CAN – J1939 Protocol Stack
****

In this session, we'll be learing about the protocol stack that we need for our task i.e. "J1939 secure communication".

For this entire task, we need to choose some protocols for different layers of OSI/ISO model. Let's first choose some protocols for different layers of OSI model and then create a protocol stack.

For our communication, all the layers of OSI model are not necessary just like many exceptions, we just need to choose protocols for : L1 , L2, L4, L7 only, and why we dont need other protocols can be understood by the sense that their properties and workflow are of no use to our communication scheme.

L1	CAN (developed)
L2	CAN 2.0B (developed)
L4	J1939 Transport protocol (already developed, provided by ECOM libraries).
L7	J1939 Application protocol (already developed, provided by ECOM libraries).

It's time to explore above protocols briefly...

1.) ***CAN Physical layer protocol*** :

This protocol uses 3 hardware components generally which include a CAN cable, a CAN transceiver and a CAN controller.

Let's dicsuss these components in details...

a.) CAN cables :

These cables generally contains 3 wires which includes 'common ground', 'CAN_H' and 'CAN_L'.
CAN_H means CAN HIGH and CAN_L means CAN LOW.
CAN physical layer protocol uses differential signaling mechanism, thus the signal is taken as the potential difference between CAN_H and CAN_L.

CAN bus is a one way shared bus shared by multiple nodes in the CAN network. the bus generally remains in logical 1 state when no data is being transmitted.

To transmit logical 0 :

CAN_H is maintained at higher voltage than CAN_L, CAN_H is typically kept at ~3.5V and CAN_L is kept at ~1.5V, differential signaling is used in CAN Bus protocol, reasons for using this type of signaling have already been discussed in '01_Signaling.pdf'.
This state of CAN cable is called as 'dominant state' and a 'dominant bit' is being transferred.

To transmit logical 1 :

CAN_H and CAN_L are both kept at same voltages, typically around ~2.5V.
This state of CAN cable is called as 'recessive state' and a 'recessive bit' is being transferred.

The differential amplifier on transmitter and receiver side takes care of transmitting and receiving the bits as potential difference between CAN_H and CAN_L.

b.) CAN Transceiver :

This is an electronic device which is used in interconversion of the digital signals and CAN signals. This takes digital signals of CAN frames from CAN controller and converts those signals precisely into the CAN signals i.e. in differential signals and thus does vice-versa, taking differential signals from CAN bus and converting them into digital signals which can be read by CAN controller.

c.) CAN controller :

This is a 2 way device which again receives data from one side and send it after processing to other side.
This, when receives signals from micro-controller, microcontroller sends the payload to the CAN controller, packs that payload packet into the CAN frame based on some hardware configurations of itself which we'll discuss briefly in CAN 2.0B Data-link layer protocol.
The frame is given to the CAN transceiver which in turn transmits the frame onto the CAN bus.
And when it receives signals from CAN Transceiver, it sends out the relevant information from frame like payload or CAN ID to the firmware or user program entirely depending on some hardware filtering of itself which may be configured by micro-controller to the micro-controller .

The collision etc are all handled by the physical layer i.e. by the CAN controller.
The CAN bus is a wired-AND bus, meaning that when multiple nodes drive the bus simultaneously, the resulting bus state is the logical AND of all the individual node states.
In idle or disconnected state, the bus remains in recessive state i.e. logical 1 state.
In recessive state, the node is still actively participating in the bus communication, but it's not actively driving the bus to a dominant state.

So, this is actually what we needed to learn about CAN 2.0B Physical layer protocol for our task.

2.) ***CAN 2.0B Data-link layer protocol*** :

Just like normal communication protocols like TCP/IP etc which at data-link layer uses ethernet etc as its data-link protocol which creates a L2 frame by encapsulating IP packet into it with some other data-link interface information, we, in CAN uses this protocol, this protocol also does something similar, it takes the payload from firmware, the payload may be raw user data or a packet of some other high layer protocols.
It when receives a payload from firmware, encapsulates it into the CAN frame with some interface information like CAN ID etc, note that these interface informations can be altered by the mean of some upper level protocols.

This protocol have multiple variants, some of them are listed below but we'll studying only one out of them which our task demands.

1. **CAN 2.0A (Standard CAN)**: This is the original CAN protocol, also known as Standard CAN. It uses an 11-bit identifier and has a maximum data rate of 1 Mbps.

2. **CAN 2.0B (Extended CAN)**: This variant uses a 29-bit identifier, which allows for more nodes on the network. It also has a maximum data rate of 1 Mbps.

3. **CAN FD (Flexible Data Rate)**: This variant allows for faster data rates (up to 8 Mbps) and larger data payloads (up to 64 bytes). It uses a 29-bit identifier and is backward compatible with CAN 2.0B.

4. **CAN XL (Extra Large)**: This is a newer variant that allows for even larger data payloads (up to 2048 bytes) and faster data rates (up to 10 Mbps). It uses a 32-bit identifier and is not backward compatible with previous CAN variants.

5. **CANopen**: This is a higher-layer protocol that builds on top of the CAN 2.0B protocol. It adds additional features such as node addressing, data types, and communication protocols.

6. **DeviceNet**: This is a variant of CAN that is used in industrial automation applications. It uses a 29-bit identifier and has a maximum data rate of 500 kbps.

7. **CAN-FD-Light**: This is a variant of CAN FD that is optimized for low-power applications. It uses a 29-bit identifier and has a maximum data rate of 2 Mbps.

For our task, we need CAN 2.0B (extended CAN).

So, in our case, the work of data-link layer will be to take user/firmware data and encapsulate it into a CAN 2.0B frame and send it to transceiver, this protocol uses a physical interface called CAN controller, we've already discussed the working of CAN controller, now let's discuss what a CAN controller do, what frame it creates by encapsulating the firmware data.

CAN 2.0B frame structure :

Field	***Number of bits***	***Description***
Start Of Frame (SOF)	1	Dominant bit always.
Identifier	29	Unique identifier indicates priority.
Remote Transmission Request (RTR)	1	Dominant in data frames, recessive in remote frames.
Data Length Code (DLC)	4	Number of data bytes (0–8)
Data	0-8 bytes	Data length determined by DLC field.

Cyclic Redundancy Check (CRC)	15	Used for error checking.
CRC Delimiter	1	Must be recessive.
Acknowledge (ACK)	1	Transmitter sends recessive, receiver overrides with dominant.
Acknowledge delimiter	1	Must be recessive.
End Of Frame (EOF)	7	Must be recessive.

One can see that the entire CAN protocol heavily relies on the dominant bits, since in normal scenario when their is no data transmission, the state of the bus is recessive i.e. in logical 1 state, thus to indicate any changes and signaling that some node has started transmission, dominant bits must be sent first so that other nodes can identify the change in potential across the CAN wires and interpret it accordingly.
This is also called as bit stuffing.

Let’ s discuss all the fields of the CAN 2.0B frame and see that what interface information does it have, and which information can be altered by higher-layer protocols...

1.) **Start Of Frame (SOF)** : This bit is used to indicate that a frame is following the bit, this bit is '0' since CAN bus is in recessive state when there is no frame transmission, thus when all other receiving nodes captures this dominant bit, they get ready for the CAN frame which is following the SOF bit.

2.) **Identifier** : This is an identity which is unique for each CAN frame (not CAN node), it contains several sub-fields which actually represents the higher layer protocol information thus for now, its just a field which is used in bus arbitration, the information included in this field is set into the CAN controller’s hardware from where the controller reads it and creates CAN frames, the configuration is done by user program (ECOM library in our case.), this field can be said as the ground base of the enitre CSMA/CA mechanism and bus arbitration (bus arbitration is discussed in '02_collision_handling.pdf'), lower the value of a sub-field of this quantity, higher will be its priority.
This is the only field which differentiates 'CAN 2.0A' and 'CAN 2.0B', since 29 bits are assigned to this field and it represents identities of CAN frames uniquely.
Thus via the standard format of 'CAN 2.0B', the entire CAN network can have up to 2^29 CAN frame identities (including the programmer node).
When two nodes transmits simultaneously, both of them observes the change in potential on CAN wires when their non equal bit collides which happens at the initial phase before any actual data has been transferred, the node transmitting 0 will dominate and continue to transmit whereas the node transmitting 1 will lose the race and stop transmitting.

3.) **Remote Transmission Request (RTR)** :
This bit represents the type of the frame, whether the frame is a data

frame i.e. data sent by a node for some specific nodes or the frame is a remote frame i.e. a query/request frame asking for data from some specific node as per the firmware program.

This bit is 0 for data frames and 1 for remote frames.

4.) **Data Length Code (DLC)** :

Represents the number of data bytes in next field, can be useful sometimes to shorten the frame size.
DLC number of bytes are actually extracted by CAN controller on receiver's node and are sent to firmware on request.
4 bits are given to this field which can easily represent the 8 bytes of data.
When the payload is ready from the micro-controller’s point of view, it triggers some sort of signal to activate the CAN controller to read specific number of bytes from the given memory, thus DLC should be passed to handler function as argument or some how.
This field is criticial because it tells the CAN controller to read only required number of bytes from the memory.

5.) **Data field** : This field holds the actual data which the CAN controller receives from the firmware program running on the transmitter's side.
for 'CAN 2.0B' the maximum data which one frame can transfer is only 8 bytes, if larger data/payload is given to the CAN controller, i.e. if DLC value is set more than 8, then many scenarios are possible like :

Error or exception: The CAN controller's firmware or library might detect the invalid DLC value and raise an error or exception. This could be in the form of a return code, an interrupt, or a fault flag being set.
Truncation: The CAN controller might truncate the data to 8 bytes, ignoring any excess bytes beyond the 8-byte limit. This could result in data loss or corruption.
Wraparound: In some cases, the CAN controller might wrap around the DLC value to a valid range (0-8). For example, if the DLC is set to 9, the controller might interpret it as 1 (9 mod 8 = 1).
Undefined behavior: The CAN controller's behavior might be undefined or unpredictable when faced with an invalid DLC value. This could lead to unexpected results, such as transmitting incorrect data or causing bus errors.

The transport layer is supposed to chunk down the payload.

6.) **Cyclic Redundancy Check (CRC)** :

This field is of 15 bits, CRC is used to detect errors in transmitted data.
The transmitter calculates a CRC value for the payload data using a polynomial algorithm (e.g., CRC-15 or CRC-21) and the calculated CRC is placed into this field.
The receiver calculates the CRC value for the received payload data using the same polynomial algorithm. the receiver compares the calculated CRC value

with the received CRC value. If they match, the data is assumed to be error-free. If they don't match, an error is detected.

7.) **CRC Delimiter** : This is a 1 bit field which is just used to separate the CRC field from the next field.
This field's bit value must be dominant so that CRC ending can be detected by the differential amplifiers for which the idle state is recessive.

8.) **Acknowledge (ACK)** : This is also a 1 bit field, this is a very clever technique which is is implemented in the CAN data-link layer and CAN physical layer protocol, this allows the acknowledgement process in the communication to be completed without sending extra frames or packets.

At the time of the transmission of this bit, the frame sender will send a recessive bit i.e. drive the bus in logical 1 state or just monitors the potential by letting the bus to float to recessive state, driving the bus to recessive state. the hardware or nodes is designed in such a way under this protocol that they will respond to this bit by driving the bus to the dominant state i.e. the receiver when receives this bit, it overrides this bit with dominant bit which is monitored by frame sender as well, thus if the change in bus state is detected, the acknowlegdement gets completed.
If some of the node failed to receive the frame and dont respond to this bit, it will let drive the bus into the recessive state and since CAN bus is an AND bus, the bus will fall to recessive state which when detected by sender results in error handling and error frame transmission, then all nodes will take a little nap i.e. will free up the bus to recessive state and then transmitter will again transmit the frame.
The bus arbitration and collision detection mechanism is used here to complete the acknowledgement process, clever huhh…?

9.) **ACK delimiter** : This is same as the CRC delimiter, its just a 1 bit field which separates thhe ACK field from the EOF field.
This field is typically recessive.

10.) **End Of Frame (EOF)** :
This frame is used to indicate the end of the CAN frame, its size is 7 bit and all of them are recessive bits.

So these are the bits which are transmitted to exchange one CAN frame, but if there is continuous flow of CAN frames, there is a little delay added between consecutive frames in order to ensure that the bus is free and in its natural state i.e. in recessive state between each two frames.

Once the frame is received by a CAN controller, it just does the CRC check for the payload which the CAN frame is carrying. Once the CRC check is done, it passes the relevant part of the frame to the user program/firmware like the payload and CAN ID which is crucial since

it contains information which is necessary for application point of view.

If an error is detected, error frames will be sent by the transmitter and all node will be detached from the CAN bus for some small time, then the node will re-initiate the transmission.

frame overloading is also possible, this happens when the frame transmission rate is more than the processing power of individual node, this situation is also handled appropriately by this protocol.

A lot of error handling is done in CAN data-link layer protocol to avoid potential conflicts.

So, the CAN controller will take the 8 byte payload from firmware and encapsulate it into the CAN 2.0B frame and hand it over to the CAN transceiver.

3.) ***J1939 Transport layer protocol*** :

ECOM libraries are great in terms of providing the implementation of both J1939 Transport layer and J1939 Application layer protocols.

We don't need to implement this protocol as well.

let's just have a look inside this protocol.
If we just have a look at higher layer protocol or more specifically J1939 applications layer prtocol, it will provide us a sequence of byte i.e. payload or J1939 L7 packet.
For now, we'll just not look into the J1939 L7 packet and take it as a black box or simply a payload or a byte stream.
Now, say the byte stream has length of some 'N' bytes, and since the maximum size of data that can be loaded into a single CAN 2.0B frame is 8 bytes, so we need to distribute the N bytes into chunks of 8 bytes (where N>8).

But wait...

What about the information of Transport layer protocols ???
Where do we store the transport layer information like packet number etc which is useful in segmentation and reassembly of J1939 application packets ???

So, what one can do is to not take the 8 bytes chunks of J1939 application packet, instead one can take 7 or less bytes for than and 1 or more bytes will be used for transport-layer metadata like packet number/serial number etc.

Now, there are many ways via which one can implement the transport layer protocol in ECOM library,
we'll be discussing some of them...

* One way is as discussed just above, instead of filling 8 bytes data field entirely with application packet chunks, one can partially fill these 8 bytes with payload chunk and thus remaining bytes can be used to fill the transport layer metadata.

* A more efficient approach can be sending an initial frame before sending the actually the actual payload data which contains information like type of data i.e. it is a normal

data frame or a firmware update frame or a special error frame.

So, all the Transport layer stuff like J1939 Application packet segmentation, appending appropriate and relevant transport layer information, reassembly of CAN frame's data field etc are taken care by ECOM library's transport handlers.

In CAN 2.0B data-link layer protocol, we discussed that DLC field will be passed as some argument to the CAN controller handler functions or set into some CAN controller registers, transport handlers are supposed to do so, they will call the CAN 2.0B data-link handler functions to read the payload chunks from the memory and construct the CAN 2.0B frame.

For transporting chunks of a single J1939 Application packet, CAN hardware configuration generally kept same until and unless some specific requirement is necessary but it's not the part of our task.

4.) ***J1939 Application layer protocol*** :

As per our task description, we dont need to implement this protocol and thus use ECOM library for this as well.

The handlers of this protocols defined in ECOM libraries just needs the firmware/user data, and may require some additional information depending on library to library,
these handlers will create the entire J1939 Application packet, thus once J1939 Application packet is ready, transport handlers can be called upon for handling transportation-layer.

let's have a look at the J1939 Application packet's structure which is generally same in all ECOM libraries.

Structure of J1939 Application packet :

Application packet is divided into two fields : Header and User data.
Further, the 'Header' field is divided into 6 sub-fields.

Header (29 bits) :

Sub-fields	**Size (in bits)**	**Description**
Priority	3 bits	Indicates the priority of the message. The priority level ranges from 0 (lowest) to 7 (highest).
Reserved bit	1 bit	Reserved for future and is typically set to 0 i.e. dominant bit.
Data page	1 bit	Used to extend PDU format field.
PDU format	8 bits	Determines message format and

Protocol Data Unit		mode of message.
PDU specific	8 bits	Plays significant role in message addressing and routing.
Source address	8 bits	Represents the transmitting node address.

Let’s discuss each of the sub-fields in details...

1.) Priority bits :

this sub-field is necessary for ‘Bus arbitration’, it represents 7 different levels of priority (0 – 7), where 0 has the highest priority and thus priority decreases on increasing towards 7.
this sub-field covers 3 bits out of 29 bit J1939 header.

But priority is handled at data-link layer, then how this L1 protocol is judging the priority ???
Actually, its the higher layer protocol which decides the priority of the frame which CAN controller will transmit, the CAN controller is configured dynamically and this information along with all the sub-fields of header are mapped to CAN controller’s CAN ID register , thus CAN controller reads the CAN ID register and creates a CAN frame.

If two frames are having same priority bits, then subsequent bits of CAN ID will be used to prioritize one node over other via bus arbitration.

2.) Reserved bit :

This bit is reserved for future use and is typically dominant.

3.) Data page :

This is a single bit sub-field which is used to extend the next sub-field, this bit is typically 0 if next sub-field is not extended but is 1 when the next field is extended, it just doubles the count that next sub-field can address.

4.) PDU format (PF) :

This is a 8 bit field which represents the message format that the CAN frame will carry like whether the message is intended for broadcasting or for specific nodes.

There are two ranges of values for thi field :
a.) PDU1 format : values between 0x00 to 0xEF.
b.) PDU2 format : values between 0xF0 to 0xFF.

PDU1 format is used for specifying the node for which the frame is created, PDU2 format is used to broadcast the frame to different groups.

In PDU1 format, different values of this field represents different data like 0x44 may represent engine temperature while 0x55 may represent coolant temperature, so depending on this value, different handlers may be called.

In PDU2 format, different values of this field represents different data for being broadcast , like 0xF1 may represent vehicle diagnostics and 0xF3 may represent the enviromental conditions, basically message format for broadcasting instead of unicasting like PDU1 format.
PDU2 format values represents the general categories for broadcasting frame types

The entire protocol stack doesn't inherently supports the frame boradcasting to the specific subgroups of the CAN network, if PF contains value greater than 0xEF, then the frame will be broadcasted to all of the nodes.

Note that this field represents the type of message not the node identity.

5.) PDU specific (PS) :

This is a 8 bit field, what it represents actually depends on PF field, if the PF field contains PDU1 values, then PS will contain the address of that specific node for which this specific frame is created.
And if the PDU2 values are their in PF field, then it doesn't actually represents the destination address since the message has to be broadcasted to some specific node groups, in this case, this field works as extension of PF field and in conjugation with PF field, it gives more details about the frame like the message is of type PF : 0xF4 ; PS : 0x23 means the message frame/ frame data belongs to the category PF : 0xF4 and if we further categorize PF0xF4, then the frame may belong to sub-catefory PS0x23.
More clearly, one can say that when PF belongs to PDU2 format and thus PF represents general categories of the frame, in this case, PS represents the specific categories under those general categories to make the message more specified and detailed.

6.) Source address :

This represents the source address of the frame transmitting node,
It's generally used by higher layer protocols in order to call appropriate hanlders based on source address since messages from some superior node address can always be sensitive, thus can be used by higher layer protocols.

This field also limits the this entire protocol stack to only support a CAN network with maximum 255 nodes (including the programmer node as well.)
We wont' go into the details of the node address allocation, this can be done in two ways : static allocation and dynamic, but all this is handled by the protocol stack that we'll be using.

The data field of the J1939 Application protocol can max hold upto 1785 bytes.

When a J1939 Application packet is created, it contains both header and data fields, then since the type of J1939 packet is known like what payload is it carrying, and whether the packet is intended for specific node or for broadcast, message category etc,
thus all the information about L7 protocol is present into the header field,
This field plays a crucial role in CAN frame transmission and design, the CAN 2.0B frame contains a field named CAN ID, the header field from the J1939 application packet is directly mapped into each CAN frame carrying the chunk of that specific J1939

application packet since the J1939 application packet can be larger (~1800 bytes max).

When the application packet is ready (say its large enough thus needed to be chunked down), thus transport layer will chunk it down and then calls data-link layer handlers, in this case, the initial/first CAN frame will contain data that contains 29 bit header + 35 bits of data, but we actually don't need the header duplication since the header is being mapped to CAN ID field of each CAN frame carrying the chunks of the J1939 application packet, thus only user data from data field of J1939 application packet if chunked down and packed into the CAN frames where each CAN frame will have the J1939 application packet header mapped into its CAN ID.

The data-link layer is responsible for configuring the CAN controller and thus it maps the header field to CAN ID field.
Application lyer can also configure the CAN controller like configuring CAN ID register for each new J1939 application packet.