Application Note

A Basic Guide to I²C

Joseph Wu

ABSTRACT

Communication between microcontrollers and different peripheral devices require some sort of digital protocol. I²C is a common communication protocol that is used in a variety of devices from many different product families produced by TI. This application note begins with a basic overview of the I²C protocol, describing the history of the protocol, different I²C speed modes, the physical layer of the digital communication, and the structure of the data. Several examples of the communication protocol are shown with different data converters. Finally, this application note covers some uncommon aspects of the protocol, including reserved addresses, clock arbitration and stretching, electrical timing and voltage specifications, and pullup resistor calculation.

Table of Contents

List of Figures

List of Tables

Trademarks

SMBus® is a registered trademark of System Management Interface Forum, Inc..
PMBus® is a registered trademark of Intel.
All trademarks are the property of their respective owners.

1 I²C Overview

I²C is a two-wire serial communication protocol using a serial data line (SDA) and a serial clock line (SCL). The protocol supports multiple target devices on a communication bus and can also support multiple controllers that send and receive commands and data. Communication is sent in byte packets with a unique address for each target device.

1.1 History

I²C, often called I 'two' C, stands for the Inter-Integrated Circuit protocol. I²C was developed in 1982 by Philips Semiconductor (now NXP Semiconductor) as a low-speed communication protocol for connecting *controller* devices such as microcontrollers and processors with *target* devices such as data converters and other peripheral devices. Since 2006, implementing the I²C protocol does not require a license, and many semiconductor device companies, including TI, have introduced I²C-compatible devices.

I²C is a widely-used protocol for many reasons. The protocol requires only two lines for communications. Like other serial communication protocols, there is a serial data line and a serial clock line. I²C can connect to multiple devices on the bus with only the two lines. The controller device can communicate with any target device through a unique I²C address sent through the serial data line. I²C is simple and economical for device manufacturers to implement.

1.2 I²C Speed Modes

I²C has several speed modes starting with the *Standard-mode (Sm)*, which is a serial protocol that operates up to 100 kilobits per second (kbps). This mode is followed by the *Fast-mode (Fm)* which tops out at 400 kilobits per second. Fast-mode can be used by the controller if the bus capacitance and drive capability allow for the faster speed. Both of these protocols are widely supported.

The *Fast-mode Plus (Fm+)* mode allows for communication as high as 1 megabit per second (Mbps). To achieve this speed, drivers in the devices require extra strength to comply with faster rise and fall times.

These three modes are relatively similar, using a communication structure that is the same. However, all have different timing specifications for each of the modes and hardware implementation of the I²C in the devices are different to accommodate the different speeds.

I²C also has two other modes for higher data rates. *High-speed mode (Hs-mode)* has a data rate to 3.4 megabits per second. In this mode, the controller device must first use a controller code to allow for high-speed data transfer. This enables high-speed mode in the target device. This mode can also require an active pullup to drive the communication lines at a higher data rate.

Ultra-Fast mode (UFm) is the fastest mode of operation and transfers data up to 5Mbps. This mode is write-only and omits some I²C features in the communication protocol.

Table 1-1 shows the different I²C modes and their respective data rates

Table 1-1. Maximum Transmission Rates for Different I²C Modes

I²C Mode	Maximum Bit Rate
Standard-mode	100kbps
Fast-mode	400kbps
Fast-mode Plus	1Mbps
High-speed mode	3.4Mbps
Ultra-Fast mode	5Mbps

2 I²C Physical Layer

2.1 Two-Wire Communication

An I²C system features two shared communication lines for all devices on the bus. These two lines are used for bidirectional, half-duplex communication. I²C allows for multiple controllers and multiple target devices. Pullup resistors are required on both of these lines. Figure 2-1 shows a typical implementation of the I²C physical layer.

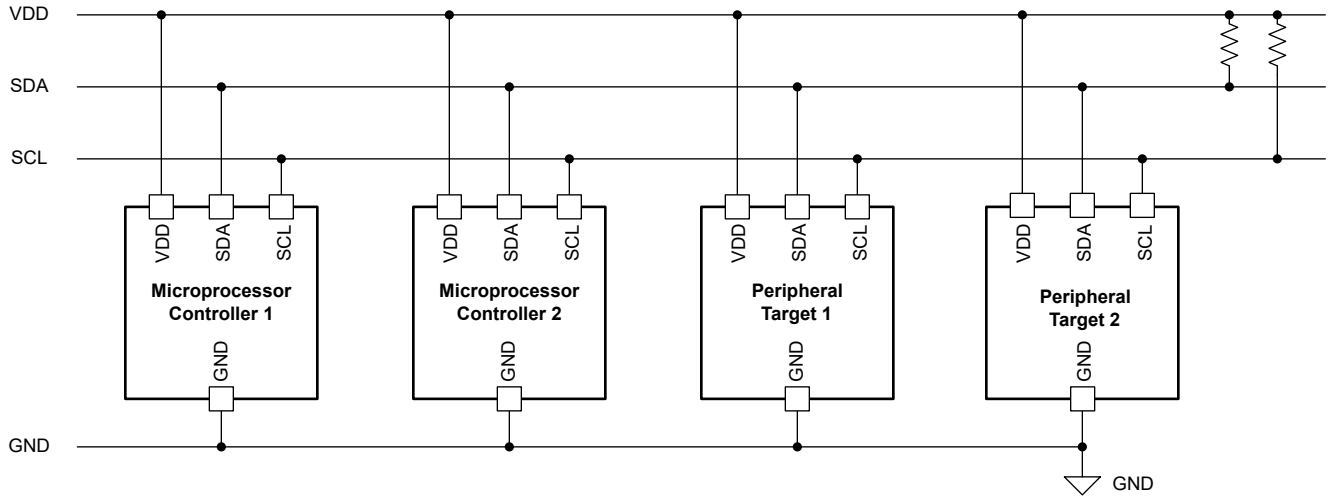

Figure 2-1. Typical I²C Implementation

One of the reasons that I²C is a common protocol is because there are only two lines used for communication. The first line is SCL, which is a serial clock primarily controlled by the controller device. SCL is used to synchronously clock data in or out of the target device. The second line is SDA, which is the serial data line. SDA is used to transmit data to or from target devices. For example, a controller device can send configuration data and output codes to a target digital-to-analog converter (DAC), or a target analog-to-digital converter (ADC) can send conversion data back to the controller device.

I²C is half-duplex communication where only a single controller or a target device is sending data on the bus at a time. In comparison, the serial peripheral interface (SPI) is a full-duplex protocol where data can be sent to and received back at the same time. SPI requires four lines for communication, two data lines are used to send data to and from the target device. In addition to the serial clock, a unique SPI chip select line selects the device for communication and there are two data lines, used for input and output from the target device.

An I²C controller device starts and stops communication, which removes the potential problem of bus contention. Communication with a target device is sent through a unique address on the bus. This allows for both multiple controllers and multiple target devices on the I²C bus.

The SDA and SCL lines have an open-drain connection to all devices on the bus. This requires a pullup resistor to a common voltage supply.

2.2 Open-Drain Connection

The open-drain connections are used on both SDA and SCL lines and connect to an NMOS transistor. This open-drain connection controls the I²C communication line and pulls the line low or releases the line high. The open-drain refers to the NMOS bus connection when the NMOS is turned OFF. Figure 2-2 shows the open-drain connection as the NMOS is turned on.

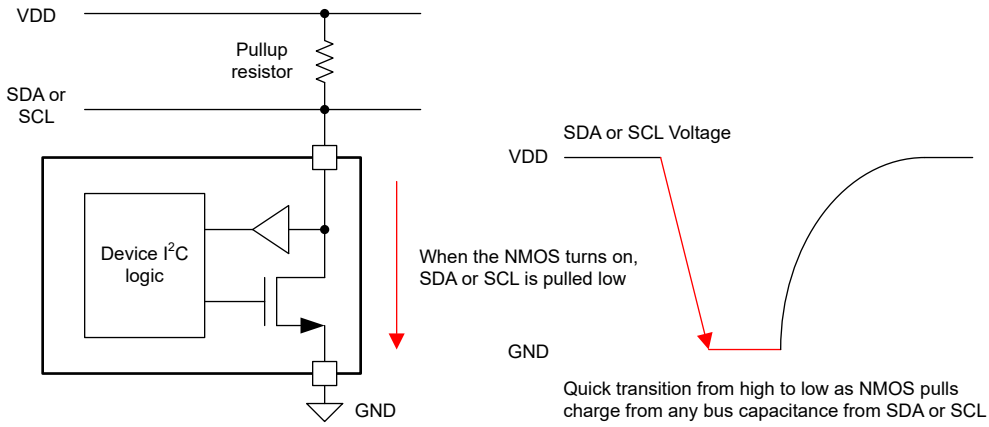

Figure 2-2. Open-Drain Connection Pulls Line Low When NMOS is Turned On

To set the voltage level of the SDA or SCL line, the NMOS is set on or off. When the NMOS is on, the device pulls current through the resistor to ground. This pulls the open-drain line low. Typically, the transition from high to low for I²C is a fast transition as the NMOS pulls down on SDA or SCL. The speed of the transition is determined by the NMOS drive strength and any bus capacitance on SDA or SCL.

When the NMOS turns off, the device stops pulling current, and the pullup resistor pulls the SDA or SCL line to VDD. Figure 2-3 shows an open-drain line as the NMOS is turned off. The pullup resistor pulls the line high. The transition of the open-drain line is slower because line is pulled up against the bus capacitance, and is not actively driven.

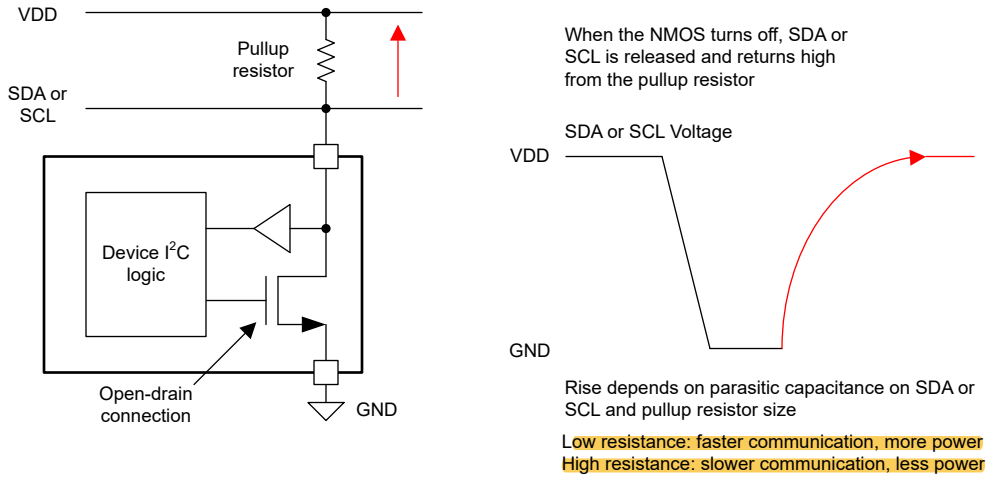

Figure 2-3. Pullup Resistor Pulls Line High When NMOS is Turned Off

Through control of this open-drain connection, both SDA and SCL can be set high and low, enabling the I²C communication.

Because of capacitance on the I²C communication line, the SDA and SCL lines discharge with an exponential settling RC time constant depending on the size of the pullup resistor and capacitance on the I²C bus. Higher capacitance limits the speed of I²C communication, the number of devices, and the physical distance between devices on the bus. A smaller pullup resistor has a faster rise time, but requires more power for communication. A larger pullup resistor has a slower rise time leading to slower communication, but requires less power.

2.3 Non-Destructive Bus Contention

One of the benefits of I²C using an open drain is that bus contention does not put the bus into a destructive state. With an open-drain output, many devices can be connected together without destructive contention. For any output on that connection, if any output pulls the line low, the line is low. This kind of connection is called a *wired-AND* connection. The output is the logical AND of all the outputs when tied together.

If the outputs are a push-pull type, the outputs cannot be tied together without the possibility of a destructive state. A push-pull output (often used for SPI communication) has complementary NMOS and PMOS transistors that drive the output high or low. Figure 2-4 shows a comparison between open-drain and push-pull outputs in contention.

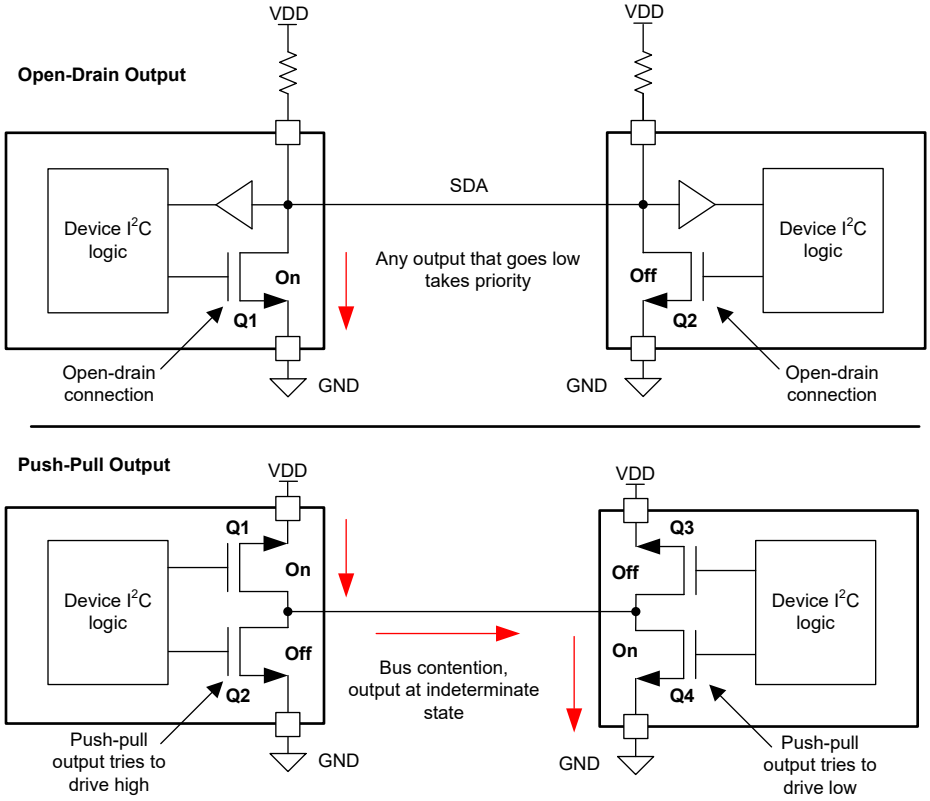

Figure 2-4. Comparison Between Open-Drain and Push-Pull Contention

With the open-drain connection, any device can pull the connection low at any time. The line appears low any time any device pulled the line low, but does not appear as destructive contention.

In the push-pull output, the outputs are also tied together. If two devices are active on the bus and one output is high and another output is low, this bus contention has an undetermined state, possibly settling at the mid-supply point. Additionally, one device has NMOS conducting current and another device has a PMOS conducting current. These devices source current from V_{DD} to GND through a very low impedance path, conducting as much current as the transistors allow. The result of this contention can be a significant amount of current, potentially damaging the devices.

3 I²C Protocol

3.1 I²C START and STOP

I²C communication is initiated from the controller device with an I²C START condition. If the bus is open, an I²C controller claims the bus for communication by sending an I²C START. To do this, the controller device first pulls the SDA low and then pulls the SCL low. This sequence indicates that the controller device is claiming the I²C bus for communication, forcing other controller devices on the bus to hold their communication.

When the controller device has completed communication, the SCL releases high and then the SDA releases high. This indicates an I²C STOP condition. This releases the bus to allow other controllers to communicate or to allow for the same controller to communicate with another device. Figure 3-1 shows the protocol for an I²C START and STOP.

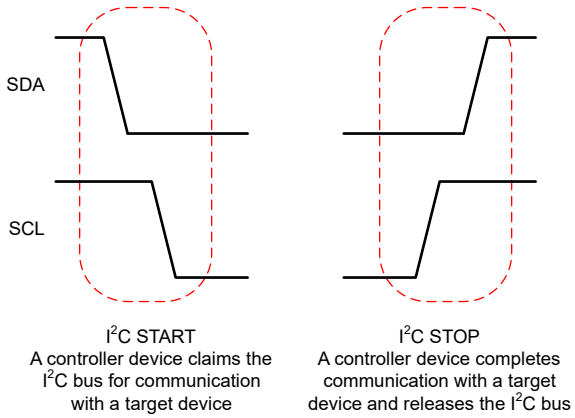

Figure 3-1. I²C START and STOP

3.2 Logical Ones and Zeros

I²C uses a sequence of ones and zeros for serial communication. SDA is used for the data bits while SCL is the serial clock that times the bit sequence. A logical one is sent when the SDA releases the line, allowing the pullup resistor to pull the line to a high level. A logical zero is sent when SDA pulls down on the line, setting a low level near ground. Figure 3-2 shows the representation of a digital one and zero for I²C communication.

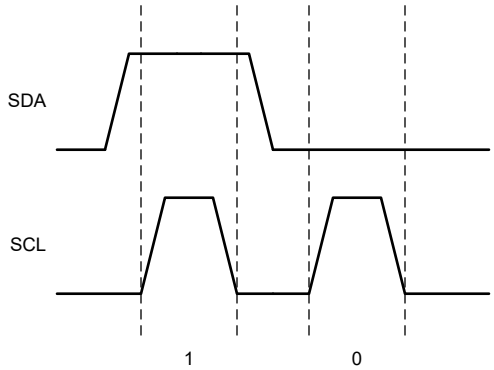

Figure 3-2. I²C Digital One and Zero Representations

The ones and zeros are received when SCL is pulsed. For a valid bit, SDA does not change between a rising edge and the falling edge of SCK for that bit. Changes of the SDA between the rising and falling edges of the SCL can be interpreted as a START or STOP condition on the I²C bus.

3.3 I²C Communication Frames

The I²C protocol is broken up into frames. Communication begins with the controller device sending an address frame after a START. The address frame is followed by one or more data frames each consisting of one byte. Each frame also has an acknowledge bit to alert the controller that the target device or the controller device has received communication. Figure 3-3 shows a diagram of two I²C communication frames.

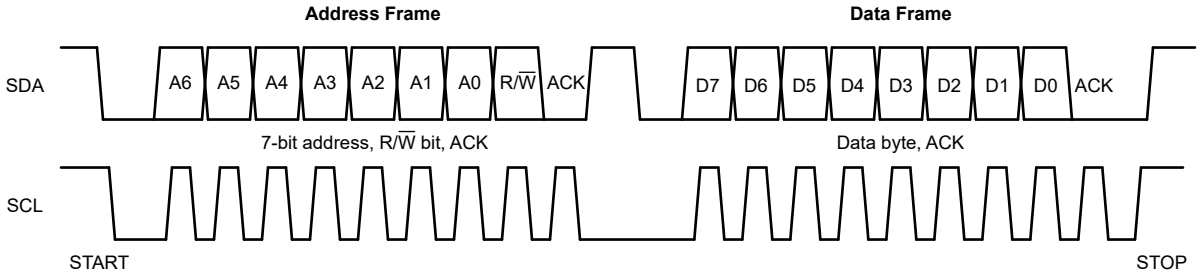

Figure 3-3. I²C Address and Data Frames

At the beginning of the address frame, the controller device initiates a START condition. The controller device first pulls SDA low and then pulls SCL low for the START. This allows the controller device to claim the bus without contention from other controller devices on the bus. Each I²C target device has an associated I²C address. When beginning communications with a particular target device the controller uses the target device address to send or receive data in the following I²C frames. The I²C address consists of 7 bits and devices on the I²C bus, each have a unique address on the bus.

A 7-bit address implies 2^7 (or 128) unique addresses. However, there are several reserved I²C addresses which limits the number of possible devices. Reserved addresses are discussed in Section 5. The address is sent with the SDA as the data and SCL as the serial clock. With this information, you can to read through the I²C communication of a device and understand what is being sent back and forth from the controller device and the target device.

The 8th bit of this frame following the address, is the read-write (R/$\overline{W}$) bit. If this bit is 1, the controller is asking to read data from the target device. If this bit is 0, the controller asks to write data to the target device.

After any communication byte, an extra 9th bit is used to verify the communication was successful. At the end of the address byte communication, the target device pulls down the SDA during the SCL pulse to indicate to the controller that the address was received. This is known as an acknowledge (ACK) bit. If this bit is high, then no target device received the address and the communication was unsuccessful. If the bit is high, this is known as a NACK and there was no ACK.

The address frame is followed by one or more data frames. These frames are sent one byte at a time. After each data byte is transferred, there is another ACK. If the data byte is a write to the device, then the target device pulls the SDA low to ACK the transfer. If the data byte is a read from the device, the controller pulls the SDA low to acknowledge the data has been received. The ACK is a useful debugging tool. The absence of this bit can indicate that the target peripheral did not receive the proper I²C address for communication, or that the controller peripheral did not receive the expected data.

After the communication is completed, the controller issues an I²C STOP condition. SCL is first released and then SDA is released. The controller uses the STOP to indicate that the communication is completed and the I²C bus is released.

This is the basic protocol for any I²C communication between the controller device and the target device. Communication can consist of more than one byte of data. In some cases where a target device has multiple data and configuration registers, a read from a device can begin with a write to the device to indicate which register is to be read. The following sections show examples on how to read from and write to different data converter devices.

4 I²C Examples

This section uses two examples to show how I2C can communicate with different data converters. First, the I²C protocol is used to write to the DAC data register of the DAC80501 to set the output voltage. Second, I²C is used to read from the conversion register from the ADS1115 ADC.

4.1 DAC80501 Example

The DAC80501 is a 16-bit precision voltage output DAC with an internal reference. When the SPI2C pin is set high at power up, the device uses an I²C interface and is capable of standard, fast, and fast-mode plus I²C modes. Figure 4-1 shows the functional block diagram of the device.

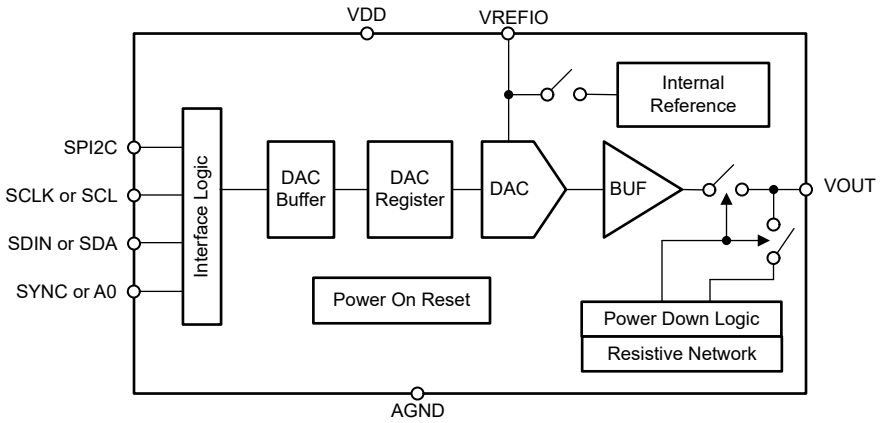

Figure 4-1. DAC80501 Functional Block Diagram

The DAC80501 has an address pin labeled A0. This pin is used to select one of four I²C addresses, meaning that four of these devices can be used on the same bus as long as the devices are programmed to different addresses. With the A0 pin connected to AGND, V_{DD}, SDA, or SCL, the device can be set to four unique addresses shown. For this example, the A0 is set to V_{DD}, so the address is 49h.

The device has a set of registers that can be used to enable the DAC reference and the output; set the output range through either a reference divider or a buffer gain; set a reset or enable the LDAC trigger; and the DAC output code. These registers can also be read to verify the settings, to identify the device from different versions of DAC resolution and power-on reset values, and to check an alarm for a low supply based on the reference.

The DAC80501 has seven internal registers, each addressed by a command byte (an additional register is a NOOP register (no operation) that is write-only and does not send a command or set a register value). Each register has 16 data bits with two data bytes for access. Communications with a command or register byte in this example is a common method used in I²C devices with multiple registers.

The DAC data register (08h) sets the DAC output voltage. For this example, the DAC data register is written to using the I²C protocol to set the output voltage of the device. The data write consists of the address byte, the register byte, and two data bytes, for a total of four communication frames.

4.1.1 DAC80501 DAC Data Register

The output to the DAC80501 is set through the DAC data register based on the transfer function in Equation 1.

$$V_{OUT} = \frac{DAC\_DATA}{2^N} \times \frac{VREFIO}{DIV} \times GAIN \quad (1)$$

where

- DAC_DATA is the 16-bit value in the DAC Data Register
- N is the number of bits of the DAC (16 bits)
- VREFIO is the reference voltage (2.5-V internal reference)
- DIV = 1 (as the default setting)
- GAIN = 2 (as the default setting)

For example, the DAC output can be set to an output voltage of 1.5 V. To do this, calculate the voltage based on the transfer function equation. Set VOUT to 1 V and calculate a value for DAC_DATA.

$$\text{DAC\_DATA} = \frac{V_{OUT}}{GAIN} \times \frac{DIV}{VREFIO} \times 2^{N} = \frac{1.5\ V}{2} \times \frac{1}{2.5\ V} \times 2^{16} = 19661 = 4CCDh \quad (2)$$

To write 4CCDh to the DAC80501, there are four bytes to be written. I²C communications start with the address byte and then is followed by the command byte to indicate which register is to be written to. The communication frame ends with the two bytes of the DAC_DATA register to set the output voltage.

4.1.2 DAC80501 I²C Example Write

The I²C write begins with a START condition. SDA is pulled low, and then SCL is pulled low. Then the I²C address is written. With the A0 pin connected to VDD, the DAC80501 responds to an address of 100 1001 (or 49h). The Read/$\overline{\text{Write}}$ (R/$\overline{\text{W}}$) bit is set low, indicating that the controller is writing to the device.

After the completion of the address byte, the DAC80501 ACKs the address by pulling down the SDA for the last bit of the address frame. The controller sends out the address and read or write information to all of the targets on the bus. If the target DAC80501 has a matching address, the device sends an ACK to indicate to the controller that a valid address is received and so that the device is ready to receive information.

After the controller sends the address with the write to the DAC80501, the controller tells the device which register is being written to. The second byte sent to the target device is the register pointer for the DAC Data register. Here, 0000 1000 is sent to the DAC80501. As a response, the DAC80501 pulls down on SDA for an ACK. Again, the target device is indicating to the controller that the device has received the address pointer data and which register is being written to.

Now, the DAC data register value is sent to the target device one byte at a time. For this byte, send in the first byte of the DAC data. Bit 0100 1100 is sent to the DAC80501. The DAC80501 ACKs the first byte.

Finally, the last byte of the configuration register is sent to the target device. Here, 1100 1101 is sent to the DAC80501. The DAC80501 ACKs this second data byte. At the end, SCL is released high and then SDA is released high. In this action, the controller releases the bus by issuing a STOP condition.

Putting the frames together, the I²C write appears as Figure 4-2. Here,the diagram shows the entire communication with the proper bit settings for all frames. If an oscilloscope plots the I²C communication for SDA and SCL, this figure can be directly compared with the plot for debugging.

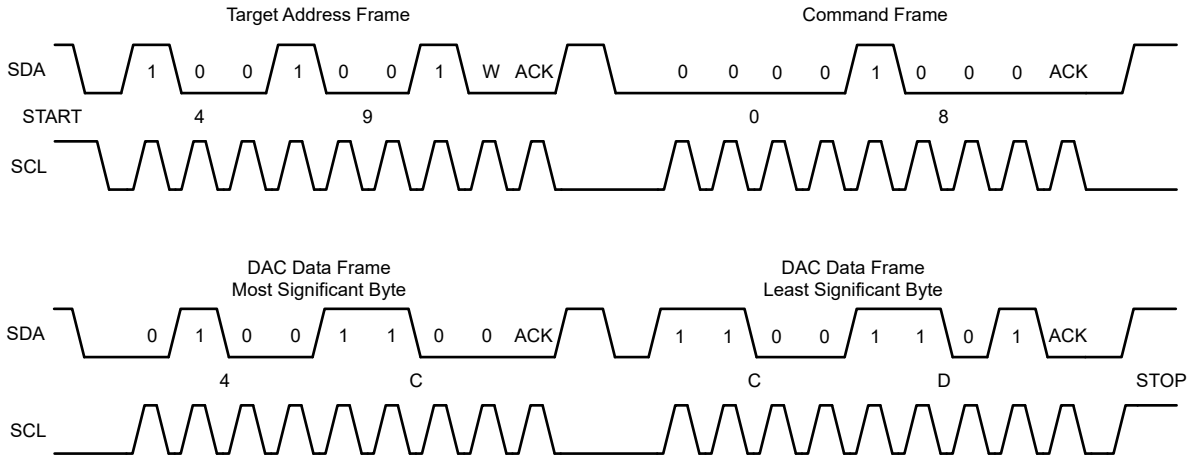

Figure 4-2. Example Write to the DAC Data Register of the DAC80501

4.2 ADS1115 Example

In this second example, I²C is used to read data from a precision ADC. The ADS1115 is a 16-bit precision ADC that uses an I²C interface and is capable of standard, fast, and high-speed modes. The device has several settings that can be set through a configuration register, including the input range set by a programmable gain amplifier (or PGA), a variety of data rates, and an input multiplexer that can be set to make differential measurements, or single-ended measurements with respect to ground. Figure 4-3 shows the functional block diagram for the ADS1115.

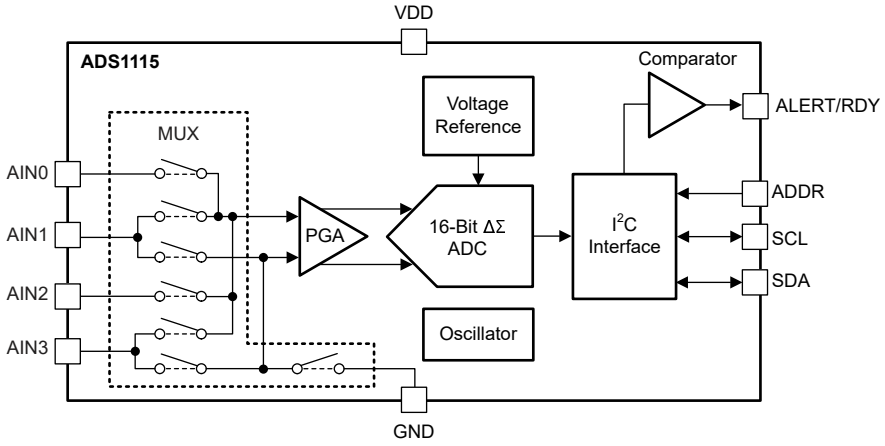

Figure 4-3. ADS1115 Functional Block Diagram

Similar to the previous example, the ADS1115 has an address pin labeled ADDR. This pin is used to select one of four I²C addresses, meaning that four of these devices can be used on the same bus as long as the devices are programmed to different addresses. The I²C address used for the device depends on the ADDR pin connection. With the ADDR pin connected to VDD, SDA, or SCL, the device can be set to other addresses shown in the table. For this example, the ADDR set to ground, so the address is set to 48h.

The ADS1115 has four internal registers, each addressed by an internal pointer register. The first register is the conversion data register. When the ADC completes a conversion, the ADC data is placed in this register and the controller device reads this register.

The second register is the configuration register. The controller device writes to this register to program the device and start a conversion. The configuration register sets the programmable gain amplifier, the input channel, the data rate, and other modes of operation for the device.

The last two registers are the Lo_threshold register and the Hi_threshold register. These two registers are used to set thresholds for a digital comparator in the device. Once the conversion data goes beyond these thresholds, the device can set an alert to the ALERT/RDY pin. For this example, the comparator and the thresholds are not configured.

For this example, the settings for the configuration register are first shown to explain the settings and range of the ADC. After that, an example I²C read from conversion register for the ADC is shown.

4.2.1 ADS1115 Configuration Register

The ADS1115 has a 16-bit configuration register. Writing to this register programs the configuration of the device and starts a conversion. This section details these settings and the operational modes of the device. Figure 4-4 shows the configuration register data fields. This shows the functions in the configuration register and their bit positions. After determining the all of the settings for the configuration register, use an I²C write to set the register.

Figure 4-4. Configuration Register

15	14	13	12	11	10	9	8
OS	MUX[2:0]			PGA[2:0]			MODE
R/W-1h	R/W-0h			R/W-2h			R/W-1h
7	6	5	4	3	2	1	0
DR[2:0]			COMP_MODE	COMP_POL	COMP_LAT	COMP_QUE[1:0]	
R/W-4h			R/W-0h	R/W-0h	R/W-0h	R/W-3h	

Figure 4-4 shows the configuration register field descriptions, giving a detailed description of the bit setting. Starting with the most significant bit, bit 15 is the single-shot conversion start bit.

Bits 14 to 12 set the multiplexer setting of the device. For this example, the device is set to a single-ended measurement from AIN0 with respect to ground. Within the configuration register, set AINP to AIN0 and AINN to GND. To do this, set bits 14 to 12 to be 100 in binary.

Bits 11 to 9 set the PGA setting of the device. This is the setting for the programmable gain amplifier. This sets the full-scale range of the input measurement, setting how large of an input signal can be measured by the ADC. Set the ADC to measure a signal as large as plus and minus 4.096 V. Set bits 11 to 9 to be 001 in binary.

Bit 8 sets the operating mode of the device. For this operation, set the device to be in single-shot conversion mode, set bit 8 to 1.

Bits 7 to 5 set the data rate for the ADC of the device. Set this to the highest data rate of 860 samples per second. Set bits 7 to 5 to 111.

The last five bits from 4 down to 0 control the digital comparator for this device. The digital comparator is not used, and is disabled with the last two bits of the register. The remaining bits are in their default setting. Set bits 4 to 0 to 00011.

This completes the configuration register setting for the ADS1115. These bits are used for the write to the register. This register can also be represented in hexadecimal as C3E3h. To write this to the device, use the same format for writing to the target device as shown in Figure 4-2 for the DAC80501.

4.2.2 ADS1115 I²C Example Read

The ADS1115 has a 16-bit ADC and therefore puts out 16-bit data conversions. The controller device reads from the conversion register to get the ADC conversion data. The conversion register address pointer is 00h. Conversion data appears as a 16-bit result in binary two's complement. A positive full-scale input produces an output code of 7FFFh and a negative full-scale input produces an output code of 8000h.

With the ADDR pin connected to GND, the device responds to address 48h. Figure 4-5 shows an example read from the Conversion Data register at the 00h address pointer.

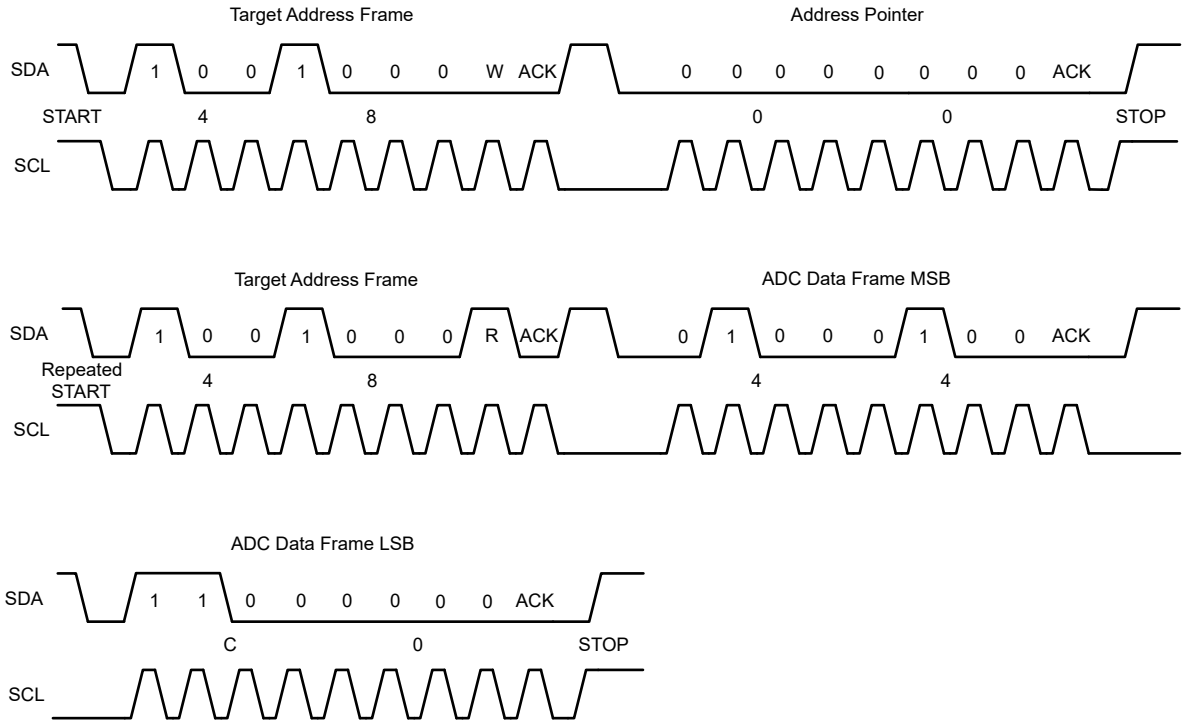

Figure 4-5. I²C Transmission for Reading from the ADS1115

The I²C write begins with a START condition. SDA is pulled low, and then SCL is pulled low. Then the controller writes the I²C address. Again, the device responds to an address of 100 1000 (or 48h).

The controller first needs to tell the device which register is to be read from. For this, the communication first sends an I²C Write to the device so that to set up the read from the Data Conversion register of the ADS1115. At this point, the R/$\overline{W}$ bit is set low, indicating that the communication begins with a write to the device.

When the address frame is completed, the ADS1115 ACKs the address by pulling down the SDA for the last bit of the address frame.

After indicating that the controller is reading from the ADS1115, the controller tells the device which register is read from. The second byte is the register pointer for the Data Conversion register. Here, send 00h to the ADS1115. As a response, the ADS1115 pulls down on SDA for an ACK. Finally, the controller issues a STOP to release the bus.

Now that the controller has told the device to access the data conversion register, the controller follows up with the read from the register. The controller writes the I²C address again. The controller has already indicated which register is to be read from, now the controller sends a read to the device so that the data conversion register of the ADS1115 can be read. At this point, the R/$\overline{W}$ bit is then set high, indicating the read. Again, after the completion of the address frame, the ADS1115 ACKs the address.

The next two data bytes are used to read the register. The first byte is the most significant byte of the conversion data and then the second byte follows as a read of the least significant byte conversion data. An ACK from the controller follows each byte controller. Finally, the controller sends a STOP to end the I²C communication.

As in the previous example, Figure 4-5 is a convenient diagram showing the I²C communication with the device. If there are problems in communication, an oscilloscope plot of the SDA and SCL can be used to compare against the this example write.

4.2.3 ADS1115 Conversion Result

Using the conversion register result, this example continues with calculating the conversion result. The conversion register is read as 44C0h, or 17600 in decimal. This is the ADC output based on the input voltage from the measurement. Figure 4-6 shows the data contents of the Conversion Register.

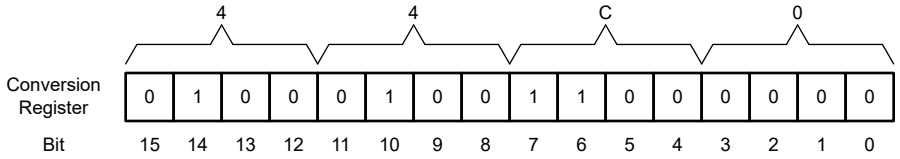

Figure 4-6. ADC Conversion Register Contents

Using the value in the conversion register, convert the conversion register to a voltage for the ADC measurement. With a positive full-scale range of 4.096 V, convert this data to a measured voltage.

$$\text{Conversion Voltage} = 4.096\ \text{V} \times \text{Data Code} / 2^{15} = 4.096\ \text{V} \times 17600 / 32768 = 2.2\ \text{V} \quad (3)$$

With an ADC data code of 17600d, the ADC reports a measurement of 2.2 V.

5 Reserved Addresses

In the I²C protocol, writing to and reading from the target device requires the use of an I²C address. The I²C address identifies which device the controller wants to communicate with. Typically, this address is written over a single byte, where the address itself is 7 bits, and an eighth additional bit is used to indicate a read from or write to the device

However, not all addresses of the 7 bits can be used for target devices. Some addresses are reserved for other purposes. This section shows what functions these reserved addresses are used for.

I²C has several sets of reserved addresses that are limited for use based on specific applications. The functions called with these reserved addresses are options for devices, and are not necessarily available in all I²C devices. Table 5-1 lists a set of these reserved addresses and their functions.

Table 5-1. List of Reserved I²C Addresses

Target Address	R/$\overline{W}$ Bit	Description
000 0000	0	General call address
000 0000	1	START byte
000 0001	X	C-Bus address
000 0010	X	Reserved for different bus format
000 0011	X	Reserved for future purposes
000 01XX	X	Hs-mode controller code
111 11XX	1	Device ID
111 10XX	X	10-bit target address

5.1 General Call

The first reserved address is I²C address 0 and is the general call address. A write to the general call address is used to address all the devices connected to the I²C bus at the same time. Not all devices are designed to respond to the general call address. However, if there is a response, then the target device can process a second byte and the following bytes after the general call. Figure 5-1 shows the two-byte format for the general call address.

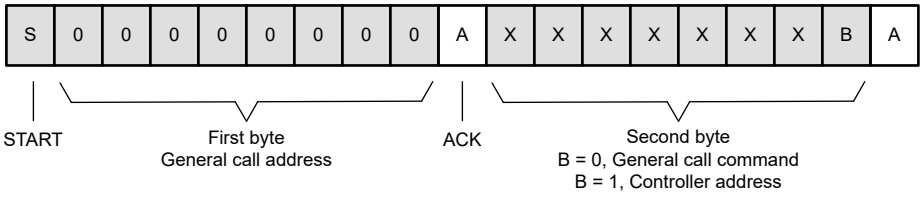

Figure 5-1. General Call Address Format

The response of the device is characterized by two different cases dependent on the least significant bit of the byte following the general call. If the least significant bit of the second byte (B) is a zero, the second byte can be used to send commands to those devices receiving the general call. As one example, a common command sent through I²C is the general call reset. If the second byte is 06h, the controller device is sending a general call reset to all devices on the I²C bus.

If B is a one, the controller is sending the two-byte sequence as a hardware general call. In this case, the controller device can be a hardware controller that cannot be programmed to transmit to a particular target address or does not know what device to send the command to. In this case, the second byte is used to send the controller address to identify itself to all the devices on the system. This address can then be recognized by another controller device in the system that can be used to direct information to the hardware controller acting as a target device.

5.2 START Byte

A read from I²C address 00 is the START byte. Not all microprocessors have a built-in, onboard I²C controller and do not necessarily have an interrupt to detect communication on the bus. In that case, a microprocessor monitoring the I²C bus must repeatedly poll the SDA and SCL lines to make sure communications are not missed. This requires the device to poll at a fast rate to detect the I²C address.

One option can be to use a START byte before the communication. When the controller uses the START byte with 7 zeros in the address, the target device can poll at a much slower rate, saving processing power.

After the target device detects that the SDA sent a zero during the START byte, the device can then switch to faster polling to detect the next I²C transmission for the address being sent. Once the I²C transfer has ended with a STOP condition, the target device can then resume polling at a slower rate, again saving processing power.

5.3 C-Bus Address, Different Bus Format, Future Purposes

The next three I²C addresses listed in the table are all reserved for different reasons. Address 01 is reserved for the C-Bus protocol so that C-Bus devices can be placed on the I²C bus. However, the C-Bus protocol is used for home and building automation in some parts of the world, but is generally not used in the United States. This I²C address is ignored by most devices.

Address 02 is reserved for different bus formats. This is designed to allow communication between different protocols. Only I²C devices that work between different protocols can respond to this address.

Address 03 is reserved for future purposes (yet to be defined).

5.4 HS-Mode Controller Code

The next reserved addresses are for the high-speed controller code. These codes are from 04 to 07. The eighth bit normally used for read or write indication is used as part of the high-speed controller code. These high-speed controller codes are reserved 8-bit codes which are not used for target addressing or other purposes. Each high-speed controller has a unique controller code and this allows for up to eight high-speed controllers on the I²C bus. The controller code for a high-speed mode controller device is software programmable and is chosen by the system designer.

Devices that support high-speed mode begin operation in standard or fast mode. The controller code enables high-speed mode. The high-speed controller code allows for arbitration between the high-speed controllers and indicates the start of a high-speed mode transfer. The code enables internal current sources allowing the I²C communication bus to be faster than with just pullup resistors.

When enabled, high-speed data transfer continues through the data transmission. A repeated START continues high-speed mode data transmission, while a STOP condition returns the I²C bus to fast or standard mode. Figure 5-2 shows the start transmission to the high-speed controller code. This diagram shows the beginning of a high-speed mode transmission.

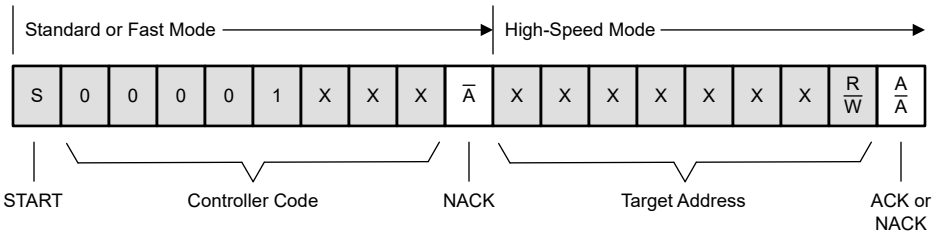

Figure 5-2. I²C High-Speed Controller Code

For this and subsequent byte and bit frame diagrams, the shaded codes are set by the controller device, while the non-shaded codes are set by the target device.

At the start of communication, the device starts in standard or fast mode. The controller sends a START condition by pulling SDA low followed by pulling SCL low. Then, the first byte is sent with the reserved address used for the high-speed controller code. The controller code enables high-speed mode for all devices that are capable of high-speed mode, and the internal circuits of the controller for high-speed mode are enabled. Figure 5-3 shows a detailed figure on how the I²C high-speed mode is enabled followed by communication with data.

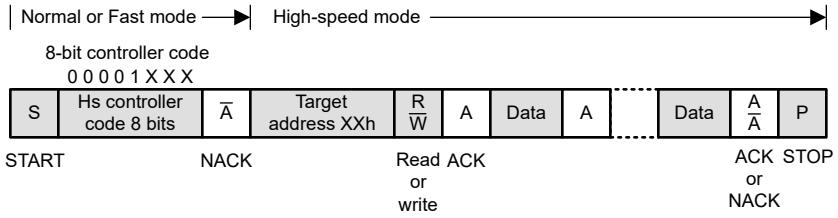

Figure 5-3. Enabling I²C High-Speed Mode

The controller then sends the target address of the high-speed mode device and follows with a read or write bit for communication. Data is transmitted by the controller or target, with ACK for each data byte similar to the standard I²C communication. The target device continues communication until receiving a STOP condition or receiving a repeated START for a new target address.

5.5 Device ID

Addresses 7C to 7F are all reserved for Device ID. The controller begins by sending the Reserved Device ID address followed by a write bit. The controller then sends the target device address to identify. The controller then sends a repeated START condition followed by the reserved Device ID address followed by a read bit.

The Device ID is sent by the target through the bytes in three I²C data frames. This data starts with 12 bits for the manufacturer ID, followed by 9 bits for the part identification, completed by 3 bits for the die revision. Figure 5-4 shows the data transmission to the Device ID.

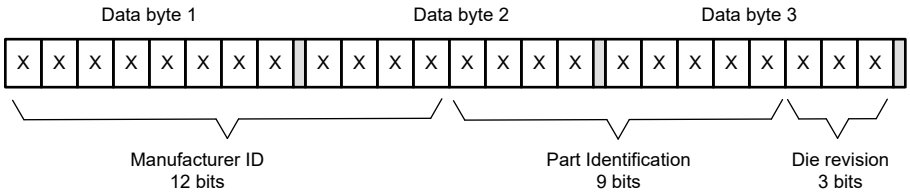

Figure 5-4. I²C Device ID Data Bits

The controller first sends a START condition. The first byte is sent with the reserved address for Device ID and is followed by a 0 for a write. At this point, there can be multiple targets that respond to Device ID, so multiple devices can ACK this address.

The controller then sends the address for the target device. The last bit of this target address byte is a “Don’t Care” followed by an ACK. At this point, there is only one device that ACKs this address.

The controller then sends a repeated START condition. After that, the controller sends the Reserved Device ID I²C-bus address followed by the read bit. The target device ACKs this reserved address. Note that the beginning of this third byte must be a repeated START. A STOP followed by a START condition, or a STOP with a repeated START condition followed by access to a different target device resets the target device state machine and the Device ID read cannot be performed.

The target device then sends three bytes for device ID. Figure 5-5 shows a detailed figure on how the Device ID is read from a device that supports the I²C Device ID.

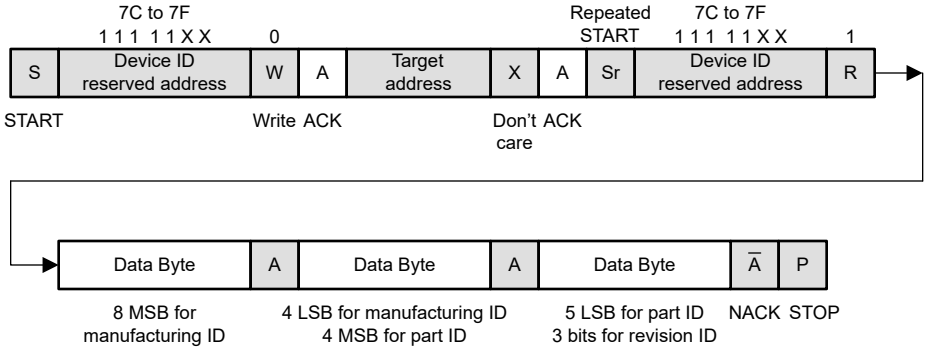

Figure 5-5. Reading the I²C Device ID

The controller NACKs the last byte and concludes the Device ID read with a STOP. The reading of the Device ID can be stopped at any time by sending a NACK. If the controller continues to ACK the bytes after the third byte, the target rolls back to the first byte and keeps sending the Device ID sequence until a NACK has been detected.

5.6 10-Bit Target Addressing

With the normal 7-bit I²C address and all of the reserved addresses, the number of possible I²C devices on a bus becomes limited. To expand the number of devices, several reserved addresses can be used to expand the address to 10-bits. In the reserved address, the last two bits of 78h to 7Bh represent the first two bits used to expand the address space. A second full byte of eight bits is used to complete the 10-bit address. Communication with the 10-bit address is very similar to the 7-bit address communication.

5.6.1 10-Bit Target Addressing Write

Figure 5-6 shows the protocol to writing to a device that supports 10-bit addressing. At the beginning of a 10-bit address write, the controller sends a START condition. Then, the first byte is sent with the reserved address for 10-bit I²C addressing and is followed by a 0 for a write. The reserved address last two bits includes the first two bits of the 10-bit address. At this point there can be multiple targets that have the same first address byte for 10-bit addressing, so multiple devices can ACK this address. The second byte includes the target address. This byte is the eight least significant bits of the 10-bit target address.

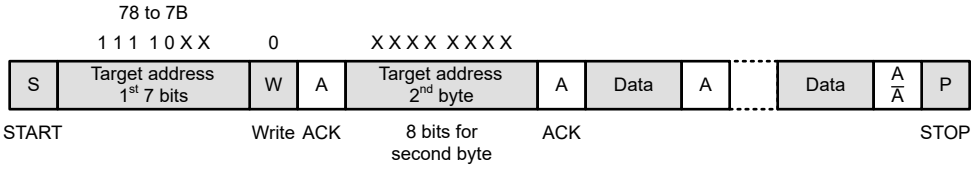

Figure 5-6. I²C Ten-Bit Target Addressing Write

With the second byte, there is presumably only one device with the unique 10-bit address. This is the only device that ACKs the communication. With the two bits of the reserved address and the eight bits of the second byte target address, this totals 10-bits of addressing.

This target stays in communication until the controller sends a STOP condition or until the controller sends a repeated START condition to communicate with a different target address.

5.6.2 10-Bit Target Addressing Read

Reading from a 10-bit addressed device is similar to a write but with added steps. At the beginning of the communication, there is a START followed by the reserved address. A 0 for a write bit is then written. This is followed by an ACK from all devices that use the reserved address. The target address second byte is then sent. An ACK from the addressed device is then received. Until this point, the communication is exactly the same as a 10-bit address write.

Figure 5-7. I²C 10-Bit Addressing Read

To read from this device, the controller then sends a repeated START. This step is followed by the reserved address that was just used. Then the read bit is sent followed by an ACK. Because the read bit is sent (and not the write bit), the device that previously ACKed this communication interprets that this is a read. Other devices with the same reserved address do not respond. The addressed device then ACKs this repeat of the reserved target address.

After the addressed device sends the ACK, data is transmitted by the device, and after each byte, the controller ACKs the data. This data transmission continues until the controller sends a STOP condition or a repeated START followed by a different target address.

6 Advanced Topics

In previous sections, the protocol basics of I²C were discussed with examples to show communications with precision data converters. Those sections show how I²C works and how to read, write, and help debug basic system communications.

However, those descriptions only scratch the surface of the I²C protocol. This section covers some advanced topics of I²C. The following is not covered in detail. However, this section introduces some topics that allow you to understand what these topics are. More detailed information is found in the I²C bus specification.

6.1 Clock Synchronization and Arbitration

The first I²C topic in this section is clock synchronization and arbitration between controller devices on the bus. In I²C, there can be multiple controllers on the same bus. Because of this, there can be two or more devices trying to claim the bus for communication at the same time. This requires multiple active controllers to resolve which device controls the bus.

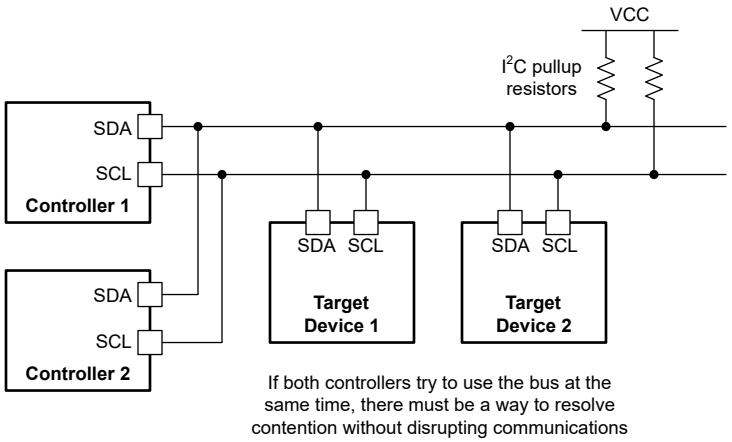

Figure 6-1. I²C Bus Contention With Multiple Controllers

I²C uses a method of clock synchronization and arbitration to make sure that one controller gains control and does so without compromising communication. Because I²C uses open-drain connections to SDA and SCL, the connections result in a wired-AND connection, where the line gives a logical AND of the device outputs. This is helpful in arbitration without disruption to the communication. In systems with only one controller, this arbitration is not necessary.

This section details clock synchronization and how multiple controllers synchronize clocks for I²C to prevent contention. How controllers use arbitration to determine which controller wins the bus without disruptive contention is also described.

To prevent bus contention, clock synchronization is first performed using the SCL line and the open-drain connections from the controllers on the bus. This wired-AND connection is low if any of the controllers pull SCL low. This connection is the logical AND of the SCL connection of the two controller devices. The output of SCL is high only if both controller devices have released the open-drain connection high. Table 6-1 details a truth table of this logical wired-AND.

Table 6-1. Wired-AND Truth Table

Controller 1 SCL	Controller 2 SCL	Resulting SCL
0	0	0
0	1	0
1	0	0
1	1	1

During a START condition where two controllers are trying to claim the bus, there is a high-to-low transition on SCL. Here is an example where two controller devices are trying to claim the bus at or near the same time.

In Figure 6-2, the controller 1 device initiates a START condition shortly before controller device 2 does the same. Controller 1 pulls SCL down before controller 2. With the wired-AND connection, SCL pulls low as soon as controller 1 pulls down on SCL.

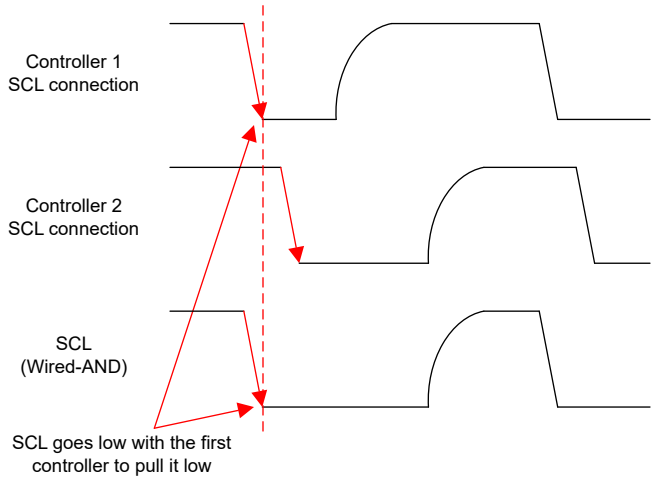

Figure 6-2. I²C Clock Synchronization SCL Going Low

After the START condition, controller 1 releases SCL to go high. However, controller 2 is still holding SCL low. Because of the wired-AND connection, SCL remains low until controller 2 releases the SCL high. At the same time, controller 1 is still monitoring SCL and must wait for the other controller to release the clock. Controller 1 cannot advance the SCL pulse until controller 2 has released SCL and becomes available.

When multiple controllers are competing for the bus, SCL stays low for as long as the longest period of time that any controller pulls down SCL. Only after all the controllers have released the SCL can the line be released high for the serial clock pulse. This synchronizes the start of the serial clock for all controllers. Figure 6-3 shows the resulting SCL as both controllers release the SCL.

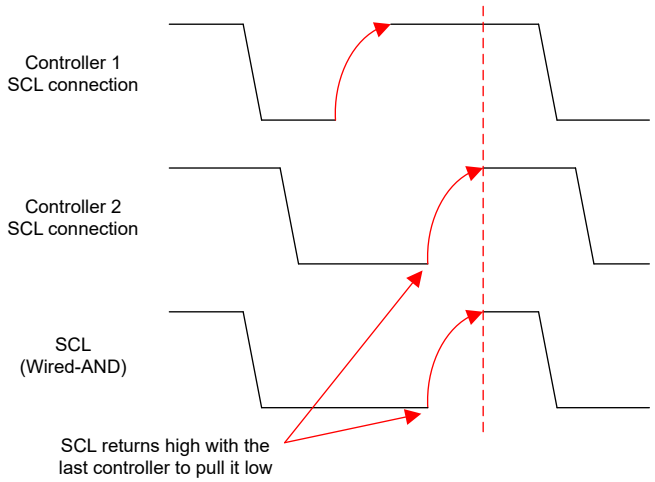

Figure 6-3. I²C Clock Synchronization SCL Returning High

For clock synchronization, each controller device must monitor the SCL line and react to cases where the SCL does not match the expected SCL output.

Similarly, after the beginning of the serial clock pulse, all the controllers pull down on SCL to complete the serial clock pulse. Again, with the wired-AND connection, SCL is then pulled down with the first controller that responds by pulling down SCL. The first controller that completes the SCL high-time period determines the high time of SCL from the wired-AND connection. Figure 6-4 shows the SCL line as SCL is again pulled low.

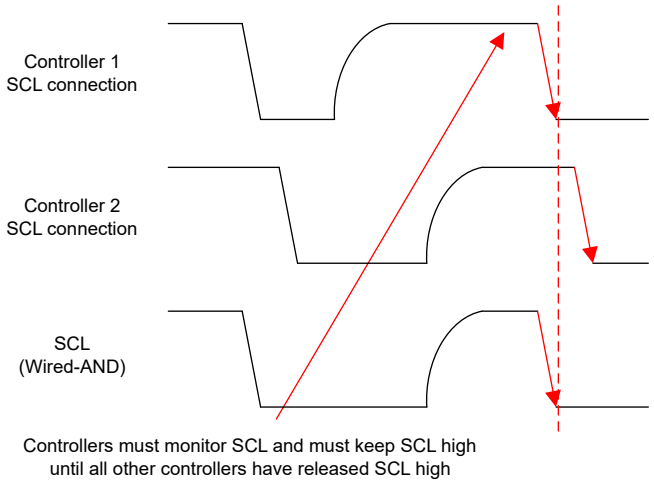

Figure 6-4. I²C Clock Synchronization Monitoring SCL

The synchronization of the SCL clock continues for subsequent clock pulses between all active controllers. Each SCL clock pulse is generated with the low period determined by the controller with the longest clock low period and the high period is determined by the controller with the shortest clock high period. Clock synchronization continues through the communication shown in Figure 6-5.

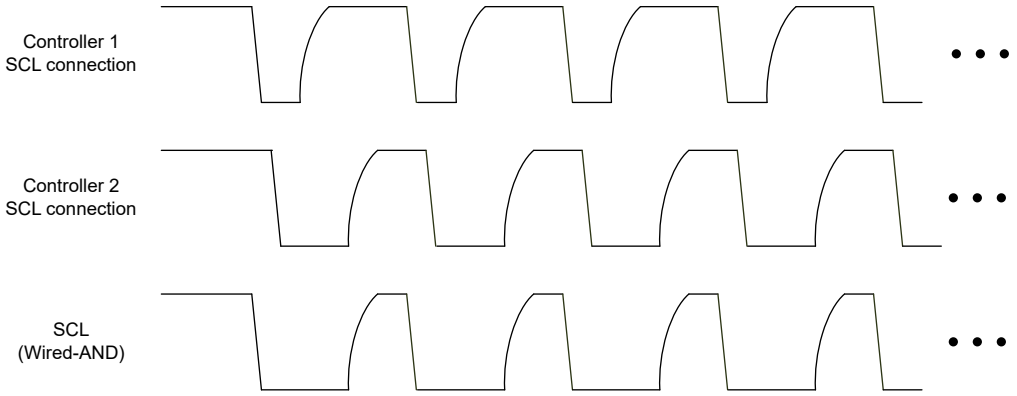

Figure 6-5. I²C Clock Synchronization Resulting Wired-And

Clock synchronization works because the controllers monitor each pulse of the SCL line and react to cases where the SCL line does not match the state that the controller expects.

Now that the serial clocks are synchronized, arbitration is done on SDA. Both controllers transmit data normally on SDA, sending their communication to the intended target device. Similar to SCL, SDA is a wired-AND connection. Figure 6-6 shows arbitration of SDA after clock synchronization.

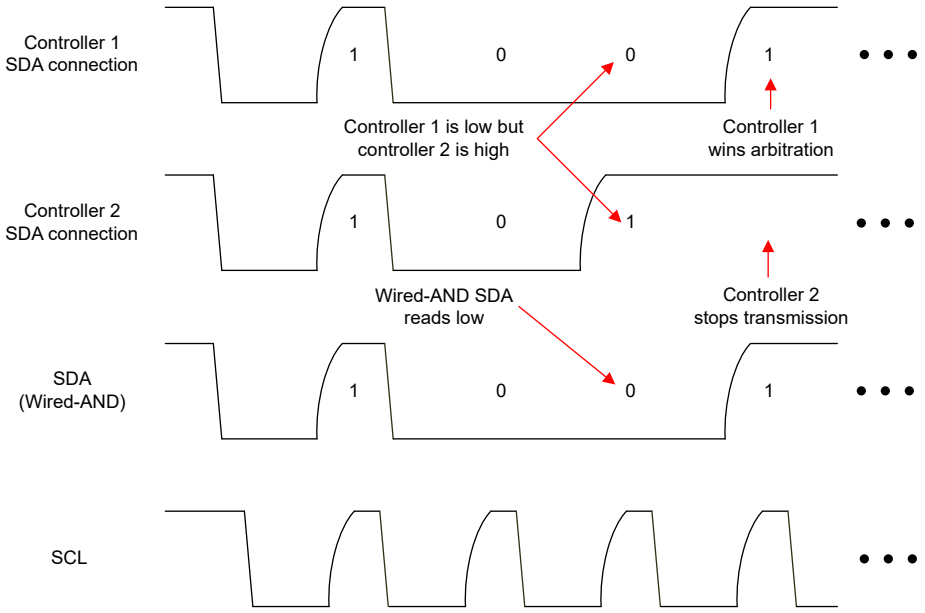

Figure 6-6. I²C Controller Arbitration

In bus arbitration, the communication continues until there is a noted difference between the data being sent.

In the figure, both devices send the I²C START at the same time. For the first two bits of the transmission, the data are the same. After the data reaches the third bit, there is a difference as controller 1 sends a 0, while controller 2 sends a 1. Because both controllers monitor the SDA and SCL lines, the contention is discovered. Controller 2 discovers that SDA is low despite sending a 1. To preserve the communication on the bus with the correct wired-AND result, controller 2 releases the bus, while controller 1 wins the arbitration.

6.2 Clock Stretching

In some I²C target devices, there are situations that the target device controls the SCL serial clock. In those cases, the target device can slow down the communication. This process is known as clock stretching.

In general, the SCL line and therefore the I²C clock rate, is controlled by the controller. However, there are instances where the target device is unable to comply with the clock rate. For example, the target device requires extra time to process a command or send data. In such cases, the target device can slow down the communication through clock stretching.

With clock stretching, after the controller sends a byte of data in transmission, the target device holds down SCL longer so that the controller is required to adjust the clock. This manipulation of the SCL is similar to clock synchronization. The controller monitors SCL and is forced to extend the SCL pulse if SCL is still low after the controller has released the clock. Any SCL pulse can be clock-stretched by the target device. However, the general implementation of clock stretching is done with the SCL pulse at the time the ACK bit is sent.

According to the I²C specification, there is no time limit to the target holding down SCL for clock stretching. Other similar specifications (like SMBus) have time limits for how long SCL can be held low.

Figure 6-7 shows an example of the target device clock stretching SCL. In this example, the controller issues a START and sends the target device address.

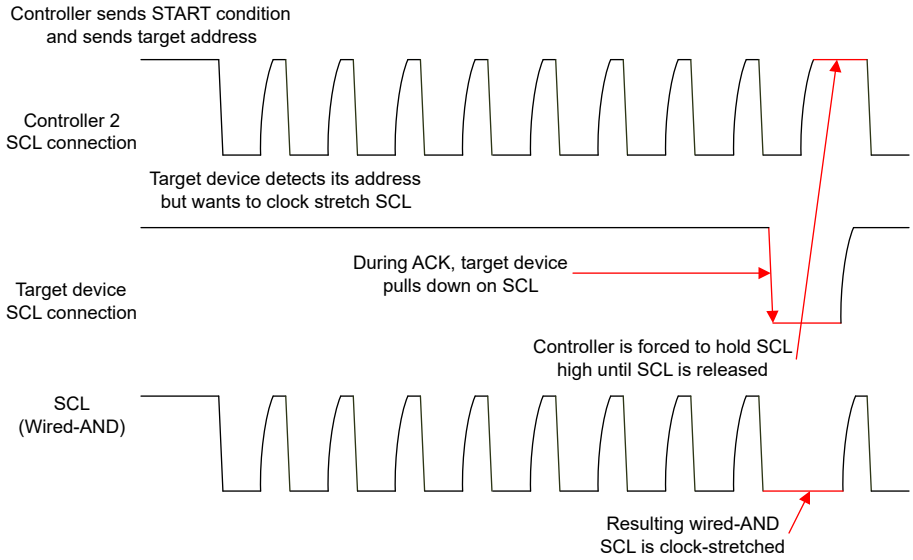

Figure 6-7. I²C Target Clock Stretching

When the target device recognizes the controller is sending the proper target address, the target device ACKs the address. If clock stretching is needed to slow down communications, the target device can pull down on SCL during the ACK. This is the only instance the target device can control the SCL.

When the target device begins clock stretching, SCL remains low even though the controller has released SCL. Because the target device has control of the clock, the controller cannot continue with the SCL pulse until the SCL is released by the target. The controller continues to monitor SCL. Once SCL is released high, the controller can then continue past the ACK of the target device and continue with the next byte transmission. The resulting wired-AND connection of SCL shows the SCL stretched. Data transmission is delayed by the target device without disrupting communication.

6.3 Electrical Specifications

The data sheet for every I²C device has electrical specifications that cover the characteristics for the I²C bus. Because I²C is a common protocol, these specifications are matched from device to device. This section discusses the electrical characteristics as shown in the I²C specification.

This application note does not go into detail about each of the specifications, but gives an overview of how these specifications are organized. Data sheets for I²C devices cover specifications on what is needed to operate TI devices.

As an example, Table 6-2 shows electrical characteristics from the ADS1119 digital input and output for the I²C digital lines.

Table 6-2. Electrical Characteristics of the ADS1119 Digital Inputs and Outputs

Parameter		Test Conditions	Minimum	Typical	Maximum	Unit
V_{IL}	Logic input level, low		DGND		0.3 DVDD	V
V_{IH}	Logic input level, high	2.3 V ≤ DVDD < 3.0 V, SCL, SDA, A0, A1, $\overline{DRDY}$	0.7 DVDD		DVDD + 0.5	V
		3.0 V ≤ DVDD ≤ 3.0 V, SCL, SDA, A0, A1, $\overline{DRDY}$	0.7 DVDD		5.5	V
		$\overline{RESET}$	0.7 DVDD		DVDD	V
V_{hys}	Hysteresis of Schmitt-trigger inputs	Fast-mode, fast-mode plus	0.05 DVDD			V
V_{OL}	Logic output level, low	I_{OL} = 3 mA	DGND	0.15	0.4	V
I_{OL}	Low-level output current	V_{OL} = 0.4 V, standard-mode, fast-mode	3			mA
		V_{OL} = 0.4 V, fast-mode plus	20			
		V_{OL} = 0.6 V, fast-mode	6			
Ii	Input current	DGND + 0.1 V < $V_{Digital\ Input}$ < DVDD – 0.1 V	–10		10	µA
Ci	Capacitance	Each pin			10	pF

Highlighting some of the parameters, the table gives specifications for low-level and high-level input and output voltages for SCL and SDA. This specifies that each I²C bus line has a voltage range that correctly transmits and receives high and low levels. This table also gives the minimum output current that the device open drains pull down on SCL and SDA. There are also specific characteristics depending on the I²C mode used.

In whatever I²C devices used, these SCL and SDA bus line characteristics are found in their respective data sheets. The data sheets give enough of these characteristics to set up the device correctly. For further information see the I²C specifications.

6.4 Voltage Level Translation

One common problem with designing large systems is the mixing of different voltage levels within the system. For example, what happens when the controller and the target device do not run on the same voltage?

Larger systems can have multiple power sources with multiple voltages. These different voltages can power different I²C controllers and target devices. This section discusses voltage level translation and how these different I²C voltages interact.

Mismatched voltages in the supply can disrupt communication or even damage a device. The connection of the pullup resistors determines if the output voltage of one overdrives or underdrives the input of the next device. The following examples show some of the consequences of the mismatch.

6.4.1 Example 1

In Figure 6-8, the controller and the pullups are set to 3.3 V, while the target device is set to 5.0 V.

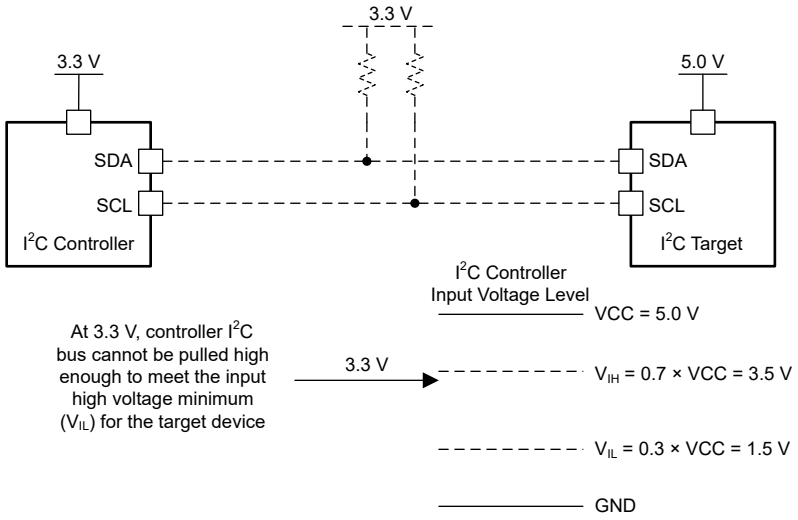

Figure 6-8. I²C Pullup Resistor Under-Drives Inputs on Mismatched Bus Voltages

In the I²C specification, there are minimum and maximum voltages required for a digital input voltage to be accurately interpreted as a digital high or low. For example, the SDA and SCL are interpreted as a digital input low voltage when the input goes below the maximum 0.3 × VCC. Also, the SDA and SCL are interpreted as a digital input high voltage when the input goes above the minimum of 0.7 × VCC. This latter specification is important for the mismatched supplies.

With the pullups tied to the lower supply of 3.3 V, the resistors are never able to pull up higher than the minimum required voltage of 3.5 V. In this case, neither the SDA, nor the SCL are designed to be high enough to be received as a digital high. This potentially prevents communication between the devices.

6.4.2 Example 2

In Figure 6-9 the controller is set to 1.8 V, but the pullups and the target device are set to 5.0 V.

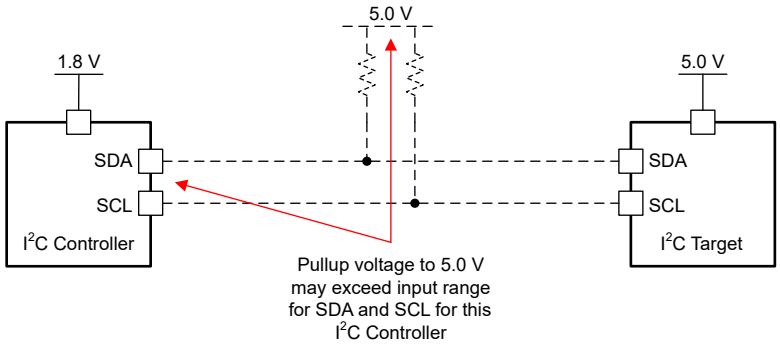

Figure 6-9. I²C Pullup Resistor Over-Drives Inputs on Mismatched Bus Voltages

In this example, the I²C bus lines can be pulled up to 5.0 V. However, one of the devices does not tolerate voltages that high. If the difference between the device voltages are too great, the lower voltage device can be damaged by the overvoltaged connection.

6.4.3 Example 3

In Figure 6-10, the controller and pullups are set to 5 V, but the target device is set to 3.3 V.

The I²C bus lines are able to be pulled up to 5 V, exceeding the target device supply. However, the target device has inputs tolerant to higher voltages. This higher tolerance is a feature in some I²C devices. This feature allows for direct connections between the I²C bus with pullups to the higher voltage supply. Check with the device data sheets for this possible feature.

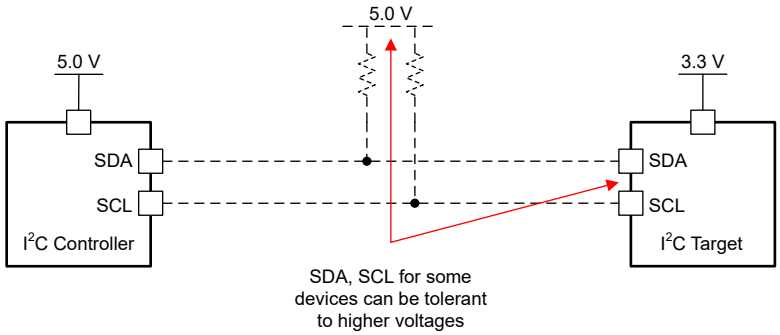

Figure 6-10. I²C Pullup Resistor Over-Driven on Higher-Voltage Tolerant Lines

As an example, the ADS1115 is just one device that has SDA and SCL lines that are tolerant to voltages higher than the supply. Looking at the Absolute Maximum Table from the data sheet, the maximum digital input voltage is 5.5 V, regardless of the supply voltage. With this type of I²C line, the target device can tolerate pullup voltages higher than the supply. This allows for I²C communication between the devices even with different supply voltages.

6.4.4 Example 4

With mismatched supply voltages, the best option is to use a special device to bridge the two supply voltages. Figure 6-11 shows an example of using an I²C voltage level translator to bridge the communication between two different supply voltages. There are two sets of pullups, one for each voltage level. As a common voltage translator, the PCA9306 allows for communication between different supply levels.

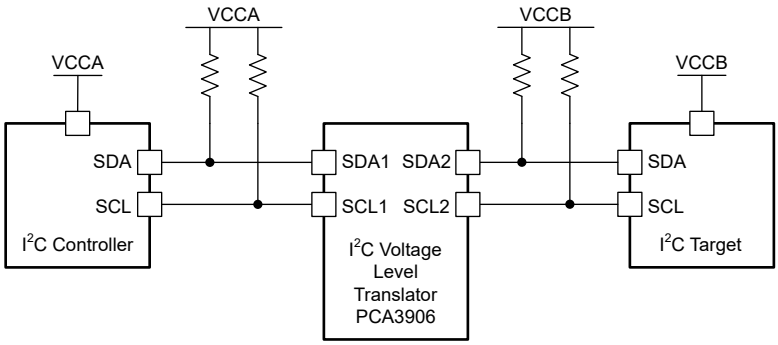

Figure 6-11. PCA3906 I²C Voltage Level Translator

6.5 Pullup Resistor Sizing

To design the system so that the bus speed is fast enough to meet the protocol bus speed, calculate the values for the pullup resistances.

With the open-drain connections of SDA and SCL, transitions from these lines from high to low and from low to high are dependent on the current sink from the device open-drain connection, the bus capacitance, and the pullup resistor value. Based on these different parameters, a minimum and maximum resistance can be calculated for the I²C bus speed.

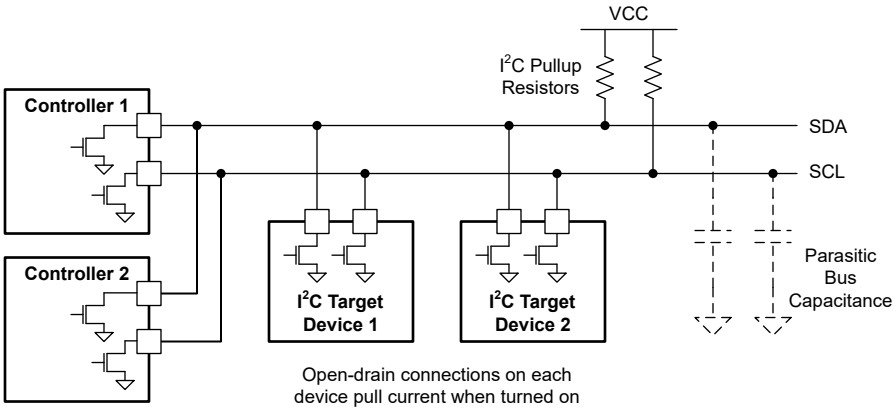

Figure 6-12. Factors Affecting Pullup Resistor Sizing

The normal pullup resistor recommendation is 1 kΩ to 10 kΩ. With higher resistances, the I²C communication is slower. With lower resistances, the I²C communication requires more power. Based on the several different parameters, a minimum and maximum resistance can be calculated for the I²C bus speed.

Table 6-3 lists some of the parametric characteristics of the I²C bus. The table lists the bit rate of the I²C bus, the maximum rise time for the bus, and the maximum capacitive load on the bus. All of these parameters are used to determine the minimum and maximum pullup resistance values.

Table 6-3. Parametric Characteristics From I²C Protocol

Parameter		Standard-mode (MAX)	Fast-mode (MAX)	Fast-mode Plus (MAX)	Unit
f_{SCL}	SCLK clock frequency	0 to 100	0 to 400	0 to 1000	kHz
t_r	Rise time of both SDA and SCL signals	1000	300	120	ns
C_b	Capacitive load for each bus line	400	400	550	pF

In addition to these parameters, the I2C input and output voltage minimums and maximums are considered. Table 6-4 describes these voltages.

Table 6-4. Characteristics of SDA and SCL Input and Output Voltages

Parameter		Standard Mode		Fast Mode		Fast Mode Plus		Unit
		MIN	MAX	MIN	MAX	MIN	MAX	
V_{IL}	Low-level input voltage	–0.5	0.3 × VCC	–0.5	0.3 × VCC	–0.5	0.3 × VCC	V
V_{IH}	High-level input voltage	0.7 × VCC	VCC + 0.5	0.7 × VCC	VCC + 0.5	0.7 × VCC	VCC + 0.5	V
V_{OL}	Low-level output voltage, 3 mA sink current; VCC > 2 V	0	0.4	0	0.4	0	0.4	V
	Low-level output voltage, 3 mA sink current; VCC ≤ 2 V	-	-	0	0.2 × VCC	0	0.2 × VCC	V

6.5.1 Minimum Pullup Resistance Sizing

Figure 6-13 shows an open drain connection to the I²C bus and the output waveform for SDA or SCL. The SDA and SCL bus transition low from the current pulling from the device.

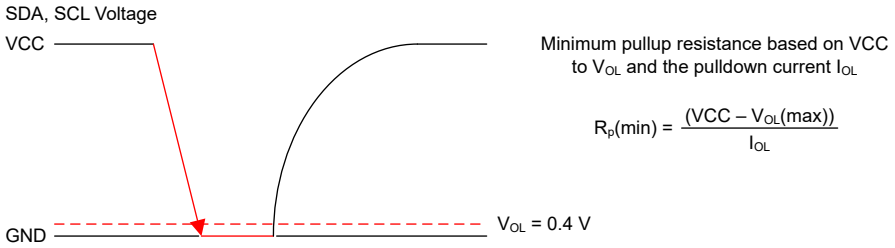

Figure 6-13. Minimum Pullup Resistance Based on Pull-Down Current

The bus line is connected to the V_{CC} voltage when the device releases the SDA or SCL. When active, the device drain pulls the bus line output to near ground. The output must drop to the output low-level voltage V_{OL}. The device pulls the bus line low with current I_{OL}. V_{OL} and I_{OL} (the 3-mA current sink) are described in Table 6-4. Based on this current, calculate the minimum resistance needed for the pullup. If the resistance is smaller, the output current cannot pull the output voltage of the bus low enough to be recognized as a digital low. This is shown in Equation 4.

$$R_P(min) = (VCC - V_{OL}(max)) / I_{OL} \quad (4)$$

Solving for the minimum pullup resistance, subtract the output low voltage of 0.4 V from the supply voltage of 3.3 V. Then divide by the current pulled by the bus line of 3 mA. This results in 967 Ω as the minimum resistance.

6.5.2 Maximum Pullup Resistance Sizing

After the open-drain connection releases the output current, pullup resistors pull the bus connection high. The bus line output waveform has an exponential settling. As the resistor pulls the voltage up from ground, the voltage settling time is based on the bus capacitance (C_B). The maximum pullup resistance is limited by the bus capacitance because of the I²C standard rise time specification. With a higher resistance, the pullup output rises too slowly, and does not reach the logical high fast enough.

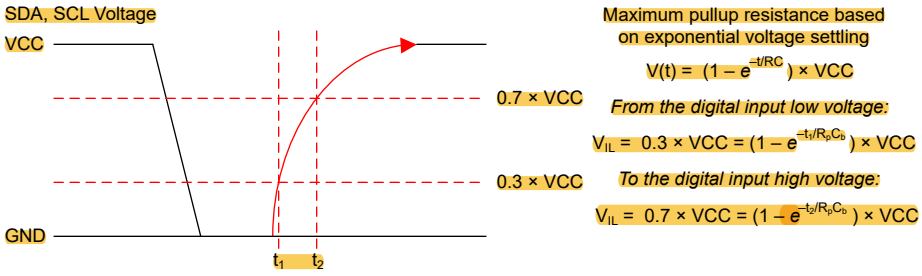

Figure 6-14. Maximum Pullup Resistance Based on Rise from Pullup and Bus Capacitance

The equation for the exponential settling over time is shown in Equation 5 with the pullup resistance.

$$V(t) = \left(1 - e^{-t/RC}\right) \times VCC \tag{5}$$

The rise time is based on the transition from the digital input low voltage (V_{IL}) of 0.3 times the supply voltage to the digital input high voltage (V_{IH}) of 0.7 times the supply voltage. The rise time is described in Table 6-3 while V_{IL} and V_{IH} are described in Table 6-4. The pullup settling with these parameters results in Equation 6 and Equation 7.

$$V_{IL} = 0.3 \times VCC = \left(1 - e^{-t_1/R_P C_B}\right) \times VCC \tag{6}$$

$$V_{IL} = 0.7 \times VCC = \left(1 - e^{-t_2/R_P C_B}\right) \times VCC \tag{7}$$

From the exponential settling equations, the rise time can be solved in terms of the maximum pullup resistance and the bus capacitance. In this example, the calculation is for a 400-pF bus capacitance (for the maximum bus capacitance) and supply voltage of 3.3 V. Equation 8 and Equation 9 solve for the rise time and then the maximum pullup resistance.

$$t_{RISE} = t_2 - t_1 = 0.8473 \times R_P \times C_B \tag{8}$$

$$R_P(max) = t_{RISE} / (0.8473 \times C_B) \tag{9}$$

The rise time is dependent on the I²C mode. For this example, the standard mode can be used. Take the rise time of 1000 nanoseconds and divide by the quantity of 0.8473 times 400 pF. This gives a maximum resistance of 2.95 kΩ.

With a minimum resistance of 967 Ω and maximum resistance of 2.95 kΩ, these values appear to give a narrow range for the resistance. However, this small range is because the pullup resistor sizing is calculated to operate with the maximum standard-mode bus capacitance of 400 pF. This amount of bus capacitance is unusually large especially for a parasitic capacitance on the board. If the design has a lower bus capacitance (which is likely), the maximum resistance can be increased, reducing the power dissipated on the I²C bus.

For a more detailed description of I²C pullup resistor calculations see the *I2C Bus Pullup Resistor Calculation* application report. An I²C pullup calculator is also found in the *Analog Engineer's Calculator*.

7 Protocols Similar to I2C

The I2C specification discusses several other communications protocols based on I2C. These other protocols can be similar and compatible with I2C communication and can be used for specific applications. Some protocols also have defined sets of commands and application-specific extensions for their systems. This section briefly describes the applications for these other protocols, but their systems, applications, and uses are found elsewhere.

The first of these similar protocols is the Two Wire Interface or TWI. This protocol is basically the same as I2C. However, there are some minor differences in that TWI does not support a START byte and does not support high-speed modes. Generally, TWI-compatible devices are expected to be compatible with I2C and the protocol can be seen with the same logic analyzers.

The System Management bus or SMBus® is a protocol similar to I2C that is tailored to a specific function. SMBus is commonly used in servers and computer motherboards for power source management. The protocol is very similar to I2C in the communication protocol, and can be understood by an I2C controller.

The SMBus protocol has some additional features in comparison to I2C. The SMBus can dynamically set addresses, allowing for quick communications at the start up of a system. Also, the bus has a 35-ms timeout which prevents one device from indefinitely tying up the bus. The protocol also has a packet error checking for error detection in data communication. There is an additional line called SMBAlert that is used by target devices as an interrupt to tell the controller about certain events detected by the target device.

The Power Management bus or PMBus® is a variant of SMBus defined by Intel. PMBus is used in the digital management of power supplies. This protocol also defines specific commands to retrieve data about voltage, current, and power in the system.

Intelligent Platform Management Interface or IPMI is another I2C-based protocol. This protocol is used by baseboard management controllers (BMC) for autonomous computer subsystems for monitoring and management of the system CPU, firmware, and operating system. The protocol uses a standardized message-based interface for a computer motherboard or server. The BMC is always running even when the main system is off. This allows for operation, measurement, and remote management of a system.

There are several other similar protocols discussed in the I2C specifications. Advanced Telecommunications Computing Architecture (ATCA) is a follow-on to Compact PCI and is used in rack-mounted telecom hardware. Display Data Channel (DDC) is a monitor or display information protocol used by hosts for control of display functions. Finally, C-Bus is another protocol that is derived from I2C. As mentioned in the reserved address section, this protocol is used in some parts of the world for home and building automation, but is generally not used in the United States.

8 Summary

I2C is a common digital communication standard used in a wide variety of products. The protocol uses a two-wire communication interface that allows for multiple controllers and multiple target peripheral devices. This application note described many important aspects of the protocol as a guide to using I2C to communicate with controller devices.

This application note discussed both the protocol and the physical layer for I2C communications. Because I2C are often used with data converter devices, examples of communications were provided for a DAC and an ADC for both writing to and reading from registers. This note also covered many not-so-common aspects of the protocol, such as clock stretching, fast-mode, and clock stretching which are not commonly implemented in devices.

The topics presented here were not all covered in depth. However, this application note provides system designers with a working knowledge of the protocol. Using this information, designers can set up their I2C systems and debug them when there are communication problems.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265