Implementing Robust PMBus System Software for the LTC3880

Nick Vergunst

INTRODUCTION

The LTC®3880 is a dual output PolyPhase® step-down DC/DC controller with integrated digital power system management. The LTC3880 can easily be controlled through a PMBus interface, which builds upon I^2C/IIC at speeds up to 400kHz. System telemetry data can be obtained quickly and painlessly with simple polling functions. This exposes all important and critical information to the system developer such as the voltage and current readings for the input and both outputs, temperature, fault conditions, general status information, and more.

This application note discusses the design requirements in regards to implementing robust firmware capable of interacting with the LTC3880.

DOCUMENT OVERVIEW

This document is broad in scope. If you already feel comfortable with I^2C/SMBus/PMBus and the transaction structure for the command set, then you may wish to skip ahead to the Communicating Robustly with the LTC3880 section.

REFERENCE DESIGN OVERVIEW

The PMBus master design used in this application note is an off-the-shelf microcontroller (Microchip PIC32MX), but any microcontroller, FPGA, or other device with a programmable PMBus/I^2C interface can be used. All the code examples are written for the PIC32MX architecture, but they are easily adapted to other target designs.

PMBus vs SMBus vs I^2C/IIC INTRODUCTION

The power management bus (PMBus) 2-wire interface is an incremental extension of the system management bus (SMBus). SMBus is built upon I^2C with some differences in timing, DC parameters, and protocol. The PMBus/SMBus protocols are more robust than simple I^2C byte commands because PMBus/SMBus provide timeouts to prevent bus hangups and optional packet error checking (PEC) to ensure data integrity.

In general, a master device configured for I^2C communication is used for PMBus communication with little or no change to hardware or firmware. Repeated starts (restarts) are not supported by all I^2C controllers but are required for PMBus/SMBus reads. If a general purpose I^2C controller is used, check that repeated starts are supported.

For a full description of the extensions and exceptions PMBus makes to SMBus, refer to PMBus Specification Part 1 Revision 1.1: Paragraph 5: Transport.

For a full description of the differences between SMBus and I^2C, refer to System Management Bus (SMBus) Specification Version 2.0: Appendix B—Differences Between SMBus and I^2C.

Quick Overview of PMBus vs SMBus vs I^2C

The power management bus (PMBus) protocol is built upon the system management bus (SMBus) which builds upon the 2-wire open drain communication interface inter-integrated circuit (I^2C or IIC).

PMBus dictates a maximum bus speed of 400kHz and has built-in timeouts important for critical systems. Though clock stretching is fully PMBus compliant, it is not required with the LTC3880 if operating at bus speeds at or below 100kHz and the guidelines below are followed. Because of the built-in timeouts, a minimum bus speed of 10kHz exists for all PMBus devices.

Some of the key differences between PMBus and SMBus in relation to the low level bus are:

- 400kHz maximum bus speed vs 100kHz SMBus limit
- Group protocol
- Block reads up to 255 bytes in length vs 32 byte SMBus limit

Multiple starts without stops (i.e. repeated starts) are used as per PMBus/SMBus specification. The host processor shall have support for these features to read from PMBus products such as the LTC3880.

PMBus Transaction Formats

PMBus has several transaction formats that should be supported: send byte, write byte, write word, read byte, and read word. For full support it is recommended you also support the PMBus read block transaction (up to 255 bytes).

The read block transaction is used to read more than three byte data streams such as the real time clock, fault log, and identification strings. Master devices should also support non-SMBus defined PMBus group command protocol.

In addition, all of these transaction formats may include a packet error checking (PEC) byte at the end of the stream to verify the validity of the data stream as a whole.

The required data ordering specifies the most significant bit (MSb) of the least significant byte (LSB) is always sent first. This makes the byte order endianness little-endian and the bit order endianness big-endian.

The eighth bit in the address byte indicates whether it is a read (value of 1) or a write (value of 0). Anytime an acknowledge is expected and not received, a communication error has occurred and the transaction is cancelled.

The gray sections in the PMBus sequence diagrams below indicates that the slave should be pulling the SDA line low to acknowledge (ACK) receipt of the byte.

PMBus Send Byte

The send byte transaction is used to send a simple command to the device. A send byte transaction transfers a command with no data. The CLEAR_FAULTS command that clears the current fault flags present in the system is an example of such a command.

A start bit, followed by the 7-bit slave address of the LTC3880 and finished by a write bit (0-value) to indicate a write make up the first stage of the transaction. If the slave ACKs the address, then the host sends the 8-bit command followed by a stop condition.

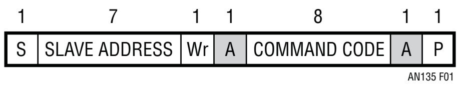


Figure 1. PMBus Send Byte

NOTE: The term endian or endianness refers to the ordering of individually addressable subcomponents within the representation of a larger data. These subcomponents can be bits, bytes, words, or an arbitrary length block.

The usual contrast is whether the most significant or least significant byte is ordered first —i.e. at the lowest byte address— within the larger data item. A big-endian machine stores the most significant byte first, and a little-endian machine stores the least significant byte first. In these standard forms, the bytes remain ordered by significance.

Big-endian is similar to how numbers are written in arabic numerals. Given the number 5000 we can break down the subcomponent to be a digit. Further we read this as big-endian style with the 5 being the most significant subcomponent giving us five thousand.

To transmit 0xA1B2C3D4 in a big-endian system, you would transfer 0xA1, then 0xB2, then 0xC3, and finally 0xD4. In a little-endian system like SMBus/PMBus you would transfer 0xD4, then 0xC3, then 0xB2, and finally 0xA1.

To transmit 0xA1 (0b10100001) in a big-endian bit order like SMBus/PMBus you would transfer 0b1 first (followed by 0b0, 0b1, 0b0, 0b0, 0b0, 0b0, and 0b1). In a little-endian bit order it would be 0b1, 0b0, 0b0, 0b0, 0b0, 0b1, 0b0, and then 0b1.

PMBus Write Byte

The write byte transaction is used to send single byte data to the LTC3880. The PAGE command that changes the current page of the device is an example of this type of transaction. Similar to the send byte transaction above, the series of start bit, 7-bit slave address of the LTC3880 with write bit (0-value), command byte, and finally the 8-bit data byte.

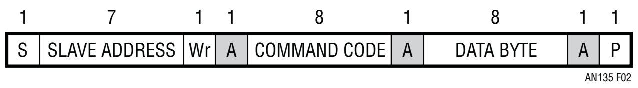


Figure 2. PMBus Write Byte

PMBus Write Word

The write word transaction is used to send a single word of data (two bytes) to the LTC3880. The VOUT_COMMAND command is an example of such a transaction. Similar to the write byte command, the only difference is that after the third acknowledge (the low data byte), the high byte is sent in addition.

PMBus Read Byte

The read byte starts out like a normal I^{2}C write transaction by sending the address and the write bit. Sending a write bit for a read command can be somewhat confusing for a novice user. See Note regarding Reading PMBus/SMBus. The second byte contains the command code, then a repeated start is sent, and following that is the address and read bit signalling the device to return data for the specified command code. The slave responds by transmitting the byte value requested and the host does **not** acknowledge (NACK) the data byte. The host NACKing the data may be confusing initially, but it is defined that way to convey that the host is finished asking for data. If during the transaction, the host NACKs the data, the LTC3880 stops transmitting and does not issue a CML fault.

PMBus Read Word

The read word transaction also starts out like a normal I^{2}C write transaction by sending the address and the write bit. The second byte contains the command code, then a repeated start is sent, and following that is the address and read bit signalling the device to return data for the specified command code. The slave responds by transmitting the value requested low byte first and high byte last. The host acknowledges (ACK) the reception of the low byte and does **not** acknowledge (NACK) the high byte.

PMBus Read Block

The read block transaction is used to read a block stream of data (up to 255 bytes) from the device. The MFR_FAULT_LOG command is an example of such a transaction. Similar to the other read commands, but the first data byte returned represents the byte count remaining for the block read. The host should then read the slave until byte-count data bytes are read in. The host acknowledges (ACK) the reception of all but the final byte and does **not** acknowledge (NACK) the final byte.

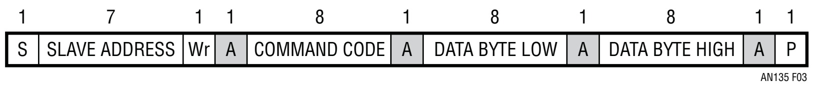


Figure 3. PMBus Write Word

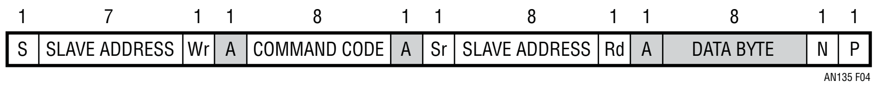


Figure 4. PMBus Read Byte

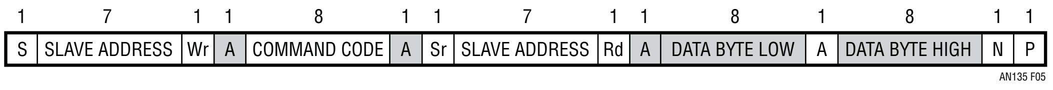


Figure 5. PMBus Read Word

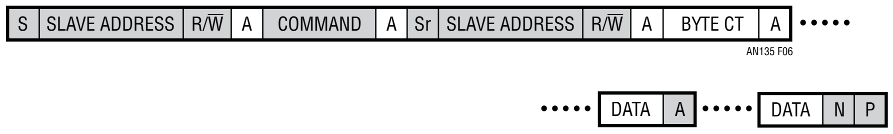
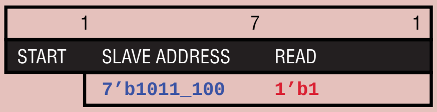


Figure 6. PMBus Read Block

Reading PMBus/SMBus

NOTE: A very common mistake made by firmware engineers familiar with I^{2}C but unfamiliar with PMBus/SMBus is attempting to read PMBus registers by immediately sending the address and read bit after the first start condition. **This is incorrect**.

The most common error made by I^{2}C experienced firmware engineers shown here:



Though the above protocol would be a valid I^{2}C read, it is **NOT** a valid SMBus Read. PMBus/SMBus read byte/word transactions **always start with the address and write bit** as shown in the LTpowerPlay™ protocol diagrams above, **not** with the address and read bit.

Multi-Master Mode

There exists the option to have a multi-master system within PMBus/SMBus where multiple devices can initiate transactions and provide the clock signal. The LTC3880 and family fully supports this, and if this is an option you are contemplating, special consideration should be given to the selection of your host IC's and the firmware written for them. One possible reason to build in support for such an option is in debug scenarios where you can have both an I^{2}C debugger and your system connected at the same time gathering telemetry and status pseudo-simultaneously.

Packet Error Checking (PEC)

All transactions may have an optional PEC byte appended to the end before the stop bit. If the PEC_REQUIRED bit of MFR_CONFIG_ALL register is set, then all writes must be accompanied by the matching PEC computation.

For writes when PEC_REQUIRED is enabled, the host must compute the PEC value and append it to the end of the transaction between the last byte's acknowledge and the stop bit. The LTC3880 then compares this PEC byte with its own internally calculated PEC value based on the data received during the current transaction. If the received PEC byte matches the internally computed PEC byte, the LTC3880 ACKs the transaction and the data is accepted as valid. The data is then copied to the write command data buffer (See PMBus Communication and Command Processing section of the data sheet). If the PEC bytes mismatch, the LTC3880 will NACK the transaction and a CML fault is logged to indicate the failure. In addition to the NACK and fault response, the LTC3880 does not act upon the data received. If PEC_REQUIRED is enabled and a PEC byte is **not** sent, the LTC3880 will ACK the command but not process the data.

When PEC_REQUIRED is disabled and a write with a PEC byte is transmit to the LTC3880 it will disregard this PEC byte and act on the transaction as if the PEC byte was not transmitted.

For reads with PEC while PEC_REQUIRED is enabled, if the host does not send a stop bit and continues to clock out data, the slave will return an extra byte that is the PEC value as calculated by the LTC3880. The host may compare its computed PEC value to the received PEC value. If the values match, the data is valid. If the PEC values are mismatched, then the data is not valid. The slave does not know or care if the received PEC is invalid or different than what the master computed. Because of the SMBus/PMBus read transaction convention the master's NACK of the data indicates that it is finished reading data, not that the PEC is invalid. Even if the PEC is valid and matches, the host must still NACK the last byte of the transaction.

Linear Numerical Formatting [L11 & L16]

The term Linear Data Format as defined in the PMBus Specification Part II Section 7.1 is one of two numerical data formats under the name Linear Numerical Formatting.

We have named the format as described in Section 7.1 of the PMBus Specification Part II as the LinearFloat5_11 (L11) format. This format encapsulates a floating point number as a 5-bit signed exponent (power of two) and an 11-bit signed mantissa in a single 16-bit word. This is a very useful format that provides reasonably high precision and high dynamic range. In other words, it is capable of representing both very small and very large numbers, both negative and positive. This eliminates the need for special lookup tables or register dependent numeric transforms in your code to convert internal codes into meaningful units like volts, amps, and others.

We have named another linear format as the LinearFloat16 (L16). This format has a 16-bit unsigned mantissa that is multiplied by a separate exponent to determine the value. This exponent is stored in the five least significant bits of the VOUT_MODE register. As examples, the LTC2978 uses a read only 2^{-13} exponent value while the LTC3880 uses a read only 2^{-12} exponent value.

All sign-formats are standard two's compliment. Generally, output voltage related registers (such as READ_VOUT and VOUT_COMMAND) use the L16 format whereas non-output-voltage related registers generally use the L11 format.

Code Examples

The following code examples illustrate the simplicity of these numeric formats. These examples are written for clarity, and not necessarily for performance. Your specific implementation may differ depending on your platform and system requirements.

The examples are written in C for a PIC32 based microcontroller compiled with the C32 compiler.

Example Code License

While we have made every effort to ensure that this example code operates in the manner described, we do not guarantee operation to be error free. This code is provided for purposes of understanding only, not as a working implementation for any physical system.

Upgrades, modifications, or repairs to this example code will be strictly at the discretion of LTC. We do not guarantee that you will be able to use this software successfully in your system.

The software and related documentation are provided AS IS and without warranty of any kind and Linear Technology Corporation expressly disclaims all other warranties, express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Under no circumstances will LTC be liable for damages, either direct or consequential, arising from the use of this code or from the inability to use this code, even if we have been informed in advance of the possibility of such damages.

Code Example for L11 to Float

```c
/*
 * Convert a LinearFloat5_11 formatted word
 * into a floating point value
 */
float L11_to_float(uint16 input_val)
{
  // extract exponent as MS 5 bits
  int8 exponent = input_val >> 11;

  // extract mantissa as LS 11 bits
  int16 mantissa = input_val & 0x7ff;

  // sign extend exponent from 5 to 8 bits
  if( exponent > 0x0F ) exponent |= 0xE0;

  // sign extend mantissa from 11 to 16 bits
  if( mantissa > 0x03FF ) mantissa |= 0xF800;

  // compute value as mantissa * 2^(exponent)
  return mantissa * pow(2,exponent);
}
```

Code Example for Float to L11

```c
/*
 * Convert a floating point value into a
 * LinearFloat5_11 formatted word
 */
uint16 float_to_L11(float input_val)
{
  // set exponent to -16
  int exponent = -16;

  // extract mantissa from input value
  int mantissa =
   (int)(input_val / pow(2.0, exponent));

  // Search for an exponent that produces
  // a valid 11-bit mantissa
  do
  {
   if((mantissa >= -1024) &&
     (mantissa <= +1023))
   {
     break; // stop if mantissa valid
   }
   exponent++;
   mantissa =
   (int)(input_double / pow(2.0, exponent));
  } while (exponent < +15);


  // Format the exponent of the L11
  uint16 uExponent = exponent << 11;

  // Format the mantissa of the L11
  uint16 uMantissa = mantissa & 0x07FF;

  // Compute value as exponent | mantissa
  return uExponent | uMantissa;
}
```

Code Example for L16 to Float

```
/*
 * Convert a LinearFloat16 formatted word
 * into a floating point value
 */
float L16_to_float(uint8 exp, uint16 input_val)
{
  int8 exponent = exp;
  int16 mantissa = input_val;

  // sign extend exponent
  if( exponent > 0x0F ) exponent |= 0xE0;

  // sign extend mantissa
 if( mantissa > 0x03FF ) mantissa |= 0xF800;

 // compute value as mantissa * 2^(exponent)
  return mantissa * pow(2,exponent);
}
```

Code Example for Float to L16

```
/*
 * Convert a floating point value into a
 * LinearFloat16 formatted word
 */
uint16 float_to_L16(float input_val)
{
  // The length of the L16 value
  // Read the VOUT_MODE register of your
  // particular device for the value to use
  // LTC3880 = -12
  // LTC2978 = -13
  int L16_Length = -12;

  // set exponent to 2^L16_Length
  float exponent = pow(2.0, L16_Length);

  // convert value to uint16 and return
  return (uint16)(input_val / exponent);
}
```

ANALOG CONSIDERATIONS

When communicating with the LTC3880 at 100kHz or greater (regardless of clock stretching being enabled or not), the bus pull-up resistors must be sized small enough to allow fast pull-up times to achieve the required PMBus setup and hold times.

For a detailed explanation, please reference Table 1 SMBus AC Specification from the System Management Bus (SMBus) Specification Version 2.0 and Table 2 Timing Parameters for Higher Speed Operation from the PMBus Power System Mgt Protocol Specification – Part 1 – Revision 1.1.

Figure 7 shows a digital I^2C representation of the beginning of a PMBus transaction. T_{RISE} is the time it takes for SCL or SDA to get from a low state to a high state on the bus, and we recommend a maximum time constant of 100ns RC up to assure a 300ns rise time.

Tau RC $\Rightarrow$ $1\tau = 63.2\%$
$2\tau = 86.5\%$
$3\tau = 95.0\%$
Thus $1T = 100$ns so that a rise time of 95% is 300ns.

The faster the better, but there are practical limitations on achieving these speeds as will be discussed below. Once the clock is high, there is a minimum time (600ns) it must be high before allowing it to go low again. Meanwhile SDA must be stable for a minimum of T_{SETUP} (100ns) before SCL begins pulling low. T_{FALL} is the time it takes to pull the clock low and a maximum time of 100ns is recom-

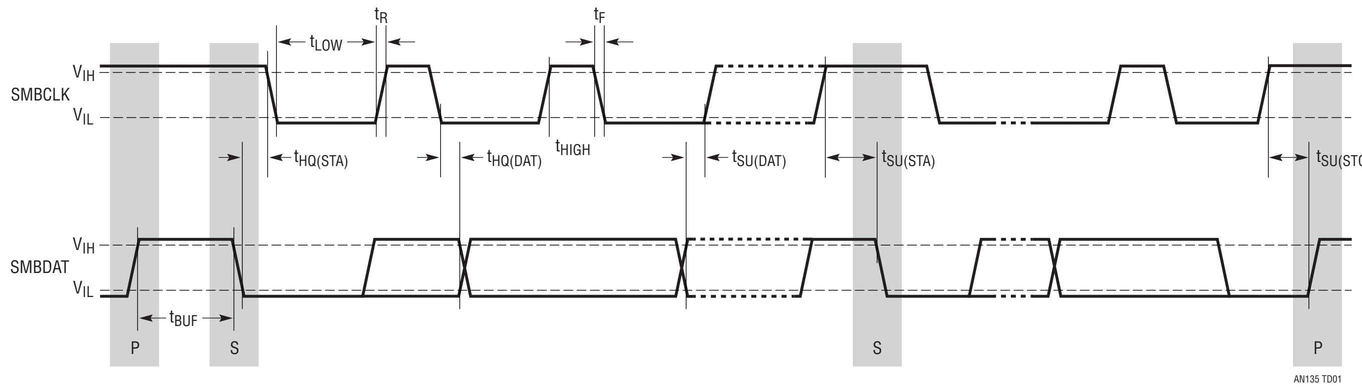


Figure 7. SMBus Timing Measurements

mended. Once SCL is low, it must remain pulled low for a minimum of 1.3µs before being released. Likewise SDA must remain stable for a minimum of T_{HOLD} (recommendation of 300ns) after SCL reaches a low state to ensure data transmission.

One factor that must be addressed when dealing with PMBus and I^2C in general is that while the bus is actively pulled down, it is released passively. This means to get SDA or SCL from a low state to a high state, the bus must charge up all capacitance on the bus through a small valued resistor with a current of no more than 3mA.

Figure 8 shows a square wave driving a FET gate to toggle the I^2C line. The I^2C line is pulled up to 3.3V via a 1k resistor, the minimum allowed. There is a 100pF capacitor in parallel with the pull-up resistor to simulate the cumulative parasitic capacitance caused by the active pull-downs themselves for **all** devices on the bus as well as all the PCB traces between devices. This combination gives us a time constant of 100ns which is the maximum recommended

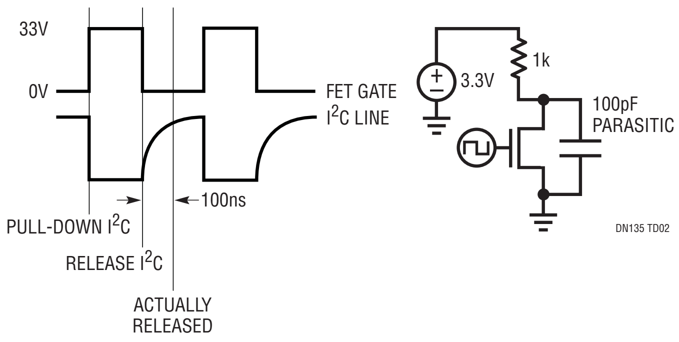


Figure 8.

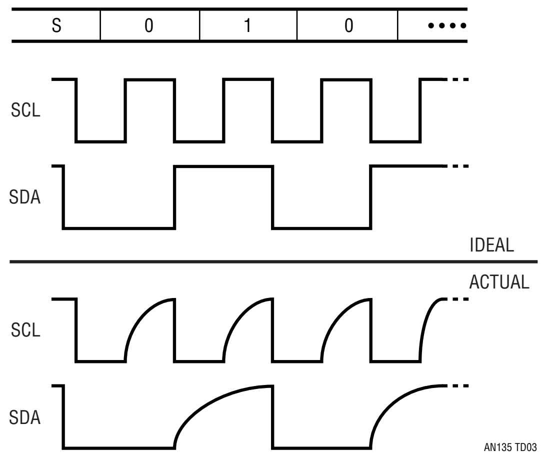


Figure 9.

rise time. From this waveform you can surmise that your bus when routed together must not have more than 100pF of capacitance or else the rise times will creep above 100ns possibly causing timing problems and bus errors.

Also take note that the larger the number of devices on the I^2C bus (any device, not just LTC3880's) there are, the higher the capacitance present on the SCL and SDA lines. The designer must account for this to ensure the rise times, fall times, setup times, and hold times are met.

Figure 10 shows a problem scenario that can result from too much capacitance on the I^2C lines. The waveforms are not square because of the passive release of the SCL and SDA lines. The start bit and the first two data bits appear to be OK because the fall time, setup time and hold times are met. The SCL line meets the minimum clock hold high and low times as well as marginally meeting the minimum rise time specifications. However, for some reason, (such as suboptimal PCB trace routing) SDA has slightly more parasitic capacitance. The rise times for SDA are longer than specified. So when clocking normally and transitioning from a logic-0 to a logic-1, the setup and hold times are violated. When the host clocks SCL by pulling it low, the data should have already been stable for T_{SETUP} and remain so for at least T_{HOLD} afterwards. During the T_{HOLD} region is when the line is sampled for the data value. Because the data line is still rising, the final value that will be read on the bus is indeterminate. As one approaches the timing limits, any little variation on the bus can potentially cause erroneous transactions or complete bus failure. Care must be taken to ensure rise times are met along with all other PMBus timing specifications. A 100ns margin for the rise and fall times should be observed.

The use of bus accelerators such as the LTC4311 can help by actively pulling up the bus when it detects a rising edge, but the resistors must still be sized appropriately to

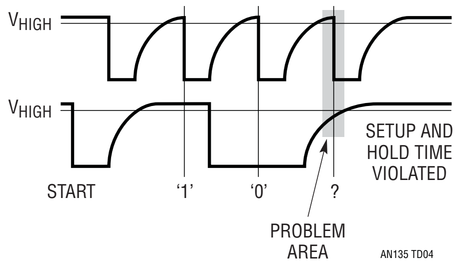


Figure 10.

account for bus variations and the number of devices on the bus. If the bus takes too long to reach the accelerator's threshold, then the timing of the actively pulled edge may be too late and the transaction will fail.

Communicating Robustly with the LTC3880

Implementing a robust communication layer between the LTC3880 and your host controller is a straightforward task with some caveats that must be followed to ensure proper behavior. The LTC3880 also supports an optional, speed-limited communication mode for non-PMBus-compliant or less sophisticated host devices (bus masters) that may be used trading off bus speed for firmware simplicity.

ROBUST HIGH-SPEED COMMUNICATION

Command Order

The LTC3880 has a one deep buffer to hold the last data written for each supported command prior to processing as shown in Figure 11 Write Command Data Processing. When the LTC3880 receives a new command from the bus, it copies the data into the write command data buffer and indicates to the internal processor that this command data needs to be fetched and converted to its internal format so that the command can be executed.

Two distinct parallel blocks manage command buffering and command processing (fetch, convert, and execute) to ensure the last data written to any command is never lost. Command data buffering handles incoming PMBus writes by storing the command data to the write command data buffer and marking these commands for future processing. The internal processor runs in parallel and handles the sometimes slower task of fetching, converting and executing commands marked for processing. Some computationally intensive commands (e.g.: timing parameters, temperatures, voltages and currents) have internal processor execution times that may be long relative to PMBus timing. If the part is busy processing a command, and new command(s) arrive, execution may be delayed or processed in a different order than received. The part indicates when internal calculations are in process via bit 5 of MFR_COMMON (calculations not pending). When the part is busy calculating, bit 5 is cleared. When this bit is set, the part is ready for another command. An example polling loop is provided in Code Example, Wait Until Not Busy, that ensures commands are processed in order and simplifies error handling routines.

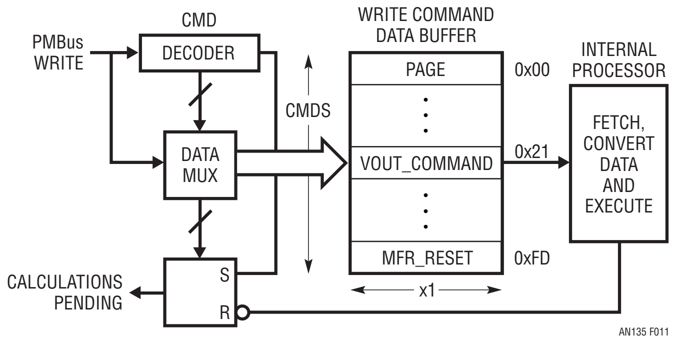


Figure 11. Write Command Data Processing

Clock Stretching (a.k.a. Clock Synchronization)

If communicating above 100kHz on a high speed PMBus, we highly recommend using clock stretching to simplify firmware logic. Clock stretching will only occur if enabled and the bus communication speed exceeds 100kHz.

When a slave device requires more time to process a command than the current host clock speed allows, it may slow down the communication temporarily by stretching the SCL line. To stretch the SCL line simply means extending the low state of the SCL line by the slave actively pulling down the clock line regardless of whether or not the master(s) have released it already. While the SCL line is low, any device on the bus may additionally hold it low preventing it from rising again until the device is ready.

This is illustrated in Figure 12. After the first byte transfer is complete, the slave needs time to process this data before returning the requested value. The slave then holds the SCL line low while the interrupts are being serviced. Once the slave is ready to continue, it releases its hold on SCL and communications continue as normal.

Note that LT PMBus devices will only clock stretch if the clock stretching feature is enabled and the bus speed exceeds 100kHz. Clock stretching is enabled by setting bit 1 of MFR_CONFIG_ALL. In the code example below, MFR_CONFIG_ALL is referenced as MFR_CONFIG_ALL_LTC3880_0xD1 but they are equivalent.

When the master releases the SCL line, it must then read back the physical state of the wire and wait until it reaches a high level again before beginning its next clock sequence. Many devices have hardware level support for clock stretching that can be enabled by simply setting a bit. If you are planning to use a software bit-banged I^{2}C implementation it is not difficult to add in such a check after each transaction subroutine before continuing on.

Clock stretching can be enabled by asserting bit 1 of MFR_CONFIG_ALL_LTC3880. If clock stretching is not enabled and you want to enable it, you must take precautions to ensure the bit is set correctly. In the Code Example, Enabling Clock Stretching we take several steps to ensure the bus is operating at 100kHz before writing enabling clock stretching and then increase the speed to 400kHz when done. To change the speed of the peripheral bus we first wait until the bus is idle. Then we change the bus speed. Next we do a read-modify-write on the MFR_CONFIG_ALL_LTC3880 register (0xD1) to enable clock stretching. We then again wait for the bus to be idle and then increase the speed up to full 400kHz speed.

A Simple Alternative to Clock Stretching with Bus Speeds at or Below 100kHz

If your bus master does not support clock stretching we **strongly** advocate that you use communication bus speeds between 10kHz and 100kHz. This will allow for a simpler firmware implementation, and you may disregard the next sub-section regarding clock stretching.

Operating above 100kHz without clock stretching enabled will require more complex firmware. Your firmware must gracefully handle NACK's and empty read-backs and may have to deal with unwanted ALERT events when the device

Code Example: Enabling Clock Stretching

```
/*
* Enable clock stretching for robust
* communication
*/
void enable_clock_stretching(uint8 dev_addr)
{
  // Wait until I2C bus is idle
  while(I2C2CON & 0x001F > 0 ||
     I2C2STATbits.TRSTAT != 0)
  {
    Nop(); // Twiddle
  }

  I2C2BRG = 0xE6; // Set I2C to 100KHz

 // Read old value of MFR_CONFIG_ALL_LTC3880
  uint8 old_MFR_CONFIG_ALL_LTC3880_0xD1 =
    smbus_read_byte(dev_addr, 0xD1);

  // Logical OR in the clock stretching bit
  uint8 new_MFR_CONFIG_ALL_LTC3880_0xD1 =
   (old_MFR_CONFIG_ALL_LTC3880_0xD1 | 0x02);

 // Write new value of MFR_CONFIG_ALL_LTC3880
  smbus_write_byte(dev_addr, 0xD1,
        new_MFR_CONFIG_ALL_LTC3880_0xD1);

  // Wait until I2C bus is idle
  while(I2C2CON & 0x001F > 0 ||
     I2C2STATbits.TRSTAT != 0)
  {
    Nop(); // Twiddle
  }

  I2C2BRG = 0x46; // Set I2C to 400KHz
}
```

is busy. Robust error recovery code is required to handle these conditions.

When the part receives a new command while it is busy, it will communicate this fact using standard PMBus protocol. Depending on part configuration it may either

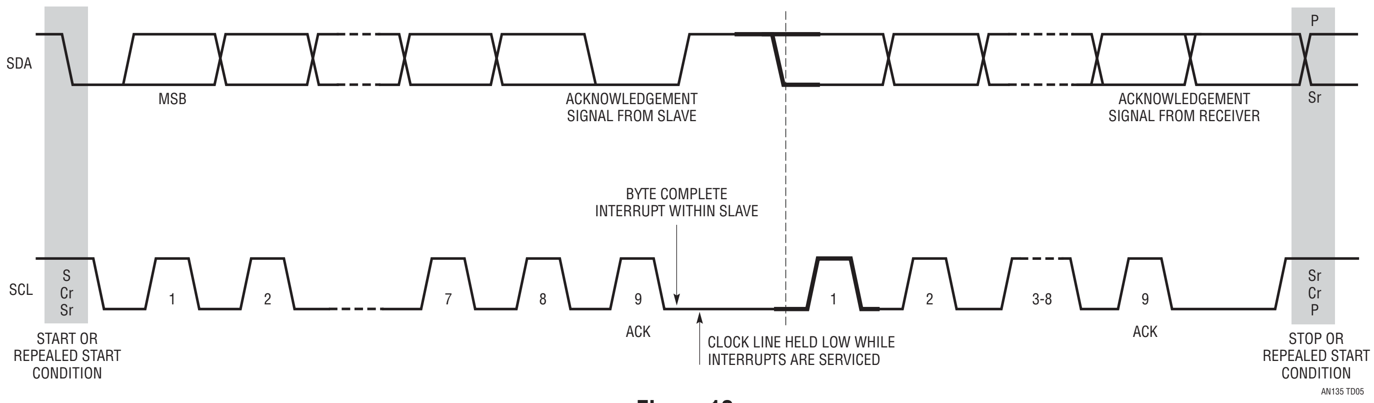

SDA
MSB
ACKNOWLEDGEMENT SIGNAL FROM SLAVE
ACKNOWLEDGEMENT SIGNAL FROM RECEIVER
P
Sr
BYTE COMPLETE INTERRUPT WITHIN SLAVE
SCL
S Cr Sr
1
2
7
8
9
ACK
1
2
3-8
9
ACK
Sr Cr P
START OR REPEALED START CONDITION
CLOCK LINE HELD LOW WHILE INTERRUPTS ARE SERVICED
STOP OR REPEALED START CONDITION
AN135 TD05

Figure 12.

Code Example: Wait Until Not Busy

```
/*
 * Wait until device not busy.
 * Call this before sending each command.
 */
void wait_for_device_not_busy(uint8 dev_addr)
{
  uint8 status;

  // Wait until the device is ready
  // to receive new commands...
  while(1)
  {
    // Read status mask
    if(SMBus_read_byte(dev_addr, 0xEF,
             (uint8*)&status))
    {
      // Check for busy flags of MFR_COMMON
      // bits 4, 5, and 6
      if((status & 0x70) == 0x70)
      {
        // Device is ready
        return;
      }
    }
  } // end of while(1) polling loop
}
```

NACK a command or return all ones (0xFF) for reads. It may also generate a BUSY fault and ALERTB notification, or stretch the SCL clock low. For more information refer to PMBus Specification V1.1, Part II, Section 10.8.7 and SMBus V2.0 Section 4.3.3.

PMBus busy protocols are well accepted standards, but can make writing system level software somewhat complex. The part provides three hand shaking status bits which reduce complexity enabling robust system level communication: MFR_COMMON bits 4, 5, and 6.

The three hand shaking status bits are in the MFR_COMMON register. When the part is busy executing an internal operation, it will clear bit 6 of MFR_COMMON (chip not busy). When the part is busy specifically because it is in a transitional VOUT state (margining hi/lo, power off/on, moving to a new output voltage set point, etc.) it will clear bit 4 of MFR_COMMON (output not in transition). When internal calculations are in process, the part will clear bit 5 of MFR_COMMON (calculations not pending). These three status bits can be polled with a PMBus read byte of the MFR_COMMON register until all three bits are set. A command immediately following the status bits being set will be accepted without NACKing or generating a BUSY fault/ALERT notification. The part can NACK commands for other reasons, however, as required by the PMBus spec (for instance, an invalid command or data).

NACK's and Empty Reads

A slave receiver is allowed to not acknowledge (NACK) its slave address during I^{2}C transactions. For example if it's unable to receive because it's performing some real time task, it may NACK. PMBus/SMBus requires devices to always acknowledge (ACK) their own address, as a mechanism to detect a removable device's presence on the bus. I^{2}C specifies that a slave device, although it may acknowledge its own address, may decide at some future time, that it cannot receive any more data bytes. I^{2}C specifies that the device may indicate this by generating the NACK on the first byte to follow.

PMBus/SMBus also uses the NACK mechanism to indicate the reception of an invalid command or data. Since such a condition may occur on the last byte of the transfer, it is required that PMBus/SMBus devices have the ability to generate the NACK after the transfer of each byte and before the completion of the transaction.

This is important because PMBus/SMBus does not provide any other signalling method for resending. This difference in the use of the NACK signalling has implications on the specific implementation of the communication port, especially in devices that handle critical system data such as the PMBus/SMBus host.

To make dealing with these issues simpler for the firmware designer, the LTC3880 has status registers that can be read at any time under the specified conditions without the possibility of missing acknowledges indicating its current internal status. By knowing what these bits represent, you can then wait until the device is ready to accept another command before issuing it. This prevents sending transactions that will be unsuccessful by delaying their transmission until the device is ready to accept them.

However, there are instances where the slave device cannot indicate that it is busy with PMBus/SMBus. If the host requests data from some register, the device ACKs the command, and then the device becomes busy because of an internal computation or interrupt which makes it unable to comply with the request, a NACK is not possible.

The slave is responsible for returning data but the host is responsible for the remaining ACKs in the read transaction.

Because the device must always ACK its address there is no chance for a slave device to signal the host that it is not ready to send back the requested data. In this case the only things the device can do is hold the clock line (clock stretching) or return an empty read which is the SDA line high or all 1's (0xFF). Clock stretching must be enabled for the device to use clock stretching, and in the case where it is not enabled an empty read is the only solution. These empty reads will not occur at or below 100kHz, or if clock stretching is enabled.

DEVICE ADDRESSING with the LTC3880

The LTC3880 offers five different types of addressing over the PMBus interface, specifically: 1) global, 2) device, 3) channel, 4) rail addressing and 5) alert response address (ARA).

Global addressing provides a means of the PMBus master to address all LTC3880 devices on the bus. The LTC3880 global address is fixed 0x5A (7-bit) or 0xB4 (8-bit) and cannot be disabled. Commands sent to the global address act the same as if PAGE is set to a value of 0xFF. Commands sent are written to both channels simultaneously.

Global command 0x5B (7-bit) or 0xB6 (8-bit) is paged and allows channel specific command of all LTC3880 devices on the bus.

Device addressing provides the standard means of the PMBus master communicating with a single instance of an LTC3880. The value of the device address is set by a combination of the ASEL configuration pin and the MFR_ADDRESS command. When this addressing means is used, the PAGE command determines the channel being acted upon. Device addressing can be disabled by writing a value of 0x80 to the MFR_ADDRESS.

Channel addressing provides a means of the PMBus master addressing a single channel of the LTC3880 without using the PAGE command. The value assigned to the paged MFR_CHANNEL_ADDRESS determines the specific channel the user wishes to act upon, which is useful for multi master systems.

Example: If MFR_CHANNEL_ADDRESS for page 0 is set to 0x57 and the MFR_CHANNEL_ADDRESS for page 1 is set to 0x54, the user can address channel 0 of the device by performing PMBus device commands using address 0x57 (7 bit). The user can address channel 1 of the device by performing PMBus device commands using address 0x54 (7 bit). This eliminates the user from first assigning the PAGE command and then the command to be acted upon.

Rail addressing provides a means of the PMBus master addressing a set of channels connected to the same output rail, simultaneously. This is similar to global addressing, however, the PMBus address can be dynamically assigned by using the MFR_RAIL_ADDRESS command. The MFR_RAIL_ADDRESS is paged, so channels can be independently assigned to a specific rail. It is recommended that rail addressing should be limited to command write operations.

In addition to the standard SMBus/PMBus ARA address and reserved bus addresses, the LTC3880 uses the address of 0x7C for NVM error reporting. If the internal NVM of the LTC3880 becomes invalid and cannot be automatically repaired/rebuilt, the device will disable itself except for bus communication. Only in this circumstance the LTC3880 will ACK the address 0x7C. The device will also set bit 3 of MFR_COMMON to indicate that the NVM has not initialized properly. Therefore it is recommended to leave address 0x7C open if at all possible.

All five means of PMBus addressing require the user to employ disciplined planning to avoid addressing conflicts.

The color coded table, (Table 1), quickly shows a mapping of restricted addresses.

Table 1

0x00	**0x01**	**0x02**	**0x03**	**0x04**	**0x05**	**0x06**	**0x07**
0x08	**0x09**	**0x0A**	**0x0B**	**0x0C**	0x0D	0x0E	0x0F
0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F
0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
0x28	0x29	0x2A	0x2B	0x2C	0x2D	0x2E	0x2F
0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
0x38	0x39	0x3A	0x3B	0x3C	0x3D	0x3E	0x3F
0x40	0x41	0x42	0x43	0x44	0x45	0x46	0x47
0x48	0x49	0x4A	0x4B	0x4C	0x4D	0x4E	0x4F
0x50	0x51	0x52	0x53	0x54	0x55	0x56	0x57
0x58	0x59	▨ 0x5A	▨ 0x5B	0x5C	0x5D	0x5E	0x5F
0x60	0x61	0x62	0x63	0x64	0x65	0x66	0x67
0x68	0x69	0x6A	0x6B	0x6C	0x6D	0x6E	0x6F
0x70	0x71	0x72	0x73	0x74	0x75	0x76	0x77
0x78	0x79	0x7A	0x7B	▨ 0x7C	0x7D	0x7E	0x7F

Address	PMBus Reserved Addresses
Address	Available Addresses
▨ Address	LTC3880 Reserved Addresses