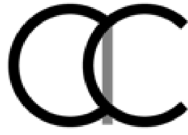


YAPP Communications for Taurus Platform

November 2023

600-0055-000 Rev A

Version History

Rev	Date	Notes
A	2023-11-01	Initial Release

Table of Contents

Version History 1
Table of Contents 2
Introduction 3
Interfaces 3
- Electrical Interface 3
- CAN 3
- Serial 3

YAPP 4
- UART (RS-232, RS-422), and Ethernet Framing 4
- CAN Framing 5
  - CAN Message IDs 5
  - Single Frame CAN YAPP 5
  - Multi-Frame CAN YAPP 6
    - Computing the UART and Ethernet Frame IDs from the CAN ID 6
    - Multi-Frame Start 7
    - Multi-Frame Continued 8
    - Multi-Frame End 8
- CRC Calculation 9
- YAPP Data Types 10
  - YAPP Float Compression 10
    - Float 16 Example 10
  - Time 10

Taurus Messages 11
- Taurus Command 11
- Taurus Motor Data 11
- Taurus Health 14

Examples 15
- UART Stream Framing 15
- Single CAN Frame: Taurus Command 16
- Multiple CAN Frames: Taurus Motor Data Msg 18

Introduction

The purpose of this document is to assist allocortech clients in implementing the **Y**et **A**nother **P**acket **P**rotocol (YAPP) over RS-422/485, RS-232 or packed into CAN messages (CAN YAPP) in order to communicate with allocortech's Taurus Electronic Speed Controller (ESC) Platform.

Interfaces

Electrical Interface

Taurus devices all support CAN 2.0 interfaces which are electrically isolated from the DC Link supply. Some Taurus platforms also support UART serial interfaces (e.g. RS232, RS485, RS422) which are also typically isolated, but users should consult the device specification for the specific device for details.

CAN

The isolated CAN interface on all of the Taurus devices supports CAN 2.0B. This includes standard 11-bit identifiers as well as extended 29-bit identifiers as defined in the CAN 2.0 specification.

The YAPP protocol defined herein uses 29-bit identifiers to encapsulate the YAPP message identifiers.

Serial

The UART serial interfaces, regardless of physical signal levels (i.e. RS232, etc.) are typically configured at 500kbps, 8bit, no parity, and 1 stop bit (8N1.) This may be customized through user settings or custom firmware as needed. By default, the Taurus may be configured to reserve the serial interface for a user shell, but can be configured to emit a YAPP frame serial stream.

YAPP

UART (RS-232, RS-422), and Ethernet Framing

YAPP messages sent via UART or Ethernet are encapsulated in a frame with a 12 byte header and 4 byte footer as shown below.

YAPP Framing

Header						Payload	CRC
SYNC[2]	Seq[1]	CTL[1]	ID[4]	Size[2]	RSVD[2]	*Size* Bytes	32-bit CRC

Messages sent over CAN are similar, but are chunked and aspects of the ID and sequence numbers are incorporated into the CAN ID. For more information about how CAN YAPP works, continue to the YAPP over CAN section.

`SYNC`:	Synchronization header of "`YP`"
`Seq:`	Sequence number, can optionally be used to increment for every newly created message under the given ID. By default this is not used on the Taurus.
`YAPP CTL:`	Control byte, always 0 for Taurus. The planned use for this field is to differentiate important commands, responses, routine telemetry, and low priority maintenance traffic.
`ID:`	Little Endian YAPP message identifier
`Size:`	Little Endian size of Payload blob in bytes
`RSVD:`	Reserved bytes for future use
`CRC:`	Little Endian cyclic redundancy check value protecting the YAPP header and message payload computed from a Koopman Hamming distance 6 order 32 (aka `CRC-32K/6.4`) polynomial of `0x1'32c0'0699`, and is computed with a starting value of `0xFFFF'FFFF` and non inverting output.

CAN Framing

As the CAN protocol provides framing, YAPP over CAN is mostly limited to providing a chunking mechanism for payloads greater than 8 bytes. Messages shorter than or equal to 8 bytes are placed into a single CAN message (or less than 64 bytes if CAN-FD is used.) CRC protection is added to chunked frames as part of the chunk header as described below.

Note that Taurus currently does not support CAN-FD, but for completely

CAN Message IDs

The 29 bit CAN ID contains an ID, control bits, and a sequence number counter and is designed to facilitate standard CAN prioritization and bus arbitration.

CAN YAPP 29 Bit Header

CAN IDs are shifted onto the bus most significant bit first.

	Msg ID											CAN CTRL			
Bit No.	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14

	YAPP CTRL						Seq No							
Bit No.	13	12	11	10	9	8	7	6	5	4	3	2	1	0

`Msg ID:` 11 bits for 2047 IDs (1-2047.) In Allocortech's code this value is always little endian, but will be presented on the CAN bus big endian therefore preserving CAN ID priority where lower IDs take precedence over higher IDs.

`CAN CTRL:`
- `0` - Single Frame Message
- `1` - Start of Multi-Frame Message
- `2` - Middle of Multi-Frame Message
- `3` - End of Multi-Frame Message

`YAPP CTRL:` Control byte, always 0 for Taurus

`Seq No:` Sequence number, can optionally be used to increment for every newly created message under the given ID. By default this is not used on the Taurus.

Single Frame CAN YAPP

YAPP message payloads less than 8 bytes long (CAN 2.0, 64 bytes for CAN FD) will be packaged into a single CAN message with no overhead. The CAN CRC is deemed sufficient for validation of the entire payload, and thus the YAPP CRC is not sent.

Multi-Frame CAN YAPP

Message payloads more than 8 bytes long (CAN 2.0) will be packaged into multiple CAN messages with the first message identifying the total payload length and the serial framed CRC. Additionally the CAN ID will identify the start of message, continuation of message, and end of message conditions.

Below we have a single instance of a Multi-Frame YAPP msg over CAN showing the start of frame, 3 continuation packets, and the end of frame.

Candump of Taurus Motor Data Msg

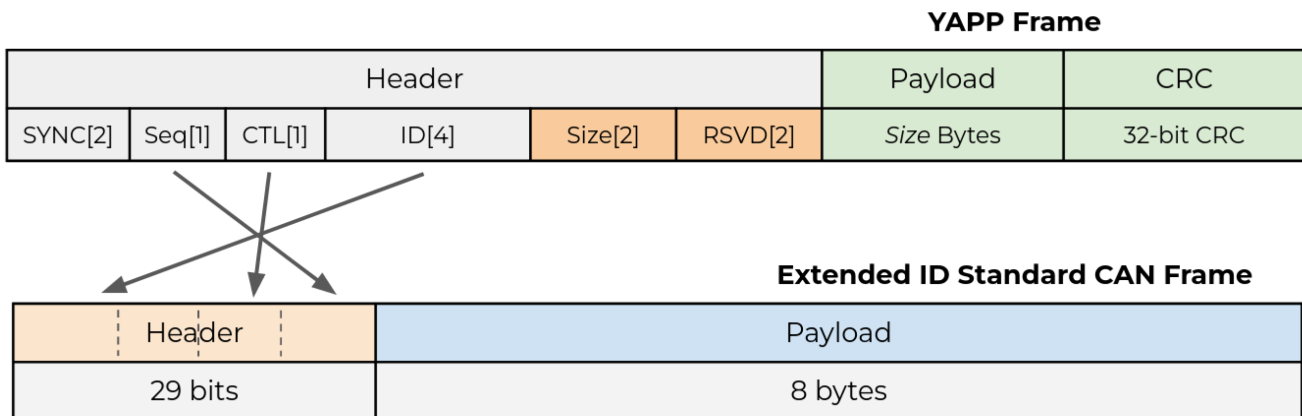
```
can0  08404000   [8]  BB BE 6F C7 20 00 00 00
can0  08408000   [8]  FD 7F FD 7F FD 7F FD 7F
can0  08408000   [8]  F5 AF FD 7F 18 01 00 00
can0  08408000   [8]  01 40 00 00 00 00 E3 8C
can0  0840C000   [8]  D2 3C E0 4E 00 00 05 FF
```

Computing the UART and Ethernet Frame IDs from the CAN ID

The CAN multiframe CRC is computed including the UART frame header in order to allow a received CAN YAPP to be directly forwarded via another protocol without recomputing the CRC. This means that the CAN YAPP ID must be able to be translated to an equivalent RS-232 frame ID.

Briefly, the two sync bytes are added, the sequence and YAPP control fields are rearranged, and the 11 bit CAN ID is prepended with zeros to form the UART YAPP ID. This rearrangement allows IDs to be used to assign relative message priorities in CAN arbitration.

YAPP Header Stuffing into 29 Bit CAN Header



Multi-Frame Start

```
can0  08404000   [8]  BB BE 6F C7 20 00 00 00
can0  08408000   [8]  FD 7F FD 7F FD 7F FD 7F
can0  08408000   [8]  F5 AF FD 7F 18 01 00 00
can0  08408000   [8]  01 40 00 00 00 00 E3 8C
can0  0840C000   [8]  D2 3C E0 4E 00 00 05 FF
```

The Multi-Frame Start message must set its CAN CTRL field to "1" with a payload consisting of the payload CRC and payload size.

Multi-Frame Start Example 29 Bit Header

	Msg ID (1-2047)	CAN Ctrl	YAPP Ctrl	Seq No
Bits	28:18	17:14	13:8	7:0
Value	528 (0x210)	0x1	0	0

Multi-Frame Start Example Payload

	CAN YAPP Multi Frame Metadata							
Byte No.	0	1	2	3	4	5	6	7
	YAPP CRC				Payload Size		Reserved	
	0xBB	0xBE	0x6F	0xC7	0x20	0x00	0x00	0x00

`YAPP CRC:` `0xC76FBEBB` (CRC of the following YAPP Frame Data: `0x595000001002000020000000FD7FFD7FFD7FFD7FF5AFFD7F1801000001400000000 00E38CD23CE04E000005FF`)

`Payload Size:` 32 (0x0020)

`Reserved:` 0

In more detail, the protocol constructs a RS-232 YAPP frame header to compute a CRC against. The header consists of

- `YP (0x5950)`
- 2 byte little endian sequence number, zero extended (`0x0000` if the sequence number had been 1 in this example, then the CRC'd value would have been `0x0100`)
- 4 byte little endian ID `(0x10020000)`
- 2 byte little endian packet length (`0x2000`)
- 2 byte reserved data

Multi-Frame Continued

```
can0  08404000  [8]  BB BE 6F C7 20 00 00 00
can0  08408000  [8]  FD 7F FD 7F FD 7F FD 7F
can0  08408000  [8]  F5 AF FD 7F 18 01 00 00
can0  08408000  [8]  01 40 00 00 00 00 E3 8C
can0  0840C000  [8]  D2 3C E0 4E 00 00 05 FF
```

The Multi-Frame Continued message must set its CAN CTRL field to “2”. A series of Multi-Frame continued messages are expected until the last 8 bytes or less of the YAPP payload are sent in the Multi-Frame End Message.

Multi-Frame Continued Example 29 Bit Header

	Msg ID (1-2047)	CAN Ctrl	YAPP Ctrl	Seq No
Bits	28:18	17:14	13:8	7:0
Value	528 (0x210)	0x2	0	0

Multi-Frame End

The last data packet, consisting of the last 8 or fewer bytes (CAN 2.0, 64 or fewer for CAN FD), must set its CAN CTRL field to “3”. For CAN-FD, if the data is less than the minimum packet size then the data will be post-padded with zeros.

```
can0  08404000  [8]  BB BE 6F C7 20 00 00 00
can0  08408000  [8]  FD 7F FD 7F FD 7F FD 7F
can0  08408000  [8]  F5 AF FD 7F 18 01 00 00
can0  08408000  [8]  01 40 00 00 00 00 E3 8C
can0  0840C000  [8]  D2 3C E0 4E 00 00 05 FF
```

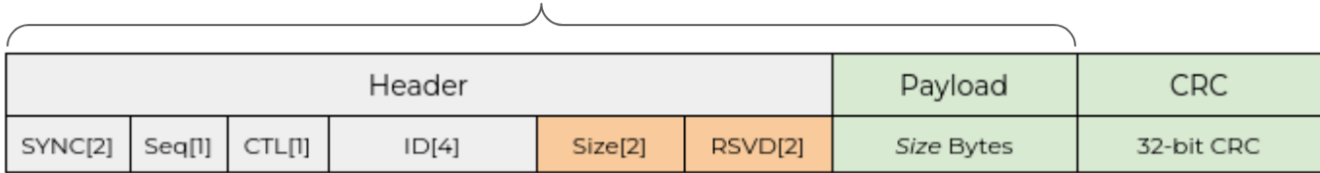
Multi-Frame End Example 29 Bit Header

	Msg ID (1-2047)	CAN Ctrl	YAPP Ctrl	Seq No
Bits	28:18	17:14	13:8	7:0
Value	528 (0x210)	0x3	0	0

CRC Calculation

The YAPP header and message payload is protected with the [Koopman Hamming distance 6 order 32 (aka `CRC-32K/6.4`)]() cyclic redundancy check. This CRC has a polynomial of `0x1'32c0'0699`, is computed with a starting value of `0xFFFF'FFFF,` and does not invert the output.

Header and Payload CRC'd



To verify your CRC calculations you can use an [online crc tool](), with the parameters shown in the example below. In this example a

YAPP frame data:

`5950000010020000200000000FD7FFD7FFD7FFD7FF5AFFD7F18010000014000`
`000000E38CD23CE04E000005FF`

Resulting CRC: `0xC76FBEBB`

Example CRC Calculation

CRC width
Bit length: ○ CRC-8 ○ CRC-16 ◉ CRC-32 ○ CRC-64

CRC parametrization
○ Predefined CRC32 ◉ Custom

CRC detailed parameters
Input reflected: ☐ Result reflected: ☐
Polynomial: 0x32C00699
Initial Value: 0xFFFFFFFF
Final Xor Value: 0x00000000

CRC Input Data
○ String ◉ Bytes ○ Binary string
59500000100200002000000000FD7FFD7FFD7FFD7FF5AFFD7F1801000001400
0000000E38CD23CE04E000005FF

Show reflected lookup table: ☐ (This option does not affect the CRC calculation, or

Calculate CRC!

Result CRC value: **0xC76FBEBB**

YAPP Data Types

All types in YAPP are defined to be little endian. Floating point numbers, if transmitted raw on the wire, are IEEE 754. Signed numbers are represented in 2's complement format.

YAPP Float Compression

Some amount of protocol compression is allowed for floating point types, and is identified by `<min, max>` after the type name and width. When a data type is compressed, there are 5 reserved values:

`Not a Number`	`max`	Regardless of signaling or non signaling NaN
`Positive Infinity`	`max - 1`	Special case of out of range high
`Negative Infinity`	`max - 2`	Special case of out of range low
`Out of Range High`	`max - 3`	Initial value was greater than `max` (but not infinity)
`Out of Range Low`	`max - 4`	Initial value was less than `min` (but not infinity)

Float 16 Example

`float16 <-10.0; 5.0>` would map the following values:

```
-10.0    -> 0 counts
0.0      -> 43,687 counts
5.0      -> 65,530 counts
NaN      -> 65,535 counts
```

Time

In the allocore framework, time is encoded as signed 64 bit integer nanoseconds. For embedded platforms like Taurus time will be counted starting from zero at power up. Other platforms, like the Taurus GUI, may use some other timebase such as monotonic time since boot, time smeared epoch, or GPS time of week.

Inside the allocortech SDK, this time representation is named `RawTime`.

Taurus Messages

Taurus Command

C++ Struct Name: `TaurusCommandMsg`
YAPP ID: `0x00`
Expected Rate: Greater than 5 Hz
Payload Length: 7 bytes

Byte	Name	Type	Description
`0`	Enabled	`uint8`	`0` – Disabled `1` – Enabled
`1`	Key	`uint8`	Magic number for extra protection against spurious commands. `0xA5` – Full Operation `0x5A` – No Regeneration Allowed `Other` – Invalid, the command message will be ignored
`2`	Motor Mode	`uint8`	`0` – Torque Mode `1` – Speed Mode
`3:4`	Torque IQ	`float16` <-200; 200>	Commanded Current, A
`5:6`	RPM	`float16` <-100k; 100k>	Commanded Speed, RPM

Taurus Motor Data

C++ Struct Name: `TaurusMotorDataMsg`
YAPP ID: `0x210`
Expected Rate: 10 Hz
Payload Length: 32 bytes

Byte	Name	Type	Description
`0:1`	Torque IQ Commanded	`float16` <-128; 128>	Commanded Phase Torque, A
`2:3`	Torque IQ Measured	`float16` <-128; 128>	Measured Phase Torque, A
`4:5`	RPM Commanded	`float16` <-60,000; 60,000>	Commanded speed, RPM

Byte	Name	Type	Description
6:7	RPM Measured	float16 <-60,000; 60,000>	Measured speed, RPM
8:9	DC Voltage	float16 <-128; 128>	Measured input voltage, V
10:11	DC Current	float16 <-128; 128>	Estimated input current, A
12	Motor Temperature	float8 <-40; 210>	Degrees Centigrade
13	Motor Mode	uint8	0 – Torque Mode 1 – Speed Mode
14:17	Status Flags	uint32	0x 2 – FOC in align 0x 1000 – Start in Motion, Back EMF signal not detected 0x 2000 – Start in Motion, Back EMF signal detected 0x 4000 – Waiting for motor to stop 0x 8000 – Motor is braking 0x 1 0000 – Motor is stopped 0x 2 0000 – Motor is enabled 0x 4 0000 – FOC in open loop mode 0x 8 0000 – FOC in open loop speed ramp up 0x 10 0000 – FOC in closed loop speed ramp down 0x 20 0000 – FOC in closed loop mode 0x 40 0000 – Motor in reverse direction mode 0x 80 0000 – Motor in Field Weakening range 0x 100 0000 – Motor operating in Over-Modulation region 0x 200 0000 – Motor Position taken from Hall Sensor 0x 400 0000 – Motor Position is estimated 0x4000 0000 – Flag MPOS: Update motor position values
18:21	Fault Flags	uint32	0x 1 – OC limit higher than measurable current 0x 2 – CBC OC LPDAC 0x 4 – PWM duty cycle is saturated 0x 8 – Stack overflow or underflow 0x 10 – Open loop speed is lower than closed-loop minimum speed 0x 20 – Estimator AngleDiff out of range 0x 40 – FOC control time has been exceeded 0x 80 – Bus Voltage Min/Max limit exceeded 0x 100 – PI controller gain out of range 0x 200 – Stall filter value floored at minimum 0x 400 – Estimator SpeedDiff out of range 0x 800 – ATPI Warning -> Check ATPI Status

Byte	Name	Type	Description
			0x 1000 – Control Freq to Estimator > 10:1 0x 2000 – Hall Timer Period Invalid 0x 4000 – Motor Coast timeout 0x 8000 – EstSpeed/AngleDiff timeout 0x 1 0000 – Motor overcurrent 0x 2 0000 – Motor disabled due to bus voltage min/max fault 0x 4 0000 – Motor Min/Max Speed 0x 8 0000 – Motor Open Phase Detection 0x 10 0000 – Flash CRC Test Status Failed 0x 20 0000 – Attempted change to critical parameter while running 0x 40 0000 – AFE not initialized properly 0x 80 0000 – Motor Stall Detection 0x 100 0000 – PPM pulse timeout, no received valid PPM pulse 0x 200 0000 – ADC calibration failed 0x 400 0000 – Hall Angle Sequence or State is Invalid 0x 800 0000 – Estimator inputs are invalid 0x1000 0000 – Hall timer expired without detecting hall transition 0x2000 0000 – Start in motion, wrong direction detected 0x4000 0000 – Hardware exceeded temperature threshold 0x8000 0000 – Reserved
22:29	Timestamp	int64	Nanoseconds since boot
30	Motor State	uint8 bitfield	0x 1 – Ready 0x 2 – Running 0x 4 – Stopped 0x 8 – Overmodulated 0x10 – Saturated 0x20 – Faulted
31	ESC Temperature	float8 <-40; 210>	Temperature of Taurus printed circuit board

Taurus Health

C++ Struct Name: `TaurusHealthMsg`
YAPP ID: `0x200`
Expected Rate: 2 Hz
Payload Length: 17 bytes

Byte	Name	Type	Description
`0:7`	Timestamp	`int64`	Nanoseconds since boot
`8`	Control Thread CPU Usage	`float8 <0, 100>`	Percentage averaged over 1 second
`9`	Taurus Thread CPU Usage	`float8 <0, 100>`	Percentage averaged over 1 second
`10`	CPU Temperature	`float8 <-40; 210>`	Degrees Centigrade
`11`	Capacitor Temperature	`float8 <-40; 210>`	Degrees Centigrade
`12`	FET Temperature	`float8 <-40; 210>`	Degrees Centigrade
`13`	Vin RMS Ripple	`float8 <0; 12.5>`	Volts
`14`	Vin Peak to Peak Ripple	`float8 <0; 12.5>`	Volts
`15`	Taurus Status	`uint8 bitfield`	`0x 1` – Regeneration Enabled `0x 2` – Reversed `0x 4` – Precharging
`16`	Board Revision	`uint8`	Used by Taurus firmware to distinguish hardware revisions and appropriately configure I/O

Examples

UART Stream Framing

This is an example Taurus Motor Data message.

Header											
SYNC		Seq	CTL	ID				Size		RSVD	
0x59	0x50	0x00	0x00	0x10	0x02	0x00	0x00	0x02	0x00	0x00	0x00

Payload											
Current Command		Current Measured		RPM Command		RPM Measured		DC Voltage		DC Current	
0xFD	0x7F	0xFD	0x7F	0xFD	0x7F	0xFD	0x7F	0xF5	0xA5	0xFD	0x7F

Payload (cont)									
Motor Temp	Motor Mode	Status Flags				Fault Flags			
0x18	0x01	0x00	0x00	0x01	0x40	0x00	0x00	0x00	0x00

Payload (cont)										CRC			
Timestamp								Motor State	ESC Temp				
0xE3	0x8C	0xD2	0x3C	0xE0	0x4E	0x00	0x00	0x05	0xFF	0xBB	0xBE	0x6F	0xC7

Single CAN Frame: Taurus Command

Candump of Taurus Command Msg

```
can0  00000000   [7]  01 5A 01 4F 80 F3 80
```

Single Frame Example 29 Bit Header

	Msg ID - 11 bits for 2047 IDs (1-2047)											CAN CTRL			
Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bit No.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

	YAPP CTRL						Seq No							
Value	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Bit No.	15	16	17	18	19	20	21	22	23	24	25	26	27	28

`Msg ID:` 0 (Default Msg ID For Taurus Command)
`CAN CTRL:` 0 (Single Frame)
`YAPP CTRL:` 0
`Seq No:` 0
`Combined header:` 0x00000000

Single Frame Example Payload (7 Bytes)

	CAN YAPP Single Frame Payload Data						
	Enable	Key	Motor Mode	Torque Command		RPM Command	
	0x01	0x5A	0x01	4F	80	F3	80
Byte No.	0	1	2	3	4	5	6

`Enable:` 0x01 (boolean, 1=enabled)
`Key:` 0x5A (magic number for extra protection against spurious commands, overloaded to encode for regen disabling, 0xA5 = Full Operation, 0x5A = No Regen)
`Motor Mode:` 0x01 (0 = Torque Mode, 1 = Speed Mode)

`Torque Cmd (Cnts):` 0x804F (32847 counts)
`Float16<-200, 200>` (200 - (-200)) / (2^16 - 5) = 400 A / 65531 counts = 0.0061 A / count
`Torque Cmd (A):` 32847 counts * (0.0061 A / count) + (-200A) = 0.4975 IQ Current Amps *
* When in speed mode the torque command is ignored

`RPM Cmd (Cnts):`	0x80F3 (33011 counts)
`Float16<-100k,100k>:`	(100,000 - (-100,000)) / (2^16 - 5) = 200,000 RPM / 65531 counts = 3.052 RPM / count
`RPM Cmd (RPM):`	33011 counts * (3.052 RPM / count) + (-100,000 RPM) = 749.26 RPM

Multiple CAN Frames: Taurus Motor Data Msg

Candump of Taurus Motor Data Msg

```
can0  08404000   [8]  BB BE 6F C7 20 00 00 00
can0  08408000   [8]  FD 7F FD 7F FD 7F FD 7F
can0  08408000   [8]  F5 AF FD 7F 18 01 00 00
can0  08408000   [8]  01 40 00 00 00 00 E3 8C
can0  0840C000   [8]  D2 3C E0 4E 00 00 05 FF
```

Multi-Frame Start Example Payload

	CAN YAPP Multi Frame Metadata							
Byte No.	0	1	2	3	4	5	6	7
	YAPP CRC				Payload Size		Reserved	
	0xBB	0xBE	0x6F	0xC7	0x20	0x00	0x00	0x00

`YAPP CRC:` `0xC76FBEBB` (CRC of the following YAPP Frame Data: `0x595000001002000020000000FD7FFD7FFD7FFD7FF5AFFD7F1801000001400000000 00E38CD23CE04E000005FF`)

`Payload Size:` 32 (0x0020)

`Reserved:` 0

Multi-Frame Continued Example Payload

	CAN YAPP Multi Frame Payload Data							
	DC Voltage		DC Current Amps		Motor Temp	Motor Mode	Status Flags (partial)	
	0xF5	0xAF	0xFD	0x7F	0x18	0x01	0x00	0x00
Byte No.	0	1	2	3	4	5	6	7

`DC Voltage (Cnts):` 0xAFF5 (45045 counts)

`Float16<-128, 128>` (128 - (-128)) / (2^16 - 5) = 256 V / 65531 counts = 0.0039065 V / count

`DC Voltage (V):` 45045 counts * (0.0039065 V / count) + (-128V) = 47.97 V

`DC Current (Cnts):` 0x7FFD (32765 counts)

`Float16<-128, 128>` (128 - (-128)) / (2^16 - 5) = 256 A / 65531 counts = 0.0039065 A / count

`DC Current (A):` 32765 counts * (0.0039065 A / count) + (-128A) = 47.97 V

`Motor Temp (Cnts):` 0x18 (24 counts)

`Float8<-40, 210>` (210 -(-40)) / 2^8 - 5) = 250 °C / 250 counts = 1 °C / count

`Motor Temp (°C):` 24 counts * (1 °C / count) + (-40 °C) = -16 °C *

* Note the motor thermistor was not connected so the temperature from this example wasn't valid

`Motor Mode (Cnts):` 0x01 (1 count)

`Enum` 0x00 = Torque Mode
0x01 = Speed Mode

`Status Flags:` **0x00 0x00** 0x01 0x40 (only first two bytes are included in this frame)

`Status Flags(32):` 0x4001**0000**

`Status Flags: Motor Stopped Flag, Update MPOS Flag`

Multi-Frame End Example Payload

	CAN YAPP Multi Frame Metadata							
	Timestamp (partial)						Motor State	ESC Temp
	0xD2	0x3C	0xE0	0x4E	0x00	0x00	0x05	0xFF
Byte No.	0	1	2	3	4	5	6	7

`Timestamp:` 0xE3 0x8C **0xD2 0x3C 0xE0 0x4E 0x00 0x00** (only last six bytes are included in this frame)

`Timestamp(64 signed):` 0x**00004EE03CD2**8CE3

`RawTime (nsec):` 86725000072419 nanoseconds (24.09 hrs)

`Motor State:` 0x05 (0b 0000 0101)

`Motor State(Flags)` Ready, Stopped

`ESC Temp:` 0xFF

`Float8<-40, 210>` *Nan (one of the 5 reserved values at the top of the range)
*ESC Thermistor was not in place on this unit