USB Error Handling For Electrically Noisy Environments

Introduction

In order to provide robust operation, USB device drivers must process completed URBs to detect and handle errors appropriately. This application note focuses on handling errors due to electrically noisy environments that may cause USB requests to be retired due to too many timeout transmission errors.

Background

As defined in section 10.2.6 in all revisions of the USB specification, a timeout condition occurs “when the addressed endpoint is unresponsive or the structure of the transmission is so badly damaged that the targeted endpoint does not recognize it.” When the latter scenario occurs, the correct response for the targeted endpoint is to not return a handshake response.

The host controller maintains an error count for all transactions of all endpoint types, except isochronous. The host controller increments that error count whenever a transmission error occurs. When the error count is incremented to three, the host controller retires the transfer and provides the error information corresponding to the last transmission error. When a transaction succeeds, the error count is reset to zero. A NAK is considered neither a transmission error nor a successful transaction as the transaction is still pending afterwards.

When a device detects a corrupted packet, the correct response is to ignore the transaction. The host controller then increments the error count and retransmits the transaction. If this happens three times within a transaction, the transaction times out and the host retires the transaction. In the current Windows USB stack, the UHCI/OHCI driver will set the URB status to USBD_STATUS_DEV_NOT_RESPONDING.

Devices that NAK IN or OUT PIDs for long periods of time are susceptible to the PID being corrupted by electrically noisy environments. A real world example that may be particularly susceptible would be a hub or a hid device. The host controller issues an interrupt IN request every *bInterval* frames. If the endpoint does not have data to return, the endpoint must NAK the IN request. Since such devices will continuously NAK a single transaction a large number of times, these devices are particularly susceptible to electrically noisy environments that may corrupt three IN requests within that transaction and cause the transaction to be retired.

Therefore, device drivers must monitor the status value when the URB has completed and take the necessary steps to reestablish data transmissions.

Considerations with USB 2.0

Device drivers of low-speed and full-speed devices need to be prepared for the characteristics of the USB 2.0 specification. When a full or low-speed bulk or control transaction times out behind the transaction translator of a high-speed configured USB 2.0 hub, the transaction translator will issue a STALL response to the complete split transaction from the EHCI controller. This behavior is defined in section 11.17.1 of the USB 2.0 specification. For interrupt and isochronous endpoints, an ERR handshake PID is used to indicate a transmission error.

For full-speed bulk and full-speed/low-speed control endpoints whose transactions timeout, the EHCI host controller driver will most likely set the URB status to USBD_STATUS_ENDPOINT_HALTED. Therefore, a device driver cannot distinguish between a timeout and an actual endpoint HALT without issuing a get endpoint status command.

For full-speed and low-speed interrupt transfers whose transactions timeout, the author speculates that the EHCI host controller driver will set the URB status to either USBD_STATUS_ENDPOINT_HALTED or, USBD_STATUS_DEV_NOT_RESPONDING. However, it is possible that some other value may be used. Device drivers that implement this sort of error handling may need to be updated when Microsoft® releases USB 2.0 support in their operating systems.

Solution

Since a device driver is not guaranteed to be able to distinguish between a timeout and a STALL condition for bulk and control transfers, the driver may issue a get endpoint status command, and if necessary, a reset pipe request prior to reissuing the failing USB transfer. Since a minimum of one request must be issued, it is recommended to simply issue a reset pipe request and then reinitialize and resubmit the failed URB.

If the failed URB was a blocking URB issued at PASSIVE_LEVEL (i.e., the driver issued the URB when running at PASSIVE_LEVEL and waited until the URB completed), then the device driver simply needs to issue a blocking ResetPipe URB, reinitialize the failed URB, and resubmit the URB to the USB stack.

If the failed URB is processed in a completion routine running at DISPATCH_LEVEL, then the recovery mechanism is more complicated. The device driver must create a worker thread to issue the recovery steps (i.e., a system thread or a work item). The reason for this is due to the fact that ResetPipe URBs must be issued at PASSIVE_LEVEL. The worker thread must then issue a blocking ResetPipe URB, reinitialize the failed URB, and resubmit that URB to the USB stack.

Conclusion

Due to uncontrollable situations, such as electrically noisy environments, USB device drivers must implement error handling to provide robust operation. By implementing the techniques provided in this application note, device drivers will provide overall better usability to the end consumer.