BMS772 software

Release Notes: v6.0 Release

Rev. 6.0 — Jun 7, 2024 **Release Notes**

Document information

Info	Content
Keywords	BMS772, RDDRONE, S32K144, MC33772, UJA1169, NTAG5, Mobile Robotics, DroneCAN, CyphalCAN, LiPo (NMC), LiFePO4 (LFP), Sodium-ion (Na-ion/SIB)
Abstract	Release notes describing package contents, instructions, open issues, fixes and limitations.

Revision history

Rev	Date	Description
0.1	20200630	Release notes for RDDRONE-BMS772
0.2	20200702	Completing the document
2.6	20200725	Updated after internal review
3.x	20200805	Updated to version 3.x (new help, features, parameters, etc)
3.4	20200911	Updated to the new release, modified state diagrams, tables and more improvements are implemented. Added self-discharge-enable variable and corrected some variables. Changed the software block diagram and added the SBC part to it.
3.6	20210125	Updated to the new release, added Changes table, added some variables, modified state diagrams, adjusted the description of the states, modified the UAVCAN table and added the self-test and timed based and voltage-based balancing.
3.7	20210712	Updated to the new release, changed state diagrams, adjusted state description. Added SMBus, added NFC, added watchdog, added task priorities, changed the software block diagram.
4.0	20210830	Processed comments from GK, added correct UAVCAN information, changed pictures and diagrams.
5.0	20211209	Added display and the NFC fix.
6.0	20240607	Combined tasks, code cleanup, upgraded to new NuttX RTOS version: 11.0+. Added display 5V support, added DroneCAN support for communicating information and controlling the BMS parameters, updated the CyphalCAN part. Default can-mode is “OFF”. Delayed the interface up can and added HW ID filter to dronecan. Added sodium-ion (Na-Ion/SIB) type. Added getting started chapter.

Contact information

For more information, please visit: http://www.nxp.com

1. Introduction

This document describes the release notes for the **RDDRONE-BMS772.** This evaluation kit allows customers to evaluate and perform early (software) development/prototyping on the BMS772 chipset.

The release notes describe the contents of the kit, open issues, changes, fixes and limitations of the released version.

Instructions on how to use the BMS772 evaluation kit can be found on-line:

- Gitbook: https://nxp.gitbook.io/rddrone-bms772/
- NXP webpage: https://www.nxp.com/design/designs/rddrone-bms772-smart-battery-management-for-mobile-robotics:RDDRONE-BMS772
- GitHub with public RTOS example code and binary: https://github.com/NXPHoverGames/RDDRONE-BMS772

For the table of contents see: 13 Contents.

2. Release package

The released package consists of:

- Hardware
  - RDDRONE-BMS772 board
    - Battery Cell Controller (BCC) - MC33772B
      - Measuring all cells, balancing, current measurement including coulomb counting and temperature sensing.
    - Automotive grade MCU – S32K144
      - Programmable MCU for your application
    - System basic chip (SBC) (LDOs, CAN and WD) – UJA1169
      - CAN communication
    - NFC – NTAG5
      - Enables NFC communication
    - Power switch
      - Controlled via the MCU
    - Shunt current resistor
      - Read by the BCC
    - Display
  - Battery selection connectors for 3S up to 6S batteries.
- Documentation and software
  - Release notes & open source RTOS BMS example
    - https://github.com/NXPHoverGames/RDDRONE-BMS772
    - Software updates will be available through gitbook/GitHub
  - On-line documentation on gitbook
    - https://nxp.gitbook.io/rddrone-bms772
  - NXP webpages
    - https://www.nxp.com/design/design-center/development-boards-and-designs/smart-battery-management-for-mobile-robotics:RDDRONE-BMS772

Be sure to check out other enabled platforms:

- Model-Based Design Toolbox MATLAB Simulink
  - BMS772 Board is supported, see the example at: https://community.nxp.com/t5/NXP-Model-Based-Design-Tools/Example-Model-RDDRONE-BMS772/ta-p/1550394
- IDE for bare-metal programming the MCU “S32DS”

3. Changes

3.1 Changes relative to the last 5.0 release

Table 1. Changes 6.0

Item	Description
tasks	Combined the meas, otherCalc, sdChar changedDataHandler tasks in a new batManag task, changed the task priorities picture.
OS	Updated to new NuttX 11.0 version.
Updater parameters	The updater task will now first copy the 2 structs and use that data to update over all the communication ways.
Bms default	This will now wake up the BMS in sleep state before writing the default values.
Fault state	Split the FAULT state in FAULT_ON and FAULT_OFF (as it might be that the output is kept on).
Code style	Used a .clang-format file to format the adjusted code.
Display	Added 5V support for the display.
DroneCAN	Added DroneCAN support, both the BMS messages and a way to control the BMS, with setting parameters, saving, rebooting, etc. Added HW CAN ID filter. Delayed interface up (ifup can0).
UAVCAN / CyphalCAN	Has been given the correct name of CyphalCAN. Delayed interface up (ifup can0). And changed the LiPo and LiFePO4 technology message to the correct 100 and 101 instead of 110 and 111.
parameters	Added: gate-check-enable, dronecan-node-static-id, dronecan-bat-continuous, dronecan-bat-periodic, dronecan-bat-cells, dronecan-bat-info, dronecan-bat-info-aux, can-mode, f-v-out-divider-factor. Added Sodium-ion (Na-ion) and NMC as battery type. Changed: Changed the order of all parameters. Changed parameter name t-cyclic to t-sleep-bcc-cyclic. t-full is 0 when not charging. Changed UAVCAN to Cyphal: uavcan-node-static-id to cyphal-node-static-id, uavcan-es-sub-id to cyphal-es-sub-id, uavcan-bs-sub-id to cyphal-bs-sub-id, uavcan-bp-sub-id to cyphal-bp-sub-id, uavcan-legacy-bi-ssub-id to cyphal-legacy-bi-sub-id. Changed uavcan-fd-mode to can-fd-mode, uavcan-bitrate to can-bitrate and uavcan-fd-bitrate to can-fd-bitrate. Removed: s-charge-stdev.

3.2 Changes of 5.0 relative to 4.0

Table 2. Changes 5.0

Item	Description
NFC	Added the NFC bug fix that works on all batches.
Display	Added LCD/OLED to display BMS information.

3.3 Changes of 4.0 relative to 3.6

Table 3. Changes 4.0

Item	Description
Power	Implemented the sleep state to be low power (VLPR mode). Added a lower power state during the charge-relaxation mode to not discharge the cells too much.
NFC	Enabled NFC to read out BMS information using the NTAG5.
SMBus	SMBus can be enabled to retrieve BMS information using the I^2C slave interface.

Item	Description
Watchdog	Added the watchdog on the SBC (UJA1169TK/F/3).
Task priority	Added task priorities.
Time	Added the “bms time” command.
Charge end current	Added a variable to enter the system current to subtract this from the measured current during charging.
Measurements in fault	Enabled the measurements in the fault state.
Emergency button	Enabled the option to have an emergency button to disconnect the power.
Self-test watchdog	Added a self-test to check if the watchdog jumper is mounted.
CLI	Added the BMS state, added that the top command enables/disables the measurements on the CLI, re-arranged the output.
Cell OV TH	Made sure that the BCC OV TH is at least the set OV (8 bit register). Added a software cell OV TH to check for the actual threshold using the cell voltage measurements.
Updater task	Added an updater task to take care of updating the UAVCAN, SMBus, CLI and the NFC.
Charge re-charge	Added that if the voltage drop is too much again after charge-complete, it will initiate a re-charge.
Flight mode	The s-in-flight parameter will be used to determine if the power should be cut off or not during a fault. A peak overcurrent will always cut off the power of the battery. the flight mode parameter now is saved, and read during startup. The way the s-in-flight is determined is different than the previous flight mode.
Current check	i-batt is now used for i-peak-max and i-batt-avg is used for the i-out-max and the i-charge-max.
UAVCAN	Implemented the UAVCAN V1 0.2 battery service message (source state, battery status and battery parameters). Added a way to get the UAVCAN messages on a terminal. Added the UAVCAN V0 legacy message.
Parameters	Changed uavcan-ess-sub-id to uavcan-es-sub-id . Changed the default value for the subject ids (uavcan-es-sub-id, uavcan-bs-sub-id and uavcan-bp-sub-id) from 65535 to 4096, 4097 and 4098, i-peak to i-range-max. Added: i-batt-10s-avg, s-in-flight, v-cell-nominal, i-system, i-peak-max, v-recharge-margin, emergency-button-enable, smbus-enable, uavcan-legacy-bi-sub-id

3.4 Changes of 3.6 relative to 3.4

Table 4. Changes 3.6

Item	Description
Power on self-test	More components are tested at startup (NTAG5, A1007), with better output on terminal.
NuttX version	Upgraded to NuttX version 10.0 (stable release).
Sense resistor test	Added a test that will detect if the sense resistor is connected to the measurement input.
Negative input for unsigned variables	The CLI will notify you and will not set the variable when a negative number is entered for an unsigned variable.
Self-test state	The self-test is only done at startup, made it a separate state (not only init state anymore).
LED color	LED is now solid red in the self-test state.
SoC calibration	Added the OCV state to calibrate the state of charge when in sleep mode. Added the SoC calibration to the self-discharge state.
CLI set parameter	Fixed that using the CLI, variables can’t be set with more than its variable type can hold.

Item	Description
UV fault to deepsleep	Added that when an undervoltage occurs, it will go to the deepsleep mode after some time to protect the battery.
Reboot	Added the reboot command to the CLI to reboot the microcontroller.
Flight-mode	Added the flight-mode-enable and i-flight-mode parameters. This can be used to not cut the power to the FMU and motors in flight.
Modified a-factory parameter	After setting the factory capacity, the BMS will recalculate and set the full charge capacity and end of charge current as well.
Floating point variables	The floating point variables are better limited now.
Charge to deepsleep	Added that if the on-board push button SW1 is hold for five seconds in the charge state, the state will transition to deepsleep (via self-discharge).
NFC	Is hard-powered down with GPIO.
Discharge to storage	After a long time-out, the BMS will discharge to storage level and go to the deepsleep state.
CLI syntax	Some wrong syntaxes are now fixed in the CLI.
CLI help messages	Improved the help message if the wrong number of cells are attached/inserted.
Wrong messages	The wrong BMS under-temperature detected message if the sensor is disabled has been fixed. The first “pin rising edge BCC_FAULT” message has been deleted.
Message limit	Limited the number of “Rising edge BCC_FAULT” and “clearing CC overflow” messages.
CLI color messages	Added functions for the green (good), yellow (warning) and red (error) messages.
BCC com errors	This will not be reset (wrongly), but it will be counted and with 255 errors it will reset it correctly.
UAVCAN	New draft of the UAVCAN V1 standard message is implemented.
Data struct	Added the unit and type string to the data struct. Added floating point accuracy to the floating point limitations.
Parameters	Added the t-sleep-timeout, i-charge-nominal, i-out-nominal, i-flight-mode, battery-type, flight-mode-enable and m-mass. Changed the model-id to uint64_t, t-fault-timeout to use with uv to go to the deepsleep state, t-ocv-cyclic0 and t-ocv-cyclic1 are implemented, changed the uavcan-subject-id to 3 different parameters for each message: uavcan-ess-sub-id, uavcan-bs-sub-id and uavcan-bp-sub-id. v-cell-margin is default 50mV instead of 30mV. ocv-slope is in mV/Amin instead of V/Amin.
State diagram	Added LED indication, Added SELF TEST state, added button hold to the charge to self-discharge state transition, Added SLEEP_TIMEOUT to the sleep to self-discharge state transition, FAULT_TIMEOUT used for transition to go to deepsleep with an undervoltage.
Charge state diagram	Added LED indication and added button hold to the charge to self-discharge state transition.
Cell balancing	The cell balancing is not only based on the cell voltage compared to the desired voltage, but it is based on the calculated estimated cell balance minutes as well.

4. Limitations

Table 5. Limitations

Item	Description
Software stack	**Limitation:** The evaluation kit only contains the basic battery management system code to access and evaluate the NXP technology. **Impact:** A full BMS requires additional software. The code is supplied as is to be used at own risk.
Charging	**Limitation:** The BMS will not limit the incoming current or voltage while charging, it can only disconnect the battery. **Impact:** A current and voltage limiting power supply needs to be attached to the BMS to charge the battery.

5. Known issues

Table 6. Known issues

Item	Description
CyphalCAN battery status	**Applies to:** Release Package. **Issue:** Only the draft of the status message is implemented. **Workaround:** Update the CyphalCAN code to implement the standard.
Authentication	**Applies to:** Release Package. **Issue:** Isn't implemented. **Workaround:** Upcoming releases.
State of charge calibration	**Applies to:** Release Package. **Issue:** The state of charge calibration table can be different for each battery. **Workaround:** Add more precise dedicated calibration table of your own used battery.
Diagnostics	**Applies to:** Release Package. **Issue:** Diagnostics will not be part of the basic release **Workaround:** Diagnostics functionality can be added by obtaining NDA and safety library, please see chapter 8.8.
No open cell connection or short cell check on sense wires.	**Applies to:** Release Package. **Issue:** There is no check if the cell leads are open or shorted, as this is part of the safety library, which requires NDA. **Workaround:** Safety functionality can be added by obtaining NDA and safety library, please see chapter 8.8.

For issues of older releases, please consult the respective release notes.

6. Getting started / quick start guide

This chapter describes the things to consider when first starting or using the BMS772 HW and SW. This can be seen as a quick start guide.

6.1 Configure the HW

Start with configuring the HW.

1. Solder the solder jumpers to configure the HW for the correct cell count
    a. See Table 7 or look at https://nxp.gitbook.io/rddrone-bms772/user-guide/getting-started-with-the-rddrone-bms772/configuring-the-hardware#cell-terminal-connection
2. Solder the correct balance header (JP1) for the correct cell count.
    a. Keep in mind, pin 7 is ground.
    b. See https://nxp.gitbook.io/rddrone-bms772/user-guide/getting-started-with-the-rddrone-bms772/configuring-the-hardware#cell-terminal-connection
3. Check how to connect the battery (power in) and rest of the system (power out) to the BMS772 board.
    a. You could solder the provided XT90 connectors.
    b. See https://nxp.gitbook.io/rddrone-bms772/user-guide/getting-started-with-the-rddrone-bms772/configuring-the-hardware#power-connectors
    c. Or see chapter 7.1 Board organization.
4. Check to add the temperature sensor of the battery.
    a. You could add the provided NTC cable and connect this to J1.
        i. Note: you need to enable the temperature sensor later.
    b. See chapter 8.3 How to configure the temperature sensor & 7.1 Board organization or https://nxp.gitbook.io/rddrone-bms772/software-guide-nuttx/how-to-enable-the-battery-temperature-sensor
5. Check to add a UART or (Drone)CAN connection to configure the BMS772
    a. The UART connection is most easy way to configure the BMS parameters.
        i. See 8.2 How to use the CLI and 9.1 How to configure the parameters via the CLI.
    b. This can also be done via CAN using the DroneCAN protocol and the DroneCAN GUI tool.
        i. 8.4 How to use the DroneCAN interface, 8.4.2 How to use the DroneCAN GUI tool to configure the BMS and 9.2 How to configure the parameters via the DroneCAN GUI tool
6. Check to add the provided display to the BMS772
    a. See chapter 7.1 Board organization.

6.2 Power the BMS772

It is always advisable to start with a current limiting power supply (6-25V, e.g. 12V ±100mA).

You can mimic the cell voltages by putting resistors (e.g. 1k) in series and supply these with the same power supply as the BMS power in. These resistors can replace the cells to simulate a battery in a safe matter. Like a cell, connect every resistor to the balance connector JP1 using XHP-7* connector plus 01SXHSXH-26L150 wires. *7-way for 6S.

Hook up the power supply or battery to the “power in” connector J4. And if you have them, connect it to the resistors as well.

Note: if you do not mimic the cell voltages, the NuttX RTOS BMS example will go into FAULT mode, not enabling the output and the red LED will be red blinking.

6.3 Get the latest SW

It is always advisable to use the latest version of the software.

See https://github.com/NXPHoverGames/RDDRONE-BMS772/tree/main/Binary for the latest binary.

See 8.1 How to program the BMS.

Note: there are also BMS772 example in the MATLAB SimuLink MBDT environment: https://community.nxp.com/t5/NXP-Model-Based-Design-Tools/Example-Model-RDDRONE-BMS772/ta-p/1550394

6.4 Configure the BMS772 using variables

The BMS has many configurable parameters. Depending on the need, these can be adjusted. It is advisable to go over the parameter and see what can be configured.

These can be found:

- 9.3.1 The common battery variables list
- 9.3.2 The calculated battery variables list
- 9.3.3 The additional battery variables list
- 9.3.4 The configuration variables list
- 9.3.5 The can variables list
- 9.3.6 The hardware parameters

6.4.1 The most important BMS variables

Do not forget to save the parameter with the “bms save” command.

The most important variables to look at are:

- n-cells
  - the amount of cells of the battery [3 ... 6]
  - This should reflect the soldered configuration.
- sensor-enable
  - To enable the battery temperature sensor [0=disable, 1=enable]

- battery-type
  - Which battery type do you have?

    [0=LiPo, 1=LiFePO4, 2=LiFeYPO4, 3=NMC, 4=Sodium-ion]
  - This will change the under- and over-voltage, storage voltage, nominal voltage and the OCV curve it uses to correct the state of charge.
- a-rem
  - The remaining capacity [Ah]
  - The BMS does an estimate on what the remaining capacity is based on an OCV(open cell voltage)/SoC (state of charge) table from one specific battery. Keep in mind that every battery is different.
  - It is advisable to insert the correct OCV/SoC table for the used battery.
  - You can correct the state of charge (SoC) with this variable.
- a-factory
  - The factory capacity of the battery [Ah]
  - This should be the capacity stated on the battery.
- a-full
  - The full charge capacity of the battery [Ah]
  - This degrades over time. Less capacity can be stored in the battery.
  - State of health is calculated based on this value.
  - If unknown, set to the same value as factory capacity.
- model-name
  - The name of the battery
- i-charge-full
  - The end of charge current of the battery (could be 10% from i-charge-max) [mA]
- i-charge-max
  - The maximum charge current [A]
- i-peak-max
  - Maximum peak current threshold [A]

6.5 Debugging problems

- First thing you could do is looking at the LED, see Table 15 LED states
- Look at the reported problem via the CLI, see 8.2 How to use the CLI
- Check if everything is connected correctly.
- Check if everything is configured correctly.
- Reboot the BMS772 (can be done via the CLI and via the DroneCAN GUI tool).
- Measure (safely) some (test) points if you think something is not correct.
- Repower the full BMS.
- Do a visual inspection of the board.

7. Block diagram

In the following images you can see the hardware block diagram. For more information about the hardware look at the gitbook: https://nxp.gitbook.io/rddrone-bms772/ or see the user manual RDDRONE-BMS772_UG on nxp.com.

7.1 Board organization

The board is organized as shown in the figures below, Figure 2 Board block layout -- Top and Figure 1 Board map -- Top.

On the top level the main connectors are located. The battery should be connected to Power in (J4) and balance input (JP1). A correct cell terminal should be mounted, which fits the type of battery. The FETs Q1 up to Q8 are used to (dis)connect the Power in (+) (J4) from the Power out (+) (J5).

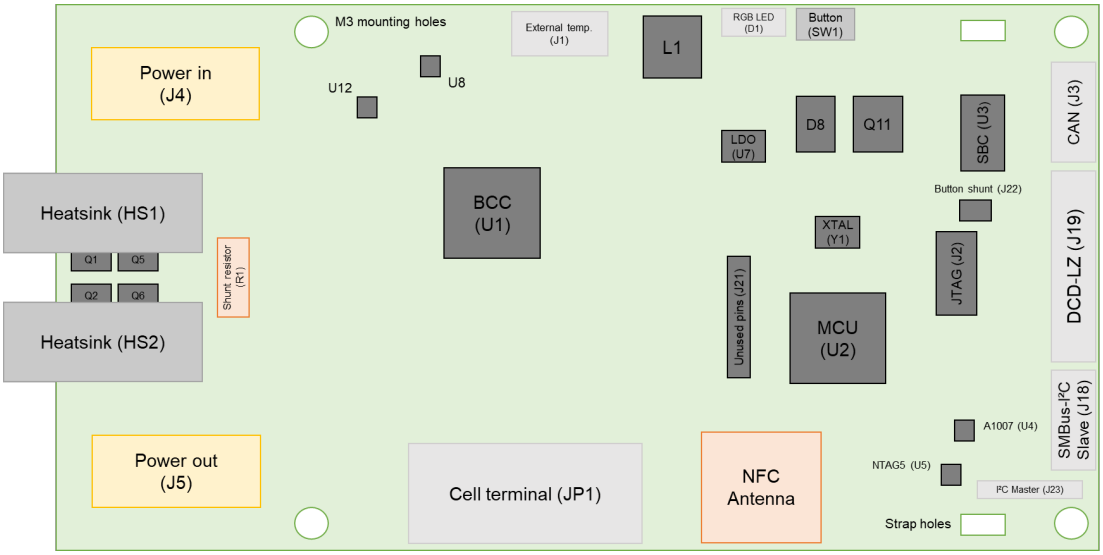


Figure 2 Board block layout -- Top

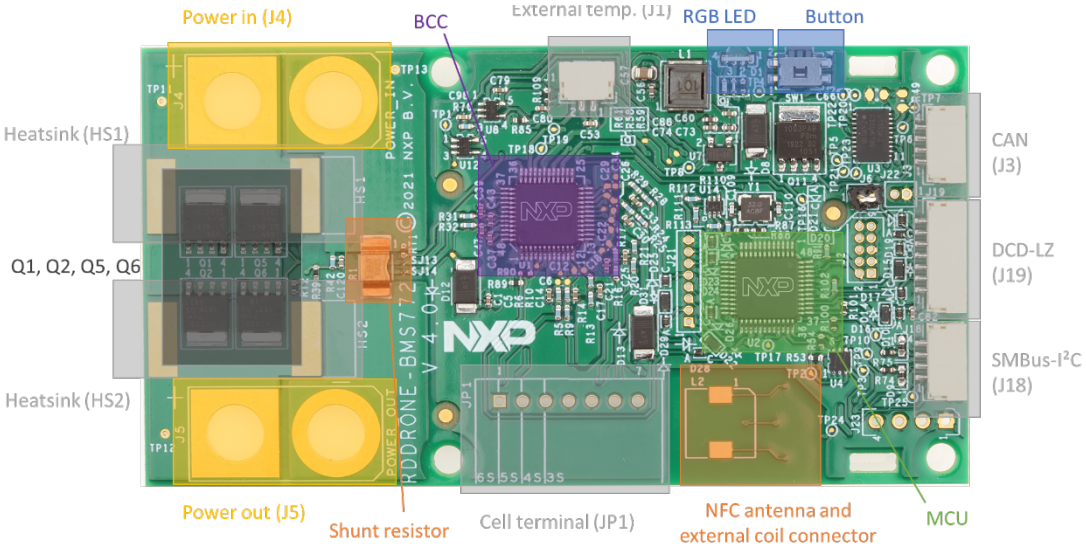


Figure 1 Board map -- Top

For bottom the important bottom connector is J20, this is normally used to attach the CAN termination.

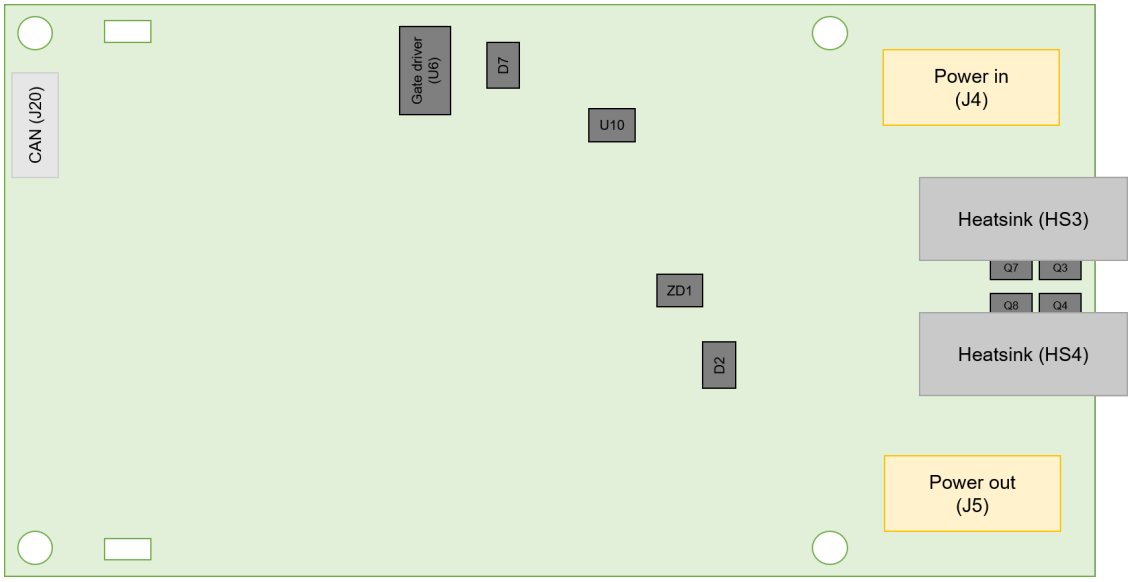


Figure 4 Board block layout -- Bottom

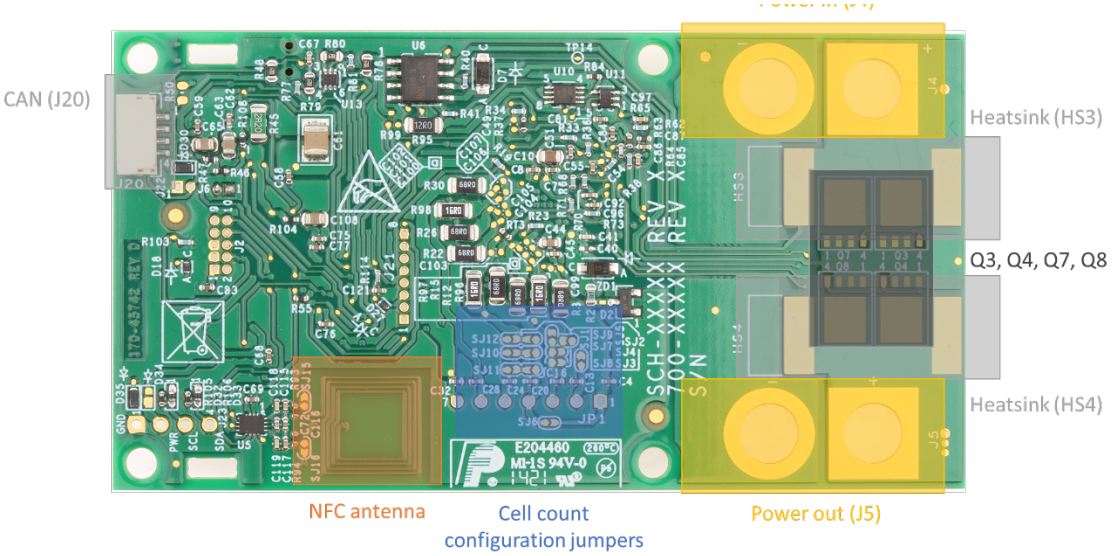


Figure 3 Board map -- Bottom

Solder jumpers should be set to the correct number of cells. Below in Table 7 you can see which solder jumpers needs to be soldered for a specific cell count/configuration.

Table 7. Cell configuration table

8. How to

8.1 How to program the BMS

8.1.1 Software setup and debugger adapter board

The software can only be written to the board using a debugger. The HoverGames drone kit includes a J-Link EDU Mini debugger. To use it, you need to install J-Link Software Pack. For this example, the J-Link commander is used. Links are provided at the Downloads chapter.

The debugger can be plugged into the BMS using a small adapter board. This small PCB comes with a 3D printed case that can easily be put together. The J-Link debugger can be connected using an **SWD cable**. The connectors have to be oriented such that the **wires directly go to the side of the board**, as shown in Figure 5 below.

While you do not need it right now, the adapter board also has a 6-pin connector for a **USB-TTL-3V3 cable**, which you can use to access the system console (CLI) of the BMS. The 3D printed case has a small **notch** on one side of the connector. The USB-TTL-3V3 cable needs to be plugged in such that the **black (ground) wire is on the same side as this notch** in the case. Make sure the cables are plugged in as shown in the picture below.

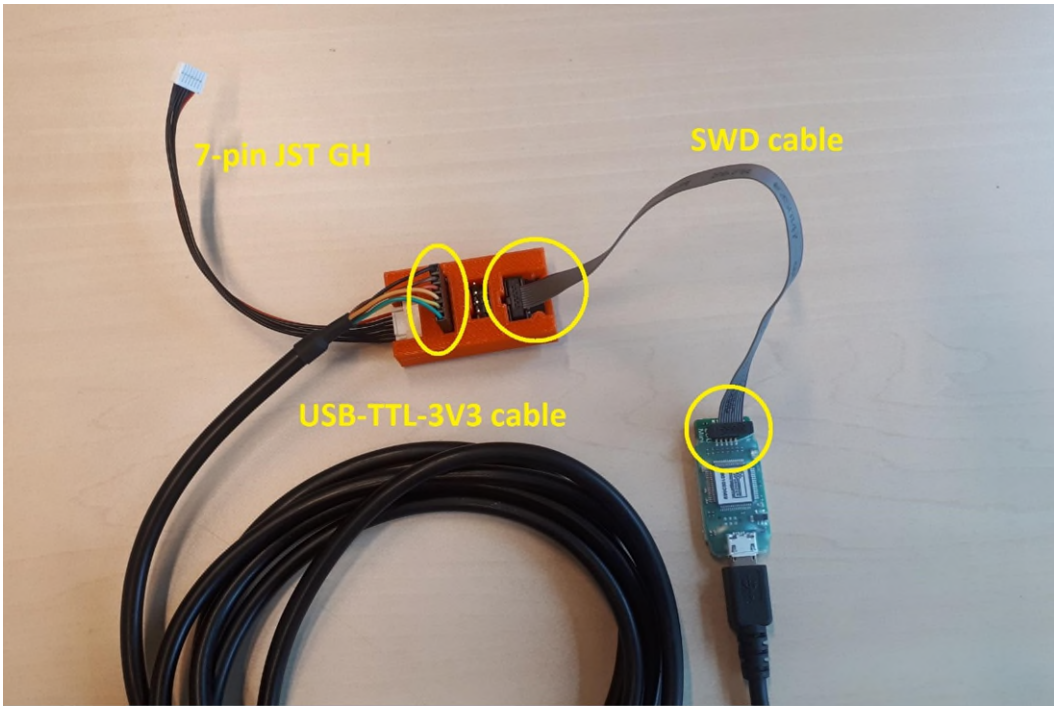


Figure 5: The debug adapter board

8.1.2 Programming the software

The firmware can be programmed to the board with the J-Link debugger. You first need to download a firmware binary (.bin file). We keep up-to-date links to the recommended releases on our downloads page because it is important that you start with the most stable and up-to-date version of the firmware.
See https://github.com/NXPHoverGames/RDDRONE-BMS772 for the most recent release. Connect the debugger to the BMS using the 7-pin JST-GH connector from the debug adapter board to J19 and plug the USB coming from the debugger into your computer. You should provide power to the BMS using the battery or a power supply. Then start the J-Link Commander program, and follow these steps:
To program the binary a file could be made, make a new notepad file (to enter some text). This file could be named “flash.jlink” or something else.
Add the following to the file:

```
si 1
speed 1000
device S32K144
connect S32K144
S
r
w1 0x40020007, 0x44
w1 0x40020000, 0x80
sleep 1000
loadbin "<absolute path>nutt.bin" 0
r
g
q
```

note: Change the path to the location of the nuttx binary file.

To program the binary to the BMS use the command “JLinkExe flash.jlink” in the terminal where the “flash.jlink” script is located. If you named the “flash.jlink” script differently, this needs to change in the command as well.

note: If you are using the JLink commander IDE, don’t forget to stop the core and erase the chip.

8.1.3 Downloads

Download J-Link Software and Documentation Pack
J-Link Commander is used to flash binaries onto the RDDRONE-BMS772 board. The latest (stable) release of the J-Link Software and Documentation Pack is available at the SEGGER website for different operating systems.

8.2 How to use the CLI

The command line interface (CLI) can be easily used over the UART protocol.
If you have the DCD-LZ breakout board, this can be used to easily connect a TTL-232R-3V3 (FTDI cable) to this board and connect the DCD-LZ board to the BMS772. This is done using the 7-pin JST-GH connector from the debug adapter board (DCD-LZ) to connector J19 of the BMS

If you do not have the DCD-LZ breakout board, you can make (solder) a connection between a TTL-232R-3V3 (FTDI cable) to this board using 3 wires (GND, TXD and RXD) to the 7-pin JST-GH connector J19 of the BMS (pin 7 (GND), pin 3 (UART_RX) and pin 2 (UART_TX)).

Plug the USB coming from the FTDI cable into your computer. For windows, make sure you have the correct FTDI drivers installed on your windows PC. Here is a manual: https://support.arduino.cc/hc/en-us/articles/4411305694610-Install-or-update-FTDI-drivers. Make sure you can see the device in your “Device Manager” under “Ports (COM & LPT).

Open a UART terminal like minicom on a Linux machine or for example Tera Term, PuTTY or MobaXterm for a windows machine. Make the connection to the correct port. Windows users can find the COM port in the step above. For minicom, start minicom with: “minicom -c on” to enable the colors on the CLI.

The CLI works only with lowercase commands. The settings are:

- 115200 Baud
- 8 data bits
- 1 stop bit

Type “bms help” to get the help for the CLI.

Note: you could recompile the code using a different UART port to route all UART console communication to J21. So J21 will be used for the CLI. See the NuttX menuconfig quickstart guide.

8.3 How to configure the temperature sensor

By default, the temperature sensor is not enabled. To configure the temperature sensor to be used, this temperature sensor needs to be connected to J1 using a 2-pin JST-GH connector. The maximum length of the cable that leads to the temperature sensor needs to be 20cm. This temperature sensor needs to be a 10k NTC, like the NTCLE100E3103JB0 from Vishay. With the CLI type:
“bms set sensor-enable 1”.

8.4 How to use the DroneCAN interface

Connect the DroneCAN device (like the RDDRONE-UCANS32K146, a device, for drones this is typically an FMU. Or you could connect the BMS to a PC using a PCAN-USB (CAN to USB converter). This can be connected to one of the 2x JST-GH 4 pin connectors (J3 or J20) with 1 on 1 wires. End the CAN bus with a 120Ω terminator resistor between CAN high and CAN low.
For the pinout of these connectors, please see the schematic on nxp.com.

More information on DroneCAN can be found at: https://dronecan.github.io/

Make sure the BMS has DroneCAN enabled, see the parameter “can-mode”. Use the CLI to “bms set can-mode dronecan”. Keep in mind to do a “bms save” and “reboot” to apply this.

8.4.1 How to set up the DroneCAN GUI tool and start dynamic node ID allocation

The DroneCAN GUI tool information and download can be found here:
https://github.com/dronecan/gui_tool

Download and install the DroneCAN GUI tool on a native Linux device.
Attach this native Linux device to the BMS via a USB to CAN converter, like PCAN-USB.
On the Linux device, check if can0 is present with the following Linux command.

```
ifconfig -a | grep can
```

You sould see something like: “can0: flags=128<NOARP> mtu 16”.
To enable this CAN device, enter the following Linux command:

```
sudo ip link set can0 up type can bitrate 1000000 dbitrate 4000000 fd on
```

After can is enabled, start the DroneCAN GUI tool with the following command:

```
dronecan_gui_tool
```

can0 should be seen as the CAN interface, like the CAN interface setup screenshot in Figure 6, then press OK.
Keep in mind *that for the Dynamic node ID allocation server, you do not want to add the DSDL definitions! With DSDL definitions it will not work!*

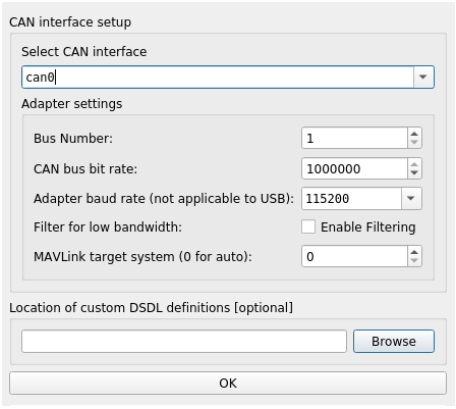


Figure 6: DroneCAN GUI CAN interface setup

The BMS CLI should have indicated that it is waiting for a dynamic node ID allocation: “DroneCAN: Waiting for dynamic node ID allocation”.

You will see the following screen in Figure 7. This is the DroneCAN GUI tool. In the DroneCAN GUI tool, set your local node ID (or keep the default), and press right next to it to enable the local node ID.

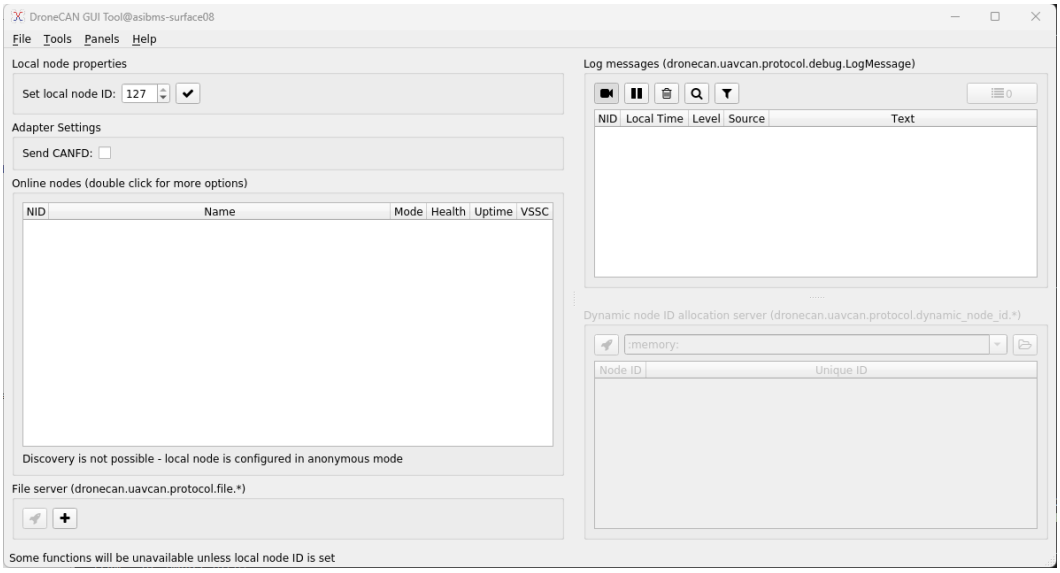


Figure 7: DroneCAN GUI tool node ID unset

After setting your local node ID, you should be able to click on the rocket symbol in the right bottom part of the screen to start the dynamic node ID allocation server. This can be seen below in Figure 8:

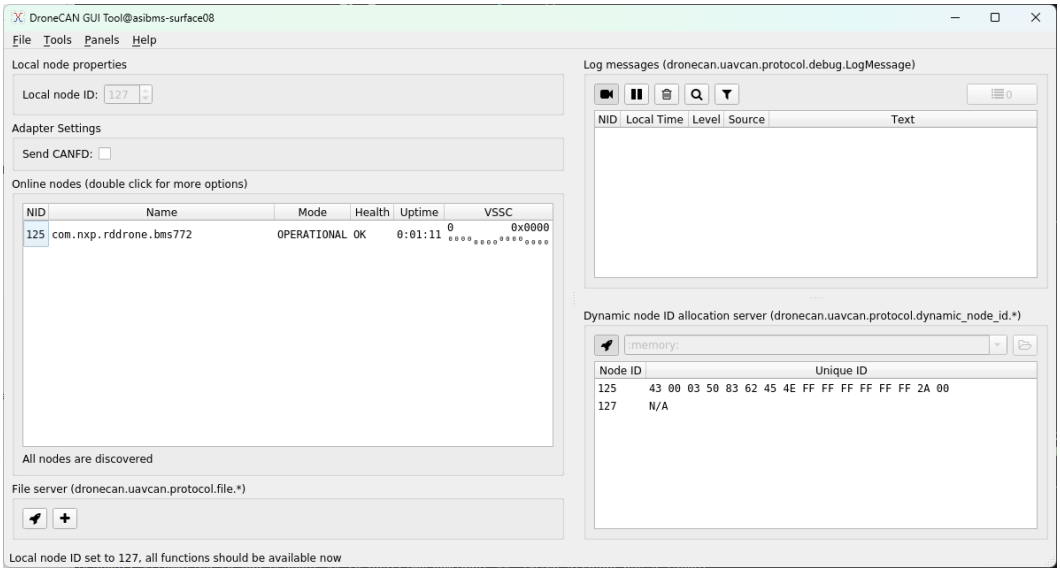


Figure 8: DroneCAN GUI tool dynamic allocation server

The BMS should now be seen in this tool. The BMS should have reported via the CLI that it has been given a node ID. In this example, the given node ID of the BMS is 125.

8.4.2 How to use the DroneCAN GUI tool to configure the BMS

See chapter 9.2 How to configure the parameters via the DroneCAN GUI tool.

8.4.3 How to use the DroneCAN GUI tool to monitor the DroneCAN messages

In order to monitor the DroneCAN messages with the DroneCAN GUI tool, make sure you have the DroneCAN DSDL messages stored on your device.

This can be done by cloning the following git repository to your device where the DroneCAN GUI tool is running: https://github.com/dronecan/DSDL\

```
git clone https://github.com/dronecan/DSDL
```

In the CAN interface setup, click Browse and navigate to DSDL/ardupilo and click OK. See

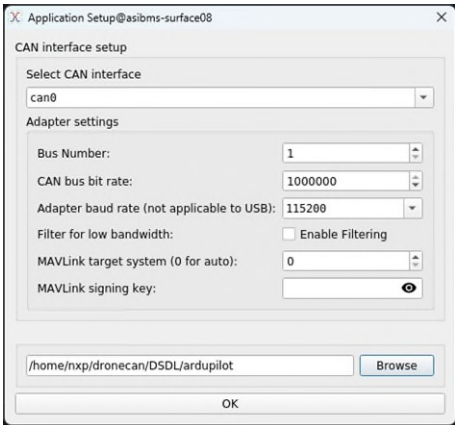


Figure 9: DroneCAN GUI CAN interface setup with DSDL

Figure 9 for the example. It could be that in newer version, the DSDL message definition does not need to be added.

It can take some time for the GUI tool to startup for the first time, but you will see the following window, see Figure 10. It could be that you need to have a DroneCAN device on the bus which is sending messages.

In the DroneCAN GUI tool, set your local node ID (or keep the default), and press right next to it to enable the local node ID.

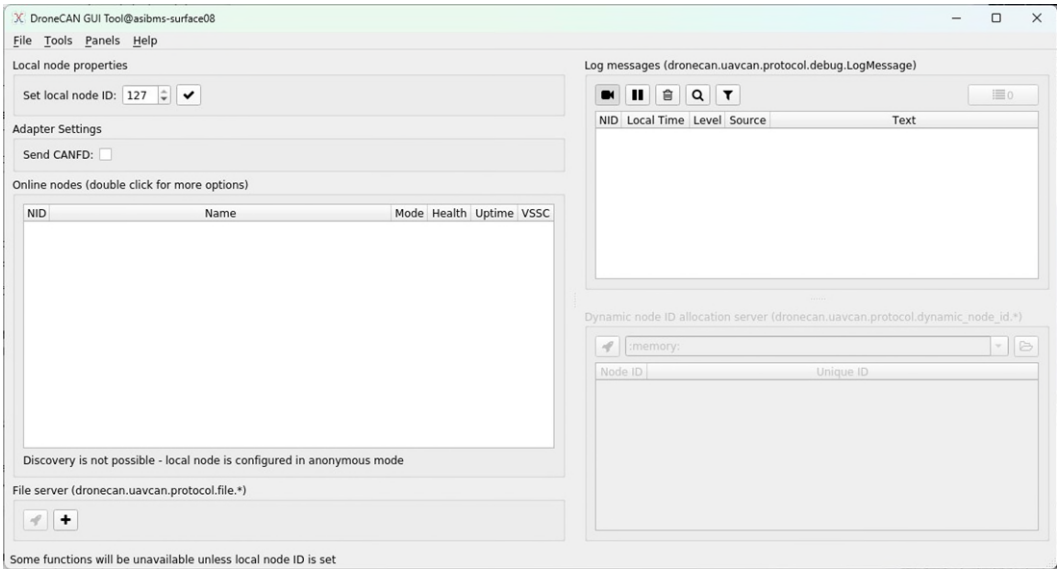


Figure 10: DroneCAN GUI tool

Keep in mind that with dynamic node ID allocation, the BMS needs to have a node ID allocated via a different instance. With DSDL messages, the dynamic node allocation server cannot be started. See chapter 8.4.1 for more information.

In the current instance of the DroneCAN GUI tool go to Tools -> Bus Monitor.
In the Bus Monitor window, click on the recording symbol to start seeing the messages.

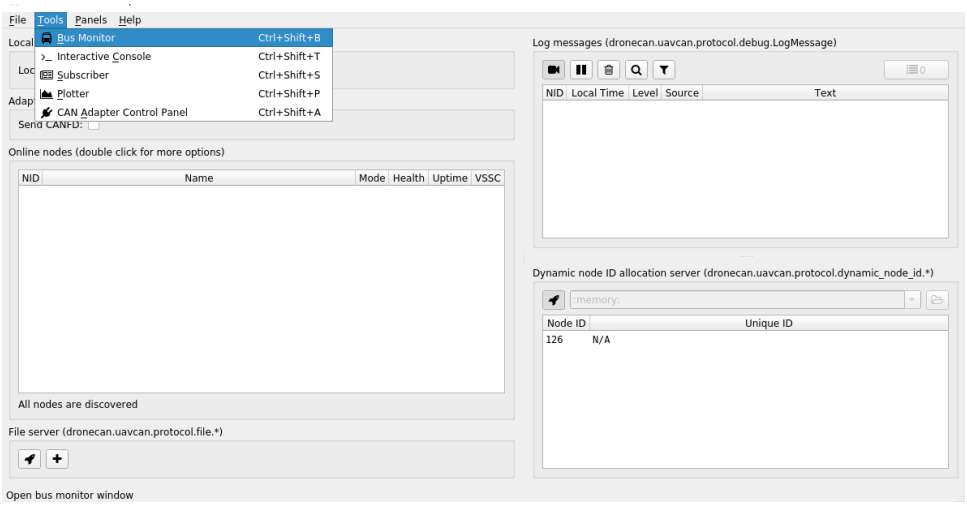


Figure 11: DroneCAN GUI tool how to monitor

In Figure 12 the Bus Monitor tool can be seen before the recording is enabled:

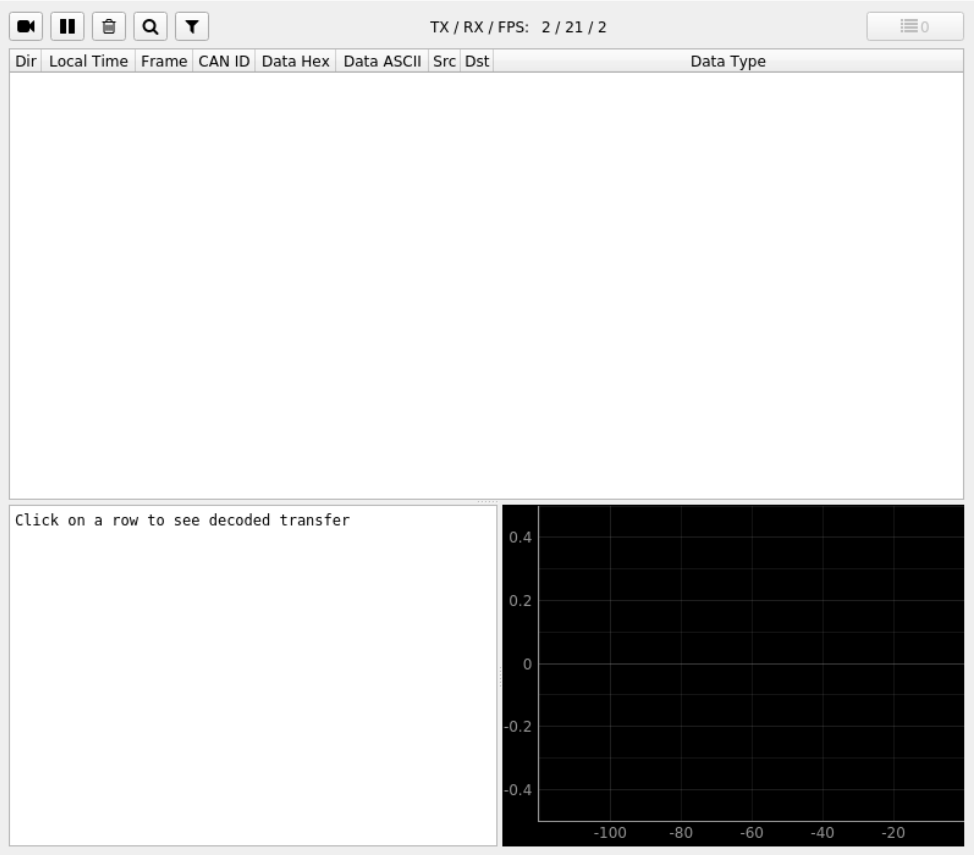


Figure 12: DroneCAN GUI tool before recording

In Figure 13 the Bus Monitor tool can be seen with the recording enabled and multiple messages are shown to be received.

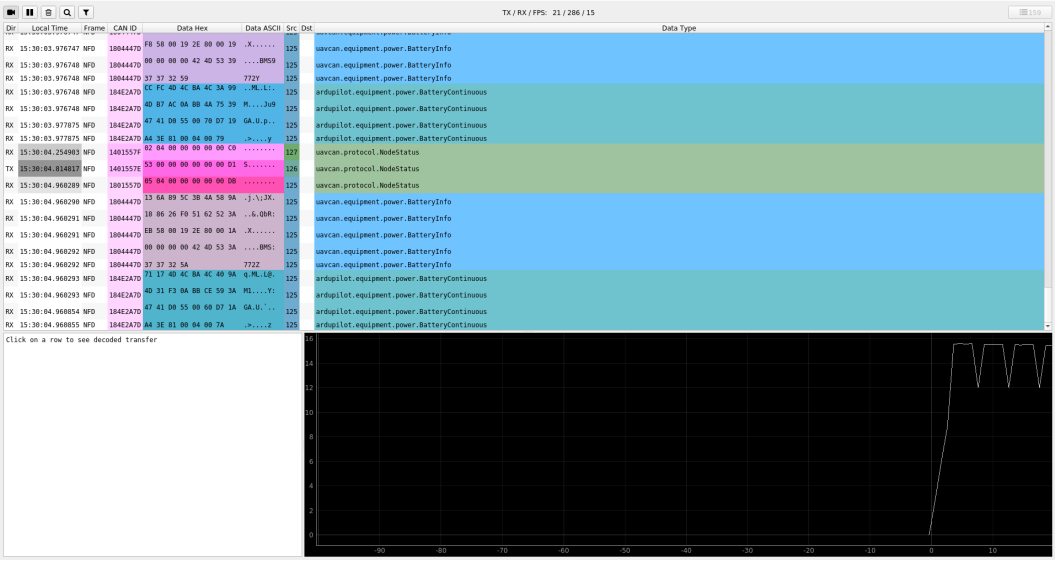


Figure 13: DroneCAN GUI tool recording BMS messages

8.5 How to use the CyphalCAN interface

For Cyphal these messages are implemented:
https://github.com/OpenCyphal/public_regulated_data_types/tree/master/reg/udral/service/battery
For the tables with the implemented messages see chapter 11.5.8.2.

For more information on CyhalCAN, see https://opencyphal.org/
For PX4 and ArduPilot the recommendation is to use DroneCAN.

Connect the CyphalCAN device (like the RDDRONE-UCANS32K146, a device, for drones this is typically an FMU. This can be connected to one of the 2x JST-GH 4 pin connectors 11.5.8.2 (J3 or J20) with 1 on 1 wires. End the CAN bus with a 120Ω terminator resistor between CAN high and CAN low.
For the pinout of these connectors, please see the schematic on
https://www.nxp.com/design/design-center/designs/smart-battery-management-for-mobile-robotics:RDDRONE-BMS772

Make sure the BMS has CyphalCAN enabled, see the parameter “can-mode”. Use the CLI to “bms set can-mode cyphal”. Keep in mind to do a “bms save” and “reboot” to apply this.

See 8.5.1 How to test out the CyphalCAN interface if you would like to test this.

8.5.1 How to test out the CyphalCAN interface

There is a python script to test the CyphalCAN communication from the BMS. The needed components are a working BMS, a can device (for example a PCAN-USB device) and a can terminator resistor. For this example a native Linux device is used.

Hook up the system according to 8.5 How to use the CyphalCAN interface.

On the Linux device:
If not already done, install the can-utils and python3:
```
sudo apt install can-utils python3
```

Install pyuavcan version 1.1.3:
```
python3 -m pip install pyuavcan==1.1.3
```

Clone the cannode v1 tools, and update:
```
git clone https://github.com/PetervdPerk-NXP/cannode-v1-tools
cd cannode-v1-tools/
git submodule update –init
```

Adjust print_battery_service.py to make it work. For example, check that it uses the correct can device, for can0 this should be:
media = pyuavcan.transport.can.media.socketcan.SocketCANMedia('can0', mtu=8)

Check if there is a CAN interface available:
```
ifconfig -a | grep can
```

Enable the CAN interface:
```
sudo ip link set can0 up type can bitrate 1000000 dbitrate 4000000 fd on
```

Then run the python script:
```
python3 print_battery_service.py
```

On the BMS772 CLI (UART)
If you have the node ID allocation on (plug_and_play (pnp), do not do this following step, otherwise the BMS needs to get a node ID
This ID needs to be (<127) and not ID 42! So for example:

```
bms set cyphalcan-node-static-id 100
```

Set the BMS in cyphalcan mode, save the change and reboot to take effect:

```
bms set can-mode cyphal
bms save
reboot
```

8.6 How to use SMBus (I²C slave)

To enable the update, be sure to set smbus-enable to 1 with “bms set smbus-enable 1”. If this is enabled, one can use the BMS as an I²C slave device to get the information. Use the J18 connector (I2C/SMBUS) on the BMS, hook up your I²C master device with pull-ups to this connector. The SMBus information is based on the SBS1.1 specification. These are the supported messages:

Table 8. SMBus variable list

Parameter	Unit	Datatype	Description	I²C Address
temperature	K	uint16_t	The temperature of the external battery temperature sensor, 0 otherwise.	0x08
voltage	mV	uint16_t	The voltage of the battery.	0x09
current	mA	uint16_t	The last recorded current of the battery.	0x0A
average_current	mA	uint16_t	The average current since the last measurement (period t-meas (default 1s)).	0x0B
max_error	%	uint16_t	Just set to 5%. Not tracked.	0x0C
relative_state_of_charge	%	uint16_t	Set to the state of charge value.	0x0D
absolute_state_of_charge	%	uint16_t	Set to the state of charge value.	0x0E
remaining_capacity	mAh	uint16_t	The remaining capacity of the battery.	0x0F
full_charge_capacity	mAh	uint16_t	The full charge capacity of the battery.	0x10
run_time_to_empty	min	uint16_t	Calculated time to empty based on current and remaining_capacity.	0x11
average_time_to_empty	min	uint16_t	Calculated the time to empty based on average_current and remaining_capacity.	0x12
cycle_count	cycle	uint16_t	Set to the n-charges value.	0x17
design_capacity	mAh	uint16_t	Set to the factory capacity.	0x18
design_voltage	mV	uint16_t	Set to the cell overvoltage value.	0x19
manufacture_date	-	uint16_t	Set to the defines in the code. Not actual manufacturer dates. (year-1980)*512 + month*32 + day	0x1B
serial_number	-	uint16_t	Set to the battery id (batt-id).	0x1C
manufacturer_name	-	char *	Set to “NXP”.	0x20
device_name	-	char *	Set to "RDDRONE-BMS772"	0x21
device_chemistry	-	char *	This is a 3 letter battery device chemistry "LiP", "LFP" or "LFY" (LiPo, LiFePo4, LiFeYPo4).	0x22
manufacturer_data	-	uint8_t *	Set to 0x0. (length 1).	0x23
cell1_voltage	mV	uint16_t	Cell voltage of cell1.	0x3A
cell2_voltage	mV	uint16_t	Cell voltage of cell2.	0x3B
cell3_voltage	mV	uint16_t	Cell voltage of cell3.	0x3C
cell4_voltage	mV	uint16_t	Cell voltage of cell4.	0x3D
cell5_voltage	mV	uint16_t	Cell voltage of cell5.	0x3E
cell6_voltage	mV	uint16_t	Cell voltage of cell6.	0x3F

8.7 How to use NFC

The BMS has an NTAG5 on board to have NFC communication with an NFC enabled device. In the current example an NDEF text record is implemented. This text record has the actual battery information and is updated each measurement time. If the data is read out via NFC, the new updated data cannot be written to the NTAG at the same time. To read the data with NFC, approach the BMS with an NFC enabled mobile phone. It should automatically pop up with the text message after a read, as can been seen in Figure 14: NFC read screenshot. An NFC read application could be used as well. If the BMS is in a low power state, the NFC is disabled and it will show some information on the state. If the BMS is in the sleep state, an NFC interaction can be used to wake up the BMS.

The following information can be found using an NFC read:

- Output voltage
- Battery current
- State of charge
- State of health
- Output current
- Number of charges
- Battery id
- Model id
- Current BMS state.

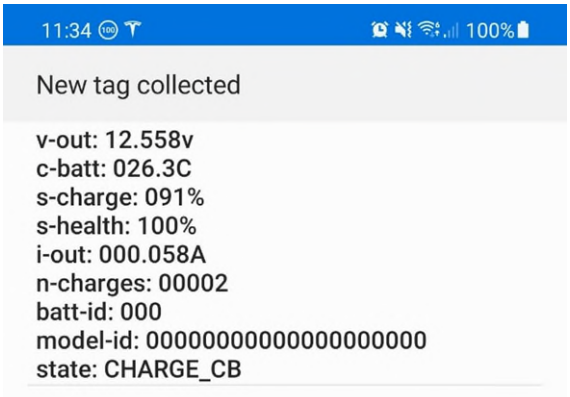


Figure 14: NFC read screenshot

8.8 How to get access to the BCC (BMS IC) safety library.

There is a full safety library for the Battery Cell Controller (BCC) IC. This library is only available under NDA and thus has not been added to this example code.

This software is available to selected customers via a required non-disclosure agreement (NDA). For additional information contact support or your local sales representative.

9. The parameters

9.1 How to configure the parameters via the CLI

At startup the saved parameters are read from the flash. If there is nothing saved, the CRC check will fail, and it will set the default parameters.
The parameters can be set with the CLI. use “help parameters” or look at the parameter list to see what parameter you would like to change.

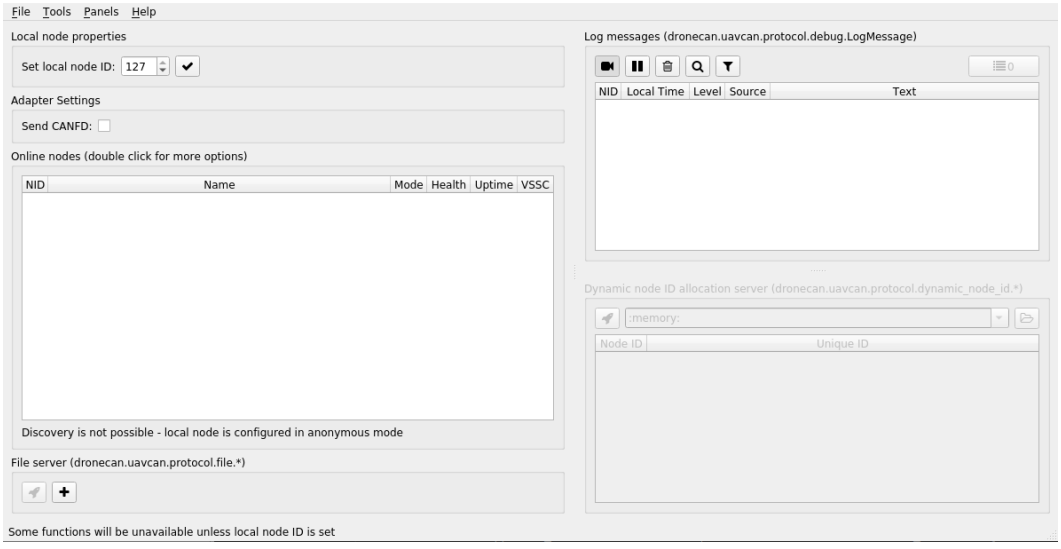
The parameter can be read with a “bms get <parameter>” command in the CLI, where <parameter> is the parameter **in full lower case**.

A parameter can be written with a “bms set <parameter> <x>” command in the CLI, where <parameter> is the to be written parameter and <x> is the new parameter value. Keep in mind that that this new value needs to be in the range of the to be written values. For decimals (float) use a “.” to separate it. Some parameters can’t be saved.

When you want to save a configuration to flash use “bms save” in the CLI, this will be loaded at startup or when the user types “bms load” in the CLI. If you want to restore the default values, type “bms default”. Check 9.3 to see which parameters need to be configured.

9.2 How to configure the parameters via the DroneCAN GUI tool

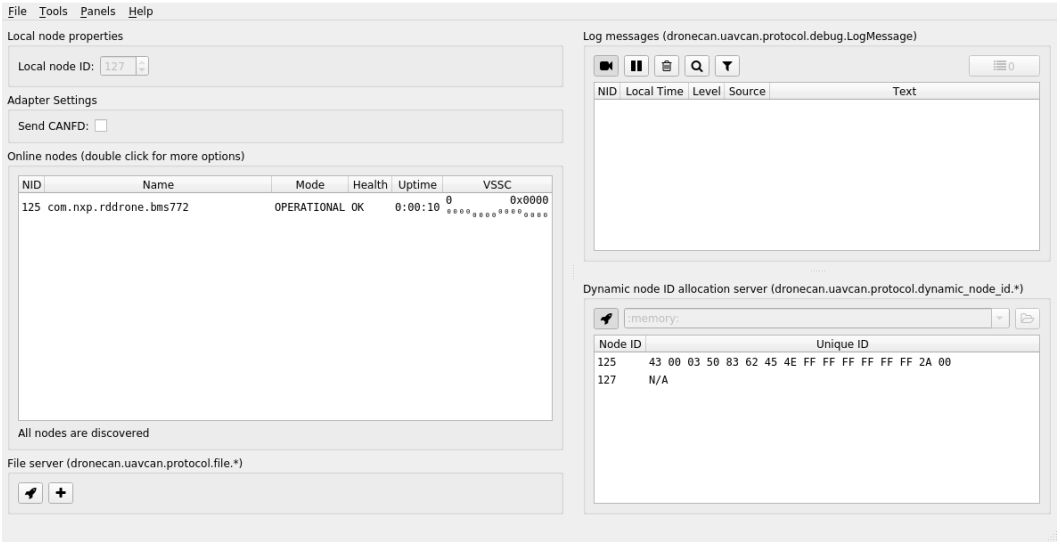
See chapter 8.4.1 to see how to set up the DroneCAN GUI tool and start the dynamic node allocation server.
After starting the DroneCAN GUI tool and using the correct CAN device (for example a PCAN).
You should see the following screen:



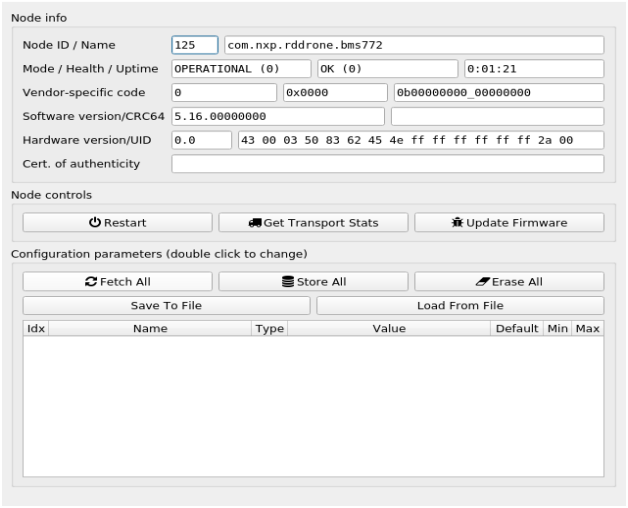
Set your local node ID (or keep the default), and press right next to it to enable the local node ID.

After setting your local node ID, you should be able to click on the rocket in the right bottom of the screen to start the dynamic node ID allocation server.

You should see the following screen:



Double click on the online node, com.nxp.rddrone.bms772 to get the node info.
After double clicking the node name, you should get the following view:



Click on “Fetch All” to get all the BMS parameters, plus an extra parameter to reset the BMS fault, called “BMS_RESET_FAULT”.

You can now click on the parameters and change them.

The “BMS_RESET_FAULT” parameter is used to reset the BMS when it is in a FAULT state, similar to the button or the “bms reset” CLI command.

“Store All” will save the parameters to flash, similar to the “bms save” CLI command.
“Erase All” will set all parameters to its default value, like “bms default”.
“Restart” will issue the NuttX reboot command and reboot the BMS application.

9.3 The parameter lists

9.3.1 The common battery variables list

Table 9. Common battery variables list

Parameter	Unit	Datatype	Description	Default	RO/ RW	No
v-out	V	float	The voltage of the BMS output	0	RO	0
v-batt	V	float	The voltage of the battery pack	0	RO	1
n-cells	-	uint8_t	Number of cells used in the BMS board	3	RW	2
v-cell1	V	float	The voltage of cell 1	0	RO	3
v-cell2	V	float	The voltage of cell 2	0	RO	4
v-cell3	V	float	The voltage of cell 3	0	RO	5
v-cell4	V	float	The voltage of cell 4	0	RO	6
v-cell5	V	float	The voltage of cell 5	0	RO	7
v-cell6	V	float	The voltage of cell 6	0	RO	8
i-batt	A	float	The last recorded current of the battery	0	RO	9
i-batt-avg	A	float	The average current since the last measurement (period t-meas (default 1s))	0	RO	10
i-batt-10s-avg	A	float	The 10s rolling average current, updated each t-meas, default 1000 (ms)	0	RO	11
sensor-enable	-	bool	This variable is used to enable or disable the battery temperature sensor, 0 is disabled, 1 is enabled	0	RW	12
c-batt	C	float	The temperature of the external battery temperature sensor	0	RO	13
c-afe	C	float	The temperature of the analog front end	0	RO	14
c-t	C	float	The temperature of the transistor (switch)	0	RO	15
c-r	C	float	The temperature of the sense resistor	0	RO	16

9.3.2 The calculated battery variables list

Table 10. Calculated battery variables list

Parameter	Unit	Datatype	Description	Default	RO/ RW	No
p-avg	W	float	Average power consumption over the last 10 seconds.	0	RO	17
e-used	Wh	float	Power consumption since device boot	0	RO	18
t-full	h	float	Charging is expected to complete in this time; zero if not charging.	0	RO	19
a-rem	Ah	float	Remaining capacity in the battery	0	RW	20
a-full	Ah	float	Full charge capacity, predicted battery capacity when it is fully charged. Decreases with aging.	4.6	RW	21
a-factory	Ah	float	Battery capacity stated by the factory	4.6	RW	22
s-charge	%	uint8_t	Percentage of the full charge 0 - 100%.	0	RO	23
s-health	%	uint8_t	Health of the battery in percentage, use STATE_OF_HEALTH_UNKNOWN = 127 if cannot be estimated.	127	RO	24
s-out	-	bool	This is true if the output power is enabled.	0	RO	25
s-in-flight	-	bool	This is true if the system is in flight (with flight-mode-enable and i-flight-mode)	0	RO	26
batt-id	-	uint8_t	Identifies the battery within this vehicle, 0 - primary battery.	0	RW	27

9.3.3 The additional battery variables list

Table 11. Additional battery variables list

Parameter	Unit	Datatype	Description	Default	RO/ RW	No
v-cell-ov	V	float	Battery maximum allowed voltage for one cell. exceeding this voltage, the BMS will go to fault mode	4.2	RW	28
v-cell-uv	V	float	Battery minimum allowed voltage for one cell. Going below this voltage, the BMS will go to fault mode followed by deepsleep after t-fault-timeout	3	RW	29
v-cell-nominal	V	float	Battery nominal voltage for one cell. will be used for energy calculation.	3.7	RW	30
v-storage	V	float	The voltage what is specified as storage voltage for a cell	3.8	RW	31
v-cell-margin	mV	uint8_t	Cell voltage charge margin to decide or not to go through another topping charge cycle	50	RW	32
v-recharge-margin	mV	Uint16_t	Cell voltage charge complete margin to decide or not to do a battery re-charge, to keep the cell voltages at max this much difference with the cell-ov	200	RW	33
i-peak-max	A	float	Maximum peak current threshold to open the switch during normal operation, can't be overruled	200	RW	34
i-out-max	A	float	Maximum current threshold to open the switch during normal operation, if not overruled	60	RW	35
i-out-nominal	A	float	Nominal discharge current (informative only)	60	RW	36
i-flight-mode	A	uint8_t	Current threshold to not disable the power in flight mode	5	RW	37
i-sleep-oc	mA	uint8_t	Overcurrent threshold detection in sleep mode that will wake up the BMS and also the threshold to detect the battery is not in use	30	RW	38
i-system	mA	uint8_t	Current of the BMS board itself, this is measured (as well) during charging, so this needs to be subtracted	40	RW	39
i-charge-max	A	float	Maximum current threshold to open the switch during charging	4.6	RW	40
i-charge-nominal	A	float	Nominal charge current (informative only)	4.6	RW	41
i-charge-full	mA	uint16_t	Current threshold to detect end of charge sequence	50	RW	42
c-cell-ot	C	float	Over temperature threshold for the cells. Going over this threshold and the BMS will go to FAULT mode	45	RW	43
c-cell-ut	C	float	Under temperature threshold for the cells. Going under this threshold and the BMS will go to FAULT mode	-20	RW	44
c-pcb-ot	C	float	PCB Ambient temperature over temperature threshold	45	RW	45
c-pcb-ut	C	float	PCB Ambient temperature under Temperature threshold	-20	RW	46
c-cell-ot-charge	C	float	Over temperature threshold for the cells During charging. Going over this threshold and the BMS will go to FAULT mode.	40	RW	47

c-cell-ut-charge	C	float	Under temperature threshold for the cells during charging. Going under this threshold during charging and the BMS will go to FAULT mode	0	RW	48
n-charges	-	uint16_t	The number of charges done	0	RW	49
n-charges-full	-	uint16_t	The number of complete charges	0	RW	50
ocv-slope	mV/A. min	float	The slope of the OCV curve. This will be used to calculate the balance time.	5.3	RW	51
battery-type	-	uint8_t	The type of battery attached to it. 0 = LiPo, 1 = LiFePO4, 2 = LiFeYPO4, 3 = NMC (LiPo type, LiNiMnCoO2), 4 = Na-ion (Sodium-ion, SIB). Could be extended. Will change OV, UV, v-storage, OCV/SoC table if changed runtime.	3	RW	52

9.3.4 The configuration variables list

Table 12. Configuration variables list

Parameter	Unit	Datatype	Description	Default	RO/ RW	No
t-meas	ms	uint16_t	Cycle of the battery to perform a complete battery measurement and SOC estimation can only be 10000 or a whole division of 10000 (For example: 5000, 1000, 500)	1000	RW	53
~~**t-ftti**~~	~~ms~~	~~uint16_t~~	~~Cycle of the battery to perform diagnostics (Fault Tolerant Time Interval)~~	~~1000~~	~~RW~~	~~54~~
t-bms-timeout	s	uint16_t	Timeout for the BMS to go to SLEEP mode when the battery is not used.	600	RW	55
t-fault-timeout	s	uint16_t	After this timeout, with an undervoltage fault the battery will go to DEEPSLEEP mode to preserve power. 0 sec is disabled.	60	RW	56
t-bcc-sleep-cyclic	s	uint8_t	Wake up cyclic timing of the AFE (after front end) during sleep mode	1	RW	57
t-sleep-timeout	h	uint8_t	When the BMS is in sleep mode for this period it will go to the self-discharge mode, 0 if disabled.	24	RW	58
t-ocv-cyclic0	s	int32_t	OCV measurement cyclic timer start (timer is increase by 50% at each cycle)	300	RW	59
t-ocv-cyclic1	s	int32_t	OCV measurement cyclic timer final value (limit)	86400	RW	60
t-charge-detect	s	uint8_t	During NORMAL mode, if the battery current is positive for more than this time, then the BMS will go to CHARGE mode	1	RW	61
t-cb-delay	s	uint8_t	Time for the cell balancing function to start after entering the CHARGE mode	120	RW	62
t-charge-relax	s	uint16_t	Relaxation time after the charge is complete before going to another charge round.	300	RW	63
batt-eol	%	uint8_t	Percentage at which the battery is end-of-life and shouldn't be used anymore Typically between 90%-50%	80	RW	64
s-flags	-	uint8_t	This contains the status flags as described in BMS_status_flags_t	255	RO	65

self-discharge-enable	-	bool	This variable is used to enable or disable the SELF_DISCHARGE state, 0 is disabled, 1 is enabled	1	RW	66
flight-mode-enable	-	bool	This variable is used to enable or disable flight mode, is used together with i-flight-mode. 0 is disabled	0	RW	67
emergency-button-enable	-	bool	This variable is used to enable or disable the emergency button on PTE8.	0	RW	68
smbus-enable	-	bool	This variable is used to enable or disable the SMBus update.	0	RW	69
gate-check-enable	-	bool	This variable is used to enable or disable the gate safety check. If true, it will check if it can be turned off, based on output voltage.	1	RW	70
model-id	-	uint64_t	Model id, set to 0 if not applicable	0	RW	71
model-name	-	char[32]	Battery model name, model name is a human-readable string that could include the vendor, model, chemistry.	"BMS772 "	RW	72

9.3.5 The can variables list

Table 13. Can variables list

Parameter	Unit	Datatype	Description	Default	RO/ RW	No
cyphal-node-static-id*	-	uint8_t	This is the node ID of the CYPHAL node. Should be between 1 - 127 or 255 for PNP.	255	RW	73
cyphal-es-sub-id*	-	uint16_t	This is the subject ID of the energy source CYPHAL message (1...100Hz)	4096	RW	74
cyphal-bs-sub-id*	-	uint16_t	This is the subject ID of the battery status CYPHAL message (1Hz)	4097	RW	75
cyphal-bp-sub-id*	-	uint16_t	This is the subject ID of the battery parameters CYPHAL message (0.2Hz)	4098	RW	76
cyphal-legacy-bi-sub-id*	-	uint16_t	This is the subject ID of the battery info legacy CYPHAL message (0.2 ~ 1Hz)	65535	RW	77
dronecan-node-static-id	-	uint8_t	This is the node ID of the DRONECAN node. Should be between 1 - 127 or 255 for dynamic node id.	0	RW	78
dronecan-bat-continuous	-	uint8_t	This indicates if the particular DroneCAN topic has to be published	0	RW	79
dronecan-bat-periodic	-	uint8_t	This indicates if the particular DroneCAN topic has to be published	0	RW	80
dronecan-bat-cells	-	uint8_t	This indicates if the particular DroneCAN topic has to be published	0	RW	81
dronecan-bat-info	-	uint8_t	This indicates if the particular DroneCAN topic has to be published	1	RW	82
dronecan-bat-info-aux	-	uint8_t	This indicates if the particular DroneCAN topic has to be published	1	RW	83
can-mode	-	char[32]	Options “OFF”, "DRONECAN" and "CYPHAL". To indicate which is used.	“OFF”	RW	84
can-fd-mode*	-	uint8_t	If true CANFD is used, otherwise classic CAN is used	0	RW	85
can-bitrate*	bit/s	int32_t	The bitrate of classical can or CAN FD arbitration bitrate	1000000	RW	86
can-fd-bitrate*	bit/s	int32_t	The bitrate of CAN FD data bitrate	4000000	RW	87

A ~~line~~ means this is not implemented yet.

* These parameters will only be implemented during startup of the BMS

9.3.6 The hardware parameters

Table 14. BMS hardware parameters list

Parameter	Unit	Datatype	Description	Default	RO/ RW	No
v-min	V	uint8_t	Minimum stack voltage for the BMS board to be fully functional	6	RW	88
v-max	V	uint8_t	Maximum stack voltage allowed by the BMS board	26	RW	89
i-range-max	A	uint16_t	Maximum current that can be measured by the BMS board	300	RW	90
i-max	A	uint8_t	Maximum DC current allowed in the BMS board (limited by power dissipation in the MOSFETs). For info only. Use i-out-max for a limit.	60	RW	91
i-short	A	uint16_t	Short circuit current threshold (typical: 550A, min: 500A, max: 600A)	500	RW	92
t-short	us	uint8_t	Blanking time for the short circuit detection	20	RW	93
i-bal	mA	uint8_t	Cell balancing current under 4.2V with cell balancing resistors of 82 ohms	50	RW	94
m-mass	kg	float	The total mass of the (smart) battery	0	RW	95
f-v-out-divider-factor	-	float	The factor of the output voltage divider as component tolerances could be different to not result in 11.0.	11.0	RW	96

A ~~line~~ means this is not implemented yet.

10. Charging

10.1 How to charge the battery using the BMS

The BMS cannot limit the current. It can only disconnect the battery when a fault occurs. To charge the BMS, connect a power supply that can **limit the current and the voltage**. Set this current limitation to the correct charge current and the voltage to the maximum battery pack voltage. The BMS will monitor the current and voltages. It can balance the cells by discharging cells having a higher voltage. See Main state machine explained for more information.

11. Software example guide – NuttX

11.1 Introduction

The NuttX software example of the BMS uses an RTOS named NuttX. NuttX is a real-time operating system (RTOS) with an emphasis on standards, compliance and small footprint. Scalable from 8-bit to 32-bit microcontroller environments, the primary governing standards in NuttX are POSIX and ANSI standards.

At startup the CLI will print the version number: this explanation is about bms6.0-11.0. the first number before the "–" is the BMS application version. The second number after the – is the NuttX version.

This chapter will first introduce the software block diagram and its components. Then the BMS application state machine running in the main loop and then the tasks with its priorities.

11.2 Software block diagram

In Figure 15 the software block diagram can be found. The BMS application consists of several tasks that run semi parallel (since it is still a single core processor). These tasks use functions from software components. The NuttX RTOS will take care of switching between tasks. The CLI part is called by calling the BMS application from the nuttshell interface with commands. The nuttshell is the serial (or UART) communication to the NuttX RTOS and can be used to communicate with the BMS application. The explanation of the blocks can be found in below in Component description.

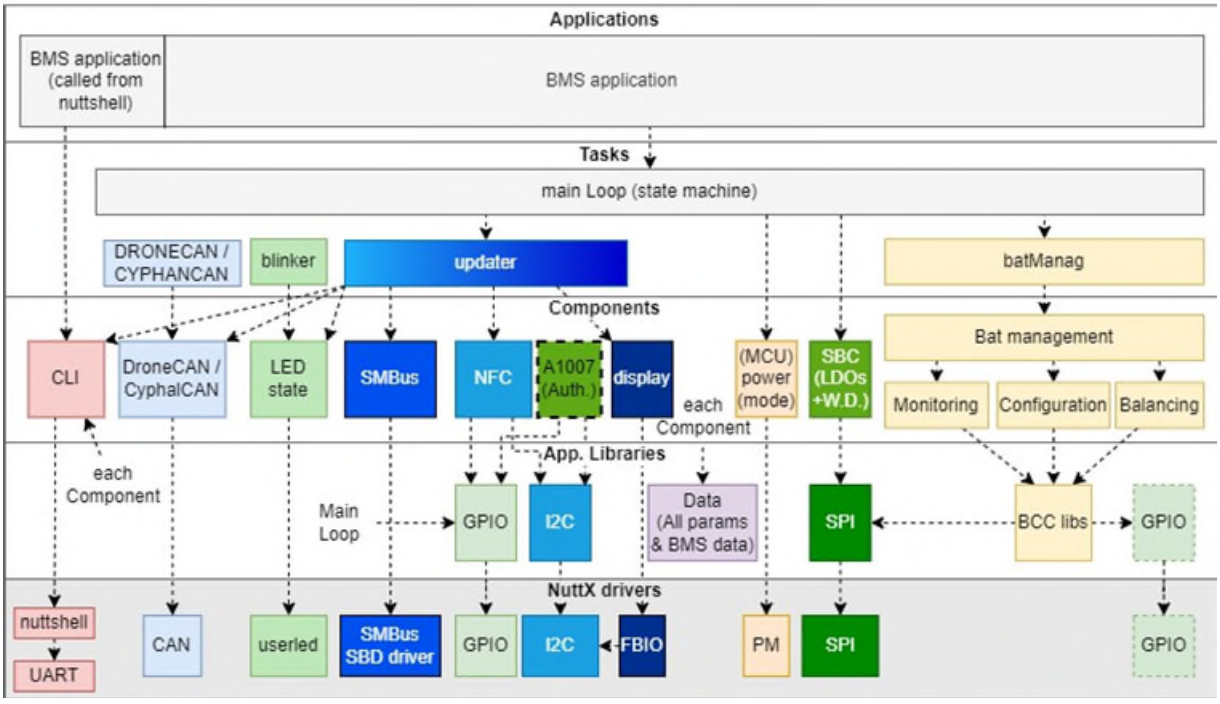


Figure 15: Software block diagram

11.2.1 Component description

CLI

The command line interface (CLI) component takes care of communication with the user through the NuttX nutshell, it can be used during debugging of the smart battery application or a specific battery under test. The communication is mapped to use a universal asynchronous receiver-transmitter (UART) also known as the root console. The CLI can output messages in colors if the ANSI escape sequences are enabled in the terminal.

The application command may be followed by optional arguments such as:

- sleep
- deepsleep
- wake
- reset
- help
- show
- set
- get

With the set or get command the user can read and write every value, including the configuration parameter list. These values can be read/written by calling the BMS application followed by a set or get command followed by the name of the variable. In the case of a set command this would instead be followed by the new value of the variable. Try the command “bms help” to see the help of the CLI. For more information on using the parameters, see chapter 9.1.

DroneCAN / CyphalCAN

There can only be one of these two CAN components active at the same time, either the DroneCAN component or the CyphalCAN component. This can be set via the parameter “can-mode”. After setting this, do a “bms save” and a “reboot” to apply the change.

The DroneCAN module manages the DroneCAN communication as the name already suggested. There are 2 parts of this communication, the synchronous and the asynchronous communication. For that reason, there is a task running to take care of both communications.
The synchronous communication is the updates of the different battery information messages. This is done with a certain update rate, depending on the message. If the BMS measurement interval is more than 1 second, it will influence this as well. These messages can then be sent regularly to the DroneCAN bus. Other devices connected to this bus can then read the messages, these devices could for example be an FMU/VMU for a drone or rover or other devices. This update with new data gets triggered via the updater task.

The asynchronous communication is when another DroneCAN device on the DroneCAN bus is requesting data. This can be battery data or a BMS parameter. Or another DroneCAN device would like to change a parameter of the BMS. When this happens, it is not triggered by the updater task, but the DroneCAN task will take care of this.

For the CAN communication, it will use the CAN PHY in the SBC (UJA1169).

If configured corrected, you could see the battery information in PX4, ArduPilot or QGroundControl.

CyphalCAN is supported as well, please see chapter 8.5.

LED state

The LED state module can be used to control the RGB LED. It can set an RGB color on or off and blink the LEDs at given intervals. If a LED needs to blink a blinker task will be used to ensure it blinks. This module is used to inform the user visually of various states and status.

This part is used implement the LED states of Table 15.

Table 15. LED states

State	LED state
Self-test	Red
Deep sleep	Off (after 1 second white LED on)
Sleep	Off
Wake-up	Green
Normal	Green blinking (with state indication) 1 blink 0-40% 2 blinks 40-60% 3 blinks 60-80% 4 blinks 80-100%
Fault_on (output power on)	Red
Fault_off (output power off)	Red blinking
Charging	Blue
Charging done	Green
Balancing/self-discharge	Blue blinking
Charger connected at startup	Red-blue blinking

SMBus

SMBus is an alternative way to communicate BMS information to a host device that has I^2C, like an FMU/VMU. The BMS could be seen as an I^2C peripheral device. Reading from specified BMS I^2C registers allows the device to read BMS data. Data like voltages, temperatures, state of charge, average current could all be read from these registers. The SMBus needs to be enabled with the variable “smbus-enable” to work.

NFC - NTAG5

The NFC module manages NFC communication. NFC communication can be used to read all kind of battery parameters. Via NFC a device should be able to read the values with a refresh rate of once a second. The updater task will be used to update the data in the NTAG5 NFC device. It can operate in a similar manner to a double ported EEPROM, and NFC records can include standardized messages for HTTP or text records. In this way the NFC tag could be updated regularly with status information. That information could be added to text message, and for example a smartphone would be capable of reading the message with data attached, this is done with minimal coding effort. This method removes the need for any custom software on the reading device.

Authentication - A1007

The authentication module will take care of the authentication using the A1007 chip. The A1007 is capable of secure asymmetric key exchange and storage as well as secure monotonic counters and flags for use in such things as counting charge or discharge cycles or permanently flagging under-voltage or over-temperature conditions.
This component is not implemented yet. Only IC presence verification via I^2C is implemented.

Display

The display module manages information presented on an optional local I^2C LCD display (e.g. SSD1306 type). The display should be connected to J23. Both 3.3V and 5V are supported in the software. There is a framebuffer which makes sure the data is transferred to the display. Information like SoC, SoH, output status, BMS mode, battery voltage, average current, temperature and ID can be found on the display. This makes sure the user can easily see the battery information it needs.

Power (MCU power mode)

The power component is used to control the MCU power modes. There are 3 BMS application power modes that are made which each mode for its own purpose.

- RUN mode
  - In this mode, the MCU power mode is set to RUN mode. The MCU will run on an 80MHz clock. All needed peripherals are enabled.
  - This is the usual mode of the MCU.
- STANDBY mode
  - In this mode, the MCU power mode is set to (Very Low Power Run) VLPR mode. The MCU will run on an 2MHz clock. At this low clock speed, the BMS is saving a lot of power.
  - Both SPI modules and the serial interface (CLI) is active to still have communication with the BCC, SBC for the WD and potentially a user.
  - This mode is a lot slower, but it saves a lot of power while still maintaining a connection to be able to measure as well.
  - One use case of this mode is during the CHARGE-RELAXATION state. You need to monitor the voltages, but this does not need to be a fast loop as the power to the charger is disconnected and thus you make sure the cells are not drained as much as in the RUN mode.
- VLPR mode
  - In this mode, the MCU power mode is set to (Very Low Power Run) VLPR mode. The MCU will run on an 2MHz clock. At this low clock speed, the BMS is saving a lot of power.
  - Only the SBC SPI module and the serial interface (CLI) is active to still have communication with the SBC for the WD and potentially a user.
  - This mode is a lot slower, but it saves the most power. There is no monitoring of the battery. The MCU relies on an GPIO interrupt pin from the BCC for waking up.
  - One use case of this mode is during the SLEEP state. No power is drawn, and you want to drain the battery as little as possible, while still reacting on a use command to for example configure the BMS. If current is drawn, the mode will change to NORMAL or CHARGE and the MCU mode will change as well.

SBC (LDOs + CAN + W.D.) - UJA1169

The SBC module manages the power of the voltage regulators in the SBC. With this module the SBC can be set in normal mode, standby mode and sleep mode. In the normal mode both V1 (powers the MCU and more “VCC_3V3_SBC”) and V2 (powers internal CAN PHY and 5V “VCC_5V_SBC”) are powered. In standby mode, V2 is off and in sleep mode both regulators V1 and V2 are off. The sleep mode is needed for the DEEP SLEEP state. The SBC provides an external watchdog as well. The main loop will “kick” (reset) the watchdog at the end of the loop.

Bat management - MC33772B

The Bat management (battery management) part is the most important part, it will oversee the whole battery management. It will be used to monitor the battery, the PCB (temperatures) and calculate voltages, temperatures, current, SoC, SoH, average power and more. It will ensure the BCC chip (MC33772B) reacts if thresholds are exceeded. Functions of this part can be used to drive the gate driver IC, which allows it to disconnect or connect the battery from the BMS output power connector. Because this is such a large part of the system, the Bat management part has its own tasks. This task can not only utilize the functions in the batmanagement component, but has a configuration component to configure the BCC, a monitoring component to monitor the battery data and a balancing component to take care of balancing. All these components can access the BCC libraries. These libraries can access the NuttX drivers like the timers and the GPIO (reset BCC pin). The batManag task will oversee the measurements and if triggered, it will do the calculations, check for errors, check for transition currents and handle the cell balancing / self-discharge. In the SW, the measured battery data and calculated battery data is put in two structs. It will save the newly made battery data and calculation data structs in the data part and trigger the updater task that there is new data available.

Data

Since different parts need to use the same data, a data library is made to take care of this. This library will make sure it is protected against usage at the “same” time by multiple tasks. If you set a parameter, it will make sure the system will act on it.

11.3 BMS application state machine

This chapter will show the designed state machine and the description of its different states

11.3.1 The main state machine

In Figure 16 the main state machine that will be implemented in the BMS can be found. This state diagram will be implemented in the BMS main loop.

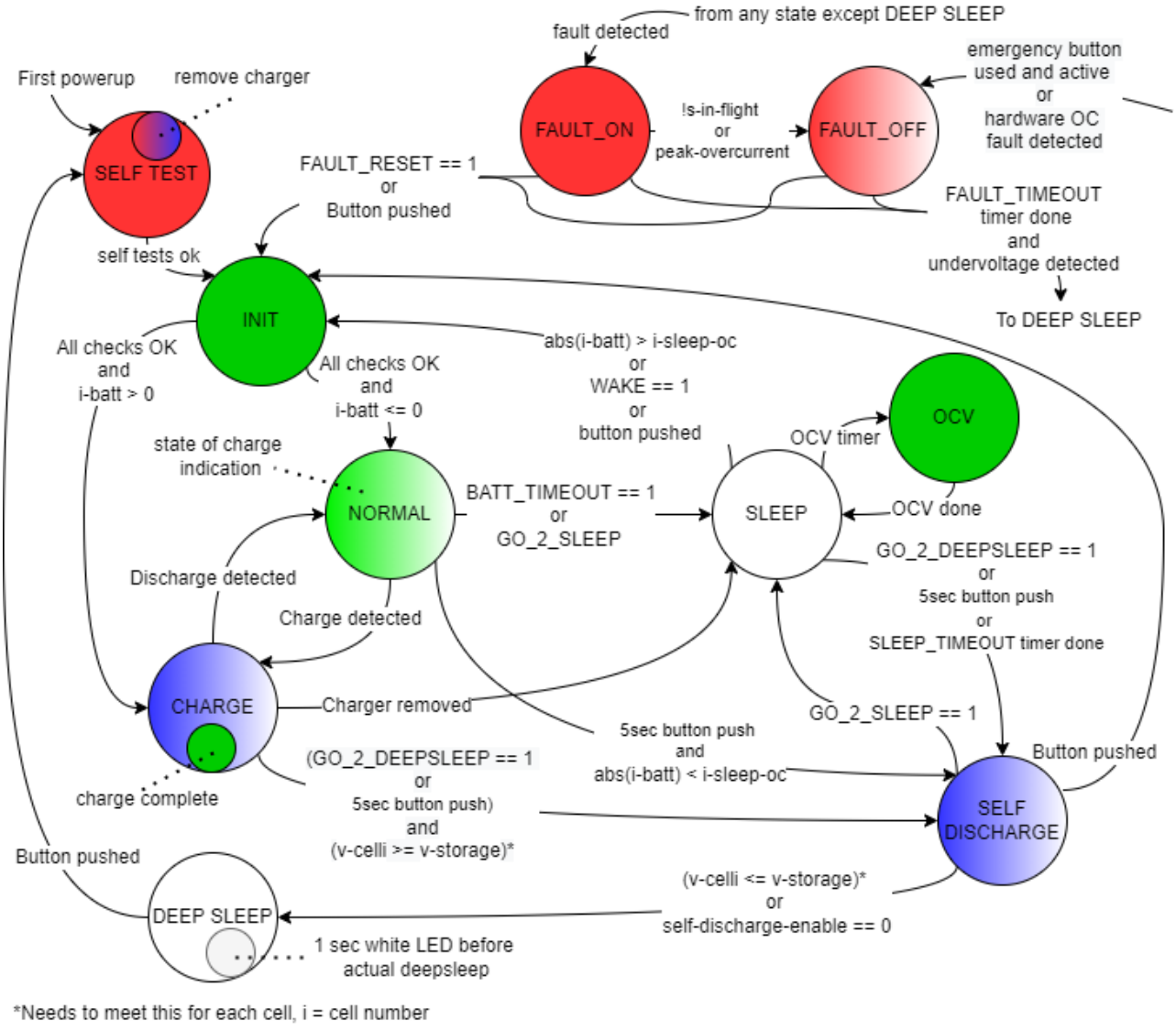


Figure 16: Battery main state machine

11.3.2 Main state machine explained

SELF TEST state

The SELF TEST state is entered at power-up of the microcontroller. In this state the microcontroller initializes everything and performs the self-test, for example check if communication with a component is possible or check if the set output can be read. If everything is OK, it will go to the INIT state. If a watchdog reset has occurred not every self-test will be done to make sure the power is not turned off. The LED will be solid red in this state.

INIT state

The INIT state is typically entered from the SLEEP state. In this state the microcontroller unit (MCU) will wake up and it will verify configurations, fault registers and functions. This is needed because it can enter the INIT state when the user resets from a fault in one of the FAULT states as well. When everything is OK, it will close the power switch if not already closed and proceed to the next state depending on the current direction. The LED will be steady green in this state.

NORMAL state

This is the state where the battery operates as intended, it is being discharged by the connected device, for example a drone. Meaning that the power switches are closed. The LED will be blinking green to indicate the state of charge. In this state the BMS performs the following tasks:

- Battery voltage, cell voltage and current is measured and calculated every measurement cycle.
- SoC and SoH are estimated every measurement cycle.
- Optionally: The CyphalCAN/DroneCAN BMS battery status will be send over the CAN bus every measurement cycle.
- The user can read the BMS status and parameters with NFC and the CLI. The user may change the state to SLEEP when the load is below the set threshold.
- A timer will monitor if the current is below the sleep current for more than the timeout period. If this happens, it will go to the SLEEP state automatically.
- It will monitor if the current flows into the battery and if the current is more than the sleep current for more than the charge detect time, the state will change to the CHARGE state.
- If the current is less than the sleep current while the button is pressed for 5 seconds, it will transition to the SELF DISCHARGE state.

CHARGE state

During this state the same functions as in the NORMAL state are implemented. The charging of the battery is done in different stages and is reflected in the charging state diagram in Figure 17. These are the states and their description:

- CHARGE START: in this state the charging will begin, and a timer will start. The LED will be blue to indicate charging. After a set time (default 120 sec) or if the voltage of one of the cells reaches the cell overvoltage level (to make sure there is no cell overvoltage error) the state will change to CHARGE WITH CB.
- CHARGE WITH CB: in this state the cell balancing (CB) function will be activated. This function will calculate the estimated cell balance minutes per cell, which is based on the cell voltages, the difference compared to the lowest cell voltage, the balance resistor and the OCV-slope. The formula to calculate this estimated balance time is:

$$\text{Estimated } balance\ minutes = (Vcell - Vcell_{min}) * \frac{Rbal}{Vcell * ocvslope}$$

  Other than this calculated time, the BMS will check if the voltage of a cell that is being balanced, has reached the desired voltage as well. When the voltage of one of the cells reaches the cell overvoltage level or the charging current is less than the charge complete current, it will go to the RELAXATION state. The LED will stay blue and will blink if cell balancing is active. Balancing is finished if all the calculated cell balance minutes are expired or it has reached the lowest cell voltage.
- RELAXATION: in this state the power switches are set open, disconnecting the charger from the battery. The MCU will be put in a very low power run (VLPR) mode (BMS application STANDBY mode), SBC in standby mode and the BCC to measure only at 10Hz. This will reduce the power in this state. The battery will relax for the specified relax time (default 300 sec). During this relaxing, the cells can still be balanced since this happens with a low balancing current. At the end of the relaxation period, the system will check whether the balancing is done. If balancing is not finished, the BMS will re-estimate the balance minutes. If balancing is finished and the highest cell voltage is lower than the cell overvoltage minus the voltage margin, it will return to the CHARGE WITH CB state to continue the charge process. If the highest cell is within this margin, the charging is complete, and it will go to the CHARGE COMPLETE state. To make sure it won't endlessly go through this cycle with the CHARGE WITH CB state (this can happen if the end of charge current is met but the voltage requirement is not met), after 5 times it will not check if the highest cell voltage is within this margin and will just go to the CHARGE COMPLETE state.
- CHARGE COMPLETE: in this state, the charging is done, and the LED will be steady green. If the lowest cell voltage decreases again below the cell overvoltage minus the recharge voltage margin, it will go to the CHARGE WITH CB state again. The power switches will remain open and if the charger is disconnected it will go to the SLEEP state after the defined period of time.

If at any time the current flows from the battery to the output and this current is higher than the sleep current, the BMS transitions to the NORMAL mode. If a charger is disconnected, the state will transition to the SLEEP state. If the go to deep sleep command has been given or the button is pressed for five seconds there are two options: If one cell voltage is less than the storage voltage it will complete charging until each cell has reached the storage voltage, after this is done the BMS will transition to the SELF DISCHARGE state and this will then transition to the DEEP SLEEP state. The other option is that no cell voltage is less than the storage voltage, than the BMS will transition to the SELF DISCHARGE state.

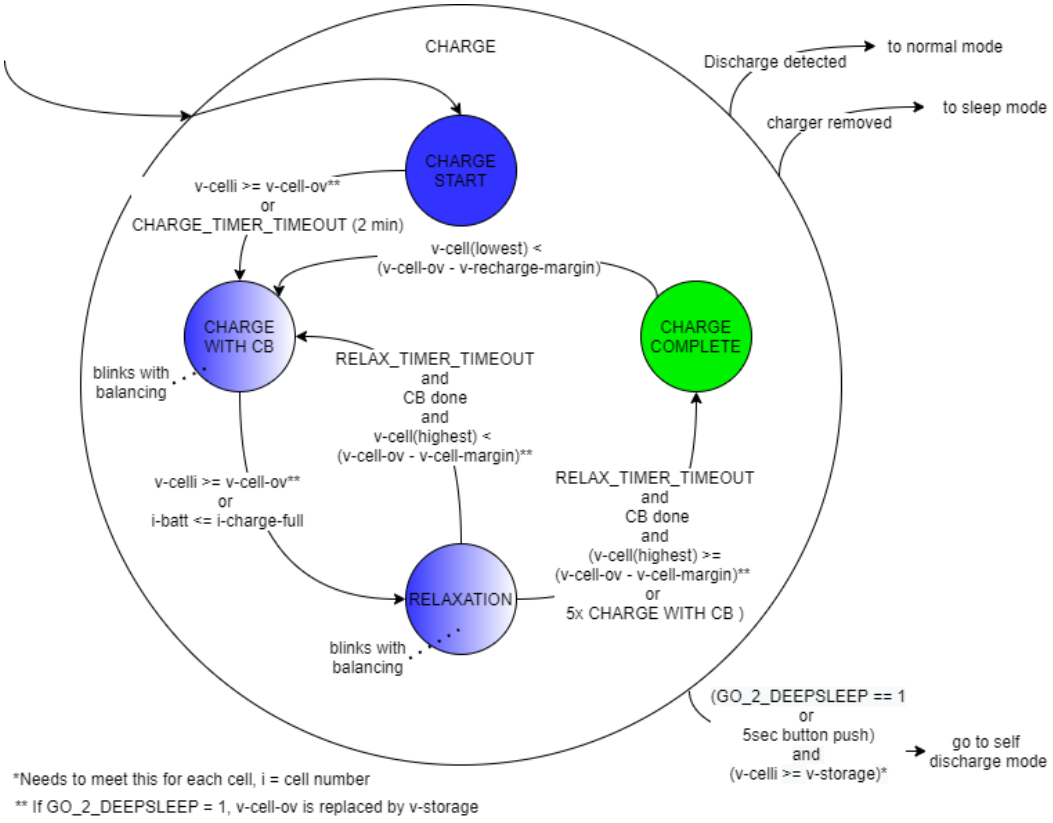


Figure 17: Charging state diagram

SLEEP state

The sleep state is typically entered when the current is very low for an amount of time. This SLEEP state is used to preserve power. The MCU will be put in VLPR mode, the SBC in standby mode, the BCC in sleep mode with measurement on to check for a sleep overcurrent and other faults. This is the BMS application VLPR mode. The power switches will be closed to make sure the battery could be used. If any threshold is met during a cyclic measurement or the button is pressed, it will wake the MCU and the BMS will transition to the INIT state to check status. If the button is pressed for five seconds, the state will change to the SELF DISCHARGE state, in order to go to the DEEP SLEEP state. If the t-sleep-timeout happens the BMS will go to the SELF DISCHARGE state and will discharge the battery to storage level. After this it will go to the DEEP SLEEP state. In this state the LED will be off.

OCV state

The OCV state will allow to record the latest open cell voltage (OCV) of the battery which is used in the state of charge (SoC) computation. Cyclically the Battery will enter this mode when the Battery stays in the SLEEP state. The period the system will go from the SLEEP state to the OCV state will depend on the time since the battery has entered the SLEEP state in the first place, without going to another state except the OCV state. The time to enter the OCV state will gradually increase each time with 50% from the set begin time until the set end time is reached. If for example the set begin time is five minutes and the set end time is twenty-four hours, it will take fifteen times to have a period of is twenty-four hours. When entering this mode, the MCU will wake up the AFE and measure. After it has calibrated the SoC, it will go to the SLEEP state again. The LED will blink green in the OCV state.

FAULT_ON state
The FAULT_ON state is entered when a critical fault has been detected (over-current, over-voltage, cell over-temperature) and requires evaluating if the battery needs to disable the output power. In this state the power will remain ON. With the flight-mode-enable and the i-flight-mode parameter, the user can make sure that the battery will not be disconnected from the power out connector and the BMS will stay in this state. If the s-in-flight parameter is high, it will stay in this state and not disconnect the power. Except, when the i-peak-max threshold is reached.
The s-in-flight parameter is high if the i-batt-avg (1s) is higher than i-flight-mode and the i-batt-avg (1s) is less than the i-out-max AND the flight-mode-enable is high.
The s-in-flight will be low again when: the i-batt-10s-avg is lower than the i-flight-mode and the i-batt-avg (1s) is less than the i-flight-mode. OR s-in-flight will be low when the flight-mode-enable is set to 0 OR when there is a i-batt-avg (1s) charge current higher than the i-sleep-oc ((1s_current_avg - i-sleep-oc) > 0). If the BMS is in the FAULT_ON state and s-in-flight will become false, it will go to the FAULT_OFF state.
If the BMS needs to turn off the power, it will go to the FAULT_OFF mode. Other ways to exit this FAULT_ON state is that the user can manually force the BMS to go to the INIT state via the reset fault command with the CLI or by activating the push button. If there is an undervoltage fault, it could transition to the DEEP SLEEP state after the t-fault-timeout time. In this state the LED will be solid RED to indicate the power is still on.

FAULT_OFF state
The FAULT_OFF state is entered when there is a fault and it needs to turn off the output power. In this state the power will be turned OFF. Usually, this state is entered because in the FAULT_ON state it noticed that it needs to turn off the power. It will go directly in the FAULT_OFF state if the emergency button is used and this button is active or if a hardware overcurrent is detected. To exit this FAULT_OFF state is that the user can manually force the BMS to go to the INIT state via the reset fault command with the CLI or by activating the push button. If there is an undervoltage fault, it could transition to the DEEP SLEEP state after the t-fault-timeout time. In this state the LED will be blinking RED to indicate the power is off.

SELF DISCHARGE state
This state is used to discharge the cells to the cell storage voltage in order to improve its life duration, when storing the battery for long time. In this mode, the power switches are open, the MCU is powered and the balancing functions are activated. When the storage voltage is reached for each cell or if cells have a lower voltage, it will transition to the DEEP SLEEP state. CAN communication is disabled. To get a better SoC estimation, the OCV is measured and this will update the SoC measurement of the battery. To exit this state and to go back to the INIT state, the button needs to be pressed. The LED will blink blue in this state.

DEEP SLEEP state
This state is used for transportation and storage. In this state, the power switches are open, disconnecting the battery, all protections are turned off, there are no cyclic measurements done, the LED is off, and it will set everything to sleep or off to ensure the lowest power usage (<100uA). Only the button can wake everything in this state. When the button is pressed, it will transition to the SELF_TEST state. When entering this mode, the MCU will check if at least one configuration parameter has changed. If configuration parameters have changed, it will save the parameters to flash to make sure they are loaded at startup. The LED will be white for 1 second before going off for the rest of the time in this state.

11.4 Tasks priorities

The tasks of the BMS have different priorities, this is needed because some tasks/activities are more important than others. For example, one needs to react fast on a fault, so the system needs to prioritize a fault above blinking the LED. The BMS uses the NuttX preemption, which means that lower priority tasks get interrupted by a higher priority task. The task priorities can be seen in Figure 18.

When one of the BCC functions is called (via the spiwrapper), the task is locked so it can't switch tasks. This way it makes sure it will execute the function OK, otherwise there could be CRC errors or NULL responses.

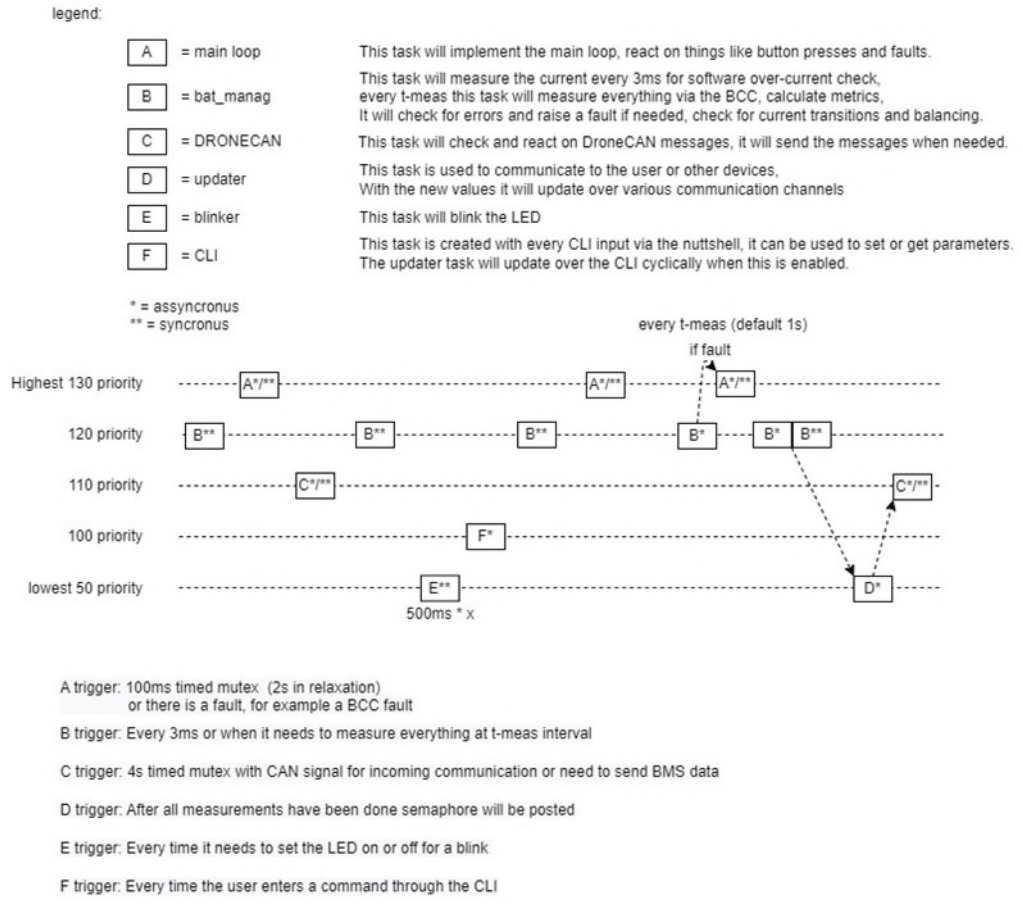


Figure 18: BMS task priorities

11.5 Realization

This chapter will show how the realization has been done. It will use diagrams to show how some of the parts were designed and it will describe each part in more detail. This is about the following code repository: https://github.com/NXPHoverGames/RDDRONE-BMS772

11.5.1 Main

In the main source file, the BMS main can be found, this is the BMS application. This function will initialize each part and start the main loop task. This task will implement the battery main state machine as seen in Figure 16. In the main source code, the state is changed. In this source file there is a function to handle a changed parameter as well. This function will call other functions from the needed parts to do something with the parameter that is changed. If for example a configuration changed, such that a configuration of the BCC needs to change, it will call the right function from the Bat management part to change the configuration of the BCC as well. At the end of the main loop, the watchdog in the SBC is kicked. If this is not done in time, it will reset the MCU.

11.5.2 Data

There is a lot of data that is needed or set by different tasks. Because it is not wise to move this big chunk of data through all the tasks there needs to be some sort of shared memory. Because NuttX is POSIX compliance there are shared memory functions that could be used. But for these shared memory functions a memory management unit (MMU) is needed and this microcontroller does not have an MMU. That is why the whole data management will be made in a data source file. This makes sure the data is only made once and is not global. With functions the data can be read or written, and these functions ensures protection against multiple threads accessing the data at the same time. These functions can be seen in Figure 20 and Figure 19.

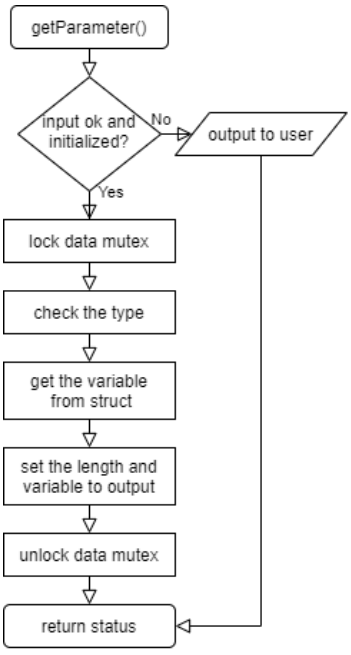


Figure 19: Get parameter

To protect the data from multiple threads trying to access it at the same time, a mutex is used. A mutex is an object that can be locked and unlocked in an atomic operation. Meaning that if both threads want to lock the mutex, the threads cannot lock the same mutex at the same time. A mutex is needed to prevent data race. The other thread needs to wait until the mutex is available.

The big data chuck is in a struct, together with a parameter info array. This array supports a fast access of the data type, the minimum, the maximum and the address of the data. This ensures it is faster to get and set data than with a large switch.

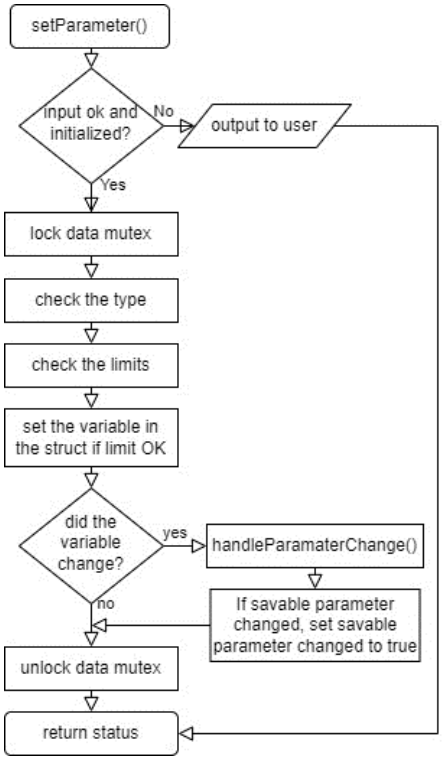


Figure 20: Set parameter flowchart

If a variable is set with the set parameter function. It will check if the variable is changed. If the variable is changed, it will call a handle parameter change function. This function will check which parameter has changed and will set other parameters if needed. At the end there is a callback to the main which will make sure the correct functions / reactions are being called. There are parameters that have an effect on the battery management and thus a function that handles changes in the battery management part is called as well to react on the parameter that changed. Next to handling the change of a parameter, this function will check if one of the savable parameters has been changed. If this is the case, it will remember this to save the parameters to flash when the DEEP SLEEP state is executed.

11.5.3 CLI

In NuttX there is a nuttshell, this is the UART communication with the MCU. In this nuttshell, applications can be called with or without arguments. There arguments will be given to the function it calls, in this case the BMS main. This means that a CLI can be created with calling the application with some arguments.

This CLI that is made, can be used by calling the BMS application in the nuttshell with a command and optionally up to 2 arguments for that command. When this happens the BMS main is called. Meaning that this main needs to be resistant against multiple calls, this should not restart the BMS application because than the battery power will be cut.

If there are commands given when calling the BMS application, the CLI process commands function will be called to handle it. It will parse the command and optionally the arguments and check if the inputs are valid. If it is valid, it will act based on which command has been given. The flowchart can be seen in Figure 21.

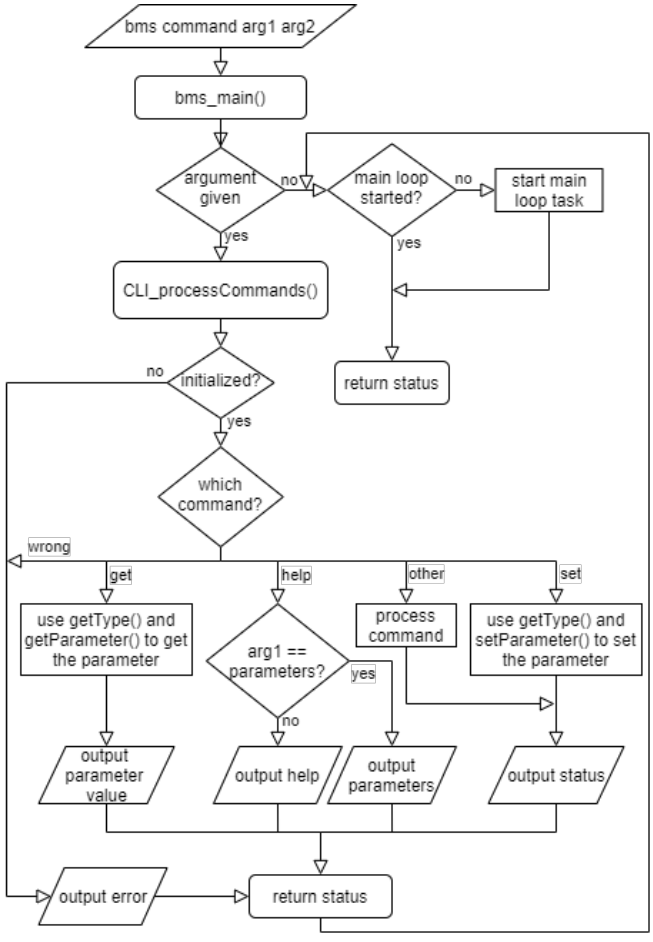


Figure 21: CLI flowchart

11.5.4 LED state

In order to set a color to the RGB LED, the set led color function should be used. The flowchart of this function can be found in Figure 22. Because this function can be used from different tasks, a mutex will be locked before it checks if the color and if blink is already set. This function will set the semaphore to start or stop the blink sequence. It will skip the semaphore timed wait function to ensure the blink sequence restarts if needed. It will begin with the new color. This function will use the NuttX userled functions.

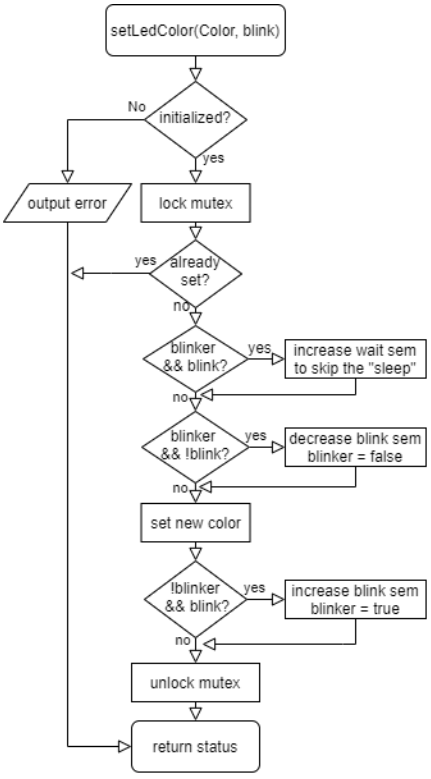


Figure 22: Set LED color flowchart

11.5.5 GPIO

In order to use a GPIO in NuttX, these GPIO's need to be defined in the board file and the board specific GPIO file. This will create devices for each GPIO pin. To use the GPIO in the application an IOCTL call need to be used. IOCTL means input-output control and it is a device specific system call.

The IOCTL is used to give commands to a driver to control a device, in this case the GPIO pins. But for an IOCTL to work, an open file descriptor needs to be given (Linux man-pages, n.d.). This is obtained by giving the path to the device as a string. This is too much work to do in the application for setting or reading a GPIO, that is why a GPIO BMS application driver is made. This will make sure that a GPIO can be read or written with simple write/read pin functions and a define to indicate the pin. An input pin can be an interrupt pin. On an interrupt it will generate a signal that will queue the action for the handler. Keep in mind that these signals can be very intrusive.

11.5.6 Bat management

The Bat management part can be used to monitor the battery and control the power switch. Because the Bat management part is quite large, there are other source files made to help with functions it needs to do and with utilizing the BCC. Like monitoring, to take care of measurements. Configuration, to take care of the whole configuration for the BCC. Balancing, to add easy to implement balancing function. For the main to implement the state machine, functions are made to let the main implement the functionality. Some functions are made to enable the measurements, to check for faults, to self-discharge etc.

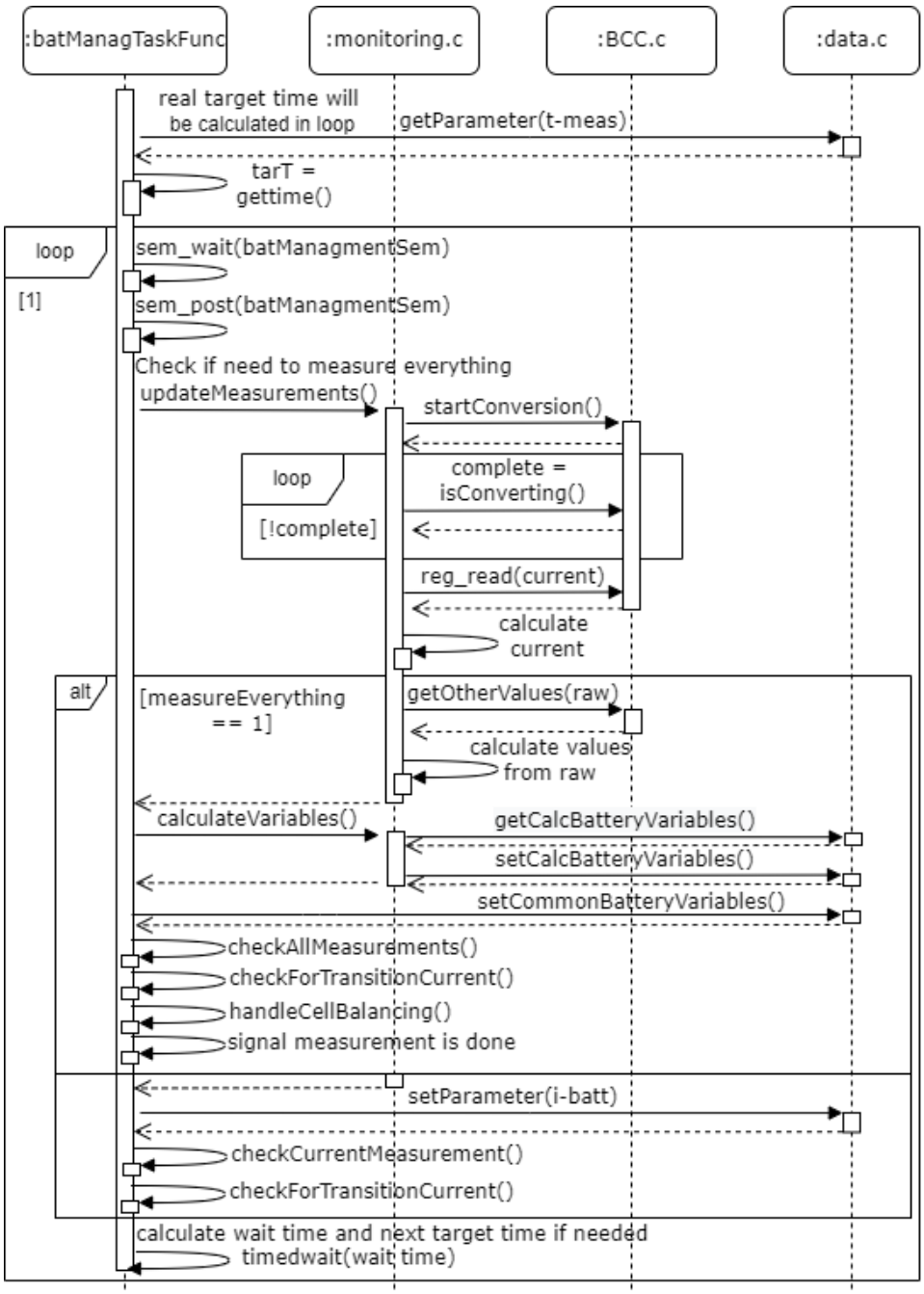


Figure 23: Calculate other values sequence diagram

The batManag task is created to handle the battery management. It will start the manual measurements with the BCC. Mostly it will calculate and check the current or it will measure and calculate every variable at t-meas interval. It will then perform a measurement, calculate the variables, save the values, it will check for software faults, it will calculate the new transition variables, handle the cell balancing if needed and it will trigger the callback that new measurements have been done. When saving the data, it will save the new measured and calculated variables in one go using the structs. Because the BCC will not check for an overcurrent, the current needs to be read and calculated every time to be compared with the current threshold. For short circuit protection there is a hardware circuit. If it measures a fault, it will trigger the main to act on it.

The sequence of the batManag task can be seen in Figure 23. The sem_wait and sem_post functions are called consecutively in the endless loop, this is used to start and stop the task with a function from the main. The meas task will check if the next measurement needs to be the measure everything or just the current and calculate the wait time to make sure the t-meas interval is met or with the remaining wait time. The task will wait for the next time or when the measurement is enabled, it will measure right away. To make sure the cyclic measurements does not drift, the time before the loop starts and the period t-meas are gained. The target time is calculated by adding the period with the previous target time. If t-meas should change, this is updated in the sequence using a global variable. This is left out of the sequence diagram because it is too detailed.

To calculate the state of charge (s-charge), the coulomb counter is used. The coulomb counter register holds the sum of the measured currents (until read). There is another register that holds the number of samples in the coulomb counter register. The average current is calculated by dividing the sum of the currents by the number of samples. When the time is known for which the average current is calculated, the difference in charge can be calculated with the following formula: $\Delta Q = I_{avg} * \Delta t$. The new remaining charge is calculated by adding the difference in charge with the old remaining charge. The state of charge can then be calculated by dividing the full charge capacity by the remaining charge.

In order to provide the average power consumption over a time period of ten seconds, a constant moving average is taken. This moving average is constructed by removing the oldest measurement and adding the new measurement, which is than divided by the amount of measurements. This way the average will only be of the last ten seconds. In order to be memory efficient, the measurements used in the moving average will be sub-sampled if the measurement period is configured as less than one second. This way maximum ten old measurements need to be known. Measurements are not lost when sub-sampling, because the BCC chip will remember an average of it.

The batManag task will check for software measured faults. The BCC chip will take care of hardware fault monitoring for the overvoltage, undervoltage, over temperature and under temperature. It will set the fault pin high when there is an error. If this happens it will trigger an interrupt in the main and it will check what fault happened. The main can than act on the fault. This ensures that the main is in control of what happens.

Since the user can change configurations in run time, sometimes a configuration needs to be changed in the BCC as well. When there is a change in the configuration, this is set with the setParameter function and a task in the main source file will handle the change. This function will call a function to handle the change in the bat management part. In this part it will call the right function from the configuration source file to change the configuration of the BCC.

11.5.7 SBC

The SBC part is used to control the power of voltage regulators V1 (The most used 3.3V (VCC_3V3_SBC)) and V2 (CAN PHY) (VCC_5V_SBC)). With the setSbcMode() function the mode of the SBC can be set. In the normal mode both V1 and V2 are active, in the standby mode V2 is off, turning off the CAN transceiver and in the sleep mode both V1 and V2 are off, turning off almost the whole BMS board. In Figure 25 the simplified flowchart of this function can be seen. Besides power regulators, the SBC has a watchdog, which is used to reset the MCU if it doesn't "kick" (reset) the watchdog within the set time. The reset of the MCU is done via the NRST pin.

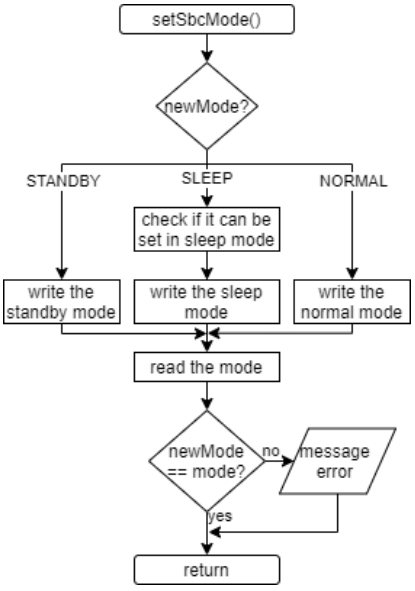


Figure 24: Set SBC mode flowchart

11.5.8 DroneCAN or CyphalCAN

This part works with a DroneCAN (or Cyphal) task that waits (it sleeps until a CAN transceiver signal comes in) for an incoming CAN transmission. It will also wake up if the main signals the task that new data needs to be send. When new data needs to be sent, the task will put the data that needs to be sent in the transmit buffer. It will check if the transmit buffer if it is filled and transmit the data if it is. Then it will wait for an incoming CAN transmission or signal again. To see the flowchart see Figure 26: DroneCAN flowchart. There are DroneCAN and Cyphal messages implemented. Keep in mind that you can only select one at the time. The DroneCAN messages can be seen below in chapter 11.5.8.1. The CyphalCAN messages are a snapshot of the CyphalCAN V1 with WIP DS-015. This consists of 3 messages, the energy source, the battery status and the battery parameters. These messages can be seen in 11.5.8.2.

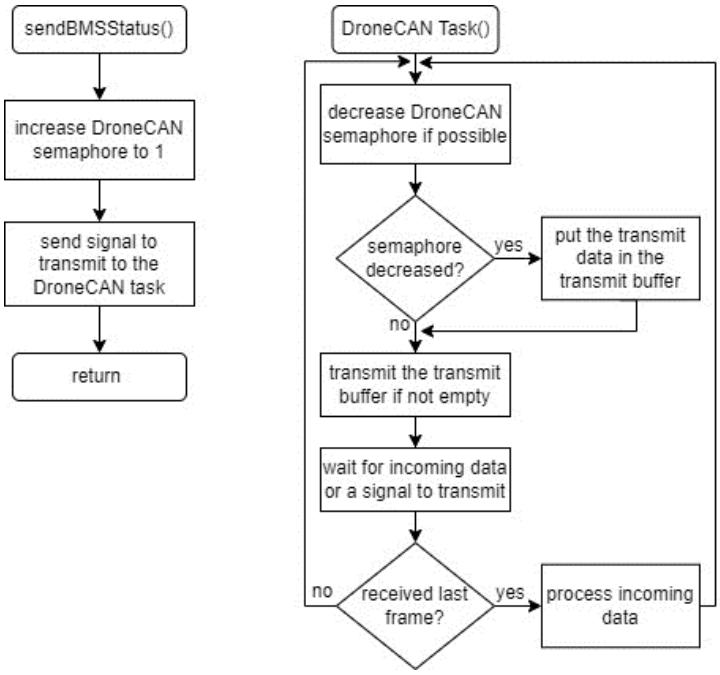


Figure 25: DroneCAN flowchart

11.5.8.1 DroneCAN messages implemented

For information on the DSDL message definition used for DroneCAN, please see the .uavcan files:

- ardupilot_equipment_power_BatteryContinuous
  - DSDL/ardupilot/equipment/power/20010.BatteryContinuous.uavcan at master · dronecan/DSDL · GitHub
- uavcan_equipment_power_BatteryInfo
  - DSDL/uavcan/equipment/power/1092.BatteryInfo.uavcan at master · dronecan/DSDL · GitHub
- ardupilot_equipment_power_BatteryPeriodic
  - DSDL/ardupilot/equipment/power/20011.BatteryPeriodic.uavcan at master · dronecan/DSDL · GitHub
- ardupilot_equipment_power_BatteryCells
  - DSDL/ardupilot/equipment/power/20012.BatteryCells.uavcan at master · dronecan/DSDL · GitHub
- ardupilot_equipment_power_BatteryInfoAux
  - DSDL/ardupilot/equipment/power/20004.BatteryInfoAux.uavcan at master · dronecan/DSDL · GitHub

Table 16. DroneCAN battery continuous message

Type	Name	Unit	Description
Float16	Temperature_cells	C	Pack mounted thermistor (preferably installed between cells), NAN: field not provided
Float16	Temperature_pcb	C	Battery PCB temperature (likely output FET(s) or current sense resistor), NAN: field not provided.
Float16	Temperature_other	C	Application dependent, NAN: field not provided
Float32	Current	A	Positive: defined as a discharge current. Negative: defined as a charging current, NAN: field not provided
Float32	Voltage	V	Battery voltage
Float16	State_of_charge	%	The estimated state of charge, in percent remaining (0 - 100).
Uint8	Slot_id	-	The physical location of the battery on the aircraft. 0: field not provided
Float32	Capacity_consumed	Ah	This is either the consumption since power-on or since the battery was full, depending on the value of STATUS_FLAG_CAPACITY_RELATIVE_TO_FULL, NAN: field not provided
Uint32	Status_flags	-	Fault, health, readiness, and other status indications. READY_TO_USE = 1 CHARGING = 2 CELL_BALANCING = 4 ~~FAULT_CELL_IMBALANCE = 8~~ AUTO_DISCHARGING = 16 REQUIRES_SERVICE = 32 BAD_BATTERY = 64 PROTECTIONS_ENABLED = 128 ~~FAULT_PROTECTION_SYSTEM = 256~~ FAULT_OVER_VOLT = 512 FAULT_UNDER_VOLT = 1024 FAULT_OVER_TEMP = 2048 FAULT_UNDER_TEMP = 4096 FAULT_OVER_CURRENT = 8192 FAULT_SHORT_CIRCUIT = 16384 ~~FAULT_INCOMPATIBLE_VOLTAGE = 32768~~ ~~FAULT_INCOMPATIBLE_FIRMWARE = 65536~~

	~~FAULT_INCOMPATIBLE_CELLS_CONFIGURATION = 131072~~ CAPACITY_RELATIVE_TO_FULL = 262144

Table 17. DroneCAN battery info message

Type	Name	Unit	Description
Float16	Temperature	K	
Float16	Voltage	V	
Float16	Current	A	
Float16	Average_power_10sec	W	Average power consumption over the last 10 seconds.
Float16	remaining_capacity_wh	Wh	Will be increasing during charging
Float16	full_charge_capacity_wh	Wh	Predicted battery capacity when it is fully charged. Falls with aging
Float16	hours_to_full_charge	h	Charging is expected to complete in this time; zero if not charging

Uint11	status_flags	-	- CHARGING must be always set as long as the battery is connected to a charger, even if the charging is complete.
			- CHARGED must be cleared immediately when the charger is disconnected.
			STATUS_FLAG_IN_USE = 1 # The battery is currently used as a power supply
			STATUS_FLAG_CHARGING = 2 # Charger is active
			STATUS_FLAG_CHARGED = 4 # Charging complete, but the charger is still active
			STATUS_FLAG_TEMP_HOT = 8 # Battery temperature is above normal
			STATUS_FLAG_TEMP_COLD = 16 # Battery temperature is below normal
			STATUS_FLAG_OVERLOAD = 32 # Safe operating area violation
			STATUS_FLAG_BAD_BATTERY = 64 # This battery should not be used anymore (e.g. low SOH)
			STATUS_FLAG_NEED_SERVICE = 128 # This battery requires maintenance (e.g. balancing, full recharge)
			STATUS_FLAG_BMS_ERROR = 256 # Battery management system/controller error, smart battery interface error
			STATUS_FLAG_RESERVED_A = 512 # Keep zero
			STATUS_FLAG_RESERVED_B = 1024 # Keep zero

Uint7	state_of_health_pct	%	Health of the battery, in percent, optional. STATE_OF_HEALTH_UNKNOWN = 127 # Use this constant if SOH cannot be estimated.
Uint7	state_of_charge_pct	%	Relative State of Charge (SOC) estimate, in percent. Percent of the full charge [0, 100]. This field is required.
Uint7	state_of_charge_pct_stdev	%	SOC error standard deviation; use best guess if unknown.
Uint8	battery_id	-	Identifies the battery within this vehicle, e.g. 0 - primary battery.
Uint32	model_instance_id	-	Set to zero if not applicable. Model instance ID must be unique within the same battery model name.
Uint8[<32]	model_name	-	Battery model name. Model name is a human-readable string that normally should include the vendor name, model name, and chemistry type of this battery. This field should be assumed case-insensitive. Example: "Zubax Smart Battery v1.1 LiPo".

Typical publishing rate should be around 0.2~1 Hz.

Table 18. DroneCAN battery periodic message

Type	Name	Unit	Description
Uint8[<=50]	Name	-	Formatted as manufacturer_product, 0 terminated
Uint8[<=32]	Serial_number	-	Serial number in ASCII characters, 0 terminated
~~Uint8[<=9]~~	~~Manufacturer_date~~	~~-~~	~~Manufacture date (DDMMYYYY) in ASCII characters, 0 terminated~~
Float32	Design_capacity		Fully charged design capacity. 0: field not provided.
Uint8	Cells_in_series	-	Number of battery cells in series. 0: field not provided.
Float16	Nominal_voltage	V	Battery nominal voltage. Used for conversion between Wh and Ah. 0: field not provided.
Float16	Discharge_minimum_voltage	V	Minimum per-cell voltage when discharging. 0: field not provided.
Float16	charging_minimum_voltage	V	Minimum per-cell voltage when charging. 0: field not provided.
Float16	charging_maximum_voltage	V	Maximum per-cell voltage when charged. 0: field not provided.
Float32	charging_maximum_current	A	Maximum pack continuous charge current. 0: field not provided.
Float32	discharge_maximum_current	A	Maximum pack continuous discharge current. 0: field not provided.
Float32	discharge_maximum_burst_current	A	Maximum pack discharge burst current for 30 seconds. 0: field not provided
Float32	full_charge_capacity	Ah	Predicted battery capacity when fully charged (accounting for battery degradation), NAN: field not provided
Uint16	cycle_count	-	Lifetime count of the number of charge/discharge cycles (https://en.wikipedia.org/wiki/Charge_cycle). UINT16_MAX: field not provided.
Uint8	state_of_health	%	State of Health (SOH) estimate, in percent (0 - 100). UINT8_MAX: field not provided.

Battery data to be sent statically upon request or periodically at a low rate
Recommend that this message is sent at a maximum of 1Hz and nominally 0.2 Hz (IE: once every 5 seconds.).

Table 19. DroneCAN battery cells message

Type	Name	Unit	Description
Float16 [<=24]	Voltages	V	Cell voltage array.
Uint16	index	-	Index of the first cell in the array, index 0 is cells at array indices 0 - 23, index 24 is cells at array indices 24 - 47, etc.

Rate: set by parameter on smart battery (default off).

Table 20. DroneCAN battery info auxiliary message

Type	Name	Unit	Description
~~uavcan.Timestamp~~	~~timestamp~~	~~-~~	~~timestamp~~
Float16 [<=255]	voltage_cell	V	Battery individual cell voltages, length of following field also used as cell count.
Uint16	cycle_count	-	
~~Uint16~~	~~over_discharge_count~~	~~-~~	~~Number of times the battery was discharged over the rated capacity.~~
~~Float16~~	~~max_current~~	~~A~~	~~Max instantaneous current draw since last message.~~
Float16	nominal_voltage	V	Nominal voltage of the battery pack.
Bool	is_powering_off	-	Power off event imminent indication, false if unknown
Uint8	battery_id	-	Identifies the battery within this vehicle, e.g. 0 - primary battery

11.5.8.2 CyphalCAN messages implemented

Table 21. CyphalCAN message subjects and update rates

Subject	Type	Typ. Rate [Hz]
energy_source	reg.udral.physics.electricity.SourceTs	1…100
status	reg.udral.service.battery.Status	~1
parameters	reg.udral.service.battery.Parameters	~0.2

Table 22. Energy source 0.1 CyphalCAN udral

Type	Name	Unit	Description
Uint56	timestamp.microsecond	us	The number of microseconds that have passed since some arbitrary moment in the past. UNKNOWN = 0.
Float32	source.power.current	A	Battery current.
Float32	source.power.voltage	V	Battery voltage.
Float32	source.energy	J	A pessimistic estimate of the amount of energy that can be reclaimed from the source in its current state.
Float32	source.full_energy	J	A pessimistic estimate of the amount of energy that can be reclaimed from a fresh source (a fully charged battery) under the current conditions.

Table 23. Battery status 0.2 CyphalCAN udral

Type	Name	Unit	Description
Uint2	heartbeat.readiness*	-	SLEEP = 0, STANDBY = 2, ENGAGED = 3.
Uint2	heartbeat.health*	-	NOMINAL = 0, ADVISORY = 1, CAUTION = 2, WARNING = 3.
Float32[2]	temperature_min_max	K	The minimum and maximum readings of the pack temperature sensors.
Float32	available_charge	C	The estimated electric charge currently stored in the battery.
Uint8	error*	-	Error status. NONE = 0, BAD_BATTERY = 10, NEEDS_SERVICE = 11, BMS_ERROR = 20, CONFIGURATION = 30, OVERDISCHARGE = 50, OVERLOAD = 51, CELL_OVERVOLTAGE = 60, CELL_UNDERVOLTAGE = 61, CELL_COUNT = 62, TEMPERATURE_HOT = 100, TEMPERATURE_COLD = 101.
Float16[<= 255]	cell_voltages	V	The voltages of individual cells in the battery pack.

** not implemented in the BMS example code*

Table 24. Battery parameter 0.3 CyphalCAN udral

Type	Name	Unit	Description
Uint64	unique_id	-	Unique number.
Float32	mass	Kg	The total mass of the battery.
Float32	design_capacity	C	Design capacity.
Float32[2]	design_cell_voltage_min_max	V	Factory cell voltages.
Float32	discharge_current	A	The discharge current.
Float32	discharge_current_burst	A	The burst discharge current at least for 5 seconds.
Float32	charge_current	A	The charge current.
Float32	charge_current_fast	A	The fast charge current.
Float32	charge_termination_treshold	A	End-of-charging current.
Float32	charge_voltage	V	Total charging voltage.
Uint16	cycle_count	-	The number of cycles.
Uint16	Void16	-	Was cell count.
Uint7	state_of_health_pct	%	The state of health.
Uint8	technology.value	-	Battery chemistry 100 = LiPo, 101 = LiFePO4, 0 = unknown.
Float32	nominal_voltage	V	The nominal battery voltage.

12. Legal information

12.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

12.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the *Terms and conditions of commercial sale* of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Translations — A non-English (translated) version of a document is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Evaluation products — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

12.3 Licenses

Purchase of NXP <xxx> components
<License statement text>

12.4 Patents

Notice is herewith given that the subject device uses one or more of the following patents and that each of these patents may have corresponding patents in other jurisdictions.

<Patent ID> — owned by <Company name>

12.5 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

<Name> — is a trademark of NXP Semiconductors N.V.

13. Contents

Date of release: Jun 7, 2024
Document identifier: Release Notes

Date of release: Jun 7, 2024
Document identifier: Release Notes