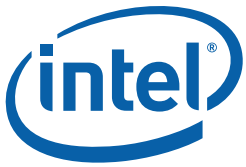


White Paper

Sam Fleming

Technical Marketing Engineer

Intel Corporation

Interfacing I²C* Devices to an Intel® SMBus Controller

January 2009

321089

Executive Summary

Intel includes one or more SMBus controllers as part of their chipset devices. Its primary purpose is to permit the chipset to communicate with SMBus devices such as the SPD EEPROMs on the DIMMs, the clock driver, and various temperature sensors. Some designs incorporate I²C* slave devices on the SMBus. Being "similar" to the I²C bus, it is often difficult to program the Intel® SMBus controller to reliably communicate with these I²C slave devices.

The purpose of this paper is to provide details on the various SMBus cycles that the Intel® SMBus controller can create, and then to provide guidelines on how to "analyze" the cycles supported by I²C devices to see if they can be successfully accessed by the Intel SMBus controller.

It is often difficult to program the Intel® SMBus controller to reliably communicate with I²C slave devices.

It is important to realize that sometimes it is not possible to program the SMBus controller's basic cycles to communicate with a given I²C device. The judicious use of the various blocks modes when combined with the setting of an "I²C Enable" bit (I2C_EN) (which changes the format of the some of the cycles slightly) often permits the Intel® SMBus controller to communicate successfully with I²C devices.

Contents

Technical Challenge

I²C * is a common interface on a wide variety of components used in IA32 designs. The Intel® SMBus controller can work with many of these devices in addition to being able to communicate with native SMBus devices.

Certain I²C protocols are different from SMBus protocols. It is often possible to get these I²C devices to work with the SMBus controller via the judicious use of specific transfer types and/or register settings in the Intel chipset.

Some of the register settings can be misleading. The Intel SMBus controller contains a setting to enable I²C mode (I2C_EN in D31:F3-0x40[2]). This is frequently confused with a specific I²C command called " I2C Read". In either case, neither of these settings put the SMBus controller into a 100% I²C mode.

This paper will discuss exactly what these two settings accomplish and show how software can leverage these two features to enable communication with several I2C devices.

Analysis of the Various Bus Cycles

This section provides details and diagrams on the various cycles that can be created by the SMBus controller. Additionally, any differences in these cycles possible with changes in configuration registers will be clearly shown.

I²C * cycles are a bit different in that the I²C specification only defines three basic cycle types: Write, Read, and Combined format cycle type. Being fairly generic, each cycle type has the ability to transfer more than one byte of data (block mode). Although this section will provide details on these three I²C cycle types, we'll see in the examples below that it's CRITICAL to analyze the EXACT cycle desired in detail. Each I²C device can configure the basic I²C "Generic" waveforms in some interesting ways.

I²C* Cycles

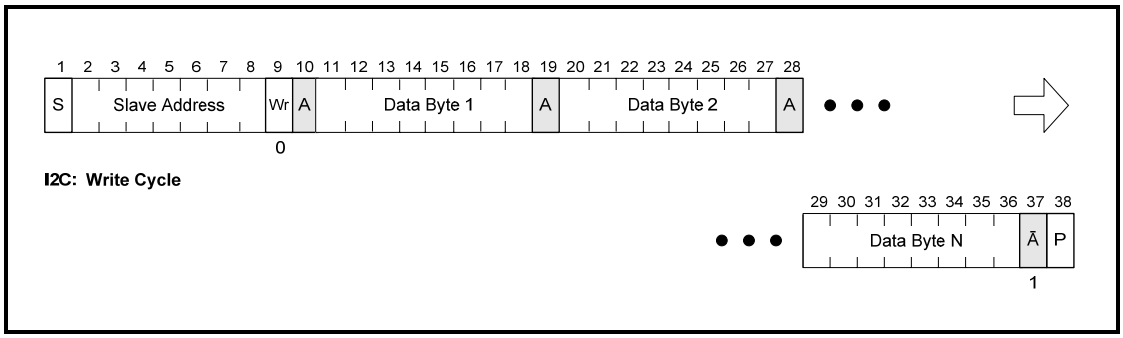
The I²C bus defines three basic cycle types:

- Write
- Read
- Combined Format

These cycles are fairly generic. As can be seen in the waveforms below, each cycle type is capable of a block mode form, capable of sending multiple bytes of data per transfer.

I²C* Write Cycle Type

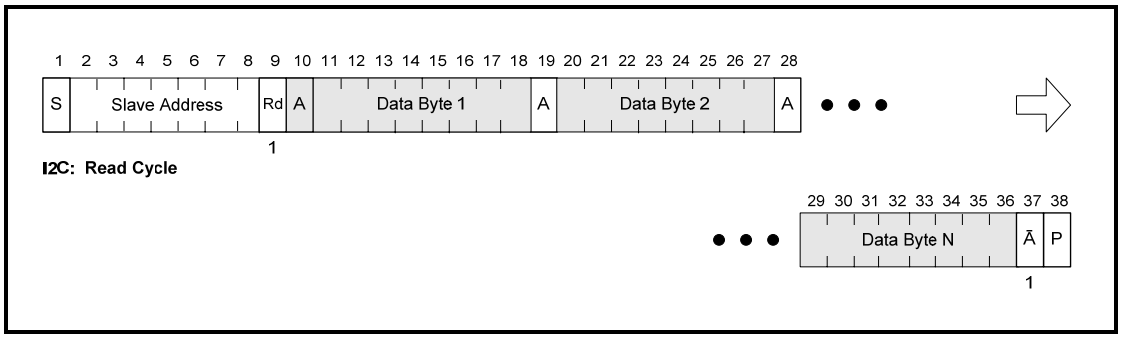
Figure 1. I²C* Write Cycle Type Diagram



With this transfer type, the I²C * master can send 1-N bytes of data to the I²C slave. Note that it's the slave device that determines the number of bytes to transfer by asserting the NAK at bit 37. This differs from the SMBus block mode write command in which the master determines the block write transfer size.

I²C Read Cycle Type

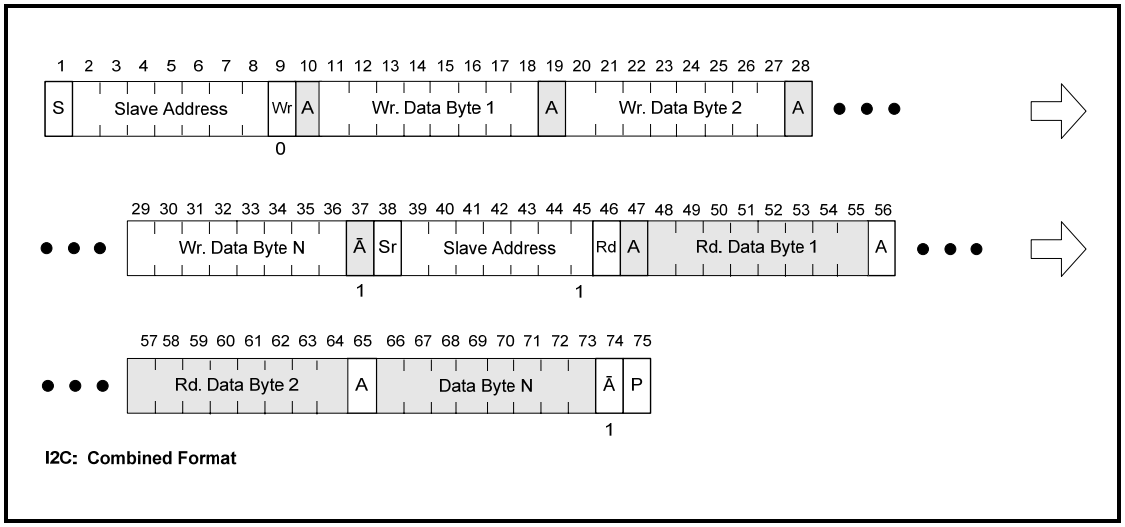
Figure 2. I²C Read Cycle Type Diagram



With this transfer type, the I²C master can read 1-N bytes of data from the I²C slave. Note that it's the master device that determines the number of bytes to read by asserting the NAK at bit 37.

I²C* Combined Format Cycle Type

Figure 3. I²C* Combined Cycle Type Diagram



The I²C* Master uses this transfer to type to do a combined W-R cycle to the slave. This cycle type can also be used to do an R-W cycle, (or even an R-R or W-W). The "Write followed by Read" sequence is by far the most common implementation of this cycle type. It's typically used to send a command to a device and then read data based on the command sent. The W-R Combined Format Cycle is also the cycle that is typically used to read from I²C EEPROMs. The initial write tells the EEPROM of the specific offset to be read from in the subsequent read cycle.

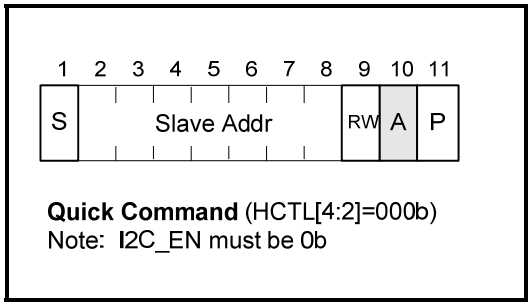
SMBus Cycles

The SMBus controller in Intel chipsets can generate several different types of SMBus cycles. Additionally, several of these cycles change when the I2C_EN bit is set. Each of these cycles will be shown in detail below.

The HCTL[4:2] bits control the exact type of cycle to be sent out on the SMBus. It is included here not only as an aid to the programmer, but also to clearly indicate that the I²C Read Command, HCTL[4:2]=110b, has nothing to do with the setting of the I2C_EN bit. This detail often confuses customers trying to determine how to create a specific cycle.

SMBus Quick Command Cycle Type

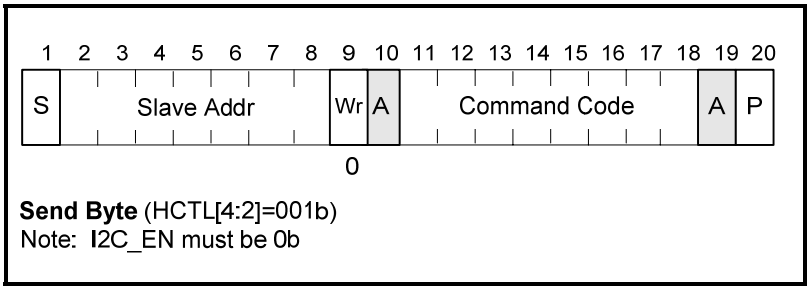
Figure 4. SMBus Quick Command Cycle Type Diagram



A short cycle that exchanges no data with the slave.

SMBus Send Byte Cycle Type

Figure 5. SMBus Send Byte Cycle Type Diagram



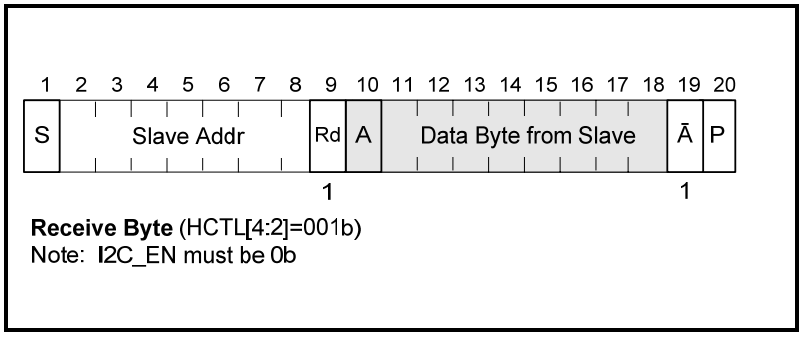
An SMBus command used to write a byte of data to a slave. It differs from the Write Data Byte command in that the SMBus controller only sends the 8 bit Command Code to the slave.

The "Command Code" is an eight bit register in the SMBus controller. The contents of this register are sent during some SMBus commands. In the figures, this byte is shown as the Command Code.

It's important to realize that Command Code is nothing more than 8 bits of data sent to the slave during SMBus cycles at specific points during the cycle. It's called Command Code because this byte was designed to send a specific command to the slave device. As an example, the command code could be used during an SMBus Send Byte cycle to put a device in a specific mode dependent upon the data sent in the Command Code. Modes such as "Wake Up" and "Go to Sleep" are common.

SMBus Receive Byte Cycle Type

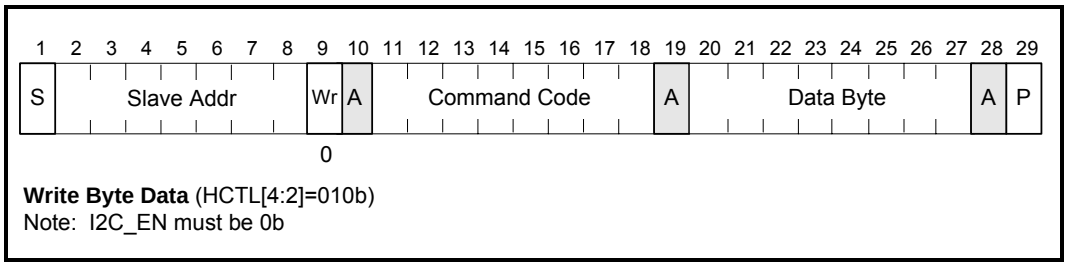
Figure 6. SMBus Receive Byte Cycle Type Diagram



This SMBus command is used to read a byte of data from a slave. It differs from the Read Data Byte command in that the SMBus controller does not send the Command Code data to the slave.

SMBus Write Byte Data Cycle Type

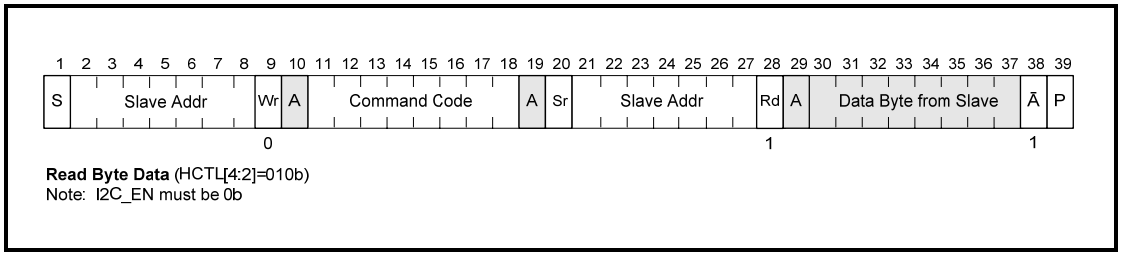
Figure 7. SMBus Write Byte Data Cycle Type Diagram



This command sends the command code and one other data byte to a slave device. This command differs from the Send Byte Command in that the command code byte is sent.

SMBus Read Byte Data Cycle Type

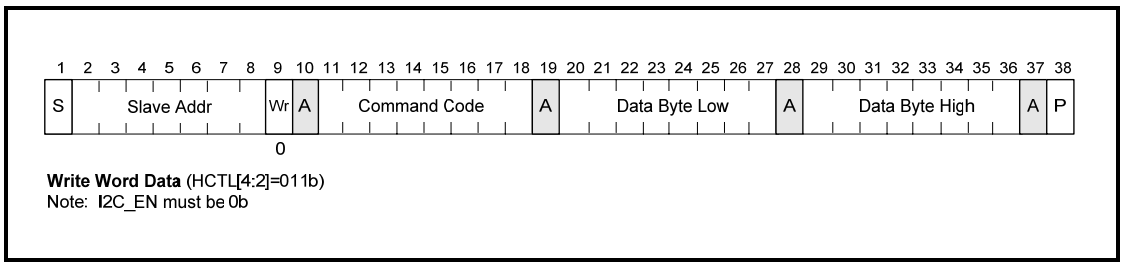
Figure 8. SMBus Read Byte Data Cycle Type Diagram



This command read eight bits of data from a slave. Note that the Command Code data is sent to the slave prior to reading the eight bits of data from the slave. This is how the cycle differs from the Receive Byte cycle type.

SMBus Write Word Data Cycle Type

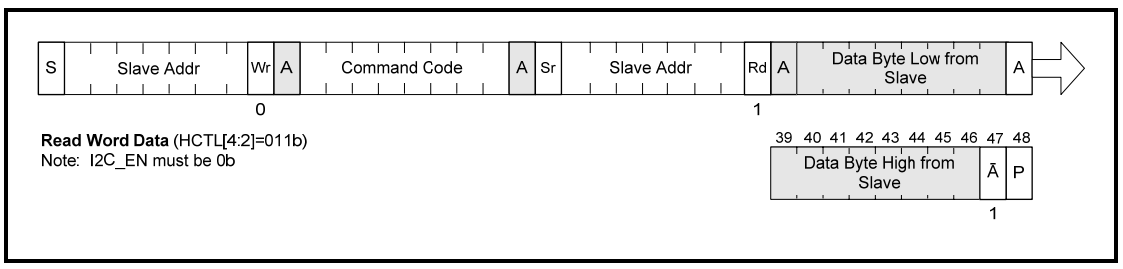
Figure 9. SMBus Write Word Data Cycle Type Diagram



This command writes two bytes of data from the slave. The Command Code data is sent to the slave prior to the data reads. Many I²C* devices can use this command if they need to receive three bytes of data.

SMBus Read Word Data Cycle Type

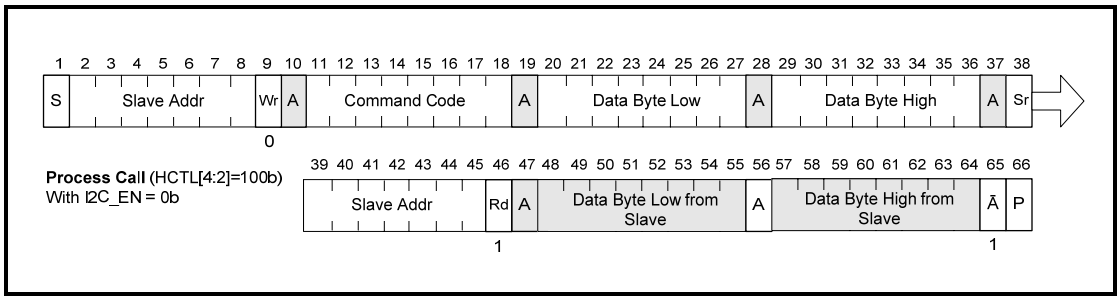
Figure 10. SMBus Read Word Data Cycle Type Diagram



This command reads two bytes of data from the slave. The Command Code data is sent to the slave prior to the data reads.

SMBus Process Call Cycle Type (with I2C_EN=0b)

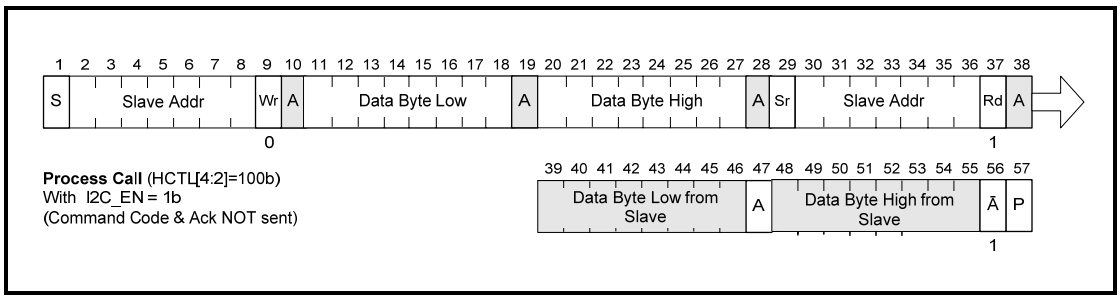
Figure 11. SMBus Process Call Cycle Type Diagram (with I2C_EN=0b)



This is a two part cycle. After the Command Code byte is sent to the slave, two other bytes are sent. A repeat start is performed on the bus by the master and then two bytes of data are read from the slave.

SMBus Process Call Cycle Type (with I2C_EN=1b)

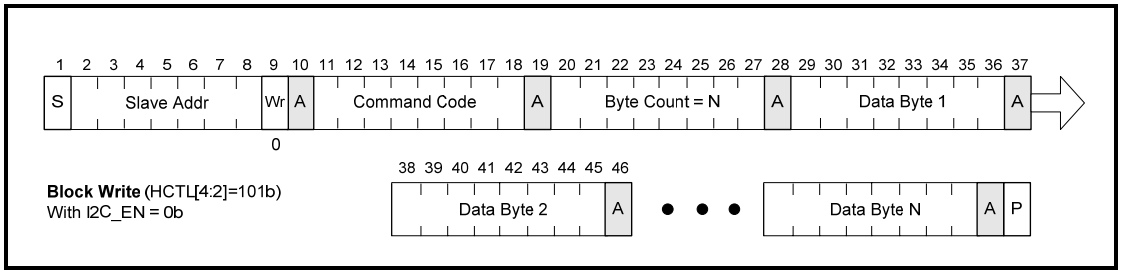
Figure 12. SMBus Process Call Cycle Type Diagram (with I2C_EN=1b)



The difference in this cycle type is that when the I2C_EN bit is set to a 1b, the Command Code is not sent. This cycle type is commonly used for I²C* devices that support the I²C Combined Format cycle with two bytes of read and write data.

Block Write Cycle Type (with I2C_EN=0b)

Figure 13. SMBus Block Write Cycle Type Diagram (with I2C_EN=0b)



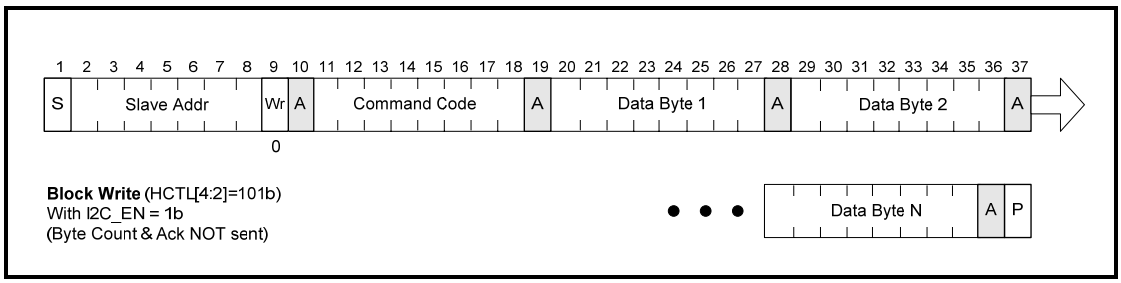
Unlike the I²C bus, during SMBus Block Write Cycles it's the master that determines the size of the data transfer. When programming the SMBus controller to perform an SMBus block command, the programmer sets the size of the data transfer, and then the SMBus controller only sends that many bytes of data during the transfer. What makes this so undesirable for I²C slaves is the Byte Count data sent during bits 20-27 of the cycle. This byte will appear to the I²C slave device as the first write data byte in the stream.

The sending of the Command Code during bits 11-18 prevents this cycle from working with I²C devices.

There is also the issue of the ACK/NACK sent during the last byte transferred. The SMBus controller expects a normal ACK from the slave. An I²C slave, however, always sends a NACK after the last byte is received.

Block Write Cycle Type (with I2C_EN=1b)

Figure 14. SMBus Block Write Cycle Type Diagram (with I2C_EN=1b)



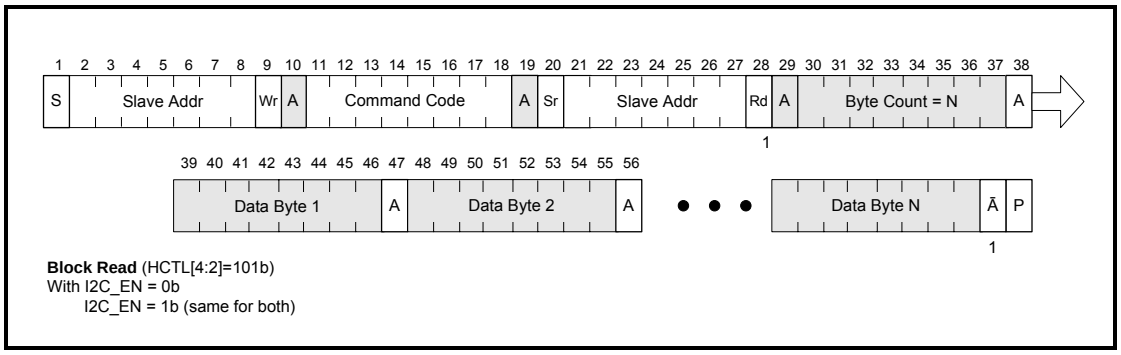
The difference when I2C_EN=1b is that the Byte Count field is not sent. This is much more acceptable for some I²C* slave devices.

Also realize that the Command Code sent during bits 11-18 is generally not a problem, as the I²C device expects it to be "Byte #1" in the data stream.

The programmer of the SMBus controller must ensure that Byte #1 is programmed into the Command Code Register instead of the data registers.

Block Read Cycle Type

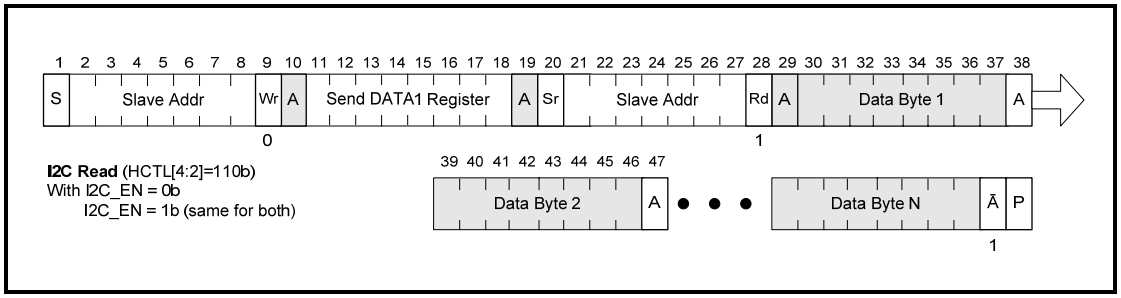
Figure 15. SMBus Block Read Cycle Type Diagram



This cycle is rarely used to communicate with I²C devices. In addition to the command code sent during bits 11-18, the master also expects that the first byte returned will be the number of bytes to be received. This cycle would only work if the first byte sent by the I²C slave device also just happened to be the size of the return data stream. This condition is highly unlikely.

I²C* Read Cycle Type

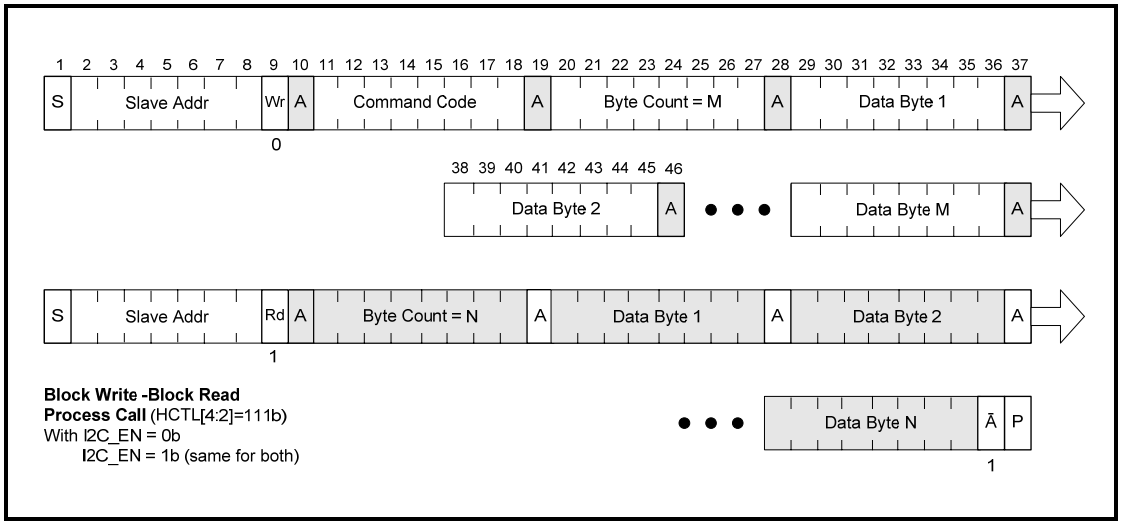
Figure 16. SMBus I²C* Read Cycle Type Diagram



The purpose of this cycle is to communicate with I²C* slave devices that implement the I²C Combined Format Cycle type (with a single byte being written before the read). This is the standard cycle type used to access I²C EEPROMs such as the SPD EEPROMs found on DIMMs.

Block Write-Block Read Process Call Cycle Type

Figure 17. SMBus Block Write-Block Read Process Call Type Diagram



This cycle type performs a block write command followed by a block read command. It doesn't work well with I²C devices because the SMBus controller always sends the Command Code and the Byte Count fields (and expects the Byte Count in return from the slave).

Key Differences between SMBus and I²C* Bus Protocols

Since SMBus cycles don't correspond 1:1 with I²C* bus cycles, it's critical to understand the key differences in the two cycles when trying to create an SMBus cycle that will work with I²C devices.

1. During burst cycle types, who controls the number of bytes transferred differs significantly from I²C to SMBus. During I²C Write cycles, the slave determines the transfer size. The slave asserts a NAK after the last byte is returned. With SMBus block write cycles, the byte count is not only controlled by the master, but is actually SENT out on the SMBus as part of the cycle.

   Likewise, for I²C burst Read cycles, the master controls the transfer size. After it receives the last byte, the master asserts a NACK followed by the STOP bit, ending the cycle. During SMBus Burst Read Cycles, however, it's the slave that determines the transfer size.

When it returns the last byte in the steam, the slave asserts NACK, which master promptly follows with STOP, ending the cycle

2. For most of the SMBus cycles, the Command Code is sent during the cycle. For I²C slave devices, this byte will appear as a write data byte.

   For some cycle types, this byte can often be omitted by setting the I2C_EN bit=1b.
3. I2C_EN is an interesting bit. It can be set during many of the SMBus cycles in order to change parts of the transfer:

   A. The Process Call command will skip sending the command code.
   B. The Block Write command will skip sending the byte count.

   Do not get this bit setting confused with the I2C Read Command, which is an actually cycle type. I2C_EN modifies other cycles as shown above.
4. The SMBus controller can only transfer 32 bytes during block read and write commands. The I²C bus puts no limit on transfer sizes.

Procedures for Interfacing I²C* Devices to an Intel® SMBus Controller

There is no secret formula to determine if an I²C* device will work properly with an Intel SMBus controller. For each I²C device that a designer wants to use, the designer must review the device's datasheet carefully analyzing which I²C cycles are needed to use the part successfully.

Once these cycles are identified, it is then a matter of perusing the SMBus waveforms (shown in Figure 4 through Figure 17) and determining if the desired cycle can or cannot be created by the SMBus controller. Be careful to note the key differences noted in the section above. Although an I²C does not know what a Command Code is and is not expecting it, if it is expecting a byte of data at that point in the I²C cycle, the programmer needs to ensure that the Command Code field is programmed with the appropriate data.

Also, whether it is poll-driven or interrupt driven, it is also important to note that the SMBus controller asserts an interrupt/sets the INTR bit after every byte is transferred during one of the block mode transfers. This has important ramifications for the programmer. When the next-to-last byte is transferred during in I²C read command cycle type, the programmer sets the LAST_BYTE bit (HCTL[05]). This lets the SMBus controller assert a NACK after the last byte is transferred. This bit can also be used during the Block Read/Write commands to end a burst cycle prematurely. This is sometimes necessary to support certain types of cycles required by I²C device.

The following examples demonstrate how this process works.

Example 1: 24LC01B-1kbit I²C* EEPROM

The Microchip* 1 Kbit (128 byte) I²C* EEPROM is a standard storage device found attached to the SMBus in IA32 designs. It is commonly used in DIMMs as the SPD EEPROM.

This device has internal "address pointer" that points to the next byte to be accessed during a read. This pointer is set to zero during power up. It is incremented after every read.

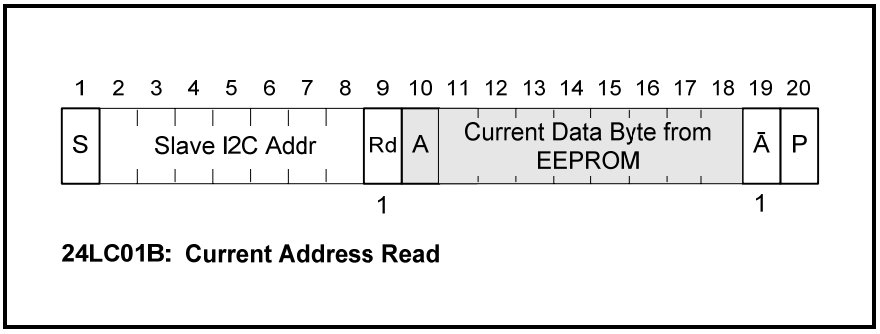
This device supports the following I²C cycles:

- Current Address Read
- Random Read
- Sequential Read

The following sections will take each cycle and compare it to the possible SMBus cycles shown above to see if a "match" can be found.

24LC01B-Current Address Read

Figure 18. 24LC01B (128 Byte I²C* EEPROM) Current Address Read Diagram

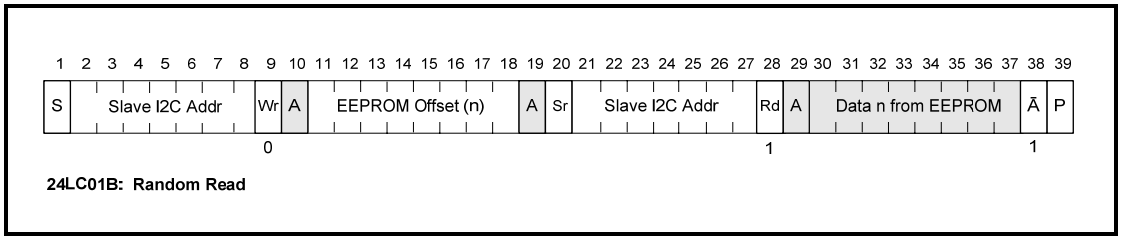


The cycle will read the next byte of data out of the array that is pointed to by the internal address pointer.

Reviewing the SMBus cycles discussed previously on pages 6 - 13, a quick match is found when using the SMBus Receive Byte Command.

24LC01B- Random Read

Figure 19. 24LC01B (128 Byte I²C* EEPROM) Random Read Diagram

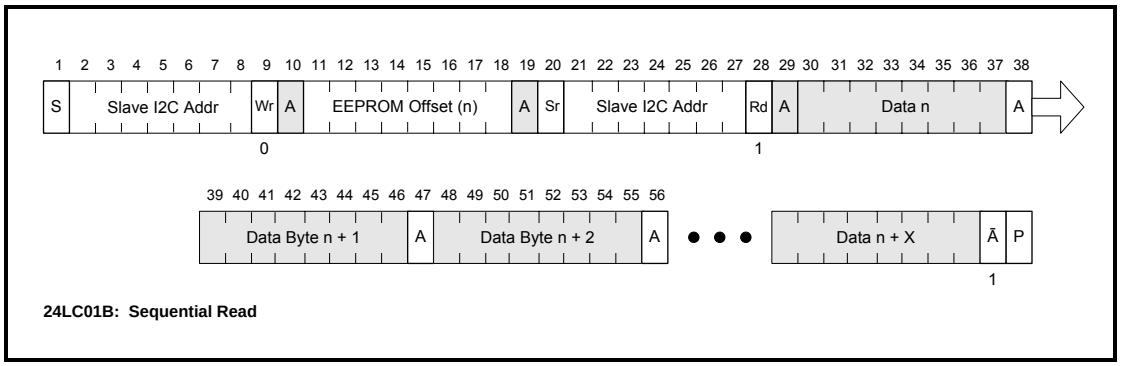


This is probably the most useful of the I²C* Bus commands. This command resets the internal address pointer to the value written during the bits 11-18, and then reads the data from that array member during bits 30-37.

Reviewing the SMBus cycles discussed previously on pages 6 - 13, a quick match is found when using the SMBus Read Byte Data Command. Note, however, that the programmer must ensure that the EEPROM array pointer value is loaded in the Command Code register prior to initiating the SMBus command.

24LC01B- Sequential Read

Figure 20. 24LC01B (128 Byte I²C EEPROM) Sequential Read Diagram



This cycle is the "block read" version of the Random Read. Following the write of the internal EEPROM array pointer during bits 11-18 of the cycle, a block of data is then sequentially read starting at that address. Note that the 24LC01B device is also expecting the master to terminate the cycle following a NACK on the last data byte transfer (Data n+X).

The I²C Read Command matches this cycle. The Data1 register, however, must be used to indicate the array offset value to be written during bits 11-18. The programmer must monitor the data bytes read, however, and set

the LAST_BYTE bit on the next to last transfer to ensure that the SMBus controller asserts the NACK after the last data transfer.

Example 2: M24512-256 Kbit I²C* EEPROM

The 24LC01B was easy in that it was a small EEPROM – only 128 bytes in size. The internal address pointer only had to be 7 bits in size (8 bits for the 256 byte 24LC02B).

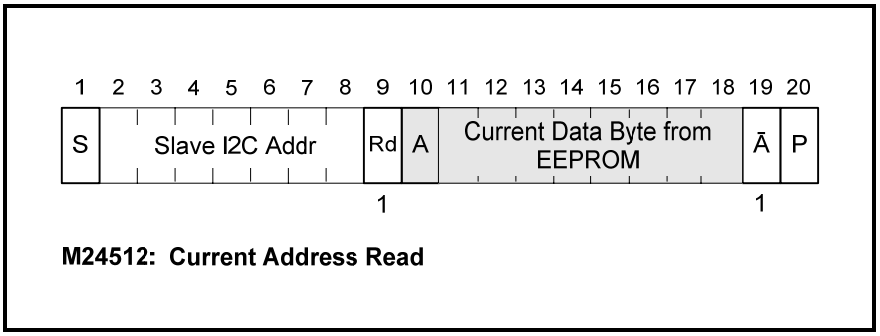
The M24512 is very similar to the 24LC01B, except that it supports more data. It contains 64 Kbytes of data, which will require up to 16 bits of addressing to access its array of data. This changes the I²C* cycles significantly.

The M24512 supports the following cycles:

- Current Address Read
- Random Address Read
- Sequential Current Read
- Sequential Random Read

M24512 – Current Address Read

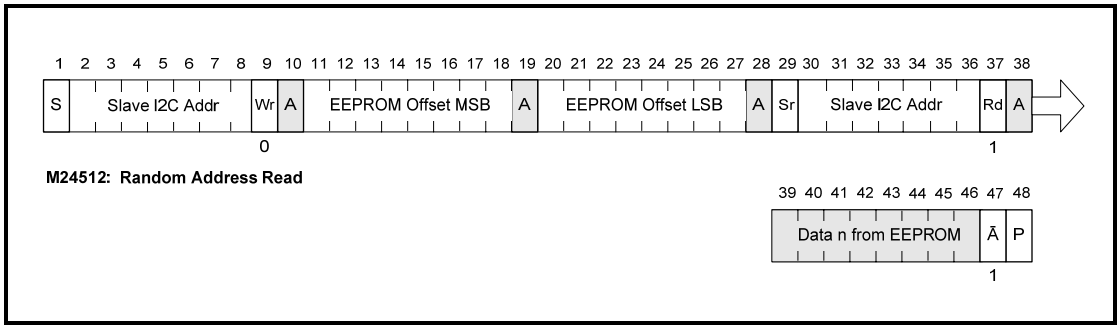
Figure 21. M24512 (64KByte I²C EEPROM) Current Address Read Diagram



This cycle reads in the next byte from the array pointed to by the internal array pointer. It is identical to the cycle for the 24LC01B, and as such, the SMBus Receive Byte Command can be used to execute this command on the M24512.

M24512 - Random Address Read

Figure 22. M24512 (64KByte I²C* EEPROM) Random Address Read Diagram



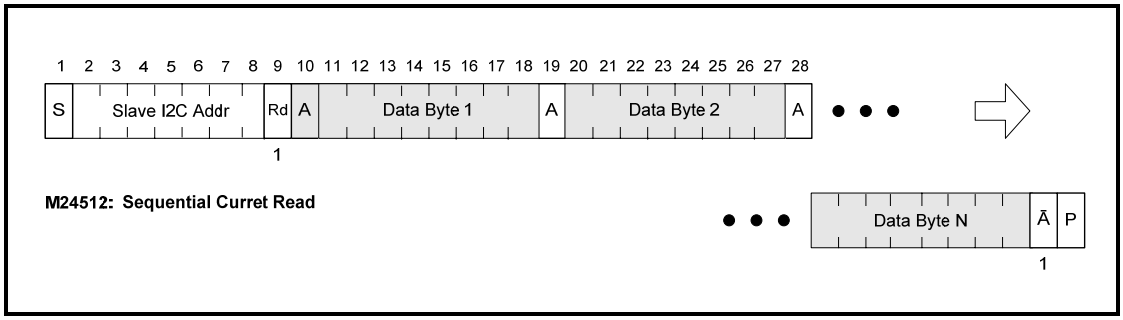
This cycle sets the internal array pointer with a word write preceding the data read. Reviewing the possible SMBus cycles does not reveal a match for this cycle. For the 24LC01B it was possible to use the written Command Code in lieu of the EEPROM address offset. With the M24512, however, this cannot be done. There is no SMBus cycle with a write of two bytes followed by a read of a single byte. The Process Call command (with I2C_EN=1b) comes very close, but it expects two bytes to be returned by the I²C* slave. Since this command is not a block mode type of command (interrupt / INTR is not asserted after every byte is transferred), the programmer cannot prematurely end the cycle after a single byte has been transferred. Using the this Process Call command will actually hang the SMBus since the SMBus controller will be stuck indefinitely waiting for the second byte of data to be returned by the M24512 I²C slave device.

This cycle cannot be recreated by the SMBus controller. This does not preclude a designer from using this part, however, since the Sequential Random Read Command of the M25512 can be replicated using the SMBus controller and this command can be used to access all of the data in the M24512.

It is important to realize that just because one cycle of an I²C device cannot be recreated, it is still often possible to use the I²C device since all of its desired functionality can still be used using other commands that can be supported.

M24512 – Sequential Current Read

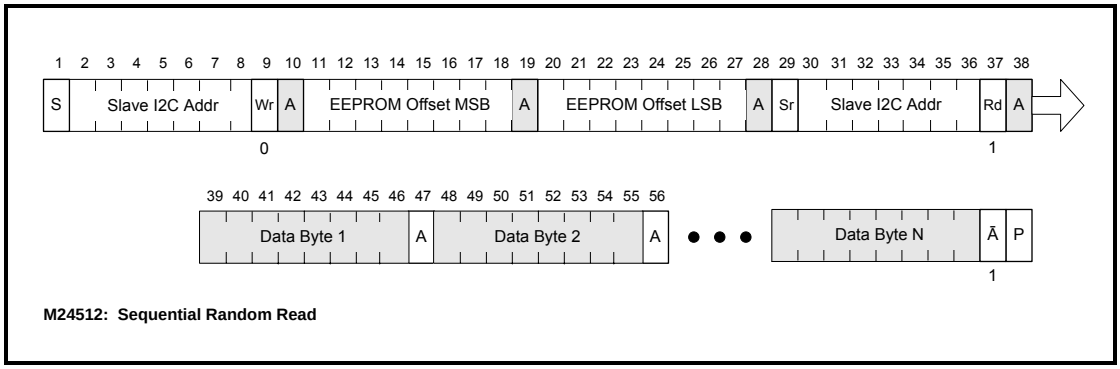
Figure 23. M24512 (64KByte I²C* EEPROM) Sequential Current Read Diagram



This is the basic block read command reading data sequentially from the current address pointer. This cycle also cannot be recreated using the SMBus controller, however, the Sequential Random Read cycle below can be recreated by the SMBus controller and this cycle can be used to access all of the data in the M24512 array.

M24512 – Sequential Random Read

Figure 24. M24512 (64KByte I²C EEPROM) Sequential Random Read Diagram



This command can be supported by using the SMBus Process Call command with I2C_EN=1b. The two bytes comprising the address pointer will be written, and two bytes of data will be returned by the slave before the master asserts the NAK during bit 56 (see Figure 12) before ending the cycle with STOP in bit 57.

A programmer can easily read the M24512 with this command, note that it will have to be done in two byte chunks. It might be faster to use this command to read the first two bytes of the array, and then use the Current

Address Read (SMBus Receive Byte) to read the rest of the array. Testing would have to be done to calculate which method is faster.

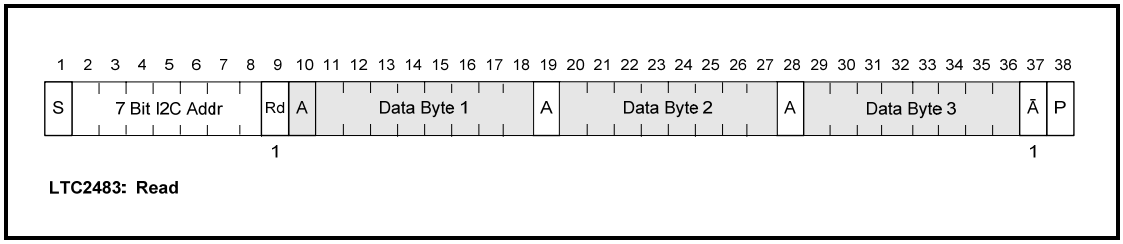
This is not usually much of an issue since the SMBus is not designed to be a high speed bus anyway. Speed is usually not much of a concern.

Example 3: LTC2483 16-Bit Analog-to-Digital Converter

This device is a 16 bit Analog-to-Digital converter (ADC) by Linear Technology*. This device supports a single I²C* command that reads a 24-bit value from the converter (which then fires off another conversion).

The cycle used to read the 24 bits of data from the LTC2483 looks like this:

Figure 25. LTC2483 (16 bit ADC) Read Diagram



The SMBus controller can generate a myriad of byte and word cycles, but since this cycle requires a read of three bytes, it will have to be created using one of the SMBus block mode type commands, and unfortunately, nothing fits. Command Code data and Byte Count data ruins any possible match.

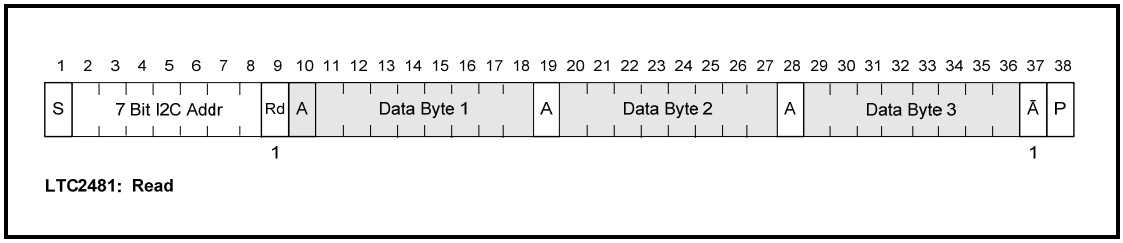
Since there are no other commands supported by the LTC2483, this device cannot be used with an Intel® SMBus controller.

Example 4: LTC2481 16-Bit ADC

This device is identical to the LTC2483, but it contains the ability to be configured in different configurations. Because of this, it supports two additional cycles which include write data.

LTC2481 –Read Cycle

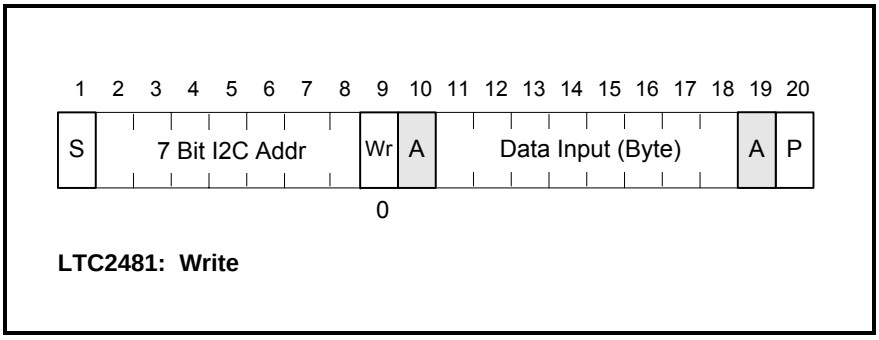
Figure 26. LTC2481 (16 bit ADC) Read Diagram



This is identical to the LTC2483's Read cycle. This cycle cannot be recreated by the SMBus controller.

LTC2481 –Write Cycle

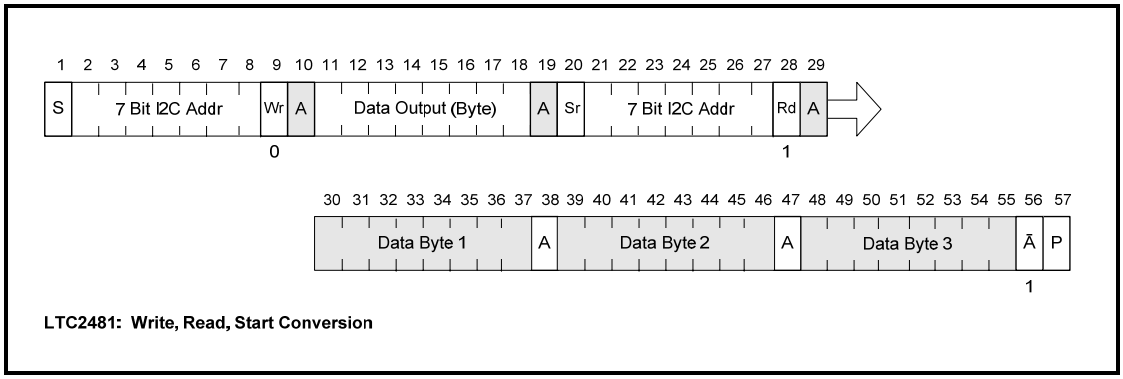
Figure 27. LTC2481 (16 bit ADC) Write Diagram



This is a write of eight bits of data to the LTC2481 that is used to configure the device. This cycle can easily be supported with the SMBus Send Byte command. Since the primary function of the device is to provide data that must be read, supporting this cycle does not permit a designer to use this device, however. A cycle that reads the ADC data that is supported by the SMBus controller is still needed.

LTC2481 –Write, Read, Start Conversion Cycle

Figure 28. LTC2481 (16 bit ADC) Write, Read, Start Conversion Diagram



This is the LTC2481's "all in one cycle". The first part of the I²C* device sets the configuration for the device and then the three bytes of data are read from the device.

A perusal of the supported SMBus cycles does result in a match. The SMBus controller can recreate this cycle by using the I²C Read Command. The configuration data to be output to the LTC2481 is loaded by the programmer into the Data1 register. The programmer must also ensure to set the LAST_BYTE bit after the second byte of data has been read from the LTC2481. This will cause the SMBus controller to assert NACK and the STOP bits to end the cycle.

Note that the data read is for the PREVIOUS configuration of the device. The programmer must be careful to load the configuration data with data to be read AFTER the next conversion. Also note that even though the LTC2483 ADC could not be used, the same functionality COULD be gained by using the LTC2481.

Conclusion

Although closely related, the SMBus controller in Intel® chipsets does not natively support all I²C cycles needed by many I²C peripherals. However, by fully understanding all of the cycle types support by the SMBus controller, and by understanding all of the nuances of block mode and the I2C_EN bit, it is often possible to successfully connect I²C devices to the Intel® SMBus controller.

References

- The I²C* Bus Specification Version 2.1 - January 2000. document order number: 9398 393 40011
  http://www.nxp.com/acrobat_download/literature/9398/39340011.pdf

- System Management Bus (SMBus) Specification Version 2.0 - August 3, 2000
  http://smbus.org/specs/

- 24LC01B/02B Module (Common I²C EEPROM used for SPD on DIMMS)
  http://datasheet.digchip.com/295/295-3-002647-24LC01B-MT.pdf

- 512Kbit and 256Kbit I²C EEPROMs
  http://www.st.com/stonline/products/literature/ds/6757.pdf

- Linear LTC2483 Datasheet
  http://www.linear.com/pc/downloadDocument.do?navId=LTC2483,D9658

- Linear LTC2481 Datasheet (16-Bit ADC)
  http://www.linear.com/pc/downloadDocument.do?navId=LTC2481,D9657

Authors

Sam Fleming is a Technical Marketing Engineer with EDC at Intel Corporation.

Acronyms

SMB: System Management Bus. This is a two-pin bus used to connect various slow speed components. It is based on the principles of operation of the I²C bus.

I²C: Intelligent Interface Controller. This two-pin bus was developed by Philips*. Various I²C devices can be connected to this slow speed bus.

SPD: Serial Presence Detect. This is a small EEPROM found on most DIMMS and SO-DIMMS. It contains information on the type and speed of the DIMM. It's read by BIOS during IA32 boot-up in order to properly configure the memory array.

EEPROM: Electrically Erasable Programmable Read-Only Memory

ADC: Analog-to-Digital Converter