Power Modules

APPLICATION NOTE 304

Microcontroller programming for digital products

Abstract

Our digital products can be configured, controlled and monitored through a digital serial interface using the PMBus™ power management protocol. This application note provides an introduction to the serial bus interface of the our digital products and accessing the products using a general purpose microcontroller. This application note is not a detailed tutorial on the PMBus power management protocol. It is recommended that you study the SMBus and PMBus specification documents before attempting to access our digital product via the serial bus interface. While it is possible to write firmware for the microcontroller that creates a serial bus by "bit banging" general purpose I/O pins, this is not recommended. It is highly recommended that the user choose a microcontroller that includes a hardware implementation of an I²C or SMBus interface. For such an interface, the programmer is generally only concerned with transferring data to and from transmit and receive buffers and monitoring flags for the status of the data transmission. The details of working with a microcontroller's I²C or SMBus interface are dependent on the microcontroller being used and are not discussed in this application note. This application note will assume that a microcontroller with a hardware I²C or SMBus port is being used.

Contents

Introduction

Flex Power Modules digital products are designed with state-of-the-art digital controllers. This provides the user with superior electrical performance and a broad capability to configure, control and monitor the products in the engineering lab, in the factory, and in the field.
This capability is provided by the use of the open-standard PMBus™ digital power management protocol. The PMBus protocol was created by the System Management Interface Forum (SMIF) and Power Management Bus (PMBus) Implementers Forum to standardize communication with a wide range of power conversion devices. The resulting PMBus standard is written in two parts.

Figure 1a: digital PoL - BMR473

The first, "Specification Part I – General Requirements Transport and Electrical Interface", specifies the transport including the physical layer, addressing, and packet structure.

The second, "Specification Part II – Command Language", specifies the command language to be used when communicating with PMBus compliant devices.

Figure 1b: digital IBC - BMR351

The PMBus specifications are freely available at the PMBus website

In addition to the capabilities provided by the PMBus, some of Flex Power Modules' digital PoL regulators feature the Group Command Bus (GCB). The GCB is an inter-device communication bus that provides additional capabilities like digital current sharing and fault propagation management.
This document, to be used in conjunction with the PMBus specifications and the Datasheets for each product, describes how to interface a microcontroller based system controller to the Flex Power Modules digital bus converters and regulators. The focus of this document is more on the issues involved in programming the microcontroller than in the details of the physical layer and bit-by-bit operation of the bus.

Figure 1c: digital PoL - BMR469

This application note is applicable for all of our digital products.

Forum websites

The System Management Interface Forum (SMIF)

The System Management Interface Forum (SMIF) supports the rapid advancement of an efficient and compatible technology base that promotes power management and systems technology implementations. The SMIF provides a membership path for any company or individual to be active participants in any or all of the various working groups established by the implementer forums.

Power Management Bus Implementers Forum (PMBUS-IF)

The PMBus-IF supports the advancement and early adoption of the PMBus protocol for power management. This website offers recent PMBus specification documents, PMBus articles, as well as upcoming PMBus presentations and seminars, PMBus Document Review Board (DRB) meeting notes, and other PMBus related news.

PMBus – power system management bus protocol documents

These specification documents may be obtained from the PMBus-IF website described above. These are required reading for complete understanding of the PMBus implementation. This application note will not re-address all of the details contained within the two PMBus Specification documents.

Specification Part I – General Requirements Transport And Electrical Interface

Includes the general requirements, defines the transport and electrical interface and timing requirements of hardwired signals.

Specification Part II – Command Language
Describes the operation of commands, data formats, fault management and defines the command language used with the PMBus.

SMBus – system management bus documents

System Management Bus Specification, version 3.2, Jan 2022

This specification specifies the version of the SMBus on which Revision 1.4 of the PMBus Specification is based. This specification is freely available from the System Management Interface Forum Web site.

SMBus basics

Introduction

We start with an introduction to the SMBus because the PMBus protocol is based on SMBus Version 3.2. The PMBus protocol does have some unique extensions to the SMBus interface that will be discussed later.

The SMBus is a two wire serial bus with electrical characteristics very similar to the I²C bus invented by Philips. If you are familiar with using a microcontroller to manage I²C devices then you will have little difficulty accessing our digital products via the serial bus interface.

SMBus signals

The SMBus has two required signal wires. SMBCLK is the clock signal that is generated only by the Main (Master) bus device. SMBDATA is a bi-directional signal that is used to transfer data between the Main and the Secondary (Slave) devices. For our digital products, these signals are identified as SCL and SDA, respectively.

The SMBus specification also includes an interrupt signal, SMBALERT#, that a Secondary device can use to notify the Main bus device that it has information for the Main bus unit. This is used by the digital products and is identified on the product technical specifications as SALERT. A description of the operation of the alert signal is given below.

PHY layer and electrical interface

The SMBus physical layer is very similar to, but not identical to, the physical layer of the I²C bus. For most purposes, SMBus and I²C devices are interoperable on the same bus. For the details of the electrical levels and signal timing, please refer to the SMBus specification, .System Management Bus Specification, Version 3.2, Jan, 2022.

The SMBus, like the I²C bus, uses open drain devices to drive the signal lines. One pull-up resistor is needed for each signal line. It is recommended that the pull-up resistor be placed at the Main bus device. The value of the pull-up resistor depends on supply voltage and the number of devices on the bus. See the SMBus specification for information on choosing the proper value of the pull-up resistor.

Make sure that the rise and fall times of the bus signals are compliant with the SMBus specification. Improper rise and fall times are one of the most common causes of unreliable bus operation.

Some microcontrollers offer the option of configuring the serial bus port for I²C or SMBus signal levels. If the microcontroller being used offers this option, configure the serial bus pins for the SMBus signal levels.

Addressing

The SMBus specification, like the I²C specification, provides for each device to have a seven bit physical address. Each device on the bus must have a unique address. It is left to the system engineer to assure that there are no address conflicts.

Our DC/DC products use resistors to program the bus address of each device. The device address is not configurable through the serial bus interface. Consult the datasheet for the digital product being used for the details on setting the device's bus address.

Transaction basics

In most microcontrollers with I²C and SMBus interfaces, the details of the transaction are invisible to the programmer. The programmer simply reads and writes data from buffers and monitors flags for data transmission status. However, we will provide here a short discussion of the how data is sent and received over the bus. This understanding will be helpful when working with more complex commands and data.

The Main bus initiates a transaction by placing a START condition on the bus. This alerts all of the Secondary devices to start listening.

After the START condition the Main transmits a seven bit address followed by an eighth bit that indicates whether the Main will be writing or reading data from the Secondary device.

Each Secondary device compares its address to the address just received. If there is a match, the device acknowledges its address with an ACK condition on the bus. When the Main device detects the ACK it proceeds with the rest of the data transaction.

During the transmission of data the receiving device must either acknowledge the byte with an ACK or tell the sender that there was a problem by not acknowledging (NACK) the byte.

The exception is the case where data is being transferred to the Main bus device. Upon receiving the last data byte, the Main bus does not acknowledge ('NACKs') the byte.

The transaction is completed when the Main bus puts a STOP condition on the bus. The bus is then considered free (not busy) after a specified minimum delay time after the STOP condition.

Bit and bytes – the data link layer

Bits and data validity

Figure 1 below shows how bits are transmitted on the SMBus.

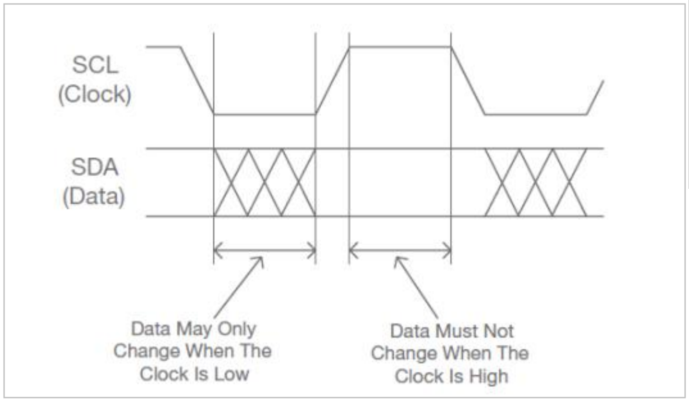
SCL (Clock)
SDA (Data)
Data May Only Change When The Clock Is Low
Data Must Not Change When The Clock Is High

Figure 2: SMBus bit transfer

The Main bus device always drives the clock (SCL) line. When the Main pulls the clock line low, it is notifying the sending device that the next bit is to be placed on the data line.
Note that the sending device may be a Secondary device transferring data to the Main bus or it may be the Main bus writing data to a Secondary device. The data line is allowed to change state only during the time that the clock is low.

The Main bus then allows the clock line to go high. This signals the receiving device that the next bit is ready to be read from the data line. During the time that the clock is high, the data line must not change state.

START, ACK, NACK and STOP conditions.

Transactions on the SMBus are initiated when the Main bus puts a START condition on the bus and are ended when the Main bus puts a STOP condition on the bus.

Figure 2 shows a SMBus start condition. Initially, the bus is idle with both the clock and data lines high for minimum specified amount of time (see the SMBus specification for the details). The Main then pulls the data line low while the clock is high. This signals all devices on the bus that a transaction is about to start.

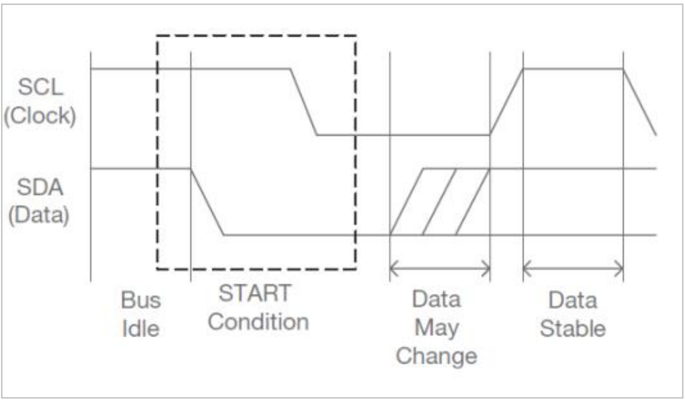

Figure 3: SMBus START condition

Once the START condition has been placed on the bus, the Main continues to toggle the clock line to control the transfer of bits. Figure 3 shows the transfer of the first data byte. The Main device keeps toggling the clock to cause the transfer of the eight bits in the byte.

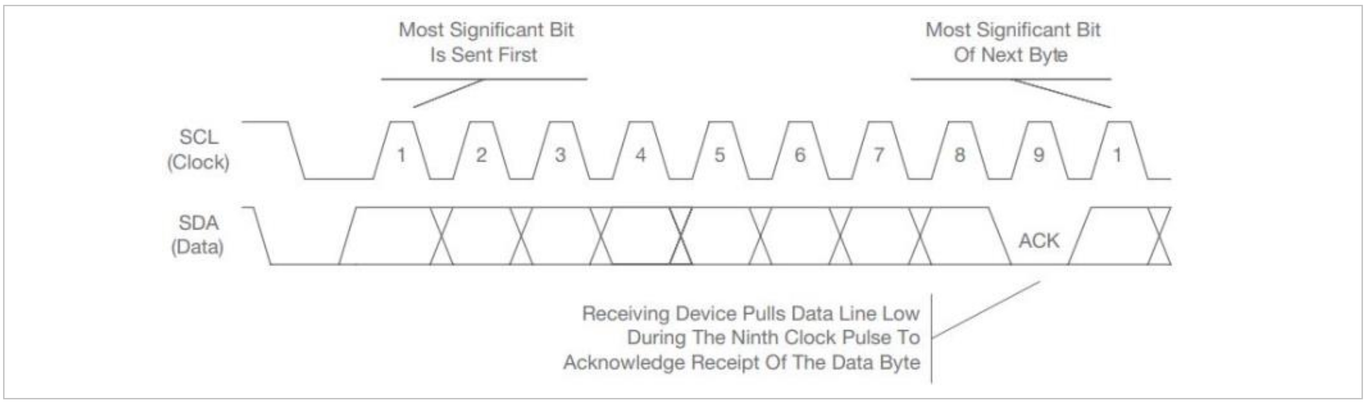
Most Significant Bit Is Sent First
Most Significant Bit Of Next Byte
SCL (Clock)
1
2
3
4
5
6
7
8
9
1
SDA (Data)
ACK
Receiving Device Pulls Data Line Low During The Ninth Clock Pulse To Acknowledge Receipt Of The Data Byte

Figure 4: SMBus byte transfer with acknowledge

Then the Main lowers the clock for a ninth bit. If the receiving device has received the byte it pulls the data line low during the ninth clock period. This acknowledges (ACKs) to the sending device that the byte was received and that the transaction can continue.

NACK (Not Acknowledge)

If the receiving device did not successful receive the byte, it does not pull the data line low during the ninth bit. This lack of acknowledgement (NACK) notifies the sending device that the byte was not successfully received. In this case, the sending device typically ends the transaction and initiates its programmed error response and recovery function. Figure 4 shows a data byte that ends with a NACK.

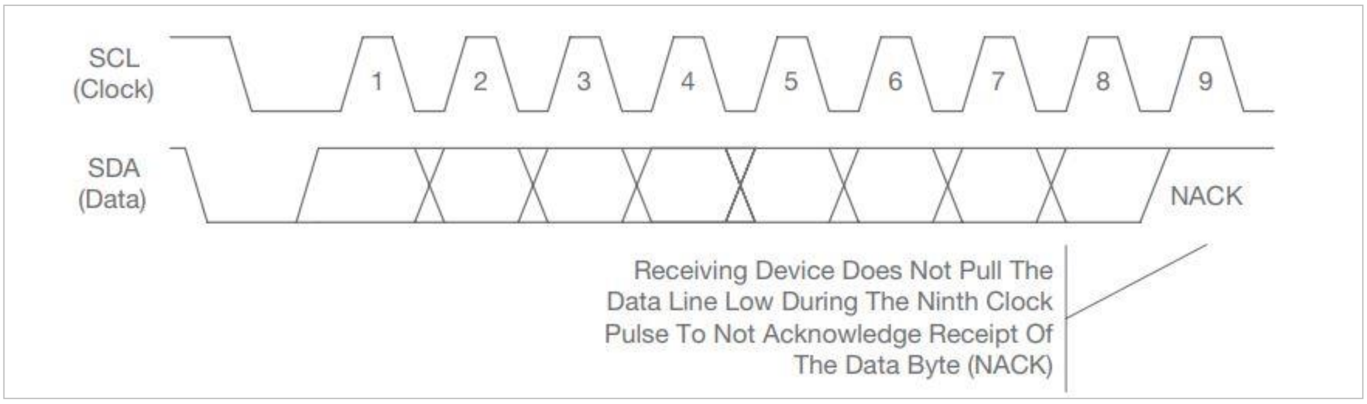
SCL (Clock)
1
2
3
4
5
6
7
8
9
SDA (Data)
NACK
Receiving Device Does Not Pull The Data Line Low During The Ninth Clock Pulse To Not Acknowledge Receipt Of The Data Byte (NACK)

Figure 5: SMBus data byte without acknowledge

Other than a Main device not acknowledging the last data byte of a read from a Secondary device as described below, each byte should be acknowledged. A device may not acknowledge a byte for one of several different reasons.

Possible reasons include the device being too busy to respond or the device considers the data to be invalid.

There is no direct way to know why a device NACK'ed a data byte. The digital regulators and bus converters, through the comprehensive status and fault reporting in the PMBus protocol specification, can provide the Main bus information that will generally let the Main know why a data byte was NACK'ed.

STOP condition

A Main device notifies all devices on the bus that the current transaction is complete by placing a STOP condition on the bus. Figure 5 shows STOP condition after an ACK of the last data byte of the transaction. After the ACK, when the data line is low, the Main continues to hold both the data and clock lines low. It then releases the clock line and allows it to go high. After a time interval specified in the SMBus specification, the Main allows the data line to go high. Again, the transition of the data line while the clock is high, which is not allowed during the normal transmission of the data bits, signals to all devices on the bus that the transaction is complete. At this point the bus is again idle with both the clock and data lines high.

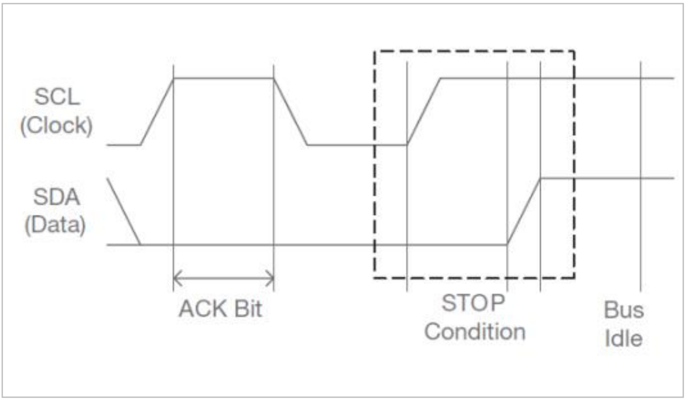

Figure 6: STOP condition after ACK

In some cases the STOP condition will be issued after the last data byte is not acknowledged (NACK'ed). In this case the data line is high during the ninth clock period. After the ninth clock period is complete, the Main pulls the clock line and the data line low. The Main then releases the clock line and allows it to go high. After the specified delay time, the Main releases the data line and allows it to go high. This STOP condition signals to all of the devices on the bus that the current transaction is complete. Figure 6 shows the clock and data signals when a STOP condition is put on the bus after a NACK.

Is should be noted that a Main may put a STOP condition on the bus at any time, not just at the end of a data byte. While this is an abnormal condition, it is permitted by the SMBus specification.

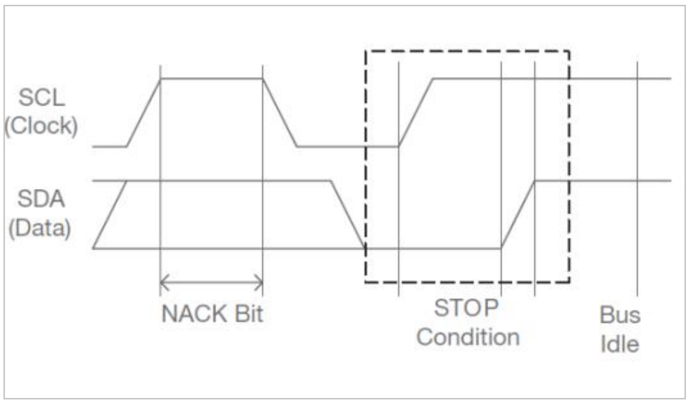

Figure 7: STOP condition after a NACK

Clock stretching

There may be occasions when a device involved in a SMBus transaction may want to pause the transaction. For example, a device that just received a data byte may want to check that received data byte is valid. A Secondary device being asked to supply data may need extra time to retrieve that data from a relatively slow EEPROM memory.

The SMBus specification allows for a device to pause a transaction by holding the clock line low. This stops the bus until the clock line is allowed to go high. This is called "clock stretching". This practice is discouraged as a matter of good system practice but may be unavoidable. Any device acting as a Main bus must be prepared to accept clock stretching by a Secondary device.

Figure 7 illustrates how clock stretching by a Secondary device works and how it can cause problem for a Main bus that is not SMBus compliant. In this case the Main bus is sending data to the Secondary device. After the eighth bit, the Secondary device needs time to validate the data in the byte and decide whether to ACK or NACK. After the Main lowers the clock to end the eighth bit, the Secondary device turns on its clock output to hold the clock line low and to pause the bus. Since the clock and data outputs on SMBus devices are open-drain, any device that turns on its clock or data output will hold that signal line low.

The Main bus, in its normal timing cycle, turns off its clock output for the ninth (acknowledge) bit. However, because the Secondary device is holding the clock line low, the clock line does not go high. For a Main device that cannot handle clock stretching by a Secondary device, this would be an error condition. That Main would typically terminate the transaction and set an error flag.

For a Main bus device that is compatible with clock stretching by a Secondary device, it will leave its clock output off and wait. One the Secondary device releases the clock line and the clock goes high, the Main bus reads the data line to determine if the Secondary device has ACK'ed or NACK'ed the data byte. In Figure 7 the Secondary device has ACK'ed the data byte and the Main bus continues with sending the next byte of data.

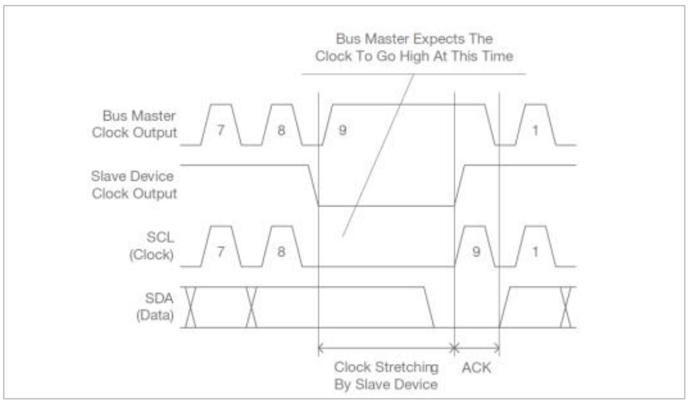

Figure 8: Clock stretching by a SMBus Secondary device

The best way to avoid problems with clock stretching by Secondary devices is to make sure that the Main bus device is fully compliant to the SMBus specification. It is up to the system engineer for the system to assure that any microcontroller used as a Main bus is SMBus compliant and will accept clock stretching by a Secondary device.

The alternative, which is not recommended, is to use bit banged general purpose I/O pins for the SMBus interface.

Bus timeout limits

While the SMBus specification allows for clock stretching, it does not allow for an unlimited pausing of the bus. The details are important so it is recommended that you carefully study the SMBus specification.

In general, during one bus transaction, a Secondary device may not extend the clock by total cumulative time of more than 25 ms (TLOW:SEXT).

Main bus devices are not permitted to extend the clock low by more than 10 ms in any one byte (TLOW:MEXT).

The SMBus specification also requires that a Main bus device that detects that the clock has been held low for more than 25 ms (TTIMEOUT,MIN) must put a STOP condition on the bus during or just after the current data byte. Any device that has detected an excessive clock low time are required to reset their communications controller and be ready to receive a new START condition within 35 ms (TTIMEOUT,MAX).

This is an important feature of the SMBus. While it cannot recover from all faults, such as a clock line with an electrical short to ground, it can help recover a bus that has become stuck due to software, logic, or even some noise errors.

This is in contrast to the I^{2}C bus which a clock line held low indefinitely is a valid condition (minimum clock speed is 0 Hz).

And again, it is best that these timing details are handled in the hardware of an I/O port that is fully compliant with the SMBus specification.

SMBus data transfer protocols

Information is transmitted in atomic transactions. SMBus transactions are completed only through one of several formats defined in the SMBus specification. This is different from the I^{2}C specification which does not address how data is to be transferred between devices on the bus. For a complete list of the permitted transactions see the SMBus Specification (Version 3.2).

Only the Main bus may initiate a transaction and all commands and data are transferred in a continuous transaction. The bus remains busy until the transaction is complete.

Bit and byte order

In a SMBus transaction, the bytes are transmitted starting with the lowest order byte and ending within the highest order byte.

For example, suppose the data to be transmitted was decimal value 31899 which can be represented by the two byte hexadecimal value 0x7C9B. When the device sending the data queues the bytes for transmission the byte 0x9B would be sent first, followed by the byte 0x7C.

Note that within the data byte, the data is written with the most significant bit first.

WRITE BYTE protocol example

To illustrate how SMBus transactions work, consider Figure 8 which illustrates the SMBus WRITE BYTE protocol. This protocol is used by the Main bus device to write a single byte of data to a Secondary device.

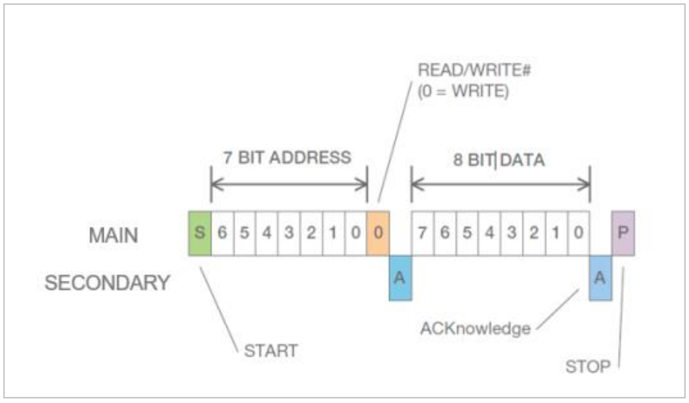

Figure 9: SMBus WRITE BYTE protocol

The transaction proceeds as follows:

- The Main device puts a START condition on the bus to notify the Secondary devices that a transaction is beginning.
- The Main sends the 7 bit address of the device to receive the data followed by the READ/WRITE# bit set to 0 (to indicate that this will be a transaction in which the Main writes data to the Secondary device).
- The receiving device acknowledges its address and that it is ready to receive data (ACK).
- The Main device sends the data byte.
- The receiving device ACKs the received data byte.
- The Main device puts a STOP condition on the bus to notify the Secondary devices that the transaction is complete.

From the programmer's point of view, all that is needed is to prepare the first byte with the address and READ/WRITE# bit and the data byte. These are passed to the hardware I²C or SMBus interface in accordance with the microcontroller's specification.

READ WORD protocol example

Figure 9 illustrates the SMBus READ WORD protocol that a Main device uses to read two bytes of data from a Secondary device.

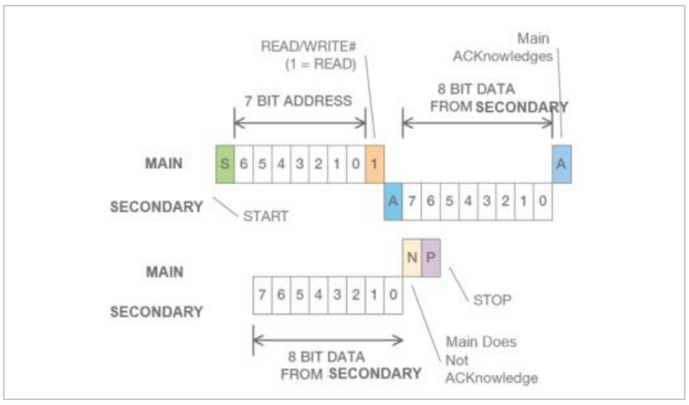

Figure 10: SMBus READ WORD protocol

The READ WORD protocol transaction with packet error checking proceeds as follows:

- The Main device puts a START condition on the bus to notify the Secondary devices that a transaction is beginning.
- The Main sends the 7 bit address of the device to receive the data followed by the READ/WRITE# bit set to 1 (to indicate that this will be a transaction in which the Main reads data from the Secondary device).
- The receiving device acknowledges its address and that it is ready to send data (ACK).
- The Secondary device sends the first data byte. Remember that if this is a 16 bit value, this is the lower order byte.
- The Main bus ACKs the received data byte.
- The Secondary device sends the second data byte. Remember that if this is a 16 bit value, this is the higher order byte.
- The Main device does not acknowledge ("NACKs") the received packet error checking data.
- The Main device puts a STOP condition on the bus to notify the Secondary devices that the transaction is complete.

PMBus data formats

Before discussing how commands are sent to a digital product over the PMBus, it is important to understand the formats of the data being written to or read from a digital product.

The data for a PMBus command used with a digital product may be one of three formats:

- 16 bit linear for output voltage related commands
- 11 bit linear for other numerical values, and
- Custom formats for commands with non-numerical data (such as commands that set the response to a given type of fault).

16 bit linear for output voltage related commands

The PMBus specification describes three different possible formats for the data used in commands that send or receive data related to setting or adjusting the output voltage.

The VOUT_MODE command specifies the format in use.

The 16 bit Linear Format is not used for commands that set fault or warning thresholds related to the output voltage. The 11 bit Linear Format (described below) is used to set the fault and warning threshold values.

The digital regulators and converters use the Linear format, which is a 16 bit fixed point integer representation. This 16 bit data provides for the very fine resolution needed to set and adjust the voltage on today's high performance logic and processors.

The commands that use this format and whether the data is unsigned or two's complement is listed in Table 1.

In the PMBus specification all output voltages are treated as positive and commands that directly set the output voltage, such as VOUT_COMMAND, are unsigned.

Commands that modify the output voltage, such as VOUT_TRIM, are signed so that the voltage can be increased or decreased.

Command name	Command description	Data format
VOUT_COMMAND	Sets the nominal output voltage	Unsigned
VOUT_TRIM	Used to trim or adjust the output voltage	Two's complement
VOUT_CAL_OFFSET	Used to calibrate the output voltage	Two's complement
VOUT_MAX	Sets the maximum nominal voltage to which the output can be programmed. This command is typically used to prevent unintentionally programming the voltage to a level that could damage the load.	Unsigned
VOUT_MARGIN_HIGH	Sets the output voltage when the regulator or converter is commanded to margin the output voltage to a greater than the nominal value.	Unsigned
VOUT_MARGIN_LOW	Sets the output voltage when the regulator or converter is commanded to margin the output voltage to a less than the nominal value.	Unsigned
READ_VOUT	Returns the actual, measured output voltage	Unsigned

Table 1: Output voltage related commands and data formats

Data format for digital PoL regulators

Data format

For the digital PoL regulators, the VOUT_MODE command is read only. This means that the fixed point format (location of the binary point) is fixed and cannot be changed for the user.
For commands that directly set the output voltage, the data is a 16 bit unsigned value in the format F3.13 (F means unsigned fixed point, there are three bits to the left of the binary point and 13 bits to the right of the binary point).
For commands that modify the output voltage, the data is a 16 bit two's complement value.

Command name	Command description	Range and resolution
VOUT_COMMAND	F3.13 (Unsigned, 3 bits to the left of the binary point, 13 bits to the right of the binary point)	Resolution (1 LSB): 122.07 µV/bit (2^{-13} V/bit) Maximum Value: 7.99987793 V (0xFFFF) (= (2^{16} – 1) ·× 1 LSB = 8 V – 1 LSB)
VOUT_TRIM	Q2.13 (Two's complement, 1 sign bit plus 2 bits to the left of the binary point, 13 bits to the right of the binary point)	Resolution (1 LSB): 122.07 µV/bit (2^{-13} V/bit) Maximum Positive Value: 3.99987793 V (0x7FFF) (= (2^{15} – 1) ·× 1 LSB = 4 V – 1 LSB) Most Negative Value: -4.0 V (0x8000) (= -2^{15} × 1 LSB)

Table 2: Output voltage related commands and range & resolution

Example 1: Setting the output voltage

Suppose the output is to be set to 3.3 V. The command to use is VOUT_COMMAND. The data that goes with that command would be determined as follows:

$$\frac{3.3\,\mathrm{V}}{2^{-13}\,\frac{\mathrm{V}}{\mathrm{bit}}} = \frac{3.3\,\mathrm{V}}{122.07\,\frac{\mu\mathrm{V}}{\mathrm{bit}}} = 27033.6 \Rightarrow 27034 = \mathrm{0x699A}$$

Remembering that in the SMBus protocols the low bytes are transmitted first, the Main would send 0x9A as the first data byte and 0x69 as the second data byte.

Example: 2: Trimming the output voltage

Suppose now that the output voltage is be reduced by 50 mV using the VOUT_TRIM command. The data bytes would be determined as follows:

$$\frac{-50\,\mathrm{mV}}{2^{-13}\,\frac{\mathrm{V}}{\mathrm{bit}}} = \frac{-50\,\mathrm{mV}}{122.07\,\frac{\mu\mathrm{V}}{\mathrm{bit}}} = -409.6 \Rightarrow\ 410 = \mathrm{0xFE66}$$

Again, as the low bytes are sent first, the Main would send 0x66 as the first data byte and 0xFE as the second data byte.

Data format for the digital Intermediate Bus Converters

Data Format

For the digital intermediate bus converters, the VOUT_MODE command is read only. This means that the fixed point format (location of the binary point) is fixed and cannot be changed for the user. For commands that directly set the output voltage, the data is a 16 bit unsigned value in the format F5.11 (F means unsigned fixed point, there are five bits to the left of the binary point and 11 bits to the right of the binary point).
For commands that modify the output voltage, the data is a 16 bit two's complement value.

Command name	Command description	Range and resolution
VOUT_COMMAND	F5.11 (Unsigned, 5 bits to the left of the binary point, 11 bits to the right of the binary point)	Resolution (1 LSB): 488 µV/bit (2^{-11} V/bit) Maximum Value: 31.99951 V (0xFFFF) (= (2^{16} – 1) ·× 1 LSB = 32 V – 1 LSB)
VOUT_TRIM	Q4.11 (Two's complement, 1 sign bit plus 4 bits to the left of the binary point, 11 bits to the right of the binary point)	Resolution (1 LSB): 488 µV/bit (2^{-11} V/bit) Maximum Positive Value: 15.99951 V (0x7FFF) (= (2^{15} – 1) ·× 1 LSB = 16 V – 1 LSB) Most Negative Value: -16.0 V (0x8000) (= -2^{15} × 1 LSB)

Table 3: Output voltage related commands and range & resolution

Example 1: Setting the output voltage
Suppose the output of a BMR4530000/002 (nominal 9 V output) is to be set to 9.6 V. The command to use is VOUT_COMMAND. The data that goes with that command would be determined as follows:

$$\frac{9.6\,\mathrm{V}}{2^{-11}\frac{\mathrm{V}}{\mathrm{bit}}} = \frac{9.6\,\mathrm{V}}{488\frac{\mu\mathrm{V}}{\mathrm{bit}}} = 19660.8.6 \Rightarrow 19661 = 0\mathrm{x}4\mathrm{CCD}$$

Remembering that in the SMBus protocols the low bytes are transmitted first, the Main would send 0xCD as the first data byte and 0x4C as the second data byte.

Example: 2: Trimming the output voltage
Suppose now that the output voltage of an intermediate bus converter is be reduced by 150 mV using the VOUT_TRIM command. The data bytes would be determined as follows:

$$\frac{-150\,\mathrm{mV}}{2^{-11}\frac{\mathrm{V}}{\mathrm{bit}}} = \frac{-150\,\mathrm{mV}}{488\frac{\mu\mathrm{V}}{\mathrm{bit}}} = -307.2 \;\Rightarrow\; 307 = 0\mathrm{xFECD}$$

Again, as the low bytes are sent first, the Main would send 0xCD as the first data byte and 0xFE as the second data byte.

11 Bit linear format

For settings and measurements that do not need the high resolution of the output voltage, such as measuring the input voltage, the PMBus specification provides for an 11 bit two's complement fixed point data format.

This data format is the same for the digital regulators and the digital intermediate bus converters.

When using this format, there are sixteen data bits. The five high order bits are a two's complement number that sets the location of the binary point. Another way to think of these five bits is that they set the scale factor.

The eleven low order bits are a two's complement that contains the basic number information. Figure 10 shows how the two data bytes are structured.

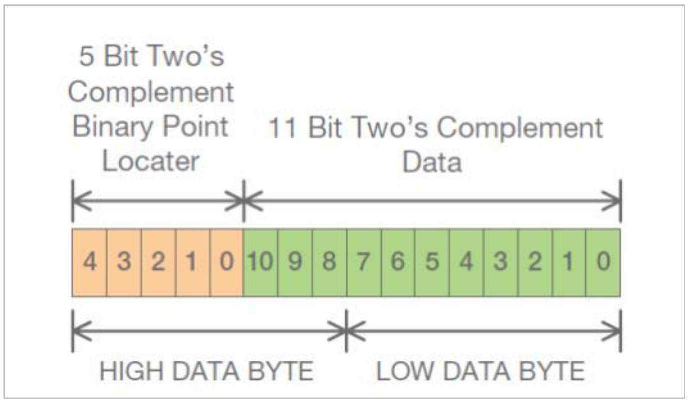
5 Bit Two's Complement Binary Point Locater
11 Bit Two's Complement Data
4 3 2 1 0 10 9 8 7 6 5 4 3 2 1 0
HIGH DATA BYTE
LOW DATA BYTE

Figure 10: 11 bit linear format data byte structure

Table 4 shows the possible ranges of values and resolutions that can be expressed with the 11 Bit Linear format.

Parameter	Value	Calculation
Maximum Positive Value	33,521,664 (0x7FFF)	2^{+15} x (210 – 1) = 2^{+15} x 1023
Minimum Positive Resolution (LSB)	15.26×10^{-6} (0x8001)	2^{-16} x 1
Minimum Negative Value Resolution	-15.26×10^{-6} (0x87FF)	2^{-16} x -1
Maximum Negative Value	-33,554,432 (0x7C00)	2^{+15} x -210 = 2^{+15} x (-1024)

Table 4: 11 Bit linear format data ranges and resolutions

Example 1: 11 bit linear data to be written to a PMBus device

Suppose the output overcurrent threshold is to be set to 10 A using the IOUT_OC_FAULT_LIMIT command. There are many possible values of the two data bytes depending on where the binary point (scaling factor) is set. To use the smallest possible resolution, start by calculating the scaling factor

$$N = integer\left(\frac{\log\left(\frac{10}{1023}\right)}{\log(2)}\right) = integer(-6.677) = -6 = 0b11010$$

Note that for any given 5 bit binary point locater value, the smallest resolution is about 0.1% (1 part in 1024).

Note that when calculating N, we need a function that takes the integer portion (truncates). If the result were rounded up, then the size of the least significant bit (LSB) would be so small that later, when the 11 bit data value is calculated, it would exceed the maximum values of +1023 or -1024. Now use the scaling factor to calculate the resolution of the overcurrent threshold:

$$\text{Overcurrent Threshold Resolution} = 1\text{A} \times 2^{-6} = 15.526 \frac{\text{mA}}{\text{bit}}$$

Calculate the number of LSBs:

$$\text{Overcurrent Threshold Data (\# Of LSBs)} = \frac{10\,\text{A}}{15.625 \frac{\text{mA}}{\text{bit}}} = 640 = 0\text{b}010\ 1000\ 0000$$

Put the scaling factor and data bits together to form the two data bytes:

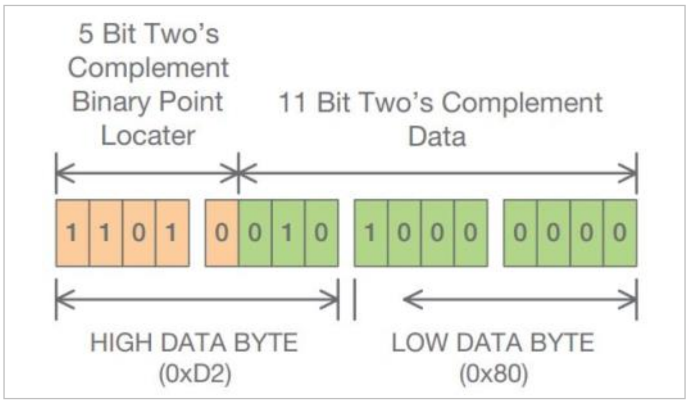

Figure 11: Data bytes for 11 bit linear format example 1

Remembering that the low order byte is transmitted first, the Main would send the value 0x80 followed by 0xD2.
Note that this value is not unique. Other scaling factors could be used, resulting in a different binary representation of the same decimal value.

Example 2: 11 decoding 11 bit linear data received from a PMBUS device
Suppose that a digital intermediate bus converter returns the data bytes 0x85 followed by 0xE0 in response to the READ_IOUT command. The question is what output current is the device reporting? Assembling the two data bytes into the correct order gives the hexadecimal value 0xE085. If we separate this into the five most significant and 11 least significant bits:

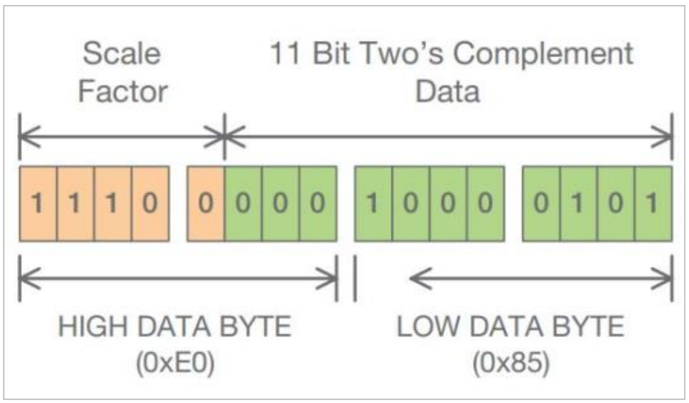

Figure 12: Data bytes for 11 bit linear format example 2

The scale factor is 0b11100 which is -4 (decimal). The data bits are 0b000 1000 0101 which is 133 (decimal). The value of the output current is calculated as:

$$\text{IOUT} = 2^{-4} \frac{\text{A}}{\text{bit}} \cdot 133 = 62.5 \frac{\text{mA}}{\text{bit}} \cdot 133 = 8.3125\,\text{A}$$

Non-numeric data

Many PMBus commands, such as those that read the status of the PMBus device or configure the fault response, have a non-numeric format. The details of these formats are given in Part II of the PMBus specification.

PMBus commands on the bus

When PMBus commands are sent over the bus, the general structure of the transaction is:

- 7 bit address followed by a zero to indicate that the next data byte is being written to the Secondary device
- A one byte command code that instructs the receiving device to take some action
- And for most PMBus commands, data is either written to the device (such as the data setting the output voltage) or read from the device (such as a reading of the current output voltage)

Example PMBus transaction 1: A command with one byte of data written to a PMBus device

Figure 13 illustrates a bus transaction for a PMBus command that writes one byte of data to the PMBus device.

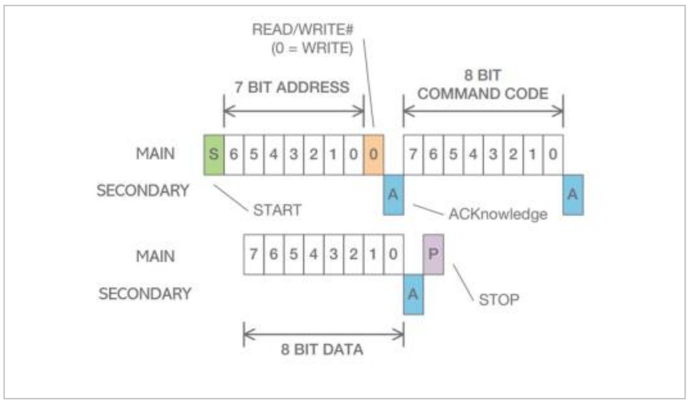

Figure 13: Example PMBus transaction protocol with one byte written to the PMBus device

This transaction proceeds as follows:

- The Main device puts a START condition on the bus to notify the Secondary devices that a transaction is beginning.
- The Main sends the 7 bit address of the device to receive the data followed by the READ/WRITE# bit set to 0 (to indicate that this will be a transaction in which the Main writes data to the PMBus device).
- The PMBus device acknowledges its address and that it is ready to receive data (ACK).
- The Main bus device sends the one byte command code.
- The PMBus device ACKs the received command code.
- The Main device puts a REPEATED START condition on the bus to notify the Secondary devices that a transaction is beginning.
- The Main sends the 7 bit address of the device to receive the data followed by the READ/WRITE# bit set to 1 (to indicate that this will be a transaction in which the Main reads data from the PMBus device).
- The PMBus device acknowledges its address and that it is ready to send data (ACK).
- The PMBus device sends the first data byte for the received command code.
  The Main device ACKs the received data byte.
- The PMBus device sends the second data byte for the received command code.
- The Main device does not acknowledge (NACKs) the received data byte.
- The Main device puts a STOP condition on the bus to notify the Secondary devices that the transaction is complete.

Please consult the microcontroller's documentation for information on how to implement the REPEATED START and read of data from a PMBus device.

Example PMBus transaction 3: A command that reads a block of data from a PMBus device

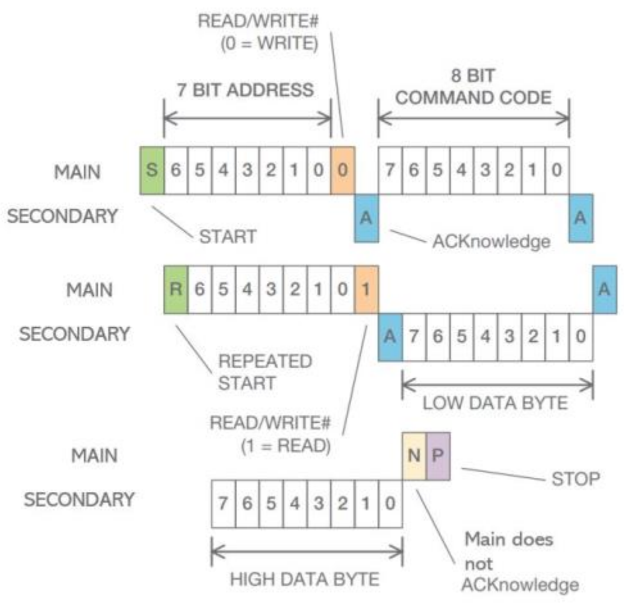

Figure 14: PMBus command that reads a block of data from a PMBus device

The transaction proceeds as follows:

- The Main device puts a START condition on the bus to notify the Secondary devices that a transaction is beginning.
- The Main sends the 7 bit address of the device to receive the data followed by the READ/WRITE# bit set to 0 (to indicate that this will be a transaction in which the Main writes data to the Secondary device).
- The PMBus device acknowledges its address and that it is ready to receive data (ACK).

The Main bus device sends the one byte command code.

The PMBus device ACKs the received command code.

- The Main device puts a REPEATED START condition on the bus to notify the Secondary devices that a transaction is beginning.
- The Main sends the 7 bit address of the device to receive the data followed by the READ/WRITE# bit set to 1 (to indicate that this will be a transaction in which the Main reads data from the PMBus device).
- The PMBus device acknowledges its address and that it is ready to send data (ACK).
- The PMBus device sends the number of bytes of data to follow (byte count). The byte count does not include the byte including the byte count information.
- The Main device ACKs the received data byte with the byte count.
- The PMBus device sends the first data byte for the received command code.
- The Main device ACKs the received data byte.
- The PMBus device sends the second data byte for the received command code.
- The Main device ACKs the received data byte.
- The sending of data bytes and acknowledgment by the Main continues until the PMBus device sends the Nth data byte.
- The Main device does not acknowledge (NACKs) the reception of the Nth data byte.
- The Main device puts a STOP condition on the bus to notify the Secondary devices that the transaction is complete.

Responding to a NACK from a PMBus device

If a PMBus device does not acknowledge a command or data byte, the current transaction should be ended by putting a STOP condition on the bus. Any further data transfer cannot be considered reliable. There is no immediate way to know why the PMBus device did not acknowledge the command or data byte. The Main bus device must interrogate the PMBus device using status commands to determine the cause of the NACK.

Packet error checking

SMBus packet error checking

The SMBus specification provides for an optional basic means to detect (but not correct) errors in the packet – packet error checking (PEC). In the SMBus packet error checking a one byte cyclic redundancy check (CRC) sum is added at the end of each transaction. Each device can compare the received checksum, calculated by the sender, with the checksum it computes from the received data. If the checksums match there is good assurance that the data was received uncorrupted. The digital regulators and digital intermediate bus converters support SMBus packet error checking.

Calculating the checksum

The checksum is calculated using all data bytes plus the byte containing the address and READ/WRITE# bit. The formula for calculating the checksum is:

$$C(x) = x^8 + x^2 + x + 1$$

There are several ways of calculating the checksum. Some use a pure arithmetic computation, which uses less memory but takes more time. Other algorithms use a lookup table which is less computation time but takes more memory.

Appendix 1 gives an example in C language of a way to calculate the PEC checksum in a direct way (no tables). There is no best algorithm for all applications and the choice of algorithm is left to the PMBus Main device firmware engineer.

Checking PEC support in a Secondary device

Before using packet error checking with a PMBus device, the Main should determine if the device supports packet error checking. This is done with the PMBus QUERY command.

If bit [7] of the data byte returned in response to a QUERY command is set (=1), the device supports packet error checking. If bit [7] is cleared (= 0) then the device does not support packet error checking.

Writing data with a PEC byte

Figure 16 illustrates a PMBus WRITE WORD transaction using packet error checking.

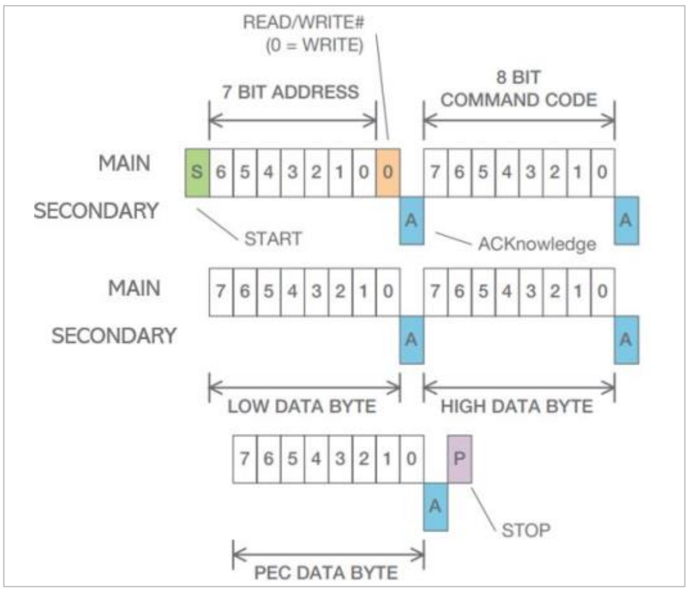

Figure 15: PMBus WRITE WORD command with packet error checking byte

The transaction proceeds as follows:

- The Main device puts a START condition on the bus to notify the Secondary devices that a transaction is beginning.
- The Main sends the 7 bit address of the device to receive the data followed by the READ/WRITE# bit set to 0 (to indicate that this will be a transaction in which the Main writes data to the PMBus device).
- The PMBus device acknowledges its address and that it is ready to receive data (ACK).
- The Main bus device sends the one byte command code.

- The PMBus device ACKs the received command code.
- The Main bus sends the low data byte.
- The PMBus device ACKs the received data byte.
- The Main bus sends the high data byte.
- The PMBus device ACKs the received data byte.
- The Main bus sends the packet error checking data byte.
- The PMBus device ACKs the received data byte.
- The Main device puts a STOP condition on the bus to notify the Secondary devices that the transaction is complete.

Reading data with a PEC byte

Figure 16 illustrates a PMBus READ WORD transaction using packet error checking.

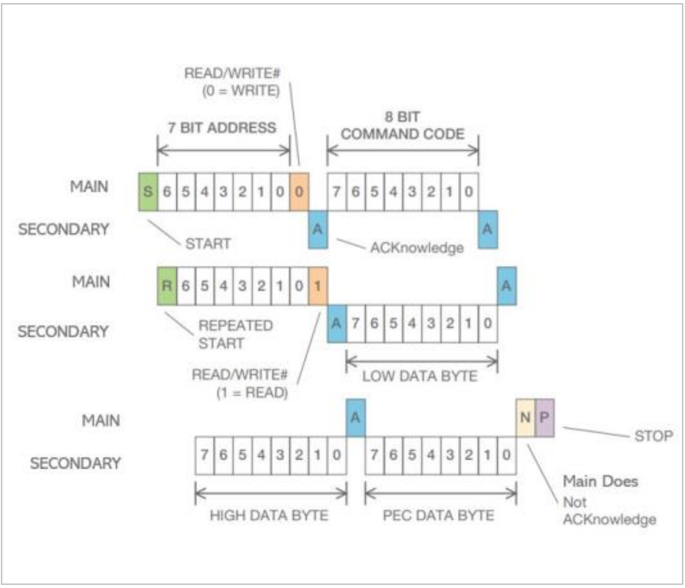

Figure 16: PMBus WRITE WORD command with packet error checking byte

The transaction proceeds as follows:

- The Main device puts a START condition on the bus to notify the Secondary devices that a transaction is beginning.
- The Main sends the 7 bit address of the device to receive the data followed by the READ/WRITE# bit set to 0 (to indicate that this will be a transaction in which the Main writes data to the Secondary device). .
- The PMBus device acknowledges its address and that it is ready to receive data (ACK).
- The Main bus device sends the one byte command code.
- The PMBus device ACKs the received command code.
- The Main device puts a REPEATED START condition on the bus to notify the Secondary devices that a transaction is beginning.
- The Main sends the 7 bit address of the device to receive the data followed by the READ/WRITE# bit set to 1 (to indicate that this will be a transaction in which the Main reads data from the PMBus device).
- The PMBus device acknowledges its address and that it is ready to send data (ACK).
- The PMBus device sends the first data byte for the received command code.
- The Main device ACKs the received data byte.
- The PMBus device sends the second data byte for the received command code.
- The Main device ACKs the received data byte. This notifies the PMBus device that the Main is expecting another data byte.
- The PMBus device sends the packet error checking data byte.
- The Main device does not acknowledge (NACKs) the packet error checking data byte.
- The Main device puts a STOP condition on the bus to notify the Secondary devices that the transaction is complete.

Please consult the microcontroller’s documentation how to implement the REPEATED STARTand read of data from a PMBus device.

No PEC support

If you are using PEC with a PMBus Device that does not support packet error checking then the Main sends a packet error checking data byte to a PMBus device that does not support packet error checking, it will treat this as a communications fault (too many data bytes for the command).

If a Main device attempts to read a packet error checking byte from a device that does not support packet error checking, the device will:

- Send a 0xFF value (by not driving the data line while the Main is requesting the data bits) and
- Declare a communications fault because the Main asked for more data bytes than are specified for the command.

Handling a failed checksum comparison

The SMBus packet error checking only provides a means to detect errors. There is no ability to correct corrupted data. If a PMBus Main device receives data for which the checksums do not match the only recourse is to read the data again.

SALERT (SMBALERT#) protocol

The SMBALERT# protocol is an important element of the SMBus and PMBus protocols. Suppose a PMBus has a change condition, an overtemperature condition for example, the Main bus device should be notified. One way to do that would be for the PMBus device to become a Main bus and send a message. However, multi-Main bus systems are not favored due to issues with congestion and conflict.

The notification method that is preferred, and implemented in our digital products, is for the PMBus device to use a separate, dedicated signal line to notify the Main bus of a change of condition. That signal in our digital products is the SALERT.

One possible implementation is for each digital product to have its SALERT signal connected to a dedicated input on the Main bus device. With this implementation the Main bus knows instantly and unambiguously which digital product needs attention.

The disadvantage to this approach is the number of signal lines on the system board and the number of I/O pins needed on the Main bus microcontroller.

Another possible implementation is to have one SALERT line that is common to all of the digital products and that terminate on one I/O pin of the microcontroller. With this approach, the Main bus must use the SMBALERT# protocol to determine which digital product or products need attention.

An important aspect of the alert protocol is that the SALERT outputs on the digital products are all open drain. It is possible that more than one digital product or other PMBus device is simultaneously asserting the SALERT signal by pulling the signal low. With that in mind, here is how the alert protocol works.

First, one or more digital products or other PMBus devices assert the SALERT signal by pulling it low.

The SALERT input to the microcontroller can either be a polled I/O pin or a pin with interrupt-on-change functionality. Once the microcontroller detects the SALERT has been asserted, it reads the SMBALERT Response Address (SRA).

The seven bit SMBALERT Response Address, 0001 100, is a special and reserved SMBus address.

When a PMBus device detects the SRA with the READ/ WRITE# bit set for read, and it is asserting the SALERT signal, it responds with its address. If more than one device responds, the open-drain wired-AND connection of the SALERT signal assures that the PMBus device with the lowest address will have a valid response. A PMBus device that attempts to send a 1 as part of its address will see that the value on the bus is a 0. That indicates to the PMBus device that it has lost the bit-wise arbitration. It stops trying to send its address and leaves the SALERT signal asserted. The bitwise arbitration is illustrated in Figure 17.

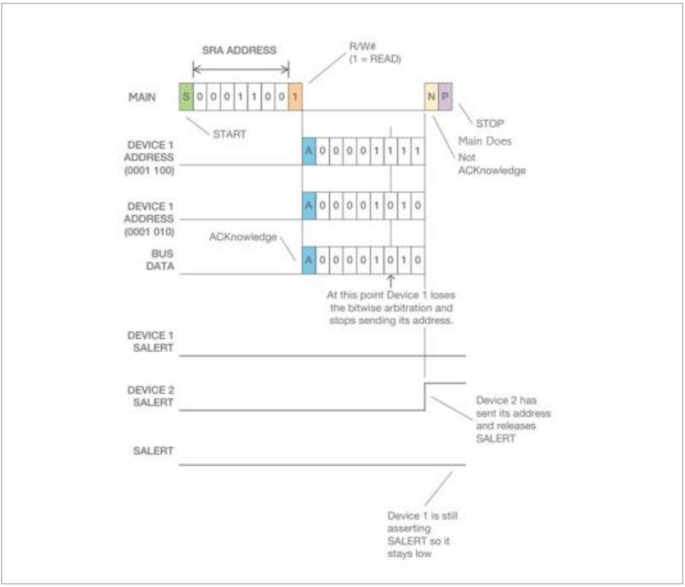

Figure 17: Alert response bitwise arbitration

<u>Note</u> that is the only time a Main bus device addresses a PMBus device with the READ/WRITE# bit set for a read.

The Main bus device now has the address of the PMBus device that was asserting the SALERT signal. The Main bus should then check the state of the SALERT signal.

If another PMBus device is asserting SALERT, the SALERT signal reMains low. The Main bus device should then send another read to the SMBALERT# Response Address. The Main bus will then get the address of another PMBus device that was asserting SALERT.

Again, the Main bus should check SALERT. If SALERT is high, then no more PMBus devices are asserting SALERT. If SALERT is still low, the Main bus device should keep reading the SRA until SALERT is no longer asserted.

At that point, the Main bus can follow its programmed response to an alert condition. For example, the next step might be to send a STATUS_BYTE or STATUS_WORD command to each device that was asserting SALERT to get the first level of diagnostic information. With this status information, the Main can decide to take action on the information it has to make inquiries to the Secondary about its status. What action to take in case of a fault or abnormal condition is the decision of the system engineer and is not part of the PMBus specification.

Other PMBus signals

The PMBus specification provides for two dedicated signals in addition to the standard SMBus signals (data, clock and SMBALERT#).

CTRL (PMBus CONTROL)

The PMBus CONTROL signal (labeled as CTRL on the product datasheets) can be used to turn the regulator output on and off.

How the CTRL signal works is configured with the PMBus ON_OFF_CONFIG command. For example, the CTRL signal can be configured as active high or active low. Please see AN302, PMBus Command Set, and the PMBus specifications for the details.

The CTRL signal can be driven with a general purpose I/O signal that operates from either 3.3 V or 5.0 V power supply.

WP (Write Protect)

The PMBus specification also provides for a Write Protect (WP) command that can be used to prevent unwanted or unauthorized changes to the PMBus device configuration. Our digital products do not provide a Write Protect pin and this functionality is not supported.

PMBus variations from SMBus specification

Signals

The PMBus specification adds two signals, CTRL (CONTROL) and WRITE PROTECT (WP) that are not in the SMBus specification. These are described below.

Speed

The maximum bus speed described in the SMBus specification (v3.2) is 1 MHz. The PMBus specification (v1.4) also allows bus speeds up to 1 MHz.

All of our digital devices can operate at 100 kHz. Our latest digital intermediate bus converters and POL regulators can also support bus speeds of 400 kHz and 1 MHz, but please check individual datasheets for full compatability information.

It is possible to have both devices on the same bus and communicate with them at different data rates (bus speeds). However, this requires care on the part of the programming of the Main device. If possible, operating the bus only at the lowest maximum speed supported by any device on the bus is the recommended practice.
If the bus is operated at 400 kHz or 1 MHz, it is up to the system engineer to assure that all timing parameters are met under all conditions.

GROUP protocol

With the standard SMBus transaction protocols and the PMBus requirement that a PMBus device start processing the received command when the STOP condition is detected, only one device can be given a command at a time. It is not possible to send commands to multiple PMBus devices and have them respond simultaneously.

To eliminate this restriction, the PMBus specification added the GROUP protocol. This protocol uses REPEATED START conditions to essentially send commands to many PMBus devices in one bus transaction. At the end of the transaction, when the STOP condition is detected, the multiple PMBus devices start processing the received commands (which do not have to be the same command for each device) simultaneously.

This would be useful, for example, during margin testing. All of the PMBus devices on the bus could be commanded to change their margin states simultaneously.

When using a general purpose microcontroller to manage multiple PMBus devices there can be a problem with the GROUP protocol. Most general purpose microcontrollers with hardware I²C interfaces do not support the multiple REPEATED START conditions in one transaction. Consult the microcontroller manufacturer's documentation to determine whether or not the GROUP protocol can be used in your system.

Summary

This application note has shown how to use a general purpose microcontroller to interface with Flex Power Modules digital products using the PMBus protocol. Firmware engineers writing code for this purpose are strongly encouraged to read the SMBus and PMBus specifications for more detailed information.

The first key concept is the structure of the PMBus transactions over the SMBus.
First, it is important to understand the difference in packet construction when writing data to a PMBus device and when reading from a PMBus device.
Next, the various data formats must also be understood and applied properly. Examples were given to show how data is converted to and from real world values and the PMBus data formats.
The use of the SMBus packet error checking protocol was then explained. While the 8 bit CRC checksum is not perfect and does not provide a means to correct errors, a match of the checksums provides high confidence that the data was received correctly.
Finally, the use of the SALERT signal and the SMBus SMBAlert protocol was explained. The SALERT signal provides the PMBus devices a way to quickly signal the Main that a device has a warning, fault, or other condition that needs attention. This eliminates the need for the Main to constantly and continuously poll the PMBus devices for their status, reducing the load on the Main device as well as minimizing traffic on the bus

Appendix 1: Example PEC checksum calculation code

The code below is provided as an example of one way to calculate the SMBus PEC checksum using the direct method.

```
/* pec.c
 * Implements a CRC-8 checksum using the direct method.
*/

pec.c

#include  "PEC.h"

/* CRC_Process_Byte performs a direct-mode calculation of
   one byte along with a previous CRC calculation */
void PEC_ProcessByte( uint8  crcInput )
{
   uint16 crcTemp;
   uint16 polyTemp = CRCPoly;    /* The polynomial shifts as
                                  * opposed to CRC */
   uint16 testMask = 0x8000;     /* testMask is used to evaluate
                                  * whether we should XOR */

/* XOR previous CRC and current input for multi-byte CRC
 * calculations.  The temporary result, crcTemp, is shifted
 * left one byte to perform direct mode calculation */

   crcTemp = ( ( ( uint16 )( PEC_CurrentCRC ^ crcInput ) )<<8 );
   while( polyTemp != CRCDone ){
      if( crcTemp & testMask )
         crcTemp = crcTemp ^ polyTemp;
      testMask = testMask>>1;
      polyTemp = polyTemp>>1;
   };

/* Update the current value of the PEC with the new result */
 * PEC_CurrentCRC = ( uint8 )crcTemp;
}


/* CRC_Reset will reset PEC_CurrentCRC to 0.  This
 * should be called before a new multi-byte calculation
 * needs to be done */
void PEC_ResetCRC( void ){

   PEC_CurrentCRC = 0;
}

/* pec.h
 * Packet Error Checking Header File
 * Contains Defines and Prototypes for pec.c
 */

#define CRCPoly  0x8380     /* The CRC polynomial of
                             * x^8 + x^2 + x^1 + 1 is in
                             * the most significant 9 bits. 8/

#define CRCDone  0x0083     /* CRC is done after the polynomial
                             * shifts one byte. */

/* Public global values */

extern uint8 PEC_CurrentCRC;     /* The CRC calculation result
                                  * of all bytes called through
                                  * CRC_Process_Byte since the
                                  * last call to CRC_Reset */

/* Public prototypes */

Void PEC_ProcessByte( uint8 crcInput );
Void PEC_ResetCRC( void );
```

flex. Power Modules

Flex Power Modules, a business line of Flex, is a leading manufacturer and solution provider of scalable DC/DC power converters primarily serving the data processing, communications, industrial and transportation markets. Offering a wide range of both isolated and non-isolated solutions, its digitally-enabled DC/DC converters include PMBus compatibility supported by the powerful Flex Power Designer.

EMEA (Headquarters) | Torshamnsgatan 28 A, 16440 Kista, Sweden

APAC | 33 Fuhua Road, Jiading District, Shanghai, China 201818

Americas | 6201 America Center Drive, San Jose, CA 95002, USA

pm.info@flex.com

flexpowermodules.com

flexpowerdesigner.com

youtube.com/flexintl

twitter.com/flexpowermodule

flexpowermodules.com/wechat

linkedin.com/showcase/flex-power-modules