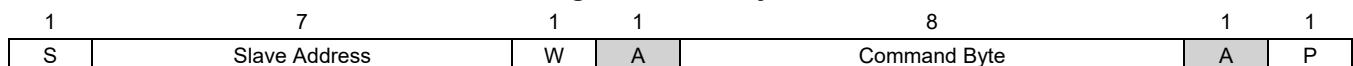
1.3 PMBus Formats

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. The shaded area in [Figure 1](#) is defined in the legend in bold.

Figure 1. Send Byte Format

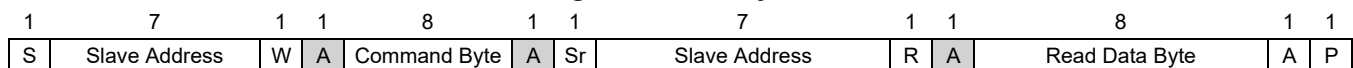


LEGEND: S = Start; P = Stop; Sr = Repeated Start; W = Write; R = Read; A = Master ACK; **A = Slave ACK**

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. The shaded area in [Figure 2](#) is defined in the legend in bold.

Figure 2. Read Byte Format



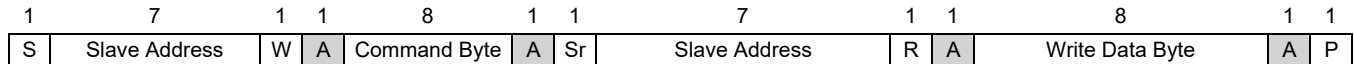
LEGEND: S = Start; P = Stop; Sr = Repeated Start; W = Write; R = Read; A = Master ACK; **A = Slave ACK**

Piccolo, C2000, Code Composer Studio, controlCard are trademarks of Texas Instruments. All other trademarks are the property of their respective owners.

2.1.1 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. Note that there are no *Write Word only* commands. The shaded area in [Figure 3](#) is defined in the legend in bold.

Figure 3. Write Byte Format



LEGEND: S = Start; P = Stop; Sr = Repeated Start; W = Write; R = Read; A = Master ACK; **A = Slave ACK**

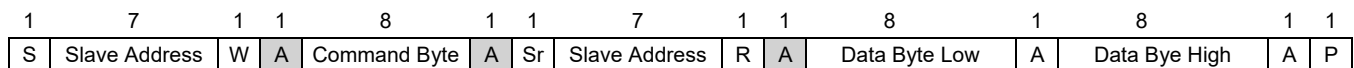
2.1.2 Read/Write Byte

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 the read/write bit determining the transaction direction. In general, the read/write byte commands are used to access registers or parameters that can be either written to or read from.

2.1.3 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. The shaded area in [Figure 4](#) is defined in the legend in bold.

Figure 4. Read Word Format



LEGEND: S = Start; P = Stop; Sr = Repeated Start; W = Write; R = Read; A = Master ACK; **A = Slave ACK**

2.1.4 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.

2.1.5 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 calculate 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 you. Some requirements of PMBus that are not covered by this application report and should be handled by you include the following:

- PMBus devices must start up in a controlled manner without interaction from the serial bus – this is left up to you.
- 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 application report.

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, you 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 10 kHz and 400 kHz as specified in the PMBus specification. The required prescale value should be set accordingly (see the function description of `I2CMaster_Init()`).
- You should change the GPIOs used for the alert and control lines to match the pins used in your application. For master devices, the alert line currently triggers `xint1`. See the device-specific documentation to correctly configure the desired general-purpose input/output (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. For more details, see the PMBus slave function descriptions in [Section 3.2](#) of this document and the documentation in the code comments.
- While the command bytes for all PMBus commands are defined in the *PMBus.h* file, the actions performed in response to the commands (defined in Part II of the *Power Management Bus Specification* <http://pmbus.org/specs.html>) are not. This is left up to you, based on your application and your set of PMBus commands. For slave devices, you 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, you 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 *PMBus.h* file 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 *PMBus.h* file 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 manner also allows Code Composer Studio™ 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 you. Note 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; however, 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 you to implement this functionality as part of your command implementations. For more information about how the control line operates with different commands, see the *Power Management Bus Implementer's Forum* (PMBus-IF) <http://pmbus.org>.

3 Function Descriptions – PMBus

The *PMBusMaster.c* and *PMBusSlave.c* files contain code for implementing PMBus as a master or slave. By defining PEC in *PMBus.h* as 0 or 1, the implementation can be built with or without PEC functionality.

3.1 Master Functions

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

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; you 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. For advice on choosing a prescale value, see `I2CMaster_Init()`. 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
<code>PMBusMaster_SlaveAddress</code>	The slave device address
<code>PMBusMaster_Prescale</code>	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, the packet error checking functionality is implemented in `PMBusMaster()`. Note that 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	Value	Description
PMBusMaster_CommandByte		The PMBus command passed by you, from the defined list of commands in <i>PMBus.h</i> (for example, STATUS_WORD). This is an index to a table containing the codes of the PMBus commands, as defined in the PMBus specification.
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.
	0	Write
	1	Read
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. You 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 *PMBusSlave.c* file.

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

You should alter the portion of code marked *User Code* to implement your 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;
}

```

3.2.3 PMBusSlave()

This function waits for a PMBus command byte from the master, then performs PMBus reads and writes accordingly. You 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 *PMBusMaster.c* and *PMBusSlave.c* files contain code for implementing PMBus with 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 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 function `PMBusMaster_Crc8MakeBitwise()` or `PMBusSlave_Crc8MakeBitwise()` with `get_CRC8()`. This function can be 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
        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 *I2CMaster.c* file contains functions to control the I2C layer and the *PMBusSlave.c* file 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:

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

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 its ACK status. If an ACK is received, the function returns '1', 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 the *PMBusSlave.c* file 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 slave, configures two GPIOs to be the I2C clock and data lines, and sets up the appropriate I2C interrupts. It sets the slave's address to be I2CSlave_OwnAddress.

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 as shown in Figure 5. 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.

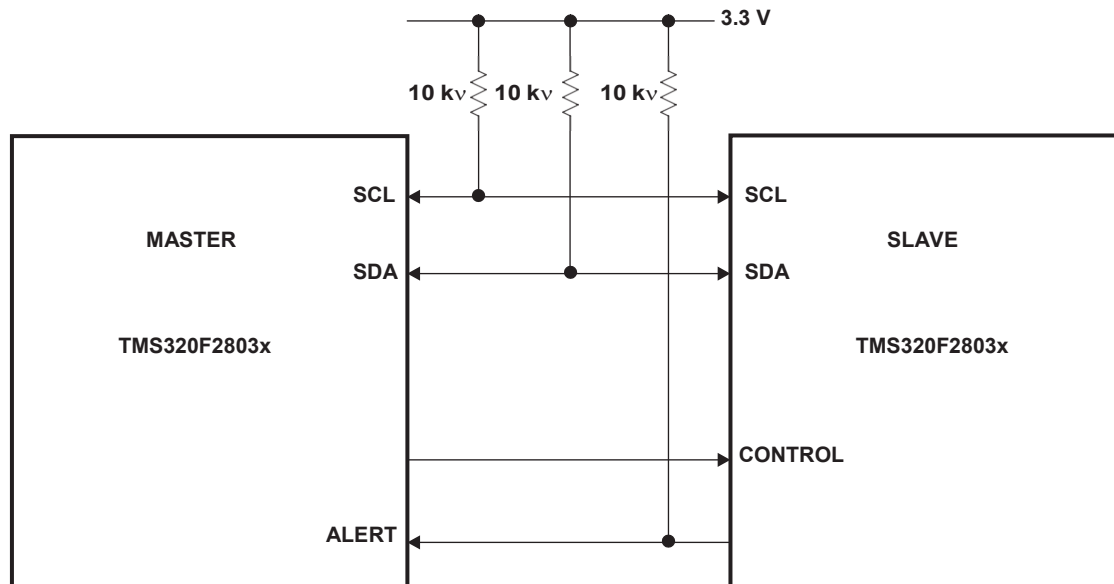


Figure 5. Example Project Setup: F2803x Master and Slave

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 3.3 V 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:

1. Set the active build configuration to *Slave*.
2. Connect the slave Piccolo device, and build and load the project.
3. Disconnect the slave and connect the master.
4. Set the active build configuration to *Master* and click Debug to build and load the project.
5. Make sure the slave is connected via the clock and data lines and is running before running the code on the master.
6. 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:

1. Load the code on the master device first.
2. Run the debugger on the slave side.
3. Start the slave running before turning on the master to ensure that the master does 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 application report 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. I2CSlave_Init(I2CSlave_OwnAddress) Parameter Descriptions

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 your application. PEC, defined at the top of this file, determines whether 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 <i>User Code</i> should be modified to match the PMBus commands implemented in your application.
Master.c	An example program that shows how to use functions in the <i>PMBusMaster.c</i> file
Slave.c	An example program that shows how to use functions in the <i>PMBusSlave.c</i> file

8 References

- *TMS320F28030, TMS320F28031, TMS320F28032, TMS320F28033, TMS320F28034, TMS320F28035 Piccolo Microcontrollers Data Manual* ([SPRS584](#))
- *TMS320x2802x, 2803x Piccolo Inter-Integrated Circuit (I2C) Module Reference Guide* ([SPRUFZ9](#))
- *TMS320F2803x Piccolo System Control and Interrupts Reference Guide* ([SPRUGL8](#))
- *2803x C/C++ Header Files and Peripheral Examples* ([SPRC892](#))

For more information regarding PMBus and access to current PMBus Specification documentation, see:

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

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to TI's Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2019, Texas Instruments Incorporated