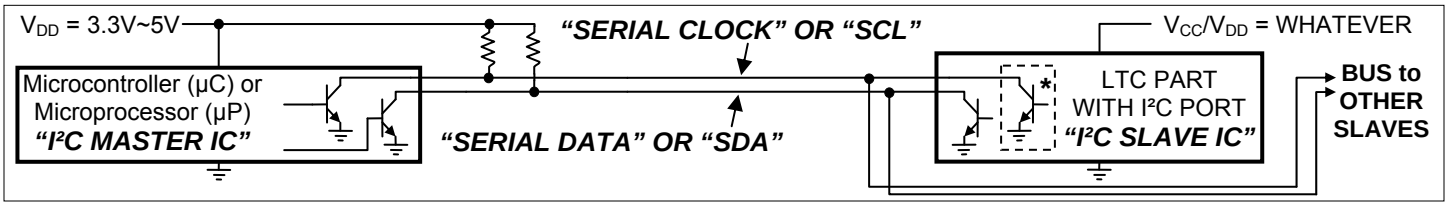
I²C/SMBus/PMBus Quick Reference Guide Rev. 1.0

Jason Sekanina – jsekanina@linear.com
µModule™ Power Products Design Engr.

REFERENCE MATERIALS:

[1] I²C-Bus Tutorial, Dilian Reyes, 11/7/02 (more, better details than here)
[2] Proper Software Impl. of I2C/PMBus/SMBus, Rev. 1.0, Chuen Ming Tan/Mark Gurries, 3/12/09 (Details multi-master)
[3] I²C and Related Bus Standards, Ver. 1.0, Mark Gurries, 9/11/08 (Compares I²C vs. SMBus vs. other system stds.)
[4] SMBus Rules to Follow for Successful SMBus Comm., Ver. 1.2, Mark Gurries, 4/16/08 (Eff. superceded by [2] and [3])
[5] I²C-Bus Spec. and User Manual, Rev. 3, 6/19/07 (Where it all began; elec. and other requirements to "do" I²C)
[6] System Mgmt. Bus (SMBus) Spec., Ver. 2.0, 8/3/00, SBS Implementers Forum (Standardizes 11 I²C-based protocols)
[7] PMBus™ Power System Mgmt. Protocol Spec., Part I, Rev. 1.1, 2/5/07 (New protocols to "do" PMBus)
[8] PMBus™ Power System Mgmt. Protocol Spec., Part II, Rev. 1.1, 2/5/07 (Register descriptions for PMBus devices)

THE MOST BASIC AND MOST COMMON I²C-TYPE SYSTEM ARCHITECTURE:



NORMAL OPERATION—POPULAR MISCONCEPTIONS—EXCEPTIONS—AND PITFALLS:

1. **IDLE:** When the I²C bus is idle—i.e., when there is no transmitting of data going on—the master IC's and all slave IC's SCL and SDA pins are high-impedance. External pull-up resistors pull SCL and SDA signals high. ***Always check this on an oscilloscope***, because stuck SCL or SDA signals to GND (due to soldering issues, damaged parts, bad grounding, etc.) is a very common cause of communication problems, data "full of zeroes", strange GUI behavior, etc.
2. **WHO DRIVES SCL and SDA:** The master IC only toggles the SCL line when it wants to transmit (write) data to a slave IC, or when it wants to receive (read) data from a slave IC. When SCL is high, only the master is allowed to change the state of the SDA signal. A slave should never toggle the state of SDA while SCL is high. By toggling SDA when SCL is high, the master can signal to all slaves on the bus what is known as "start", "repeat-start", and "stop" conditions. When the master is transmitting data to the slave, it also obeys the rule about not toggling the state of its SDA while its SCL is high. Most slave ICs do not have the capability to drive SCL low.

   *** EXCEPTION: SOME slave ICs contain a BJT/FET to pull SCL low** when it needs to. Customers are responsible for writing the code that goes into the µC/µP; they need to know ahead of time whether they need to write code that accommodates a slave IC with this so-called "clock-stretching" or "clock low extending" capability—i.e., the ability to pull SCL low. There exist cheap (or free) sources of code for implementing I²C, but: that kind of code usually does not accommodate for clock-stretching scenarios. Understandably, you may find customers are irked when they have to work with slave ICs with this feature; and therefore, I advise design engrs. to ***never*** put clock-stretching in their ICs, unless absolutely unavoidable. Product literature ***must*** state if, why, and when a slave clock-stretches. See Ref. [2] for example waveforms and free code for LTC customers.
3. **MORE ON SCL:** SCL does ***not*** run continuously like an oscillator; the master only operates SCL when it's time for someone on the bus (master IC or slave IC) to write **or** read data. The SCL signal is driven (by the master) approximately as a ~square-wave at some frequency—the "I²C Operating Frequency" or "Bus Speed". Note, the waveform need not be square-wave, but the signals do need to adhere to digital logic V_{IH} and V_{IL} requirements. The higher the frequency, the faster data can be transmitted over the bus: 100kHz ("Sm", Standard-mode) and 400kHz ("Fm", Fast-mode) are popular rates. Data rate is ultimately limited by the total parasitic capacitance on the bus reducing the rise/fall-time of SCL and SDA signals below acceptable limits.
4. **MORE ON SDA:** When SCL is pulled low (by the master), the opportunity arises for ***whomever is transmitting data*** (can be the master IC; or a slave IC) to pull their SDA pin low—for transmitting "logic 0"—or, to release their SDA pin high—for transmitting "logic 1". On the rising edge of SCL, the recipient of the data looks at the state of the SDA signal to determine whether a "0" or "1" is being transmitted. Recall that a slave should never toggle SDA when SCL is high—this ensures that, at the instant SCL goes high, the recipient will see the correct value of SDA on the bus.
5. *** CLOCK-STRETCHING:** If a slave IC is being asked to transmit data, but busy—e.g., performing other tasks such as trying to complete an A/D conversion or computations lasting a few milliseconds that would affect whether the data to transmit would be a "0" or "1"—a slave IC with clock-stretching capability can legitimately pull and hold SCL low immediately after the master IC pulls its SCL low. As long as SCL is held low, the recipient of data (master) is waiting for the transmitter of data (slave) to "set up" SDA. In other words, holding SCL low "buys" the transmitter (slave) time to finish other tasks before driving SDA to the proper logic setting. "Cheap and dirty" I²C code implementations freely assume that if the master isn't driving SCL low, then no one else is—but this assumption is correct only if no slaves have clock-stretching capability. This is where low-grade code in the master IC would need to be modified to

recognize whether a slave is holding SCL low after the master's internal circuitry has released its own SCL. There exist rules in SMBus on how long a slave can hold SCL low (25-35ms); in I²C, there is no restriction on how long a slave IC can legitimately extend the clock to GND, but as a result everyone has paranoia about "something faulty happening" in the slave such that the slave never releases its SCL from GND—known as a fatal "hung bus" condition.

6. **KEY TERMS: BITS, NIBBLES, BYTES, AND WORDS.** There are 8 bits in a byte. When 8 bits are written down on a piece of paper, the leftmost bit is called "bit 7", and the rightmost bit is called "bit 0". "Bit 7" in this example is the msb, or "most significant bit"; and "bit 0" is the lsb or "least significant bit". The byte of data can be referred to as "bits [7:0]", pronounced "bits 7 through 0". The middle two bits can be referred to as "bits [4:3]". There are 2 bytes in a word. In a word of data, the 16 bits may be referred to as "bits [15:0]". Bit 15 is the msb of the word. Bit 0 is still the lsb of the word. The first 8 bits of the word—i.e., the "upper byte"—are "bits [15:8]", and can collectively be referred to as the word's most significant byte, or MSB (CAPS!). The rightmost 8 bits of the word—i.e., the "lower byte"—are "bits [7:0]", and can be referred to as the word's least significant byte, or LSB (CAPS!). The middle 6 bits of the word are "bits [10:5]". The bottom 5 bits of the MSB can be referred to as "bits [12:8]". Got it? Oh, I almost forgot. There are 4 bits in a nibble, and therefore 2 nibbles in a byte. The upper nibble of the byte (bits [7:4]) can be referred to as the most significant nibble, and the lower nibble (bits [3:0]) as the least significant nibble—the abbreviations are "msn" and "lsn", respectively, but I honestly have not seen those abbreviations used in any vendor-customer interactions.
   PITFALL: "MSB" (in caps) stands for "most significant byte". "msb" (in lowercase) stands for "most significant bit". "LSB" (in caps) stands for "least significant byte". "lsb" (in lowercase) stands for "least significant bit". I didn't know this for the longest time, and I developed a habit of referring to "MS-bit", "MS-byte", "LS-bit", "LS-byte" after encountering others confused or unfamiliar with the abbreviations. **WARNING:** In Ref. [6] and in datasheets I've seen, "MSB" refers to the most significant bit—so multiple conventions exist in the industry. ***Use these abbreviations cautiously!***

7. **BINARY AND HEXADECIMAL:** In the digital world, one must have at least a mediocre handle on binary and hexadecimal (hex) numbers. Counting from 0 to 15 in decimal, and converting to binary, and hex, respectively:

Decimal	0d	1d	2d	3d	4d	5d	6d	7d	8d	9d	10d	11d	12d	13d	14d	15d
Binary	0000b	0001b	0010b	0011b	0100b	0101b	0110b	0111b	1000b	1001b	1010b	1011b	1100b	1101b	1110b	1111b
Hex	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F

   Notice that any nibble of data can be converted to a hex digit. Data on the I²C bus is frequently dealt with in "8-bit-chunks", which is two hex digits. Tip: Microsoft Windows' Calculator program, calc.exe, contains an effective tool for converting binary numbers to hex and vice-versa. Select "View->Scientific" menu and see "Hex" and "Bin" buttons in the upper-left region of the GUI; note also shortcut "F5" and "F8" keys, for hex- and binary-modes, respectively.
   PITFALL: A "b" suffix indicates a binary number, whereas a "0x" prefix indicates a two-digit hex number. A prefix of "8'h" or "'h" can also indicate a two-digit hex number. In casual correspondence, the suffix and prefix indicating base are frequently dropped due to laziness, and the base is clear from context. In hex, leading zeros are sometimes dropped when referring to memory register locations, but can lead to confusion if dropped when referring to a stream of actual data. When sharing data in binary format, I tend not to drop leading zeros, but I do drop the "b" suffix.

8. **SLAVE ADDRESS:** Every slave IC on a bus must have its own, unique slave address. A slave address is usually comprised of 7 bits (see exception below), i.e. bits [6:0]. An example slave address is "0010100" (binary). For reasons which will become clear shortly, it is common to write the slave address with its first nibble and last 3 binary digits separated with a space, or in this example: "0010 100". It may be tempting and seem innocent to convert this binary number to hex and refer to this slave address as "0x14", but this is actually an ***extremely dangerous practice*** and should not be done unless you know your customer/audience shares your philosophy. (See pitfall below.) In practice, unique slave addresses are typically assigned by uniquely tying each slave ICs' chip address pins (CA0, CA1,…) low, high, or left floating, or by tying a unique slave-address-programming resistor from each slave IC's SADDR pin(s) to GND—product literature must have a slave address table to show what slave addresses the slave IC supports. In most applications, a slave address is constructed of 7 bits, as is the case when implementing "I²C 7-bit addressing". In theory, this would yield a maximum of 2^7=128 slave addresses, the lowest being 0000000b ("b" for "binary format") to 1111111b—however, 21 slave addresses are "reserved" cumulatively amongst the I²C, SMBus, and PMBus specs., and therefore not recommended to be supported by a slave IC. See chart for reserved addresses:

RESERVED SLAVE ADDRESSES

Slave Address Bits [6:0]	Reserved For:	I²C Spec. (Ref. [5])	SMBus Spec. (Ref. [6])	PMBus Specs. (Refs. [7]and [8])
0000 000	General Call Address and Start	x	x	x
0000 001	CBUS Address	x	x	x
0000 010	Different Bus Format	x	x	x
0000 011	Future Use	x	x	x
0000 1XX	Hs-Mode Master Code	x	x	x
0001 000	SMBus Host		x	x
0001 100	SMBus Alert Response Address		x	x
0101 000	ACCESS.bus Host		x	x
0110 111	ACCESS.bus Default Address		x	x
1100 001	SMBus Device Default Address		x	x
1111 0XX	10-Bit Slave Addressing	x	x	x
1111 1XX	Future Use	x	x	x

EXCEPTION: I²C spec. ([5]) describes a way for the user to implement 10-bit addressing, which is fully compatible with 7-bit addressing, and enables a user's bus to contain up to ~2^{10} slave ICs. 10-bit addressing is uncommon because of the excessive amount of parasitic capacitance so many slaves present to the bus—and their resulting impact to achievable bus speed. 10-bit addressing is beyond the scope of this document—see Ref. [1] for more info.

PITFALL: Tragically, in the industry, IC vendors are inconsistent in how they document what slave addresses their devices support, i.e., how a slave address is indicated in a product literature's slave address table. 7-bit addressing is accomplished with—as the name suggests—7 bits in the slave address. To minimize customer and FAE confusion, designers should ***explicitly*** indicate what 7 bits ([6:0]) correspond to a particular slave address setting. Table 1 illustrates the most unambiguous way to convey slave address information to a customer. Believe it or not: to the trained eye, the contents of every single column in Table 2 adequately indicates in shorthand notation *fully equivalent* information to what is presented in the 7 columns of Table 1 entitled "A6", "A5",… "A0". For an I²C novice, this is non-obvious and requires explanation (you may want to read about I²C transactions first, however).

TABLE 1. EXAMPLE HYPOTHETICAL SLAVE ADDRESS MAP

CA1	CA0	A6	A5	A4	A3	A2	A1	A0
GND	GND	0	0	1	0	0	0	0
GND	FLOAT	0	0	1	0	0	0	1
GND	VCC	0	0	1	0	0	1	0
…etc…	…etc…	…	…	…	…	…	…	…
GLOBAL ADDRESS		1	1	1	0	0	1	1

TABLE 2. FULLY EQUIVALENT SLAVE ADDRESS TO TABLE 1; NON-OPTIMAL PRESENTATION

…	Address[7:1] (binary)	Address (7-bit address plus R/W bit)	Address[7:0] (R/W=0)	Address[7:1] (hex)	Address[7:0] (R/W=0) (hex)
…	0010000	0010000_R/W	00100000	0x10, or 7'h10	0x20, or 8'h20
…	0010001	0010001_R/W	00100010	0x11, or 7'h11	0x22, or 8'h22
…	0010010	0010010_R/W	00100100	0x12, or 7'h12	0x24, or 8'h24
…	…	…	…	…	…
…	1110011	1110011_R/W	11100110	0x73, or 7'h73	0xE6, or 8'hE6

When the master IC indicates on the bus which slave it wants to talk to, it will call out the target slave's 7 address bits plus one more bit called the "read/write" or R/W bit. The R/W bit is "1" for a read or "0" for a write. Because this 8th bit is always appended to the slave address when addressing the slave, vendor documentation sometimes lists slave addresses with the R/W bit implied in the address. Furthermore, some vendor documentation takes the liberty to report the slave address in hex while applying the R/W bit set to 0—and not even tell you! From Table 2, it should be clear why it is asking for trouble to indicate a slave address in hex: the slave address is 7 bits long—do you convert the 7 bits directly to hex, or do you lump the R/W bit into the slave address with R/W=0, and then convert that 8-bit number to hex? The results produce two different interpretations of the address; the latter number is a multiple of 2 of the former number. (I have not seen any vendors or customers report slave address in hex while setting the R/W bit to 1.) If a customer talks about any slave addresses in hex that are odd-valued, then they are converting bits [6:0] to hex directly (no R/W bit); if a customer talks about even-valued slave addresses exclusively—or, if their slave address exceeds 0x77, they probably are appending a logic "0" R/W bit and then converting to hex—but you can't be sure without asking. Additionally, be aware some people refer to "7-bit addressing" as "8-bit addressing", when the R/W bit is again lumped into to slave address's name and not distinguished from the 7 msbs—the real slave address.

9. **I²C TRANSACTIONS AND PROTOCOLS:** Data is transferred on the bus in byte-sized chunks, literally: in 8 toggles of SCL, one byte of data is transmitted over SDA from one IC to another. Now, the master IC cannot simply "ask" the bus for 8 bits of data, nor blindly "send" 8 bits of data to the bus: it needs to specify which particular slave IC it wants data from or write data to. This is why the slaves need to have unique slave addresses—to allow the master to converse with a specific slave IC (**EXCEPTION—SEE "GENERAL CALL ADDRESS"**). To go a step further: the most simple I²C-enabled ICs may only have 8 bits of data total in their ownership, but more complex ICs have 3 bytes, 12 kb (kilobits), 4kB (kilobytes—notice capital "B"), or much, much more data within its confines. How does a master IC tell the slave which particular byte of data within the slave's confines it wants to read?

   What is important to establish early on here is: there exists sentence structures called "protocols" that govern how different kinds of data are written to or read from a slave. Start and repeat-start conditions can be thought of as the master IC "clearing its 'throat', preparing to 'speak'" to the slave ICs on the bus. A stop condition is the master "finishing its sentence and 'telling'" the slaves that the bus is returning to an idle state.

   The situation begins with an idle bus. The master decides it wants to either: write data to; or, read data from; a slave IC. Coming out of an idle bus state, the master must, in sequence:

   1. First, produce a start condition or "start bit" on the I²C bus. (This requires one toggle of SCL.)
      (This alerts the slaves that the master is going to address a slave on the bus)
   2. Second, indicate the target slave IC's 7-bit slave address on SDA—one bit at a time. (7 toggles of SCL, total.)
   3. Next, indicate whether the instruction to the slave will be a read or a write by clocking through a "1" or a "0" on SDA, respectively. This bit is called the "read/write" or R/W bit, and all protocols require appending the R/W bit to the slave address when addressing the slave. (This requires one toggle of SCL.)

   After the master has indicated the target slave IC and R/W bit, the master releases the SDA line. The slave IC has one SCL clock pulse of time in which to "acknowledge" (or ACK) the master. The slave ACKs the master by pulling SDA low (logic "0") before the SCL clock goes logic high. The purpose of the slave ACKing the master at this point is so that the master knows that the intended target slave IC has heard the master's calling. If the target IC fails to pull

SDA low (leaving SDA floating; logic “1”), this is called a “no ACK” (or NACK), and the master knows that something is wrong: either its target slave IC didn’t hear its name called, or the target slave IC is otherwise occupied and cannot accept queries from the master at this time. An unresponsive target IC is usually the result of noise on the bus, a soldering problem, lack of Vcc/Vdd power to the target IC, or too much capacitance on the bus impeding proper communications. NACKs of this kind cannot be accepted in a robust system!

What happens next depends on what the product literature says about the IC’s supported I²C protocol transactions.

I need to continue with a specific example, say the “send byte” protocol, in which one byte of data is written from the master IC to a slave IC. This applies to slave ICs who have only one byte of data contained in them. Continuing from where I left off in step 3, the master’s R/W-bit would have been a “0” (write), and:

(R/W=0 example)

…

4. The slave ACKs the master. (This requires one toggle of SCL.)
5. The master feeds its byte of data to the target slave IC over the SDA line—one bit at a time, starting with the msb, and ending with the lsb. (This requires 8 toggles of SCL, total.)
6. The slave ACKs the master to reaffirm the receipt of 8 bits of data. (This requires one toggle of SCL.)
   (The slave replaces its byte of data with the newly receive byte of data from the master IC.)
7. The master sees the slave’s ACK and issues a stop condition on the bus. (This requires one toggle of SCL.)
   (All slave ICs see the stop condition on the bus and know that the bus has returned to an idle state.)

Alternatively, let us explore the “receive byte” protocol, in which the master IC reads one byte of data from a slave IC. This again applies to slave ICs who have only one byte of data contained in them. Continuing from where I left off in step 3, the master’s R/W-bit would have been a “1” (read), and:

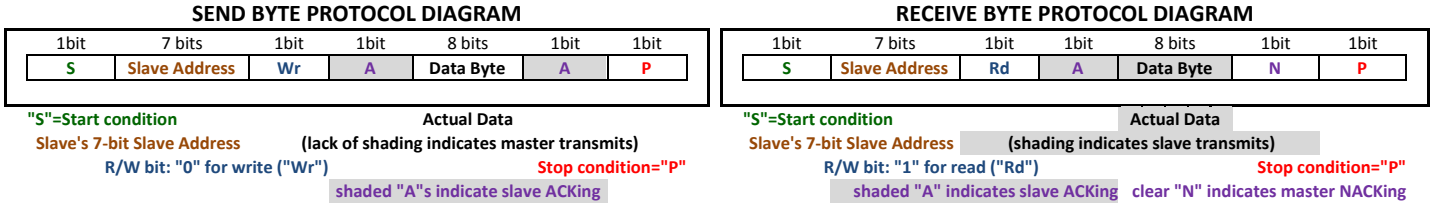
(R/W=1 example)

…

4. The slave ACKs the master. (This requires one toggle of SCL.)
5. The slave feeds its byte of data to the master IC over the SDA line—one bit at a time, starting with the msb, and ending with the lsb. (This requires 8 toggles of SCL, total.)
6. The ***master*** leaves SDA floating to NACK the slave (!). This informs the slave that the master is done retrieving data from the slave. (One toggle of SCL, total.)
   (The master uses this data however it wishes; the data in the slave is unchanged.)
7. The master issues a stop condition on the bus. (This requires one toggle of SCL.)
   (All slave ICs see the stop condition on the bus and know that the bus has returned to an idle state.)

By the way, if the slave ever NACKs the master IC, it is prudent for the master to issue a stop condition immediately after the slave’s NACK, to abort the transaction.

10. **SIMPLE PROTOCOL DIAGRAMS:** It is tedious to describe what transpires between master and slave ICs on a bit-by-bit and even byte-by-byte basis. A shorthand notation exists and must be understood and utilized in order to peacefully coexist with I²C. Below are diagrams for the Send Byte and Receive Byte Protocols described in (9):



Key features to recognize and appreciate in this shorthand notation:

- The diagrams illustrate a specific sequence for how a Data Byte is communicated between master and slave ICs.
- Each rectangular box contains an abbreviation or name for the kind of data that is transferred over the bus during that segment of time, i.e. “S” for start condition; “Sr” (not shown) for repeat-start; “Rd” or “Wr” for a logic 1 or 0 for the R/W bit, respectively; “A” for ACK or “N” for NACK; “P” for a stop condition. In the above Send/Receive Byte example protocols, the “Data Byte” is the real data of interest to the master and slave ICs; product literature would explain what each bit in the Data Byte represents as it pertains to IC features, functionality, or the application.
  EXCEPTION: In SMBus spec. ([6]), NACK is indicated with “A”, but with a small “1” beneath the box; the “1” indicates the ACK/NACK bit is logic “1” (NACK) in normal communications. In I²C spec. ([5]), NACK is indicated with $\overline{A}$ (the bar over the “A” indicates the “logical inverse of ACK”). In PMBus specs. ([7], [8]), NACK is indicated with “NA” vertically written: $\frac{N}{A}$. (Technically, I am introducing non-standard notation here using “N” for NACK!)
- A ***shaded*** rectangle indicates the slave IC is transferring data to the master—meaning, the slave transfers one bit of data onto the SDA bus per toggle of SCL. A ***clear*** rectangle indicates the master IC is transferring data to the slave(s)—meaning, the master transfers one bit of data onto the SDA bus per toggle of SCL.

EXCEPTION: Sometimes the convention of "shaded-box=slave/clear-box=master" is reversed to "shaded-box=master/clear-box=slave"; the start, repeat-start, and stop bits are always transmitted only by the master, and therefore the shading (or lack of shading) of the "S" rectangle will clue you in to the convention being used.

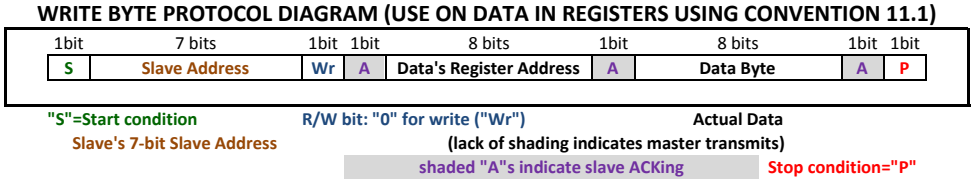
- Above each rectangular box is a number indicating the number of bits transmitted during that block of time—and consequently, the number of SCL toggles that occur to transfer that number of bits.

11. **MORE THAN 1 BYTE OF DATA IN A SLAVE? DATA REGISTER ADDRESSES!** Thus far, I have only described the simple Send Byte and Receive Byte Protocols, which are only applicable to slave ICs with only 8 bits of data in their ownership—another way of saying this: …slave ICs with only 8 bits of data in their memory. Data is stored in memory. If a slave has two bytes of data in its memory, how would a master tell a slave IC whether it wants to read/write the first byte vs. the second byte? There exist two very popular approaches for tackling this:
    1. The first byte of data is assigned a ***data register address***. The second byte is also assigned a data register address, but one that is different from the first byte's register address. Example: the first byte resides at register address 0x01, bits [7:0]. The second byte resides at register address 0x02, bits [7:0]. Extending this convention, additional data bytes are assigned unique register addresses, e.g. 0x03, 0x04, ….
    2. The first byte and second byte are concatenated to form a data word with an MSB and LSB. The resulting data word is assigned a data register address, example: 0x01, bits [15:0]. Extending this convention, additional data words are assigned unique register addresses, e.g. 0x02, 0x03, ….

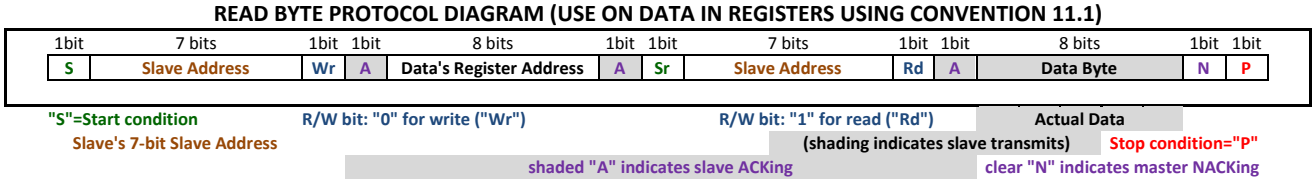
    By breaking up data into byte-sized (or word-sized) chunks and uniquely identifying each data byte through a register address (either as itself, or by its MSB or LSB location within that register), it becomes possible for the master IC to address a particular byte of data within a particular slave IC on the I²C bus—using more advanced protocols of course, such as: Write Byte, Read Byte, Write Word, and Read Word. Just as slave addresses are used to distinguish slave ICs from one another on the bus, register addresses are used to distinguish data bytes and/or words from one another within a slave IC. The collection of data within a slave IC's memory has no meaning without adequate product literature explaining via a "memory map" or "register map" or table what each bit of data in memory means to the user.
    PITFALL: There exist IC slave addresses, and data register addresses. Often in conversation or correspondence, the terms "slave" and "register" are dropped, and the address type becomes clear from context. Formal documents need to be clear to the reader in discerning data register addresses from IC slave addresses.
    PITFALL: A slave IC's register map is not restricted to utilizing 8-bit registers (convention 11.1) or 16-bit registers (convention 11.2), exclusively. Depending on the nature of the data and logical groupings of the data, the IC designer could e.g. choose to make registers 0x01 and 0x04 8-bit registers, and registers 0x02 and 0x03 16-bit registers. Although this is legitimate, there exists motivation for convenience to the user to minimize the number of unique data structures—and hence, protocols—required for the I²C master to interact with the slave IC. More protocols results in the need for more complexity in the µC or µP programming code—so from this point of view, it is not recommended to mix usage of 8-bit and 16-bit register data type in an IC's memory without good justification.

12. **WRITE/READ BYTE PROTOCOLS:** With limited knowledge of the start/stop bit, slave address, R/W bit, ACK/NACK bit, register address, and knowing that data is transmitted in byte-sized chunks, it becomes possible to see how slave ICs with multiple bytes of data can receive and transmit specific bytes of data from/to a master IC. Below are protocol diagrams for the popular Write Byte and Read Byte transactions:

WRITE BYTE PROTOCOL DIAGRAM (USE ON DATA IN REGISTERS USING CONVENTION 11.1)



READ BYTE PROTOCOL DIAGRAM (USE ON DATA IN REGISTERS USING CONVENTION 11.1)

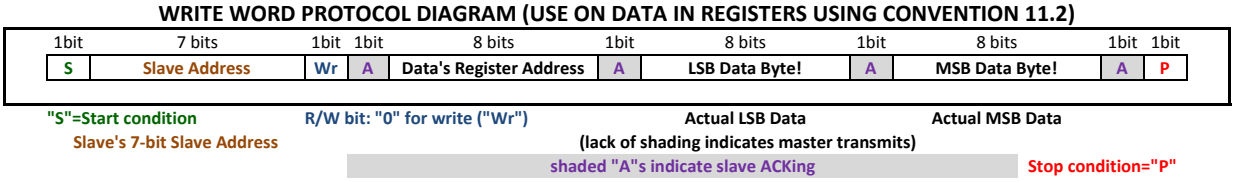


I personally find the Write Byte protocol to be very intuitive to an I²C newcomer. The master states the target slave IC's slave address; indicates a "write" in the R/W bit; indicates the target register address in the target slave IC; and finally writes the Data Byte of interest to that slave's memory location (with the Slave ACKing along the way).
PITFALL: "Data's Register Address" is actually referred to as "Command Code" by everyone else in the industry. I chose to deviate from that nomenclature because "Command Code" doesn't initially make me think of "Data's Register Address". "Command Code" always refers to the register address of the data that you're interested in reading from or writing to.
PITFALL: In contrast to the Write Byte protocol, I find the Read Byte protocol less intuitive, even now. The master begins by stating the target slave IC's slave address; indicating a "write" (!) in the R/W bit; the master then indicates to

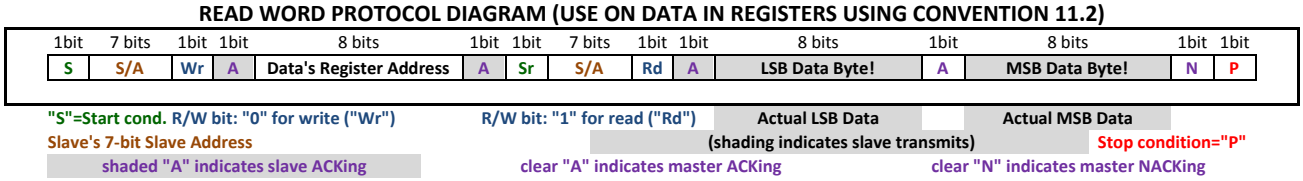
the slave the target register address from which it wants to read data, but it does not make the intention to read that data known until issuing a repeat-start condition; re-stating the slave-address; and finally, using R/W bit="1". Upon the slave's ACK of the master's read request, the slave transmits the byte of data of interest to the master. The master NACKs to signal it is done reading data from the slave; followed by issuing the usual stop condition.

13. **WRITE/READ WORD PROTOCOLS:** The Write Byte and Read Byte protocols are applicable for interacting with byte-sized data, where the target byte of data has a dedicated register address. In an analogous fashion, the Write Word and Read Word protocols exist for writing or reading word-sized data—i.e., data composed of two concatenated bytes (MSB and LSB), but with one register address. Below are protocol diagrams for the popular Write Word and Read Word transactions:

WRITE WORD PROTOCOL DIAGRAM (USE ON DATA IN REGISTERS USING CONVENTION 11.2)



READ WORD PROTOCOL DIAGRAM (USE ON DATA IN REGISTERS USING CONVENTION 11.2)



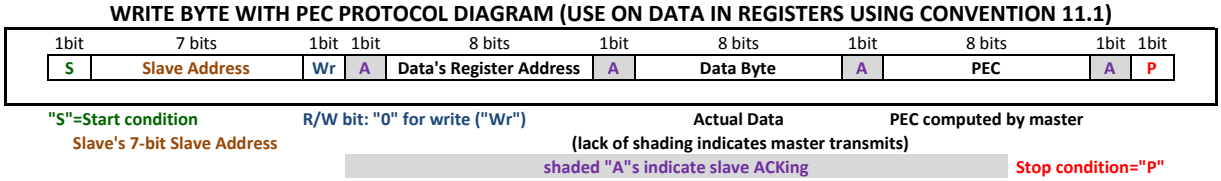
PITFALL: The Write Word protocol is relatively intuitive, except: the LSB data byte is transmitted prior to the MSB data byte. Again, I repeat: the register's data contents are bits [15:0], but bits [7:0] are transmitted prior to bits [15:8].

PITFALL: As was the case with the Read Byte protocol, the Read Word protocol requires a write to the slave IC's register of interest prior to issuing a read request (R/W bit=1). Note, in a fashion analogous to the write word protocol, the read word protocol has the LSB data (bits [7:0]) transmitted to the master prior to the MSB data (bits [15:8]). However, something new: observe that the **master** ACKs the slave following receipt of the LSB data byte. The ACK indicates to the slave that the master is not done retrieving data from the target slave's register.

PITFALL: PMBus specs. ([7], [8]) use SMBus Rev. 1.1 as a reference document. Tragically, the Read Word protocol in SMBus Rev. 1.1 shows the LSB and MSB bytes erroneously reversed! This typo was corrected in SMBus Rev. 2.0 ([6]). In writing/reading words, failing to reverse the MSB and LSB byte order is a common error.

14. **PACKET ERROR CHECKING:** Noise on the I²C bus, poor grounding, and/or sluggish rise/fall times on the SCL or SDA signals can cause data transmission errors to occur. In the protocols discussed thus far, you will note that the use of start, stop, repeat-start, ACK, and NACK bits gives some assurances that a coherent conversation is occurring between master and slave ICs without loss of data integrity (packet errors, or "packet loss"). However, the master IC does not really have any assurances that the data in between the ACKs and NACKs of the conversation is being accurately transmitted to the receiving party. To improve communication robustness, a customer may be interested in implementing a bus that uses packet error checking—though, this feature is optional and seems to be seldom supported by slave IC products in the industry. For every data byte or data word transmitted on the bus, the transmitter of the data and the recipient of the data can independently compute what is called a PEC, or "packet error code". ("PEC" is pronounced "peck"; not spelled out.) Explaining how the PEC is computed is beyond the scope of this document, but in short: the PEC is an 8-bit binary number whose value depends on a computation seeded by the data being transmitted from one IC to another on the bus. If the data is transmitted from one IC to another without errors, then each participant in the transaction independently computes the same PEC. If both ICs in a conversation independently compute the same PEC based on data they transmitted or perceived on the bus, the master can have some relative comfort that the data was correctly transmitted to the slave or received from the slave—however, ironically, note that even the ACK bit is vulnerable to data corruption, and hence implementing PEC in-and-of itself does not guarantee flawless data transmission and verification. Different PEC algorithms exist, though an 8-bit cyclic redundancy check ("CRC-8") is most popular amongst those who do implement PEC (endorsed in SMBus spec., Ref. [6]). Minor variations to the protocols discussed thus far exist, in which the PEC byte can be transmitted over the bus subsequent to the data byte(s). A brief overview of the Write Byte and Read Byte protocols with PEC is in order:

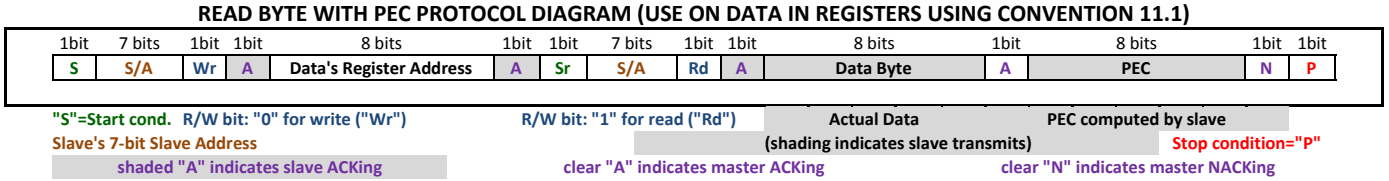
WRITE BYTE WITH PEC PROTOCOL DIAGRAM (USE ON DATA IN REGISTERS USING CONVENTION 11.1)



The Write Byte with PEC protocol resembles the Write Byte protocol, except: instead of issuing the stop condition after the data byte transfer and slave's ACK, the master proceeds to transmit the PEC it computed to correspond with

the data byte it transmitted. The slave ACKs the master's PEC if it independently computed the same PEC value based on the data in the data byte it received; otherwise, it NACKs the PEC. Note that the master IC can legitimately issue a stop condition after the slave ACKs the data byte; the slave must recognize that this aborts PEC-verification. Observe that slave ICs with PEC capability can *and must* be compatible to master implementations not using PEC.

READ BYTE WITH PEC PROTOCOL DIAGRAM (USE ON DATA IN REGISTERS USING CONVENTION 11.1)



The Read Byte with PEC protocol resembles the Read Byte protocol, except: (1) the master *ACKs* the slave after receipt of the data byte, indicating it is not done retrieving information from the slave IC, and (2) the slave IC subsequently transmits the PEC it computed on its data byte. The master NACKs the PEC *only* to inform the slave that it is done retrieving information, and *does not signal* whether the slave's PEC agrees with the PEC the master independently computed on the data byte it received. Note that the master IC can legitimately NACK the slave's transmission of the data byte—signaling to the slave completion of data retrieval—and the slave must comply in aborting PEC-transmission. Observe again that slave ICs with PEC capability can *and must* be compatible to master implementations not using PEC. In similar fashion, nearly all read/write protocols are extensible to support PEC.

15. **OTHER SMBUS PROTOCOLS; BLOCK WRITE AND BLOCK READ:** SMBus spec. 2.0 ([6]) standardizes eleven (11) command protocols; SMBus spec. 1.1 standardizes ten (10)—not counting those "with PEC" and those "without PEC" separately. I have covered six (6) popular SMBus protocols thus far: Write/Read Byte/Word and Send/Receive Byte. The other protocols are well defined in the SMBus specs., illustrated with protocol diagrams very familiar to the ones explained here to help you learn on your own. It is worthwhile to mention two additional protocols in the SMBus specs. that are popularized by PMBus: Block Write, and Block Read. Block Write/Read commands are very similar to the Write/Read Byte protocols, except they are flexible in accommodating data registers containing a variable number of bytes: from 1 byte, minimum, to 32 bytes, maximum. The first data byte transmitted in the Block Write/Read protocols contains the integer "N", where N corresponds to the number of data bytes to be written to or read from the IC's target register—converted from decimal to binary, and prefixed with leading zeros to make it 8 bits long, total. IC product literature must indicate how many blocks of data the IC's target registers support.
PITFALL: When transmitting data using block write/read protocols, the data bytes following N's byte are transmitted in an intuitive, sequential order, with MSB first and LSB last. Note that this is inconsistent with the Write/Read Word convention, in which the LSB is transmitted before the MSB. Watch out for incorrect implementations!
EXCEPTION: PMBus allows Block Write/Read transfers with 1≤N≤255 bytes. SMBus limits it to 1≤N≤32 bytes.
16. **GENERAL CALL ADDRESS:** 0000000b is a reserved slave address for "general call addressing" or "global address calling". The master IC can use the "write byte" protocol with slave address 0000000b to write a byte of data simultaneously to all slave ICs on the bus. It is not required for slave ICs to support the general call address of 0000000b. There are variations on this theme, e.g. all LTC2487 ADC ICs will respond in unison to a global address call at slave address 1110111b; LTC2487's datasheet explains in detail what behaviors can be synchronized.
17. **SMBALERT#:** There is a signal called SMBALERT# described in SMBus—optional to implement—where a slave-only IC can pull low on a bused SMBALERT# signal to indicate to the master IC an "interrupt"—that it needs attention.
18. **WHAT ABOUT "MULTI-MASTER" SYSTEMS?** I²C, SMBus, and PMBus specs. all accommodate having multiple master ICs on the same serial bus, meaning, multiple ICs can be capable of driving the SCL signal for purposes of clocking-through data on SDA. However, multi-master systems introduce a lot of complexity to the system requiring arbitration in case of "bus collisions" (masters fighting for control of the SCL)—are very unpopular because of that—and are beyond the scope of this document. See Ref. [2] for good details on multi-master systems.
19. **SO WHAT *IS* THE DIFFERENCE BETWEEN I²C, SMBUS, AND PMBUS?** I²C is the basic foundation for defining a general serial interface between master ICs and slave ICs. SMBus is in many ways a re-hash of I²C, but introduces the notion of using only ~10-11 protocols that standardize transfers of various kinds of data over the bus (bytes, words, block bytes)—in this respect, SMBus is truly a subset of I²C. SMBus and its standardization of protocols have been positively received in the industry. Leveraging protocols defined in SMBus spec. can reduce (or eliminate!) the overhead in product literature associated with illustrating the protocol diagrams used in accessing each and every register of a product. Products that conform to I²C spec. are not limited to using the ~10-11 SMBus protocols defined by SMBus spec.—in other words: a product that is "only" I²C means that the product literature will need to contain detailed protocol diagrams for each register of the product that may not resemble the ~10-11 protocols in the SMBus spec.—whereas, an SMBus device will "mostly" limit itself to using the ~10-11 standardized protocols in its transactions to its registers. Leveraging the SMBus spec. enables product literature for a device to summarize, say, in a tabular format—rather than lengthy and/or repeated protocol diagrams—how many bytes of data are contained in a register, and what SMBus protocols is/are used to read and write to each data register. The main thrust of PMBus is to standardize a power supply device's register address locations for popular power supply

parameters, such as: switching frequency (reg. 0x33), output voltage (regs. 0x20 and 0x21), margin-low (reg. 0x26) and margin-high (reg. 0x25) voltages, over-current inception threshold (reg. 0x46), UVLO-rising (reg. 0x35), UVLO-falling (reg. 0x59), turn-on-rise time (reg. 0x61), fault responses (misc. regs.), etc.; and power supply telemetry, such as: temperature (regs. 0x8D - 0x8F), output voltage readback (regs. 0x20 and 0x8B), load current (reg. 0x8C), input voltage (reg. 0x88), input current (reg. 0x89), etc.—using SMBus standard protocols (and the corresponding register types: bytes, words, block-bytes) for register addresses it defines between 0x00 - 0xCF. Appendix I at the end of Part II of the PMBus spec. summarizes in tabular form these registers and data formats. PMBus defines data registers 0xD0 - 0xFD as "manufacturer-specific registers" or "manufacturer-specific commands", intended for allowing interaction with data outside the categories covered in registers 0x00 - 0xCF but pertinent to the power supply's functions or features, e.g. programming discontinuous vs. burst-mode vs. forced continuous operation, phase interleaving settings for a multi-phase operation, phase-locked-loop and/or synchronization (PLL and/or sync) options, etc.—the protocols for registers 0xD0 – 0xFD are up to the power supply's vendor. Register 0xFE is defined by PMBus to provide an extension to the manufacturer-specific commands when registers 0xD0 - 0xFD are insufficient. You may be wondering: how does one register (0xFE) "extend" the available command codes beyond more than the one register, 0xFE itself? The answer is: 0xFE is not really a register. Think of 0xFE as a "door", with additional registers numbered between 0x00 and 0xFF behind the door. This distinguishes the registers 0x00 - 0xFF behind the 0xFE door as being different data register locations from the 0x00 - 0xFD you can read/write to without accessing the 0xFE door. More specifically: transactions to 0xFE do not follow a SMBus protocol: Part I of the PMBus spec. contains protocol diagrams illustrating how one reads or writes to command codes 0x00 - 0xFF residing behind the 0xFE door. Nothing prevents power supply and IC designers from implementing extension codes with the 0xFE-prefix, but it is most intuitive and friendly to the user of the product if features and functions not covered by register categories 0x00 - 0xCF be implemented in data registers 0xD0 - 0xFD *before* resorting to implementing extended commands in "0xFE-space". I did not yet mention command code 0xFF, which PMBus reserves for future extension of the PMBus-standard command codes 0x00 - 0xCF in an analogous way as the 0xFE extension code extends the manufacturer-specific command codes 0xD0 - 0xFD. (Some registers in the range 0x00 - 0xCF are currently "reserved for future use", and will likely be populated before command codes behind the 0xFF door are tapped). I believe it would be in everyone's best interest at Linear Tech for IC designers to try coordinating efforts so as to standardize their usage of common command codes in PMBus-heavy IC products: registers 0xD0 - 0xFD and extended commands 0x00 - 0xFF behind 0xFE; this would help customers and FAEs become familiar with certain registers having consistent functions, across multiple products. (I know: "easier said than done".) Lastly, PMBus spec., Part I, introduces the "Group Command Protocol"—a protocol not in SMBus—which explains a rather clever and elegant way for the master IC to synchronize multiple PMBus slave ICs to perform I²C transactions in unison at the execution of the stop condition. I believe the group command protocol is very novel and may have usefulness beyond power supply applications: I encourage IC designers to consider implementing support of the group command protocol in ICs that they otherwise intended to be I²C- or SMBus- "only". Support of the group command protocol would, for example, elegantly remove the need for implementing a "custom" global call address of 1110111b in LTC2487 (see paragraph 16). The differences that I see important to distinguish between I²C, SMBus, and PMBus specs. are:

TABLE 3. SPEC. COMPARISONS	I²C Spec. Rev. 03 ([5])	SMBus Specs. 1.1 and 2.0 ([6])	PMBus Spec., Part I, Rev. 1.1 ([7])	PMBus Spec., Part II, Rev. 1.1 ([8])
Purpose	Introduces a general 2-wire bus concept (SCL and SDA)	Standardizes ~10-11 protocols for data transfer over a 2-wire bus	Mostly uses SMBus 1.1, but introduces 2 add'l protocols	Defines register map and register functions for a generic power supply
Bus Speed	Various; DC, min. - 3.4MHz, max.	10kHz, min. - 100kHz, max.	capability for 100kHz required; 400kHz support optional	
SCL time-out-detection	None exists ("bus hang" can occur)	25ms min. - 35ms max.		
Slave ACKing a master calling out its slave address	Not required, if e.g. slave is busy with a computation or other real-time task	IC is required to ACK the master regardless of readiness to process data transactions		
PEC	Not discussed	Concept introduced; optional to implement		
SMB_Alert#	Does not exist	Concept introduced; optional to implement		
PMBus Group Command Protocol	Does not exist	Does not exist	Protocol defined in Part I; optional to implement	
SCL/SDA filtering; Data Hold Time after SCL low-to-high transition	Suppress noise spikes less than 50ns; no Data Hold Time	No filtering, but Data Hold Time of 300ns helps obtain noise immunity		
Nominal Vdd	No real limits	2.7V, min. - 5.5V, max.		
V_{IH} (SCL/SDA)	70% of Vdd, min. (or legacy 3V, fixed)	2.1V, min.		
V_{IL} (SCL/SDA)	30% of Vdd, max. (or legacy 1.5V, fixed)	0.8V, max.		
V_{OL} (SCL/SDA)	0.4V max., sinking 3mA max. (Vdd>2V)	0.4V, max., sinking 350uA, max.		

20. **PMBUS PITFALL: WHY DOES PMBUS HAVE TWO SPECS.?** The PMBus spec. has two documents, Part I ([7]) and Part II ([8]), whose revision numbers can be independently rolled to maximize confusion—but at the time of writing this document, both specs. are at Rev. 1.1. [7] describes the new group command and extended command protocols, and minor deviations from the SMBus spec. [8] details the register map of a PMBus-compliant power supply product, including register names, addresses, and descriptions of what the bits within the data mean. Appendix I at the end of [8] summarizes register names and meanings, byte lengths, and their protocols.
21. **ONE MORE PITFALL:** An unpowered master or slave IC is not allowed to load down the SCL or SDA lines. The implication for the IC designer is: be careful how you implement ESD protection on your SCL and SDA pins: a basic reverse-biased diode from VDD to SCL/SDA is not acceptable! See Figure 2-2 in SMBus Spec. ([6]) for more info.