Debugging Common USB Issues

Total Phase, Inc.

The widespread integration of USB into embedded applications presents many developers with the challenge of working with this protocol for the first time. The complexity of USB can mask problems, leaving issues that are difficult to detect and isolate when a USB device misbehaves. USB protocol analyzers allow engineers to gain greater visibility into the bus to help them quickly pinpoint the exact nature of a bug, whether it is in the physical, electrical, or protocol layer. In this article, we will explore how a USB protocol analyzer can be used to debug potential problems encountered in USB development.

The Case for USB Protocol Analyzers

Faced with a wide selection of debugging tools such as logic analyzers, oscilloscopes, and protocol analyzers, finding the ideal debugging tool can be a daunting task. Fortunately, the complexity of USB guides the choice of an ideal debugging solution.

As a result of the USB protocol's complexity, tools like logic analyzers or oscilloscopes may be limited by their low-level view, making it difficult to sort through large amounts of serial data. In contrast, protocol analyzers are able to non-intrusively monitor the bus, view data as packets, and capture higher-level protocol-specific data in large volumes.

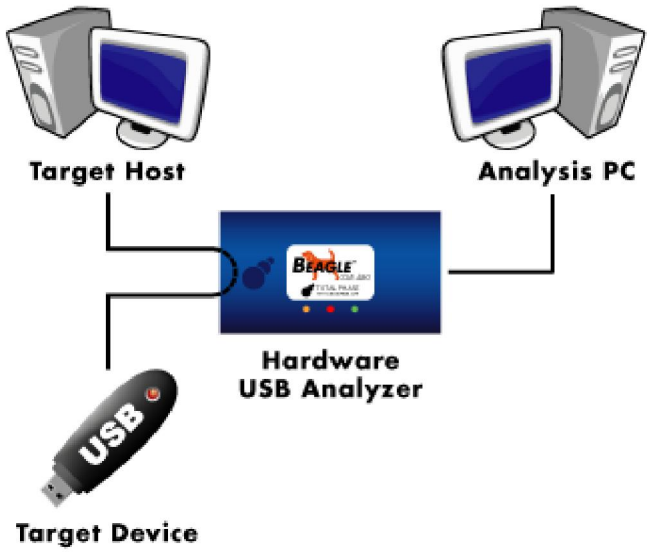


Figure 1. Typical device configuration for capturing USB data.

The setup for capturing USB data is a straightforward process. In Figure 1, the USB Analyzer is connected in-line between the Target Host and the Target Device to non-intrusively capture data. As communication between the host and device begins, the data is sent immediately to the analysis computer which runs the capture software for real-time display and filtering.

In contrast to scopes and logic analyzers, the USB capture software can display detailed information such as timestamp, device and endpoint address, packet identifiers (PIDs), and

data in a human-readable format. The software also includes search and/or filter features to help developers quickly locate data of interest within a large amount of data.

We will now look at some examples of how a USB protocol analyzer such as the Beagle USB 480 Protocol Analyzer (Figure 1) can be used to help identify common problems in USB development.

USB Data Validity

USB employs two error checking methods to ensure that data is sent correctly. A cyclic redundancy check (CRC) is sent with all data transmissions to validate data integrity within a packet. In addition, a toggle bit is encoded in the packet identifier (PID) of the data packet to ensure that packets are sent in the correct sequence. Correct data sequencing is especially important when attempting to transfer large files across multiple independent USB transmissions.

Normally when transferring data over multiple packets, the data PID will toggle between DATA0 and DATA1 on each consecutive successful transmission. Specifically, as data is successfully transmitted (i.e. CRC is valid), the receiver acknowledges (ACK) the data and both transmitter and receiver toggle their DATA bit. If there is a data error and the CRC check fails, however, the receiver will not reply with an ACK, and the transmitter is required to resend the data with the same toggle bit. The transmitter will continue to resend the same data with the same toggle bit until the receiver ACKs its reception.

In some cases, the data is sent correctly but the ACK handshake gets corrupted on the bus. When this occurs, the receiver thinks that the data was sent properly and updates its toggle bit, but the transmitter does not actually know if the data was received correctly. Therefore, the transmitter will send the same data with the same toggle bit. Since the toggle bit has not changed, the receiver assumes that this is a re-transmission of the same data, and silently ignores the data. The receiver will then ACK, causing the transmitter's toggle bit to update correctly.

Finding Problems in Data Bit Toggling

Incorrect handling of the toggle bit is a common USB problem that is hard to identify, since the symptoms may not necessarily render a device unusable. A device may simply appear to have a reduced throughput or individual data transmissions may be dropped. Without the aid of a hardware protocol analyzer, it is nearly impossible to deduce that improper data toggling is the cause of the problem.

To illustrate this issue, we will look at a situation where a host-side application is failing to receive any data from the device. To help in the debugging process, the device has been configured to send a counter value which is updated with each successful transmission. The root of the problem could be traced to a variety of bugs related to firmware, software, and/or hardware.

The use of a hardware protocol analyzer can quickly pinpoint this type of error. In Figures 2 and 3, data is being captured from two devices; one is functioning properly and the other is not. In both cases, it is clear that low-level hardware is functioning correctly, as valid data is

being transmitted without CRC errors. To aid in the analysis, software display filters have been used to display only DATA packets in Figures 2 and 3.

Sp	m:s.ms.us	Len	Err	Dev	Ep	Record	Data
HS	0:00.100.011	7 B		01	02	DATA0 packet	C3 00 00 00 00 FF DB
HS	0:00.100.023	7 B		01	02	DATA0 packet	C3 01 00 00 00 FE 27
HS	0:00.100.035	7 B		01	02	DATA0 packet	C3 02 00 00 00 FE 63
HS	0:00.100.046	7 B		01	02	DATA0 packet	C3 03 00 00 00 FF 9F
HS	0:00.100.058	7 B		01	02	DATA0 packet	C3 04 00 00 00 FE EB
HS	0:00.100.070	7 B		01	02	DATA0 packet	C3 05 00 00 00 FF 17

Figure 2. The consecutive DATA0 PIDs in the Record column show that the device is not toggling its data PID.

Sp	m:s.ms.us	Len	Err	Dev	Ep	Record	Data
HS	0:00.100.011	7 B		01	02	DATA0 packet	C3 00 00 00 00 FF DB
HS	0:00.100.023	7 B		01	02	DATA1 packet	4B 01 00 00 00 FE 27
HS	0:00.100.035	7 B		01	02	DATA0 packet	C3 02 00 00 00 FE 63
HS	0:00.100.046	7 B		01	02	DATA1 packet	4B 03 00 00 00 FF 9F
HS	0:00.100.058	7 B		01	02	DATA0 packet	C3 04 00 00 00 FE EB
HS	0:00.100.070	7 B		01	02	DATA1 packet	4B 05 00 00 00 FF 17

Figure 3. The alternating DATA0 and DATA1 PIDs show proper toggling between subsequent packets.

Upon inspection, it is obvious that one device did not toggle the data PID (Figure 2) while the other one did toggle the data PID after each packet (Figure 3). As discussed earlier, sequential DATA0s should not be passed to the application because the receiver will ignore packets that are repeating the same toggle bit. This explains why data is not being passed to the application. However, the reason why the same toggle bit is being used is still unknown.

To investigate this issue further, we can examine the entire transaction sequences (Figure 4). In this view, it is clear that the transaction is completing successfully because the capture shows the ACK for each data packet, but the DATA bit is not actually toggling. Furthermore, transmissions that use sequential toggle bits are supposed to resend identical data. In this case, the device is not actually resending the data, but continues to update its counter. The error must therefore lie in the handling of the DATA toggle within the device. Specifically, the firmware is not toggling the bit on each successful transmission. Without a hardware protocol analyzer, this small mistake could cost days or weeks of a developer's time.

Sp	m:s.ms.us	Len	Err	Dev	Ep	Record	Data
HS	0:00.100.010	4 B		01	02	IN txn	00 00 00 00
HS	0:00.100.010	3 B		01	02	IN packet	69 01 C1
HS	0:00.100.011	7 B		01	02	DATA0 packet	C3 00 00 00 00 FF DB
HS	0:00.100.013	1 B		01	02	ACK packet	D2
HS	0:00.100.021	4 B		01	02	IN txn	01 00 00 00
HS	0:00.100.021	3 B		01	02	IN packet	69 01 C1
HS	0:00.100.023	7 B		01	02	DATA0 packet	C3 01 00 00 00 FE 27
HS	0:00.100.025	1 B		01	02	ACK packet	D2
HS	0:00.100.033	4 B		01	02	IN txn	02 00 00 00
HS	0:00.100.033	3 B		01	02	IN packet	69 01 C1
HS	0:00.100.035	7 B		01	02	DATA0 packet	C3 02 00 00 00 FE 63
HS	0:00.100.036	1 B		01	02	ACK packet	D2

Figure 4. An expanded view of transaction shows new data being sent with the same DATA toggle bit.

Low-Level Bus Events

Another common error occurs with low-level bus events. The USB specification defines a number of crucial bus-level signaling events that follow a specific timing criteria and determines such things as suspend, resume, and the high-speed handshake. As an example, one type of low-level bus event error could result in the host's failure to begin the enumeration process, ultimately failing to recognize the USB device. These types of events are difficult, if not impossible, to debug without the aid of a hardware tool.

One step of the high-speed handshake, called the chirp sequence, requires the host to issue at least 3 cycles of alternately driven D- (Chirp K) and D+ (Chirp J) for 40-60 us. Even though the USB specification only requires 3 cycles, hosts will often send hundreds of these cycles. While it is possible to use a scope to measure this sequence of events and calculate durations with the cursors, utilizing engineering time to verify each chirp with a scope would be a tedious and error-prone process. A hardware protocol analyzer, in contrast, can save time by automatically measuring each signal and indicating potential errors. Furthermore, with an analyzer, these measurements can be done on every test run, thus automatically catching new or intermittent bugs.

In Figure 5, the capture software highlights an error with a mis-timed chirp sent from a USB host that is under development. In this particular example, the mistake occurred halfway through the chirp sequence, and a cursory look on a scope would have missed it. The consequences of such an error are undetermined as this is out of spec, and any number of processes could malfunction. While it is possible that a device is robust to this situation and can continue to function properly, other devices may be more sensitive to the out-of-spec signaling. If internal testing was only run with a small subset of devices that all passed, a developer without a hardware protocol analyzer may never know that such an error exists. A malfunctioning product could be sent out to the field, where the error would be exposed later to the frustration of many customers. By having a hardware-based analyzer, engineers can avoid escalated costs due to unforeseen errors.

Sp	m:s.ms.us	Len	Err	Dev	Ep	Record	Data
FS	0:00.107.526	50.0 us				<K chirp>	
FS	0:00.107.576	50.0 us				<J chirp>	
FS	0:00.108.026	30.0 us	U			Bus event	<Reset> / <K chirp> / <Tiny K>
FS	0:00.108.056	50.0 us				<J chirp>	
FS	0:00.108.106	5.05 ms				[50 K-J Chirp Pairs] [1 K Chirp]	
FS	0:00.108.106	50.0 us				<K chirp>	
FS	0:00.108.156	50.0 us				<J chirp>	
FS	0:00.108.156	50.0 us				<J chirp>	

Figure 5. The highlighted row shows an unexpected bus event halfway through the chirp sequence.

The development and debugging stage is a crucial step in the product life-cycle. As the situations above illustrate, the use of a hardware-based USB protocol analyzer presents the data packets of a complicated protocol such as USB in an accessible and human-readable

format. Using an analyzer, engineers can easily test their applications and quickly identify problem areas while reducing development time and simplifying the debugging process.

About Total Phase

Total Phase is a leading provider of embedded systems development tools for engineers all over the world. Total Phase's mission is to create powerful, high-quality, and affordable solutions for the embedded engineer. For years, Total Phase has developed products that have become the tools of choice for companies of all sizes, from Fortune 500 companies to small business alike. Our satisfied customers represent a diverse array of industries such as automotive, consumer electronics, medical devices, semiconductors, and more.