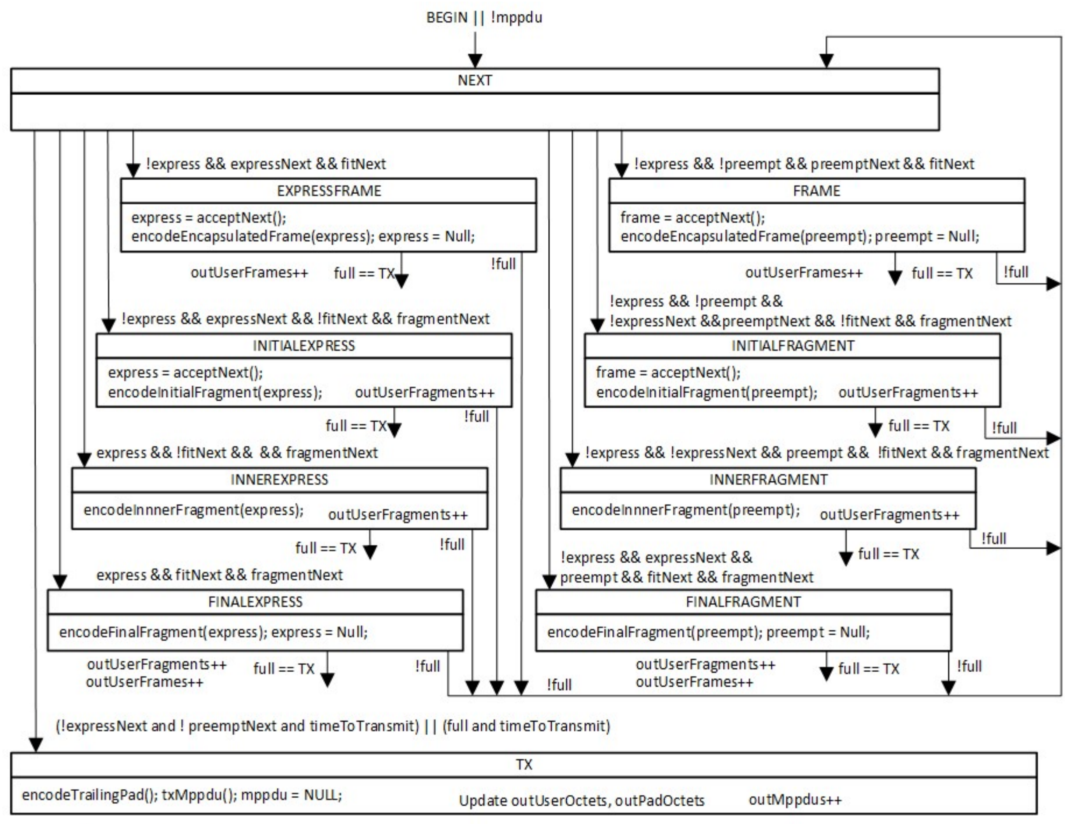
MAC Privacy Frame Ordering

By Don Fedyk April 2021

This paper reviews the aspects of ordering as encode and the subsequent decoding and recommends the standards language to be used. It is a working document.



BEGIN || !mppdu
NEXT
!express && expressNext && fitNext
EXPRESSFRAME
express = acceptNext();
encodeEncapsulatedFrame(express); express = Null;
outUserFrames++
full == TX
!full
!express && !preempt && preemptNext && fitNext
FRAME
frame = acceptNext();
encodeEncapsulatedFrame(preempt); preempt = Null;
outUserFrames++
full == TX
!full
!express && expressNext && !fitNext && fragmentNext
INITIALEXPRESS
express = acceptNext();
encodeInitialFragment(express);
outUserFragments++
full == TX
!full
!express && !preempt &&
!expressNext &&preemptNext && !fitNext && fragmentNext
INITIALFRAGMENT
frame = acceptNext();
encodeInitialFragment(preempt);
outUserFragments++
full == TX
!full
express && !fitNext && && fragmentNext
INNEREXPRESS
encodeInnnerFragment(express);
outUserFragments++
full == TX
!full
!express && !expressNext && preempt && !fitNext && fragmentNext
INNERFRAGMENT
encodeInnnerFragment(preempt);
outUserFragments++
full == TX
!full
express && fitNext && fragmentNext
FINALEXPRESS
encodeFinalFragment(express); express = Null;
outUserFragments++
outUserFrames++
full == TX
!full
!express && expressNext &&
preempt && fitNext && fragmentNext
FINALFRAGMENT
encodeFinalFragment(preempt); preempt = Null;
outUserFragments++
outUserFrames++
full == TX
!full
!full
(!expressNext and ! preemptNext and timeToTransmit) || (full and timeToTransmit)
TX
encodeTrailingPad(); txMppdu(); mppdu = NULL;
Update outUserOctets, outPadOctets
outMppdus++

State machine conditions and procedures:

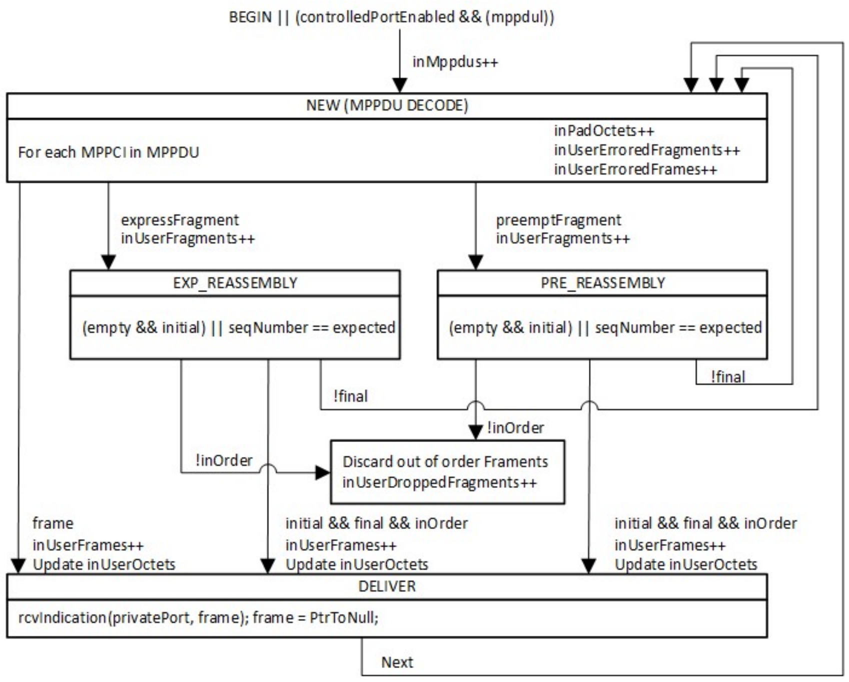
mppdu : True iff (if and only if) an MPPDU has been generated and not yet transmitted.
express : True iff the PrY is holding the remainder or all of an Express user data frame.
fitNext : True iff the any frame remainder can be encoded in the remaining MPPDU octets.
fragmentNext : True iff the next Frame remainder can be fragmented
expressNext : True iff the PrY's user has a frame available for transmission, and has currently selected an Express frame for its next transmission.
preempt : True iff the PrY is holding the remainder or all of a Preemptable user data frame.
preemptNext : True iff the PrY's user has a frame available for transmission, and has currently selected a Preemptable frame for its next transmission.
full: True iff the MPPDU has less than the minimum fragment length left.
express = acceptNext() : Accept the next user data frame (an Express frame) for transmission, similarly frame = acceptNext() for a Preemptable frame.
encodeEncapsulatedFrame(express), encodeEncapsulatedFrame(preempt) : Encode the user frame in the MPPDU, reducing the number of octets remaining.
encodeInitialFragment(express), encodeInitialFragment(preempt) : Encode an Initial Fragment, encapsulating the greatest multiple of 64 octets that will fit in the MPPDU leaving at least 64 octets of the frame as a remainder.
encodeInnerFragment(express), encodeInnerFragment(preempt) : Encode a Frame Fragment (with Initial and Final bits clear), encapsulating the greatest multiple of 64 octets that will fit in the MPPDU leaving at least 64 octets of the frame as a remainder
encodeFinalFragment(express),encodeFinalFragment(preempt) : Encode the remainder of the user data frame in a Final Fragment.
encodeTrailing Pad() : Encode the value 0 in all the remaining octets (if any) of the MPPDU.
txMppdu() : Transmit the MPPDU through the PrY's Controlled Port.
Statistic update points are illustrated with outXxx where appropriate counters are adjusted

Figure 1 Encode State machine

MAC privacy has an encoding format that maintains ordering in keeping with current Ethernet goals. Frames are processing in order. MAC privacy has an express class and a preemptable class that allows Express frames to be encoded before Preemptable class frames when present at the same time allowing recording between classes.

MAC Privacy uses encapsulation using Ethernet frames that are organized by Mac Privacy Packet Data Units (MMPDUs) contain one or more MAC Privacy Component Instances (MPPCIs) that contain either whole frames or fragmented frames with an Initial, Final and sequence number indication. Express frames are all encoded and transmitted in order with respect to the express class and the Preemptable class frames are all encoded in ordered with respect to the express class.

This also means that once a express frame is fragmented all express fragments for that frame are sent before any other whole express frames. Similarly, all preemptible fragments for a preemptable frame are sent before any other whole preemptable frames. Express fragments and frames may preempt the encoding of preemptable frames when the share the same MAC privacy channel (MPPDU structure) . The state machine for encoding is illustrated in Figure 1. Note that the express and preemptable frames can be in one MAC privacy channel or in two separate MAC Privacy channels each with their own access priority.



controlledPortEnabled : Enabling contion.
empty : True iff the assemby has no pending fragments.
expressFragment : True iff the fragment has a fragment header and an express indication
preeemptFragment : True iff the fragment has an express indication false and a fragment header
initial : True iff the fragment is an initial fragment
final:True iff the fragment is a final fragment
inOrder:True iff all the current fragments are in order
seqNumber : the sequence number of the current fragment.
expected: True iff the sequence number received is the next expected sequence number
frame: True iff the MPPCI indicates a frame
mppdu: True iff the frame is an MAC Privacy PDU
Statistic update points are illustrated with inXxx where appropriate counters are adjusted

Figure 2 Decode State Machine

When MAC Privacy frames are transmitted, they are encrypted and protected by MACsec on the same device or on devices in close proximity.

Figure 2 illustrates the decode state machine. This decapsulates frames and fragments and reassembles the fragments and transmits both. Ordering for the encode state machine is predictable. A sender may or may not choose to always strictly order express frames before preemptable frames, but a preference should be given to express frames over preemptable frames.

Receiver Side Miss ordering Conditions

Normally the reception by MACsec prevents any miss ordering. However there are cases to be considered that could cause ordering issues.

Frame Drops

MAC privacy may be collocated with MACsec or may be in a separate device usually as close as possible to the device doing MACsec. This means MAC Privacy will normally be decoding in order frames with the occasional dropped frame if there is a transmission error or congestion. If an MPPDU with whole frames is dropped, this is undetectable by MAC privacy since the MPCI for whole frames does not contain any sequence information. If an MMPPU with a fragment is dropped this will be detected and any affected frames will be discarded.

Fragment Strict order checking Cases

1. Fragment sequence number unexpected
2. Initial fragment indication missing
3. Final fragment indication missing
4. Sequence number missing between Initial and Final indication

Any Frame drop will be detected by these conditions unless the sequence number is repeated and happens to lose exactly enough frame such that the sequence number is expected.

Therefore, one additional check is that fragments should not remain in the reassembly area long enough for the sequence number to wrap and a source should never repeat a sequence number in less than that time. Reasonable time periods are based on the maximum rollover time of the sequence numbers which is based on the transmission speed. A rule of thumb is 100 ms for a 10Gb/s link rate assuming 100% minimum fragments of 64 bytes (an unlikely condition).

The frame handling rules for these cases are in the Ordering: Actionable cases section.

Frame Miss ordering

This case can be totally prevented by MACsec as mentioned earlier and frame loss is the only type of condition that would be encountered. However, if MACsec is set to allow a small amount of miss ordering MAC privacy might encounter MMPDUs that are miss ordered.

Ordering MAC Privacy Undetectable cases

While MACsec can detect lost, or miss ordered frames MAC privacy may not always be able to. The first case of miss ordering is when the MPPDUs carry only full unfragmented frames or padded frames. There is no way for MAC privacy to detect miss ordering in this example, and the end systems would have to decide to drop or reorder the subsequent data. There are instances where if an MPPDU carrying an express frame is overtaken by MPPDU carrying a preemptable frame there is no impact other than additional transmit delay. Similarly if pure padded MPPDUs are miss ordered, with MPPDUs, there may be no end system impact.

The second example is when a frame carrying an initial fragment arrives earlier than it should (a prior MPPDU is delayed for example) but there are no other fragments for the same class between the next in order fragment. This is similar to the full frame examples except pertaining to fragments. A miss ordering has happened, but it is undetectable. Similarly a final fragment could be delayed by a miss ordering event and there may be only a slight delay but no loss.

Ordering: Unactionable cases

There is a class of subtle cases where miss ordering can be detected. These detection cases are not worth acting on unless they become an actionable case.

Note that since the preemptable reassembly and the express reassembly may be handled by separate queues at the source, no inferences between received order of frames versus fragments can be assumed. Therefore, while a receiver could expect that no whole frames should arrive until a fragmented express stream is completely received, there is no logic to prevent this at a receiver because in some cases this event is valid with no miss ordering.

Within an MPPDU the MPPCI should all have the correct order. The logic in the state machines does not enforce any particular rules, it is a stream of MPPCI components versus any MPPDU based logic. This means that a large MPPDU could contain all fragments for a frame – a rational case for a preemptable frame preempted by an express frame – but not expected for an express frame.

Ordering: Actionable cases

The actionable cases for miss ordering fall into the same categories as the frame drop cases. Receiving an MPPDU with fragments late or out of order may appear the same as if it had been dropped for strict ordering. If any miss ordered MPPDUs contain fragments of the same class, then it will be detected.

Strict order checking cases:

1. Fragment sequence number unexpected
2. Initial indication missing
3. Final indication missing
4. Sequence number missing between Initial and Final indication

The action for case 1 is to purge any fragments in the reassembly area. If the MPPCI just received is a initial fragment, reassembly for that fragment can start and the sequence number is reset to match this MPPCI and the next sequence number expected is adjusted. Every fragment is checked when decoded for Case 1.

The action for case 2 is to purge fragments until a fragment with an initial indication is received.   Any whole frames can be processed. In most instances, Case 1 would catch this as well – i.e. a frame could not get by check 1 if there was not initial indication. It could be a while before another fragment comes along with an initial indication, so this runs into case 3.

The action for case 3 is to purge fragments after a subsequent initial fragment indication is received, or the timeout for reassembly queue happens, whichever occurs first. Any complete whole frames can be processed.

The action for case 4 is to purge the fragments in the reassembly queue. Note that with strict ordering this case never happens since case 1 would prevent the inclusion any fragments that were not in order.

Sequence recovery is achieved by synchronizing the sequence number to the next fragment with an initial indication for the respective class, express or preemptable. If subsequent MPPCIs are in order processing continues from that point on otherwise the process for the ordering cases occurs again.

The intention in the standard specification is simply worded that all complete sequences of in order fragments with initial and final indication will be reassembled and delivered.

A frame with and initial and final indication MUST have its sequence number set and it is a valid frame. It follows the same rules, so it is always accepted by the logic of cases 2 and 4.

Reordering (Unsupported) Considerations

The question arises in the case of miss ordering is there a less strict ordering condition that allows the delivery of more frames? This is currently not specified by the base specification. This text points out some implications to be considered.

The considerations for allowing reordering are as follows. The first issue with supporting detectable miss ordered frames is the support of an acceptable window of sequence number (changing case 1). The second issue is there is no way to discern loss and or delayed delivery of a frame when first detected.

For the strict ordering case the window is zero. The next sequence number must match or the whole frame (all fragments) will be discarded except this fragment if it an initial fragment .

Addition of a small window of acceptable sequence numbers requires both accepting sequence numbers beyond the accepted within that window(leading window) and some number of skipped sequence number from those received up to the new sequence number (trailing window). The sequence number also wraps so the window must take that into account. The issue with windows here is that the fragments are not always encoded at the source. Therefore the storage of fragments based on a stream of fragment sequence numbers is event driven. After some time passes the frames should be discarded in it is a real loss. This timeframe is rate dependent but probably in the 100 ms range as explained earlier.

When processing an MPPCI the cases would have to invoke logic along the lines of creating partially assembled data structures (or other logic that achieves the same). The rule then becomes accepting fragments within the sliding windows, reordering the fragments in the partially assembled area, discarding fragments that are now beyond the sliding windows and delivering reassembled frames as soon as they have an initial and final fragment with all consecutive fragment sequence numbers in between (zero or more).

There is nothing that prevents a device maker from implementing this more elaborate reassembly, but it is considered outside the scope of the standard.

Recommendation

Strict ordering should be the only option specified in the standard, but it should not be specified to preclude other mechanisms. In other words the order enforcement is at the sole discretion of the receiving side.

A configuration option for reducing or turning off fragmentation for lower quality links might improve the number of frames delivered at the cost of some link efficiency.

Prototype state machine Validation

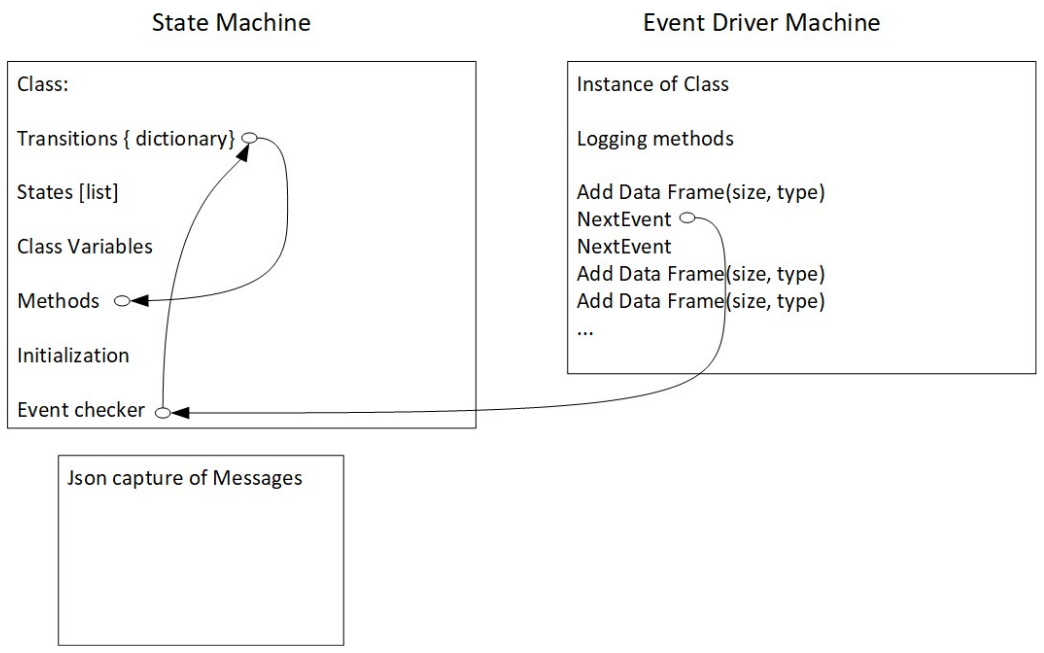


Figure 3 Prototype Encode state machine

The State machine logic for the state machines has been prototyped for strict ordering with a small python state machine. Figure 3 illustrates the basic Encode state machine. (there is one for encode and decode and the organization is the same.) There is a Python class that contains the complete state machine logic – to any degree of accuracy required. This is a non-real time machine that uses a transition package which allows the state table to be captured in a single python dictionary (essentially a table of function calls). An event checker is called for each step that creates an event based on the current state and any other inputs. Messages can be injected into the machine and a JSON structure representing the encoding (without padding) is saved to a file. This file can then be run through the second state machine to validate the decode logic.

Loss and miss ordering can be simulated by editing the intermediate JSON files.

Event Logic

A one-to-one logic of events to states is captured in the conditions. Comparing to the encoding state machine we can see these conditions create the events that drive the state machine.

```
    def __express_frame_c (self):
        """ Private method creating an express frame event """
        return not self.express and self.express_next and self.fit_next

    def __initial_express_c (self):
        """ Private method creating an initial express frame event """
        return (not self.express and self.express_next and
                not self.fit_next and self.fragment_next)

    def __inner_express_c (self):
        """ Private method creating an inner express frame event """
        return self.express and not self.fit_next and self.fragment_next

    def __final_express_c (self):
        """ Private method creating an final express frame event """
        return self.express and self.fit_next and self.fragment_next

    def __preemptable_c (self):
        """ Private method creating an preempt frame event """
        return (not self.express and not self.preempt and
                self.preempt_next and self.fit_next)

    def __initial_preemptable_c (self):
        """ Private method creating an initial preempt frame event """
        return (not self.express and not self.preempt and not self.express_next and
                self.preempt_next and not self.fit_next and self.fragment_next)

    def __inner_preemptable_c (self):
        """ Private method creating an inner preempt frame event """
        return (not self.express and not self.express_next and self.preempt and
                not self.fit_next and self.fragment_next)

    def __final_preemptable_c (self):
        """ Private method creating an final preempt frame event """
        return (not self.express and not self.express_next and self.preempt and
                self.fit_next and self.fragment_next)
```

Sample Trace

Input: A serial number indicating the order of reception of the frames has been added for tracking the frames, but this number is informational only can plays no part in the state machine logic. Additional Timing information not illustrated, not all frame arrived at the same time.) Also the express/preempt logic is derived from the Priority in practice. In the simulation data here it is expressly indicated but will not show up for full frames at the egress since the original frame priority carries this information (it is not required but could be added if desired).

```
{'length': 1200, 'serial_num': 0, 'express': True}
{'length': 1200, 'serial_num': 1, 'express': True}
{'length': 1400, 'serial_num': 2, 'express': False}
{'length': 1400, 'serial_num': 3, 'express': True}
{'length': 1200, 'serial_num': 4, 'express': True}
{'length': 140, 'serial_num': 5, 'express': False}
{'length': 140, 'serial_num': 6, 'express': False}
{'length': 140, 'serial_num': 7, 'express': False}
{'length': 1200, 'serial_num': 8, 'express': True}
{'length': 140, 'serial_num': 9, 'express': False}
{'length': 140, 'serial_num': 10, 'express': False}
{'length': 460, 'serial_num': 11, 'express': False}
{'length': 1400, 'serial_num': 12, 'express': False}
```

Output of first State machine encoding from the input (13 frames in 8 MPPDUs). (This represents the MPPDUs {1} and the MPPCIs {}{1..N} The MPPDU can carry 1500 bytes of data. {overhead is not computed for this simulation} Paddind is

```
{'1': {'length': 1200, 'serial_num': 0, 'express': None, 'seq': None, 'initial': None, 'final': None, 'pad': None}, '2':
    {'length': 300, 'serial_num': None, 'express': None, 'seq': None, 'initial': None, 'final': None, 'pad': True}}
{'1': {'length': 1200, 'serial_num': 1, 'express': None, 'seq': None, 'initial': None, 'final': None, 'pad': None}, '2':
    {'length': 300, 'serial_num': 2, 'express': False, 'seq': 0, 'initial': True, 'final': False, 'pad': None}}
{'1': {'length': 1400, 'serial_num': 3, 'express': None, 'seq': None, 'initial': None, 'final': None, 'pad': None}, '2':
    {'length': 100, 'serial_num': 4, 'express': True, 'seq': 0, 'initial': True, 'final': False, 'pad': None}}
{'1': {'length': 1100, 'serial_num': 4, 'express': True, 'seq': 1, 'initial': False, 'final': True, 'pad': None}, '2': {'length':
    400, 'serial_num': 2, 'express': False, 'seq': 1, 'initial': None, 'final': None, 'pad': None}}
{'1': {'length': 700, 'serial_num': 2, 'express': False, 'seq': 2, 'initial': False, 'final': True, 'pad': None}, '2': {'length':
    800, 'serial_num': 8, 'express': True, 'seq': 2, 'initial': True, 'final': False, 'pad': None}}
{'1': {'length': 400, 'serial_num': 8, 'express': True, 'seq': 3, 'initial': False, 'final': True, 'pad': None}, '2': {'length':
    140, 'serial_num': 5, 'express': None, 'seq': None, 'initial': None, 'final': None, 'pad': None}, '3': {'length': 140,
    'serial_num': 6, 'express': None, 'seq': None, 'initial': None, 'final': None, 'pad': None}, '4': {'length': 140,
    'serial_num': 7, 'express': None, 'seq': None, 'initial': None, 'final': None, 'pad': None}, '5': {'length': 140,
    'serial_num': 9, 'express': None, 'seq': None, 'initial': None, 'final': None, 'pad': None}, '6': {'length': 140,
    'serial_num': 10, 'express': None, 'seq': None, 'initial': None, 'final': None, 'pad': None}, '7': {'length': 400,
    'serial_num': None, 'express': None, 'seq': None, 'initial': None, 'final': None, 'pad': True}}
{'1': {'length': 460, 'serial_num': 11, 'express': None, 'seq': None, 'initial': None, 'final': None, 'pad': None}, '2':
    {'length': 1040, 'serial_num': 12, 'express': False, 'seq': 3, 'initial': True, 'final': False, 'pad': None}}
{'1': {'length': 360, 'serial_num': 12, 'express': False, 'seq': 4, 'initial': False, 'final': True, 'pad': None}, '2': {'length':
    1140, 'serial_num': None, 'express': None, 'seq': None, 'initial': None, 'final': None, 'pad': True}}
{'1': {'length': 1500, 'serial_num': None, 'express': None, 'seq': None, 'initial': None, 'final': None, 'pad': True}}
```

Output of second state machine showing reassembly. This state machine breaks the MPPDUs back into MPPCI and processes each one in the order received. Compare to this the original stream.

```
{'length': 1200, 'serial_num': 0, 'express': None}
{'length': 1200, 'serial_num': 1, 'express': None}
{'length': 1400, 'serial_num': 3, 'express': None}
{'length': 1200, 'serial_num': 4, 'express': True}
{'length': 1400, 'serial_num': 2, 'express': False}
{'length': 1200, 'serial_num': 8, 'express': True}
{'length': 140, 'serial_num': 5, 'express': None}
{'length': 140, 'serial_num': 6, 'express': None}
{'length': 140, 'serial_num': 7, 'express': None}
{'length': 140, 'serial_num': 9, 'express': None}
{'length': 140, 'serial_num': 10, 'express': None}
{'length': 460, 'serial_num': 11, 'express': None}
{'length': 1400, 'serial_num': 12, 'express': False}
```

A couple points on the data: express/preempt indication is unknown = None when a whole frame is encapsulated. . (Booleans in Python dictionaries have three values True, False and None meaning does not exist) In a real implementation this data is normally stripped, and the original user priority is restored. It is illustrated here since it indicates reassembled frames as seen by the receiving MAC Privacy entity. The Serial number is for illustrative purposes only it shows that express frames do get processed ahead of preempt frames and fragments.

Ordering logic in the receiving state machine

The logical ordering presented here is captured in the state machine. This shows a rough implementation of the 4 cases when processing an express or preemptable fragment.

(the data structures are an incoming MPPCI list of a python dictionary representation of an MPPCI, and an assembly list per express/preemptable for ordered fragments (dictionary).

```
def process_fragment (self):
    # Check in sequence
    if self.frame_list:
        fragment = self.frame_list.pop(0)
        if fragment["express"] is True:
           exp_pre = EXPRESS
        else:
           exp_pre = PREEMPT
        print ("Frame {} ".format(fragment))
        initial = fragment["initial"]
        final = fragment["final"]
        seq = fragment["seq"]
        is_expected = self.check_expected(exp_pre, seq) #Case 1 & 2
        if is_expected:
            self.assembly_list[exp_pre].append(fragment.copy())
            self.seq_num[exp_pre] = self.seq_increment (self.seq_num[exp_pre])
            # Check in order
            is_in_order = self.__check_in_order(exp_pre) # Case 4
            if not is_in_order:
                print ("Purging {}".format(self.assembly_list[exp_pre]))
                for _ in range(len(self.assembly_list[exp_pre])):
                    self.in_user_dropped_fragments += 1
                self.assembly_list[exp_pre].clear()
            #  Initial and Final and in order
            if self.assembly_list[exp_pre]:
                if final and self.assembly_list[exp_pre][0]["initial"] is True: # Case 2 & 3
                    self.__set_ready(exp_pre)
        else:
            print ("Purging {}".format(self.assembly_list[exp_pre]))
            for _ in range(len(self.assembly_list[exp_pre])):
                self.in_user_dropped_fragments += 1
            self.assembly_list[exp_pre].clear()
            if initial and seq > self.seq_num[exp_pre]: # Case 3
                self.seq_num[exp_pre] = self.seq_increment (self.seq_num[exp_pre])
```

```
                        self.assembly_list[exp_pre].append(fragment.copy())
```

Sample Run of Encode state machine output:

This section is purely informational – it shows alignment with the state machine diagram.

```
$ python state-test-encode.py firstTest.json
state = NEXT
Adding data set_express_next length 1200
Event = express_frame
State:EXPRESSFRAME Encode Left:300 Exp left:[] Prt left:[]
express:False express_next:False fit_next:True fragment_next:True preempt:False preempt_next:False
out_mppdus:0 out_user_frames:1 out_user_octets:1200 out_pad_octets:0 out_user_fragments:0
Event = next
State:NEXT Encode Left:300 Exp left:[] Prt left:[]
express:False express_next:False fit_next:True fragment_next:True preempt:False preempt_next:False
out_mppdus:0 out_user_frames:1 out_user_octets:1200 out_pad_octets:0 out_user_fragments:0
Event = tx
State:TX Encode Left:1500 Exp left:[] Prt left:[]
express:False express_next:False fit_next:True fragment_next:True preempt:False preempt_next:False
out_mppdus:1 out_user_frames:1 out_user_octets:1200 out_pad_octets:300 out_user_fragments:0
Event = next
State:NEXT Encode Left:1500 Exp left:[] Prt left:[]
express:False express_next:False fit_next:True fragment_next:True preempt:False preempt_next:False
out_mppdus:1 out_user_frames:1 out_user_octets:1200 out_pad_octets:300 out_user_fragments:0
Adding data set_express_next length 1200
Adding data set_preempt_next length 1400
Event = express_frame
State:EXPRESSFRAME Encode Left:300 Exp left:[] Prt left:[{'length': 1400, 'serial_num': 2}]
express:False express_next:False fit_next:False fragment_next:True preempt:False preempt_next:True
out_mppdus:1 out_user_frames:2 out_user_octets:2400 out_pad_octets:300 out_user_fragments:0
Event = next
State:NEXT Encode Left:300 Exp left:[] Prt left:[{'length': 1400, 'serial_num': 2}]
express:False express_next:False fit_next:False fragment_next:True preempt:False preempt_next:True
out_mppdus:1 out_user_frames:2 out_user_octets:2400 out_pad_octets:300 out_user_fragments:0
Event = initial_preemptable
State:INITIALFRAGMENT Encode Left:0 Exp left:[] Prt left:[{'length': 1100, 'serial_num': 2}]
express:False express_next:False fit_next:False fragment_next:True preempt:True preempt_next:True
out_mppdus:1 out_user_frames:2 out_user_octets:2700 out_pad_octets:300 out_user_fragments:1
Adding data set_express_next length 1400
Adding data set_express_next length 1200
Event = tx
State:TX Encode Left:1500 Exp left:[{'length': 1400, 'serial_num': 3}, {'length': 1200, 'serial_num': 4}] Prt
left:[{'length': 1100, 'serial_num': 2}]
express:False express_next:True fit_next:True fragment_next:True preempt:True preempt_next:True
out_mppdus:2 out_user_frames:2 out_user_octets:2700 out_pad_octets:300 out_user_fragments:1
Event = next
```

```
State:NEXT Encode Left:1500 Exp left:[{'length': 1400, 'serial_num': 3}, {'length': 1200, 'serial_num': 4}]
Prt left:[{'length': 1100, 'serial_num': 2}]
express:False express_next:True fit_next:True fragment_next:True preempt:True preempt_next:True
out_mppdus:2 out_user_frames:2 out_user_octets:2700 out_pad_octets:300 out_user_fragments:1
Event = express_frame
State:EXPRESSFRAME Encode Left:100 Exp left:[{'length': 1200, 'serial_num': 4}] Prt left:[{'length': 1100,
'serial_num': 2}]
express:False express_next:True fit_next:False fragment_next:True preempt:True preempt_next:True
out_mppdus:2 out_user_frames:3 out_user_octets:4100 out_pad_octets:300 out_user_fragments:1
Event = next
State:NEXT Encode Left:100 Exp left:[{'length': 1200, 'serial_num': 4}] Prt left:[{'length': 1100,
'serial_num': 2}]
express:False express_next:True fit_next:False fragment_next:True preempt:True preempt_next:True
out_mppdus:2 out_user_frames:3 out_user_octets:4100 out_pad_octets:300 out_user_fragments:1
Event = initial_express
State:INITIALEXPRESS Encode Left:0 Exp left:[{'length': 1100, 'serial_num': 4}] Prt left:[{'length': 1100,
'serial_num': 2}]
express:True express_next:True fit_next:False fragment_next:True preempt:True preempt_next:True
out_mppdus:2 out_user_frames:3 out_user_octets:4200 out_pad_octets:300 out_user_fragments:2
Event = tx
State:TX Encode Left:1500 Exp left:[{'length': 1100, 'serial_num': 4}] Prt left:[{'length': 1100,
'serial_num': 2}]
express:True express_next:True fit_next:True fragment_next:True preempt:True preempt_next:True
out_mppdus:3 out_user_frames:3 out_user_octets:4200 out_pad_octets:300 out_user_fragments:2
Adding data set_preempt_next length 140
Adding data set_preempt_next length 140
Adding data set_preempt_next length 140
Event = next
State:NEXT Encode Left:1500 Exp left:[{'length': 1100, 'serial_num': 4}] Prt left:[{'length': 1100,
'serial_num': 2}, {'length': 140, 'serial_num': 5}, {'length': 140, 'serial_num': 6}, {'length': 140,
'serial_num': 7}]
express:True express_next:True fit_next:True fragment_next:True preempt:True preempt_next:True
out_mppdus:3 out_user_frames:3 out_user_octets:4200 out_pad_octets:300 out_user_fragments:2
Event = final_express
State:FINALEXPRESS Encode Left:400 Exp left:[] Prt left:[{'length': 1100, 'serial_num': 2}, {'length': 140,
'serial_num': 5}, {'length': 140, 'serial_num': 6}, {'length': 140, 'serial_num': 7}]
express:False express_next:False fit_next:False fragment_next:True preempt:True preempt_next:True
out_mppdus:3 out_user_frames:4 out_user_octets:5300 out_pad_octets:300 out_user_fragments:3
Event = next
State:NEXT Encode Left:400 Exp left:[] Prt left:[{'length': 1100, 'serial_num': 2}, {'length': 140,
'serial_num': 5}, {'length': 140, 'serial_num': 6}, {'length': 140, 'serial_num': 7}]
express:False express_next:False fit_next:False fragment_next:True preempt:True preempt_next:True
out_mppdus:3 out_user_frames:4 out_user_octets:5300 out_pad_octets:300 out_user_fragments:3
Event = inner_preemptable
```

```
State:INNERFRAGMENT Encode Left:0 Exp left:[] Prt left:[{'length': 700, 'serial_num': 2}, {'length': 140,
'serial_num': 5}, {'length': 140, 'serial_num': 6}, {'length': 140, 'serial_num': 7}]
express:False express_next:False fit_next:False fragment_next:True preempt:True preempt_next:True
out_mppdus:3 out_user_frames:4 out_user_octets:5700 out_pad_octets:300 out_user_fragments:4
Event = tx
State:TX Encode Left:1500 Exp left:[] Prt left:[{'length': 700, 'serial_num': 2}, {'length': 140, 'serial_num':
5}, {'length': 140, 'serial_num': 6}, {'length': 140, 'serial_num': 7}]
express:False express_next:False fit_next:True fragment_next:True preempt:True preempt_next:True
out_mppdus:4 out_user_frames:4 out_user_octets:5700 out_pad_octets:300 out_user_fragments:4
Etc…
```

Sample output from the decode state machine

Again this is purely informational. Observe that express/preempt is only available for fragments. While the encoding was aware of this for ordering the decode machine is not aware for decoding (the user frames carry the real user priority. In this example padding has been added and is recorded.

```
$ python state-test-decode.py firstTest.json firstConvertBack.json
state = NEW
Event = processFrame
Sending {'length': 1200, 'serial_num': 0, 'express': None}
State:DELIVER
express_next:False express_ready:False preempt_next:False preempt_ready:False
in_mppdus:1 in_errored_mppdus :0 in_user_frames:1 in_errored_user_frames:0 in_user_octets:1200
in_pad_octets:0 in_user_fragments :0 in_user_dropped_fragments:0 in_user_errored_fragments:0
Event = new
State:NEW
express_next:False express_ready:False preempt_next:False preempt_ready:False
in_mppdus:1 in_errored_mppdus :0 in_user_frames:1 in_errored_user_frames:0 in_user_octets:1200
in_pad_octets:0 in_user_fragments :0 in_user_dropped_fragments:0 in_user_errored_fragments:0
Event = processFrame
State:DELIVER
express_next:False express_ready:False preempt_next:False preempt_ready:False
in_mppdus:1 in_errored_mppdus :0 in_user_frames:1 in_errored_user_frames:0 in_user_octets:1200
in_pad_octets:300 in_user_fragments :0 in_user_dropped_fragments:0 in_user_errored_fragments:0
Event = new
State:NEW
express_next:False express_ready:False preempt_next:False preempt_ready:False
in_mppdus:1 in_errored_mppdus :0 in_user_frames:1 in_errored_user_frames:0 in_user_octets:1200
in_pad_octets:300 in_user_fragments :0 in_user_dropped_fragments:0 in_user_errored_fragments:0
Event = processFrame
Sending {'length': 1200, 'serial_num': 1, 'express': None}
State:DELIVER
express_next:False express_ready:False preempt_next:True preempt_ready:False
in_mppdus:2 in_errored_mppdus :0 in_user_frames:2 in_errored_user_frames:0 in_user_octets:2400
```

```
in_pad_octets:300 in_user_fragments :0 in_user_dropped_fragments:0 in_user_errored_fragments:0
Event = new
State:NEW
express_next:False express_ready:False preempt_next:True preempt_ready:False
in_mppdus:2 in_errored_mppdus :0 in_user_frames:2 in_errored_user_frames:0 in_user_octets:2400
in_pad_octets:300 in_user_fragments :0 in_user_dropped_fragments:0 in_user_errored_fragments:0
Event = preemptReassembly
List exp False [{'length': 300, 'serial_num': 2, 'express': False, 'seq': 0, 'initial': True, 'final': False, 'pad':
None, 'mppdu_ix': 2}]
State:PRE_REASSEMBLY
express_next:False express_ready:False preempt_next:True preempt_ready:False
in_mppdus:2 in_errored_mppdus :0 in_user_frames:2 in_errored_user_frames:0 in_user_octets:2400
in_pad_octets:300 in_user_fragments :0 in_user_dropped_fragments:0 in_user_errored_fragments:0
Event = new
State:NEW
express_next:False express_ready:False preempt_next:True preempt_ready:False
in_mppdus:2 in_errored_mppdus :0 in_user_frames:2 in_errored_user_frames:0 in_user_octets:2400
in_pad_octets:300 in_user_fragments :0 in_user_dropped_fragments:0 in_user_errored_fragments:0
Event = processFrame
Sending {'length': 1400, 'serial_num': 3, 'express': None}
State:DELIVER
express_next:True express_ready:False preempt_next:True preempt_ready:False
in_mppdus:3 in_errored_mppdus :0 in_user_frames:3 in_errored_user_frames:0 in_user_octets:3800
in_pad_octets:300 in_user_fragments :0 in_user_dropped_fragments:0 in_user_errored_fragments:0
Event = new
State:NEW
express_next:True express_ready:False preempt_next:True preempt_ready:False
in_mppdus:3 in_errored_mppdus :0 in_user_frames:3 in_errored_user_frames:0 in_user_octets:3800
in_pad_octets:300 in_user_fragments :0 in_user_dropped_fragments:0 in_user_errored_fragments:0
Event = expressReassembly
List exp True [{'length': 100, 'serial_num': 4, 'express': True, 'seq': 0, 'initial': True, 'final': False, 'pad':
None, 'mppdu_ix': 3}]
State:EXP_REASSEMBLY
express_next:True express_ready:False preempt_next:True preempt_ready:False
in_mppdus:3 in_errored_mppdus :0 in_user_frames:3 in_errored_user_frames:0 in_user_octets:3800
in_pad_octets:300 in_user_fragments :0 in_user_dropped_fragments:0 in_user_errored_fragments:0
Event = new
State:NEW
express_next:True express_ready:False preempt_next:True preempt_ready:False
in_mppdus:3 in_errored_mppdus :0 in_user_frames:3 in_errored_user_frames:0 in_user_octets:3800
in_pad_octets:300 in_user_fragments :0 in_user_dropped_fragments:0 in_user_errored_fragments:0
Event = expressReassembly
```

```
List exp True [{'length': 100, 'serial_num': 4, 'express': True, 'seq': 0, 'initial': True, 'final': False, 'pad':
None, 'mppdu_ix': 3}, {'length': 1100, 'serial_num': 4, 'express': True, 'seq': 1, 'initial': False, 'final': True,
'pad': None, 'mppdu_ix': 4}]
State:EXP_REASSEMBLY
express_next:True express_ready:True preempt_next:True preempt_ready:False
in_mppdus:3 in_errored_mppdus :0 in_user_frames:3 in_errored_user_frames:0 in_user_octets:3800
in_pad_octets:300 in_user_fragments :0 in_user_dropped_fragments:0 in_user_errored_fragments:0
Event = sendExpress
Sending Combined = {'length': 1200, 'serial_num': 4, 'express': True}
State:DELIVER
Etc..
```