

Decoding Automotive TPS CAN Data

<https://gemini.google.com/app/db1596803ef74219>

User prompt: Hi gemini. i have a automotive CAN table i need help under standing. my target is the TPS
Byte Channel Data Type Range Multiplier/ Divider Factor Offset Unit EMU stream base ID+0 (default: 0x600)
0..1RPM16-bit U0 – 160001/110RPM 2TPS8-bit U0 – 1001/20.50% 3IAT8-bit S-40 – 1271/110C 4..5MAP16-bit
U0 – 6001/110kPa 6..7Injector PW16-bit U0 – 501/620.0161290ms EMU stream base ID+1 (default: 0x601)
0..1AIN #116-bit U0–55/10240.00488281250V 2..3AIN #216-bit U0–55/10240.00488281250V 4..5AIN
#316-bit U0–55/10240.00488281250V 6..7AIN #416-bit U0–55/10240.00488281250V

Response: To read the **Throttle Position Sensor (TPS)** value from this CAN frame, focus on **Byte 2** of the **Base ID** message (default 0x600).

Parameter	Value / Setting
CAN Message ID	Base ID + 0 (Default: 0x600 / 1536 decimal)
Byte Location	Byte 2 (0-indexed: 3rd byte in the 8-byte payload)
Data Type	8-bit Unsigned Integer (uint8_t)
Raw Value Range	0 to 200 (maps to 0% – 100%)
Scaling Multiplier	0.5 (or divide raw value by 2)
Offset	0
Unit	%

Formula & Conversion

TPS (%)=Raw Value×0.5

- **Raw CAN Byte = 0x00 (0):** 0×0.5=0% (Fully Closed)
- **Raw CAN Byte = 0x64 (100):** 100×0.5=50% (Half Throttle)
- **Raw CAN Byte = 0xC8 (200):** 200×0.5=100% (Wide Open Throttle)

C/C++ Decoding Example

```
// Assuming frame.data is an array of 8 bytes (uint8_t frame[8])
uint8_t raw_tps = frame[2];
float tps_percent = raw_tps * 0.5f;
```

User prompt: so ill need to send that over CAN to the ECM on this specific application. how would the whole CAN message look if the throttle was 22%

Response: To send a TPS value of **22%** to the ECM, you need to calculate the raw byte value and assemble the entire 8-byte CAN frame for message **0x600**.

Step 1: Calculate the TPS Byte Value

Using the formula Raw Value=TPS (%)÷0.5:

Raw Value=22÷0.5=44 (Decimal)=0x2C (Hex)

Step 2: Complete CAN Frame Layout (0x600)

Assuming all other engine parameters are zeroed out:

Byte	Channel	Value (Decimal)	Multiplier / Scale	Raw Hex Byte	Notes
0	RPM (Low Byte)	0	1	0x00	16-bit uint (Little Endian)
1	RPM (High Byte)	0	1	0x00	
2	TPS	22%	0.5	0x2C	44 in decimal
3	IAT	0°C	1 (Offset -40)	0x28	0°C+40=40 (0x28)
4	MAP (Low Byte)	0 kPa	1	0x00	16-bit uint

Byte	Channel	Value (Decimal)	Multiplier / Scale	Raw Hex Byte	Notes
5	MAP (High Byte)	0 kPa	1	0x00	
6	Injector PW (Low Byte)	0 ms	0.016129	0x00	16-bit uint
7	Injector PW (High Byte)	0 ms	0.016129	0x00	

Step 3: Complete Raw Payload

- **CAN ID (Standard 11-bit):** 0x600
- **Data Length (DLC):** 8 Bytes
- **Payload (Hex):**

00 00 2C 28 00 00 00 00

C/C++ Construct Code

```
uint8_t can_payload[8] = {0};

float tps_percent = 22.0f;
uint8_t raw_tps = (uint8_t)(tps_percent / 0.5f); // 44 (0x2C)

can_payload[0] = 0x00; // RPM Low
can_payload[1] = 0x00; // RPM High
can_payload[2] = raw_tps; // TPS
can_payload[3] = 40; // IAT (0 deg C encoded as 40 due to -40 offset)
can_payload[4] = 0x00; // MAP Low
can_payload[5] = 0x00; // MAP High
can_payload[6] = 0x00; // Inj PW Low
can_payload[7] = 0x00; // Inj PW High
```

User prompt: why is the payload that again?

Response: Each byte position in the 8-byte payload maps to specific engine data defined by the Ecumaster EMU stream format.

Here is why every byte in 00 00 2C 28 00 00 00 00 is set the way it is:

Payload Breakdown Byte-by-Byte

- **Bytes 0 & 1 (00 00) — RPM (16-bit Unsigned)**
 - Formula: Raw=RPM×1

- If engine RPM is 0, the 16-bit raw value is 0x0000. Split across two bytes, both are 0x00.
- **Byte 2 (2C) — TPS (8-bit Unsigned)**
 - Formula: $\text{Raw} = \text{TPS} (\%) \div 0.5$
 - For **22%** throttle: $22 \div 0.5 = 44$ in decimal, which converts to **0x2C** in hexadecimal.
- **Byte 3 (28) — IAT / Intake Air Temp (8-bit Signed, -40 Offset)**
 - Formula: $\text{Raw} = \text{Temp} (^{\circ}\text{C}) + 40$
 - Even if temperature is 0°C , the raw value cannot be 0x00 because the range starts at -40°C .
 - For **0°C** : $0 + 40 = 40$ in decimal, which converts to **0x28** in hexadecimal. (*If you sent 00 here, the ECM would read it as -40°C .*)
- **Bytes 4 & 5 (00 00) — MAP / Manifold Pressure (16-bit Unsigned)**
 - Formula: $\text{Raw} = \text{MAP} (\text{kPa}) \times 1$
 - If pressure is 0 kPa, the 16-bit raw value is 0x0000 (0x00, 0x00).
- **Bytes 6 & 7 (00 00) — Injector Pulse Width (16-bit Unsigned)**
 - Formula: $\text{Raw} = \text{PW} (\text{ms}) \div 0.016129$
 - If pulse width is 0 ms, the raw value is 0x0000 (0x00, 0x00).

The Complete Assembly

Putting all 8 bytes together in order:

Byte 0(RPM L)00Byte 1(RPM H)00Byte 2(TPS)2CByte 3(IAT)28Byte 4(MAP L)00Byte 5(MAP H)00Byte 6(PW L)00Byte 7(PW H)00

User prompt: so i only have the control over the TPS for this project. the other values are controlled independently. how can i inject the TPS value ONLY. via can

Response: To update **only** the TPS without overwriting or interfering with the other parameters on CAN frame 0x600, you must use a **Read-Modify-Write** approach on your CAN receiver node/gateway, or use CAN ID masking/filtering if you are acting as an inline proxy/bridge.

Because standard CAN bus frames are transmitted as full 8-byte payloads, **you cannot broadcast a "partial" frame containing only Byte 2**. Every transmission of 0x600 overwrites all 8 bytes at the receiving end.

Here are the two ways to handle this in hardware and software:

Option 1: Read-Modify-Write (If you share the bus)

If the main ECU/node is already broadcasting 0x600 onto the bus:

1. **Listen:** Intercept/read the incoming 0x600 CAN message.
2. **Modify:** Keep Bytes 0–1, 3–7 exactly as received, and replace **Byte 2** with your calculated TPS value (0x2C).
3. **Transmit:** Immediately output the modified 8-byte frame to the target ECM.

C/C++ Implementation Example

```
void injectTPS(CAN_message_t *frame, float tps_percent) {
    // Check if this is the target ECU frame (0x600)
    if (frame->id == 0x600) {
```

```
    // Calculate raw 8-bit value (22% -> 44 -> 0x2C)
    uint8_t raw_tps = (uint8_t)(tps_percent / 0.5f);

    // Inject TPS into Byte 2 while leaving all other 7 bytes untouched
    frame->data[2] = raw_tps;

    // Send the updated frame down the line
    can_send_frame(frame);
}
}
```

Option 2: CAN Bus Bridge / Man-in-the-Middle

If another device is continually broadcasting 0x600 with its own TPS data, simply writing your own 0x600 frame onto the same wire will cause **bus flooding and frame collisions** (the ECM will rapidly alternate between the original TPS and your injected TPS).

To isolate and replace it cleanly:

- **Cut/Separate the CAN Bus Line:** Place your microcontroller with dual CAN transceivers between the source device (CAN1) and the ECM (CAN2).
 - **Forwarding Logic:** Pass all traffic from CAN1 to CAN2 continuously. When 0x600 passes through, alter data[2] on the fly before transmitting it out to CAN2.
-