```
#include <SPI.h>
#include <mcp2515.h>

// all can frames we will send
struct can_frame BmsLimits;
struct can_frame BmsSOC;
struct can_frame BmsStatus1;
struct can_frame BmsErrors;
struct can_frame ACCharging;

// standard can frame for reseaving
struct can_frame canMsg;

MCP2515 mcp2515(10); // enable the can controller

// some global variables to keep track of the parameters we are sending
float current = 0;
uint8_t soc = 0;
uint16_t voltage = 0xffff;
uint16_t maxACCurrent = 0;

int maxcellbalansing = 0;
int maxceltemp = 255;
int maxcelmodtemp = 255;

bool ohshit = false;
bool Power_Reduction = false;

// use millis() to periodically send the can frames (a timer interupt based implemantation would be more elagant for furure improvments, also I am a bit woried about what happens when millis() overflows)
unsigned long previousMillis = 0;
const long interval = 100;

// setup (runs ones)
void setup() {
  Serial.begin(115200); // start serial monitoring at 115200 baud

  // start the CAN bus
  mcp2515.reset();
  mcp2515.setBitrate(CAN_500KBPS, MCP_8MHZ); // can speeds for both the zero_ev ccs controller and emus_bms(when selected 500)
  mcp2515.setNormalMode();

  Serial.println("Done setting up, starting transmitting can frames");
}

// loop (loops infinit or untill stoped)
void loop() {
  unsigned long currentMillis = millis(); // get the current time

  if (currentMillis - previousMillis >= interval) {
    previousMillis = currentMillis;


    // charge parameters
    int current = 80;
    uint16_t ChargeVoltageLimit = 393 * 10; // 393
```

```
    uint16_t ChargeCurrentLimit = current * 10;
    uint16_t DischargeVoltageLimit = 307 * 10;
    uint16_t DischargeCurrentLimit = current * 10;

    if (Power_Reduction) {
      ChargeCurrentLimit = ChargeCurrentLimit / 2;
      DischargeCurrentLimit = DischargeCurrentLimit / 2;
    }


    uint16_t HvBatterySOC = soc; // soc sent of CCS_Controller

    uint16_t HvBatteryVoltage = voltage / 10; // battery voltage sent to CCS_Controller
    int16_t HvBatteryCurrent = current * 10; // battery current sent to CCS_Controller


    int16_t HvBatteryTemp = 0;
    if (maxceltemp != 255) { // if no cel sensors are connected (wich they should be but ok) use the cel module temperature for the CCS_Controller
      HvBatteryTemp = maxceltemp * 10;
    } else if (maxcelmodtemp != 255) {
      HvBatteryTemp = maxcelmodtemp * 10;
    }

    uint16_t AClimit = maxACCurrent * 10; // the ac current thet the CCS_Controller gives to the bms, wich the bms tells the ac charger
    uint16_t ACvolt = 230 * 10; // we are in europa so 230 is the standard.


    // set all the data of the can frames
    BmsLimits.can_id  = 0x351; // 849
    BmsLimits.can_dlc = 8;
    BmsLimits.data[0] = (ChargeVoltageLimit & 0xff);
    BmsLimits.data[1] = (ChargeVoltageLimit >> 8);
    BmsLimits.data[2] = (ChargeCurrentLimit & 0xff);
    BmsLimits.data[3] = (ChargeCurrentLimit >> 8);
    BmsLimits.data[4] = (DischargeVoltageLimit & 0xff);
    BmsLimits.data[5] = (DischargeVoltageLimit >> 8);
    BmsLimits.data[6] = (DischargeCurrentLimit & 0xff);
    BmsLimits.data[7] = (DischargeCurrentLimit >> 8);

    int iserror = 0;
    if (ohshit) {
      iserror = 0xff;
    } else {
      iserror = 0x00;
    }

    BmsErrors.can_id  = 0x35A; // 858 these are the errors that the CCS_Controller can exept
    BmsErrors.can_dlc = 4;
    BmsErrors.data[0] = iserror;
    BmsErrors.data[1] = iserror;
    BmsErrors.data[2] = iserror;
    BmsErrors.data[3] = iserror;

    BmsSOC.can_id  = 0x355; // 853
```

```
    BmsSOC.can_dlc = 6;
    BmsSOC.data[0] = (HvBatterySOC & 0xff);
    BmsSOC.data[1] = (HvBatterySOC >> 8);
    BmsSOC.data[2] = 0xff; // doese not have to be implemented in the case of the CCS_Controller
    BmsSOC.data[3] = 0xff; // doese not have to be implemented in the case of the CCS_Controller
    BmsSOC.data[4] = 0xff; // doese not have to be implemented in the case of the CCS_Controller
    BmsSOC.data[5] = 0xff; // doese not have to be implemented in the case of the CCS_Controller

    BmsStatus1.can_id  = 0x356; // 854
    BmsStatus1.can_dlc = 6;
    BmsStatus1.data[0] = (HvBatteryVoltage & 0xff);
    BmsStatus1.data[1] = (HvBatteryVoltage >> 8);
    BmsStatus1.data[2] = (HvBatteryCurrent & 0xff);
    BmsStatus1.data[3] = (HvBatteryCurrent >> 8);
    BmsStatus1.data[4] = (HvBatteryTemp & 0xff);
    BmsStatus1.data[5] = (HvBatteryTemp >> 8);

    ACCharging.can_id = 0x19B50407;
    ACCharging.can_dlc = 8;
    ACCharging.data[0] = 0xff;
    ACCharging.data[1] = 0xff;
    ACCharging.data[2] = (AClimit >> 8);
    ACCharging.data[3] = (AClimit & 0xff);
    ACCharging.data[4] = (ACvolt >> 8);
    ACCharging.data[5] = (ACvolt & 0xff);
    ACCharging.data[6] = 0xff;
    ACCharging.data[7] = 0xff;


    // send the can frames

    if (voltage !=  0xffff) { // only send the frames after the bms is initilized and we started reseaving data form it

      mcp2515.sendMessage(&BmsLimits);  // send the bms limits to the CCS_Controller
      mcp2515.sendMessage(&BmsErrors);  // send the errors (only sends 0's for now) to the CCS_Controller
      mcp2515.sendMessage(&BmsSOC);     // send the soc to the CCS_Controller
      mcp2515.sendMessage(&BmsStatus1); // send the bms status (voltage, current and temp) to the CCS_Controller
      mcp2515.sendMessage(&ACCharging); // sent the ac charging limits to the EMUS_BMS

      // Serial.println("all frames sent, small delay and repeat");
    }
    else {
      Serial.println("not sending can to ccs yet. is emus still booting?");
    }
  }

  // read the can bus
  if (mcp2515.readMessage(&canMsg) == MCP2515::ERROR_OK) {
    unsigned long id = canMsg.can_id; // the can id, so we can read only the
```

```
frames we want.

    if (id == 874) { // 874 is the id of a can frame that is sent from the ccs controller with the errors
      if (canMsg.data[0] & (1 << 0)) {
        Serial.print("HVPreChargeFault ");
      }
      if (canMsg.data[0] & (1 << 1)) {
        Serial.print("CCSContactorFault ");
      }
      if (canMsg.data[0] & (1 << 2)) {
        Serial.print("HVILFault ");
      }
      if (canMsg.data[0] & (1 << 3)) {
        Serial.print("BMSCommsFault ");
      }
      if (canMsg.data[0] & (1 << 4)) {
        Serial.print("BMSFault ");
      }
      if (canMsg.data[0] & (1 << 5)) {
        Serial.print("CCSECUFault ");
      }
      if (canMsg.data[0] & (1 << 6)) {
        Serial.print("PTCTempFault ");
      }
      if (canMsg.data[0] & (1 << 7)) {
        Serial.print("ChargeProtocolFault ");
      }
      if (canMsg.data[1] & (1 << 0)) {
        Serial.print("IncompatibleCCSCharger ");
      }
      if (canMsg.data[1] & (1 << 1)) {
        Serial.print("ChargeMode ");
      }
      if (canMsg.data[1] & (1 << 2)) {
        Serial.print("PlugPresent ");
      }
      if (canMsg.data[1] & (1 << 3)) {
        Serial.print("InletMotor ");
      }
      if (canMsg.data[1] & (1 << 4)) {
        Serial.print("CCSContactorStatus ");
      }
      if (canMsg.data[1] & (1 << 5)) {
        Serial.print("HVPresent ");
      }
      if (canMsg.data[1] & (1 << 6)) {
        Serial.print("InletFault ");
      }
      if (canMsg.data[1] & (1 << 7)) {
        Serial.print("StopchargeSwitch ");
      }

      Serial.print("BatteryVoltageSense:");
      Serial.print(((canMsg.data[2] << 8) | canMsg.data[3]) / 10.0);
      Serial.print(" CCSVoltageSense:");
      Serial.print(((canMsg.data[4] << 8) | canMsg.data[5]) / 10.0);

      Serial.println();
```

```
    }


    if ( id == 855 ) { // 855 is the id of the can frame that is sent from the ccs controller with the ac current limit
      maxACCurrent = canMsg.data[1];
      //Serial.print("max ac current: "); //maxACCurrent
      //Serial.println(maxACCurrent);
    }


    if (id == 0x99B50500) { // curent and soc
      uint8_t current1 = canMsg.data[0];
      uint8_t current2 = canMsg.data[1];
      current = (current1 << 8 | current2) / 10.0;
      Serial.print("amps: ");
      Serial.println(current);
      soc = canMsg.data[6];
      Serial.print("usoc: ");
      Serial.println(soc);
    }


    if (id == 0x99B50001) { // pack voltage
      uint8_t voltage1 = canMsg.data[3];
      uint8_t voltage2 = canMsg.data[4];
      Serial.print("volt: ");
      voltage = voltage1 << 8 | voltage2;
      Serial.println(voltage / 100.0);
    }


    if ( id == 0x99B50003 ) { // r we balencing? howmuch is the moste
      maxcellbalansing = map(canMsg.data[1], 0, 255, 0, 100);
      Serial.print("balencing: ");
      Serial.println(maxcellbalansing);
    }


    if ( id == 0x99B50002 ) { // cel module temp
      maxcelmodtemp = canMsg.data[1] - 100;
      Serial.print("max cel module temp: ");
      Serial.println(maxcelmodtemp);
    }


    if ( id == 0x99B50008 ) { // cel temp
      maxceltemp = canMsg.data[1] - 100;
      Serial.print("max cel temp: ");
      Serial.println(maxceltemp);
    }


    if ( id == 0x99B50000 ) { // info about bms, intresting is power reduction
      Serial.println("rx Overall Parameters");
      if (canMsg.data[0] & (1 << 0)) {
        Serial.print("Charger_Enable ");
```

```
      }
      if (canMsg.data[0] & (1 << 1)) {
        Serial.print("Heater_Enable ");
      }
      if (canMsg.data[0] & (1 << 2)) {
        Serial.print("Battery_Contactor ");
      }
      if (canMsg.data[0] & (1 << 3)) {
        Serial.print("Battery_Fan ");
      }
      if (canMsg.data[0] & (1 << 4)) {
        Serial.print("Power_Reduction ");
        Power_Reduction = true;
      } else {
        Power_Reduction = false;
      }
      if (canMsg.data[0] & (1 << 5)) {
        Serial.print("Charging_Interlock ");
      }
      if (canMsg.data[0] & (1 << 6)) {
        Serial.print("DCDC_Control ");
      }
      if (canMsg.data[0] & (1 << 7)) {
        Serial.print("Contactor_Pre-Charge ");
      }
      if (canMsg.data[0] != 0) {
        Serial.println();
      }
    }


    if ( id == 0x99B50007 ) { // energency states of bms

      Serial.println("rx energency states");
      Serial.println(canMsg.data[0], BIN);
      Serial.println(canMsg.data[1], BIN);
      Serial.println(canMsg.data[2], BIN);
      Serial.println(canMsg.data[3], BIN);

      if ((canMsg.data[0] != 0 | canMsg.data[0] != 0b10000000) &
canMsg.data[2] != 0) {
        ohshit = true;
      } else {
        ohshit = false;
      }
      if (ohshit) {
        Serial.println("oh shit");
      }

      if (canMsg.data[0] & (1 << 0)) {
        Serial.print("Under-voltage ");
      }
      if (canMsg.data[0] & (1 << 1)) {
        Serial.print("Over-voltage ");
      }
      if (canMsg.data[0] & (1 << 2)) {
        Serial.print("Discharge_Over-current ");
      }
      if (canMsg.data[0] & (1 << 3)) {
```

```
      if (canMsg.data[0] & (1 << 3)) {
        Serial.print("Charge_Over-current ");
      }
      if (canMsg.data[0] & (1 << 4)) {
        Serial.print("Cell_Module Overheat ");
      }
      if (canMsg.data[0] & (1 << 5)) {
        Serial.print("Leakage ");
      }
      if (canMsg.data[0] & (1 << 6)) {
        Serial.print("No Cell Communication ");
      }

      if (canMsg.data[2] & (1 << 3)) {
        Serial.print("Cell_Overheat ");
      }
      if (canMsg.data[2] & (1 << 4)) {
        Serial.print("No_Current_Sensor ");
      }
      if (canMsg.data[2] & (1 << 5)) {
        Serial.print("Pack_Under-Voltage ");
      }

      if (canMsg.data[0] != 0 & canMsg.data[2] != 0) {
        Serial.println();
      }
    }


  }
}

```