Application Note

I2C Stuck Bus: Prevention and Workarounds

Duy Nguyen

ABSTRACT

This document shows the I2C stuck bus glitch, how I2C stuck buses occur, and potential ways to resolve the I2C stuck bus glitch which includes a software approach and a hardware design.

Table of Contents

Trademarks

1 What is a Stuck Bus?

An I2C stuck bus is when the SDA line gets held low indefinitely while the SCL line is high. This presents a problem because typically there are multiple I2C devices on the bus and if the SDA line were to get stuck low, then I2C communication can no longer be possible. In this situation the I2C controller cannot issue a start or stop condition because the I2C controller cannot control the SDA line. In the worst case scenario, even the processor can get stuck in a state where the processor is waiting on the SDA line to go high. This can result in the entire system or end equipment getting stuck because the processor is no longer executing any other lines of code since the processor is waiting indefinitely. This document can focus on this specific type of I2C stuck bus since there are methods to try to resolve this kind of stuck bus.

There is also another type of I2C stuck bus event in which the SCL line can get stuck low. Generally the only device on the I2C bus that can control the SCL line is the I2C controller which issues the SCL pulses. However there are some I2C target devices which can execute clock stretching and in rare cases can potentially get the clock stuck low. In these cases, the only recovery method can be a reset or power cycle of the device sticking the bus low.

2 How can a Stuck Bus Occur?

A stuck bus can occur in several different ways, the main offender for most cases is a false clock edge being generated. The reason why a false clock edge is dangerous on the I^2C bus is because a false clock edge desynchronizes the I^2C target's clock relative to the I^2C controller, who is responsible for generating the clock edges. Differences between the I^2C target device ns deglitch filter and the I^2C controller's deglitch filter can contribute to one device seeing the false edge while the other does not. An example is if the I^2C target has a 70ns deglitch filter while the controller has a 50ns deglitch filter and a false edge occurs with a 60ns window which makes the controller see the edge but the target ignores the false edge. Another potential case is if there is an I^2C redriver (also known as a buffer) between the target and the controller. If a false edge occurs on one side, it cannot propagate through the redriver so either the controller or the target can see the edge while the other does not.

An example is shown in Figure 2-1. The first transaction in this example shows what is supposed to happen. The controller is sending data to the target with the data package of 0x0Ah and at the 9th clock pulse the target ACKs to tell the controller that the target receives the data then a stop condition is generated by the controller. The second transaction is the same as the first, however a false edge is generated during the 5th clock pulse of which the controller sees but not the I^2C target. This means the controller is one clock ahead in the I^2C transaction while the I^2C target is one clock behind. The data the target sees is then shifted to the left by one so it receives bad data (0x05h) instead of seeing 0x0Ah. When the 8th falling SCL edge comes (from the target's perspective), the I2C target drives the SDA line low but never sees the 9th falling SCL edge. This results in the SDA line getting stuck low indefinitely. The controller cannot issue a stop condition and depending on the controller's hardware and software, can keep SCL low or release SCL.

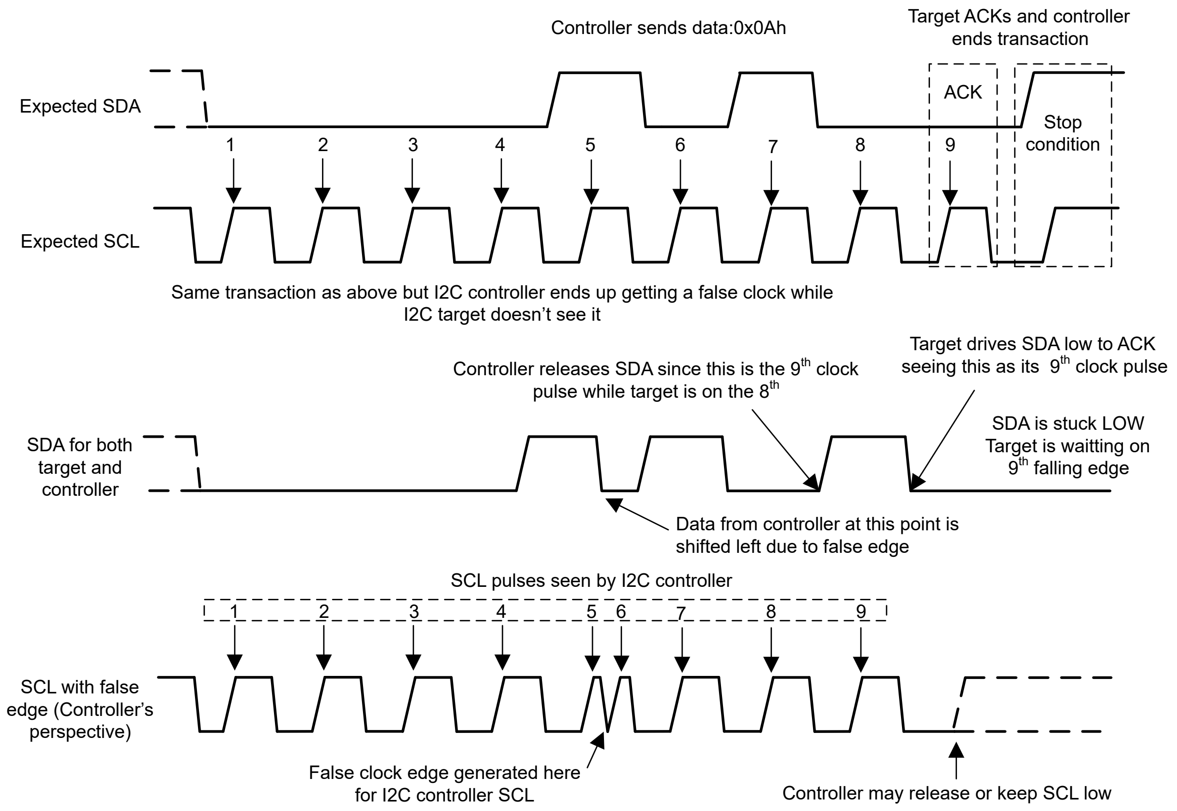
Controller sends data:0x0Ah
Target ACKs and controller ends transaction
Expected SDA
ACK
Stop condition
1
2
3
4
5
6
7
8
9
Expected SCL
Same transaction as above but I2C controller ends up getting a false clock while I2C target doesn't see it
Controller releases SDA since this is the 9th clock pulse while target is on the 8th
Target drives SDA low to ACK seeing this as its 9th clock pulse
SDA for both target and controller
SDA is stuck LOW Target is waitting on 9th falling edge
Data from controller at this point is shifted left due to false edge
SCL pulses seen by I2C controller
SCL with false edge (Controller's perspective)
False clock edge generated here for I2C controller SCL
Controller may release or keep SCL low

Figure 2-1. Example of False Edge on Controller During Write Transaction

Root causes for a stuck bus can come from crosstalk, electromagnetic interference, hot-insertion events, or bad power on reset situations.

3 Crosstalk

A common way a false edge can be generated is through crosstalk. Typically, in terms of I2C, crosstalk is generated by shared capacitance between two lines (imagine two long parallel running traces on a PCB with little separation) and a fast edge rate. Normally if an I2C line experiences crosstalk, the crosstalk event occurs when either the SDA or SCL line is in a logic HIGH state. This because the signal is more strongly biased when driving a low logic due to the open drain architecture of I2C . A logic high state is set by a pull-up resistor which is much weaker in comparison to an open drain driver. R_{on} of these open drain drivers are much stronger (have a low value R_{on}) than the external pull-up resistor to establish the logic high signal. This means in most I2C transactions, the SDA line can be the signal that sees crosstalk since the SDA does not change the state unless SCL is low. The only time this is not true is during a start condition therefore this can be a potential area where a false clock edge can occur if severe enough. If a stuck bus is occurring and crosstalk appears to be present on the SDA line, then checking the beginning of the I2C transaction at the start condition is recommended to rule out crosstalk as the culprit.

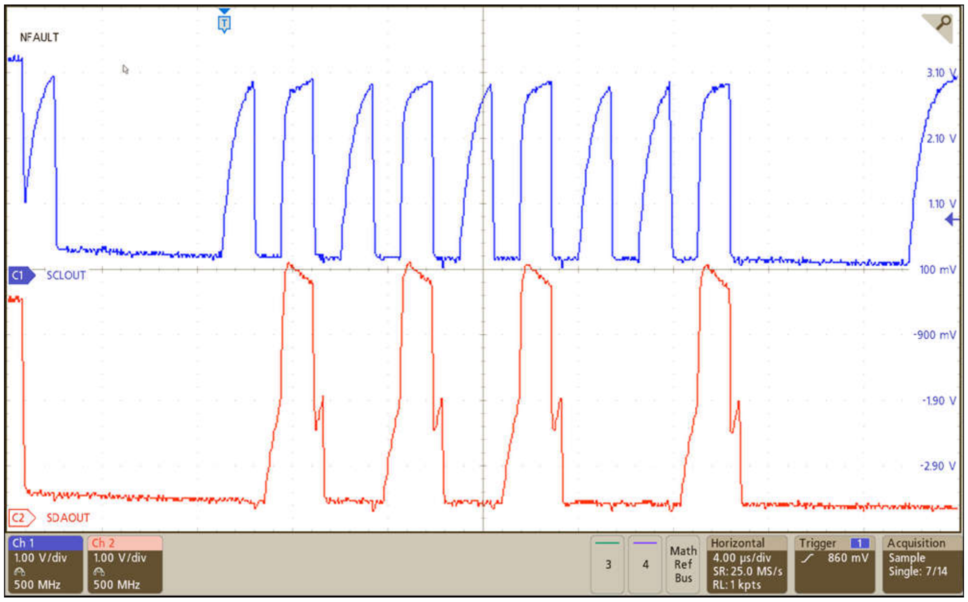

Figure 3-1. Example of Crosstalk

This does not mean the clock line isn't at risk after a clean start condition is issued, other faster edges near the SCL trace can still induce crosstalk onto the SCL bus during the high period. One of the main takeaways here is to properly layout the I2C traces such that SCL is not too close (parallel) to other traces which have fast edge rates (including SDA). Parallel can also mean high frequency or high current driven traces that run directly beneath the SCL trace.

4 EMI

Electromagnetic interference, also known as EMI, is another potential source for false edges or data corruption to occur. The concept behind this is that the SDA/SCL PCB traces and even the leaded legs of I2C device's packages can pick up EMI because the devices are made of electrically conductive material, metals, which can act as antennas to pick up this electrical noise. Generally, this kind of concern is present in electrical noisy environments such as a power stations with high switching currents which is emitted through the air. Other potential EMI sources can also pose a problem. High current loads firing off like in a laser printer or a motor in fan can emit large amounts of EMI. I2C traces near these EMI sources can see signal integrity issues and, in some cases, cause errors with the hardware peripherals within the I2C controller or I2C target since these devices weren't designed to see this kind of electrical stress. If the magnitude of the EMI is large enough and frequency is slow enough a stuck bus event can trigger on the I2C bus.

5 Hot Insertion

Hot insertion, sometimes called hot swap or hot join, is a term used in the industry to describe an event in which an non-powered PCB, is connected to a powered PCB, also known as the backplane. When a live I2C bus (the one that is already powered) sees an non-powered I2C bus (PCB), there can be a mismatch voltage levels. For example, if the live I2C bus is communicating where SCL and SDA are both logic highs, then when the non-powered I2C bus makes connection the introduction non-powered parasitic capacitance onto the bus can force both SDA and SCL to take a quick dip to GND to charge the newly introduced parasitic capacitance from the non-powered board. An example of this is shown scenario is shown in the figure below. Channel 4 (green) is the SCL pin on the live bus (backplane) and channel 3 (purple) is the SCL line on the non-powered I2C Bus (daughter card). The live bus SCL drops to match the current voltage of the daughter card's SCL voltage level due to the parasitic capacitance added to the live bus. Note that this image showcases a 1V pre-charge feature of TI's I2C hot swap devices so the voltage drop only drops down to 1V instead of ground.

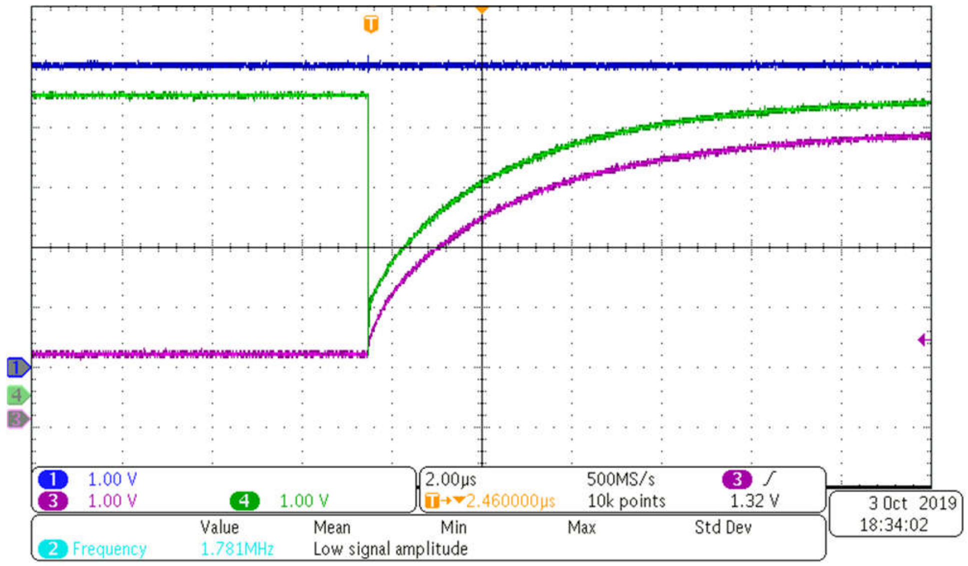

Figure 5-1. Hot Insertion Example

The parasitic capacitance voltage drop event can look like an additional false clock pulse on the I2C bus which can corrupt the transmitted data and in the worse case scenario bring about a stuck bus condition.

6 Resolving a Stuck Bus

SDA being stuck low is problematic when information is required to be read or data needs to be transmitted across the bus. There are several possible methods to unstick a bus. If the system designer of the bus has access to modifying or building the software of the processor or host on the bus, they can be able to code the processor or host to look for a potential stuck bus and react to try to resolve it. For an I2C bus, this can be done by toggling the serial clock line (SCL) eight to sixteen times and then issuing a stop condition. This can push the I2C device's state machine that is holding the serial data line (SDA) low through its process flow and reset its state machine to an idle state releasing the bus. For the SMBus, the SCL line can be held low for longer than the t_timeout, which can force all SMBus devices to all reset and resolve the stuck bus. If the I2C target devices have a dedicated reset pin, a processor on the bus can initiate a reset to unstick the bus. This however is not an efficient design if there are a large number of I2C target devices. The processor can be forced to reset all target devices connected to this reset pin, therefore causing the re-initialization of all I2C targets on the bus, although only one device caused the stuck bus condition in the first place.

From a hardware approach, using an I2C buffer device with hot insertion protection can be used to remedy a stuck bus resulting from a hot insertion event. TCA9511A and TCA4307 include dedicated logic to separate the non-powered I2C bus (daughter card) to the powered I2C bus (backplane) until both sides are ready to be connected, preventing glitches on the bus. This safe connection mechanism is provided in more detail in *I2C Solutions for Hot Swap Applications*, application note.

To try to resolve a stuck bus from crosstalk, EMI, or a bad power on reset, the TCA4307 can be designed into an I2C bus as it includes an additional feature called stuck bus recovery. The stuck bus recovery feature can detect if the SDA line is stuck low. If the device is stuck low for longer than 40ms (t_stuckbus), it then disconnects the downstream I2C bus from the upstream I2C bus and drives the RDY pin LOW. This lets the processor know that the downstream channel has been disconnected from the bus. This prevents the upstream bus, where the I2C controller/processor typically resides, from getting stuck and allows it to continue communicating to other I2C devices on the upstream bus segment. The TCA4307 then can generate up to sixteen clock pulses until the SDA line unsticks. The TCA4307 can then issue a stop condition to attempt to reset the I2C device's state machine to its idle state. If the stuck bus is resolved, the TCA4307 can reconnect the downstream bus and the upstream bus automatically.

This can be seen in the figure below. Channel 1 (blue) is the SCL line, channel 4 (green) is the SDA line, channel 3 (purple) is the RDY pin which signifies when the downstream bus and upstream bus are connected together again. Figure 6-1 shows an example where the bus gets unstuck by the TCA4307 after 6 clock pulses and then TCA4307 controlling the SDA line to issue a stop condition.

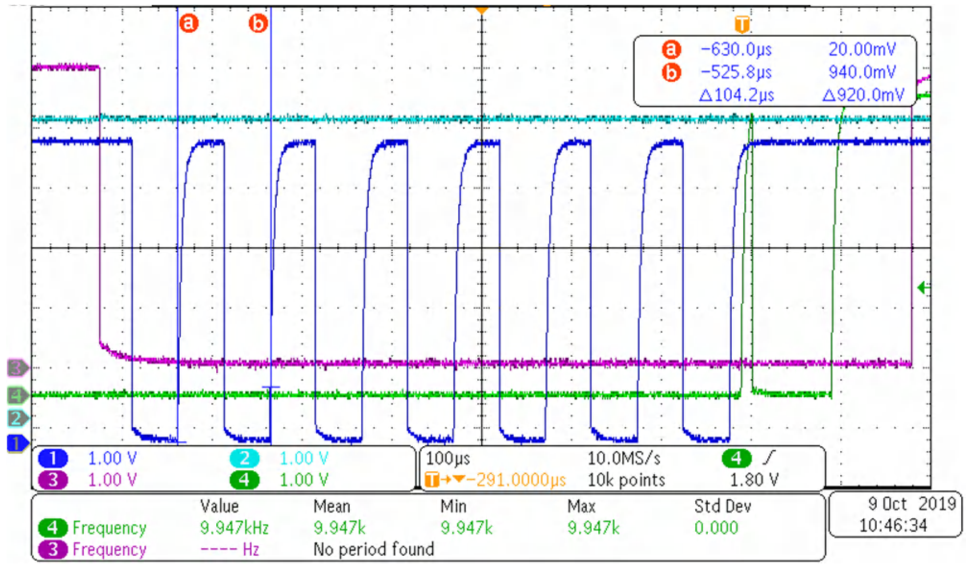

Figure 6-1. Waveform Example of the Stuck Bus Recovery Feature

7 Conclusion

I2C stuck buses can occur from a multitude of sources and prevent crucial information to be passed on the I2C bus or prevent control of I2C devices. Proper software/hardware design can and can be used to protect the I2C bus from ever getting hung and stopping I2C operation.

8 References

- Texas Instruments, *I2C Solutions for Hot Swap Applications*, application note.
- Texas Instruments, *TCA4307 Hot Swappable I2C Bus and SMBus Buffer with Stuck Bus Recovery* data sheet.
- Texas Instruments, *TCA9511A Hot Swappable I2C Bus and SMBus Buffer*, data sheet.