Xnergy Autonomous Power Technologies Firmware Challenge

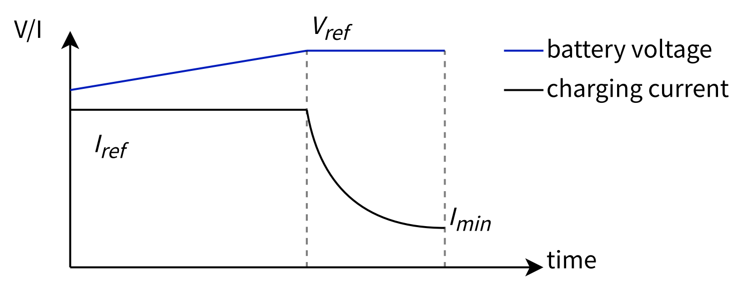
1. Implement a control algorithm for a charger that has constant-current constant-voltage (CCCV) by using c-language into a code skeleton here.

   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

```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							

Heartbeat during charging

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

Heartbeat when stop charging

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

Incoming (BMS to charger) during start charging

V_ref = 32.1 V I_ref = 100.0 A

D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0x01	0x41	0x03	0xE8	0x01			

Outgoing (Charger to BMS) during start charging

V_feedback = 32.1 V   I_feedback = 100.0 A

D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0x01	0x41	0x03	0xE8	0x01			

Incoming (BMS to charger) during stop charging

V_ref = 48.0 V   I_ref = I_min = 2.0 A

D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0x01	0xE0	0x00	0x14	0x00			

Outgoing (Charger to BMS) during stop charging

V_feedback = 48.0 V   I_feedback = I_min = 2.0 A

D[0]	D[1]	D[2]	D[3]	D[4]	D[5]	D[6]	D[7]
0x01	0xE0	0x00	0x14	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.

1. **Firmware Challenge: Question 1**

```
#include<stdbool.h>

//put definition here
#define VOLTAGE_REFERENCE   24.0    //in Volt
#define CURRENT_REFERENCE   3.0     //in Ampere
#define CURRENT_MINIMUM     0.3     //in Ampere
#define CC_PI_KP            1.0
#define CC_PI_KI            100.0
#define CC_PI_LIMIT_MIN
#define CC_PI_LIMIT_MAX
#define CV_PI_KP            1.0
#define CV_PI_KI            100.0
#define CV_PI_LIMIT_MIN
#define CV_PI_LIMIT_MAX
#define SAMPLING_PERIOD     0.0001

bool enable_command;
uint8_t state;
uint32_t PWM;
double voltage_feedback, voltage_reference;                          //Battery
terminal voltage
double current_feedback, current_reference, current_minimum;     //Battery
charging current
double charger_voltage;                                          //Charger DC
voltage
double duty_feedback, duty_reference, duty_PWM;

typedef struct {
    bool enable;
    double Kp, Ki;
    double error;
    double output;
    double output_limit_min, output_limit_max;
    double integrator_state;
    double dT;
} PI_CONTROL;

PI_CONTROL cc_pi_control;
PI_CONTROL cv_pi_control;

void Initialization(void){
    /*
    * initialize variables here
    */
    state = 0;
    enable_command = 0;
    voltage_reference = VOLTAGE_REFERENCE;
```

```c
    current_reference = CURRENT_REFERENCE;
    current_minimum = CURRENT_MINIMUM;

    cc_pi_control.integrator_state = 0.0;
    cc_pi_control.Kp = CC_PI_KP;
    cc_pi_control.Ki = CC_PI_KI;
    cc_pi_control.output_limit_min = CC_PI_LIMIT_MIN;
    cc_pi_control.output_limit_max = CC_PI_LIMIT_MAX;
    cc_pi_control.dT = SAMPLING_PERIOD;

    cv_pi_control.integrator_state = 0.0;
    cv_pi_control.Kp = CV_PI_KP;
    cv_pi_control.Ki = CV_PI_KI;
    cv_pi_control.output_limit_min = CV_PI_LIMIT_MIN;
    cv_pi_control.output_limit_max = CV_PI_LIMIT_MAX;
    cv_pi_control.dT = SAMPLING_PERIOD;

    duty_reference = 1.0;
}

void control_routine(void){
    /*
    * run the control algorithm here
    * Constant-Current state: run Current Control algorithm only
    * Constant-Voltage state: run Voltage Control algorithm and Current
Control algorithm
    */

   //Get voltage_feedback and current_feedback value
   //Assume that ADC_Conversion() convert the ADC_input value from ADC_channel
to Volt or Ampere
   voltage_feedback = ADC_Conversion(ADC_input, 1);
   current_feedback = ADC_Conversion(ADC_input, 2);

   //Voltage Control algorithm:
   if(cv_pi_control.enable){
    //Calculate error
    cv_pi_control.error = voltage_reference - voltage_feedback;

    //Calculate PI output
    cv_pi_control.output = cv_pi_control.integrator_state + cv_pi_control.Kp *
cv_pi_control.error;

    //Limit PI output if necessary
    if(cv_pi_control.output < cv_pi_control.output_limit_min)
        cv_pi_control.output = cv_pi_control.output_limit_min;
    else if(cv_pi_control.output > cv_pi_control.output_limit_max)
        cv_pi_control.output = cv_pi_control.output_limit_max;
```

```
    //Calculate next PI integrator state value when enable = true or
    //Reset PI integrator state value when enable = false
    cv_pi_control.integrator_state += cv_pi_control.Ki * cv_pi_control.error
*  cv_pi_control.dT;
   } else {
    cv_pi_control.integrator_state = 0.0;
   }


   //Current Control algorithm:
   if (cc_pi_control.enable){
    //Set current_reference value
    if (cv_pi_control.enable)
        current_reference = cv_pi_control.output;
    else
        current_reference = CURRENT_REFERENCE;

    //Calculate error
    cc_pi_control.error = current_reference - current_feedback;

    //Calculate PI output
    cc_pi_control.output = cc_pi_control.integrator_state + cc_pi_control.Kp *
cc_pi_control.error;

    //Limit PI output if necessary
    if(cc_pi_control.output < cc_pi_control.output_limit_min)
        cc_pi_control.output = cc_pi_control.output_limit_min;
    else if(cc_pi_control.output > cc_pi_control.output_limit_max)
        cc_pi_control.output = cc_pi_control.output_limit_max;

    //Calculate next PI integrator state value when enable = true or
    //Reset PI integrator state value when enable = false
    cc_pi_control.integrator_state += cc_pi_control.Ki * cc_pi_control.error
*  cc_pi_control.dT;
   } else {
    cc_pi_control.enable = 0.0;
   }

    //Assume that assume the PI output from Current control algorithm is used
to calculate the PWM duty cycle of the MCU
    //As in the proposed control block diagram
    duty_feedback = duty_reference - (voltage_feedback/charger_voltage);
    duty_PWM = cc_pi_control.output + duty_feedback;

    //Generate PWM with duty cycle = duty_PWM, 0.0 <= duty_PWM <= 1.0
    //with Generate_PWM()
    Generate_PWM(duty_PWM);
```

```c
}

void main_state_machine(void){
    /*
    * run the state transition here
    * Charging state for Constant-Current Constant-Voltage (CCCV) of charger:
    *    - state = 1 for Constant-Current State
    *    - state = 2 for Constant-Voltage State
    *    - default   for Idle State
    */

   enable_command = Digital_read(GPIO_input, 1);        //assume that
enable_command value from GPIO_input 1

    switch(state){
        case 1:
        control_routine();
        //State transition condition
        if (enable_command == false){
            state = 0;
            cv_pi_control.enable = false;
            cc_pi_control.enable = false;
            Generate_PWM(0.0);
        }
        if(enable_command == true && voltage_feedback >= voltage_reference){
            state = 2;
            cv_pi_control.enable = false;
            cc_pi_control.enable = true;
        }
    break;

        case 2:
        control_routine();
        if (enable_command == false || current_feedback == current_minimum){
            state = 0;
            cv_pi_control.enable = false;
            cc_pi_control.enable = false;
            Generate_PWM(0.0);
        }
        if(enable_command == true && voltage_feedback < voltage_reference){
            state = 1;
            cv_pi_control.enable = false;
            cc_pi_control.enable = true;
        }
    break;

        default:
        if (enable_command == true && voltage_feedback < voltage_reference){
```

```
            state = 1;
            cv_pi_control.enable = false;
            cc_pi_control.enable = true;
        }
        else if (enable_command == true && voltage_feedback >=
voltage_reference)
            state = 2;
            cv_pi_control.enable = false;
            cc_pi_control.enable = true;
    break;
    }
}

void main(void){
    Initialization();
    PieVectTable.EPWM1_INT = &control_routine;
    while(true){
        main_state_machine();
    }
}
```

2. **Firmware Challenge: Question 2**

```
#include<stdbool.h>

//put your definitions here
#define CURRENT_MINIMUM     0.3     //in Ampere
#define CC_PI_KP            1.0
#define CC_PI_KI            100.0
#define CC_PI_LIMIT_MIN
#define CC_PI_LIMIT_MAX
#define CV_PI_KP            1.0
#define CV_PI_KI            100.0
#define CV_PI_LIMIT_MIN
#define CV_PI_LIMIT_MAX
#define SAMPLING_PERIOD     0.001   //INT frequency is 1kHz

bool enable_command;
uint8_t charging_states, network_states;
uint32_t PWM;
double voltage_feedback, voltage_reference;                         //Battery
terminal voltage
double current_feedback, current_reference, current_minimum;    //Battery
charging current
double charger_voltage;                                             //Charger DC
voltage
double duty_feedback, duty_reference, duty_PWM;

uint32_t prev_time_heartbeat, prev_time_bms_rx, prev_time_bms_tx;

//PI Control Struct
typedef struct {
    bool enable;
    double Kp, Ki;
    double error;
    double output;
    double output_limit_min, output_limit_max;
    double integrator_state;
    double dT;
} PI_CONTROL;

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

PI_CONTROL cc_pi_control;
PI_CONTROL cv_pi_control;
```

```c
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 variables here
    */
    charging_states = 0;
    network_states = 0;
    enable_command = 0;

    current_minimum = CURRENT_MINIMUM;

    cc_pi_control.integrator_state = 0.0;
    cc_pi_control.Kp = CC_PI_KP;
    cc_pi_control.Ki = CC_PI_KI;
    cc_pi_control.output_limit_min = CC_PI_LIMIT_MIN;
    cc_pi_control.output_limit_max = CC_PI_LIMIT_MAX;
    cc_pi_control.dT = SAMPLING_PERIOD;

    cv_pi_control.integrator_state = 0.0;
    cv_pi_control.Kp = CV_PI_KP;
    cv_pi_control.Ki = CV_PI_KI;
    cv_pi_control.output_limit_min = CV_PI_LIMIT_MIN;
    cv_pi_control.output_limit_max = CV_PI_LIMIT_MAX;
    cv_pi_control.dT = SAMPLING_PERIOD;

    duty_reference = 1.0;

    network_management();
    prev_time_heartbeat = 0;
    prev_time_bms_rx = 0;
    prev_time_bms_tx = 0;
}

void control_routine(void){
    /*
    * run the control algorithm here
    * Constant-Current state: run Current Control algorithm only
    * Constant-Voltage state: run Voltage Control algorithm and Current
Control algorithm
    */
```

```
   //Get voltage_feedback and current_feedback value
   //Assume that ADC_Conversion() convert the ADC_input value from ADC_channel
to Volt or Ampere
   voltage_feedback = ADC_Conversion(ADC_input, 1);
   current_feedback = ADC_Conversion(ADC_input, 2);

   //Voltage Control algorithm:
   if(cv_pi_control.enable){
    //Calculate error
    cv_pi_control.error = voltage_reference - voltage_feedback;

    //Calculate PI output
    cv_pi_control.output = cv_pi_control.integrator_state + cv_pi_control.Kp *
cv_pi_control.error;

    //Limit PI output if necessary
    if(cv_pi_control.output < cv_pi_control.output_limit_min)
        cv_pi_control.output = cv_pi_control.output_limit_min;
    else if(cv_pi_control.output > cv_pi_control.output_limit_max)
        cv_pi_control.output = cv_pi_control.output_limit_max;

    //Calculate next PI integrator state value when enable = true or
    //Reset PI integrator state value when enable = false
    cv_pi_control.integrator_state += cv_pi_control.Ki * cv_pi_control.error
*  cv_pi_control.dT;
   } else {
    cv_pi_control.integrator_state = 0.0;
   }


   //Current Control algorithm:
   if (cc_pi_control.enable){
    //Set current_reference value
    if (cv_pi_control.enable)
        current_reference = cv_pi_control.output;

    //Calculate error
    cc_pi_control.error = current_reference - current_feedback;

    //Calculate PI output
    cc_pi_control.output = cc_pi_control.integrator_state + cc_pi_control.Kp *
cc_pi_control.error;

    //Limit PI output if necessary
    if(cc_pi_control.output < cc_pi_control.output_limit_min)
        cc_pi_control.output = cc_pi_control.output_limit_min;
    else if(cc_pi_control.output > cc_pi_control.output_limit_max)
        cc_pi_control.output = cc_pi_control.output_limit_max;
```

```
    //Calculate next PI integrator state value when enable = true or
    //Reset PI integrator state value when enable = false
    cc_pi_control.integrator_state += cc_pi_control.Ki * cc_pi_control.error
*   cc_pi_control.dT;
   } else {
    cc_pi_control.enable = 0.0;
   }

    //Assume that assume the PI output from Current control algorithm is used
to calculate the PWM duty cycle of the MCU
    //As in the proposed control block diagram
    duty_feedback = duty_reference - (voltage_feedback/charger_voltage);
    duty_PWM = cc_pi_control.output + duty_feedback;

    //Generate PWM with duty cycle = duty_PWM, 0.0 <= duty_PWM <= 1.0
    //with Generate_PWM()
    Generate_PWM(duty_PWM);

    time_ms++;      //assume INT frequency is 1kHz, for timing purpose
}

void main_state_machine(void){
    /*
    * run the state transition here
    * Charging state for Constant-Current Constant-Voltage (CCCV) of charger:
    *    - charging state = 1 for Constant-Current State
    *    - charging state = 2 for Constant-Voltage State
    *    - default   for Idle State
    */

    switch(charging_states){
        case 1:
        control_routine();
        //State transition condition
        if (enable_command == false){
            charging_states = 0;
            cv_pi_control.enable = false;
            cc_pi_control.enable = false;
            Generate_PWM(0.0);
        }
        if(enable_command == true && voltage_feedback >= voltage_reference){
            charging_states = 2;
            cv_pi_control.enable = false;
            cc_pi_control.enable = true;
        }
    break;
```

```
        case 2:
        control_routine();
        if (enable_command == false || current_feedback == current_minimum){
            charging_states = 0;
            cv_pi_control.enable = false;
            cc_pi_control.enable = false;
            Generate_PWM(0.0);
        }
        if(enable_command == true && voltage_feedback < voltage_reference){
            charging_states = 1;
            cv_pi_control.enable = false;
            cc_pi_control.enable = true;
        }
    break;

        default:
        if (enable_command == true && voltage_feedback < voltage_reference){
            charging_states = 1;
            cv_pi_control.enable = false;
            cc_pi_control.enable = true;
        }
        else if (enable_command == true && voltage_feedback >=
voltage_reference)
            charging_states = 2;
            cv_pi_control.enable = false;
            cc_pi_control.enable = true;
    break;
    }
}

void CAN_write_handler(void){
    /*
    * CAN tx
    */

   if(Can_tx.ID == 0x181){
    //Prepare Can_tx.Data from voltage_feedback, current_feedback,
enable_command
    uint16_t voltage_feedback_tx, current_feedback_tx;

    voltage_feedback_tx = (uint16_t) (voltage_feedback * 10);
    Can_tx.Data[0] = (uint8_t) (voltage_feedback_tx >> 8 & 0xFF);
    Can_tx.Data[1] = (uint8_t) (voltage_feedback_tx >> 0 & 0xFF);

    current_feedback_tx = (uint16_t) (current_feedback * 10);
    Can_tx.Data[2] = (uint8_t) (current_feedback_tx >> 8 & 0xFF);
    Can_tx.Data[3] = (uint8_t) (current_feedback_tx >> 0 & 0xFF);
```

```
    Can_tx.Data[4] = (uint8_t) enable_command;
   }
   CAN_write(&Can_tx);
   }

void CAN_read_handler(void){
    /*
    * CAN rx
    */
   if(Can_rx.ID == 201){
    if (Can_rx.Data[4]) {
        //Prepare Can_rx.Data for voltage_reference, current_reference,
enable_command
        uint16_t voltage_reference_rx = 0;
        voltage_reference_rx += Can_rx.Data[1] << 0;
        voltage_reference_rx += Can_rx.Data[0] << 8;

        voltage_reference = ((double) voltage_reference_rx) / 10;

        uint16_t current_reference_rx = 0;
        current_reference_rx += Can_rx.Data[3] << 0;
        current_reference_rx += Can_rx.Data[2] << 8;

        current_reference = ((double) current_reference_rx) / 10;

        enable_command = Can_rx.Data[4];
        network_states = 2;
    } else {
        enable_command = false;
        network_states = 1;
    }
   }
}

void network_management(void){
    /*
    * run the network management here
    * Network states:
    *   0 : Initialization state
    *   1 : Pre-Operational state
    *   2 : Operational state
    *
    * CAN ID:
    *   0x701 : Charger ID when transmit heartbeat
    *   0x181 : Charger ID when transmit data to BMS
    *   0x201 : BMS ID
    */
```

```
    //Initialization state
    if(network_states == 0){
     Can_tx.ID = 0x701;
     Can_tx.Length = 1;
     Can_tx.Data[0] = 0;
     CAN_write_handler();
     network_states = 1;
     prev_time_heartbeat = time_ms;

     //Pre-Operational state
    } else if (network_states == 1){
     if((time_ms-prev_time_heartbeat) >= 1000){
         Can_tx.ID = 0x701;
         Can_tx.Length = 1;
         Can_tx.Data[0] = 1;

         CAN_write_handler();
         prev_time_heartbeat = time_ms;
     }

     if(CAN_read(&Can_rx) && Can_rx.ID == 0x201 && (time_ms-prev_time_bms_rx)
>= 200){
         CAN_read_handler();
         prev_time_bms_rx = time_ms;
     }

    //Operational state
    } else if(network_states == 2){
     if((time_ms-prev_time_heartbeat) >= 1000){
         Can_tx.ID = 0x701;
         Can_tx.Length = 1;
         Can_tx.Data[0] = 2;
         CAN_write_handler();
         prev_time_heartbeat = time_ms;
     }

        if(CAN_read(&Can_rx) && Can_rx.ID == 0x201 && (time_ms-
prev_time_bms_rx) >= 200 && (time_ms-prev_time_bms_rx < 5000)){
         CAN_read_handler();
         prev_time_bms_rx = time_ms;
     } else if (time_ms-prev_time_bms_rx >= 5000)
         network_states == 1;

        if((time_ms-prev_time_bms_tx) >= 200){
         Can_tx.ID = 0x181;
         Can_tx.Length = 5;
         CAN_write_handler();
     }
```

```
    }
}

void main(void){
    Initialization();
    PieVectTable.EPWM1_INT = &control_routine;
    while(true){
        main_state_machine();
        network_management();
    }
}
```