CAN bus protocol for BMS12 modules

Ian Hooper, October 2013

Introduction

This document is intended to assist people integrating the ZEVA BMS12 modules with their own BMS master controller over CAN bus. It describes the required specifications for the CAN bus, packet format and communications protocols.

Standard bus settings

The standard bus speed is 125kbps. Packet format is CAN 2.0A, which uses 11-bit packet IDs. Please contact us if you require modules programmed to suit a different bus speed or packet format.

All packets are the Data frame type, though some require no actual data. Remote frames were not used for request packets since they are not supported by some CAN devices and protocols which may be on the same bus.

Physical Layer

Most ZEVA devices use 5-pin connectors for the CAN bus. Two pins are for the CAN H and CAN L signals, two pins for the 12V supply and Ground, and the fifth pin for transferring shield grounding along the chain of devices (and should be attached to the vehicle chassis at one end of the CAN bus chain). The BMS12 modules power themselves off the CAN bus, so it is recommended that the 12V supply should be capable of supplying up to 20mA per BMS12 module.

CAN bus transceivers often draw around 20mA per node, which can accumulate to a significant quiescent load on the vehicle's 12V auxiliary battery, causing it to go flat if the CAN bus is left powered up while the vehicle is idle for extended periods of time. As such it is recommended that the CAN bus is only enabled when the vehicle is either driving or being charged. (This is handled automatically by all CAN-enabled ZEVA BMS master controllers.)

Molex C-Grid SL plugs are used for products supplied as bare circuit boards (such as BMS12 modules) and aviation-style screwlock plugs for products supplied in cases (such as the EVMS Core).

We recommend using Shielded Twisted Pair (STP) cable, with two pairs of conductor – one pair for CAN H and CAN L, the other for Ground and +12V supply. Conductors should be AWG22 or larger for sufficient mechanical strength and current rating. This type of cable can be difficult to source so you can purchase the cable from us, or from vendors such as Digikey, Mouser, RS Components, or Element14.

CAN buses work best as a single chain of devices, without any branching, and with 120Ω termination resistors at both ends to prevent signal reflection.

Packet Summary

There are nine different packet types used by the BMS12 modules. Five of the packet IDs are for messages from master controller to BMS12 module, and the remaining four are BMS12 module to master controller. Packet IDs are provided in **decimal** format (*not hexadecimal)*.

The module ID selector on the board determines the **Base Packet ID** (BPID), in increments of 10 and starting at 100. For example, module ID 0 will use packet IDs 100 to 108, module ID 1 will use packet IDs 110 to 118, and so on.

Four of the packets from master to BMS12 are simply requests for data and require no data of their own (any sent will just be ignored). The BMS12 module should respond to request packets within about 2mS.

CAN bus was invented as a realtime, low latency bus so the standard only supports small data packets up to 8 bytes in length. This sometimes results in the need for strange bit-packing and numerical scaling schemes in order to fit the required data within the 8-byte limit.

Packet IDs and Structure

BPID + 0: **Request Status** *(Master to BMS12, no data)*

BPID + 1: **Reply Status** *(BMS12 to Master, 5 bytes)*

The Reply Status packet contains single bit status information on any cells which are under-voltage (Cn LV), over-voltage (Cn HV), currently shunt balancing (Cn SH), and whether the two temperature sensors are under (Tn UN) or over (Tn OV) the warning threshold. In each case, a zero represents no error, and 1 represents an error such as a cell beyond its threshold. The Reply Status packet was designed to allow fast polling of all cell statuses within a single CAN packet.

	Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Byte 1	C8 LV	C7 LV	C6 LV	C5 LV	C4 LV	C3 LV	C2 LV	C1 LV
Byte 2	C4 HV	C3 HV	C2 HV	C1 HV	C12 LV	C11 LV	C10 LV	C9 LV
Byte 3	C12 HV	C11 HV	C10 HV	C9 HV	C8 HV	C7 HV	C6 HV	C5 HV
Byte 4	C8 SH	C7 SH	C6 SH	C5 SH	C4 SH	C3 SH	C2 SH	C1 SH
Byte 5	T2 OV	T2 UN	T1 OV	T1 UN	C12 SH	C11 SH	C10 SH	C9 SH

BPID + 2: **Request Voltages 1** *(Master to BMS12, no data)*

BPID + 3: **Reply Voltages 1** *(BMS12 to Master, 8 bytes)*

The Reply Voltages packets allow the master to retrieve actual cell voltages from the BMS12 module. Reply Voltages 1 contains the first (lowest) six voltages, plus temperature #1. Due to size constraints, voltages are in hundredths of a volt, stored in 9-bits. For ease of reassembly, the bottom 8-bits are stored in different bytes for each voltage, then all the ninth bits (Cn B9) are all in the seventh byte. The eighth byte has the temperature in ˚C, with 128 as the zero point to allow +ve and –ve values. (E.g 20˚C will be a value of 148.)

	Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Byte 1	Voltage 1, bits 0-7							
Byte 2	Voltage 2, bits 0-7							
Byte 3	Voltage 3, bits 0-7							
Byte 4	Voltage 4, bits 0-7							

Byte 5	Voltage 5, bits 0-7							
Byte 6	Voltage 6, bits 0-7							
Byte 7	–	–	V6 bit 8	V5 bit 8	V4 bit 8	V3 bit 8	V2 bit 8	V1 bit 8
Byte 8	Temperature 1							

BPID + 4: Request Voltages 2 *(Master to BMS12, no data)*

BPID + 5: Reply Voltages 2 *(BMS12 to Master, 8 bytes)*

Identical to Reply Voltages 1, except it contains the last (highest) six voltages, and temperature 2.

	Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Byte 1	Voltage 7, bits 0-7							
Byte 2	Voltage 8, bits 0-7							
Byte 3	Voltage 9, bits 0-7							
Byte 4	Voltage 10, bits 0-7							
Byte 5	Voltage 11, bits 0-7							
Byte 6	Voltage 12, bits 0-7							
Byte 7	–	–	V12 bit 8	V11 bit 8	V10 bit 8	V9 bit 8	V8 bit 8	V7 bit 8
Byte 8	Temperature 2							

BPID + 6: Request Config *(Master to BMS12, no data)*

BPID + 7: Reply Config *(BMS12 to Master, 8 bytes)*

This packet contains the current high, low and shunt voltage thresholds, plus the thresholds for under- and over-temperature. These values are used to determine the status bits in the Reply Status packet. Voltages are represented as a 16-bit number in millivolts, *not hundredths of a volt like reply voltage packets*, since internally the BMS12 has higher precision than hundredths of a volt. Temperature is in ˚C, with 128 the zero point as seen previously.

	Bit 7	**Bit 6**	**Bit 5**	**Bit 4**	**Bit 3**	**Bit 2**	**Bit 1**	**Bit 0**
Byte 1	Low voltage threshold, low 8 bits							
Byte 2	Low voltage threshold, high 8 bits							
Byte 3	High voltage threshold, low 8 bits							
Byte 4	High voltage threshold, high 8 bits							
Byte 5	Shunt voltage threshold, low 8 bits							
Byte 6	Shunt voltage threshold, high 8 bits							
Byte 7	Under-temperature threshold							
Byte 8	Over-temperature threshold							

BPID + 8: Set Config *(Master to BMS12, 8 bytes)*

The master may send new config data to the BMS12 module. Packet format is identical to Reply Config above, and the BMS12 module will return a Reply Config packet to verify that updated settings were received correctly.

Code examples

To follow are some C code examples to help with programming your master controller to decode data from the BMS12 module. In all cases, assume that data[ ] contains an array of up to 8 unsigned bytes received.

Unpacking Reply Status data

This packet must be decoded bit by bit (no pun intended), and act on any high (error) bits encountered as required. The code below shows how to check if the first 8 cells are under-voltage. Similar code can be used for the final 4 cells, plus the over-voltage and shunt bits.

```
for (int n=0; n<8; n++)
{
     if (byte[0] & (1<<n))
     {
          /* Then cell n+1 is undervoltage */
     }
}
```

Unpacking Reply Voltage data

The following code snippet shows how to extract the voltages and temp from a Reply Voltage packet.

```
for (int n=0; n<6; n++)
{
     voltage[n] = data[n];
     if (data[7] & (1<<n))
          voltage[n] += 256;
}
temp1 = data[8]-128;
```

Unpacking config data

```
lowThreshold = data[0] + data[1]*256;
highThreshold = data[2] + data[3]*256;
shuntThreshold = data[4] + data[5]*256
underTempThreshold = data[6]-128;
overTempThreshold = data[7]-128;
```

Packing config data

```
data[0] = lowThreshold % 256;
data[1] = lowThreshold / 256;
data[2] = highThreshold % 256;
data[3] = highThreshold / 256;
data[4] = shuntThreshold % 256;
data[5] = shuntThreshold / 256;
data[6] = underTempThreshold+128;
data[7] = overTempThreshold+128;
```