PolarSoft® Inc.
914 South Aiken Ave
Pittsburgh PA 15232-2212
412-683-2018 voice
info@polarsoft.com
www.polarsoft.com

Source: **Markus Peters on BACnet-L (14-Jun-2012)**

Question: *I have problems to understand why the T(usage_delay) is set fixed to a value of 15ms, but the T(usage_timeout) is a variable value in the range of 20ms to 100ms.*

If a device has to send the first octet of a frame within 15ms after reception of a token or PollForMaster, than you should also have a fixed value for T(usage_timeout) to detect that the token was not used, right?

But T(usage_timeout) value is defined in the range of 20ms up to 100ms. Why it is allowed to set the value up to 100ms to detect a timeout, while the device that owns the token has to use it still within 15ms? Is it an implicit permission to the token owning device to exceed the 15ms T(usage_delay)? Because if the device that passed the token waits up to 100ms to detect the token-usage, the token-owning device can also send the first octets of a frame 60ms after receiving the token and the device that passed the token would see correct usage of the token, didn't it?

If I have devices that need about 40ms to send first octets and all devices are configured with T(usage_timeout) 90ms, then MSTP will work fine, although the devices would not meet the specification of 15ms T(usage_delay).

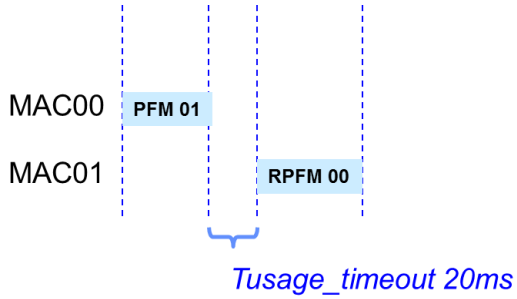
So why is T(usage_timeout) variable? Is it because it can be used to get well running MSTP even if there are devices not meeting the T(usage_delay) specification or is there any other reason why T(usage_timeout) is variable within this greater range?

QUESTION

Answer: **David Fisher**

The standard is very strict about the requirements on the receiving device. It MUST reply in less than 15ms (Tusage_delay). Any device that takes longer is not conforming to the standard, and is not expected to work correctly.

The transmitting device MUST wait at least 20ms for a reply, but it has the option to wait longer, up to 100ms (Tusage_timeout).



This leads to two questions for designers:

- Why are these two numbers different?
- Why does the standard allow the sender to wait longer than 20ms?

These are subtle points with various side effects that are not obvious.

1. Why are these two numbers different?

One of the underlying assumptions about every MS/TP implementation is that it requires a minimum timebase of 5ms ticks. Another way of saying this is to say that the MS/TP state machines should not expect any finer resolution than 5ms in any operation (of the state machines). If we imagine that the sender and receiver have their own 5ms interrupts for administrating the 5ms timers, it is obviously possible that the sender and receiver may not be exactly synchronized to each other's timers. So the sender could transmit while its 5ms counter hits zero but the receiver may still be counting down. In other words, the sender and receiver may be out-of-synchronization by as much as 5ms.

As a result, the MS/TP state machines are specified so that the timing constraints include an extra 5ms of time, even though it is normally not really required, but could be required under edge conditions. That's why Tusage_delay (the requirement on the actual response of the receiver) is 5ms less than Tusage_timeout (the requirement on the sender to wait for a reply).

A very subtle additional implication is that the ReceiveFrameStateMachine (RFSM) is specified as having a maximum period of 5ms. In a perfect world with infinite processor speed, the RFSM would be called within a receive interrupt so that each incoming octet is processed and the state machine reevaluated each time. In practice this is not always possible and was deemed to be too invasive of a requirement on MS/TP implementations. As a result, many implementations simply buffer received octets in their receive interrupt, and process them asynchronously in a RFSM based on a 5ms period. This has its own subtle implications too. Under this scenario, it could be as much as 5ms later than the arrival of reply octets that the RFSM gets to run and timers get evaluated. This is a second reason for the delta between Tusage_delay and Tusage_timeout. The other subtle implication of this case is that the buffering capacity between the receive interrupt and the RFSM must be large enough to buffer all of the octets that might be reasonably expected to arrive during the period of latency from the receive interrupt of the first octet and the execution of the RFSM. Without this, sequences of messages from other devices can be lost due to buffer overrun.

The cost of having this "arms-length" relationship between the local receive interrupt and the RFSM is one aspect of the timing constraint.

2. Why does the standard allow the sender to wait longer than 20ms?

It really shouldn't. We've come to view MS/TP implementations as being either "strict" which means that they wait at most 20 (or maybe 25) ms, or "lenient" which means they wait longer (say 50+ ms). Some implementations even allow this to be adjustable, although the standard does not talk about this.

A strict implementation is ideal because:

- It does not waste time
- It conforms to the obvious requirement of Tusage_delay with the 5ms grace period.

However, the downside of being strict is that the device will not interoperate with tardy implementations. So the strict implementation, which most closely conforms to the standard, will appear to be LESS interoperable because it is intolerant of non-conforming implementations.

A lenient implementation is allowed under the doctrine of longer Tusage_timeout, BUT it wastes time.

The evil component in this discussion is any MS/TP implementation that does not conform to Tusage_delay. However in practice it may be hard to explain this to customers. The classic scenario is, you have some number of lenient and tardy devices mixed on an MS/TP segment. Everything appears to work OK. Now you add a strict device and the segment stops working, has lots of timeouts etc. Who is going to be blamed? The "new" device, even though it is actually the most conforming to BACnet! Many implementations therefore choose to be intentionally more lenient to circumvent this.

What is the downside of being lenient?

When a master device sends a PollForMaster (PFM) message, for example, it waits for Tusage_timeout for a reply. If none is received in that period, it goes on to the next device. The total number of these PFMs depends on MaxMaster. MaxMaster is most commonly set to 127. Whatever the amount of leniency is beyond 20ms will become wasted time every time a master polls for another MAC address that doesn't exist. So, for example, if your Tusage_timeout is 50ms you have 30ms more than required time built-in to your state machine. If you have a segment with 30 devices on it, and MaxMaster=127, then you are wasting 97x30ms every PFM cycle.

This is lost network bandwidth that slows down your system, constantly 24/7. The larger the leniency, the more wasted time.

If this is so bad, why doesn't the SSPC "fix" this problem in the standard? Several attempts have been made to introduce changes to Clause 9 that would eliminate leniency in favor of strict (or much stricter) timing for Tusage_timeout.

These have all been defeated due to fear and uncertainty about the "tardy device" problem and its perceived negative effect on real installations.

