SMBus made simple

Abstract

SMBus is the most common form of communication for Texas Instruments Advanced Fuel gauges. Many customers want to design their own SMBus engine to communicate to TI Advanced Fuel gauges. This is perfectly acceptable but sometimes can cause confusion and frustration. Investigating SMBus errors or transaction failures can appear to be a very difficult or daunting task. The purpose of this document is to break down the complexity and make "getting to know" SMBus easy. It assumes some knowledge of I2C

Glossary

Ack: A bit in the SMBus communication packet used to signify an "Acknowledgement" of the previous byte of data.

Nack: A bit in the SMBus communication packet used to signify an "No-Acknowledgement" of the previous byte of data.. Usually signifies an SMBus error or it comes at the end of the last byte of a READ communication packet.

Packet: A complete SMBus transaction, either read or write, from the start bit to stop bit.

Word: For this document, an SMBus Word signifies 2 bytes of data (0xFFFF).translated from 0 – 65535 unsigned or -32768 to 32767 signed

MSB: Most significant byte in a word (2 bytes) of data.

LSB: Least significant byte in a word (2 bytes) of data

FG: Fuel gauge

Repeated start: A second start bit in the communication packet used in a SMBus read to transition from writing the SMBus command code to reading data from that command code.

Master: The master is the device on the SMBus that is controlling the current communication packet. The "Master" controls the clock line on the bus for any given packet.

Slave: This is the opposite of the "Master". The slave does not control the clock line except for clock stretching used to slow the transaction down to allow the slave more time if needed.

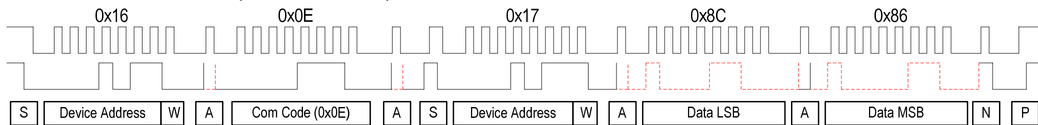
Broadcast: Broadcast is a term to indicate when the TI Fuel gauge becomes a master on the bus and "broadcasts" information to the host as a "Master". This is also sometimes called "Master Mode Messaging".

Getting to Know Smbus

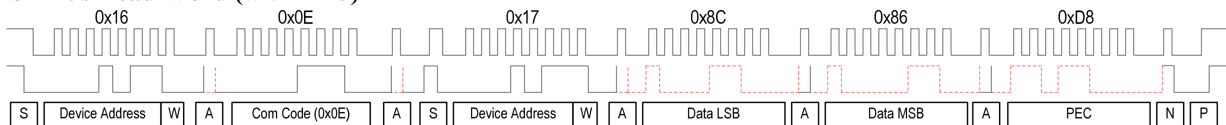
Below are shown some simple examples of generic SMBus transactions. These are read/write words with and without PEC. While your scope traces will not look exactly like these examples, it is easier to look at these theoretical examples and understand what is going on first rather than looking at actual scope traces. We will look at those later on with actual examples. Also it is important to note that there is much more detailed information to gather from these pictures that we will not describe in this document. The reason for not covering this information is so we do not get bogged down in extreme detail that will only confuse and most likely is not your problem. We only intend to touch the basics since for most troubleshooting issues the basics are all that is needed to solve whatever problem you are having.

First we will look at the entire packets for write and read. We will only be looking at "word" communications here since these are the most common and will be relevant for most problem troubleshooting.

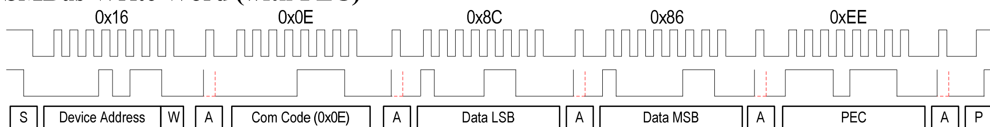
SMBus Read Word (without PEC)



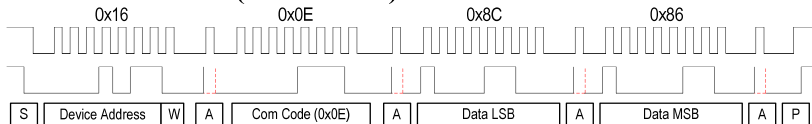
SMBus Read Word (with PEC)



SMBus Write Word (with PEC)



SMBus Write Word (without PEC)



Legend: - - - Slave control
—— Host Control

Here are the components that make up the packet and some issues to look for.

Start bit: Each packet of data must start with a Start bit denoted with an "S". The clock must wait at least 4µs after the data line goes low before it should go low.

Device address 1: The device address is sent by the host telling all the slaves on the bus which one should "acknowledge" this particular communication packet.

- SMBus can have multiple slaves so all other slaves that do not have this address should ignore the packet. Smart batteries should have device address 0x16 so this packet should be "acknowledged" by any fuel gauge.
- There cannot be more than one device on the bus with the same device address.
- The last bit of the device address is the read/write bit. A "0" for this bit denotes a write and a "1" for this bit denotes a read. The read/write bit in the first device address for a read is a "0" because we are writing a command code to the slave first. In a write packet there is only one device address because the direction (read/write) does not change.

Acknowledge: Denoted by an "A". The slave must acknowledge that the device address was received.

Command Code: This is the command or slave data address that will be written to in a write packet or read from in a read packet.

Repeated Start (Read): Denoted by an "S". A second start bit embedded within the packet used to shift the bus to a read.

Device address 2 (Read): The second device address in a read packet is a legacy component. Since a read is really 2 packets combined with a repeated start, it really shouldn't be required since we have already deemed which slave is responsible for this packet. SMBus Specification still requires this as part of the spec so it is mandatory for communicating to all devices including our fuel gauges. There is important information in this byte however. That is the read/write bit which is set to a 1. This tells the slave that this packet is a read so it will be prepared to clock out data.

Data LSB: The first byte of data is the least significant byte of the data word. The reason why SMBus sends the LSB first is because SMBus sends data in little-endian format. This just means that data is sent in increasing numeric significance. The reason for this is because most modern computers store data in memory in this order.

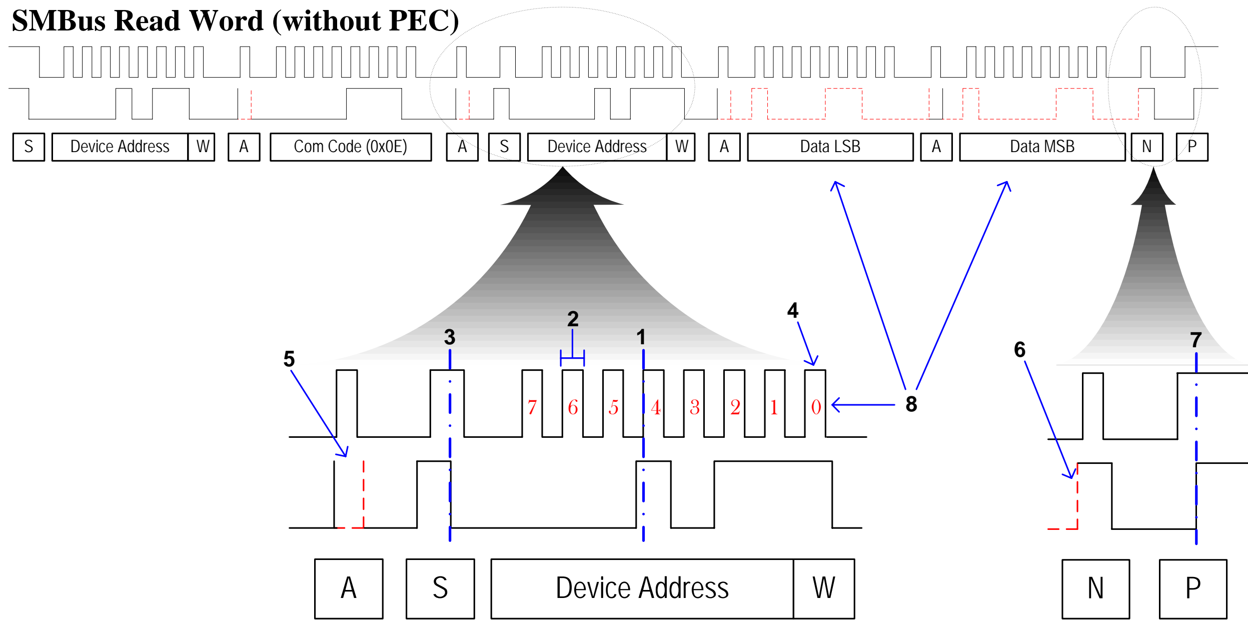
Data MSB: This is the second byte of data for the word sent. This is the most significant byte. Again it is sent this way to conform to the little-endian format.

PEC: PEC byte is a checksum of the entire packet used to protect against data corruption.

Stop bit: This is the end of the packet. It tells the slave device that the bus is done so that the slave can get ready for more communications. It is a very important part of the packet. Many people get into trouble by leaving this off if they get all the data. While our Fuel gauge will timeout and reset eventually without this, it is important to keep all devices on the bus in a known state at the end of each packet sent. Even if the host has to stop the communication in the middle of the packet for some reason it should always send the stop bit to reset everyone on the bus.

Closer inspection

Now to dissect the communication with a little more detail. The below picture zooms into one byte of a data packet and the NACK/Stop bit. This diagram gives examples of most of the important bits of a total packet.



Each byte is 8 bits long. There are several things of interest that can be derived by looking closely at this diagram:

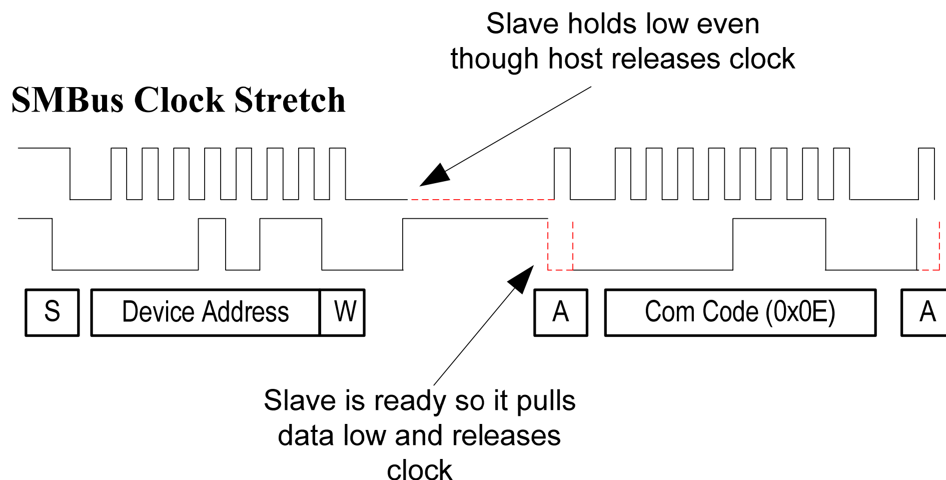
1. **Data Processing:** Each bit of data is processed by the smbus engine on the rising clock edge. In other words, this is where the data is shifted into the engine. It may be helpful to note that the data should never change levels while clock is high during an smbus transaction except to create a start, restart, or stop bit

2. **Clock Timing:** The most common cause of difficulty with SMBus is host systems not following the SMBus High clock timeout spec. Do not let the clock be high at any time during a transaction for more than 50 microseconds. The SMBus engine will interpret this as a bus idle condition and reset. This one will “getcha” more than any other requirement in the SMBus spec.

3. **Repeated Start:** The repeated start bit is unique in that it shifts the focus of the current transaction from a write to a read. Prior to the repeated start is a write to a command code with the read/write cleared in the device address, and after the repeated start the bus shifts to a read of data with the read/write bit set.

4. **Read/Write bit:** This bit is appended to the end of the device address. The device address is usually thought of as being 8 bits long but it is actually 7 bits. So the device address in an 8 bit format is a 0x16 in a write and a 0x17 after the repeated start in an SMBus read packet.

5. **Acknowledge:** All bytes are followed by an acknowledge (ACK) except for the last byte of a read packet when the host is responsible for NACK-ing the last byte. The slave expects a NACK of this byte, even if it's a PEC byte (PEC is explained later in this document). Who is responsible for sending the ACK? The simple answer is whoever is receiving the byte prior to the ACK.
6. **No Acknowledge:** A no acknowledge will follow any byte that is not understood by the device receiving the previous data byte. The exception to this rule is the NACK required from the host after the last byte of data in a read packet (see Acknowledge). This indicates to the slave that the host has received all bytes that it expected.
7. **Start/Stop Bit:** The stop bit is the final bit in the packet. Once this bit is sent by the host then the slave will ignore anything on the bus until a start is detected and then will only acknowledge its own device. In fact, by SMBus specification, the fuel gauge must always acknowledge its device address.
8. **Bit Order vs. Byte Order:** Why do we point this out? Because it can be confusing since the orders are opposite. Each BYTE starts with the MSBit first and ends with the LSBit. But the word of data is sent with the LSByte first and the MSByte last. SMBus specification requires this. The bitwise order is normal, however, the bytewise order is little endian format as explained earlier.



Some final points to look at when explaining all the points of a total packet:

Clock Stretching: Clock stretching is a simple way for the slave to tell the host to "HOLD ON a second. I'm busy!" Anytime the clock is low in the packet, the slave has the right to grab the clock and hold it low as long as required except that it must not cause a packet timeout (25ms).

In other words the entire packet must not be longer than 25ms including the clock stretch. TI fuel gauges will usually only clock stretch before or after the ACK or NACK bits.

Broadcasting (Master mode Messages): Sometimes a slave can become the master of the bus. SMBus specification allows for this possibility. All of our advanced fuel gauges have this ability. Depending on the FG (Fuel gauge) there are different ways to enable or disable this feature. If enabled, then the FG will master the bus to send alarms and charging voltage and charging current every 10-20 seconds to the host (0x10) and charger (0x12) device addresses.

An SMBus Master can only start a packet if the SMBus has been idle for more than 50 microseconds. Once this requirement has been met, then the Master immediately "grabs" the bus by sending a start bit. All TI FG's function exactly like this and since they are hardware controlled SMBus Engines, they easily detect idle.

This type of behavior is very difficult to create for an SMBus engine implamented in firmware. It is easy to detect 50 microseconds of idle time, but then the port pins used to create the bus must be switched to outputs and then the start bit must be sent. During this time, there is a window where another Master may actually grab the bus. The firmware controlled bus then does not know that the bus is no longer busy. We then lose arbitration.

Since this is difficult, most Host systems just ignore this part of the SMBus specification and assume they are always the only master on the bus. It is for this reason that we always recommend that broadcasting messages be disabled for every application unless absolutely necessary.

PEC: We introduced PEC in the examples above. So what is it exactly?
PEC is a simple form of a checksum used for error checking. It is very important to use this in all communications to insure that what you send (or what you receive) is what was intended. PEC is really just an extra byte of data added to the end of the communication packet that is derived from a simple CRC-8 checksum. All TI Fuel gauges that support PEC also have the option to add PEC to the broadcast data if so desired. However, most customers never use the broadcasts. Broadcasts should be completely disabled if not used.

A common question is "Who sends (is responsible for) the PEC byte?" For this discussion it is assumed that the master is the host system (notebook PC or some other host) and the slave is the Fuel gauge. In read, the slave (Fuel gauge) is responsible for sending the PEC packet to the host. Then the host determines if the PEC is valid. In a write, the host is responsible for sending the PEC to the slave. Some slave devices may not be fast enough to do this, but a TI Fuel gauge will always NACK the PEC byte if there is an error in the packet. This can sometimes be confusing. Therefore here is a simple and easy way to remember this. The SMBus device who is responsible for sending the Data of the last byte of the packet is the one responsible for the PEC.

There is much more in the SMBus Specification about PEC relating to protection against devices that do not do PEC reliably, however, they do not apply to TI Fuel gauges. TI parts use a hardware PEC lookup so no software will ever interfere with the process. You will never get an ACK to a bad data packets PEC byte and you will always get a NACK to a bad data packet.

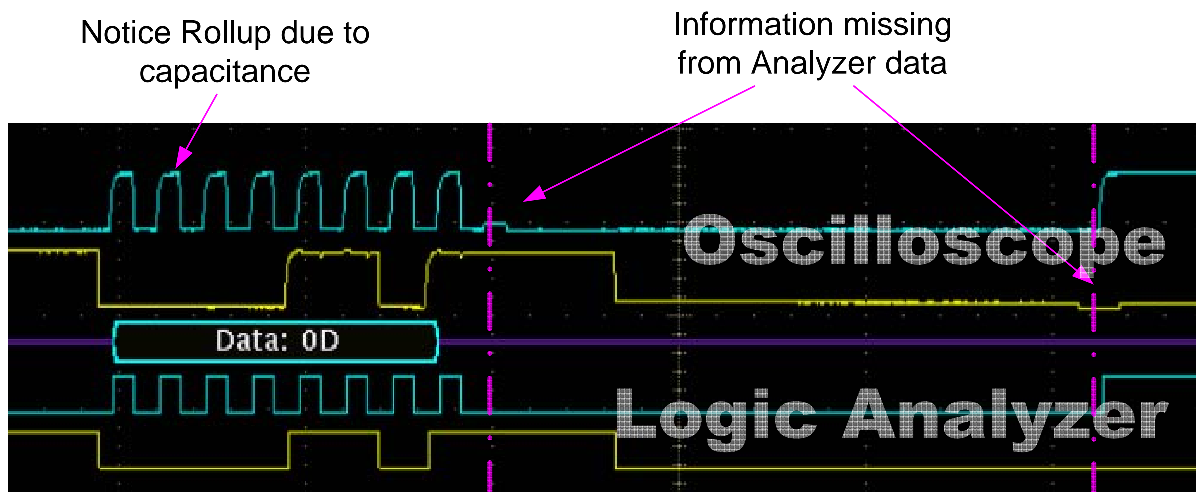
How to calculate:
PEC calculation does take some resources by the host system unless it has a hardware PEC engine. This document will not go into detail on how to do this. There are many resources on the internet for this, including explanations and even code examples. Just do a search for PEC or CRC-8 computation. There are even calculators online to check your code. An example of one is on the http://smbus.org website (http://smbus.org/faq/crc8Applet.htm).

There are 2 primary methods to calculate a PEC for a given data packet. Depending on your host CPU and memory, one of these methods should work for your application.

1. A lookup table method is very time efficient. However it takes a large amount of data memory to implement.
2. Direct calculation of the PEC is simple to understand and takes little program memory, however, it is an iterative process that takes CPU time.

Why cant I use a logic analyzer or "bus snooper" to see what's going on?
A logic analyzer or a "bus snooper" may appear to be the simplest tools to monitor the SMBus traces, however there are many reasons why an oscilloscope should be used.

1. A logic analyzer only shows a vague or high level piece of what is actually on the bus. It does not show rise times or noise or any other electrical aspects of the bus. It only shows transitions. We need to see all the information when troubleshooting. It is a fine tool to see what is happening on a bus that has no issues. But who needs that? We are working with critical communication failures so we must have more detail.



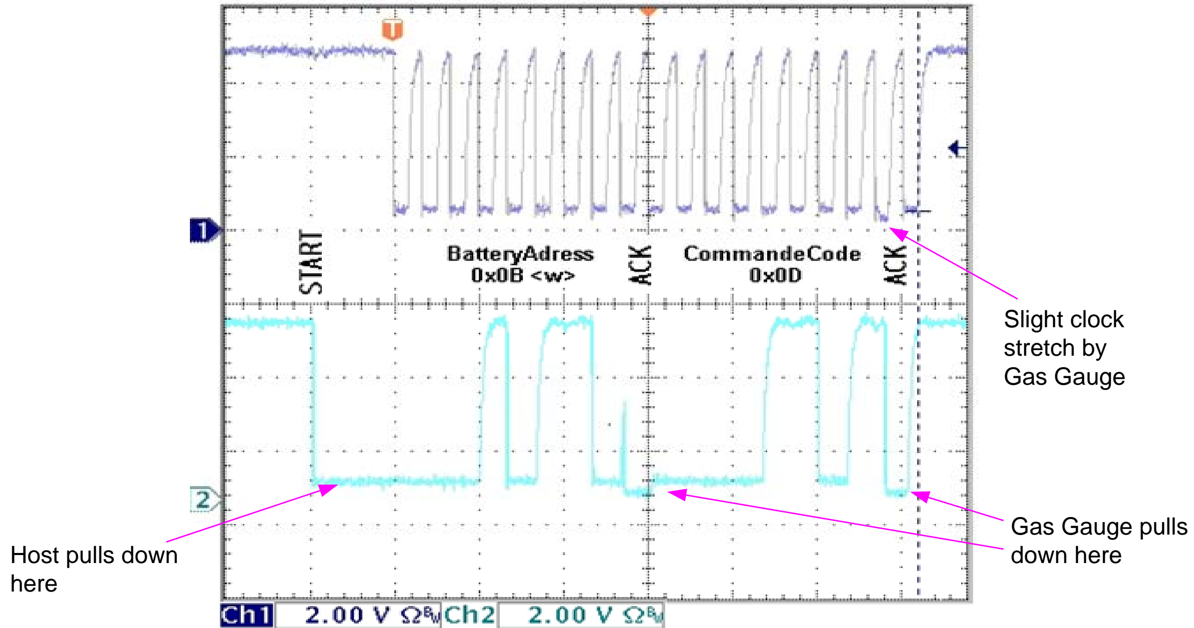
2. A bus snooper is a tool that logs communications and in that log will report ASCII text representing what is happening on the bus. It gives information at an even higher level than the logic analyzer. Here are a couple of lines out of a log file:

```
Msg 11    [S]#16 [A] #0E [A][S] #17 [A] #8C [A] #86 [A] #D8 [N][P]
Msg 12    [S]#16 [A] #0E [A][S] #17 [N]
```

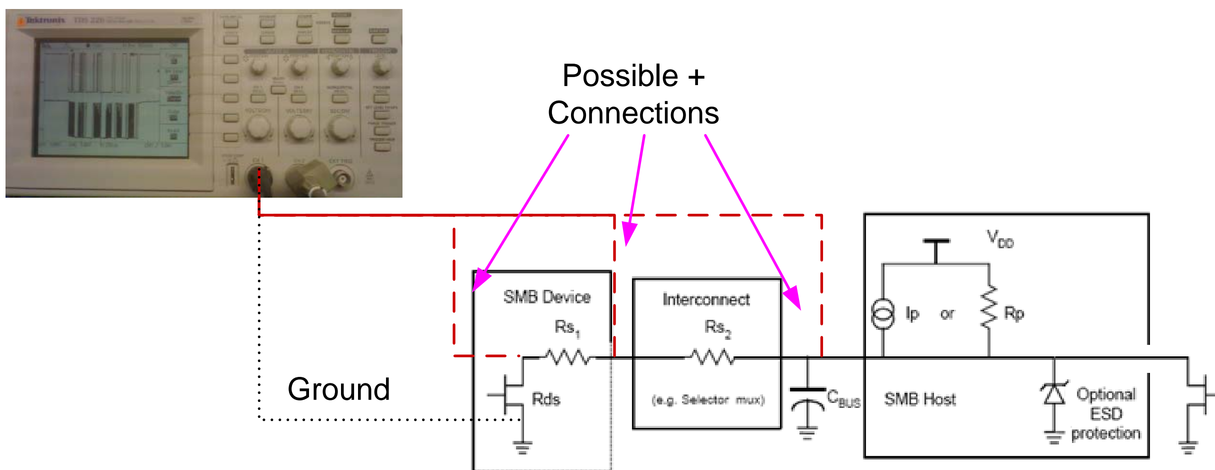
You can see there must be a problem with this communication right? But what is it? You just know that the Device NACK-ed its device address after the repeated start

Some examples of why an oscilloscope is so useful:
An actual oscilloscope is the only tool to give us the whole picture. Here are some examples of data that has been sent to me over the years that stress this point:

Example1: Who is in control around here?



Notice in this picture that you can actually tell who is pulling down on the bus at any given time. The "zero" threshold for the Fuel gauge is at a lower voltage than the "zero" threshold for the host. This is an extremely useful tool when debugging to tell who is in control of the bus. The reason this works is due to the current flowing through series protection resistors on the bus. The "zero" threshold is slightly different depending on which side of these series resistors the scope probe is connected with reference to ground. Below is a modified picture showing different connection points for a scope probe. Each point will have a different "zero" level voltage in reference to ground. The difference will also vary depending on the resistors used in the circuit. This original picture was drived from the SMBus Specification.

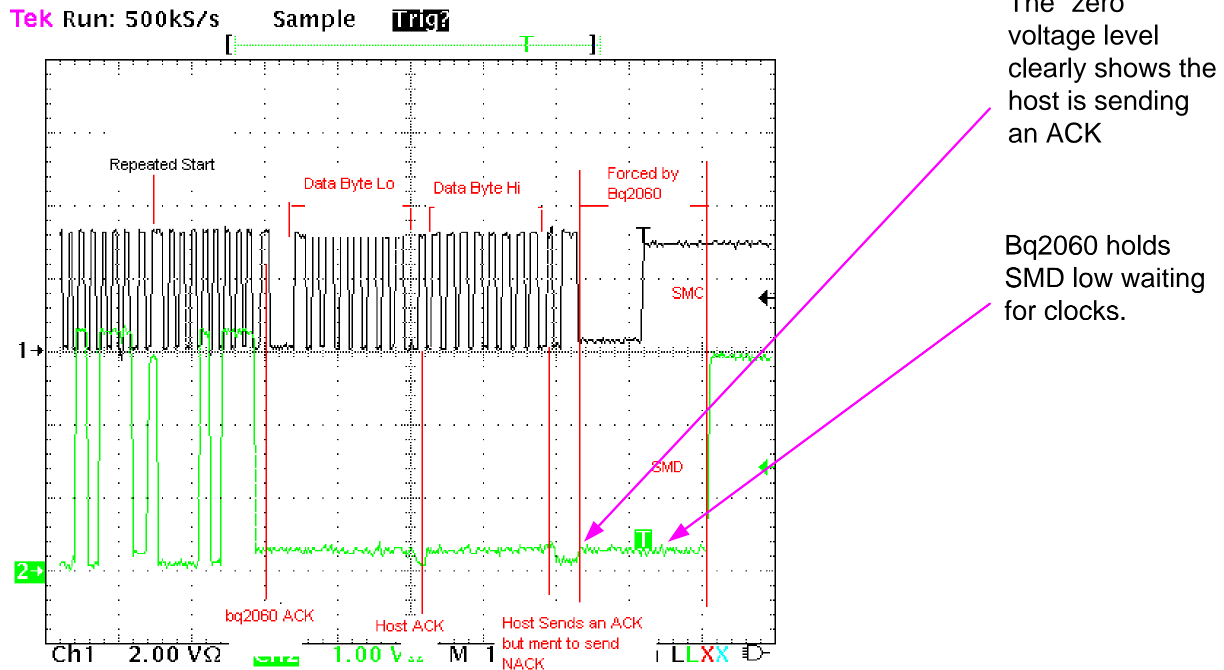


Example 2: Rising nice and slow!

Very large capacitance

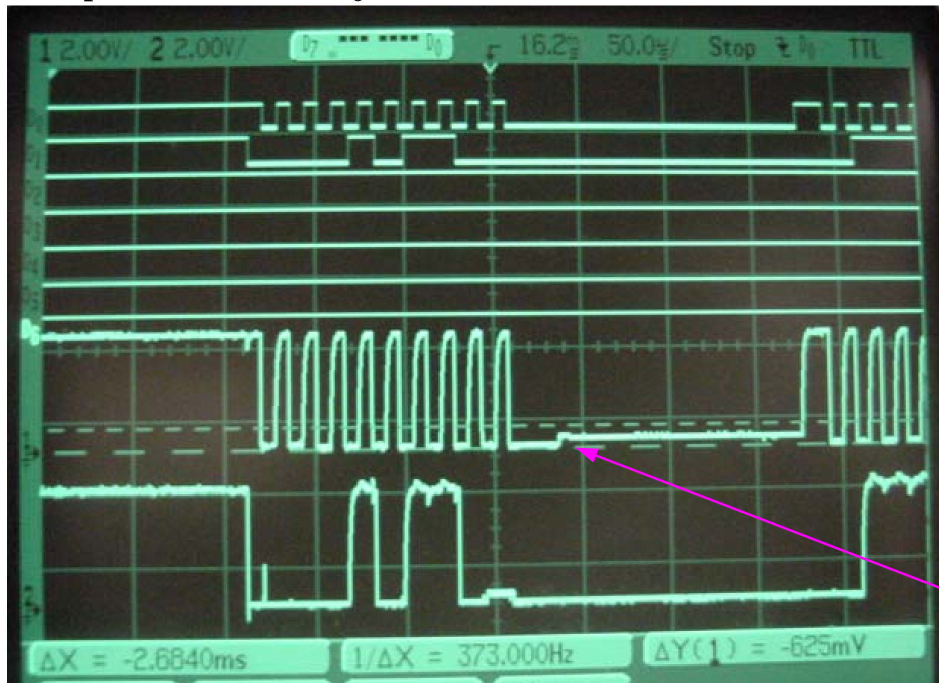
This customer complained about receiving wrong data. It was close, but sometimes there was just a bit every once in awhile that was wrong. This large capacitance on the line would sometimes cause a one to be read as a 0 because the data line was just on the edge of going up fast enough to be a "1" by the time the clock went to a "1". Later we found that the customer was using protection diodes on the bus that had a very high capacitance. Once we removed the high capacitance protection diodes and went with our more common protection circuitry as shown below, the problem went away.

Example 3: You want me to do WHAT?



This plot of a read word was sent from a customer that could not understand why the bq2060 was pulling the data line low as shown with the arrow. They first sent snooper data showing an ACK and then a STOP bit. It made no sense until this plot was sent to me. The problem was that the host did not intend to send a PEC byte but since it sent an ACK (host pulled data line low at the clock), the bq2060 interpreted this as "send another byte of data" which in this case would be the PEC byte so the bq2060 is holding the data line low (trying to send a 0) waiting for clocks from the host. The host ends up just trying to send a stop bit since it is confused. Solution was to make the host send a NACK after the High Byte of data.

Example 4: Now hold on just a sec!



Notice the very long clock stretch by the slave here

The very long clock stretch here is pretty uncommon but still easily complies with the SMBus specification (25ms packet length). This is a very clean and clear example of a clock stretch. As with this example, most clock stretches happen somewhere around the ACK and NACK clocks.

Most common "Gotcha's" (things that cause problems)

1. **Capacitance:** It is very important that the data line is clean and well within an intended defined state when the clock pulse goes high, otherwise results can be unpredictable. Capacitance or a weak pullup can cause troubleshooting nightmares because "sometimes it works and sometimes it doesn't!" Roll-off caused by capacitance can cause the data line to be in an unknown state when the clock goes high causing confusion to the SMBus engine.
2. **Clock High Time:** This is the most common problem people forget about. The clock must be high for less than 50 micro seconds during the middle of a communication packet. Absolutely no exception to this rule. If it is high longer than 50 micro seconds then the SMBus engine on the slave will most likely time out.
3. **Communicating too fast:** This is most common when using an I2C hardware engine to do the SMBus communications. The SMBus engine is only specified at 100 KHz. Communicating faster than this will cause "timing minimum" rules to be violated.
4. **Broadcast or other collisions:** In most of our advanced Fuel gauge solutions, slave broadcasting (sometimes called Master Mode Messaging) is disabled by default so this problem has been greatly reduced in recent years. Most hosts or chargers do not accept broadcast messages so this should almost always be disabled. Rule of thumb is to disable if you are not using them.
5. **Wrong sized pull-up resistors:** Overall resistance should be about 10k for the SMBus clock line and 10K for the SMBus data line. Multiple masters can cause problems here because the resistors will be in parallel, reducing the overall resistance on the line. You

will notice poor “low” voltage whenever the SMBus line is pulled low. Remember to test this using an oscilloscope on the receiving end of the line when testing.

6. **Too Fast data to clock transitions:** This is often a problem on firmware controlled host engines. For most high speed processors you must put enough “NOP” instructions between the data line going high and prior to the clock going high on each bit. This is especially important on the start, stop, and repeated start bits..
7. **Bad PEC computation:** To date, no TI Fuel gauges that support PEC have been proven to produce a wrong PEC computation. If you are getting an error because the PEC is wrong then it is most likely an error in your PEC computation code. Check for “roll off” or “carry” errors.
8. **Non-SMBus compliant host:** We get many requests for help with failed communications because the host is not SMBus compliant but the customer cannot change the host design. Sometimes we can help with this. Sometimes we cant. We have to take these on a case by case basis. Our SMBus engines are mostly hardware engines so we have little control over how to manipulate them. This is why its important to insure that the host is SMBus compliant prior to releasing the product.
9. **Using an existing I2C engine to run SMBus:** This will usually work just fine. Pay particular attention to the I2C speed. Many modern I2C hardware ports on MCU’s can run at 400 KHz or 100 KHz. Always use 100 KHz. Even then, sometimes the port will exceed 100 KHz. Also make sure that there are no possibilities for the clock to be high more than 50 microseconds.
10. **Host not allowing for Clock Stretching:** Especially if the host is firmware driven. The host must check the clock after it sends it high to verify that it went actually went there. If it does not do this then the slave could be trying to slow it down and it will ignore this and continue. This will cause loss of arbitration and both master and slave will get confused.

Reference material

Crc calculator http://smbus.org/faq/crc8Applet.htm
Sbs specification http://sbs-forum.org/specs/sbdat110.pdf
Smbus specification http://www.smbus.org/specs/smbus20.pdf

Appendix A: SMBus Cheat Sheet

SMBus Read Word (without PEC)

0x16 0x0E 0x17 0x8C 0x86

S | Device Address | W | A | Com Code (0x0E) | A | S | Device Address | W | A | Data LSB | A | Data MSB | N | P

SMBus Read Word (with PEC)

0x16 0x0E 0x17 0x8C 0x86 0xD8

S | Device Address | W | A | Com Code (0x0E) | A | S | Device Address | W | A | Data LSB | A | Data MSB | A | PEC | N | P

SMBus Write Word (with PEC)

0x16 0x0E 0x8C 0x86 0xEE

S | Device Address | W | A | Com Code (0x0E) | A | Data LSB | A | Data MSB | A | PEC | A | P

SMBus Write Word (without PEC)

0x16 0x0E 0x8C 0x86

S | Device Address | W | A | Com Code (0x0E) | A | Data LSB | A | Data MSB | A | P

Legend:
- - - Slave control
—— Host Control

SMBus Clock Stretch

Slave holds low even though host releases clock

S | Device Address | W | A | Com Code (0x0E) | A

Slave is ready so it pulls data low and releases clock