Specific firmwares can be developped upon request for special UART commands. (Specific baud rate, broadcast commands or drivers)

KGO-BMS standard firmware settings for UART:

Baud rate 115200
No parity
No flow control

The BMS communicates over UART using packets with the following format:

One Start byte (**value 2** for short packets and 3 for long packets)
One or two bytes specifying the packet length (One if packet payload is smaller or equal to 256 byte, two bytes otherwise and up to 1024 bytes)
The payload of the packet (first byte in the payload is the COMM_PACKET_ID)
Two bytes with a CRC checksum done on the payload
One stop byte (**value = 3**)

//

For external display, user can send request with 1 byte packet payload with content of = 4 or 51 or 52, the BMS will respond accordingly

Example message in decimal sent over UART from display to get the data COMM_GET_VALUES = 4:

Start byte	Packet lenght	Payload[0]	CRC[0]	CRC[1]	Stop Byte
2	1	4	30423	3

Response from the BMS will be something like this:

Start byte	Packet lenght	Payload[0]	Payloadd[1]	...	Payload[42]	CRC[0]	CRC[1]	Stop Byte
2	43	4	packVoltage[0]	...	0	CRC[0]	CRC[1]	3

//

1、One Start byte;
2、One or two bytes specifying the packet length;
3、The payload of the packet (first byte in the payload is the COMM_PACKET_ID);
4、
5、Two bytes with a CRC checksum done on the payload;
6、One stop byte (value = 3);

```
COMM_FW_VERSION = 0,              //0
    COMM_JUMP_TO_BOOTLOADER,      /
    COMM_ERASE_NEW_APP,
    COMM_WRITE_NEW_APP_DATA,
    COMM_GET_VALUES,
    COMM_SET_DUTY,
    COMM_SET_CURRENT,
    COMM_SET_CURRENT_BRAKE,
    COMM_SET_RPM,
    COMM_SET_POS,
    COMM_SET_HANDBRAKE,
    COMM_SET_DETECT,
    COMM_SET_SERVO_POS,
    COMM_SET_MCCONF,
    COMM_GET_MCCONF,
    COMM_GET_MCCONF_DEFAULT,
    COMM_SET_APPCONF,
    COMM_GET_APPCONF,
    COMM_GET_APPCONF_DEFAULT,
    COMM_SAMPLE_PRINT,
```

COMM_TERMINAL_CMD,
COMM_PRINT,
COMM_ROTOR_POSITION,
COMM_EXPERIMENT_SAMPLE,
COMM_DETECT_MOTOR_PARAM,
COMM_DETECT_MOTOR_R_L,
COMM_DETECT_MOTOR_FLUX_LINKAGE,
COMM_DETECT_ENCODER,
COMM_DETECT_HALL_FOC,
COMM_REBOOT,
COMM_ALIVE,
COMM_GET_DECODED_PPM,
COMM_GET_DECODED_ADC,
COMM_GET_DECODED_CHUK,
COMM_FORWARD_CAN,
COMM_SET_CHUCK_DATA,
COMM_CUSTOM_APP_DATA,
COMM_NRF_START_PAIRING,
COMM_GPD_SET_FSW,
COMM_GPD_BUFFER_NOTIFY,
COMM_GPD_BUFFER_SIZE_LEFT,
COMM_GPD_FILL_BUFFER,
COMM_GPD_OUTPUT_SAMPLE,
COMM_GPD_SET_MODE,
COMM_GPD_FILL_BUFFER_INT8,
COMM_GPD_FILL_BUFFER_INT16,
COMM_GPD_SET_BUFFER_INT_SCALE,
COMM_GET_VALUES_SETUP,
COMM_SET_MCCONF_TEMP,
COMM_SET_MCCONF_TEMP_SETUP,
COMM_GET_VALUES_SELECTIVE,
COMM_GET_VALUES_SETUP_SELECTIVE,
COMM_EXT_NRF_PRESENT,
COMM_EXT_NRF_ESB_SET_CH_ADDR,
COMM_EXT_NRF_ESB_SEND_DATA,
COMM_EXT_NRF_ESB_RX_DATA,
COMM_EXT_NRF_SET_ENABLED,
COMM_DETECT_MOTOR_FLUX_LINKAGE_OPENLOOP,
COMM_DETECT_APPLY_ALL_FOC,
COMM_JUMP_TO_BOOTLOADER_ALL_CAN,
COMM_ERASE_NEW_APP_ALL_CAN,
COMM_WRITE_NEW_APP_DATA_ALL_CAN,
COMM_PING_CAN,
COMM_APP_DISABLE_OUTPUT,
COMM_TERMINAL_CMD_SYNC,
COMM_GET_IMU_DATA,
COMM_BM_CONNECT,
COMM_BM_ERASE_FLASH_ALL,
COMM_BM_WRITE_FLASH,
COMM_BM_REBOOT,
COMM_BM_DISCONNECT,
COMM_BM_MAP_PINS_DEFAULT,
COMM_BM_MAP_PINS_NRF5X,
COMM_ERASE_BOOTLOADER,
COMM_ERASE_BOOTLOADER_ALL_CAN,
COMM_PLOT_INIT,
COMM_PLOT_DATA,
COMM_PLOT_ADD_GRAPH,
COMM_PLOT_SET_GRAPH,
COMM_GET_DECODED_BALANCE,
COMM_BM_MEM_READ,

```
        COMM_WRITE_NEW_APP_DATA_LZO,
        COMM_WRITE_NEW_APP_DATA_ALL_CAN_LZO,
        COMM_BM_WRITE_FLASH_LZO,
        COMM_SET_CURRENT_REL,
        COMM_CAN_FWD_FRAME,
        COMM_SET_BATTERY_CUT,
        COMM_SET_BLE_NAME,
        COMM_SET_BLE_PIN,
        COMM_SET_CAN_MODE,
        COMM_GET_IMU_CALIBRATION,
        COMM_GET_MCCONF_TEMP,

        // Custom configuration for hardware
        COMM_GET_CUSTOM_CONFIG_XML,
        COMM_GET_CUSTOM_CONFIG,
        COMM_GET_CUSTOM_CONFIG_DEFAULT,
        COMM_SET_CUSTOM_CONFIG,

        // VESC BMS commands
        COMM_BMS_GET_VALUES,
        COMM_BMS_SET_CHARGE_ALLOWED,
        COMM_BMS_SET_BALANCE_OVERRIDE,
        COMM_BMS_RESET_COUNTERS,
        COMM_BMS_FORCE_BALANCE,
        COMM_BMS_ZERO_CURRENT_OFFSET,

        // FW updates commands for different HW types
        COMM_JUMP_TO_BOOTLOADER_HW,
        COMM_ERASE_NEW_APP_HW,
        COMM_WRITE_NEW_APP_DATA_HW,
        COMM_ERASE_BOOTLOADER_HW,
        COMM_JUMP_TO_BOOTLOADER_ALL_CAN_HW,
        COMM_ERASE_NEW_APP_ALL_CAN_HW,
        COMM_WRITE_NEW_APP_DATA_ALL_CAN_HW,
        COMM_ERASE_BOOTLOADER_ALL_CAN_HW,

        COMM_SET_ODOMETER,

    COMM_EBMS_STORE_CONF = 150,
        COMM_EBMS_GET_CELLS,
        COMM_EBMS_GET_AUX,
        COMM_EBMS_GET_EXP_TEMP,
        COMM_EBMS_SET_MCCONF,
        COMM_EBMS_GET_MCCONF,
        COMM_EBMS_GET_MCCONF_DEFAULT,
        COMM_EBMS_GET_VALUES,
} COMM_PACKET_ID;
```

for example：

```
[14:57:17.949]发送→02 01 00 00 00 03
[14:57:17.976]接收←02 25 00 05 02 4D 4B 42 4D 53 2D 53 53 2D 32 34
    53 2D 4D 49 4E 49 00 26 00 35 00 04 53 57 41 35
    33 33 20 00 00 01 01 D4 89 03

[14:55:42.563]发送→02 01 03 30 63 03
[14:55:42.647]接收←02 02 03 00 55 53 03
```

```
// Private variables
static uint8_t modCommandsSendBuffer[PACKET_MAX_PL_LEN];
static void(*modCommandsSendFunction)(unsigned char *data, unsigned int len) = 0;
bool jumpBootloaderTrue;
modConfigGeneralConfigStructTypedef *modCommandsGeneralConfig;
modConfigGeneralConfigStructTypedef *modCommandsToBeSendConfig;
modConfigGeneralConfigStructTypedef  modCommandsConfigStorage;
modPowerElectronicsPackStateTypedef *modCommandsGeneralState;

void    modCommandsInit(modPowerElectronicsPackStateTypedef          *generalState,modConfigGeneralConfigStructTypedef
*configPointer) {
        modCommandsGeneralConfig = configPointer;
```

```
	modCommandsGeneralState  = generalState;
	jumpBootloaderTrue = false;
}

void modCommandsSetSendFunction(void(*func)(unsigned char *data, unsigned int len)) {
	modCommandsSendFunction = func;
}

void modCommandsSendPacket(unsigned char *data, unsigned int len) {
	if (modCommandsSendFunction) {
		modCommandsSendFunction(data, len);
	}
}

void modCommandsProcessPacket(unsigned char *data, unsigned int len) {
	if (!len) {
		return;
	}

	COMM_PACKET_ID packet_id;
	int32_t ind = 0;
	uint16_t flash_res;
	uint32_t new_app_offset;
	uint32_t delayTick;
	uint8_t cellPointer;
	uint8_t auxPointer;
	uint8_t expPointer;
	uint8_t totalNoOfCells;
	uint8_t totalNoOfAux;

	packet_id = (COMM_PACKET_ID) data[0];
	data++;
	len--;

	switch (packet_id) {
		case COMM_FW_VERSION:
ind = 0;
modCommandsSendBuffer[ind++] = COMM_FW_VERSION;
modCommandsSendBuffer[ind++] = FW_VERSION_MAJOR;
modCommandsSendBuffer[ind++] = FW_VERSION_MINOR;
strcpy((char*)(modCommandsSendBuffer + ind), HW_NAME);
ind += strlen(HW_NAME) + 1;
memcpy(modCommandsSendBuffer + ind, STM32_UUID_8, 12);
ind += 12;
modCommandsSendBuffer[ind++] = 0;
modCommandsSendBuffer[ind++] = FW_TEST_VERSION_NUMBER;

modCommandsSendBuffer[ind++] = HW_TYPE_VESC_BMS;

modCommandsSendBuffer[ind++] = 1; // One custom config

modCommandsSendPacket(modCommandsSendBuffer, ind);
break;
		case COMM_JUMP_TO_BOOTLOADER:
jumpBootloaderTrue = true;
delayTick = HAL_GetTick();
break;
		case COMM_ERASE_NEW_APP:
ind = 0;
flash_res = modFlashEraseNewAppData(libBufferGet_uint32(data, &ind));
```

```
ind = 0;
modCommandsSendBuffer[ind++] = COMM_ERASE_NEW_APP;
modCommandsSendBuffer[ind++] = flash_res == HAL_OK ? true : false;
modCommandsSendPacket(modCommandsSendBuffer, ind);
break;
		case COMM_WRITE_NEW_APP_DATA:
ind = 0;
new_app_offset = libBufferGet_uint32(data, &ind);
flash_res = modFlashWriteNewAppData(new_app_offset, data + ind, len - ind);

ind = 0;
modCommandsSendBuffer[ind++] = COMM_WRITE_NEW_APP_DATA;
modCommandsSendBuffer[ind++] = flash_res == HAL_OK ? 1 : 0;
modCommandsSendPacket(modCommandsSendBuffer, ind);
break;



		case COMM_EBMS_GET_VALUES:
ind = 0;
modCommandsSendBuffer[ind++] = COMM_EBMS_GET_VALUES;

			libBufferAppend_float32(modCommandsSendBuffer, modCommandsGeneralState->packVoltage,1e3,
		&ind);
			libBufferAppend_float32(modCommandsSendBuffer, modCommandsGeneralState->packCurrent,1e3,
		&ind);
libBufferAppend_uint8(modCommandsSendBuffer, (uint8_t)round(modCommandsGeneralState->SoC), &ind);
libBufferAppend_float32(modCommandsSendBuffer, modCommandsGeneralState->cellVoltageHigh, 1e3,
	&ind);
libBufferAppend_float32(modCommandsSendBuffer, modCommandsGeneralState->cellVoltageAverage, 1e3,
	&ind);
			libBufferAppend_float32(modCommandsSendBuffer, modCommandsGeneralState->cellVoltageLow,1e3,
		&ind);
			libBufferAppend_float32(modCommandsSendBuffer,
modCommandsGeneralState->cellVoltageMisMatch, 1e3, &ind);
libBufferAppend_float16(modCommandsSendBuffer, modCommandsGeneralState->loCurrentLoadVoltage, 1e1,
	&ind);
libBufferAppend_float16(modCommandsSendBuffer, modCommandsGeneralState->loCurrentLoadCurrent, 1e1,
	&ind);
libBufferAppend_float16(modCommandsSendBuffer, modCommandsGeneralState->chargerVoltage,1e1, &ind);
libBufferAppend_float16(modCommandsSendBuffer, modCommandsGeneralState->tempBatteryHigh, 1e1,
	&ind);
libBufferAppend_float16(modCommandsSendBuffer, modCommandsGeneralState->tempBatteryAverage, 1e1,
	&ind);
libBufferAppend_float16(modCommandsSendBuffer, modCommandsGeneralState->tempBatteryLow,1e1, &ind);
libBufferAppend_float16(modCommandsSendBuffer, modCommandsGeneralState->tempBMSHigh,1e1, &ind);
libBufferAppend_float16(modCommandsSendBuffer, modCommandsGeneralState->tempBMSAverage,1e1, &ind);
libBufferAppend_float16(modCommandsSendBuffer, modCommandsGeneralState->tempBMSLow,1e1, &ind);
libBufferAppend_float16(modCommandsSendBuffer, modCommandsGeneralState->humidity,1e1, &ind);
libBufferAppend_uint8(modCommandsSendBuffer, (uint8_t)modCommandsGeneralState->operationalState,&ind);
libBufferAppend_uint8(modCommandsSendBuffer, (uint8_t)modCommandsGeneralState->chargeBalanceActive,&ind);
libBufferAppend_uint8(modCommandsSendBuffer, (uint8_t)modCommandsGeneralState->faultState, &ind);

modCommandsSendBuffer[ind++] = modCommandsGeneralConfig->CANID;
modCommandsSendPacket(modCommandsSendBuffer, ind);

break;
		case COMM_EBMS_GET_CELLS:
ind = 0;
modCommandsSendBuffer[ind++] = COMM_EBMS_GET_CELLS;
```

```
libBufferAppend_uint8(modCommandsSendBuffer,
modCommandsGeneralConfig->noOfCellsSeries*modCommandsGeneralConfig->noOfParallelModules, &ind);
for(cellPointer = 0; cellPointer <
modCommandsGeneralConfig->noOfCellsSeries*modCommandsGeneralConfig->noOfParallelModules; cellPointer++){
	if(modCommandsGeneralState->cellVoltagesIndividual[cellPointer].cellBleedActive)
		libBufferAppend_float16(modCommandsSendBuffer,
modCommandsGeneralState->cellVoltagesIndividual[cellPointer].cellVoltage*-1.0f, 1e3, &ind);
	else
		libBufferAppend_float16(modCommandsSendBuffer,
modCommandsGeneralState->cellVoltagesIndividual[cellPointer].cellVoltage, 1e3, &ind);
}

modCommandsSendBuffer[ind++] = modCommandsGeneralConfig->CANID;
modCommandsSendPacket(modCommandsSendBuffer, ind);
break;
		case COMM_EBMS_GET_AUX:
ind = 0;
modCommandsSendBuffer[ind++] = COMM_EBMS_GET_AUX;

libBufferAppend_uint8(modCommandsSendBuffer,
modCommandsGeneralConfig->cellMonitorICCount*modCommandsGeneralConfig->noOfTempSensorPerModule, &ind);
for(auxPointer = 0; auxPointer <
modCommandsGeneralConfig->cellMonitorICCount*modCommandsGeneralConfig->noOfTempSensorPerModule;
auxPointer++){
		libBufferAppend_float16(modCommandsSendBuffer,
modCommandsGeneralState->auxVoltagesIndividual[auxPointer].auxVoltage, 1e1, &ind);
}

modCommandsSendBuffer[ind++] = modCommandsGeneralConfig->CANID;
modCommandsSendPacket(modCommandsSendBuffer, ind);
break;
		case COMM_EBMS_GET_EXP_TEMP:
ind = 0;
modCommandsSendBuffer[ind++] = COMM_EBMS_GET_EXP_TEMP;

libBufferAppend_uint8(modCommandsSendBuffer,
modCommandsGeneralConfig->noOfExpansionBoard*modCommandsGeneralConfig->noOfTempSensorPerExpansionBoard,
&ind);
for(expPointer = 0; expPointer <
modCommandsGeneralConfig->noOfExpansionBoard*modCommandsGeneralConfig->noOfTempSensorPerExpansionBoard;
expPointer++){
		libBufferAppend_float16(modCommandsSendBuffer,
modCommandsGeneralState->expVoltagesIndividual[expPointer].expVoltage, 1e1, &ind);
}

modCommandsSendBuffer[ind++] = modCommandsGeneralConfig->CANID;
modCommandsSendPacket(modCommandsSendBuffer, ind);
break;
case COMM_EBMS_SET_MCCONF:
ind = 0;
modCommandsGeneralConfig->noOfCellsSeries       = libBufferGet_uint8(data,&ind);                // 1
modCommandsGeneralConfig->noOfCellsParallel     = libBufferGet_uint8(data,&ind);                // 1
modCommandsGeneralConfig->noOfParallelModules= libBufferGet_uint8(data,&ind);                // 1
modCommandsGeneralConfig->batteryCapacity= libBufferGet_float32_auto(data,&ind);             // 4
modCommandsGeneralConfig->cellHardUnderVoltage= libBufferGet_float32_auto(data,&ind);                // 4
modCommandsGeneralConfig->cellHardOverVoltage= libBufferGet_float32_auto(data,&ind);         // 4
modCommandsGeneralConfig->cellLCSoftUnderVoltage= libBufferGet_float32_auto(data,&ind);              // 4
modCommandsGeneralConfig->cellSoftOverVoltage= libBufferGet_float32_auto(data,&ind);         // 4
modCommandsGeneralConfig->cellBalanceDifferenceThreshold= libBufferGet_float32_auto(data,&ind);              // 4
modCommandsGeneralConfig->cellBalanceStart= libBufferGet_float32_auto(data,&ind);            // 4
modCommandsGeneralConfig->cellBalanceAllTime= libBufferGet_uint8(data,&ind);            // 1
```

```
modCommandsGeneralConfig->cellThrottleUpperStart          = libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->cellThrottleLowerStart          = libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->cellThrottleUpperMargin= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->cellThrottleLowerMargin= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->packVoltageDataSource= libBufferGet_uint8(data,&ind);                  // 1
modCommandsGeneralConfig->packCurrentDataSource= libBufferGet_uint8(data,&ind);                  // 1
modCommandsGeneralConfig->buzzerSignalSource= libBufferGet_uint8(data,&ind);             // 1
modCommandsGeneralConfig->buzzerSignalPersistant          = libBufferGet_uint8(data,&ind);                  // 1
modCommandsGeneralConfig->shuntLCFactor= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->voltageLCFactor= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->voltageLCOffset= libBufferGet_int16(data,&ind);                  // 2
modCommandsGeneralConfig->loadVoltageFactor= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->loadVoltageOffset      = libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->chargerVoltageFactor= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->chargerVoltageOffset= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->throttleChargeIncreaseRate   = libBufferGet_uint8(data,&ind);                  // 1
modCommandsGeneralConfig->throttleDisChargeIncreaseRate= libBufferGet_uint8(data,&ind);                  // 1
modCommandsGeneralConfig->cellBalanceUpdateInterval= libBufferGet_uint32(data,&ind);             // 4
modCommandsGeneralConfig->maxSimultaneousDischargingCells= libBufferGet_uint8(data,&ind);             // 1
modCommandsGeneralConfig->timeoutDischargeRetry= libBufferGet_uint32(data,&ind);                  // 4
modCommandsGeneralConfig->hysteresisDischarge= libBufferGet_float32_auto(data,&ind);              // 4
modCommandsGeneralConfig->timeoutChargeRetry= libBufferGet_uint32(data,&ind);                  // 4
modCommandsGeneralConfig->hysteresisCharge= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->timeoutChargeCompleted= libBufferGet_uint32(data,&ind);                  // 4
modCommandsGeneralConfig->timeoutChargingCompletedMinimalMismatch          = libBufferGet_uint32(data,&ind);
        // 4
modCommandsGeneralConfig->maxMismatchThreshold= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->chargerEnabledThreshold= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->timeoutChargerDisconnected= libBufferGet_uint32(data,&ind);                  // 4
modCommandsGeneralConfig->minimalPrechargePercentage= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->timeoutLCPreCharge= libBufferGet_uint32(data,&ind);                  // 4
modCommandsGeneralConfig->maxAllowedCurrent= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->allowedTempBattDischargingMax= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->allowedTempBattDischargingMin= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->allowedTempBattChargingMax=libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->allowedTempBattChargingMin= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->allowedTempBattCoolingMax= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->allowedTempBattCoolingMin= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->allowedTempBMSMax= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->allowedTempBMSMin= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->displayTimeoutBatteryDead= libBufferGet_uint32(data,&ind);                  // 4
modCommandsGeneralConfig->displayTimeoutBatteryError= libBufferGet_uint32(data,&ind);             // 4
modCommandsGeneralConfig->displayTimeoutBatteryErrorPreCharge                = libBufferGet_uint32(data,&ind);
        // 4
modCommandsGeneralConfig->displayTimeoutSplashScreen= libBufferGet_uint32(data,&ind);                  // 4
modCommandsGeneralConfig->displayStyle          = libBufferGet_uint8(data,&ind);                  // 1
modCommandsGeneralConfig->maxUnderAndOverVoltageErrorCount                = libBufferGet_uint8(data,&ind);
        // 1
modCommandsGeneralConfig->maxUnderAndOverTemperatureErrorCount          = libBufferGet_uint8(data,&ind);
        // 1
modCommandsGeneralConfig->notUsedCurrentThreshold= libBufferGet_float32_auto(data,&ind);                  // 4
modCommandsGeneralConfig->notUsedTimeout       = libBufferGet_uint32(data,&ind);                  // 4
modCommandsGeneralConfig->stateOfChargeStoreInterval= libBufferGet_uint32(data,&ind);                  // 4
modCommandsGeneralConfig->stateOfChargeMethod          = libBufferGet_uint8(data,&ind);                  // 1
modCommandsGeneralConfig->CANID= libBufferGet_uint8(data,&ind);                  // 1
modCommandsGeneralConfig->CANIDStyle          = libBufferGet_uint8(data,&ind);                  // 1
modCommandsGeneralConfig->canBusSpeed                = libBufferGet_uint8(data,&ind);                  // 1
modCommandsGeneralConfig->emitStatusOverCAN  = libBufferGet_uint8(data,&ind);                  // 1
modCommandsGeneralConfig->emitStatusProtocol    = libBufferGet_uint8(data,&ind);                  // 1
modCommandsGeneralConfig->tempEnableMaskBMS          = libBufferGet_uint32(data,&ind);                  // 4
modCommandsGeneralConfig->tempEnableMaskBattery       = libBufferGet_uint32(data,&ind);                  // 4
```

```
modCommandsGeneralConfig->tempEnableMaskExpansion= libBufferGet_uint32(data,&ind);                    // 4
modCommandsGeneralConfig->noOfTempSensorPerModule= libBufferGet_uint8(data,&ind);          // 1
modCommandsGeneralConfig->noOfExpansionBoard= libBufferGet_uint8(data,&ind);               // 1
modCommandsGeneralConfig->noOfTempSensorPerExpansionBoard                  = libBufferGet_uint8(data,&ind);
        // 1
modCommandsGeneralConfig->LCUseDischarge       = libBufferGet_uint8(data,&ind);            // 1
modCommandsGeneralConfig->LCUsePrecharge       = libBufferGet_uint8(data,&ind);            // 1
modCommandsGeneralConfig->allowChargingDuringDischarge= libBufferGet_uint8(data,&ind);             // 1
modCommandsGeneralConfig->allowForceOn                  = libBufferGet_uint8(data,&ind);           // 1
modCommandsGeneralConfig->pulseToggleButton    = libBufferGet_uint8(data,&ind);            // 1
modCommandsGeneralConfig->useCANSafetyInput   = libBufferGet_uint8(data,&ind);             // 1
modCommandsGeneralConfig->useCANDelayedPowerDown= libBufferGet_uint8(data,&ind);                   // 1
modCommandsGeneralConfig->NTCTopResistor[modConfigNTCGroupLTCExt]       = libBufferGet_uint32(data,&ind);
        // 4
modCommandsGeneralConfig->NTC25DegResistance[modConfigNTCGroupLTCExt]          =    libBufferGet_uint32(data,&ind);
                // 4
modCommandsGeneralConfig->NTCBetaFactor[modConfigNTCGroupLTCExt]        = libBufferGet_uint16(data,&ind);
        // 2
modCommandsGeneralConfig->NTCTopResistor[modConfigNTCGroupMasterPCB]  = libBufferGet_uint32(data,&ind);
        // 4
modCommandsGeneralConfig->NTC25DegResistance[modConfigNTCGroupMasterPCB]= libBufferGet_uint32(data,&ind);
        // 4
modCommandsGeneralConfig->NTCBetaFactor[modConfigNTCGroupMasterPCB]   = libBufferGet_uint16(data,&ind);
        // 2
modCommandsGeneralConfig->NTCTopResistor[modConfigNTCGroupExp]          = libBufferGet_uint32(data,&ind);
        // 4
modCommandsGeneralConfig->NTC25DegResistance[modConfigNTCGroupExp]    = libBufferGet_uint32(data,&ind);
        // 4
modCommandsGeneralConfig->NTCBetaFactor[modConfigNTCGroupExp]          = libBufferGet_uint16(data,&ind);
        // 2
modCommandsGeneralConfig->cellMonitorType       = libBufferGet_uint8(data,&ind);            // 1
modCommandsGeneralConfig->cellMonitorICCount    = libBufferGet_uint8(data,&ind);            // 1
modCommandsGeneralConfig->externalEnableOperationalState                = libBufferGet_uint8(data,&ind);                    // 1
modCommandsGeneralConfig->chargeEnableOperationalState= libBufferGet_uint8(data,&ind);             // 1
modCommandsGeneralConfig->powerDownDelay      = libBufferGet_uint32(data,&ind);            // 4
modCommandsGeneralConfig->humidityICType        = libBufferGet_uint8(data,&ind);            // 1
modCommandsGeneralConfig->buzzerThreshold = libBufferGet_uint8(data,&ind); // 1

ind = 0;
modCommandsSendBuffer[ind++] = packet_id;
modCommandsSendPacket(modCommandsSendBuffer, ind);

modconfigHardwareLimitsApply(modCommandsGeneralConfig);

break;



                case COMM_EBMS_GET_MCCONF:



                case COMM_EBMS_GET_MCCONF_DEFAULT:


  if(packet_id == COMM_EBMS_GET_MCCONF_DEFAULT){
        modConfigLoadDefaultConfig(&modCommandsConfigStorage);
        modCommandsToBeSendConfig = &modCommandsConfigStorage;
    }
    Else
```

```
    {
        modCommandsToBeSendConfig = modCommandsGeneralConfig;
    }

   ind = 0;
modCommandsSendBuffer[ind++] = packet_id;




libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->noOfCellsSeries                ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->noOfCellsParallel        ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->noOfParallelModules   ,&ind); // 1
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->batteryCapacity
,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->cellHardUnderVoltage ,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->cellHardOverVoltage   ,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->cellLCSoftUnderVoltage
,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->cellSoftOverVoltage    ,&ind); // 4
libBufferAppend_float32_auto(
modCommandsSendBuffer,modCommandsToBeSendConfig->cellBalanceDifferenceThreshold,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->cellBalanceStart
,&ind); // 4
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->cellBalanceAllTime      ,&ind); // 1
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->cellThrottleUpperStart ,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->cellThrottleLowerStart ,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->cellThrottleUpperMargin
,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->cellThrottleLowerMargin
,&ind); // 4
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->packVoltageDataSource          ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->packCurrentDataSource          ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->buzzerSignalSource     ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->buzzerSignalPersistant          ,&ind); // 1
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->shuntLCFactor
,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->voltageLCFactor
,&ind); // 4
libBufferAppend_int16(modCommandsSendBuffer,modCommandsToBeSendConfig->voltageLCOffset                ,&ind); // 2
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->loadVoltageFactor       ,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->loadVoltageOffset       ,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->chargerVoltageFactor ,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->chargerVoltageOffset  ,&ind); // 4
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->throttleChargeIncreaseRate,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->throttleDisChargeIncreaseRate,&ind); // 1
libBufferAppend_uint32(      modCommandsSendBuffer,modCommandsToBeSendConfig->cellBalanceUpdateInterval,&ind); // 4
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->maxSimultaneousDischargingCells,&ind);
// 1
libBufferAppend_uint32(      modCommandsSendBuffer,modCommandsToBeSendConfig->timeoutDischargeRetry   ,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->hysteresisDischarge   ,&ind); // 4
libBufferAppend_uint32(      modCommandsSendBuffer,modCommandsToBeSendConfig->timeoutChargeRetry       ,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->hysteresisCharge
,&ind); // 4
libBufferAppend_uint32(     modCommandsSendBuffer,modCommandsToBeSendConfig->timeoutChargeCompleted
,&ind); // 4
libBufferAppend_uint32(modCommandsSendBuffer,modCommandsToBeSendConfig->timeoutChargingCompletedMinimalMism
atch,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->maxMismatchThreshold
,&ind); // 4
```

```
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->chargerEnabledThreshold
,&ind); // 4
libBufferAppend_uint32(     modCommandsSendBuffer,modCommandsToBeSendConfig->timeoutChargerDisconnected,&ind); //
4
libBufferAppend_float32_auto(
modCommandsSendBuffer,modCommandsToBeSendConfig->minimalPrechargePercentage,&ind); // 4
libBufferAppend_uint32(     modCommandsSendBuffer,modCommandsToBeSendConfig->timeoutLCPreCharge        ,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->maxAllowedCurrent     ,&ind); // 4
libBufferAppend_float32_auto(
modCommandsSendBuffer,modCommandsToBeSendConfig->allowedTempBattDischargingMax,&ind); // 4
libBufferAppend_float32_auto(
modCommandsSendBuffer,modCommandsToBeSendConfig->allowedTempBattDischargingMin,&ind); // 4
libBufferAppend_float32_auto(
modCommandsSendBuffer,modCommandsToBeSendConfig->allowedTempBattChargingMax,&ind); // 4
libBufferAppend_float32_auto(
modCommandsSendBuffer,modCommandsToBeSendConfig->allowedTempBattChargingMin,&ind); // 4
libBufferAppend_float32_auto(
modCommandsSendBuffer,modCommandsToBeSendConfig->allowedTempBattCoolingMax,&ind); // 4
libBufferAppend_float32_auto(
modCommandsSendBuffer,modCommandsToBeSendConfig->allowedTempBattCoolingMin,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->allowedTempBMSMax,&ind); // 4
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->allowedTempBMSMin ,&ind); // 4
libBufferAppend_uint32(    modCommandsSendBuffer,modCommandsToBeSendConfig->displayTimeoutBatteryDead,&ind); // 4
libBufferAppend_uint32(        modCommandsSendBuffer,modCommandsToBeSendConfig->displayTimeoutBatteryError,&ind); //
4
libBufferAppend_uint32(modCommandsSendBuffer,modCommandsToBeSendConfig->displayTimeoutBatteryErrorPreCharge
        ,&ind); // 4
libBufferAppend_uint32(modCommandsSendBuffer,modCommandsToBeSendConfig->displayTimeoutSplashScreen,&ind); // 4
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->displayStyle                ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->maxUnderAndOverVoltageErrorCount,&in
d); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->maxUnderAndOverTemperatureErrorCoun
t,&ind); // 1
libBufferAppend_float32_auto( modCommandsSendBuffer,modCommandsToBeSendConfig->notUsedCurrentThreshold
,&ind); // 4
libBufferAppend_uint32(      modCommandsSendBuffer,modCommandsToBeSendConfig->notUsedTimeout            ,&ind); // 4
libBufferAppend_uint32(        modCommandsSendBuffer,modCommandsToBeSendConfig->stateOfChargeStoreInterval,&ind); //
4
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->stateOfChargeMethod ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->CANID,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->CANIDStyle              ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->canBusSpeed             ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->emitStatusOverCAN       ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->emitStatusProtocol      ,&ind); // 1
libBufferAppend_uint32(      modCommandsSendBuffer,modCommandsToBeSendConfig->tempEnableMaskBMS    ,&ind); // 4
libBufferAppend_uint32(      modCommandsSendBuffer,modCommandsToBeSendConfig->tempEnableMaskBattery ,&ind); // 4
libBufferAppend_uint32(    modCommandsSendBuffer,modCommandsToBeSendConfig->tempEnableMaskExpansion
,&ind); // 4
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->noOfTempSensorPerModule   ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->noOfExpansionBoard  ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->noOfTempSensorPerExpansionBoard,&in
d); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->LCUseDischarge              ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->LCUsePrecharge              ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->allowChargingDuringDischarge,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->allowForceOn            ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->pulseToggleButton       ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->useCANSafetyInput       ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->useCANDelayedPowerDown   ,&ind); // 1
libBufferAppend_uint32(modCommandsSendBuffer,modCommandsToBeSendConfig->NTCTopResistor[modConfigNTCGroupL
TCExt],&ind); // 4
```

```
libBufferAppend_uint32(modCommandsSendBuffer,modCommandsToBeSendConfig->NTC25DegResistance[modConfigNTCGr
oupLTCExt]      ,&ind); // 4
libBufferAppend_uint16(modCommandsSendBuffer,modCommandsToBeSendConfig->NTCBetaFactor[modConfigNTCGroupLT
CExt],&ind); // 2
libBufferAppend_uint32(modCommandsSendBuffer,modCommandsToBeSendConfig->NTCTopResistor[modConfigNTCGroupM
asterPCB]       ,&ind); // 4
libBufferAppend_uint32(
modCommandsSendBuffer,modCommandsToBeSendConfig->NTC25DegResistance[modConfigNTCGroupMasterPCB]
,&ind); // 4
libBufferAppend_uint16(
modCommandsSendBuffer,modCommandsToBeSendConfig->NTCBetaFactor[modConfigNTCGroupMasterPCB]     ,&ind); // 2
libBufferAppend_uint32(
modCommandsSendBuffer,modCommandsToBeSendConfig->NTCTopResistor[modConfigNTCGroupExp]               ,&ind); // 4
libBufferAppend_uint32(
modCommandsSendBuffer,modCommandsToBeSendConfig->NTC25DegResistance[modConfigNTCGroupExp]
,&ind); // 4
libBufferAppend_uint16(
modCommandsSendBuffer,modCommandsToBeSendConfig->NTCBetaFactor[modConfigNTCGroupExp]               ,&ind); // 2
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->cellMonitorType                ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->cellMonitorICCount     ,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->externalEnableOperationalState,&ind); // 1
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->chargeEnableOperationalState,&ind);  //  1

libBufferAppend_uint32(      modCommandsSendBuffer,modCommandsToBeSendConfig->powerDownDelay
,&ind); // 4
libBufferAppend_uint8(modCommandsSendBuffer,modCommandsToBeSendConfig->humidityICType                ,&ind); // 1
libBufferAppend_uint8( modCommandsSendBuffer,modCommandsToBeSendConfig->buzzerThreshold ,&ind); // 1

modCommandsSendPacket(modCommandsSendBuffer, ind);
break;









                case COMM_TERMINAL_CMD:
data[len] = '\0';
modTerminalProcessString((char*)data);
break;
                case COMM_REBOOT:
modCommandsJumpToMainApplication();
break;
                case COMM_ALIVE:
break;
                case COMM_FORWARD_CAN:
modCANSendBuffer(data[0], data + 1, len - 1, false);
break;
                case COMM_EBMS_STORE_CONF:
modConfigStoreConfig();

ind = 0;
modCommandsSendBuffer[ind++] = packet_id;
```

```
modCommandsSendPacket(modCommandsSendBuffer, ind);
break;
		//Specific to VESC tool app
		case COMM_BMS_GET_VALUES:
ind = 0;

modCommandsSendBuffer[ind++] = COMM_BMS_GET_VALUES;

			libBufferAppend_float32(modCommandsSendBuffer,	modCommandsGeneralState->packVoltage,1e6,
		&ind);
			libBufferAppend_float32(modCommandsSendBuffer, modCommandsGeneralState->chargerVoltage,1e6,
		&ind);
			libBufferAppend_float32(modCommandsSendBuffer,	modCommandsGeneralState->packCurrent,1e6,
		&ind);
			libBufferAppend_float32(modCommandsSendBuffer,	modCommandsGeneralState->packCurrent,1e6,
		&ind);
			libBufferAppend_float32(modCommandsSendBuffer,	modCommandsGeneralState->packCurrent,1e3,
		&ind); //TO DO: define AhCounter instead of packCurrent
			libBufferAppend_float32(modCommandsSendBuffer,	modCommandsGeneralState->packVoltage,1e3,
		&ind); //TO DO: define WhCounter instead of packCurrent

// Cell voltages
totalNoOfCells = modCommandsGeneralConfig->noOfCellsSeries*modCommandsGeneralConfig->noOfParallelModules;
modCommandsSendBuffer[ind++] = totalNoOfCells;
for (cellPointer = 0; cellPointer < totalNoOfCells; cellPointer++) {
	libBufferAppend_float16(modCommandsSendBuffer,
modCommandsGeneralState->cellVoltagesIndividual[cellPointer].cellVoltage, 1e3, &ind);
}

// Balancing state
for (int i = 0; i < totalNoOfCells; i++) {
	modCommandsSendBuffer[ind++] = modCommandsGeneralState->cellVoltagesIndividual[i].cellBleedActive;
}

// Temperatures
totalNoOfAux                                                                                                    =
modCommandsGeneralConfig->cellMonitorICCount*modCommandsGeneralConfig->noOfTempSensorPerModule;
modCommandsSendBuffer[ind++] = totalNoOfAux;
for (auxPointer = 0; auxPointer < totalNoOfAux; auxPointer++) {
	libBufferAppend_float16(modCommandsSendBuffer,
modCommandsGeneralState->auxVoltagesIndividual[auxPointer].auxVoltage, 1e2, &ind);
}
libBufferAppend_float16(modCommandsSendBuffer, modCommandsGeneralState->temperatures[0], 1e2, &ind);

// Humidity
libBufferAppend_float16(modCommandsSendBuffer, modCommandsGeneralState->humidity, 1e2, &ind);
libBufferAppend_float16(modCommandsSendBuffer, modCommandsGeneralState->temperatures[1], 1e2, &ind);

// Highest cell temperature
libBufferAppend_float16(modCommandsSendBuffer, modCommandsGeneralState->tempBatteryHigh, 1e2, &ind);

// State of charge and state of health
libBufferAppend_float16(modCommandsSendBuffer, modCommandsGeneralState->SoC/100, 1e3, &ind);
libBufferAppend_float16(modCommandsSendBuffer, 0.0, 1e3, &ind);

modCommandsSendPacket(modCommandsSendBuffer, ind);
break;

		case COMM_BMS_SET_CHARGE_ALLOWED:
modCommandsGeneralState->chargeAllowed = true;
break;
```

```
                case COMM_BMS_SET_BALANCE_OVERRIDE:
//modCommandsGeneralConfig->cellBalanceAllTime = true;
break;

                case COMM_BMS_RESET_COUNTERS:
if (data[0]) {
//To do
}

if (data[1]) {
//To do
}
break;

                case COMM_BMS_FORCE_BALANCE:
modCommandsGeneralConfig->cellBalanceAllTime = true;
break;
                case COMM_BMS_ZERO_CURRENT_OFFSET:
modPowerElectronicsResetCurrentOffset();
                break;
                case COMM_GET_CUSTOM_CONFIG:
                case COMM_GET_CUSTOM_CONFIG_DEFAULT: {
main_config_t *conf = libMempools_alloc_conf();

int conf_ind = data[0];

if (conf_ind != 0) {
        break;
}

if (packet_id == COMM_GET_CUSTOM_CONFIG) {
        modCommandsEBMSToVESC(conf);
} else {
        confparser_set_defaults_main_config_t(conf);
}

ind = 0;
modCommandsSendBuffer[ind++] = packet_id;
modCommandsSendBuffer[ind++] = conf_ind;
int32_t len = confparser_serialize_main_config_t(modCommandsSendBuffer + ind, conf);
modCommandsSendPacket(modCommandsSendBuffer, len + ind);
libMempools_free_conf(conf);
                } break;

                case COMM_SET_CUSTOM_CONFIG: {
main_config_t *conf = libMempools_alloc_conf();

int conf_ind = data[0];

if (conf_ind == 0 && confparser_deserialize_main_config_t(data + 1, conf)) {
        modCommandsVESCToEBMS(conf);

        ind = 0;
        //modCommandsSendBuffer[50];
        modCommandsSendBuffer[ind++] = packet_id;
        modCommandsSendPacket(modCommandsSendBuffer, ind);
} else {
        modCommandsPrintf("Warning: Could not set configuration");
}
```

```
libMempools_free_conf(conf);
                } break;

                case COMM_GET_CUSTOM_CONFIG_XML: {
ind = 0;

int conf_ind = data[ind++];

if (conf_ind != 0) {
        break;
}

int32_t len_conf = libBufferGet_int32(data, &ind);
int32_t ofs_conf = libBufferGet_int32(data, &ind);

if ((len_conf + ofs_conf) > DATA_MAIN_CONFIG_T__SIZE || len_conf > (PACKET_MAX_PL_LEN - 10)) {
break;
}

ind = 0;
modCommandsSendBuffer[ind++] = packet_id;
modCommandsSendBuffer[ind++] = conf_ind;
libBufferAppend_int32(modCommandsSendBuffer, DATA_MAIN_CONFIG_T__SIZE, &ind);
libBufferAppend_int32(modCommandsSendBuffer, ofs_conf, &ind);
memcpy(modCommandsSendBuffer + ind, data_main_config_t_ + ofs_conf, len_conf);
ind += len_conf;
modCommandsSendPacket(modCommandsSendBuffer, ind);


                } break;
                default:
break;
        }

        if(modDelayTick1ms(&delayTick,1000) && jumpBootloaderTrue)
                modFlashJumpToBootloader();
}

void modCommandsPrintf(const char* format, ...) {
        va_list arg;
        va_start (arg, format);
        int len;
        static char print_buffer[255];

        print_buffer[0] = COMM_PRINT;
        len = vsnprintf(print_buffer+1, 254, format, arg);
        va_end (arg);

        if(len > 0) {
                modCommandsSendPacket((unsigned char*)print_buffer, (len<254)? len+1: 255);
        }
}

void modCommandsVESCToEBMS(main_config_t *conf) {

        modCommandsGeneralConfig->CANID          = conf->controller_id;
        modCommandsGeneralConfig->canBusSpeed = conf->can_baud_rate;
        modCommandsGeneralConfig->noOfCellsSeries = conf->cell_num;
        modCommandsGeneralConfig->noOfTempSensorPerModule = conf->temp_num;
        modCommandsGeneralConfig->cellBalanceStart = conf->balance_start_voltage;
        modCommandsGeneralConfig->maxMismatchThreshold = conf->balance_difference_threshold;
```

```
	modCommandsGeneralConfig->cellSoftOverVoltage = conf->soft_overvoltage;
	modCommandsGeneralConfig->cellLCSoftUnderVoltage = conf->soft_undervoltage;
	modCommandsGeneralConfig->cellHardOverVoltage = conf->hard_overvoltage;
	modCommandsGeneralConfig->cellHardUnderVoltage = conf->hard_undervoltage;
	modCommandsGeneralConfig->allowedTempBattChargingMax = conf->t_charge_max;
	modCommandsGeneralConfig->allowedTempBattDischargingMax = conf->t_discharge_max;
	modCommandsGeneralConfig->notUsedCurrentThreshold = conf->not_used_current_threshold;
	modCommandsGeneralConfig->notUsedTimeout = conf->not_used_timeout;

}

void modCommandsEBMSToVESC(main_config_t *conf) {
	conf->controller_id = modCommandsGeneralConfig->CANID;
	conf->can_baud_rate = modCommandsGeneralConfig->canBusSpeed;
	conf->cell_num = modCommandsGeneralConfig->noOfCellsSeries;
	conf->temp_num = modCommandsGeneralConfig->noOfTempSensorPerModule;
	conf->balance_start_voltage = modCommandsGeneralConfig->cellBalanceStart;
	conf->balance_difference_threshold = modCommandsGeneralConfig->maxMismatchThreshold;
	conf->soft_overvoltage = modCommandsGeneralConfig->cellSoftOverVoltage;
	conf->soft_undervoltage = modCommandsGeneralConfig->cellLCSoftUnderVoltage;
	conf->hard_overvoltage = modCommandsGeneralConfig->cellHardOverVoltage;
	conf->hard_undervoltage = modCommandsGeneralConfig->cellHardUnderVoltage;
	conf->t_charge_max = modCommandsGeneralConfig->allowedTempBattChargingMax;
	conf->t_discharge_max = modCommandsGeneralConfig->allowedTempBattDischargingMax;
	conf->not_used_current_threshold = modCommandsGeneralConfig->notUsedCurrentThreshold;
	conf->not_used_timeout = modCommandsGeneralConfig->notUsedTimeout;
}

void modCommandsJumpToMainApplication(void) {
	NVIC_SystemReset();
}




//Below is the code used for UART crc checksum calculation


const unsigned short libCRCLookupTable[] = { 0x0000, 0x1021, 0x2042, 0x3063, 0x4084,
		0x50a5, 0x60c6, 0x70e7, 0x8108, 0x9129, 0xa14a, 0xb16b, 0xc18c, 0xd1ad,
		0xe1ce, 0xf1ef, 0x1231, 0x0210, 0x3273, 0x2252, 0x52b5, 0x4294, 0x72f7,
		0x62d6, 0x9339, 0x8318, 0xb37b, 0xa35a, 0xd3bd, 0xc39c, 0xf3ff, 0xe3de,
		0x2462, 0x3443, 0x0420, 0x1401, 0x64e6, 0x74c7, 0x44a4, 0x5485, 0xa56a,
		0xb54b, 0x8528, 0x9509, 0xe5ee, 0xf5cf, 0xc5ac, 0xd58d, 0x3653, 0x2672,
		0x1611, 0x0630, 0x76d7, 0x66f6, 0x5695, 0x46b4, 0xb75b, 0xa77a, 0x9719,
		0x8738, 0xf7df, 0xe7fe, 0xd79d, 0xc7bc, 0x48c4, 0x58e5, 0x6886, 0x78a7,
		0x0840, 0x1861, 0x2802, 0x3823, 0xc9cc, 0xd9ed, 0xe98e, 0xf9af, 0x8948,
		0x9969, 0xa90a, 0xb92b, 0x5af5, 0x4ad4, 0x7ab7, 0x6a96, 0x1a71, 0x0a50,
		0x3a33, 0x2a12, 0xdbfd, 0xcbdc, 0xfbbf, 0xeb9e, 0x9b79, 0x8b58, 0xbb3b,
		0xab1a, 0x6ca6, 0x7c87, 0x4ce4, 0x5cc5, 0x2c22, 0x3c03, 0x0c60, 0x1c41,
		0xedae, 0xfd8f, 0xcdec, 0xddcd, 0xad2a, 0xbd0b, 0x8d68, 0x9d49, 0x7e97,
		0x6eb6, 0x5ed5, 0x4ef4, 0x3e13, 0x2e32, 0x1e51, 0x0e70, 0xff9f, 0xefbe,
		0xdfdd, 0xcffc, 0xbf1b, 0xaf3a, 0x9f59, 0x8f78, 0x9188, 0x81a9, 0xb1ca,
		0xa1eb, 0xd10c, 0xc12d, 0xf14e, 0xe16f, 0x1080, 0x00a1, 0x30c2, 0x20e3,
		0x5004, 0x4025, 0x7046, 0x6067, 0x83b9, 0x9398, 0xa3fb, 0xb3da, 0xc33d,
		0xd31c, 0xe37f, 0xf35e, 0x02b1, 0x1290, 0x22f3, 0x32d2, 0x4235, 0x5214,
		0x6277, 0x7256, 0xb5ea, 0xa5cb, 0x95a8, 0x8589, 0xf56e, 0xe54f, 0xd52c,
		0xc50d, 0x34e2, 0x24c3, 0x14a0, 0x0481, 0x7466, 0x6447, 0x5424, 0x4405,
		0xa7db, 0xb7fa, 0x8799, 0x97b8, 0xe75f, 0xf77e, 0xc71d, 0xd73c, 0x26d3,
		0x36f2, 0x0691, 0x16b0, 0x6657, 0x7676, 0x4615, 0x5634, 0xd94c, 0xc96d,
		0xf90e, 0xe92f, 0x99c8, 0x89e9, 0xb98a, 0xa9ab, 0x5844, 0x4865, 0x7806,
```

```
		0x6827, 0x18c0, 0x08e1, 0x3882, 0x28a3, 0xcb7d, 0xdb5c, 0xeb3f, 0xfb1e,
		0x8bf9, 0x9bd8, 0xabbb, 0xbb9a, 0x4a75, 0x5a54, 0x6a37, 0x7a16, 0x0af1,
		0x1ad0, 0x2ab3, 0x3a92, 0xfd2e, 0xed0f, 0xdd6c, 0xcd4d, 0xbdaa, 0xad8b,
		0x9de8, 0x8dc9, 0x7c26, 0x6c07, 0x5c64, 0x4c45, 0x3ca2, 0x2c83, 0x1ce0,
		0x0cc1, 0xef1f, 0xff3e, 0xcf5d, 0xdf7c, 0xaf9b, 0xbfba, 0x8fd9, 0x9ff8,
		0x6e17, 0x7e36, 0x4e55, 0x5e74, 0x2e93, 0x3eb2, 0x0ed1, 0x1ef0 };

unsigned short libCRCCalcCRC16(unsigned char *buf, unsigned int len) {
	unsigned int i;
	unsigned short cksum = 0;
	for (i = 0; i < len; i++) {
		cksum = libCRCLookupTable[(((cksum >> 8) ^ *buf++) & 0xFF)] ^ (cksum << 8);
	}
	return cksum;
}
```