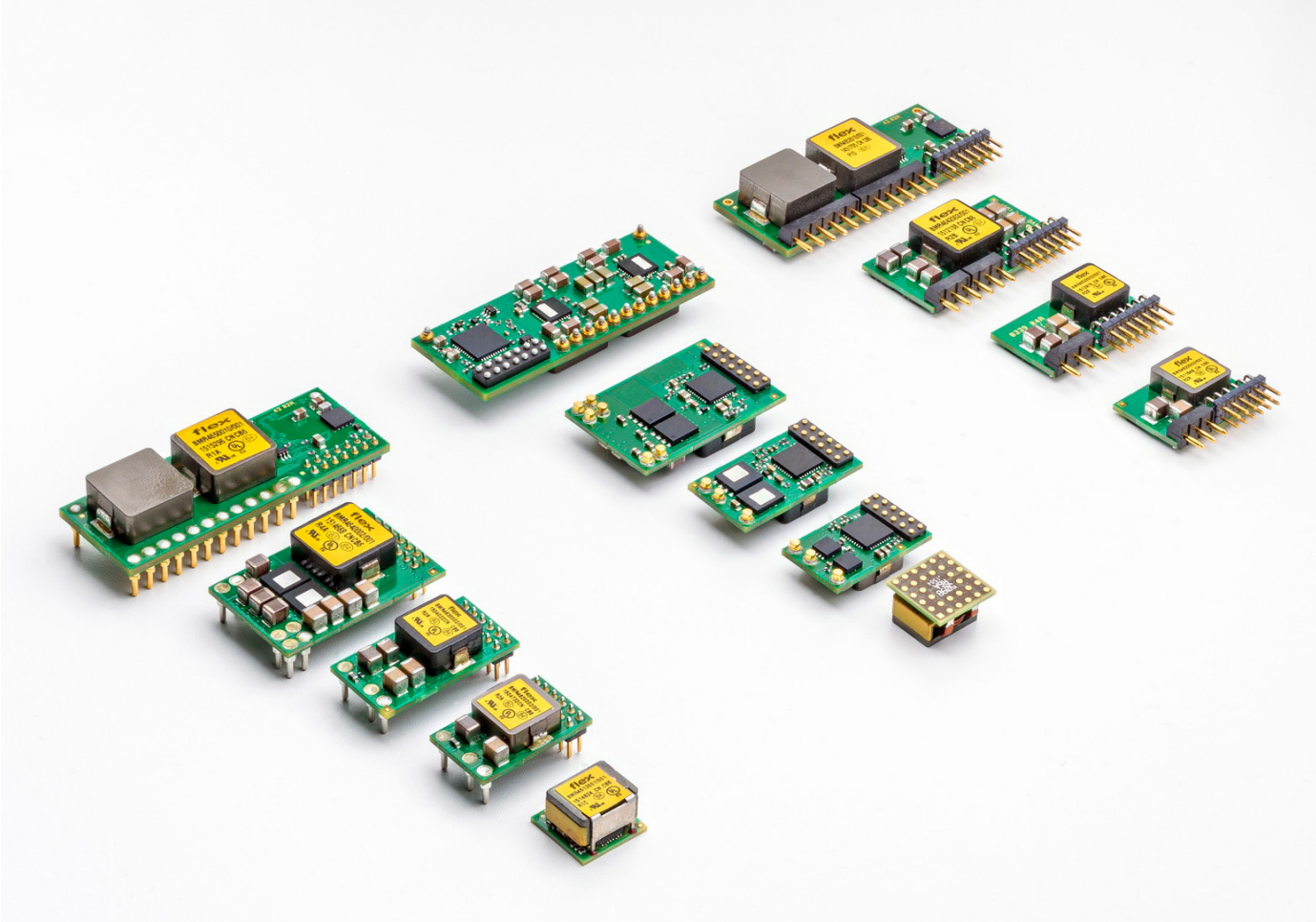
Application Note 304

Flex Power Modules



Microcontroller Programming for 3E Digital Products

Abstract

The 3E 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 3E Digital products and accessing the 3E 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 a 3E 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 I^2C 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 I^2C 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 micro-controller with a hardware I^2C or SMBus port is being used.

Introduction

Flex's 3E 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. 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. The PMBus specifications are freely available at the PMBus Web site,

http://www.pmbus.org

In addition to the capabilities provided by the PMBus, Flex's 3E POL regulators feature the Group Command Bus (GCB). The GCB is an inter-device communication bus that provides addition capabilities like digital current sharing and fault propagation management.

This document, to be used in conjunction with the PMBus specifications and the Technical Specifications for each product, describes how to interface a microcontroller based system controller to the Flex 3E 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.

Contents

Forum Websites

The System Management Interface Forum (SMIF)
http://www.powersig.org/
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)
http://pmbus.org/
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 2.0, August 3, 2000
This specification specifies the version of the SMBus on which Revision 1.2 of the PMBus Specification is based. This specification is freely available from the System Management Interface Forum Web site at:

http://www.smbus.org/specs/

Applicability

This document applies to all versions of the following products:

- BMR450 digital POL regulator
- BMR451 digital POL regulator
- BMR46x digital POL regulators
- BMR 453 digital intermediate bus converter
- BMR 454 digital intermediate bus converter

Most PMBus commands have the same data format and effect for these products. However, there are some exceptions. For example, a command may be available for the BMR450 and BMR451 products but not the BMR46x products. In other cases the same command code may have different meanings and effects depending on the specific model. The details of these differences are described in AN302 PMBus Command Set and the technical specification for each product.

SMBus Basics

We start with an introduction to the SMBus because the PMBus protocol is based on SMBus Version 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 the 3E Digital products via the serial bus interface.

For a detailed description of how the SMBus differs from the I²C bus, including detailed electrical and timing characteristics, please see Appendix B – Differences between SMBus and I²C of the SMBus Specification (Version 2.0)

SMBus Signals

The SMBus has two required signal wires. SMBCLK is the clock signal that is generated only by the bus master device. SMBDATA is a bi-directional signal that is used to transfer data between the master and the slave devices. For the 3E Digital products, these signals are identified as SCL and SDA, respectively.

The SMBus specification also includes an interrupt signal, SMBALERT#, that a slave device can use to notify the bus master device that it has information for the bus master. This is used by the 3E 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 2.0, August 3, 2000.

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 bus master 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.

The 3E products use resistors to program the bus address of each device. The device address is not configurable through the serial bus interface. Consult the technical specification for the 3E 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 bus master initiates a transaction by placing a START condition on the bus. This alerts all of the slave devices to start listening.

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

Each slave 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 master 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 bus master device. Upon receiving the last data byte,

the bus master does not acknowledge (“NACKs”) the byte.

The transaction is completed when the bus master 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.

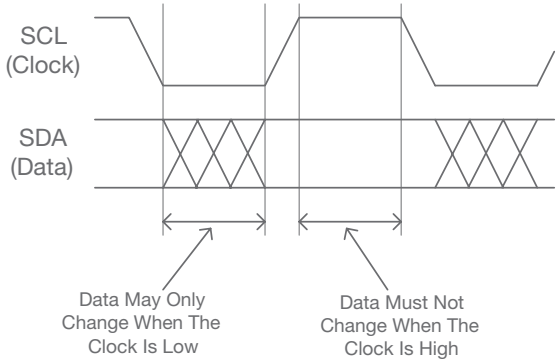


Figure 1. SMBus Bit Transfer

The bus master device always drives the clock (SCL) line. When the master 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 slave device transferring data to the bus master or it may be the bus master writing data to a slave device. The data line is allowed to change state only during the time that the clock is low.

The bus master 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 bus master puts a START condition on the bus and are ended when the bus master 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 master 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.

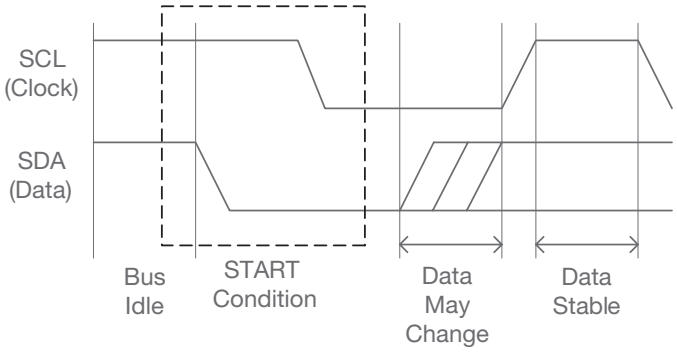


Figure 2. SMBus START Condition

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

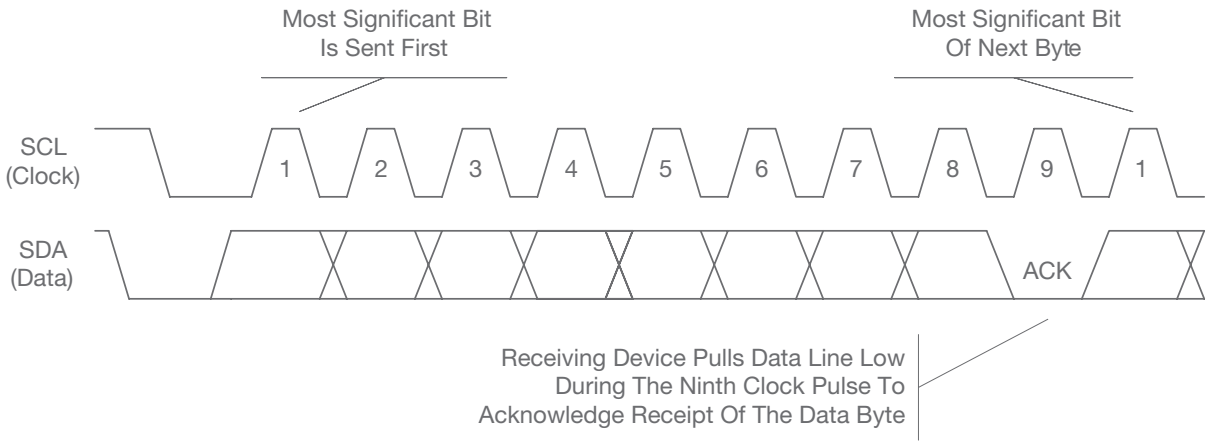


Figure 3. SMBus Byte Transfer With Acknowledge (ACK)

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.

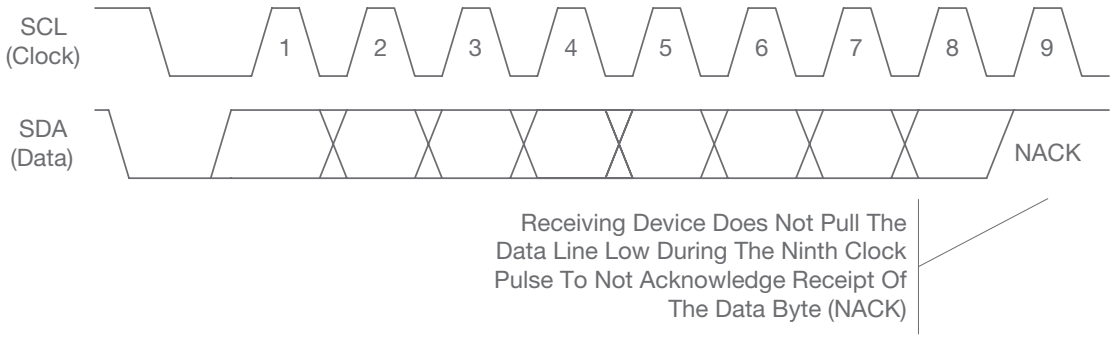


Figure 4. SMBus Data Byte Without Acknowledgement (NACK)

Other than a master device not acknowledging the last data byte of a read from a slave 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 3E regulators and bus converters, through the comprehensive status and fault reporting in the PMBus protocol specification, can provide the bus master information that will generally let the master know why a data byte was NACK'ed.

STOP Condition

A master device notifies all devices on the bus that the current transaction is complete by placing a STOP condition on the bus. Figure 5 shows s STOP condition after an ACK of the last data byte of the transaction. After the ACK, when the data line is low, the master 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 master 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.

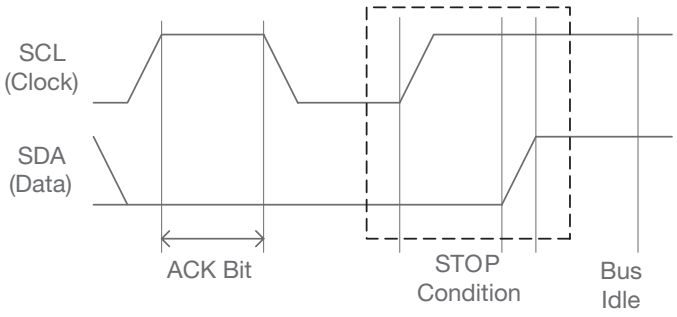


Figure 5. STOP condition after ACK

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 curing the ninth clock period. After the ninth clock period is complete, the master pulls the clock line and the data line low. The master then releases the clock line and allows it to go high. After the specified delay time, the master 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.

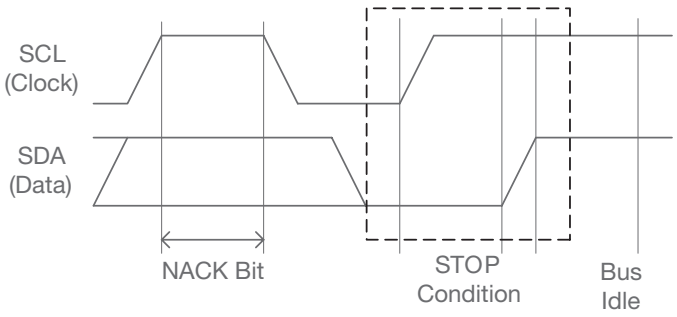


Figure 6. STOP condition after a NACK

Is should be noted that a master may put STOP condition can appear 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.

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 slave 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 bus master must be prepared to accept clock stretching by a slave device.

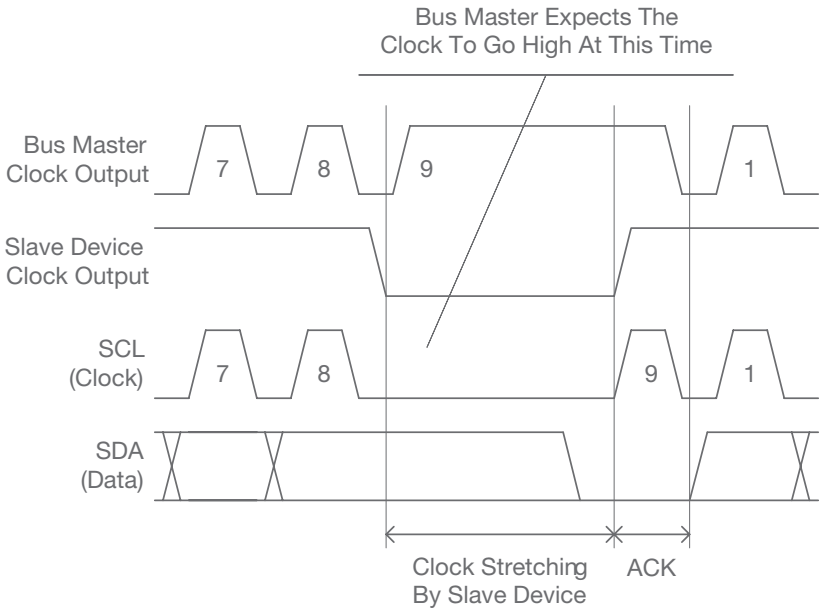


Figure 7. Clock stretching by a SMBus slave device

Figure 7 illustrates how clock stretching by a slave device works and how it can cause problem for a bus master that is not SMBus compliant. In this case the bus master is sending data to the slave device. After the eighth bit, the slave device needs time to validate the data in the byte and decide whether to ACK or NACK. After the master lowers the clock to end the eighth bit, the slave 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 bus master, in its normal timing cycle, turns off its clock output for the ninth (acknowledge) bit. However, because the slave device is holding the clock line low, the clock line does not go high. For a master device that cannot handle clock stretching by a slave device, this would be an error condition. That master would typically terminate the transaction and set an error flag.

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

The best way to avoid problems with clock stretching by slave devices is to make sure that the bus master 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 bus master is SMBus compliant and will accept clock stretching by a slave 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 you carefully study the SMBus specification.

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

Bus master 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 bus master 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 I²C bus which a clock line held low indefinitely is a valid condition (minimum clock speed is 0 Hz).

And again, it best that these timing details be 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 I²C 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 2.0).

Only the bus master 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 an 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 bus master device to write a single byte of data to a slave device.

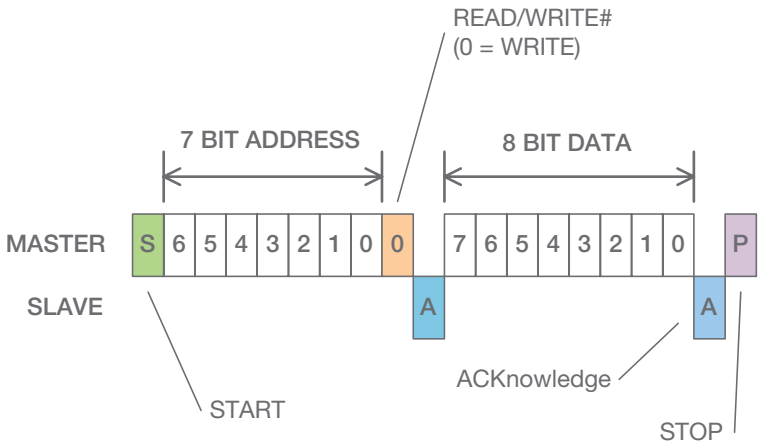


Figure 8. SMBus WRITE BYTE protocol

The transaction proceeds as follows:

- The master device puts a START condition on the bus to notify the slave devices that a transaction is beginning.
- The master 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 master writes data to the slave device).
- The receiving device acknowledges its address and that it is ready to receive data (ACK).
- The master device sends the data byte.
- The receiving device ACKs the received data byte.
- The master device puts a STOP condition on the bus to notify the slave 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 master device uses to read two bytes of data from a slave device.

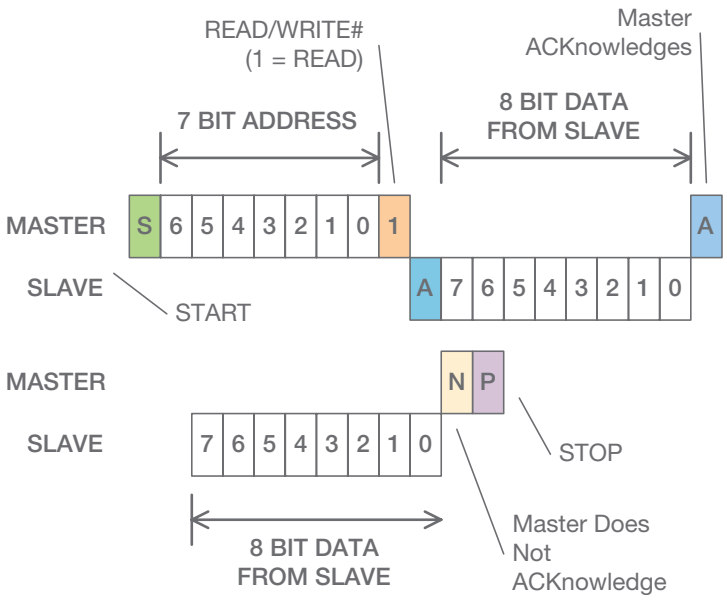


Figure 9. SMBus READ WORD protocol

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

- The master device puts a START condition on the bus to notify the slave devices that a transaction is beginning.
- The master 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 master reads data from the slave device).
- The receiving device acknowledges its address and that it is ready to send data (ACK).
- The slave device sends the first data byte. Remember that if this is a 16 bit value, this is the lower order byte.
- The bus master ACKs the received data byte.
- The slave device sends the second data byte. Remember that if this is a 16 bit value, this is the higher order byte.
- The master device does not acknowledge (“NACKs”) the received packet error checking data.
- The master device puts a STOP condition on the bus to notify the slave devices that the transaction is complete.

PMBus Data Formats

Before discussing how commands are sent to a 3E product over the PMBus, it is important to understand the formats of the data being written to or read from a 3E product.

The data for a PMBus command used with a 3E 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 3E 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.

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

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

Data Format For The 3E Digital POL Regulators

Data Format

For the 3E 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	Data Format	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)

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:

$$\frac{3.3\,\mathrm{V}}{2^{-13}\,\frac{\mathrm{V}}{\mathrm{bit}}} = \frac{3.3\,\mathrm{V}}{122.07\,\frac{\mu\mathrm{V}}{\mathrm{bit}}} = 27033.6 \Rightarrow 27034 = 0\mathrm{x}699\mathrm{A}$$

Remembering that in the SMBus protocols the low bytes are transmitted first, the master 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:

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

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

Data Format For The 3E Digital Intermediate Bus Converters

Data Format

For the 3E 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	Data Format	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)

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 master 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 master 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 3E Digital regulators and the 3E 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.

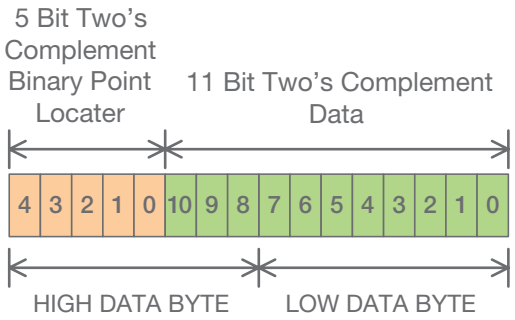


Figure 10. 11 bit Linear format data byte structure

Table 2 shows the possible ranges of values and resolutions that can be expressed with the 11 Bit Linear format. Note that for any given 5 bit binary point locater value, the smallest resolution is about 0.1% (1 part in 1024).

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

Table 2. 11 Bit Linear Format Data Ranges And Resolutions

Example 1: 11 Bit Linear Data To Be Written To A PMBus Device

Suppose the output over current threshold is to be set to 10 A using the IOUT_OC_FAULT_LIMIT covmmand. 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:

$$N = integer\left(\frac{\log\left(\frac{10}{1023}\right)}{\log(2)}\right) = integer\left(-6.677\right) = -6 = 0\text{b}11010$$

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:

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

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

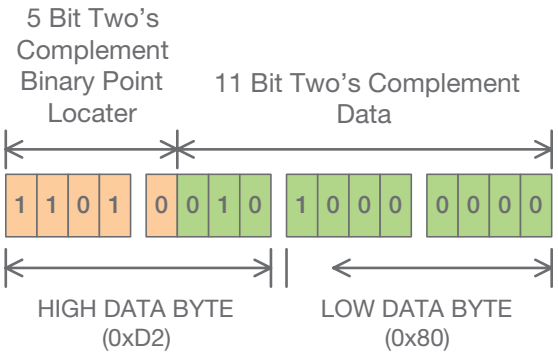


Figure 11. Data bytes for 11 bit Linear Format example 1

Remembering that the low order byte is transmitted first, the master 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 3E 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:

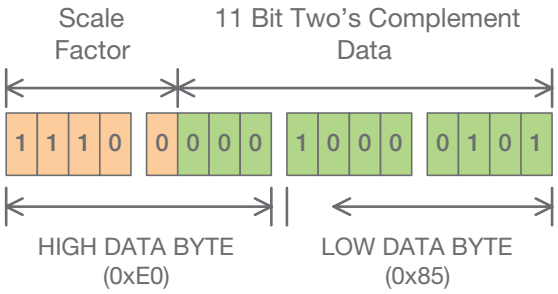


Figure 12. Data bytes for 11 bit Linear Format example 2

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:

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

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 slave 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.

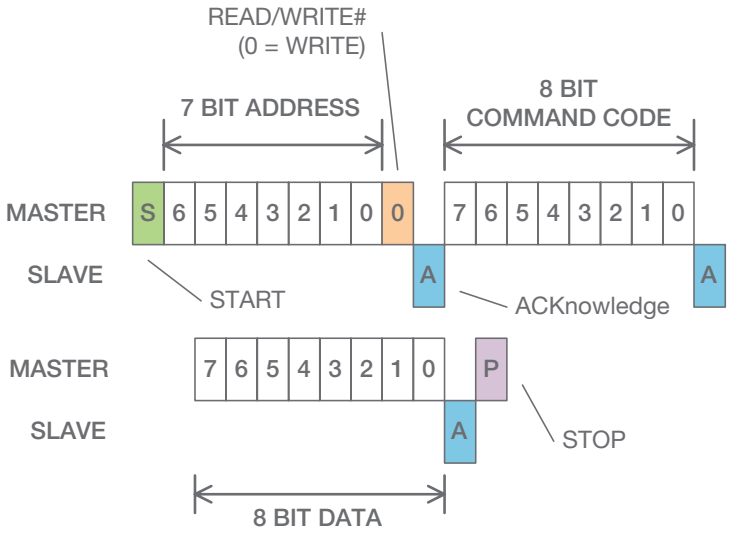


Figure 13. Example PMBus transaction protocol with one data byte written to the PMBus device

This transaction proceeds as follows:

- The master device puts a START condition on the bus to notify the slave devices that a transaction is beginning.
- The master 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 master writes data to the PMBus device).
- The PMBus device acknowledges its address and that it is ready to receive data (ACK).
- The bus master device sends the one byte command code.
- The PMBus device ACKs the received command code.
- The bus master sends the data byte that goes with the command.
- The PMBus device ACKs the received data byte.
- The master device puts a STOP condition on the bus to notify the slave devices that the transaction is complete.

Example PMBus Transaction 2: A Command That Reads Two Bytes Of Data From A PMBus Device

Note that every PMBus command will start with a write of the command code. For PMBus commands that need to read data from the PMBus device, the first part of the command is sent but the master does not end the transaction with a STOP condition. The master then sends another START condition (called a REPEATED START) with the same 7 bit address and the READ/WRITE# bit set to 1 for a read operation. The PMBus device then sends the data to the master. Figure 14 illustrates a PMBus command that has the master reading two bytes of data from a PMBus device (READ WORD protocol).

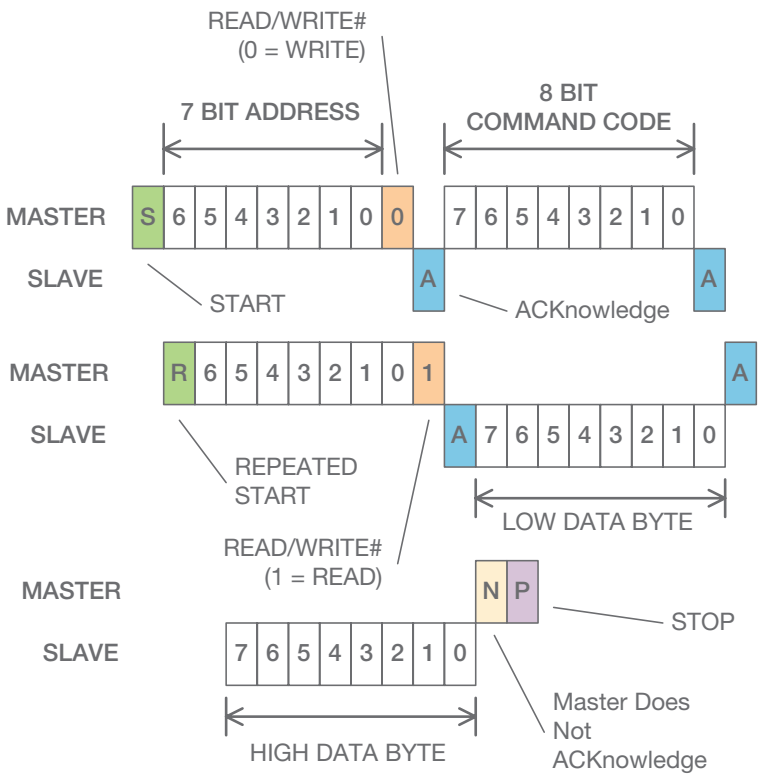


Figure 14. PMBus command that reads data from a PMBus device

The transaction proceeds as follows:

- The master device puts a START condition on the bus to notify the slave devices that a transaction is beginning.
- The master 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 master writes data to the slave device).
- The PMBus device acknowledges its address and that it is ready to receive data (ACK).
- The bus master device sends the one byte command code.
    - The PMBus device ACKs the received command code.
    - The master device puts a REPEATED START condition on the bus to notify the slave devices that a transaction is beginning.
    - The master 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 master 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 master device ACKs the received data byte.
    - The PMBus device sends the second data byte for the received command code.
    - The master device does not acknowledge (NACKs) the received data byte.
    - The master device puts a STOP condition on the bus to notify the slave 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

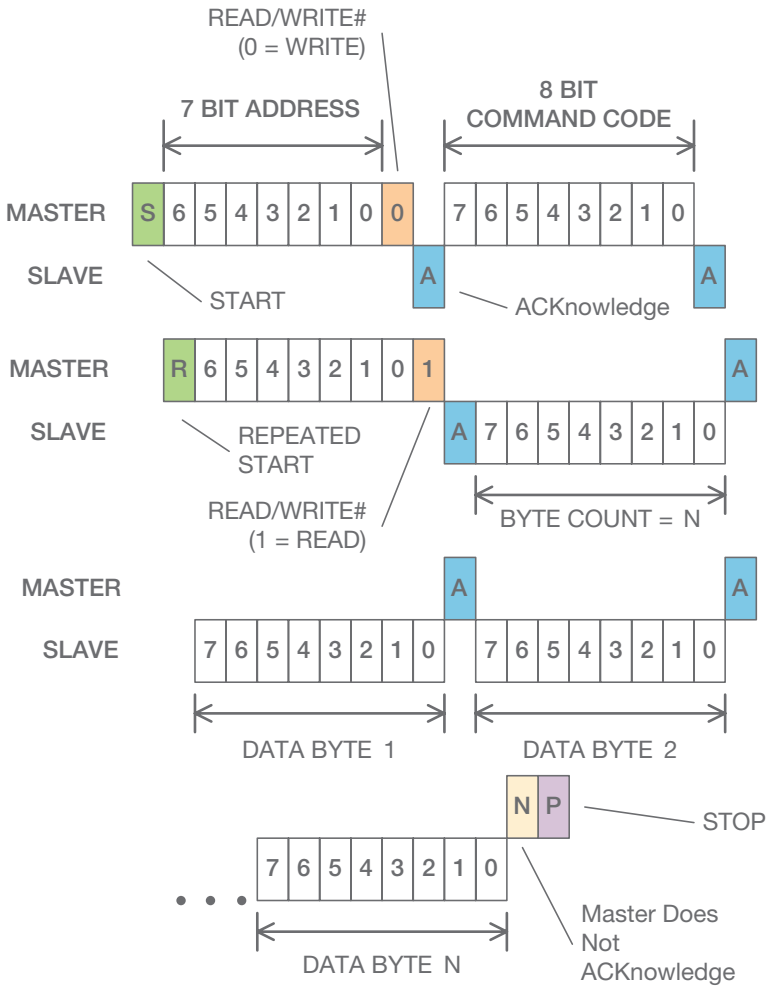


Figure 15. PMBus command that reads a block of data from a PMBus device

The transaction proceeds as follows:

- The master device puts a START condition on the bus to notify the slave devices that a transaction is beginning.
- The master 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 master writes data to the slave device).
- The PMBus device acknowledges its address and that it is ready to receive data (ACK).
- The bus master device sends the one byte command code.
- The PMBus device ACKs the received command code.
- The master device puts a REPEATED START condition on the bus to notify the slave devices that a transaction is beginning.
- The master 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 master 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 master device ACKs the received data byte with the byte count.
- The PMBus device sends the first data byte for the received command code.
- The master device ACKs the received data byte.
- The PMBus device sends the second data byte for the received command code.
- The master device ACKs the received data byte.
- The sending of data bytes and acknowledgment by the master continues until the PMBus device sends the Nth data byte.
- The master device does not acknowledge (NACKs) the reception of the Nth data byte.
- The master device puts a STOP condition on the bus to notify the slave 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 bus master 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 3E 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 master device firmware engineer.

Checking PEC Support In A Slave Device

Before using packet error checking with a PMBus device, the master 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.

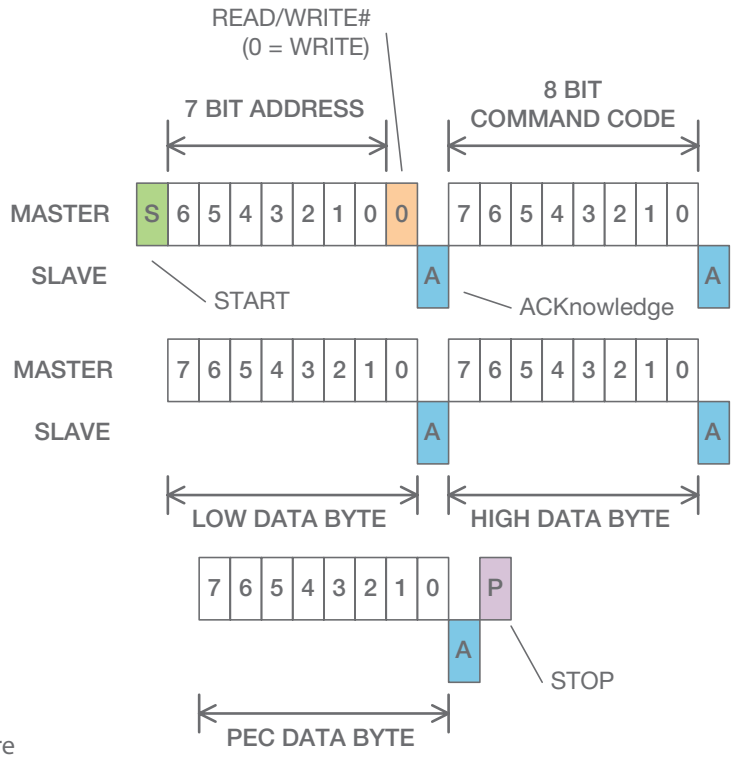


Figure 16. PMBus WRITE WORD command with packet error checking byte

The transaction proceeds as follows:

- The master device puts a START condition on the bus to notify the slave devices that a transaction is beginning.
- The master 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 master writes data to the PMBus device).
- The PMBus device acknowledges its address and that it is ready to receive data (ACK).
- The bus master device sends the one byte command code.
- The PMBus device ACKs the received command code.
- The bus master sends the low data byte.
- The PMBus device ACKs the received data byte.
- The bus master sends the high data byte.
- The PMBus device ACKs the received data byte.
- The bus master sends the packet error checking data byte.
- The PMBus device ACKs the received data byte.
- The master device puts a STOP condition on the bus to notify the slave devices that the transaction is complete.

Reading Data With A PEC Byte

Figure 17 illustrates a PMBus READ WORD transaction using packet error checking.

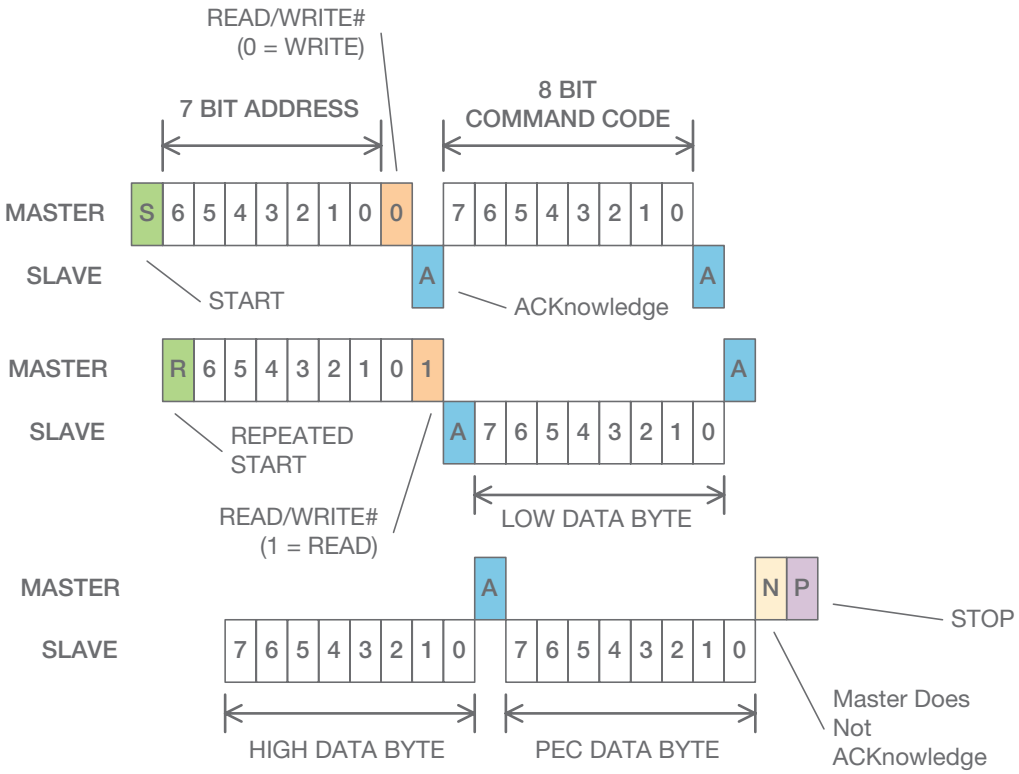


Figure 17. PMBus READ WORD command with packet error checking byte

The transaction proceeds as follows:

- The master device puts a START condition on the bus to notify the slave devices that a transaction is beginning.
- The master 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 master writes data to the slave device).
- The PMBus device acknowledges its address and that it is ready to receive data (ACK).
- The bus master device sends the one byte command code.
- The PMBus device ACKs the received command code.
- The master device puts a REPEATED START condition on the bus to notify the slave devices that a transaction is beginning.
- The master 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 master 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 master device ACKs the received data byte.
- The PMBus device sends the second data byte for the received command code.
- The master device ACKs the received data byte. This notifies the PMBus device that the master is expecting another data byte.
- The PMBus device sends the packet error checking data byte.
- The master device does not acknowledge (NACKs) the packet error checking data byte.
- The master device puts a STOP condition on the bus to notify the slave 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.

Using Packet Error Checking With A PMBus Device That Does Not Support Packet Error Checking

If the master 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 master 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 master is requesting the data bits) and
- Declare a communications fault because the master 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 master 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 bus master device should be notified. One way to do that would be for the PMBus device to become a bus master and send a message. However, multi-master bus systems are not favored due to issues with congestion and conflict.

The notification method that is preferred, and implemented in the 3E products, is for the PMBus device to use a separate, dedicated signal line to notify the bus master of a change of condition. That signal in the 3E products is the SALERT.

One possible implementation is for each 3E product to have its SALERT signal connected to a dedicated input on the bus master device. With this implementation the bus master knows instantly and unambiguously which 3E 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 bus master microcontroller.

Another possible implementation is to have one SALERT line that is common to all of the 3E products and that terminate on one I/O pin of the microcontroller. With this approach, the bus master must use the SMBALERT# protocol to determine which 3E product or products need attention.

An important aspect of the alert protocol is that the SALERT outputs on the 3E products are all open drain. It is possible that more than one 3E 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 3E 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 18.

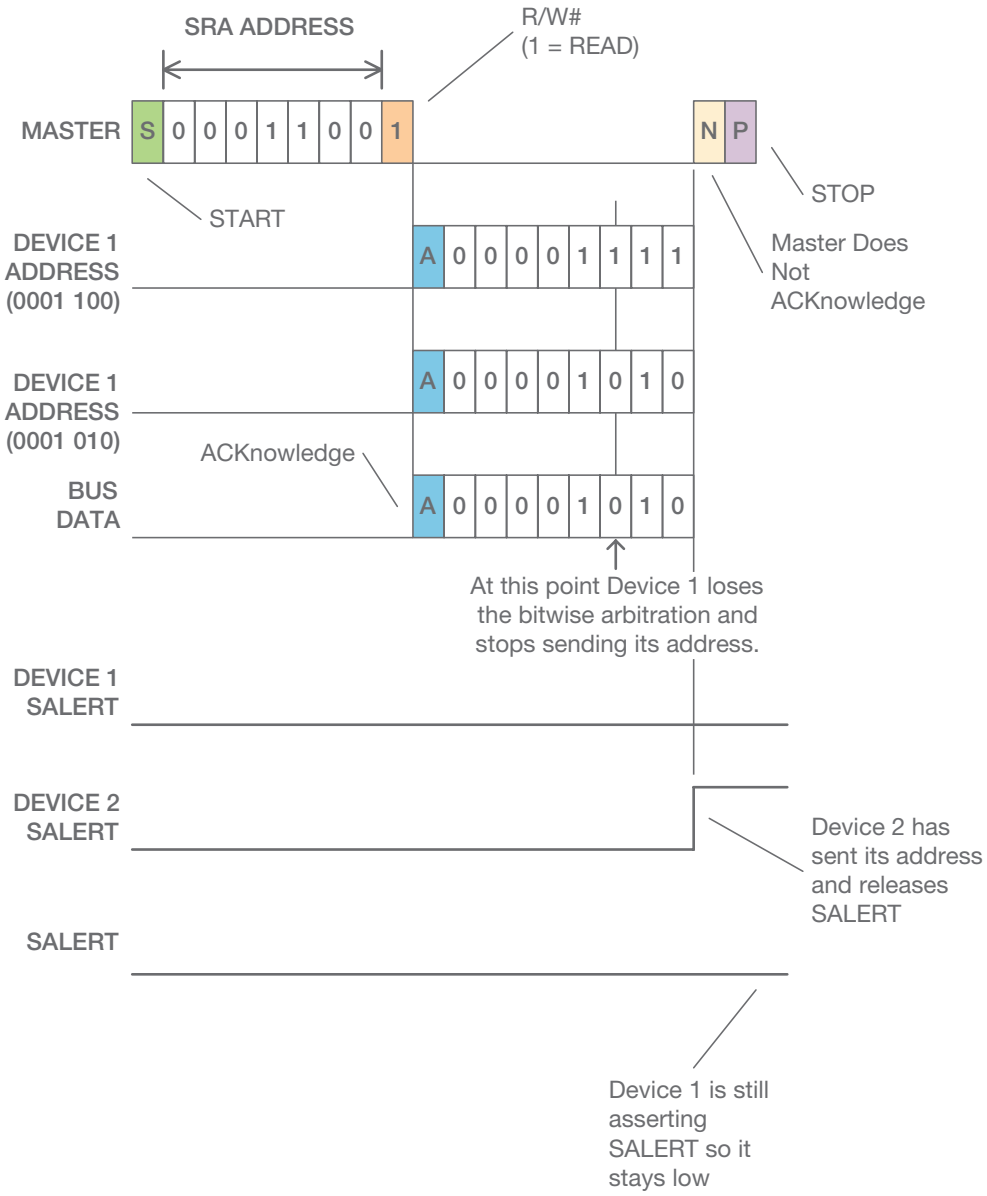


Figure 18. Alert response bitwise arbitration

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

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

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

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

At that point, the bus master 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 master can decide to take action on the information it has to make inquiries to the slave 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 3E product technical specifications) 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.

The 3E 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 above.

Speed

The maximum bus speed described in the SMBus specification is 100 kHz. The PMBus specification, by requiring the higher drive level outputs and some differences in timing, allows bus speeds to 400 kHz.

The 3E digital intermediate bus converter support bus speeds up to 400 kHz. However, the digital POL regulators only support data rates up to 100 kHz.

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 master device. If possible, operating the bus only at the lowest maximum speed supported by any device on the bus is the recommended practice. In the case with both digital intermediate bus converters and digital POL regulators on the bus, a maximum bus speed of 100 kHz is recommended.

If the bus is operated at 400 kHz, 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 I^2C 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 3E 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 master that a device has a warning, fault, or other condition that needs attention. This eliminates the need for the master to constantly and continuously poll the PMBus devices for their status, reducing the load on the master device as well as minimizing traffic on the bus.

Reference Documents

Flex Technical Specifications For The Following Products:
BMR450 Digital POL Regulators
BMR451 Digital POL Regulators
BMR462 Digital POL Regulators
BMR463 Digital POL Regulators
BMR464 Digital POL Regulators
BMR453 Digital Intermediate Bus Converters
BMR454 Digital Intermediate Bus Converters

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 );
```

Formed in the late seventies, Flex Power Modules is a division of Flex that primarily designs and manufactures isolated DC/DC converters and non-isolated voltage products such as point-of-load units ranging in output power from 1 W to 700 W. The products are aimed at (but not limited to) the new generation of ICT (information and communication technology) equipment where systems' architects are designing boards for optimized control and reduced power consumption.

Flex Power Modules
Torshamnsgatan 28 A
164 94 Kista, Sweden
Email: pm.info@flex.com

Flex Power Modules - Americas
600 Shiloh Road
Plano, Texas 75074, USA
Telephone: +1-469-229-1000

Flex Power Modules - Asia/Pacific
Flex Electronics Shanghai Co., Ltd
33 Fuhua Road, Jiading District
Shanghai 201818, China
Telephone: +86 21 5990 3258-26093