Validating and using the I2C protocol

Vera Apoorvaa - July 15, 2014

I2C is a two wire, clock synchronized protocol with a bi directional data line and a uni directional clock line. Its simplicity lies in its use of only two lines for communication and its complexity lies in the fact that these lines are shared among all the devices on the bus. The I2C bus can have several masters and slaves connected on the same two lines and bus arbitration is employed to handle bus contentions. The scope of this article is to bring out some common I2C issues that come up while validating and using the I2C protocol.

Bus Contention and Arbitration – An Overview

When a master sends data on the bus, it also repeatedly probes the bus to ensure the data it sent out is actually driven on the bus. When two masters drive data simultaneously on the bus (let us assume a simple case wherein both masters are operating at the same bit rate), all goes well until the data bits driven by both masters are the same. When the data bit they drive differ, that is, say master A drives a dominant bit (read '0' for an open drain drive low bus) and master B drives a recessive bit (read '1' for an open drain drive low bus), the resultant bus state is dominant ('0'). Master B senses it has lost arbitration and backs away from sending the rest of the message bits. Master A completes its transaction with no intervention. In fact, master A has no clue that master B has been contending for the bus (figure 1). In summary, all devices on the bus are capable of driving data on the bus and are listening to the bus. Hence, it is important that all devices function in compliance and misbehaving devices be identified and isolated.

Figure 1. Master A gains arbitration while master B loses arbitration

I2C Validation and Debug

A qualitative validation strategy aims at determining what can go wrong on the I2C bus rather than what can go right. The foremost task is to understand the scenarios that can go wrong. The next step is to determine the expected behavior of the device under test (DUT) and finally and most importantly, devise a method to generate these error conditions. After all, implementation feasibility determines testability. In the course of I2C validation, I have come across numerous issues and wish to share some of them further in this article. **Typical device combinations**

Some typical device combinations on the I2C bus include:

1. Single slave and a single master
2. Multiple masters and a single slave
3. Single master and multiple slaves
4. Multiple slaves and multiple masters

Case 1 is quite simple and not many issues are expected for a fairly well designed master/slave device. Cases 2, 3, 4 involve more than one device sharing the bus and hence it is important that the devices are well designed not to misbehave as well as are capable of recovering from error scenarios.

Interpreting a bus stuck low scenario

The SDA/SCL line being stuck low (figure 3 and 4) is a communication impairing condition. When this happens, no master can generate a start condition (figure 2) and hence disabling any further communication. This in most cases requires reset of the device holding the bus low or master to send the bus clear sequence to recover the I2C bus.

The first step to debug the bus stuck low is to identify the device holding the lines low. Each device can be unplugged from the I2C bus one at a time and the bus state observed until both SCL and SDA goes high. In case the I2C lines of each device cannot be disconnected individually, each device on the bus can be reset until the bus goes high, isolating the device that was holding the bus low. Device reset is of course the less preferred choice.

Figure 2. I2C Start condition

Figure 3. SCL stuck low condition

Figure 4. SCL and SDA stuck low condition

What can cause a bus stuck low condition

Let us assume the master is driving the SDA lines with data bit ‘0’ and is reset/ turned off at this point. It is probable that several devices are not well designed to release the SCL/SDA lines before reset/while turned off.

Another probable condition

Another probable condition is, say the master is turned off/reset while sending the 9th clock pulse (figure 5). At this bit position the slave has control over the SDA line and is driving an ACK on the bus. Now, no further clocks are driven by the master. But the slave is holding the SDA low and waiting for the next clock edge to release SDA from the ACK state. This could cause a bus stuck low condition. To retrieve the bus from such a condition, a bus clear can be send by any other master on the bus to ensure the slave releases the SDA line.

Figure 5. Master turned off/reset while driving the 9th clock pulse

Data Corruption

I2C as such does not employ an error correction, detection or even a parity check mechanism. Hence any suspicion of data corruption only arises when an application using I2C does not perform as expected. The data corruption could be due to the application layer of the system design or could be something to do with I2C itself.

I consider I2C data corruption could occur at two levels:

- *At the physical layer* - Glitches/power transients might cause data corruption on the I2C bus. Debugging this would need a waveform check on the oscilloscope or a better option is to use a protocol analyzer to verify the data on the I2C bus. If the glitch/transient disrupts the I2C frame structure, it can be easily observed as a corrupt transaction with the use of an analyzer (figure 6). If a particular bit/bits alone are corrupted, then the I2C packet will still be captured as a valid packet and we would require to analyze the data captured to determine if it corrupted (unexpected data).

Figure 6. Corrupt transaction captured by a protocol analyzer

- *At the receive buffer of the device* - the slave/master buffer that stores the I2C message could get corrupted resulting in wrong data. This would require a debugger to check the data at the receiving end.

Multiple slaves with the same address

Multiple slaves with the same address

Assume we are expecting particular data from a slave, but for some reason it does not match what is expected. A simple reason could be because there is more than one slave configured for the same address. This could result in all slaves with the same address sending data since each of them assume they are addressed. The resultant data is an AND of the data sent by all the slaves.

Differentiating data driven by different devices

Many a times, a master might detect a protocol error on the I2C bus. There could be another device on the bus driving the bus with a dominant logic at a wrong bit position (unexpected bit position) while another device is doing a transfer. Such a condition can be captured on the oscilloscope but often is not easy to determine which device on the bus drove this erroneous data. One way this can be found out is to reduce the pull up resistance value so that you can see a differentiating voltage level on the bus.

Figure 7. Differentiating the devices driving the bus

This happens since the drive current for different slaves/masters could be a little different causing the dominant bit (read ‘0’) to be of slightly different levels. This is an extremely useful tip to identify/detect a misbehaving slave.

I2C protocol validation as such has numerous interesting facets, every new issue brings about a new dimension in enhancing the validation suite and the possibilities of increasing the scenarios keeps increasing until the validation engineer decides to stop on his/her own.

More about author Vera Apoorvaa