

```

1 #include <SPI.h>
2 #include <mcp2515.h>
3
4 // all can frames we will send
5 struct can_frame BmsLimits;
6 struct can_frame BmsSOC;
7 struct can_frame BmsStatus1;
8 struct can_frame BmsErrors;
9 struct can_frame ACCharging;
10
11 // standard can frame for reseaving
12 struct can_frame canMsg;
13
14 MCP2515 mcp2515(10); // enable the can controller
15
16 // some global variables to keep track of the parameters we are sending
17 float current = 0;
18 uint8_t soc = 0;
19 uint16_t voltage = 0xffff;
20 uint16_t maxACCurrent = 0;
21
22 int maxcellbalancing = 0;
23 int maxceltemp = 255;
24 int maxcelmodtemp = 255;
25
26 bool ohshit = false;
27 bool Power_Reduction = false;
28
29 // use millis() to periodically send the can frames (a timer interrupt based
    implemantation would be more elegant for furure improvments, also I am a bit
    worried about what happens when millis() overflows)
30 unsigned long previousMillis = 0;
31 const long interval = 100;
32
33 // setup (runs ones)
34 void setup() {
35     Serial.begin(115200); // start serial monitoring at 115200 baud
36
37     // start the CAN bus
38     mcp2515.reset();
39     mcp2515.setBaudrate(CAN_500KBPS, MCP_8MHZ); // can speeds for both the
    zero_ev ccs controller and emus_bms(when selected 500)
40     mcp2515.setNormalMode();
41
42     Serial.println("Done setting up, starting transmitting can frames");
43 }
44
45 // loop (loops infinit or untill stoped)
46 void loop() {
47     unsigned long currentMillis = millis(); // get the current time
48
49     if (currentMillis - previousMillis >= interval) {
50         previousMillis = currentMillis;
51
52         // charge parameters
53         int current = 80;
54         uint16_t ChargeVoltageLimit = 393 * 10; // 393
55

```

```

56 uint16_t ChargeCurrentLimit = current * 10;
57 uint16_t DischargeVoltageLimit = 307 * 10;
58 uint16_t DischargeCurrentLimit = current * 10;
59
60 if (Power_Reduction) {
61     ChargeCurrentLimit = ChargeCurrentLimit / 2;
62     DischargeCurrentLimit = DischargeCurrentLimit / 2;
63 }
64
65
66 uint16_t HvBatterySOC = soc; // soc sent of CCS_Controller
67
68 uint16_t HvBatteryVoltage = voltage / 10; // battery voltage sent to
CCS_Controller
69 int16_t HvBatteryCurrent = current * 10; // battery current sent to
CCS_Controller
70
71
72 int16_t HvBatteryTemp = 0;
73 if (maxceltemp != 255) { // if no cel sensors are connected (wich they
should be but ok) use the cel module temperature for the CCS_Controller
74     HvBatteryTemp = maxceltemp * 10;
75 } else if (maxcelmodtemp != 255) {
76     HvBatteryTemp = maxcelmodtemp * 10;
77 }
78
79 uint16_t ACLimit = maxACCurrent * 10; // the ac current thet the
CCS_Controller gives to the bms, wich the bms tells the ac charger
80 uint16_t ACvolt = 230 * 10; // we are in europa so 230 is the standard.
81
82
83 // set all the data of the can frames
84 BmsLimits.can_id = 0x351; // 849
85 BmsLimits.can_dlc = 8;
86 BmsLimits.data[0] = (ChargeVoltageLimit & 0xff);
87 BmsLimits.data[1] = (ChargeVoltageLimit >> 8);
88 BmsLimits.data[2] = (ChargeCurrentLimit & 0xff);
89 BmsLimits.data[3] = (ChargeCurrentLimit >> 8);
90 BmsLimits.data[4] = (DischargeVoltageLimit & 0xff);
91 BmsLimits.data[5] = (DischargeVoltageLimit >> 8);
92 BmsLimits.data[6] = (DischargeCurrentLimit & 0xff);
93 BmsLimits.data[7] = (DischargeCurrentLimit >> 8);
94
95 int iserror = 0;
96 if (ohshit) {
97     iserror = 0xff;
98 } else {
99     iserror = 0x00;
100 }
101
102 BmsErrors.can_id = 0x35A; // 858 these are the errors that the
CCS_Controller can exept
103 BmsErrors.can_dlc = 4;
104 BmsErrors.data[0] = iserror;
105 BmsErrors.data[1] = iserror;
106 BmsErrors.data[2] = iserror;
107 BmsErrors.data[3] = iserror;
108
109 BmsSOC.can_id = 0x355; // 853

```

```

110     BmsSOC.can_dlc = 6;
111     BmsSOC.data[0] = (HvBatterySOC & 0xff);
112     BmsSOC.data[1] = (HvBatterySOC >> 8);
113     BmsSOC.data[2] = 0xff; // does not have to be implemented in the case of
the CCS_Controller
114     BmsSOC.data[3] = 0xff; // does not have to be implemented in the case of
the CCS_Controller
115     BmsSOC.data[4] = 0xff; // does not have to be implemented in the case of
the CCS_Controller
116     BmsSOC.data[5] = 0xff; // does not have to be implemented in the case of
the CCS_Controller
117
118     BmsStatus1.can_id = 0x356; // 854
119     BmsStatus1.can_dlc = 6;
120     BmsStatus1.data[0] = (HvBatteryVoltage & 0xff);
121     BmsStatus1.data[1] = (HvBatteryVoltage >> 8);
122     BmsStatus1.data[2] = (HvBatteryCurrent & 0xff);
123     BmsStatus1.data[3] = (HvBatteryCurrent >> 8);
124     BmsStatus1.data[4] = (HvBatteryTemp & 0xff);
125     BmsStatus1.data[5] = (HvBatteryTemp >> 8);
126
127     ACCharging.can_id = 0x19B50407;
128     ACCharging.can_dlc = 8;
129     ACCharging.data[0] = 0xff;
130     ACCharging.data[1] = 0xff;
131     ACCharging.data[2] = (AClimit >> 8);
132     ACCharging.data[3] = (AClimit & 0xff);
133     ACCharging.data[4] = (ACvolt >> 8);
134     ACCharging.data[5] = (ACvolt & 0xff);
135     ACCharging.data[6] = 0xff;
136     ACCharging.data[7] = 0xff;
137
138
139     // send the can frames
140
141     if (voltage != 0xffff) { // only send the frames after the bms is
initilized and we started reseaving data form it
142
143         mcp2515.sendMessage(&BmsLimits); // send the bms limits to the
CCS_Controller
144         mcp2515.sendMessage(&BmsErrors); // send the errors (only sends 0's
for now) to the CCS_Controller
145         mcp2515.sendMessage(&BmsSOC); // send the soc to the CCS_Controller
146         mcp2515.sendMessage(&BmsStatus1); // send the bms status (voltage,
current and temp) to the CCS_Controller
147         mcp2515.sendMessage(&ACCharging); // sent the ac charging limits to the
EMUS_BMS
148
149         // Serial.println("all frames sent, small delay and repeat");
150     }
151     else {
152         Serial.println("not sending can to ccs yet. is emus still booting?");
153     }
154 }
155
156 // read the can bus
157 if (mcp2515.readMessage(&canMsg) == MCP2515::ERROR_OK) {
158     unsigned long id = canMsg.can_id; // the can id, so we can read only the

```

frames we want.

```
159
160     if (id == 874) { // 874 is the id of a can frame that is sent from the
ccs controller with the errors
161         if (canMsg.data[0] & (1 << 0)) {
162             Serial.print("HVPreChargeFault ");
163         }
164         if (canMsg.data[0] & (1 << 1)) {
165             Serial.print("CCSContactorFault ");
166         }
167         if (canMsg.data[0] & (1 << 2)) {
168             Serial.print("HVILFault ");
169         }
170         if (canMsg.data[0] & (1 << 3)) {
171             Serial.print("BMSCommsFault ");
172         }
173         if (canMsg.data[0] & (1 << 4)) {
174             Serial.print("BMSFault ");
175         }
176         if (canMsg.data[0] & (1 << 5)) {
177             Serial.print("CCSECUFault ");
178         }
179         if (canMsg.data[0] & (1 << 6)) {
180             Serial.print("PTCTempFault ");
181         }
182         if (canMsg.data[0] & (1 << 7)) {
183             Serial.print("ChargeProtocolFault ");
184         }
185         if (canMsg.data[1] & (1 << 0)) {
186             Serial.print("IncompatibleCCSCharger ");
187         }
188         if (canMsg.data[1] & (1 << 1)) {
189             Serial.print("ChargeMode ");
190         }
191         if (canMsg.data[1] & (1 << 2)) {
192             Serial.print("PlugPresent ");
193         }
194         if (canMsg.data[1] & (1 << 3)) {
195             Serial.print("InletMotor ");
196         }
197         if (canMsg.data[1] & (1 << 4)) {
198             Serial.print("CCSContactorStatus ");
199         }
200         if (canMsg.data[1] & (1 << 5)) {
201             Serial.print("HVPresent ");
202         }
203         if (canMsg.data[1] & (1 << 6)) {
204             Serial.print("InletFault ");
205         }
206         if (canMsg.data[1] & (1 << 7)) {
207             Serial.print("StopchargeSwitch ");
208         }
209
210         Serial.print("BatteryVoltageSense:");
211         Serial.print(((canMsg.data[2] << 8) | canMsg.data[3]) / 10.0);
212         Serial.print(" CCSVoltageSense:");
213         Serial.print(((canMsg.data[4] << 8) | canMsg.data[5]) / 10.0);
214
215         Serial.println();
```

```

215     Serial.print("//");
216 }
217
218
219 if ( id == 855 ) { // 855 is the id of the can frame that is sent from
the ccs controller with the ac current limit
220     maxACCurrent = canMsg.data[1];
221     //Serial.print("max ac current: "); //maxACCurrent
222     //Serial.println(maxACCurrent);
223 }
224
225
226 if (id == 0x99B50500) { // curent and soc
227     uint8_t current1 = canMsg.data[0];
228     uint8_t current2 = canMsg.data[1];
229     current = (current1 << 8 | current2) / 10.0;
230     Serial.print("amps: ");
231     Serial.println(current);
232     soc = canMsg.data[6];
233     Serial.print("usoc: ");
234     Serial.println(soc);
235 }
236
237
238 if (id == 0x99B50001) { // pack voltage
239     uint8_t voltage1 = canMsg.data[3];
240     uint8_t voltage2 = canMsg.data[4];
241     Serial.print("volt: ");
242     voltage = voltage1 << 8 | voltage2;
243     Serial.println(voltage / 100.0);
244 }
245
246
247 if ( id == 0x99B50003 ) { // r we balencing? howmuch is the moste
248     maxcellbalansing = map(canMsg.data[1], 0, 255, 0, 100);
249     Serial.print("balencing: ");
250     Serial.println(maxcellbalansing);
251 }
252
253
254 if ( id == 0x99B50002 ) { // cel module temp
255     maxcelmodtemp = canMsg.data[1] - 100;
256     Serial.print("max cel module temp: ");
257     Serial.println(maxcelmodtemp);
258 }
259
260
261 if ( id == 0x99B50008 ) { // cel temp
262     maxceltemp = canMsg.data[1] - 100;
263     Serial.print("max cel temp: ");
264     Serial.println(maxceltemp);
265 }
266
267
268 if ( id == 0x99B50000 ) { // info about bms, intresting is power
reduction
269     Serial.println("rx Overall Parameters");
270     if (canMsg.data[0] & (1 << 0)) {
271         Serial.print("Charger_Enable ");

```

```

272     }
273     if (canMsg.data[0] & (1 << 1)) {
274         Serial.print("Heater_Enable ");
275     }
276     if (canMsg.data[0] & (1 << 2)) {
277         Serial.print("Battery_Contactor ");
278     }
279     if (canMsg.data[0] & (1 << 3)) {
280         Serial.print("Battery_Fan ");
281     }
282     if (canMsg.data[0] & (1 << 4)) {
283         Serial.print("Power_Reduction ");
284         Power_Reduction = true;
285     } else {
286         Power_Reduction = false;
287     }
288     if (canMsg.data[0] & (1 << 5)) {
289         Serial.print("Charging_Interlock ");
290     }
291     if (canMsg.data[0] & (1 << 6)) {
292         Serial.print("DCDC_Control ");
293     }
294     if (canMsg.data[0] & (1 << 7)) {
295         Serial.print("Contactor_Pre-Charge ");
296     }
297     if (canMsg.data[0] != 0) {
298         Serial.println();
299     }
300 }
301
302
303 if ( id == 0x99B50007 ) { // emergency states of bms
304
305     Serial.println("rx emergency states");
306     Serial.println(canMsg.data[0], BIN);
307     Serial.println(canMsg.data[1], BIN);
308     Serial.println(canMsg.data[2], BIN);
309     Serial.println(canMsg.data[3], BIN);
310
311     if ((canMsg.data[0] != 0 | canMsg.data[0] != 0b10000000) &
canMsg.data[2] != 0) {
312         ohshit = true;
313     } else {
314         ohshit = false;
315     }
316     if (ohshit) {
317         Serial.println("oh shit");
318     }
319
320     if (canMsg.data[0] & (1 << 0)) {
321         Serial.print("Under-voltage ");
322     }
323     if (canMsg.data[0] & (1 << 1)) {
324         Serial.print("Over-voltage ");
325     }
326     if (canMsg.data[0] & (1 << 2)) {
327         Serial.print("Discharge_Over-current ");
328     }
329     if (canMsg.data[0] & (1 << 3)) {

```

```

329     if (canMsg.data[0] & (1 << 3)) {
330         Serial.print("Charge_Over-current ");
331     }
332     if (canMsg.data[0] & (1 << 4)) {
333         Serial.print("Cell_Module Overheat ");
334     }
335     if (canMsg.data[0] & (1 << 5)) {
336         Serial.print("Leakage ");
337     }
338     if (canMsg.data[0] & (1 << 6)) {
339         Serial.print("No Cell Communication ");
340     }
341
342     if (canMsg.data[2] & (1 << 3)) {
343         Serial.print("Cell_Overheat ");
344     }
345     if (canMsg.data[2] & (1 << 4)) {
346         Serial.print("No_Current_Sensor ");
347     }
348     if (canMsg.data[2] & (1 << 5)) {
349         Serial.print("Pack_Under-Voltage ");
350     }
351
352     if (canMsg.data[0] != 0 & canMsg.data[2] != 0) {
353         Serial.println();
354     }
355 }
356
357
358 }
359 }
360

```