Ethernet "Frame Preemption"

© *Rob Hulsebos*

Version 8/3/2018

Introduction

In order to fulfil the needs of a control application, i.e. for a machine, robot, CNC-machine, or any other application requiring real-time processing, all components of such a system must work in a predictable manner. This also holds for any network used: messages must be delivered in time (or at least: not too late). But when Ethernet is used, there is no guarantee that this will happen. Traditionally, the "solution" is to keep the load on the network as low as possible; some vendors recommend to not go higher than 3%.

The problem with Ethernet is that it allows any device to send messages at any moment. Sometimes this is just too much for the network to handle. Messages are then queued in a "FIFO" manner: first-in, first-out. A message with high urgency can then be delayed by messages in front it, which just happened to arrive a little bit earlier. You recognize this probably: the same happens on the highway during rush-hours.

Figure 1: Ethernet and the highway have the same usage model, which sometimes leads to unpredictable delays.

In some industrial Ethernet protocols, like Ethercat and ProfiNet, solutions have been invented to guarantee predictable behavior. But these are always protocol-specific solutions, and not standard available on Ethernet. A more generic solution has been developed as part of the "TSN"(Time Sensitive Networking) extensions, according to the (new) IEEE standards IEEE 802.br and IEEE 802.1Qbu, popularly known as "Frame Preemption".

In the chapters below, we will first describe the basics of frame preemption, followed by a more detailed explanation about the internals.

1 Cyclic Communication

In industrial controllers, it is very often seen that the application software runs in a cycle; the same code is executed again and again. The cycle times are usually in the order of milliseconds, but this can also be in in the microsecond range for the applications with the highest demands regarding handling of events, accuracy, or output.

Every cycle the same processing is done: get all current inputs (digital, analogue, encoders, servo's, camera's, etc.), run the application software, set all outputs to their new values (digital, analogue, HMI's, servo's, etc.). This cyclic way of working is typical for PLC's (Programmable Logic Controllers) and DCS's (Distributed Control Systems), but also often seen in real-time applications.

Figure 2: Real-time applications continuously execute the same cycle, both in software as on the network they use.

The cycle time is constant. This allows the software to respond in time to any external events: in the same cycle, or perhaps the next. So when the cycle time is exceeded, timely response cannot be guaranteed, and this is often a reason to alert the operator.

Communication is done via the network to the subordinate devices, and they must respond as quickly as possible. What happens on the network is completely predictable: the number of devices is known and the amount of data per device. Thus, what happens on the network is quite boring: always the same, the whole day: read inputs from device #1, read inputs from device #2, read inputs from device #3, etc. and at the end of the cycle: set outputs on device #1, set outputs on device #2, etc. Given the fixed number of devices and fixed I/O, it can be calculated in advance how much bits are transferred. Give the bitrate, it can also be calculated how long all transmissions are going to take (figure 3).

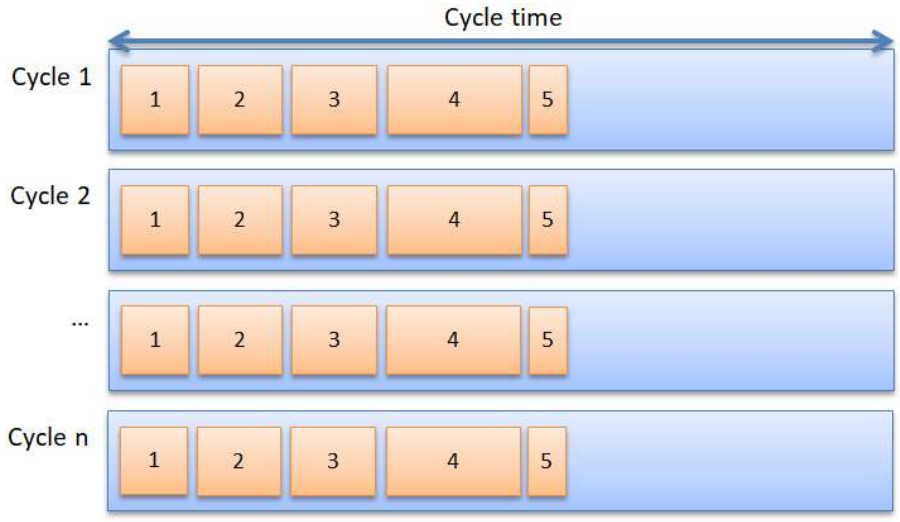


Figure 3: Cyclic transfer of 5 messages always takes the same time, every cycle.

So, we can calculate in advance how much time is needed for the network communication. Confusingly, this is also called "cycle time". Ideally, this **network** cycle time is shorter than the cycle time for the software, so it is not delayed by the network.

> *In practice, the application software and the network run in parallel: while the software is running with data from network cycle 'n', the network is collecting data for cycle 'n+1'. Even then, the network must be ready before the application software wants to start its next cycle.*

Most existing industrial networks, both of the 1st generation (i.e., Profibus, Interbus, CAN, DeviceNet) and of the 2nd generation (ProfiNet, Ethercat, etc.) function this way.

Acyclic data

Apart from the cyclic communication done on a network there is often a need for acyclic network messages. These are messages that come at (for the network) unexpected moments, for example due to the operator giving special commands ("Stop!"), because an error occurs somewhere, because diagnostic data is retrieved, because data must be downloaded, a webpage being read from a devices server, or any sort of communication over the network (even network managers starting a backup on a production network ☹). There must be some room for extra messages (figure 4) – not too much, otherwise the network cycle becomes too long and the application software must wait.

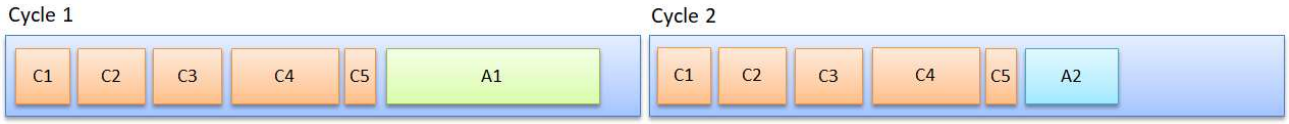


Figure 4: Two network cycles, each showing the same number of cyclic messages with the same amount of data. In both cycles one acyclic message (A1, A2) is sent, with different data.

The extra time allotted may also not be too short, otherwise long network messages cannot be completely transmitted within the available time. One solution is to allow for one full-length Ethernet message. Simple as it sounds, it has one drawback: a full-length Ethernet message can contain 1500 bytes of data, which is a huge amount compared to the usually small messages used for cyclic data – and, if there is no acyclic data, the network is doing nothing for the duration of one full-length Ethernet message. Additionally, is it guaranteed that there will be only *one* acyclic message? Or can there be more?

In any case, if there is too much acyclic traffic to fit in a cycle, the next cycle will suffer – its cyclic data is transmitted too late (figure 5). They must wait for message A3 to be finished (FIFO).

Figure 5: A too long acyclic message exceeds the time allotted to cycle 1. The cyclic messages in cycle 2 are transmitted too late.

What happens next depends on the application software. Did it miss important signals? Is equipment going to be damaged? Is production delayed? Can people get hurt?

Guard band

To prevent that too long or too many acyclic messages extend the cycle time, the "guard band" is used. It is a (configurable) period of time at the end of each cycle. Transmission of a network message may only start when it is guaranteed that it finishes in the guard band (= before the end of the cycle).

The next cycle can thus always start at the expected moment. If there are network messages that could not be transmitted in the previous cycle, they can now be transmitted (time allowing).

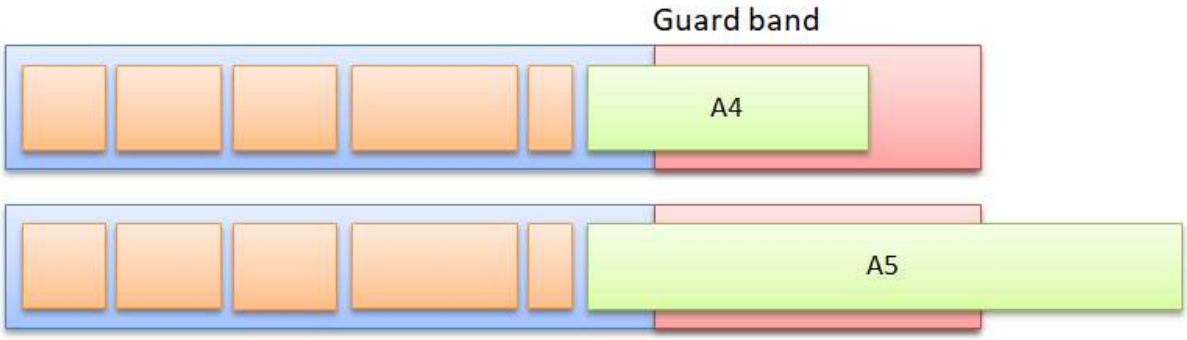
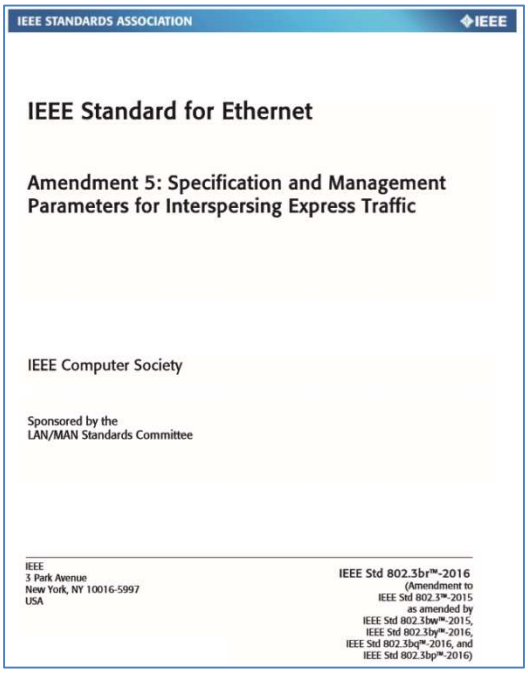


Figure 6: A message may only be transmitted if it finishes in the guard band. A4 complies to this rule, but A5 may not be transmitted.

Figure 6 (top) shows the transmission of only one acyclic message A4. But there is still time left in the guard band, so more messages may follow (if there are any). If there are none, the time is still wasted. So the network manager will tend to make the guard band as short as possible, but this may give a problem: too long messages (A5) cannot be transmitted anymore, they never fit! (figure 6 bottom). This is especially troublesome in protocols where the application / user has no control over the size of network messages, such as in TCP/IP.


IEEE STANDARDS ASSOCIATION IEEE

IEEE Standard for Ethernet

Amendment 5: Specification and Management Parameters for Interspersing Express Traffic

IEEE Computer Society

Sponsored by the LAN/MAN Standards Committee

IEEE
3 Park Avenue
New York, NY 10016-5997
USA

IEEE Std 802.3br™-2016
(Amendment to IEEE Std 802.3™-2015 as amended by IEEE Std 802.3bw™-2015, IEEE Std 802.3by™-2016, IEEE Std 802.3bq™-2016, and IEEE Std 802.3bp™-2016)

So the guard band solves one problem, but there is still one to go: how to handle very long messages. It is this issue which "**frame preemption**" is going to solve for us. Its functionality is specified in IEEE 802.3br. The document can be requested from the IEEE for free. With a size of only 58 pages it is not too large, but you must know something about Ethernet in order to understand it well. The interesting part about the functional way-of-working start at page 38 in section 99.3 (which we will discuss below).

Express messages

Before we continue with the detailed explanation of the inner working of Ethernet frame preemption, we must first agree on the terminology. The real-time messages that are sent first in a cycle are called the "**Express**" messages. They are never subject to preemption. All other messages (called acyclic above), are called "**Normal**" messages. If they do not fit in the guard band, they are subject to preemption.

Frame preemption

This (for Ethernet) new technique solves the problem of a too long network message which doesn't fit in a cycle. Simply explained, it does the following:

- Stop (preempt) the transmission of too long messages *before* the end of the guard band.
- Continue the transmission of the remainder of the message in the next cycle, following the express messages.

In Ethernet terminology, the preempted message is sent in multiple (>= 2) **fragments**. At the receiver, the fragments are assembled again to re-create (a copy of) the original network message.

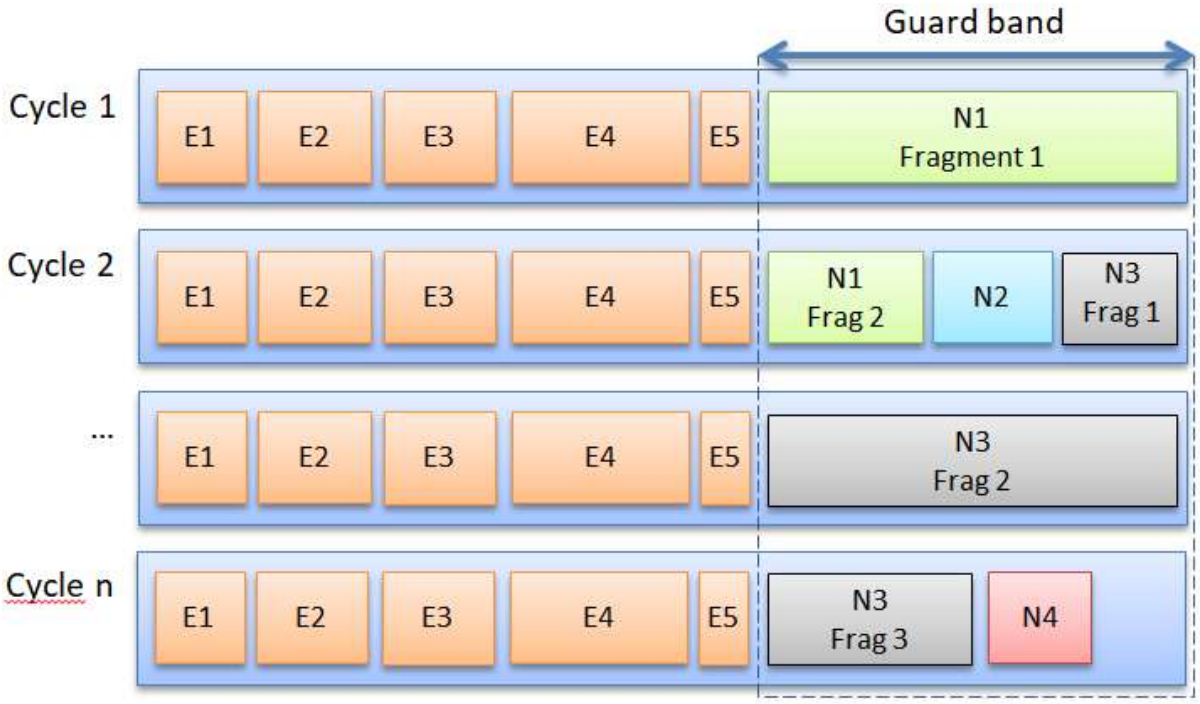


Figure 7: A too long message N1 is transmitted in fragments over multiple cycles. As long as there is time in the guard band, additional messages may be sent too (N2, N3, N4). If they don't fit they are fragmented too, etc. etc.

In figure 7, message N1 is too long to be completely transmitted in the guard band of cycle 1. It is thus preempted. After the transmission of this first fragment is stopped, the next cycle 2 can start at the expected moment. When the express messages have been transmitted, transmission of the second fragment of N1 is done. As there is still time in the guard band available, N2 can be transmitted too. With still time left, the first fragment of N3 can be transmitted. This is such a long message that even in the next cycle it cannot be completely transmitted, so it is preempted again, and the third fragment is sent in cycle 3. With enough time available, N4 can be transmitted. With nothing else to do, the line is silent for the remainder of the guard band.

For the receiving software, the fragmentation of messages is completely invisible. N1 is passed on to higher protocol levels only when the second fragment is completely received; identically so for message N3 after reception of the third fragment. So, for higher protocol levels it looks like any

other Ethernet message. That the total transmission time is a little bit longer is unnoticeable; the message could as well have been sent a little later.

Error handling

What happens when the data in a fragment is corrupted while in-transit? The receiver detects this with a CRC which is appended to each fragment. Corrupted fragments are summarily discarded, just as any normal Ethernet message with corrupted data. But discarding a fragment means that, even when all other fragments are received without errors, the original Ethernet message cannot be re-created, so the message is completely lost.

The frame preemption algorithm does not perform any error recovery, unlike TCP/IP which can detect missing fragments in a data stream. But that is at a higher protocol level, which has more intelligence. Adding such intelligence to frame preemption implementation would unnecessarily make it more complex, error-prone and slow.

So the frame preemption algorithm just follows the "I've tried it and it didn't work out; so now it's your problem" way of working, also called "Best Effort" in Ethernet jargon. Or: let higher protocol-levels detect and handle the missing message(s).

Backwards compatibility

Most Ethernet devices will never support frame preemption, as it is very specific for real-time applications. But how does a device with support for frame preemption communicate with a device that doesn't? Simply: without frame preemption being used. The device with frame preemption capabilities asks the other device whether it supports it (via a special housekeeping message). Because the other device doesn't support it, it doesn't understand it, so no answer comes back, and so it is decided not to use the frame preemption feature.

If it so happens that the other device *does* support frame preemption, it will say so, and the new feature can be used. Both devices (at both ends of the cable) must ask the other, so usually both parties will use frame preemption, but it is also possible that only one device does so (however unlikely). This is not a problem for Ethernet, because it is a full-duplex technology.

Note that on a switch frame preemption usage can thus be different for each port.

Port mirroring

How does frame preemption work with port mirroring, are all fragments mirrored individually? The answer is no. Any Ethernet message must have been completely received on a switch before the switch can sent it out via another port. After all, the other port may perhaps not be using frame preemption, or may have a different configuration regarding the express messages or the length of the guard band.

Additionally, if case fragments are mirrored, the device connected to the mirror port (usually a network analyzer or IDS) would have to implement the fragment assembly algorithm for *all* fragments for *all* ports being mirrored, and perhaps using normal Ethernet messages for ports not using preemption. Normally, a device only does fragment assembly for *only one* other device (at the other end of the cable). It doesn't seem impossible, but just not practical.

This concludes the "simple" explanation of frame preemption. The next chapter goes deeper into the internals.

2 Preemption Implementation

As described in the previous chapter, frame preemption can chop a message in smaller fragments, and the receiver assembles them again to re-create the original (long) message. This looks simpler than it sounds, as it is a fundamental change to the Ethernet implementation as it has been for some 35 years.

Basic terminology

We have seen that in order to keep the real-time requirements, some messages must have the guarantee that they can be sent in time. In Ethernet terminology, these are called the "Express" messages, shown as the 'cyclic' messages in figure 2. All messages ***not*** being express messages are labelled "Normal" messages (we called them 'acyclic' above).

Depending on the available time in a cycle, the normal messages can be subject to fragmentation due to preemption. The first fragment of the preemption procedure is called the "Start Fragment", after which a number of "Continuation Fragments" can follow, followed by the "Last Fragment".

Additionally, for management purposes, a device should check whether its communication partner (at the other end of the cable) is able to support frame preemption. A "Verify" message is sent to check this out. The other device must send a "Respond" message back to indicate its capabilities.

Changes to the message format

The Ethernet message format hasn't changed much in 35 year, except for the introduction of the VLAN/QoS (4-byte) field. For frame preemption, a much larger change is necessary, needed to store the additional administrative fields. These fields contain the data that the receiver needs to properly handle fragments (first/intermediate/last) in the right order, and detect errors.

The original Ethernet message format is shown in figure 8. Each message starts with a 7-byte "Preamble", each byte with value 0x55 (binary 01010101). This bit-pattern is used to synchronize the electronics of the receiver with those of the sender, so they run at exactly the same speed. Even though the nominal speed is the same (10/10/1000/10000 Mbit/s), differences in quality of the electronics, and ambient temperature changes, cause small fluctuations in speed, which are now resolved. This preamble has been with Ethernet from day #1.

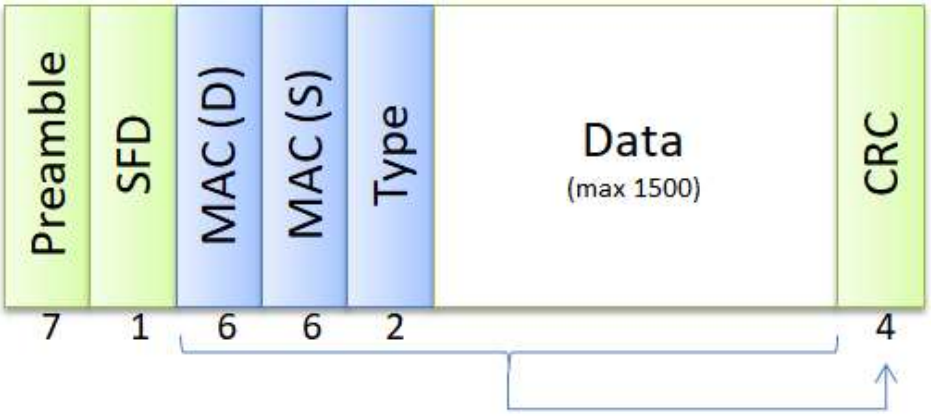


Figure 8: The standard Ethernet message format.

Following the preamble comes the 1-byte field "SFD" (Start Frame Delimiter), with the fixed value 0xD5. Then comes the data part, containing the sender's MAC-address (6 bytes), receiver MAC-address (6 bytes), Type field (2 bytes), and up to 1500 bytes of data (or more with Jumbo frames). The message ends with a 32-bit CRC[1] (Cyclic Redundancy Check), calculated by the sender and used by the receiver to detect that the data part has not been corrupted while in transit. If there is corrupted data, the message is immediately discarded (thrown away) without any further attempt to repair the error(s). This is left to higher protocol-levels (usually TCP, see description below).

> *Frame preemption has similar algorithms in the procedures for checking for errors in received fragments. When an error is detected, the same procedure is followed as Ethernet normally does: discard any fragment (partial) data, do not attempt to repair any errors, and let higher protocol-levels handle this.*

The original Ethernet message format is still used with frame preemption, in the following circumstances:

- Transmission of "Express" messages. The SMD has value 0xD5 (unchanged).
- Transmission of "Verify" and "Respond" bookkeeping messages. These are used to check whether the device at the other end of the cable is able to support preemption. The SMD has value 0x07 or 0x19.
- Transmission of the ***first*** fragment of a message. The SMD-Sx has value 0xE6, 0x4C, 0x7F or 0xB3 (see below for an explanation).

A different message format is used for the ***continuation*** fragments (2..n-1, if any) of a message, and the ***last*** fragment. This requires some additional administration, so the receiver knows how to recognize what it receives. Additionally, it also allows the receiver to detect errors (such as a missing fragment). The new message formats are shown in figure 9 (three at the right). The SMD-Cx has value 0x61, 0x52, 0x9E or 0x2A (see below for an explanation).

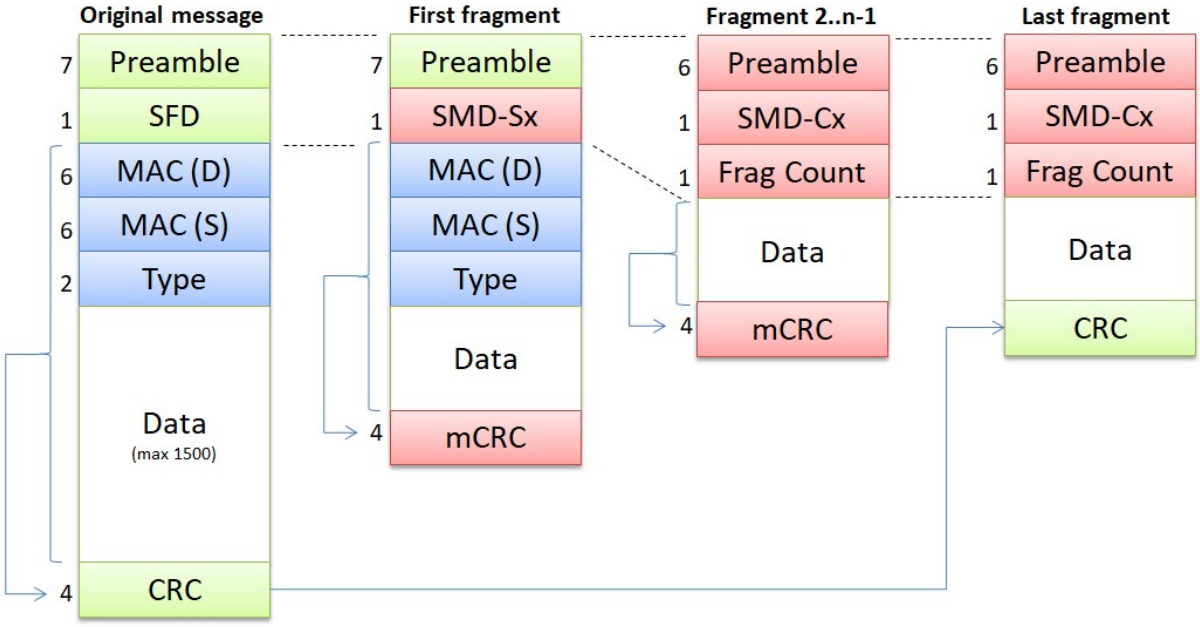


Figure 9: The different three types of fragmented messages as compared to the original Ethernet message format (left). Green and blue fields retain their original value; red fields have different values. The "data" field is not shown to size.

The IEEE has decided to shorten the preamble by one byte; it is now 6 bytes (6 times 0x55) long. Next comes the SMD, followed by a new field called "fragment counter". This makes the initial fields (preamble, SMD, fragment counter) again 8 bytes long. This is good for implementations, the data *always* starts at the 8th byte, independent of the type of message.

[1] This field is also often called "FCS" (Frame Check Sequence) in Ethernet documentation, but since IEEE 802.3br specifically calls it CRC, I follow this convention.

Then comes the data, which is always at least 60 bytes (to satisfy Ethernet timing requirements). Finally we have the CRC field. As usual, the CRC field (in 802.3br renamed to "mCRC") is calculated over all bytes following the first 8 bytes. Note that the SMD-Sx, SMD-Cx, and the fragment counter, are not covered by the mCRC; therefore additional measures are necessary to detect any errors in these fields (see below).

The least significant 16 bits of the mCRC are inverted just before transmission. However, for the **last** fragment, the original CRC is used, and the last 16 bits are **not** inverted. This difference with the mCRC's allows the receiver to detect that the last fragment has been received. It also enables the verification that all the data in the concatenation of all received fragments are identical to the original message (figure 9 left). This is a way to detect the integrity of a received message: all fragments taken together must be equal to the original message.

Although there is more error detection built-in, the CRC check is the ultimate check on a reassembled message before it is processed further (or discarded when the CRC does not match).

Backwards compatibility

For any new extension new Ethernet, there is always the question: is it interoperable with older Ethernet equipment that does not support this new functionality? The IEEE has also striven to remain backwards compatible as much as possible. This policy has undoubtedly been fundamental to the popularity of Ethernet, an there is thus also backwards compatibility with equipment that doesn't support frame preemption (read: almost all Ethernet-devices in the world today).

When two devices are not backwards compatible, it is not possible to use frame preemption. Due to the changed implementation of the message header (see figure 9), and the changed implementation of the CRC, a receiver would not accept these messages as being valid Ethernet messages (and just discard them).

Therefore, when frame preemption is intended to be used, two devices must negotiate this with each other. This is done via a different mechanism than used for auto-negotiation. Here, two specially crafted Ethernet message are used, called a "Verify" and a "Respond" message.[2]

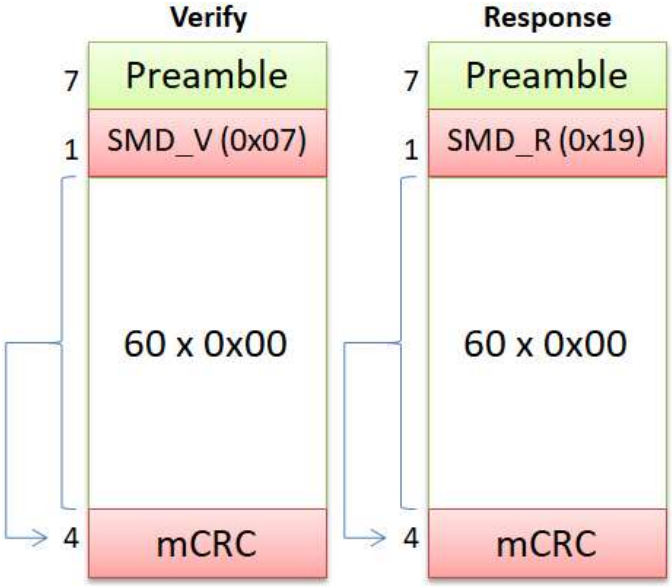


Figure 10: The contents of a "Verify" and of a "Response" message.

[2] Note that there are no MAC-addresses used at the locations where you would normally expect them; this would be impossible anyway because how can a device know the MAC-address of the device at the other end of the cable? It is not necessary to know the MAC-address anyhow, because there can be only one device at the other end of the cable.

A device that wants to *send* frame preempted messages first sends a "Verify" message to the other device (at the other end of the cable). When this device sends a "Respond" message back, it states that it also supports frame preemption, and from that moment on it may be used.

If no valid "Respond" message is received, there is an error – the "Verify" may have been lost in transit, or damaged, or the "Response" may have been lost in transit, or damaged. In such cases, a new attempt (retry) is done; a "Verify" is sent again in the hope that now an answer is received back. If this also fails, a new retry is done. If this (3rd) attempt fails again, it is reported to higher protocol layers.

Because Ethernet is full-duplex, the other device must also send a "Verify" message, and get a "Response" back. Only then can frame preemption be used bidirectional. In normal circumstances this will probably be always the case. It would be strange if a device intends to use frame preemption itself when sending, but not allowing the other side to use it. Nevertheless, theoretically it is allowed to do so.

Once frame preemption is enabled, it remains in use until an Ethernet link failure is detected. This can be caused by the removal of a cable between the two devices, or one device being switched off.

Verification disabling
In closed networks, where the capabilities of devices are guaranteed to support frame preemption, it is not necessary to perform the verification procedure as described above. This makes no difference run-time.

Verification enabling
The verification algorithm does not run automatically. Via the LLDP protocol (Link Layer Discovery Protocol), a device must first indicate that it has frame preemption capabilities. If present, then the verification procedure can begin.

Fragment indication

The most difficult part in frame preemption algorithm is the error handling: any fragment can be lost while 'in transmission' on the cable. It must be assured that this is always properly detected.

The receiving device must have some intelligence in recognizing the fragments, as they must be concatenated in the right order. This sounds simple, as long as they are sent in the right order, how can they be received in the wrong order? In principle this is true in a perfect world. But in reality network messages can be damaged while in transit; one (or more) bits in the fragment are corrupted (inverted), and since Ethernet does not know which bits are involved it cannot repair these errors, so the only sensible action is to discard the whole fragment. Then there remains a gap in the sequence of fragments received: 1,2,4,5… (in case the third fragment is lost). Of course, you would not want to process network messages with missing data.

In order to detect this, each fragment is given a "Fragment Counter" (2 bits in size). Following the start fragment (S), the next fragment gets value 0, subsequent fragments increment this counter modulo 4: S,0,1,2,3,0,1,2,3,0,1… etc.

When the receiving device notices the wrong sequence, for example S,0,1,2,0,2,3… a fragment has gone missing (3) in this case. Even two fragment may be missed, for example S,0,1,0,1… is still seen as wrong. In case three subsequent fragments are missed this is still detected: S, 0,0,1,2,…

Should 4 subsequent fragments be missed, this is no longer detectable with the fragment counter (S,0,1…). It can also not be detected by noticing that less fragments are received than anticipated, because the receiver does not know the total number of fragments. The only way to detect the missing fragments is that the CRC of the reconstructed message does not match (see above).

When one (or more) fragment(s) are lost, it is not possible to reconstruct the original message, since Ethernet performs no repair actions. The receiver will wait for the first fragment of the *next* message (see below).

Of course, just completely discarding an Ethernet message is not good for higher-protocol levels. But it is in line with the standard Ethernet behavior for over 40 years: corrupted data is discarded.

Higher-level protocols may have their own error detection and repair algorithms (i.e. the checksum with retries in TCP/IP). If higher level protocols other than TCP/IP are used, they probably have their own error detection and repair algorithms.

One byte for two bits?
Even though the fragment counter is only 2 bits in size, in the Ethernet message it occupies 8 bits. The four possible values are not sent as-is (00,01,10,11) but as 8-bit values 0xE6, 0x4C, 0x7F and 0xB3, respectively. Why is this done? Remember that the fragment count field is not covered by the mCRC. When using only 2 bits, a bit error in transmission could therefore (unknowingly to us) change one valid fragment counter value (say, 01) into another valid bit pattern (11 or 00). This if of course very dangerous.

The 4 values 0xE6, 0x4C, 0x7F and 0xB3 are chosen because of their mathematical property called "Hamming Distance". Changing any of the 8 bits never produces one of the other three values. For example when taking value 0xE6: changing one bit produces either 0x66, 0xA6, 0xC6, 0xF6, 0xEE, 0xE2, 0xE4 or 0xE7, none of which is one of the other three. Even 2 or 3 faulty bits never produces one of the others. Only when at least 4 bits of any value are inverted one of the listed numbers can result And even then there is only a ¼ chance that the value for the expected fragment count is produced.

This so-called "Hamming Distance" of value 4 is a way to detect errors in network messages. Even though the likelihood that a bit is damaged while in transmission is very small, the change that 4 bits are damaged is even smaller (see below for some mathematics about this).

The first fragment

The first fragment of a message is indicated by a special value in the SMD field: a so-called "SMD Start" value SMD-S. When the receiver sees an SMD-S, it knows that it can discard any previously received fragments (if present) and start the assembly of fragments for a *new* message.

Actually there are four values defined for SMD-S: SMD-S0 (value 0xE6), SMD-S1 (0x4C), SMD-S2 (0x7F) and SMD-S3 (0xB3). The sender cycles through these four values for every new first fragment (SMD-S0, SMD-S1, SMD-S2, SMD-S3, SMD-S0, SMD-S1, etc.). The reason is that the receiver can unambiguously detect that a new first fragment (for a new message) is received. Otherwise, there would be confusion if two subsequent fragments with the same SMD-Sx would be seen: is this twice the same fragment, of the first fragment of two different messages?

> *The reason for the apparently peculiar values of the four SMD-Sx values is not explained in the IEEE standard, but becomes clear when reading the design notes of the IEEE committee. As the Hamming-distance Hd of any combination of two values is at least 4, so many corrupted bits are needed to erroneously change a valid SMD-Sx value to another valid SMD-Sx value. Simple math[3] states that if the chance that one bits get corrupted is (say) 10^{-7}, the chance that 4 bits get corrupted to form another valid SMD-Sx is $(10^{-7})^4 = 10^{-28}$.*

The receiver can distinguish corrupted SMD-S values from correct values up to a maximum of 4 corrupted bits. The reason for having this dedicated error detection capability is that the CRC/mCRC (at the end) does not cover the SFD field.

If a fragment is received which does not contain an SMD-S0, S1, S2 or S3 it is ignored.

Continuation fragments

Continuation fragments of a message are sent according the new message format (figure 9 right). The SMD field contains a so-called "SMD Continuation" value SMD-C.

Which SMD-C is used depends on the original SMD-S. When it was SMD-S0, the continuation fragments contain an SMD-C0; when it was an SMD-S1, the continuation fragments use an SMD-

[3] Ignoring burst errors for simplicity's sake.

C1, etc. This assures that the receiver always knows to what network message the continuation fragment belongs. This is needed in case of a complete loss of a fragment.

Suppose that in normal circumstances the following fragments are transmitted: (1) SMD_S0, (2) SMD-C0 (first fragment), (3) SMD-C0 (2nd and last fragment), (4) SMD-S0, (5) SMD-C0 (first fragment), (6) SMD-C0 (second fragment), etc.

There are several different error scenarios:

> (1) Suppose that message (3) is completely lost. The receiver now gets an SMD-S again, so it knows a new message is started without the previous message completely having been received. The two fragments are discarded. Ethernet proceeds with processing message (4) and beyond.
>
> (2) Suppose that messages (3) **and** (4) are lost. The receiver gets two fragments, both with the same fragment counter. This is not allowed, all received fragments are discarded, processing continues with the next SMD-S in message (7) or later.
>
> (3) Suppose that messages (3) and (4) **and** (5) are lost. The receiver gets two fragments with succeeding fragment counters 0 and 1. This is as it is supposed to be! But it is still wrong, as fragments from two different messages are now concatenated.

In order to detect scenario 3, Ethernet gives the second group of fragments a different SMD_S value, i.e. SMD_S1 in our example. On the line we would then see: (1) SMD_S0, (2) SMD-C0 (first fragment), (3) SMD-C0 (2nd and last fragment), (4) SMD-S1, (5) SMD-C1 (first fragment), (6) SMD-C1 (second fragment), etc.

If messages (3) and (4) and (5) are now lost, the receiver gets an SMD-C0 (first fragment) followed by an SMD-C1 (second fragment), so now sees that they belong to two different messages (C0 and C1). The error is now detected.

The last fragment

Every preempted frame has a last fragment. How does the receiver recognize it? The fragment counter cannot be used for it, it just counts 0,1,2,3,0,1,2,3… etc. Ethernet now employs a trick: it uses a differently calculated CRC (called mCRC) on all fragments but the last: the 16 least-significant bits are inverted (a 0 becomes a 1, a 1 becomes a zero). The last fragment has the CRC value as originally calculated for the original Ethernet message as it would be if no fragmentation would have taken place[4].

The receiver can detect these differences, and so it knows it now has received all fragments. They can be concatenated together, the original 'long' network message is now available (including the original CRC) and it can be passed on to higher protocol layers.

When earlier a fragment has been lost (due to data corruption) or has not been received at all, the receiver can of course not reassemble the original network message. Reception of the last message with the normal CRC now signals that it can discard all received fragments, and prepare for receiving the first fragment of a new network message.

Fragment assembly

The receiver assembles (concatenates) all received fragment as long as the fragment counter (in the message) corresponds to the expected fragment counter. If there is a mismatch, the most likely reason is that a fragment has been lost while in transit, usually caused by data corruption.

[4] On its own this is very strange, as this last fragment thus has a CRC which does *not* match the data it contains. The IEEE apparently does this for efficiency reasons.

The receiver then simply discards all received fragments, and starts waiting for a new first fragment (with an SMD-Sx), or an express message, or a verify/respond message. Any following fragments received are ignored; this is caused by the sender not knowing that a fragment is missing, it continues to send all remaining fragments until all have been done. This is a small bandwidth loss, but acceptable as long as it doesn't happen too often.

3 Various

"PAUSE" frames

Ethernet has a feature called "Pause frames". These are useful when a device is not capable of processing any more incoming network messages. It then sends a so-called "Pause frame" to a device, instructing it to stop transmitting for a (configurable) number of milliseconds. When this time has elapsed, or the (remaining) pause period is cancelled, transmission may continue again.

When this is used in applications requiring real-time behavior, the usage of pause frames completely disrupts the cyclic transmission of network messages, as they may come at unexpected moments. Therefore, the IEEE 802.3br specifies that the "Pause" feature is to be disabled.

But didn't QoS do this already?

The 'Quality of Service' (QoS) features of Ethernet are often used to implement prioritization of Ethernet, the higher the priority, the quicker a message is processed in a switch. That is, the message is placed at the front of the outgoing queue for a specific port. But: it is still a queue. Any traffic already waiting in the queue must be sent out first. So there is still some unpredictability, of course less than normal, but there still is some.

Figure 11: Quality of Service priority for public transport: priority lanes improve throughput, but red traffic lights add delays.

QoS and frame preemption are not competitors. Given one of the eight QoS priorities, there can be express messages and normal messages mixed in that priority queue. This means that express messages waiting in a low-priority QoS queue still have to wait.

A difference between QoS and frame preemption is the way the administration is done. With QoS, an Ethernet message carries (in the VLAN tag) its priority-information all the way: from sender to receiver. With frame preemption, the Ethernet message is sent from one device to the next (for example, device > switch > switch > switch > device), according configuration settings on all devices and software settings. The timely arrival of message on its final destination depends on what happens on all (4 in this example) intermediate links, which must be individually properly configured. IEEE 802.3br has no influence on this, but there are other standards in the IEEE TSN standard group that help here.

Differences with TCP/IP

Those familiar with the internals of the TCP/IP protocol suite will probably recognize that there is some similarity. TCP/IP also can fragment and reassembly data, check for missing fragments, handle errors, etc. Why is TCP/IP not used, and has the IEEE invested in inventing frame preemption ?

The reason is that Ethernet and TCP/IP both operate on different levels: Ethernet at wire-level (OSI-layers 1 and 2), and TCP/IP at internet-level (OSI-layers 3 and 4). TCP handles the "end-to-end" transport of data, from the original sender to some other device at home, in your company or somewhere on internet. TCP is very capable in detecting and repairing any errors occurring between source and destination.

TCP does not know of any underlying technology which handles the transmission of data; it lets IP handle this. IP has some knowledge of Ethernet limitations, for example the maximum message size of 1500 bytes. Larger amounts of data are fragmented, and reassembled at the destination. When a piece of data goes missing, TCP repairs it. In principle this works between any two devices anywhere on a network, in a company, or on internet.

Ethernet frame preemption only knows about two devices: at both ends of the **cable** (usually: the device and its switch). It does its best to send the data to the device at the other end of the cable, but does nothing regarding handling errors – it detects them, but does not repair any errors. This is left to higher protocol layers (such as TCP).

This article is likely not complete in describing all possible frame preemption features. If you have any suggestions, comments or additions, please do not hesitate to contact me (email: rh[at]enodenetworks.com).