```
/*-----------------------------------------------------------------------
 *
 *   Project: PYLON TECH BATTERY Emulator for DiyBMS
 *            Using Canbus @ 500kbps and 11-bit addresses.
 *
 *   Author:  Trajilovic Goran (ZoMiGo)
 *
 * ----------------------------------------------------------------------
 *
 *   Description:
 *    - Manages communication between PylonTech batteries, DiyBMS, and Victron via CAN bus.
 *    - Detects when a module requests "Charge Stop" and adjusts the charging current accordingly.
 *   - Calculates and sends the corrected charge current (0x386) to Victron.
 *     - Automatically detects failed slave modules and excludes them from charging calculations.
 *   - Transmits highest and lowest cell voltages per module (0x381 + Module ID).
 *   - Sends BMS identification (0x305) to Victron.
 *    - Implements specific PylonTech CAN messages (0x351, 0x355, 0x356, 0x359, 0x35C, 0x35E).
 *    - Ensures that Victron dynamically adjusts the charge current instead of abruptly stopping charging.
 *
 */

#include "pylon_canbus.h"
#include "bms_id_manager.h"

extern uint8_t totalSlaves;
#define USE_ESP_IDF_LOG 1
static constexpr const char *const TAG = "diybms-pylon";

ConsolidatedData consolidatedData;

uint16_t lowest_cell_voltage_per_module[16];
uint16_t highest_cell_voltage_per_module[16];
bool module_charging_allowed[16];

/**
 * Sends the BMS identification to Victron (CAN ID: 0x305).
 * This message identifies the BMS as "PYLONTEC" to ensure compatibility.
 */
void sendDeviceIdentification() {
    uint8_t data[8] = {'P', 'Y', 'L', 'O', 'N', 'T', 'E', 'C'};
    send_canbus_message(0x305, data, sizeof(data));
}

/**
  * Sends the module status including highest/lowest cell voltage and charging permissions.
 * This is sent to Victron using CAN ID: 0x381 + Module ID.
 */
void send_module_status() {
    uint8_t data[8] = {0};
    for (uint8_t i = 0; i < totalSlaves; i++) {
        data[0] = i + 1;
        data[1] = (lowest_cell_voltage_per_module[i] >> 8) & 0xFF;
        data[2] = lowest_cell_voltage_per_module[i] & 0xFF;
        data[3] = (highest_cell_voltage_per_module[i] >> 8) & 0xFF;
        data[4] = highest_cell_voltage_per_module[i] & 0xFF;
        data[5] = module_charging_allowed[i] ? 0x01 : 0x00;
        send_canbus_message(0x381 + i, data, sizeof(data));
    }
}

/**
 * Calculates and sends the corrected charge current (CAN ID: 0x386).
 * If some modules request "Charge Stop," their current is subtracted.
 */
void send_corrected_charge_current() {
    int16_t current_charge = currentMonitor.modbus.current;
    int16_t stopped_module_current = 0;
    for (uint8_t i = 0; i < totalSlaves; i++) {
        if (!module_charging_allowed[i]) stopped_module_current += 10;
    }
    int16_t corrected_charge_current = current_charge - stopped_module_current;
    if (corrected_charge_current < 0) corrected_charge_current = 0;
     uint8_t data[8] = {(corrected_charge_current >> 8) & 0xFF, corrected_charge_current
& 0xFF};
    send_canbus_message(0x386, data, sizeof(data));
}

/**
 * Sends battery voltage, charge/discharge current limits (CAN ID: 0x351).
 */
void pylon_message_351() {
    struct data351 {
        uint16_t battery_charge_voltage;
        int16_t battery_charge_current_limit;
        int16_t battery_discharge_current_limit;
        uint16_t battery_discharge_voltage;
    };

    data351 data;
    data.battery_discharge_voltage = mysettings.dischargevolt;

    uint16_t default_charge_voltage = 1;
    int16_t default_charge_current_limit = 1;
    int16_t default_discharge_current_limit = 1;

    if (mysettings.canbusinverter == CanBusInverter::INVERTER_DEYE) {
        default_charge_voltage = rules.lowestBankVoltage / 100;
        default_charge_current_limit = 0;
        default_discharge_current_limit = 0;
    }

    data.battery_charge_voltage = default_charge_voltage;
    data.battery_charge_current_limit = default_charge_current_limit;
    data.battery_discharge_current_limit = default_discharge_current_limit;

    if (rules.IsChargeAllowed(&mysettings)) {
        if (rules.numberOfBalancingModules > 0 && mysettings.stopchargebalance) {
        } else {
            data.battery_charge_voltage = rules.DynamicChargeVoltage();
            data.battery_charge_current_limit = rules.DynamicChargeCurrent();
        }
    }

    if (rules.IsDischargeAllowed(&mysettings)) {
        data.battery_discharge_current_limit = mysettings.dischargecurrent;
    }

    send_canbus_message(0x351, (uint8_t *)&data, sizeof(data351));
}

/**
 * Sends the State of Charge (SoC) and State of Health (SoH) (CAN ID: 0x355).
 */
void pylon_message_355() {
    if (_controller_state != ControllerState::Running) return;

    struct data355 {
        uint16_t stateofchargevalue;
        uint16_t stateofhealthvalue;
    };

    if (mysettings.currentMonitoringEnabled && currentMonitor.validReadings) {
        data355 data;
              data.stateofchargevalue  =  rules.StateOfChargeWithRulesApplied(&mysettings,
currentMonitor.stateofcharge);
        data.stateofhealthvalue = (uint16_t)(trunc(mysettings.soh_percent));

        send_canbus_message(0x355, (uint8_t *)&data, sizeof(data355));
    }
}

/**
 * Sends battery voltage, current, and temperature (CAN ID: 0x356).
 */
void pylon_message_356() {
    struct data356 {
        int16_t voltage;
        int16_t current;
        int16_t temperature;
    };

    data356 data;

    if (mysettings.currentMonitoringEnabled && currentMonitor.validReadings) {
        data.voltage = currentMonitor.modbus.voltage * 100.0;
        data.current = currentMonitor.modbus.current * 10;
    } else {
        data.voltage = rules.highestBankVoltage / 10;
        data.current = 0;
    }

    if (rules.moduleHasExternalTempSensor) {
        data.temperature = (int16_t)rules.highestExternalTemp * (int16_t)10;
    } else {
        data.temperature = 0;
    }

    send_canbus_message(0x356, (uint8_t *)&data, sizeof(data356));
}

/**
 * Sends PylonTech BMS identification message (CAN ID: 0x35E).
 */
void pylon_message_35e() {
    uint8_t pylon[] = {0x50, 0x59, 0x4c, 0x4f, 0x4e, 0x20, 0x20, 0x20};
    send_canbus_message(0x35e, (uint8_t *)&pylon, sizeof(pylon));
}

/**
 * Main loop: Sends CAN messages every 5 seconds.
 */
void loop() {
    send_module_status();
    send_corrected_charge_current();
    pylon_message_351();
    pylon_message_355();
    pylon_message_356();
    pylon_message_35e();
    delay(5000);
}
```