

SpiderMesh Integration Guide

Mesh EOB parallel-polling strategy
caveats and mitigation strategies

smartrek

Contents

1	Abstract	4
2	The EOB Strategy	4
3	Failure Mode A	8
4	Failure Mode B	8
5	Failure Mode C (hybrid)	11
6	What are the typical bottleneck/time delays/jitter sources, and how to mitigate those?	13

List of Figures

1	Proper operation of a typical computer-driven direct-driven EOB strategy	5
2	Failure mode A: TR 2 RF overspill	10
3	Failure mode B: EOB desynch	10
4	Next broadcast cycle starts	11
5	Failure Mode C	12

List of Tables

1	Glossary	6
2	Glossary (continued)	7
3	Basic Time Interval Variable Relationships	9

1 Abstract

The Smartrek Mesh EOB parallel-polling feature allowing speed optimization of the mesh, a.k.a. the End-Of-Broadcast marker (code-named EOB) mechanism in direct-mode (i.e. not pre-buffered by an external state machine), has a few caveats to be considered in order to ensure full reliability when designing a state machine leveraging said feature for a gateway at high polling rates.

The intent of this document is to clarify those caveats and their associated potential failure modes, as a guide to the integrator, so that the EOB marker feature can be leveraged without hiccups. It is worth mentioning that this feature is fully stable, functional and has been in use in production designs since the first version of Smartrek Spidermesh. Issues can arise due to desynchronization caused by client-side delays or reaction time when receiving the EOB markers. This can be due to OS CPU overuse, or to design issues such as (1) UI-thread contention limiting the rate at which processing can go, and jitters depending on OS's UI thread management; (2) executing long running operations that take too long to finish executing, such as remote database queries, logging, data mining/processing, etc. in between EOB markers.

Warning!

Because of the complexities of the client-side integration of Smartrek Mesh EOB parallel-polling feature, it is recommended to **use an unsynchronized design paradigm for a faster development-to-production cycle**, or to field test cycle, unless power efficiency or raw speed is a prime concern.

2 The EOB Strategy

Figure 1 shows the proper operation of a typical computer-driven direct-driven EOB strategy—implementing Smartrek mesh network constituting of 2 nodes, a gateway, and a host computer interfaced to the gateway via serial port (FTDI cable), running at a mesh duty cycle of 50%, with sleeping gateway mode disabled (that mode enables a functionality where a serial request must come in before a set interval, after which gateway shuts down for power-saving sleep; this mode is disabled at factory reset).

The white rectangles show an individual broadcast cycle, complete with 2 phases, 1 broadcast out phase (dedicated to gateway-to-node broadcast beacons), 1 broadcast in phase (dedicated to node-to-gateway broadcast responses). The shaded gray rectangles are sleep intervals.

All timings in the RF-POV side are deterministic to the millisecond. The bulk of the timing jitter is typically caused by the HOST-POV interfaces (a.k.a FTDI Serial-2-USB cable, OS serial drivers, and software application messaging and threading delays).

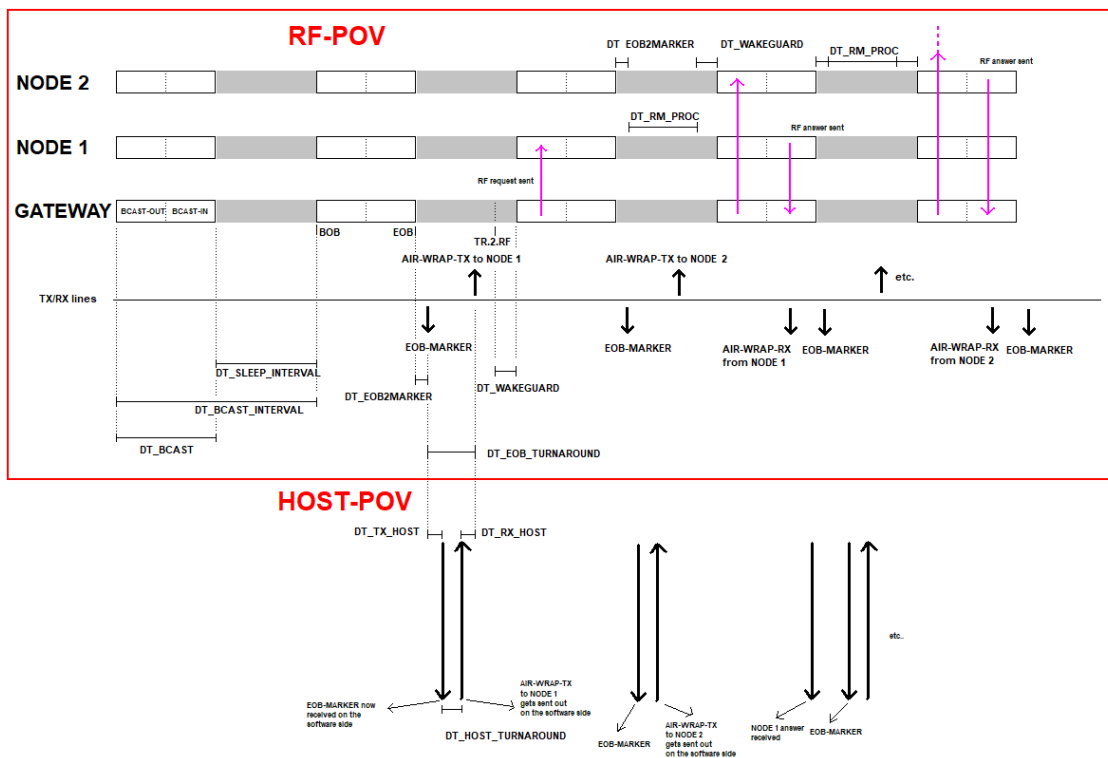


Figure 1: Proper operation of a typical computer-driven direct-driven EOB strategy

Figure 1 shows the general process under which a host waits for an EOB-MARKER to arrive on the serial electrical line, and then sends a request to a given end node (AIR-WRAP-TX, where the wrapped command can be anything, but typically is a VM execute command for remote sensor acquisition/control for Smartrek networks).

- After a certain time delay determined by host hardware and software, it sends a high-level message to its serial port driver, which gets routed to the low level (RF-POV).
- As one can see, the AIR-WRAP-TX serial command then arrives on the serial line and is then fed into PORTIA's internal serial buffer, which holds it until the TR.2.RF event as shown on the diagram (when a node typically has to wake up from sleep for pre-processing data and for clock stabilization before a BOB event).
- When TR.2.RF event arrives, the serial buffer flushes into the RF mesh engine, which then sends it over the air using Smartrek's mesh technology, to the destination node whose target

Table 1: Glossary

Acronym	Name	Description
SPIDERMESH	Spidermesh protocol	Codename for Smartrek's cooperative mesh protocol
PORTIA	Smartrek Portia RF node	Codename for Smartrek's Spidermesh technology flagship RF module
RF	Radio Frequency	Technology set for transporting data over radio link
TX	Gateway Portia Serial Transmit	UART (serial 3.3V) port's electrical serial transmit line
RX	Gateway Portia Serial Receive	UART (serial 3.3V) port's electrical serial receive line
CTS	Serial Clear-To-Send line	UART (serial 3.3V) line being pulled down when client is cleared to send data towards Portia via RX line
TINY	ATTiny816	The co-processor (typically holding a serial buffer to allow for sensor usage without requiring CTS line)
TR.L.	Transparent layer	the ATTiny816's transparent firmware layer that allows abstraction of CTS (with a very small buffer size caveat emptor)
BCAST	Broadcast	Broadcast cycle (the time interval when the CPU is fully dedicated for RF synchronized operations)
BOB	Begin-Of-Broadcast	When a broadcast cycle begins (when Portia's CPU starts running full-bore for RF operations)
EOB	End-Of-Broadcast	When a broadcast cycle ends (when Portia's CPU previously dedicated for RF operations releases to execute less-time-sensitive operations)
EOB-MARKER	End-Of-Broadcast Marker	Marker that gets sent by Portia over serial TX line to flag host that a broadcast just ended (typical jitter: on the order of microseconds). Typical packet format seen is 0xFB 0x03 0x00 0x26 0xFF 0x00
OTA	Over-The-Air	Anything that gets sent over the air to a RF node
AIR-WRAP-TX	Air command wrapping OTA packet transmit	Mechanism allowing transport of a command sent to Portia's RX line (that typically can reconfigure or execute something locally on a node) toward a remote node somewhere. Typically sent from gateway to an end-node
AIR-WRAP-RX	Air command wrapping OTA packet receive	"The receive mechanism where the answer from a remote node gets transported via RF back to gateway, and sent over the gateway's TX wire"
DYN	Dynamic mesh configuration	The timing parameters and hop count parameters that regulate proper mesh operation. Encoded via 6 bytes.
DT_{SLOT}	Mesh slot duration	Mesh base slot duration (for 50 kbit/sec operation, 10 msec for single rate, 20 msec for double rate).
DT_{BCAST}	Broadcast cycle duration	The full duration of a broadcast cycle (from BOB to EOB)
$DT_{SLEEP_INTERVAL}$	Sleep interval	The time spent "sleeping" or executing non-time-critical non RF operations (such as serial parsing)

Table 2: Glossary (continued)

Acronym	Name	Description
$DT_{BCAST_INTERVAL}$	Broadcast interval	Time between 2 consecutive EOB (or 2 consecutive BOB)
$DT_{WAKEGUARD}$	Wake-up guard time	The guard time before a BOB starts, where serial operations are not allowed (CTS line asserts)
$DT_{EOB2MARKER}$	Delay between EOB and EOB-MARKER	The delay between the actual end of broadcast event, and when the EOB marker gets sent on serial line
DT_{RM_PROC}	Remote processing time	Remote processing time of an end-node
$DT_{TX_SER2USB}$	Serial 2 usb cable latency, transmit	Latency time between the moment a serial packet is sent on the Portia TX wire, and the moment the USB equivalent flows to the host CPU
$DT_{RX_SER2USB}$	Serial 2 usb cable latency, receive	Ibid. but from a USB request from host CPU, back to Portia RX wire
DT_{TX_OS}	OS latency, transmit	Latency time between USB reception of serial packet flown thru Portia TX wire, until OS driver delivers message to user host software
DT_{RX_OS}	OS latency, receive	Ibid, but from host software message sent to serial driver, back to USB interface
DT_{TX_HOST}	Total host-side latency, transmit	$DT_{TX_HOST} = DT_{TX_SER2USB} + DT_{TX_OS}$
DT_{RX_HOST}	Total host-side latency, receive	$DT_{RX_HOST} = DT_{RX_SER2USB} + DT_{RX_OS}$
$DT_{EOB_TURNAROUND}$	Effective EOB reaction time	The effective total reaction time from the moment an EOB-MARKER gets sent on TX wire, to the moment an answer from host gets received on RX wire
$DT_{EOB_MAX_TURNAROUND}$	Max effective EOB reaction time	The maximum value allowable for $DT_{EOB_TURNAROUND}$ for proper operation
$DT_{HOST_TURNAROUND}$	Host reaction time	The host application-side reaction time from receipt of a serial message in the application-level, to the time it sends an answer
$DT_{HOST_MAX_TURNAROUND}$	Max host reaction time	The maximum value allowable for $DT_{HOST_TURNAROUND}$ for proper operation
BUF.OVF.	Buffer overflow	When a serial buffer overflows, and there is loss of bytes
TR-2-RF	Transfer to RF	Event transferring serial buffer content in Portia module to its internal RF transmit engine
RF-POV	Point-of-view of low level RF	Reference frame of a SPIDERMESH running set of module(s)
HOST-POV	Point-of-view of host	Reference frame of a host connected to a gateway via serial link

address was sent as part of the AIR-WRAP-TX serial frame), during the BCAST-OUT phase of the broadcast cycle.

- Then, after the current broadcast cycle elapses, the next sleep interval is when the end node being targeted fetches its sensor data (by a mechanism typically being custom defined by either the TINY's firmware, or an EVM sitting inside the PORTIA module itself). For proper operation, the end node must process and collate return data faster than the sleep interval in which it is executing said processing, lest it overflows over its allowed processing time, in which case the VM engine will kill user code to make sure the end node does not desynchronize from the mesh network as a result. Note that at the beginning of said sleep interval, an EOB-MARKER was received by the gateway.
- After this sleep interval, during the next broadcast cycle, the targeted end node sends its answer, which gets transported via RF to the gateway, which generates a serial answer packet AIR-WRAP-RX as a result, and after which an EOB-MARKER is also sent (at the end of the current broadcast cycle).

This signifies that any host software correctly employing the EOB paradigm in direct mode as currently described **should expect exactly 1 EOB-MARKER received in between an AIR-WRAP-TX request and its answer counterpart**, from the moment the request gets sent on the RX line, to the moment the answer transits on the TX line. If ever an AIR-WRAP-RX answer is not received within 1 EOB-MARKER counted after the corresponding AIR-WRAP-TX request was sent, it either means that:

- The request packet or its answer counterpart was dropped in transit (either from gateway to node, or vice versa), or;
- Somehow the user software is not properly synchronized to the EOB-MARKER, most likely due to host-side latency issues.

Here are the timings relevant to the diagram above. The conditions for proper mesh operation under the EOB strategy umbrella are also calculated here (in red):

3 Failure Mode A

In layman's terms, if the reaction between an EOB-MARKER event and the next packet to be sent "overflows" the boundary marked as TR.2.RF on the diagram, but the time interval between requests is shorter than the time interval between broadcast cycles $DT_{BCAST_INTERVAL}$, then the mesh controller does not have time to buffer the request before a broadcast cycle starts, and is forced to buffer it for the one after, therefore making the system "lag" behind by 1 broadcast cycle, and yet, since the time interval is lower than the broadcast cycle interval, it means that the host still "follows" the mesh synchronization. Let this be called failure mode A, and is described by the diagram in Figure 2.

4 Failure Mode B

If the host-side processing time delays are more severe and are higher than the time interval between broadcast cycles, then the resulting failure mode B can be described as presented in Figure 3.

Table 3: Basic Time Interval Variable Relationships

Basic Time Interval	Variable relationships	Comments
DT_{SLOT}	$500/(b * (double_rate?2 : 1))$	In msec. B is the bitrate in kbit/sec, double_rate is a flag whether system is operating in double rate (aka <i>presetRFregister</i> >= 16) or not."
$DT_{WAKEGUARD}$	$DT_{WAKEGUARD} = DT_{SLOT}$	
$DT_{EOB2MARKER}$	$DT_{EOB2MARKER} = 0$	Typically 0 msec (i.e. EOB gets sent as soon as a slot shuts down RF front-end, even before "logical" slot ends), meaning that this can be discarded from calculations in practice
DT_{BCAST}	$DT_{BCAST} = DT_{SLOT} * [(bcast_{in} + bcast_{out}) * num_seq + (redux_enable?(num_redux == 0?num_seq : num_redux))]$	Dependent on DYN configuration, as a multiple of DT_{SLOT} . See Portia user manual V.4
$DT_{BCAST_INTERVAL}$	$DT_{BCASTCYCLE} = DT_{BCAST} * duty_cycle_div$	Dependent on DYN configuration parameter <i>duty_cycle_div</i> . See Portia user manual V.4
$DT_{SLEEP_INTERVAL}$	$DT_{SLEEP_INTERVAL} = DT_{BCAST_INTERVAL} - DT_{BCAST} = DT_{BCAST} * (duty_cycle_div - 1)$	
DT_{RM_PROC}	$DT_{RM_PROC} = DT_{SLEEP_INTERVAL} - DT_{WAKEGUARD} - DT_{EOB2MARKER}$	

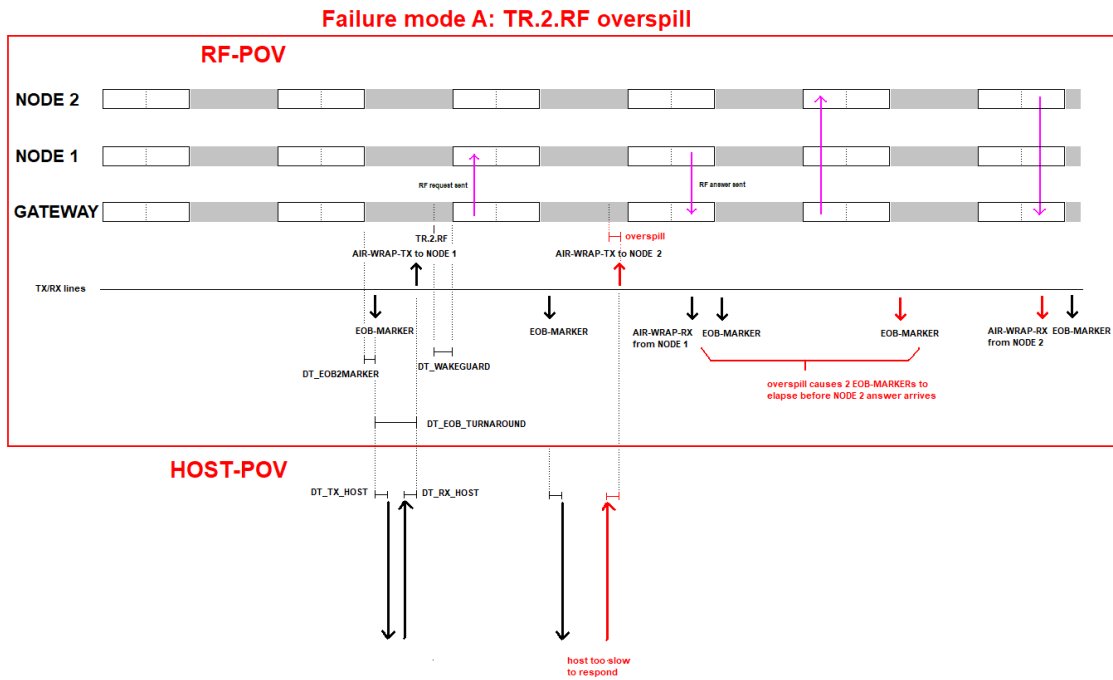


Figure 2: Failure mode A: TR 2 RF overspill

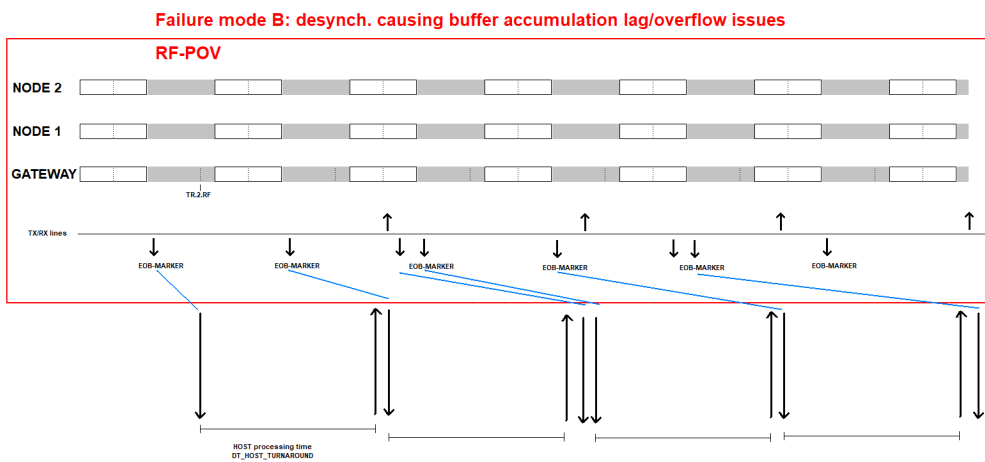


Figure 3: Failure mode B: EOB desynch

In such a case, the EOB-MARKERs accumulate progressively in the low level serial buffer between the RF network and the host-side, since during the time the host processes data to be sent for a single EOB-MARKER received, more EOB-MARKERs and RX packets accumulate, and they accumulate faster than they are removed, that is, until this causes a buffer overflow condition, or a state machine crash on the host side if it is not specially tailored to mitigate that failure mode via back-off time delays or similar corrective measures. Since the timing of the received serial messages on the host side **depends on the host's processing speed in this case** because that processing speed is the bottleneck, this means that the timestamp at which the host receives a given message (a.k.a. the perceived message reception time in the HOST-POV) does not correspond to the real timestamp as seen if someone were to probe the TX and RX lines (RF-POV), and in fact would lag behind progressively as time passes by.

5 Failure Mode C (hybrid)

There is a third failure mode, which is seen when the buffer depth for OTA requests is small enough, and results in literal serial message drop or message corruption.

An example of this possibility is with the ATTINY transparent layer prototype dating from August 2019, which uses (1) a processor having <512 bytes total RAM space, which means its receive buffer is quite small and can hold only a few double-rate messages at most (a double-rate message for Portia V2.3+ firmware at 50 kbit/sec can be up to 72 bytes in length).

Not only that, but the integration layer between the ATTINY and Portia, although for local serial communications between gateway and host (for instance, in order to modify local registers), has a buffer depth equivalent to the ATTINY's allocated serial receive buffers, only has a buffer depth of 1 for air transported commands (AIR-WRAP-TX commands).

As a matter of fact, the mesh controller has a buffer depth of 1 by default for all OTA commands, if one considers only the low-level UART behavior at the TX, RX and CTS pins of the PORTIA module. This can be verified by probing the serial lines and observing the behavior of the module when an AIR-WRAP-TX command is sent out to a remote node on the gateway side: the CTS pin immediately asserts itself to block further UART messages, until the next broadcast cycle starts, as shown in Figure 4.

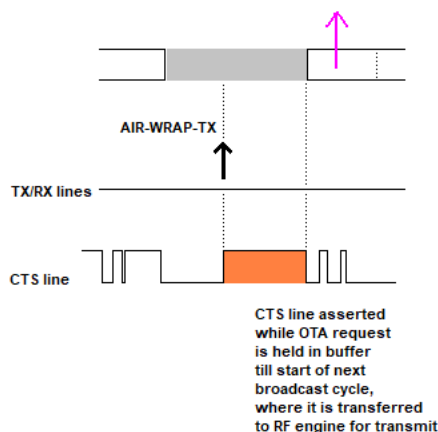


Figure 4: Next broadcast cycle starts

However, the regular mode of operation entails leveraging the CTS pin when coupling the module to a host (be it a computer for the examples shown in this document, or a serial-to-Bluetooth bridge to an Android core for Smartrek's demo app, code-named Sugarheld, which implements an EOB-based synchronous strategy to optimize throughput as well).

When the CTS pin is used, the module might have a buffer depth of 1 only for OTA requests, but it would assert the CTS pin in that case, forcing the host low-level UART hardware to buffer the incoming OTA request messages upstream to the module, therefore offloading the buffering capabilities to the host itself, and allowing deep-buffer-like functionality even if PORTIA has a buffer depth of 1 for such messages.

However, since the ATTINY design dispenses of the CTS pin, the buffering limitations are immediately apparent and applicable for any host connected to a PORTIA module that has this co-processor

feature activated.

If one is only using the ATTINY for local UART communication, say for example if one attempts to interface sensor heads to an end-node via a UART port that does not support CTS lines, then the ATTINY RX buffer depth limitation applies, which is a buffer depth not by message count but rather by the total number of bytes (and because a typical sensor head single message is typically much shorter than the ATTINY's buffer depth maximum byte count, this strategy can be correctly leveraged without any further ado).

However if one is using the ATTINY transparent layer (TR.L.) directly on the gateway for OTA requests, then the message-level buffer depth of 1 applies (for PORTIA versions up as of 2019/10). In this case, the host polling might see UART message drop issues as soon as there are too many requests sent in the interval between two consecutive TR-2-RF events. And in the case where too many UART messages are sent before the ATTINY buffer can be purged and transferred in full to the PORTIA's main processor (each ATTINY purge event occurs after an EOB event), then a corruption where a serial message gets truncated, which could invalidate the internal PORTIA's packet frame parser during a while until it re-recovers the correct 0xFB preamble (for API mode, at least).

Here is a diagram showing how such events can occur: they occur in the specific case when failure mode B occurs, but where high variability in the $DT_{HOST_TURNAROUND}$ delays due to host-side design concerns occasionally cause a burst of multiple serial requests (worse if they are OTA requests) in-between two TR-2-RF events (Figure 5).

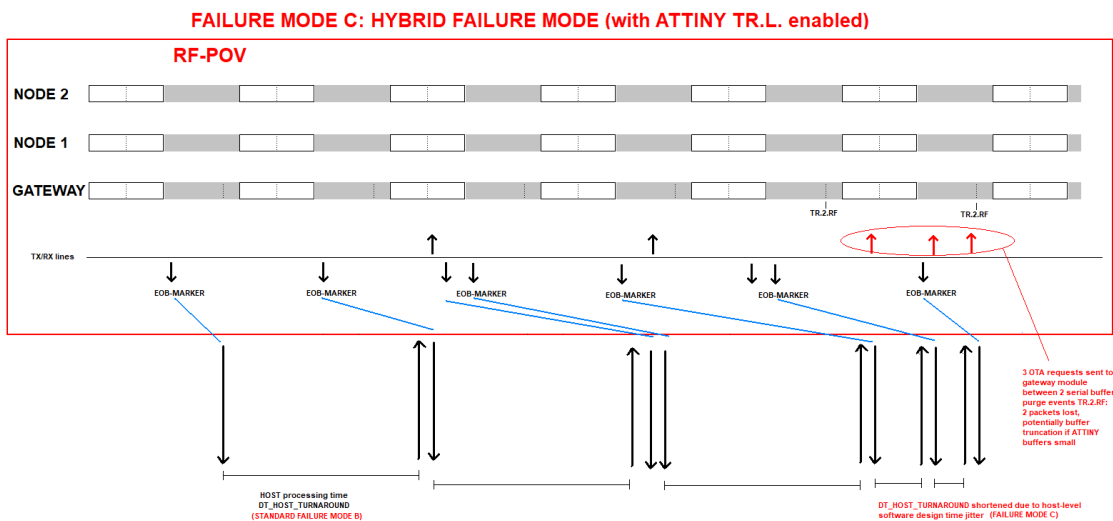


Figure 5: Failure Mode C

Thus, the 1st-level corrective action to mitigate this hybrid failure mode C, is to disable any ATTINY buffering on the gateway PORTIA module, and leverage the CTS pin of the module's serial interface in order to buffer upstream within the host's serial driver (for example, when using a FTDI USB-Serial bridge, the deep-buffering is thereby offloaded to the FT232 chipset).

6 What are the typical bottleneck/time delays/jitter sources, and how to mitigate those?

Even though mitigation of failure mode C is straightforward, to remove all possible failure modes (A and B) would rather entail deeper software design decoupling, namely via the implementation of separation of concerns between the low-level serial-port-facing state machine polling nodes, and the high-level software architecture.

What can cause high variability in the host processing reaction time $DT_{HOST_TURNAROUND}$? There could be many causes, sometimes simultaneously, depending on how the host-side design is implemented. For example, a common mistake is employing data-fetching or data-updating blocking operations that take a long time to complete, yet forces the software to wait until they complete, before being able to send the next serial request to PORTIA.

Here are a few examples of possible host-side bottleneck-creating operations with unpredictable timing:


- improper use of threads and mutexes
- processing messages on an overworked UI thread
- fetching data from remote databases or servers in a blocking, non-buffered manner (which usually entails use of TCP sockets where query times are unpredictable and depend highly on network conditions, such as when the computer running the host software suddenly run other high-load network services, for example)
- long running disk fetching operations
- data crunching and data parsing on demand needed to create a given OTA request

The typical measures to ensure that no bottleneck impede network operation using EOB strategy are to buffer the data-layer holding the values that need to be updated/read to/from the mesh.

For example, if the data storage DB for a node network is held on a remote server, then a local copy shall be periodically synchronized to the remote copy using a separate thread/process, while the local copy data is fetched/updated on a separate, high priority thread, which is the only thread that directly interacts with the serial drivers in order to communicate with the PORTIA gateway module. All UI update/fetch operations are buffered using inter-thread or IPC communication methods, such as messaging, pipes, etc.

An example of such an implementation is Smartrek's own App reference design (Sugarheld) which holds a state machine leveraging the EOB-MARKER functionality. In this case, a local SQLITE DB is fetched/updated using a dedicated high-priority Android service, with all UI communication being done via IPC OS-level Broadcast Intents (the equivalent to Window messages) to ensure no UI interference with the polling service.

This SQLITE DB is synchronized to the cloud using a completely separate thread/service. Moreover, mutexes are used only very sparsely as a thread synchronization within the polling state machine operating the serial port, and messaging strategies are favored, in order to eliminate potentially time-jittery events such as thread starvation via mutex hogging, for instance.

 **Any design reference available?**

Smartrek Technologies Inc. can provide engineering services, and software libraries implementing such strategies. Please contact Smartrek Technologies Inc. for further details.