Xnergy Autonomous Power Technologies Firmware Challenge

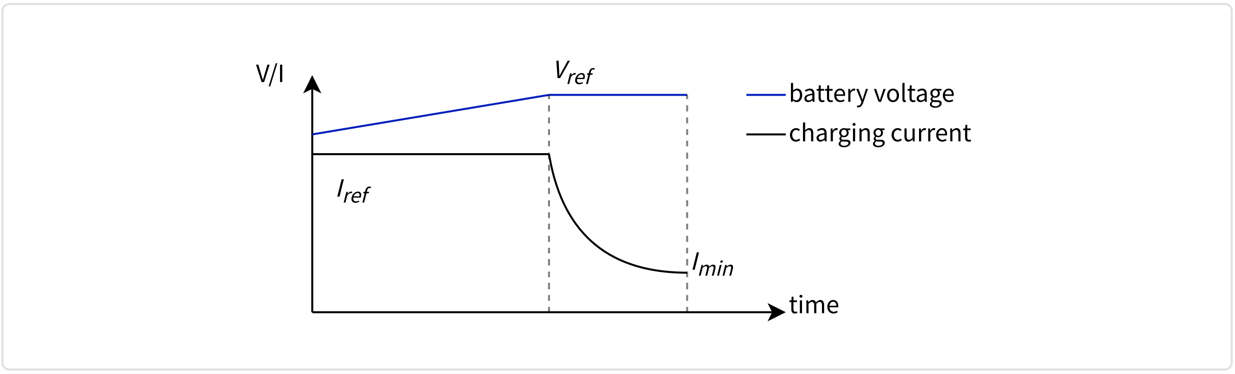
1. Implement a control algorithm for a charger that has constant-current constant-voltage (CCCV) by using c-language into a code skeleton here.

   As shown below, the charger shall have two control loops, current control and voltage control to realise CCCV charging curve as shown below. Each control loop shall have its own PI control loop.

   Iref is charging current reference during constant current stage.

   Vref is battery voltage reference during constant voltage stage.

   Imin is the minimum charging current which the charger shall stop after the charging current reach this value.



   The charger shall have 3 states, such as:

   a. *Idle* : charger wait enable_command to be **True** to go to the next state, constant current.
   b. *Constant current* : charger runs PI current control to regulate **current_feedback** to **current_reference**. When **voltage_feedback** reach **voltage_reference**, charger shall go to the next state, constant voltage.
   c. *Constant voltage*: charger runs PI voltage control to regulate **voltage_feedback** to **voltage_reference**. When **current_feedback** reach **minimum_current**, charger shall set the **enable_command** to False and go back to idle state.

C

```c
//put your definition here

void Initialization(void){
    //initialize your variables here
    ...
    ...
}
void control_routine(void){
    //run the control algorithm here
    ...
    ...
}
void main_state_machine(void){
    //run the state transition here
    ...
    ...
}
void main(void){
    Initialization();
    PieVectTable.EPWM1_INT = &control_routine;
    while(true){
        main_state_machine();
    }
```

2. Implement a CAN communication protocol between a charger and a battery management system (BMS) by using c-language into a code skeleton here.

   There are 3 network states that represent the state of charging and determine the CAN message to be received and transmitted.

   a. Initialization : this state is run once after the device is boot-up or initialized. After initialization is finished, the network will go to pre-operational state.
   b. Pre-operational : this state is to indicate the charger is idle and not charging. Once, the charger receives the enable_command, it will go to operational state.
   c. Operational : the charger is in operational mode or charging. If charging stops, the state should go back to pre-operational.

   Charger shall transmit heartbeat message every 1 second on all states.

CAN ID	Length	D[0]	Interval (ms)
0x701	1	0 : initialization	1000

		1 : pre-operational 2 : operational	

There are 3 network states that represent the state of charging and determine the CAN message to be received and transmitted.

Once the charger is connected to battery, bms will send voltage request, current request, and enable_command through a CAN message that is sent every 200ms.

If start charging command is received, network state shall change to operational.

CAN ID	**Length**	**Interval (ms)**
0x201	4	200
Data	**Parameter**	**Desciption**
Byte 0	voltage reference high	*10, ex: 321 = 32.1V
Byte 1	voltage reference low	
Byte 2	current reference high	*10, ex: 1000 = 100A
Byte 3	current reference low	
Byte 4	Enable command	0: stop Charging 1: start Charging

In operational network state, charger will start to send outgoing message to BMS in 200ms interval in addition to heartbeat message.

CAN ID	**Length**	**Interval (ms)**
0x181	4	200
Data	**Parameter**	**Desciption**
Byte 0	voltage feedback high	*10, ex: 321 = 32.1V
Byte 1	voltage feedback low	
Byte 2	current feedback high	*10, ex: 1000 = 100A
Byte 3	current feedback low	
Byte 4	Charging status	0: Not Charging 1: Charging

If stop charging command is received, charger shall stop the outgoing message to BMS but keep the heartbeat message. The network state will go back to pre-opreational in this case.

If there is no incoming message from BMS to charging in 5sec, charger shall stop the charging and change the network state to pre-operational.

a. Integrate the network management routine into the code in Question 1.

```c
//put your definition here

//CAN struct example
typedef struct {
        uint8_t  Data[8];
        uint16_t Length;
        uint32_t ID;
} CAN_msg_typedef;
CAN_msg_typedef Can_tx;
CAN_msg_typedef Can_rx;

void CAN_write(CAN_msg_typedef *msg);
bool CAN_read(CAN_msg_typedef *msg);    //return true if there is received msg

uint32_t time_ms;
void Initialization(void){
    //initialize your variables here
    ...
    ...
}
void control_routine(void){
    //run the control algorithm here
    ...
    ...

    time_ms++;    //assume INT frequency is 1kHz, for timing purpose
}
void main_state_machine(void){
    //run the state transition here
    ...
    ...
}
void CAN_write_handler(void){
    //CAN tx
    ...
    ...
}
```

```
void CAN_read_handler(void){
    //CAN tx
    ...
    ...
}
void network_management(void){
    //run the network management here
    ...
    ...
}
void main(void){
    Initialization();
    PieVectTable.EPWM1_INT = &control_routine;
    while(true){
        main_state_machine();
        network_management();
    }
```

b. Show the example of the heartbeat, incoming, and outgoing message when charging start and charging stop (idle) in **hexadecimal** (per byte of CAN data).

Heartbeat during idle

D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0x01	0x00	0x00	0x00	0x00	0x00	0x00	0x00

Heartbeat during charging

D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0x02	0x00	0x00	0x00	0x00	0x00	0x00	0x00

Heartbeat when stop charging

D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0x01	0x00	0x00	0x00	0x00	0x00	0x00	0x00

Incoming (BMS to charger) during start charging

D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0x01	0x41	0x00	0x64	1	0	0	0

Outgoing (Charger to BMS) during start charging

D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0x01	0x41	0x00	0x64	0x01	0x00	0x00	0x00

Incoming (BMS to charger) during stop charging

D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0x01	0x41	0x00	0x64	0x00	0x00	0x00	0x00

Outgoing (Charger to BMS) during stop charging

D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0x1	0x41	0x00	0x64	0x00	0x00	0x00	0x00

3. Setup a git remote repository for the code above and configure them to public.

Guide : for question 1 & 2, please ignore the low lever driver of the microcontroller (e.g. Clock, adc, interrupt initialization). You can use the provided dummy functions above as low level driver (CAN_write, CAN_read) as api.