Collecting data from our system and uploading it to the cloud so we can process it and use it later.

In our car the system consists of number of controllers that gather information from different sensors and output data to some actuators, through some type of serial communication we can collect these data and send them to a main board, from it we can upload it to cloud our main board that has the ability to connect to the internet through WIFI or Internet USB is the raspberry pi3.

So the need to come up with some sort of communication protocol to handle the transferring of data from the controllers inside the car to our main board that is connected to the cloud became one of our duties.

We chose to build our protocol above the SPI (Serial Peripheral Interface) protocol as it's a full duplex serial communication protocol very efficient for the short distance communication besides it's the fastest serial communication protocol between the UART, I2C and SPI.

SPI is supported by H.W in the PI3 and the TIVA c controller, we've managed to implement the SPI handler driver for the TIVA c controller with respect to AUTOSAR_SWS_4.3.1_SPIHandlerDriver with some modifications and added APIs to facilitate working in slave mode as the PI3 slave mode isn't obvious and there are few number of documents speaking about the slave mode in SPI for PI3.

For that reason the TIVA C is the apparent slave in our communication and the pi3 is the master, but this is not totally true as in our protocol if TIVA C wants to send information it can trigger the PI3 to indicate that it needs to send information through a GPIO pin that connects the TIVA C and pi3.

This GPIO pin normal state is HIGH at the start of the communication this pin is pulled low so the PI3 can detect if there is a controller connected to it or not then after PI3 detects the TIVA C the pin is high again.

Whenever the TIVA C is ready and the data is in the H.W buffer of the TIVA C it pull the GPIO pin low so a falling edge is detected on the pi3 causing a H.W interrupt through that interrupt PI3 knows TIVA C wants to transmit a four bytes of data so PI3 pushes four bytes of data through the SPI bus so it can get the TIVA C four bytes of data and that is the **main** idea behind our protocol.

Four bytes of data pushed from PI3 to TIVA C trigger a H.W interrupt on the TIVA C as it's mentioned in the TIVA C data sheet that a H.W interrupt is triggered when there are four bytes or more in the TIVA C SPI H.W buffer.

From here we can deduce the communication protocol is event driven whenever the TIVA C wants to transmit useful information to PI3 all it has to do is to pull a GPIO pin from high to LOW, and for the PI3 whenever it needs to send useful data a four bytes sent from PI3 to TIVA C will do the job.

That was the H.W part, from it we can say that our protocol least number of bytes to be transmitted is four bytes so the SW-C (Software component) that wants to send data has to make sure that the amount of bytes it wants to transmit is a number multiple of four.

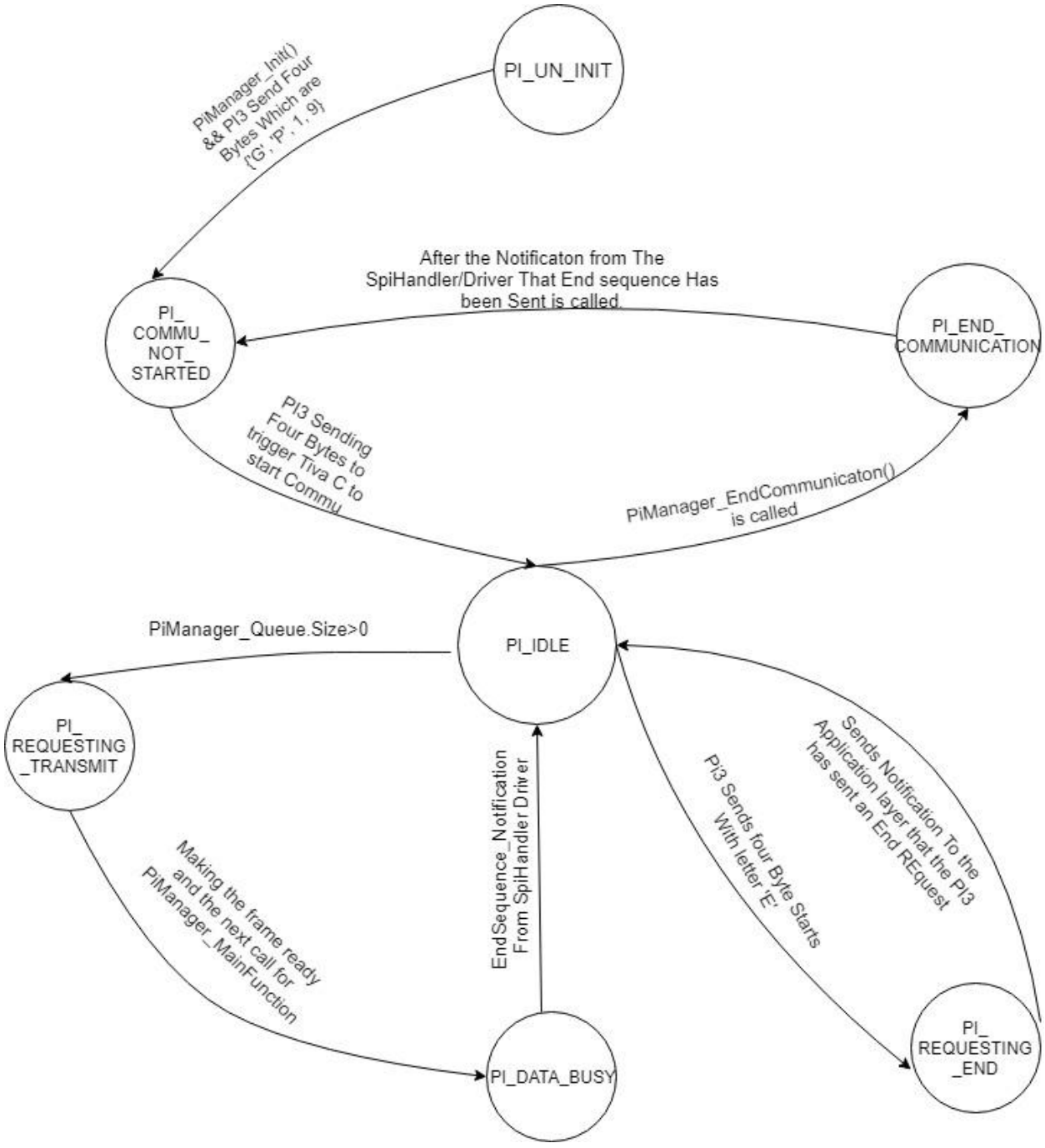
Any frame from our protocol starts with four bytes two bytes represent our protocol signature 'G', 'P' (graduation project), one byte for the frame length and last is the information ID so the PI3 can classify the data, and put it in its buffer.

To configure a new frame and send it correctly:

- Add the frame(channel) in the post build configuration PiManager_PBCfg.c file
- From the application layer you have to provide two pointers the first to send data from whenever there is a need to transmit data, the other to receive data in it from the PI3 API used
  Std_ReturnType PiManager_SetupEB(
  PiManager_ChannelType Channel, PiManager_DataBufferType* SrcDataBufferPtr,
  PiManager_DataBufferType* DesDataBufferPtr,
  PiManager_NumberOfDataType Length);
- Call the asynchronous transmit API which will store your request and whenever the PiManager is in idle state the request'll be handler Std_ReturnType PiManager_AsyncTransmit(
  PiManager_ChannelType Channel);

Our Protocol consists of two main finite state machines one for the TIVA C and the other one for the PI3 I'll try to explain them in the next pages.

TIVA C finite state machine:

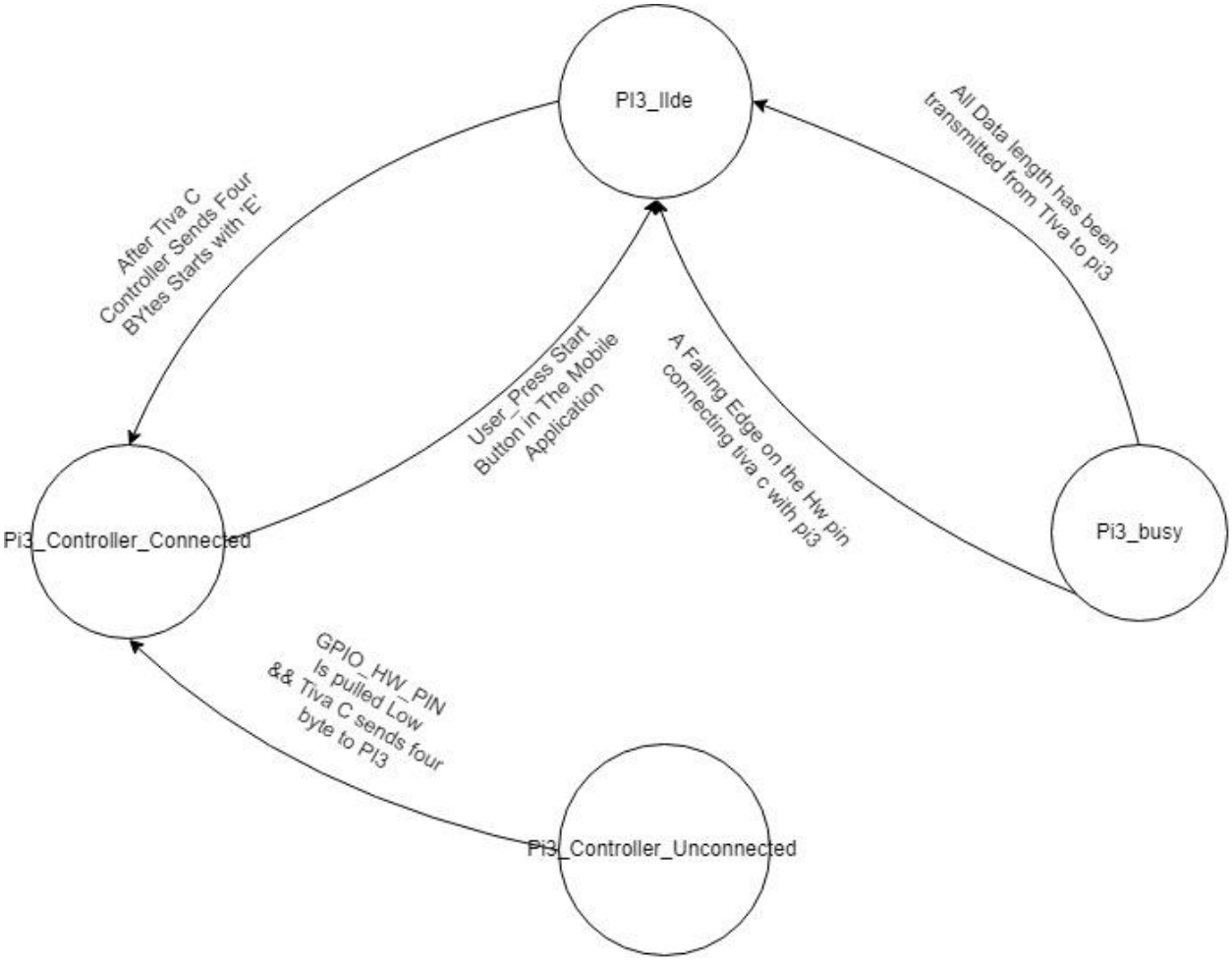


The main states in the TIVA C finite state machine:

- PI_UNINIT
- COMMU_NOT_STARTED
- IDLE
- REQUESTING_TRANSMIT
- DATA_BUSY
- REQUESTING_END
- END_COMMUNICATION

1. **After H.W reset** the Pi_Manager (software that manages the protocol) is in the UN_INIT state waiting for the SW-C to call the PiManager_Init(ConfigPtr) API, and for the PI3 to send four bytes which are {'G', 'P', 1, 9} as a check that PI3 is connected and the wires is connected successfully.
2. **Then a transition to COMMU_NOT_STARTED:** comes to wait for the PI3 to initiate the communication by sending and four bytes that will trigger a H.W interrupt in the TIVA C announcing a communication start.
3. **Then the Pi_Manager go into the IDLE state**: which is the main state that manages the requests from SW-C in TIVA C and any request from the PI3, if the Software buffer that manages the request from TIVA C has any request on it then the TIVA C has received valuable information and wants to transmit it then a transition to REQUESTING_TRANSMIT is a must here.
   However if the PI3 wants to send information like (The user has pressed END button) then a transition to REQUESTING END has the higher prioriy.
4. **REQUESTING_TRANSMIT state:** is the state which prepares the frame by adding the Frame ID, length and the signature of our protocol and requesting from the SPI handler to initiate the frame
5. **DATA_BUSY state:** is the state where the pi_manager waits for the SPIHandler notification that the data has been transmitted successfully to notify the SW-C**.**
6. **REQUESTING_END:** comes after the PI3 sends four bytes starts with letter 'E' to announce an END for the communication, a notification to the SW-C is sent.
7. **END_COMMUNICATION:** is entered after the SW-C call PiManager_EndCommunication() API a sequence is initiated to tell the PI3 to return to Controller Connected state and ends the ongoing communication.

PI3 Finite State Machine:



PI3 Main State:

- PI3_CONTROLLER_UNCONNECTED
- PI3_CONTROLLER_CONNECTED
- PI3_IDLE
- PI3_BUSY
- PI3_END

PI3_ CONTROLLER _UNCONNECTED this is the initial state PI3 starts from a transition to PI3_CONTROLLER_CONNECTED is a must when the H.W GPIO pin connecting PI3 and Controller is low, and the controller sends four bytes which are {‘G’, ‘P’, 1, 9}

PI3_CONTROLLER_CONNECTED: the state in which we wait for an event to start the communication in our case a start is fetched from the cloud, then four bytes are sent to the controller announcing a communication start

PI3_IDLE: manages the requests, checks the first four bytes of every frame to classify the message by checking the last two bytes the message length and the message ID.

PI3_BUSY: when a valid message ID is detected a transition to PI3_BUSY state is a must until the counter of the received bytes equal the message length that is transmitted in the first four bytes.

In our application we've five valid frames which are listed in the coming table with their ID and length

Frame	ID	Length
GPS	100	52 bytes
Left Servo Read	101	4 bytes
Right Servo Read	102	4 bytes
Left Servo Write	103	4 bytes
Right Servo Read	104	4 bytes
Temperature	105	4bytes
IMU	106	4bytes
END	'E'	4byes