CHAPTER 18

Slaves

(Only Section 18.1 is used.)

18.1 Slave 12-Cell `v2.1.6 and above`

18.1.1 Overview

Important: The following description only applies for the 12-cell BMS-Slave Board hardware versions `2.1.6` and above.

The documentation for the 12-cell BMS-Slave Board version `2.1.2` to `2.1.5` can be found *here*.

The documentation for the 12-cell BMS-Slave Board version `2.0.3` to `2.1.1` can be found *here*.

The documentation for the 12-cell BMS-Slave Board version `1.x.x` can be found *here*.

Hint: All connector pinouts described below follow the *Convention for Connector Numbering*.

Block Diagram

A block diagram of the BMS-Slave Board is shown in fig. 18.1

Schematic and Board Layout

More information about the board schematic and layout files can be found in section *Design Resources*.

Mechanical Dimensions

The size of the foxBMS Slave PCB is 160x100mm. A `*.step` file of the PCB can be found in section *Design Resources*.

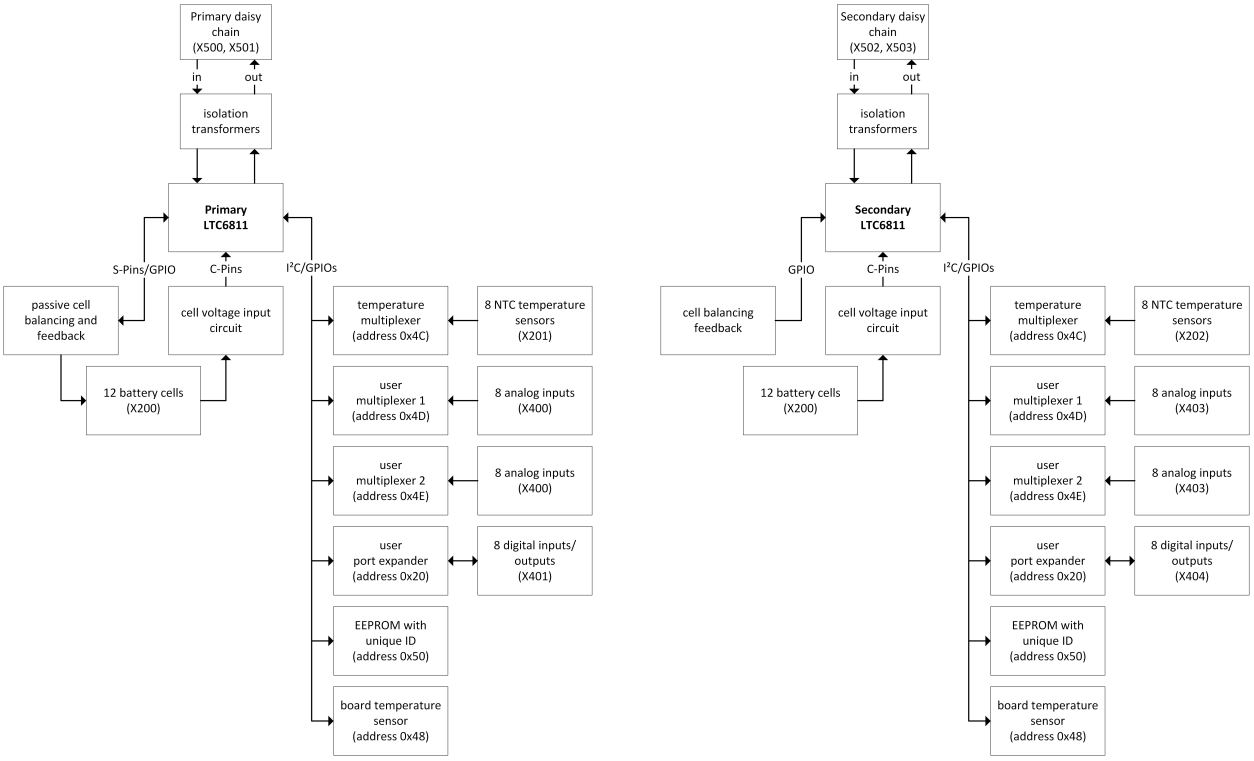


Fig. 18.1: BMS-Slave Board 12-Cell Block Diagram

18.1.2 Functions

The following general description applies to both, the primary and the secondary parts of the BMS-Slave Board. If there are any differences in hardware between the primary and the secondary they will be marked as such.

Cell Voltage Measurement

The cell voltage sense lines are input on the connector X200. The pinout is described in table 18.1.

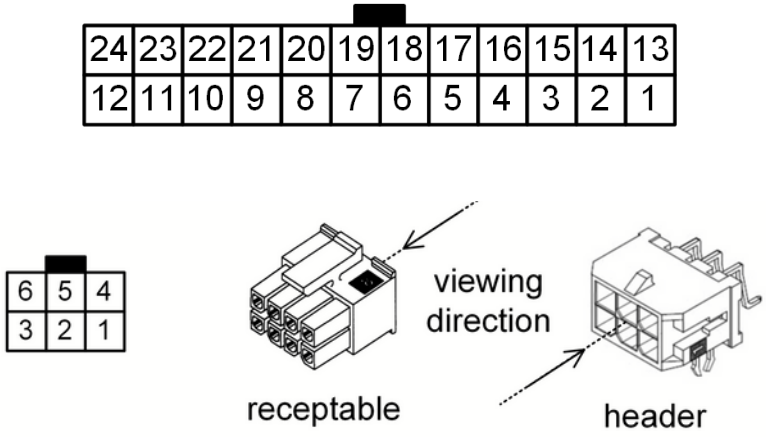


Table 18.1: Cell voltage sense connector

Pin	Signal	Direction	Description
1	VBAT-	Input	Battery module negative terminal
2	CELL_0+	Input	Cell 0 positve terminal
3	CELL_2+	Input	Cell 2 positve terminal
4	CELL_4+	Input	Cell 4 positve terminal
5	CELL_6+	Input	Cell 6 positve terminal
6	CELL_8+	Input	Cell 8 positve terminal
7	CELL_10+	Input	Cell 10 positve terminal
8	VBAT+	Input	Battery module positive terminal
9	NC	–	–
10	NC	–	–
11	NC	–	–
12	NC	–	–
13	CELL_0-	Input	Cell 0 negative terminal
14	CELL_1+	Input	Cell 1 positive terminal
15	CELL_3+	Input	Cell 3 positive terminal
16	CELL_5+	Input	Cell 5 positive terminal
17	CELL_7+	Input	Cell 7 positive terminal
18	CELL_9+	Input	Cell 9 positive terminal
19	CELL_11+	Input	Cell 11 positive terminal
20	NC	–	–
21	NC	–	–
22	NC	–	–
23	NC	–	–
24	NC	–	–

Each of these lines is protected by a 250mA fast fuse surface mount device (F301 - F313) on the board except of the VBAT+ and VBAT- lines which are protected by a value of 500mA (F300 and F314). This is especially important for a test environment. The VBAT+ and VBAT- connections are used for the internal power supply of the BMS-Slave Board board. If the battery module does not contain these separate wires to the positive and negative module terminals, the solder jumpers SJ401 and SJ402 have to be shorted. In this case the power required by the BMS-Slave Board will be supplied through the sense lines CELL_0- and CELL_11+. Running the BMS-Slave Board in this configuration could result in cell measurement errors due to voltage drop over the sense wires.

The cell input lines are filtered by a grounded or differential capacitor filter: both possibilities are provided on the PCB of the BMS-Slave Board. More information on the corner frequency of this filtering can be found in the schematic. The grounded capacitor filter should be used in environments affected with a high noise as it offers a high level of battery voltage ripple rejection. The differential capacitor filter can be used when noise is less occurrent or the design is subjected to cost optimization.

Passive Cell Balancing

The passive balancing circuit is realized by a parallel connection of two 68Ω discharge resistors that can be connected to each single cell in parallel. The MOSFET switches (T1500 - T1511) that control the connection to the cells are controlled by the primary LTC6811-1 monitoring IC. The secondary LTC6811-1 does not support balancing. The resistor value of 2x 68Ω results in a balancing current of about 100mA at a cell voltage of 3.6V. This current results in a power dissipation of about 0.36W per balancing channel (at 3.6V).

Global Cell Balancing Feedback

In order to check the proper function of the balancing process or to detect a malfunction in the balancing circuit, a global balancing feedback signal is connected to the LTC6811-1. This allows the BMS-Master Board to check wheather any balancing action is currently taking place. The feedback signal is connected to the GPIO3 of the LTC6811-1. The signal remains in a logic zero state until any balancing action on at least one cell in the module starts.

Temperature Sensor Measurement

The cell temperature sensors are connected to the connectors X201 (primary) and X202 (secondary). The pinout is identical for the primary and secondary and is described in table 18.2.

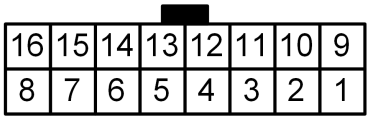


Table 18.2: Temperature sensor connector

Pin	Signal	Direction	Description
1	T-SENSOR_0	Input	NTC Sensor 0 terminal 1
2	T-SENSOR_1	Input	NTC Sensor 1 terminal 1
3	T-SENSOR_2	Input	NTC Sensor 2 terminal 1
4	T-SENSOR_3	Input	NTC Sensor 3 terminal 1
5	T-SENSOR_4	Input	NTC Sensor 4 terminal 1
6	T-SENSOR_5	Input	NTC Sensor 5 terminal 1
7	T-SENSOR_6	Input	NTC Sensor 6 terminal 1
8	T-SENSOR_7	Input	NTC Sensor 7 terminal 1
9	FUSED_VBAT-	Input	NTC Sensor 0 terminal 2
10	FUSED_VBAT-	Input	NTC Sensor 1 terminal 2
11	FUSED_VBAT-	Input	NTC Sensor 2 terminal 2
12	FUSED_VBAT-	Input	NTC Sensor 3 terminal 2
13	FUSED_VBAT-	Input	NTC Sensor 4 terminal 2
14	FUSED_VBAT-	Input	NTC Sensor 5 terminal 2
15	FUSED_VBAT-	Input	NTC Sensor 6 terminal 2
16	FUSED_VBAT-	Input	NTC Sensor 7 terminal 2

Standard 10kΩ NTC resistors (e.g., Farnell-Nr. 1299926) are recommended for use. When using other values than these, the series resistors (R100-1/2 to R107-1/2) on the board may have to be adjusted. Please note that the accuracy of the internal voltage reference VREF2 decreases heavily with a load of over 3mA. Using 8x 10kΩ NTC resistors with the corresponding 10kΩ series resistors results in a current of 1.2mA (at 20°C) which is drawn from VREF2.

Each of the 8 temperture sensors are connected to an analog multiplexer. The analog multiplexer can be controlled via I^2C by the LTC6811-1 (7-bit address: 0x4C). In order to ensure fast settling times after swiching the multiplexer input, the output signal of the multiplexer is buffered by an operational amplifier. Finally the analog voltage of the selected sensor is measured on the GPIO1 pin of the LTC6811-1.

On-board EEPROM

Attention: The BMS-Slave Board hardware versions `2.1.0` and above use a different EEPROM IC (ST M24M02) than all other previous hardware versions.

The primary as well as the secondary unit of the BMS-Slave Board board is equipped with an EEPROM (IC800-1/2). The EEPROM for example can be used for storing data such as calibration values or minimum and maximum temperatures seen by the module during its lifetime. Similar to the analog multiplexers, the EEPROM device is connected to the I^2C bus of the LTC6811-1 (7-bit address: 0x50).

On-board Ambient Temperature Sensor

For an additional monitoring of the ambient temperature an on-board temperature sensor is used. This temperature sensor can be read by the LTC6811-1 via the I^2C bus (7-bit address: 0x48). It is possible to program an alert temperature level. Once the measured temperature reaches this alert temperature level, the alert pin of the IC is set to a logic low level. Currently, this signal is not used on the BMS-Slave Board board, but it is accessible on the connector X402.

Additional Inputs and Outputs

Several additional analog and digital inputs and outputs are provided on the BMS-Slave Board ~~board~~ via pin headers. Each 16 analog inputs are provided on connector X400 (primary) and X403 (secondary). The pinout for the connectors for the primary and secondary unit is identical and is described in table 18.3.

Table 18.3: Connector for analog inputs

Pin	Signal	Direction	Description
1	ANALOG-IN_0	Input	Analog input 0
2	ANALOG-IN_1	Input	Analog input 1
3	ANALOG-IN_2	Input	Analog input 2
4	ANALOG-IN_3	Input	Analog input 3
5	ANALOG-IN_4	Input	Analog input 4
6	ANALOG-IN_5	Input	Analog input 5
7	ANALOG-IN_6	Input	Analog input 6
8	ANALOG-IN_7	Input	Analog input 7
9	ANALOG-IN_8	Input	Analog input 8
10	ANALOG-IN_9	Input	Analog input 9
11	ANALOG-IN_10	Input	Analog input 10
12	ANALOG-IN_11	Input	Analog input 11
13	ANALOG-IN_12	Input	Analog input 12
14	ANALOG-IN_13	Input	Analog input 13
15	ANALOG-IN_14	Input	Analog input 14
16	ANALOG-IN_15	Input	Analog input 15
17	+3.0V_VREF2	Output	LTC6811-1 3.0V voltage reference
18	FUSED_VBAT-	Output	GND

Every group of ~~Each~~ 8 analog inputs are connected to an analog multiplexer. Two The analog multiplexers can be controlled via I^2C by the LTC6811-1 (7-bit addresses: 0x4D and 0x4E). In order to ensure fast settling times after switching the multiplexer input, the output signals of the multiplexers are buffered by operational amplifiers. Finally the analog voltage of the selected sensor can be measured on the GPIO2 pin of the LTC6811-1.

A group of
~~Each~~ 8 digital inputs/outputs are provided on the connectors X401 (primary) and X404 (secondary). The pinout for the connectors for the primary and secondary unit is identical and is described in table 18.4.

Table 18.4: Connector for digital IOs

Pin	Signal	Direction	Description
1	DIGITAL-IO_0	Input/Output	Digital input/output 0
2	DIGITAL-IO_1	Input/Output	Digital input/output 1
3	DIGITAL-IO_2	Input/Output	Digital input/output 2
4	DIGITAL-IO_3	Input/Output	Digital input/output 3
5	DIGITAL-IO_4	Input/Output	Digital input/output 4
6	DIGITAL-IO_5	Input/Output	Digital input/output 5
7	DIGITAL-IO_6	Input/Output	Digital input/output 6
8	+5.0V_VREG	Output	LTC6811-1 5.0V regulated voltage
9	FUSED_VBAT-	Output	GND

Each 8 digital inputs/outputs are connected to an I^2C controlled port expander (7-bit address: 0x20). The direction of the inputs/outputs as well as the logic levels on the pins can be selected by register settings. Each of the 8 digital inputs/outputs has a discrete pull up resistor that for example can be used for directly connecting a tactile switch.

isoSPI Daisy Chain Connection

or
The data transmission between the slaves and between the slaves and the basic board takes place using the isoSPI interface. The isoSPI signals are input on the connectors X500 (primary) and X502 (secondary). The isoSPI signals for daisy-chaining are output on the connectors X501 (primary) and X503 (secondary). The isoSPI connections are isolated galvanically using pulse transformers (TR1400-1/2). The voltage amplitude of the differential signal can be adjusted by setting resistors (see paragraph *Daisy Chain Communication Current*).

The pinout of the isoSPI connectors is described in table 18.5 and table 18.6.

Table 18.5: isoSPI Daisy Chain Input Connectors

Connector Pin	Daisy Chain
1	IN+ (Primary/Secondary LTC6811-1)
2	IN- (Primary/Secondary LTC6811-1)

Table 18.6: isoSPI Daisy Chain Output Connectors

Connector Pin	Daisy Chain
1	OUT+ (Primary/Secondary LTC6811-1)
2	OUT- (Primary/Secondary LTC6811-1)

Hardware Settings / Options

Software Timer

The internal software timer of the LTC6811-1 can be enabled/disabled by a dedicated external pin (SWTEN, pin 36 of the LTC6811-1). In order to support all features, the BMS-Slave Board board offers a possibility to switch the software timer. The software timer is enabled in the standard configuration, which means pin 36 is pulled to VREG via a zero-ohm resistor (R221/R321). The timer can be disabled by removing ~~the resistor R1402-1/2~~ R221/R321 and placing a zero-ohm resistor to R1403-1/2.

Daisy Chain Communication Current

The daisy chain communication current can be set by ~~the~~ resistors R1419-1/2 and R1421-1/2. The default value is 820Ω for R14211/2 and 1.21kΩ for R1419-1/2. This values result in a bias current of approximately 1mA and a differential signal amplitude of 1.18V. Theses values are suitable for high noise environments with cable lengths of over 50m. More information can be found in the LTC6811-1 datasheet.

Status LED

The status LEDs LD1400-1/2 show the current mode of each, the primary and secondary LTC6811-1. The LED is on in STANDBY, REFUP or MEASURE mode, whereas the LED is off in SLEEP mode. The LED can be disabled by removing the resistor R1407-1/2.

GPIO Extension Connector

The internal GPIO lines of the primary or secondary LTC6811-1 can be connected to the GPIO extension pin header X402 via optional zero-ohm resistors. In the standard configuration these resistors are not placed. Of course it is possible to place each both resistors for a parallel connection of the internal signals to the GPIO extension connector. The placement of the resistors and the resulting connection is shown in table 18.7.

Table 18.7: GPIO extension connector

GPIO	connect to pin header	connect to internal function
1	R1405-1/2	R1406-1/2 (default)
2	R1409-1/2	R1410-1/2 (default)
3	R1412-1/2	R1413-1/2 (default)
4	R1414-1/2	R1417-1/2 (default)
5	R1418-1/2	R1420-1/2 (default)

The pinout of the extension connector X402 is described in table 18.8.

Table 18.8: Extension connector

Pin	Signal	Direction	Description
1	+3.0V_VREF2_0	Output	Primary LTC6811-1 3.0V reference voltage 2
2	+3.0V_VREF2_1	Output	Secondary LTC6811-1 3.0V reference voltage 2
3	+5.0V_VREG_0	Output	Primary LTC6811-1 5.0V regulated voltage
4	+5.0V_VREG_1	Output	Secondary LTC6811-1 5.0V regulated voltage
5	PRIMARY-GPIO1-OPT	Input/Output	Primary LTC6811-1 GPIO1
6	SECONDARY-GPIO1-OPT	Input/Output	Secondary LTC6811-1 GPIO1
7	PRIMARY-GPIO2-OPT	Input/Output	Primary LTC6811-1 GPIO2
8	SECONDARY-GPIO2-OPT	Input/Output	Secondary LTC6811-1 GPIO2
9	PRIMARY-GPIO3-OPT	Input/Output	Primary LTC6811-1 GPIO3
10	SECONDARY-GPIO3-OPT	Input/Output	Secondary LTC6811-1 GPIO3
11	PRIMARY-GPIO4-OPT	Input/Output	Primary LTC6811-1 GPIO4
12	SECONDARY-GPIO4-OPT	Input/Output	Secondary LTC6811-1 GPIO4
13	PRIMARY-GPIO5-OPT	Input/Output	Primary LTC6811-1 GPIO5
14	SECONDARY-GPIO5-OPT	Input/Output	Secondary LTC6811-1 GPIO5
15	PRIMARY-WDT	Output	Primary LTC6811-1 watchdog output
16	SECONDARY-WDT	Output	Secondary LTC6811-1 watchdog output
17	PRIMARY-TEMP-ALERT	Output	Primary board T-sensor alarm output
18	SECONDARY-TEMP-ALERT	Output	Secondary board T-sensor alarm output
19	FUSED_VBAT-	Output	GND
20	FUSED_VBAT-	Output	GND

External Isolated DC-Supply

Note: The external isolated DC-supply is only available in the BMS-Slave Board hardware versions `2.1.0` and above.

It is possible to supply the BMS-Slave Board by an external DC power supply with a voltage range of 8V to 24V. The DC input is protected against reverse voltage and over-current (with a 1.25A fuse). The external DC supply has to be connected on connector X1001 or X1002 (both connectors are in parallel for daisy chaining the supply). The pinout of the connectors X1001 and X1002 is shown in table 18.9.

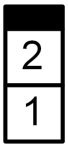


Table 18.9: External DC supply connector

Pin	Signal	Direction	Description
1	DC+	Input	positive supply terminal
2	DC-	Input	negative supply terminal

18.2 Slave 12-Cell `v2.1.2 to v2.1.5`

18.2.1 Overview

Important: The following description only applies for the 12-cell BMS-Slave Board hardware versions `2.1.2` to `2.1.5`.

The documentation for the 12-cell BMS-Slave Board version `2.0.3` to `2.1.1` can be found *here*.

The documentation for the 12-cell BMS-Slave Board version `1.x.x` can be found *here*.

Hint: All connector pinouts described below follow the *Convention for Connector Numbering*.

Block Diagram

A block diagram of the BMS-Slave Board is shown in fig. 18.2

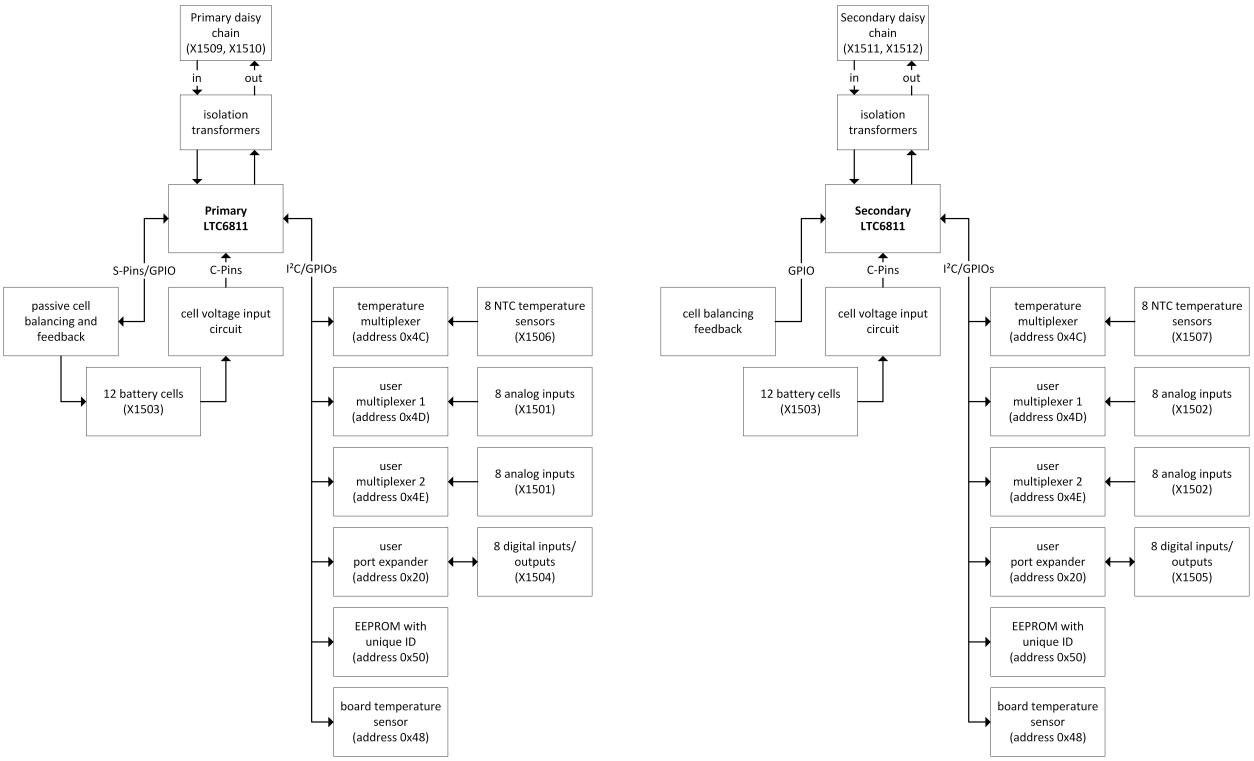


Fig. 18.2: BMS-Slave Board 12-Cell Block Diagram

Schematic and Board Layout

More information about the board schematic and layout files can be found in section *Design Resources*.

Mechanical Dimensions

The size of the foxBMS Slave PCB is 160x100mm. A `*.step` file of the PCB can be found in section *Design Resources*.

18.2.2 Functions

The following general description applies to both, the primary and the secondary of the BMS-Slave Board. If there are any differences in hardware between the primary and the secondary they will be marked as such.

Cell Voltage Measurement

The cell voltage sense lines are input on the connector X1503. The pinout is described in table 18.10.

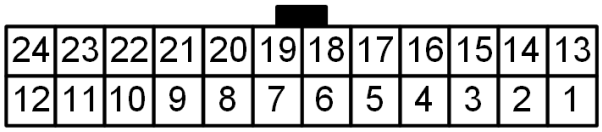


Table 18.10: Cell voltage sense connector

Pin	Signal	Direction	Description
1	VBAT-	Input	Battery module negative terminal
2	CELL_0+	Input	Cell 0 positve terminal
3	CELL_2+	Input	Cell 2 positve terminal
4	CELL_4+	Input	Cell 4 positve terminal
5	CELL_6+	Input	Cell 6 positve terminal
6	CELL_8+	Input	Cell 8 positve terminal
7	CELL_10+	Input	Cell 10 positve terminal
8	VBAT+	Input	Battery module positive terminal
9	NC	–	–
10	NC	–	–
11	NC	–	–
12	NC	–	–
13	CELL_0-	Input	Cell 0 negative terminal
14	CELL_1+	Input	Cell 1 positive terminal
15	CELL_3+	Input	Cell 3 positive terminal
16	CELL_5+	Input	Cell 5 positive terminal
17	CELL_7+	Input	Cell 7 positive terminal
18	CELL_9+	Input	Cell 9 positive terminal
19	CELL_11+	Input	Cell 11 positive terminal
20	NC	–	–
21	NC	–	–
22	NC	–	–
23	NC	–	–
24	NC	–	–

Each of these lines is protected by a 250mA fast fuse surface mount device (F402 - F414) on the board except of the VBAT+ and VBAT- lines which are protected by a value of 500mA (F401 and F415). This is especially important for a test environment. The VBAT+ and VBAT- connections are used for the internal power supply of the BMS-Slave Board board. If the battery module does not contain these separate wires to the positive and negative module terminals, the

solder jumpers SJ401 and SJ402 have to be shorted. In this case the power required by the BMS-Slave Board will be supplied through the sense lines CELL_0- and CELL_11+. Running the BMS-Slave Board in this configuration could result in cell measurement errors due to voltage drop over the sense wires.

The cell input lines are filtered by a grounded or differential capacitor filter: both possibilities are provided on the PCB of the BMS-Slave Board. More information on the corner frequency of this filtering can be found in the schematic. The grounded capacitor filter should be used in environments affected with a high noise as it offers a high level of battery voltage ripple rejection. The differential capacitor filter can be used when noise is less occurrent or the design is subjected to cost optimization.

Passive Cell Balancing

The passive balancing circuit is realized by a parallel connection of two 68Ω discharge resistors that can be connected to each single cell in parallel. The MOSFET switches (T701 - T712) that control the connection to the cells are controlled by the primary LTC6811-1 monitoring IC. The secondary LTC6811-1 does not support balancing. The resistor value of 2x 68Ω results in a balancing current of about 100mA at a cell voltage of 3.6V. This current results in a power dissipation of about 0.36W per balancing channel (at 3.6V).

Global Cell Balancing Feedback

In order to check the proper function of the balancing process or to detect a malfunction in the balancing circuit, a global balancing feedback signal is connected to the LTC6811-1. This allows the BMS-Master Board to check wheather any balancing action is currently taking place. The feedback signal is connected to the GPIO3 of the LTC6811-1. The signal remains in a logic zero state until any balancing action on at least one cell in the module starts.

Temperature Sensor Measurement

The cell temperature sensors are connected to the connectors X1506 (primary) and X1507 (secondary). The pinout is identical for the primary and secondary and is described in table 18.11.

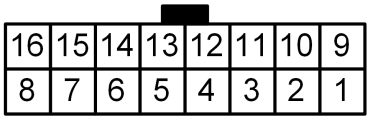


Table 18.11: Temperature sensor connector

Pin	Signal	Direction	Description
1	T-SENSOR_0	Input	NTC Sensor 0 terminal 1
2	T-SENSOR_1	Input	NTC Sensor 1 terminal 1
3	T-SENSOR_2	Input	NTC Sensor 2 terminal 1
4	T-SENSOR_3	Input	NTC Sensor 3 terminal 1
5	T-SENSOR_4	Input	NTC Sensor 4 terminal 1
6	T-SENSOR_5	Input	NTC Sensor 5 terminal 1
7	T-SENSOR_6	Input	NTC Sensor 6 terminal 1
8	T-SENSOR_7	Input	NTC Sensor 7 terminal 1
9	FUSED_VBAT-	Input	NTC Sensor 0 terminal 2
10	FUSED_VBAT-	Input	NTC Sensor 1 terminal 2
11	FUSED_VBAT-	Input	NTC Sensor 2 terminal 2
12	FUSED_VBAT-	Input	NTC Sensor 3 terminal 2
13	FUSED_VBAT-	Input	NTC Sensor 4 terminal 2
14	FUSED_VBAT-	Input	NTC Sensor 5 terminal 2
15	FUSED_VBAT-	Input	NTC Sensor 6 terminal 2
16	FUSED_VBAT-	Input	NTC Sensor 7 terminal 2

Standard 10kΩ NTC resistors (e.g., Farnell-Nr. 1299926) are recommended for use. When using other values than these, the series resistors (R901-R908 / R1001-R1008) on the board may have to be adjusted. Please note that the accuracy of the internal voltage reference VREF2 decreases heavily with a load of over 3mA. Using 8x 10kΩ NTC resistors with the corresponding 10kΩ series resistors results in a current of 1.2mA (at 20°C) which is drawn from VREF2.

Each of the 8 temperture sensors are connected to an analog multiplexer. The analog multiplexer can be controlled via I^2C by the LTC6811-1 (7-bit address: 0x4C). In order to ensure fast settling times after swiching the multiplexer input, the output signal of the multiplexer is buffered by an operational amplifier. Finally the analog voltage of the selected sensor is measured on the GPIO1 pin of the LTC6811-1.

On-board EEPROM

Attention: The BMS-Slave Board hardware versions `2.1.0` and above use a different EEPROM IC (ST M24M02) than all other previous hardware versions.

The primary as well as the secondary unit of the BMS-Slave Board board is equipped with an EEPROM (IC1301 / IC1401). The EEPROM for example can be used for storing data such as calibration values or minimum and maximum temperatures seen by the module during its lifetime. Similar to the analog multiplexers, the EEPROM device is connected to the I^2C bus of the LTC6811-1 (7-bit address: 0x50).

On-board Ambient Temperature Sensor

For an additional monitoring of the ambient temperature an on-board temperature sensor is used. This temperature sensor can be read by the LTC6811-1 via the I^2C bus (7-bit address: 0x48). It is possible to program an alert temperature level. Once the measured temperature reaches this alert temperature level, the alert pin of the IC is set to a logic low level. Currently, this signal is not used on the BMS-Slave Board board, but it is accessible on the connector X1508.

Additional Inputs and Outputs

Several additional analog and digital inputs and outputs are provided on the BMS-Slave Board board via pin headers. Each 16 analog inputs are provided on connector X1501 (primary) and X1502 (secondary). The pinout for the connectors for the primary and secondary unit is identical and is described in table 18.12.

Table 18.12: Connector for analog inputs

Pin	Signal	Direction	Description
1	ANALOG-IN_0	Input	Analog input 0
2	ANALOG-IN_1	Input	Analog input 1
3	ANALOG-IN_2	Input	Analog input 2
4	ANALOG-IN_3	Input	Analog input 3
5	ANALOG-IN_4	Input	Analog input 4
6	ANALOG-IN_5	Input	Analog input 5
7	ANALOG-IN_6	Input	Analog input 6
8	ANALOG-IN_7	Input	Analog input 7
9	ANALOG-IN_8	Input	Analog input 8
10	ANALOG-IN_9	Input	Analog input 9
11	ANALOG-IN_10	Input	Analog input 10
12	ANALOG-IN_11	Input	Analog input 11
13	ANALOG-IN_12	Input	Analog input 12
14	ANALOG-IN_13	Input	Analog input 13
15	ANALOG-IN_14	Input	Analog input 14
16	ANALOG-IN_15	Input	Analog input 15
17	+3.0V_VREF2	Output	LTC6811-1 3.0V voltage reference
18	FUSED_VBAT-	Output	GND

Each 8 analog inputs are connected to an analog multiplexer. The analog multiplexers can be controlled via I^2C by the LTC6811-1 (7-bit addresses: 0x4D and 0x4E). In order to ensure fast settling times after switching the multiplexer input, the output signals of the multiplexers are buffered by operational amplifiers. Finally the analog voltage of the selected sensor can be measured on the GPIO2 pin of the LTC6811-1.

Each 8 digital inputs/outputs are provided on the connectors X1504 (primary) and X1505 (secondary). The pinout for the connectors for the primary and secondary unit is identical and is described in table 18.13.

Table 18.13: Connector for digital IOs

Pin	Signal	Direction	Description
1	DIGITAL-IO_0	Input/Output	Digital input/output 0
2	DIGITAL-IO_1	Input/Output	Digital input/output 1
3	DIGITAL-IO_2	Input/Output	Digital input/output 2
4	DIGITAL-IO_3	Input/Output	Digital input/output 3
5	DIGITAL-IO_4	Input/Output	Digital input/output 4
6	DIGITAL-IO_5	Input/Output	Digital input/output 5
7	DIGITAL-IO_6	Input/Output	Digital input/output 6
8	+5.0V_VREG	Output	LTC6811-1 5.0V regulated voltage
9	FUSED_VBAT-	Output	GND

Each 8 digital inputs/outputs are connected to an I^2C controlled port expander (7-bit address: 0x20). The direction of the inputs/outputs as well as the logic levels on the pins can be selected by register settings. Each of the 8 digital inputs/outputs has a discrete pull up resistor that for example can be used for directly connecting a tactile switch.

isoSPI Daisy Chain Connection

The data transmission between the slaves and between the slaves and the basic board takes place using the isoSPI interface. The isoSPI signals are input on the connectors X1509 (primary) and X1511 (secondary). The isoSPI signals for daisy-chaining are output on the connectors X1510 (primary) and X1512 (secondary). The isoSPI connections are isolated galvanically using pulse transformers (TR201 / TR301). The voltage amplitude of the differential signal can be adjusted by setting resistors (see paragraph *Daisy Chain Communication Current*).

The pinout of the isoSPI connectors is described in table 18.14 and table 18.15.

Table 18.14: isoSPI Daisy Chain Input Connectors

Connector Pin	Daisy Chain
1	IN+ (Primary/Secondary LTC6811-1)
2	IN- (Primary/Secondary LTC6811-1)

Table 18.15: isoSPI Daisy Chain Output Connectors

Connector Pin	Daisy Chain
1	OUT+ (Primary/Secondary LTC6811-1)
2	OUT- (Primary/Secondary LTC6811-1)

Hardware Settings / Options

Software Timer

The internal software timer of the LTC6811-1 can be enabled/disabled by a dedicated external pin (SWTEN, pin 36 of the LTC6811-1). In order to support all features, the BMS-Slave Board board offers a possibility to switch the software timer. The software timer is enabled in the standard configuration, which means pin 36 is pulled to VREG via a zero-ohm resistor (R221/R321). The timer can be disabled by removing the resistor R221/R321 and placing a zero-ohm resistor to R220/R320.

Daisy Chain Communication Current

The daisy chain communication current can be set by the resistors R206/R306 and R208/R308. The default value is 820Ω for R206/R306 and 1.21kΩ for R208/R308. This values result in a bias current of approximately 1mA and a differential signal amplitude of 1.18V. Theses values are suitable for high noise environments with cable lengths of over 50m. More information can be found in the LTC6811-1 datasheet.

Status LED

The status LEDs LD201 and LD301 show the current mode of each, the primary and secondary LTC6811-1. The LED is on in STANDBY, REFUP or MEASURE mode, whereas the LED is off in SLEEP mode. The LED can be disabled by removing the resistor R205 (primary) or R305 (secondary).

GPIO Extension Connector

The internal GPIO lines of the primary or secondary LTC6811-1 can be connected to the GPIO extension pin header X1508 via optional zero-ohm resistors. In the standard configuration these resistors are not placed. Of course it is possible to place each both resistors for a parallel connection of the internal signals to the GPIO extension connector. For more information see page 2/3 of the schematic file. The placement of the resistors and the resulting connection is shown in table 18.16.

Table 18.16: GPIO extension connector

GPIO	connect to pin header	connect to internal function
1	R209/R309	R210/R310 (default)
2	R211/R311	R212/R312 (default)
3	R213/R313	R214/R314 (default)
4	R215/R315	R216/R316 (default)
5	R217/R317	R218/R318 (default)

The pinout of the extension connector X1508 is described in table 18.17.

Table 18.17: Extension connector

Pin	Signal	Direction	Description
1	+3.0V_VREF2_0	Output	Primary LTC6811-1 3.0V reference voltage 2
2	+3.0V_VREF2_1	Output	Secondary LTC6811-1 3.0V reference voltage 2
3	+5.0V_VREG_0	Output	Primary LTC6811-1 5.0V regulated voltage
4	+5.0V_VREG_1	Output	Secondary LTC6811-1 5.0V regulated voltage
5	PRIMARY-GPIO1-OPT	Input/Output	Primary LTC6811-1 GPIO1
6	SECONDARY-GPIO1-OPT	Input/Output	Secondary LTC6811-1 GPIO1
7	PRIMARY-GPIO2-OPT	Input/Output	Primary LTC6811-1 GPIO2
8	SECONDARY-GPIO2-OPT	Input/Output	Secondary LTC6811-1 GPIO2
9	PRIMARY-GPIO3-OPT	Input/Output	Primary LTC6811-1 GPIO3
10	SECONDARY-GPIO3-OPT	Input/Output	Secondary LTC6811-1 GPIO3
11	PRIMARY-GPIO4-OPT	Input/Output	Primary LTC6811-1 GPIO4
12	SECONDARY-GPIO4-OPT	Input/Output	Secondary LTC6811-1 GPIO4
13	PRIMARY-GPIO5-OPT	Input/Output	Primary LTC6811-1 GPIO5
14	SECONDARY-GPIO5-OPT	Input/Output	Secondary LTC6811-1 GPIO5
15	PRIMARY-WDT	Output	Primary LTC6811-1 watchdog output
16	SECONDARY-WDT	Output	Secondary LTC6811-1 watchdog output
17	PRIMARY-TEMP-ALERT	Output	Primary board T-sensor alarm output
18	SECONDARY-TEMP-ALERT	Output	Secondary board T-sensor alarm output
19	FUSED_VBAT-	Output	GND
20	FUSED_VBAT-	Output	GND

External Isolated DC-Supply

Note: The external isolated DC-supply is only available in the BMS-Slave Board hardware versions `2.1.0` and above.

It is possible to supply the BMS-Slave Board by an external DC power supply with a voltage range of 8V to 24V. The DC input is protected against reverse voltage and over-current (with a 1.25A fuse). The external DC supply has to be

connected on connector X1513 or X1514 (both connectors are in parallel for daisy chaining the supply). The pinout of the connectors X1513 and X1514 is shown in table 18.18.

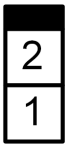


Table 18.18: External DC supply connector

Pin	Signal	Direction	Description
1	DC+	Input	positive supply terminal
2	DC-	Input	negative supply terminal

18.3 Slave 12-Cell `v2.0.3 to v2.1.1`

18.3.1 Overview

Note: The following description only applies for the BMS-Slave Board hardware versions `2.0.3` to `2.1.1`.

Hint: All connector pinouts described below follow the *Convention for Connector Numbering*.

Block Diagram

Schematic and Board Layout

More information about the board schematic and layout files can be found in section *Design Resources*.

Mechanical Dimensions

The size of the foxBMS Slave PCB is 160x100mm. A mechanical drawing of the PCB can be found in section *Design Resources*.

18.3.2 Functions

The following general description applies to both, the primary and the secondary of the BMS-Slave Board. If there are any differences in hardware between the primary and the secondary they will be marked as such.

Cell Voltage Measurement

The cell voltage sense lines are input on the connector X1503. The pinout is described in the table below.

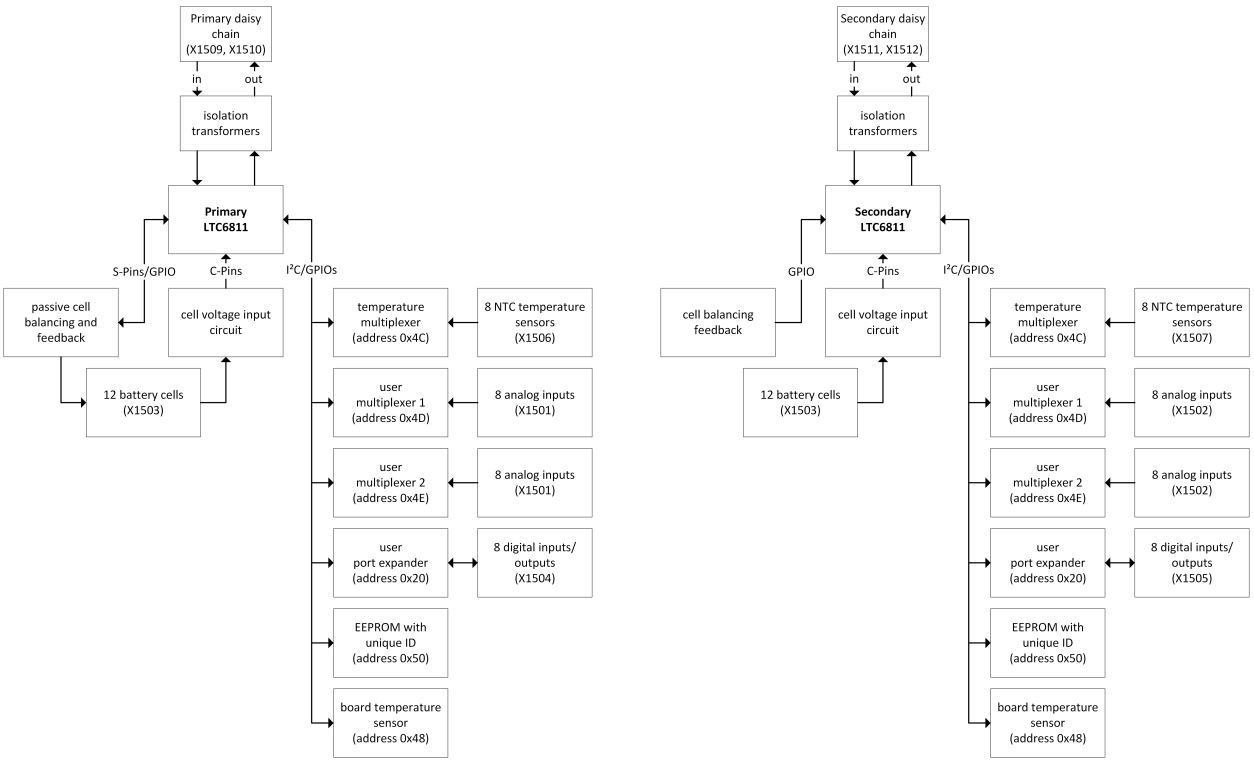
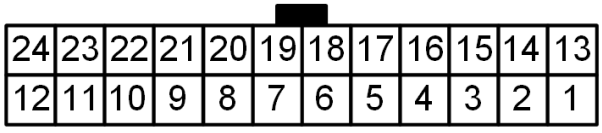


Fig. 18.3: BMS-Slave Board 12-Cell Block Diagram



Pin	Signal	Direction	Description
1	VBAT-	Input	Battery module negative terminal
2	CELL_0+	Input	Cell 0 positve terminal
3	CELL_2+	Input	Cell 2 positve terminal
4	CELL_4+	Input	Cell 4 positve terminal
5	CELL_6+	Input	Cell 6 positve terminal
6	CELL_8+	Input	Cell 8 positve terminal
7	CELL_10+	Input	Cell 10 positve terminal
8	VBAT+	Input	Battery module positive terminal
9	NC	–	–
10	NC	–	–
11	NC	–	–
12	NC	–	–
13	CELL_0-	Input	Cell 0 negative terminal
14	CELL_1+	Input	Cell 1 positive terminal
15	CELL_3+	Input	Cell 3 positive terminal
16	CELL_5+	Input	Cell 5 positive terminal
17	CELL_7+	Input	Cell 7 positive terminal
18	CELL_9+	Input	Cell 9 positive terminal
19	CELL_11+	Input	Cell 11 positive terminal
20	NC	–	–
21	NC	–	–
22	NC	–	–
23	NC	–	–
24	NC	–	–

Each of these lines is protected by a 250mA fast fuse surface mount device (F402 - F414) on the board except of the VBAT+ and VBAT- lines which are protected by a value of 500mA (F401 and F415). This is especially important for a test environment. The VBAT+ and VBAT- connections are used for the internal power supply of the BMS-Slave Board board. If the battery module does not contain these separate wires to the positive and negative module terminals, the solder jumpers SJ401 and SJ402 have to be shorted. In this case the power required by the BMS-Slave Board will be supplied through the sense lines CELL_0- and CELL_11+. Running the BMS-Slave Board in this configuration could result in cell measurement errors due to voltage drop over the sense wires.

The cell input lines are filtered by a grounded or differential capacitor filter: both possibilities are provided on the PCB of the BMS-Slave Board. More information on the corner frequency of this filtering can be found in the schematic. The grounded capacitor filter should be used in environments affected with a high noise as it offers a high level of battery voltage ripple rejection. The differential capacitor filter can be used when noise is less occurrent or the design is subjected to cost optimization.

Passive Cell Balancing

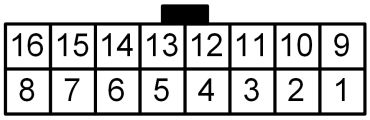
The passive balancing circuit is realized by a parallel connection of two 68Ω discharge resistors that can be connected to each single cell in parallel. The MOSFET switches (T701 - T712) that control the connection to the cells are controlled by the primary LTC6811-1 monitoring IC. The secondary LTC6811-1 does not support balancing. The resistor value of 2x 68Ω results in a balancing current of about 100mA at a cell voltage of 3.6V. This current results in a power dissipation of about 0.36W per balancing channel (at 3.6V).

Global Cell Balancing Feedback

In order to check the proper function of the balancing process or to detect a malfunction in the balancing circuit, a global balancing feedback signal is connected to the LTC6811-1. This allows the BMS-Master Board to check wheather any balancing action is currently taking place. The feedback signal is connected to the GPIO3 of the LTC6811-1. The signal remains in a logic zero state until any balancing action on at least one cell in the module starts.

Temperature Sensor Measurement

The cell temperature sensors are connected to the connectors X1506 (primary) and X1507 (secondary). The pinout is identical for the primary and secondary and is described in the table below.



Pin	Signal	Direction	Description
1	T-SENSOR_0	Input	NTC Sensor 0 terminal 1
2	T-SENSOR_1	Input	NTC Sensor 1 terminal 1
3	T-SENSOR_2	Input	NTC Sensor 2 terminal 1
4	T-SENSOR_3	Input	NTC Sensor 3 terminal 1
5	T-SENSOR_4	Input	NTC Sensor 4 terminal 1
6	T-SENSOR_5	Input	NTC Sensor 5 terminal 1
7	T-SENSOR_6	Input	NTC Sensor 6 terminal 1
8	T-SENSOR_7	Input	NTC Sensor 7 terminal 1
9	FUSED_VBAT-	Input	NTC Sensor 0 terminal 2
10	FUSED_VBAT-	Input	NTC Sensor 1 terminal 2
11	FUSED_VBAT-	Input	NTC Sensor 2 terminal 2
12	FUSED_VBAT-	Input	NTC Sensor 3 terminal 2
13	FUSED_VBAT-	Input	NTC Sensor 4 terminal 2
14	FUSED_VBAT-	Input	NTC Sensor 5 terminal 2
15	FUSED_VBAT-	Input	NTC Sensor 6 terminal 2
16	FUSED_VBAT-	Input	NTC Sensor 7 terminal 2

Standard 10kΩ NTC resistors (e.g., Farnell-Nr. 1299926) are recommended for use. When using other values than these, the series resistors (R901-R908 / R1001-R1008) on the board may have to be adjusted. Please note that the accuracy of the internal voltage reference VREF2 decreases heavily with a load of over 3mA. Using 8x 10kΩ NTC resistors with the corresponding 10kΩ series resistors results in a current of 1.2mA (at 20°C) which is drawn from VREF2.

Each of the 8 temperture sensors are connected to an analog multiplexer. The analog multiplexer can be controlled via I^2C by the LTC6811-1 (7-bit address: 0x4C). In order to ensure fast settling times after swiching the multiplexer input, the output signal of the multiplexer is buffered by an operational amplifier. Finally the analog voltage of the selected sensor is measured on the GPIO1 pin of the LTC6811-1.

On-board EEPROM

Note: The BMS-Slave Board hardware versions `2.1.0` and above use a different EEPROM IC (ST M24M02) than all other previous hardware versions.

The primary as well as the secondary unit of the BMS-Slave Board board is equipped with an EEPROM (IC1301 / IC1401). The EEPROM for example can be used for storing data such as calibration values or minimum and maximum temperatures seens by the module during its lifetime. Similar to the analog multiplexers, the EEPROM device is connected to the I^2C bus of the LTC6811-1 (7-bit address: 0x50).

On-board Ambient Temperature Sensor

For an additional monitoring of the ambient temperature an on-board temperature sensor is used. This temperature sensor can be read by the LTC6811-1 viathe I^2C bus (7-bit address: 0x48). It is possible to program an alert temperature level. Once the measured temperature reaches this alert temperature level, the alert pin of the IC is set to a logic low level. Currently, this signal is not used on the BMS-Slave Board board, but it is accessible on the connector X1508.

Additional Inputs and Outputs

Several additional analog and digital inputs and outputs are provided on the BMS-Slave Board board via pin headers. Each 16 analog inputs are provided on connector X1501 (primary) and X1502 (secondary). The pinout for the connectors for the primary and secondary unit is identical and is described in the table below.

Pin	Signal	Direction	Description
1	ANALOG-IN_0	Input	Analog input 0
2	ANALOG-IN_1	Input	Analog input 1
3	ANALOG-IN_2	Input	Analog input 2
4	ANALOG-IN_3	Input	Analog input 3
5	ANALOG-IN_4	Input	Analog input 4
6	ANALOG-IN_5	Input	Analog input 5
7	ANALOG-IN_6	Input	Analog input 6
8	ANALOG-IN_7	Input	Analog input 7
9	ANALOG-IN_8	Input	Analog input 8
10	ANALOG-IN_9	Input	Analog input 9
11	ANALOG-IN_10	Input	Analog input 10
12	ANALOG-IN_11	Input	Analog input 11
13	ANALOG-IN_12	Input	Analog input 12
14	ANALOG-IN_13	Input	Analog input 13
15	ANALOG-IN_14	Input	Analog input 14
16	ANALOG-IN_15	Input	Analog input 15
17	+3.0V_VREF2	Output	LTC6811-1 3.0V voltage reference
18	FUSED_VBAT-	Output	GND

Each 8 analog inputs are connected to an analog multiplexer. The analog multiplexers can be controlled via I^2C by the LTC6811-1 (7-bit addresses: 0x4D and 0x4E). In order to ensure fast settling times after swiching the multiplexer input, the output signals of the multiplexers are buffered by operational amplifiers. Finally the analog voltage of the selected sensor can be measured on the GPIO2 pin of the LTC6811-1.

Each 8 digital inputs/outputs are provided on the connectors X1504 (primary) and X1505 (secondary). The pinout for the connectors for the primary and secondary unit is identical and is described in the table below.

Pin	Signal	Direction	Description
1	DIGITAL-IO_0	Input/Output	Digital input/output 0
2	DIGITAL-IO_1	Input/Output	Digital input/output 1
3	DIGITAL-IO_2	Input/Output	Digital input/output 2
4	DIGITAL-IO_3	Input/Output	Digital input/output 3
5	DIGITAL-IO_4	Input/Output	Digital input/output 4
6	DIGITAL-IO_5	Input/Output	Digital input/output 5
7	DIGITAL-IO_6	Input/Output	Digital input/output 6
8	+5.0V_VREG	Output	LTC6811-1 5.0V regulated voltage
9	FUSED_VBAT-	Output	GND

Each 8 digital inputs/outputs are connected to an I^2C controlled port expander (7-bit address: 0x20). The direction of the inputs/outputs as well as the logiclevels on the pins can be selected by register settings. Each of the 8 digital inputs/outputs has a discrete pull up resistor that for example can be used for directly connecting a tactile switch.

isoSPI Daisy Chain Connection

The data transmission between the slaves and between the slaves and the basic board takes place using the isoSPI interface. The isoSPI signals are input onthe connectors X1509 (primary) and X1511 (secondary). The isoSPI signals for daisy-chaining are output on the connectors X1510 (primary) and X1512 (secondary). The isoSPI connections are isolated galvanically using pulse transformers (TR201 / TR301). The voltage amplitude of the differential signal can be adjusted by setting resistors (see section *Daisy Chain Communication Current*).

The pinout of the isoSPI connectors is described in table 18.19 and table 18.20.

Table 18.19: isoSPI Daisy Chain Input Connectors

Connector Pin	Daisy Chain
1	IN+ (Primary/Secondary LTC6811-1)
2	IN- (Primary/Secondary LTC6811-1)

Table 18.20: isoSPI Daisy Chain Output Connectors

Connector Pin	Daisy Chain
1	OUT+ (Primary/Secondary LTC6811-1)
2	OUT- (Primary/Secondary LTC6811-1)

Hardware Settings / Options

Software Timer

The internal software timer of the LTC6811-1 can be enabled/disabled by a dedicated external pin (SWTEN, pin 36). In order to support all features, the BMS-Slave Board board offers a possibility to switch the software timer. The software timer is disabled in the standard configuration, which means pin 36 is pulled to GND via a zero-ohm resistor (R219/R319). The timer can be enabled by removing the resistor R219/R319 and placing a zero-ohm resistor to R220/R320.

Daisy Chain Communication Current

The daisy chain communication current can be set by the resisitors R205/R305 and R207/R307. The default value is 820Ω for R205/R305 and 1.21kΩ for R207/R307. This values result in a bias current of approximately 1mA and a differential signal amplitude of 1.18V. Theses values are suitable for high noise environments with cable lengths of over 50m. More information can be found in the LTC6811-1 datasheet.

Status LED

The status LEDs LD201 and LD301 show the current mode of each, the primary and secondary LTC6811-1. The LED is on in STANDBY, REFUP or MEASURE mode, whereas the LED is off in SLEEP mode. The LED can be disabled by removing the resistor R205 (primary) or R305 (secondary).

GPIO Extension Connector

The internal GPIO lines of the primary or secondary LTC6811-1 can be connected to the GPIO extension pin header X1508 via optional zero-ohm resistors. In the standard configuration these resistors are not placed. Of course it is possible to place each both resistors for a parallel connection of the internal signals to the GPIO extension connector. For more information see page 2/3 of the schematic file. The placement of the resistors and the resulting connection is shown in the table below.

GPIO	connect to pin header	connect to internal function
1	R209/R309	R208/R308 (default)
2	R211/R311	R210/R310 (default)
3	R213/R313	R212/R312 (default)
4	R215/R315	R214/R314 (default)
5	R217/R317	R216/R316 (default)

The pinout of the extension connector X1508 is described below:

Pin	Signal	Direction	Description
1	+3.0V_VREF2_0	Output	Primary LTC6811-1 3.0V reference voltage 2
2	+3.0V_VREF2_1	Output	Secondary LTC6811-1 3.0V reference voltage 2
3	+5.0V_VREG_0	Output	Primary LTC6811-1 5.0V regulated voltage
4	+5.0V_VREG_1	Output	Secondary LTC6811-1 5.0V regulated voltage
5	PRIMARY-GPIO1-OPT	Input/Output	Primary LTC6811-1 GPIO1
6	SECONDARY-GPIO1-OPT	Input/Output	Secondary LTC6811-1 GPIO1
7	PRIMARY-GPIO2-OPT	Input/Output	Primary LTC6811-1 GPIO2
8	SECONDARY-GPIO2-OPT	Input/Output	Secondary LTC6811-1 GPIO2
9	PRIMARY-GPIO3-OPT	Input/Output	Primary LTC6811-1 GPIO3
10	SECONDARY-GPIO3-OPT	Input/Output	Secondary LTC6811-1 GPIO3
11	PRIMARY-GPIO4-OPT	Input/Output	Primary LTC6811-1 GPIO4
12	SECONDARY-GPIO4-OPT	Input/Output	Secondary LTC6811-1 GPIO4
13	PRIMARY-GPIO5-OPT	Input/Output	Primary LTC6811-1 GPIO5
14	SECONDARY-GPIO5-OPT	Input/Output	Secondary LTC6811-1 GPIO5
15	PRIMARY-WDT	Output	Primary LTC6811-1 watchdog output
16	SECONDARY-WDT	Output	Secondary LTC6811-1 watchdog output
17	PRIMARY-TEMP-ALERT	Output	Primary board T-sensor alarm output
18	SECONDARY-TEMP-ALERT	Output	Secondary board T-sensor alarm output
19	FUSED_VBAT-	Output	GND
20	FUSED_VBAT-	Output	GND

External Isolated DC-Supply

Note: The external isolated DC-supply is only available in the BMS-Slave Board hardware versions `2.1.0` and above.

It is possible to supply the BMS-Slave Board by an external DC power supply with a voltage range of 8V to 24V. The DC input is protected against reverse voltage and overcurrent (with a 1.25A fuse). The external DC supply has to be connected on connector X1513 or X1514 (both connectors are in parallel for daisy chaining the supply). The pinout of the connectors X1513 and X1514 is shown below:

Pin	Signal	Direction	Description
1	DC+	Input	positive supply terminal
2	DC-	Input	negative supply terminal

18.4 Slave 12-Cell `v1.x.x`

18.4.1 Overview

Note: The following description only applies for the BMS-Slave Board hardware versions `1.x.x`.

Hint: All connector pinouts described below follow the *Convention for Connector Numbering*.

Block Diagram

A block diagram of a BMS-Slave Board is shown below:

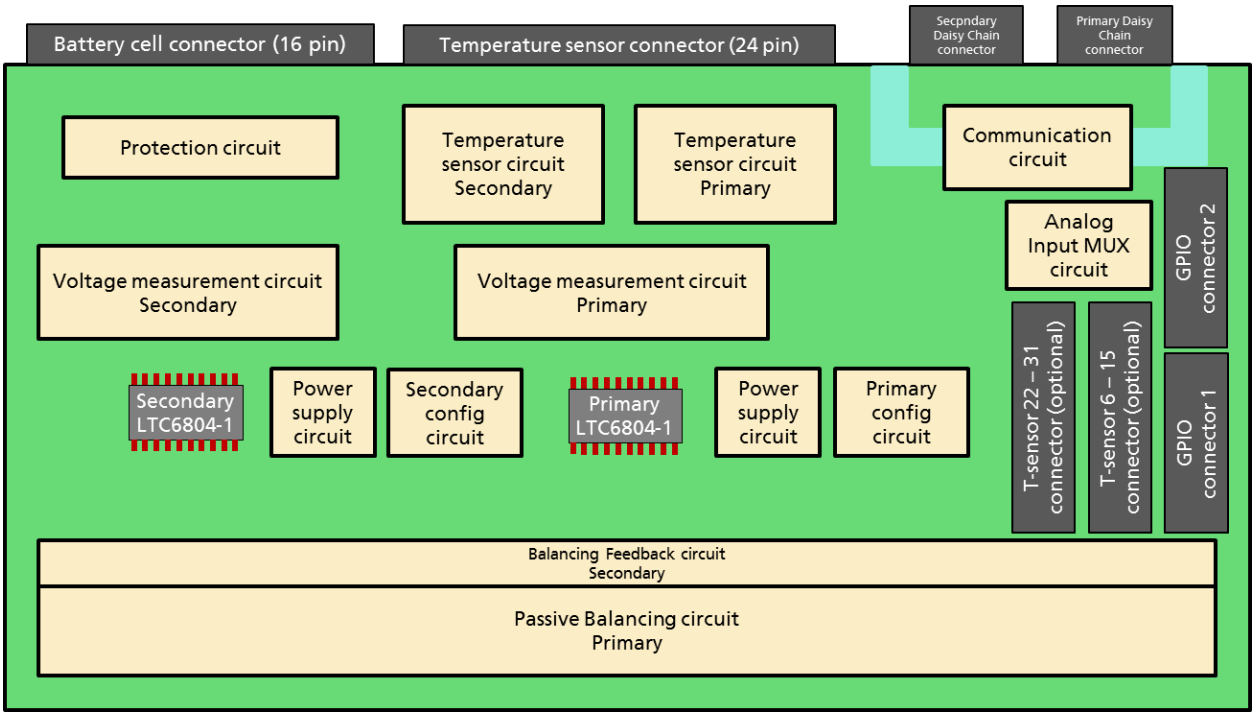


Fig. 18.4: Block diagram of a BMS-Slave Board

The connector indicated as `Daisy Chain` in fig. 12.11 must be used on the BMS-Master Board. Its layout is described in the table below:

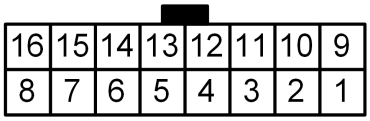


Table 18.21: BMS-Master Board Daisy Chain Connector

Pin	Signal
1	NC
2	OUT+ (Secondary LTC6804-1)
3	OUT- (Secondary LTC6804-1)
4	NC
5	NC
6	OUT+ (Primary LTC6804-1)
7	OUT- (Primary LTC6804-1)
8	NC
9	NC
10	NC
11	NC
12	NC
13	NC
14	NC
15	NC
16	NC

Note: This connector pin out is only valid for use of a foxBMS Master Interface board for the LTC6804-1 monitoring IC.

On the BMS-Slave Board, the connectors indicated as `Primary Daisy Chain connector` and `Secondary Daisy Chain connector` in fig. 18.4 must be used. Their layout is described in table 18.22.

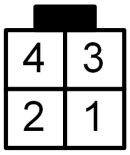


Table 18.22: Primary Daisy Chain Connector

Connector Pin	Daisy Chain
1	IN+ (Primary LTC6804-1)
2	OUT- (Primary LTC6804-1)
3	IN- (Primary LTC6804-1)
4	OUT+ (Primary LTC6804-1)

The OUT+ and OUT- pins of the BMS-Master Board go to the IN+ and IN- pins of the BMS-Slave Board. A cable with a receptable on both ends must be crimped correctly to make the connection.

In case a second BMS-Slave Board must be connected to the daisy chain, the OUT+ and OUT- pins of the first BMS-Slave Board must be connected to the IN+ and IN- pins of the second BMS-Slave Board.

18.4.2 Cell Voltage Connector on the foxBMS Slave Units

The connector indicated as `Battery cell connector (16 pin)` in fig. 18.4 has two purposes:

- Supply of the BMS-Slave Board

- Input of the cell voltages to the LTC6804-1 monitoring chip

The layout of the connector is described in table 18.23. Up to 12 battery cells can be connected in series, between VBAT+ and VBAT-. The BMS-Slave Board is supplied by VBAT+ and VBAT-. The total voltage of all cells in series must be between 11V and 55V (see [ltc_datasheet6804] and [ltc_datasheet6811]). 0- correspond to the negative pole of cell 0, 0+ to the positive pole of cell 0, 1- correspond to the negative pole of cell 1, 1+ to the positive pole of cell 1 and so one till 11+, the positive pole of cell 11. As the cells are connected in series, the positive pole of one cell is connected to the negative pole of the next cell: 0+ to 1-, 1+ to 2+ and so on. The poles should be connected to the cell voltage connector as shown in table 18.23.

If less than 12 battery cells are used, information on how to connect them can be found LTC6804-1 datasheets ([ltc_datasheet6804] and [ltc_datasheet6811]).

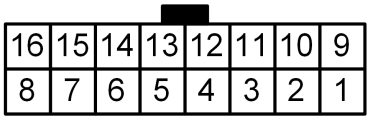


Table 18.23: BMS-Slave Board, battery cell voltage connector

Connector Pin	Battery Cell
1	VBAT-
2	0+ (1-)
3	2+ (3-)
4	4+ (5-)
5	6+ (7-)
6	8+ (9-)
7	10+ (11-)
8	NC
9	0-
10	1+ (2-)
11	3+ (4-)
12	5+ (6-)
13	7+ (8-)
14	9+ (10-)
15	11+
16	VBAT+

In case no cells are available, they can be simulated with a series of voltage divider. A voltage supplied of 30V should be used and 12 resistors with the same value connected in series between the positive and negative connectors of the voltage supply. The positive connector is linked to VBAT+, the negative connector to VBAT- and each pole of a resistor correspond to a pole of a battery cell. The voltage of 30V is chosen so that every simulated cell voltage lies around 2.5V, which lies in the center of the safe operating area defined by default in the foxBMS software.

18.4.3 Cell Temperature Connector on the foxBMS Slave Units

The connector indicated as `Temperature sensor connector (24 pin)` in fig. 18.4 is used to connect temperature sensors to the BMS-Slave Board.

Table 18.24 describes the temperature connector.

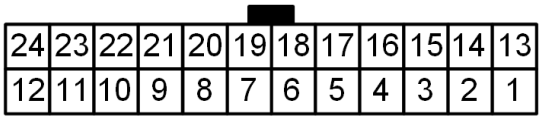


Table 18.24: Temperature Sensor Connector

Connector Pin	Temperature Sensor
1, 24	T-Sensor 0
2, 23	T-Sensor 1
3, 22	T-Sensor 2
4, 21	T-Sensor 3
5, 20	T-Sensor 4
6, 19	T-Sensor 5

Fig. 18.5 shows the functioning of a temperature sensor.

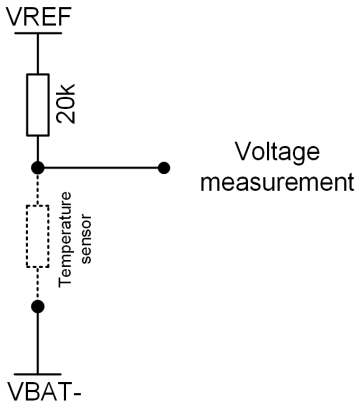


Fig. 18.5: Temperature sensor circuit.

The voltage VREF (3V) is generated by the LTC6804-1 chip. A temperature-dependent resistor must be added to build a voltage divider (drawn as a dashed line in fig. 18.5, not delivered with the BMS-Slave Boards). The resulting voltage is measured by the LTC6804-1 chip. Knowing the temperature dependence of the resistor, the relation between measured voltage and temperature can be determined.

A function is present in the code to make the conversion between measured voltage and temperature. It must be adapted to the temperature sensor used. This is described in the software FAQ (*How to change the relation between voltages read by multiplexer via LTC6811-1 and temperatures?*). In case temperatures are read incorrectly, this function is the first step to verify.

It must be noted that if no sensor is connected, 3V are measured. For the quickstart guide, no sensor needs to be connected: the conversion function is simply a multiplication by 10, so 30°C will be displayed, which again lies in the center of the safe operating area defined by default in the foxBMS software.

If sensors are added, they must be connected between the connector pins corresponding to the sensors 0 to 5, as shown in table 18.24.

18.5 Slave 18-Cell `v1.1.3 and above`

18.5.1 Overview

Important: The following description only applies for the 18-cell BMS-Slave Board hardware versions `1.1.3` and above.

Hint: All connector pinouts described below follow the *Convention for Connector Numbering*.

Block Diagram

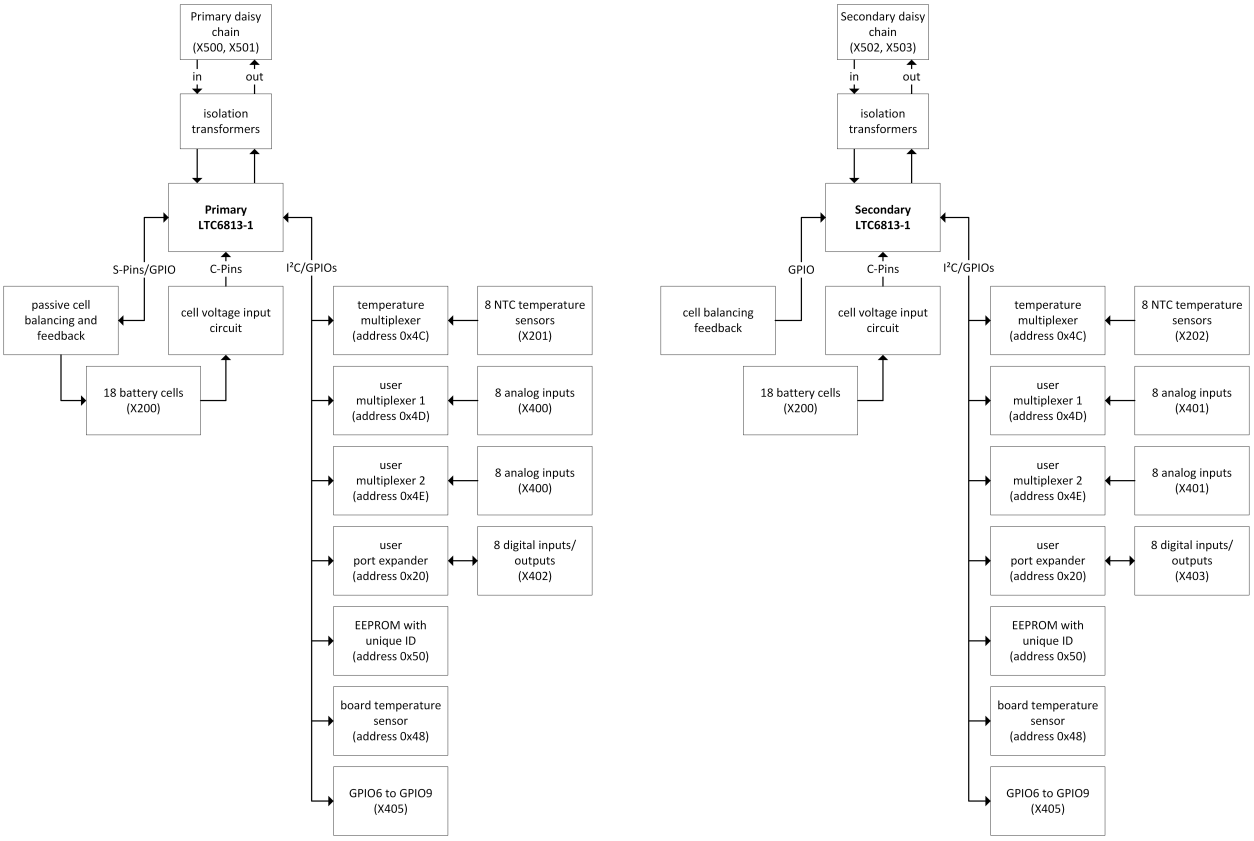


Fig. 18.6: foxBMS BMS-Slave 18-Cell Block Diagram

Schematic and Board Layout

More information about the board schematic and layout files can be found in section *Design Resources*.

Mechanical Dimensions

The size of the foxBMS Slave PCB is 160x100mm. A `*.step` file and a `3D-PDF` of the PCB can be found in section *Design Resources*.

18.5.2 Functions

The following general descriptions apply to both, the primary and the secondary unit on the foxBMS BMS-Slave. If there are any differences in hardware between the primary and the secondary unit they will be marked as such.

Cell Voltage Measurement

The cell voltage sense lines are input on the connector X200. The pinout is described in table 18.25.

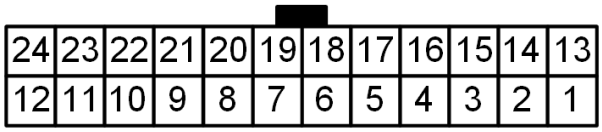


Table 18.25: Cell voltage sense connector

Pin	Signal	Direction	Description
1	VBAT-	Input	Battery module negative terminal
2	CELL_0+	Input	Cell 0 positive terminal
3	CELL_2+	Input	Cell 2 positive terminal
4	CELL_4+	Input	Cell 4 positive terminal
5	CELL_6+	Input	Cell 6 positive terminal
6	CELL_8+	Input	Cell 8 positive terminal
7	CELL_10+	Input	Cell 10 positive terminal
8	CELL_12+	Input	Cell 12 positive terminal
9	CELL_14+	Input	Cell 14 positive terminal
10	CELL_16+	Input	Cell 16 positive terminal
11	VBAT+	Input	Battery module positive terminal
12	NC	–	–
13	CELL_0-	Input	Cell 0 negative terminal
14	CELL_1+	Input	Cell 1 positive terminal
15	CELL_3+	Input	Cell 3 positive terminal
16	CELL_5+	Input	Cell 5 positive terminal
17	CELL_7+	Input	Cell 7 positive terminal
18	CELL_9+	Input	Cell 9 positive terminal
19	CELL_11+	Input	Cell 11 positive terminal
20	CELL_13+	Input	Cell 13 positive terminal
21	CELL_15+	Input	Cell 15 positive terminal
22	CELL_17+	Input	Cell 17 positive terminal
23	NC	–	–
24	NC	–	–

Each of these lines is fused with a fast acting 250 mA surface-mount fuse (F301 - F319) on the board except of the VBAT+ and VBAT- lines which are fused with a value of 500 mA (F300 and F320). This essentially is important for an evaluation environment. The VBAT+ and VBAT- connection is used for the internal power supply of the slave board. If the battery module does not contain these separate wires to the positive and negative module terminal the solder

jumpers SJ300 and SJ301 have to be shorted. In this case the slave will be supplied through the sense lines CELL_0- and CELL_11+. Running the slave in this configuration could result in cell measurement errors due to voltage drop on the sense wires.

The cell input lines are filtered by a grounded or differential capacitor filter (both possibilities provided on the PCB). More information on the corner frequency of this filtering can be found in the schematic. The grounded capacitor filter should be used in environments affected with high noise as it offers a high level of battery voltage ripple rejection. The differential capacitor filter can be used when noise is less occurrent or the design is subjected to cost optimization.

Passive Cell Balancing

The passive balancing circuit is realized by a parallel connection of two 130 Ohm discharge-resistors that can be connected to each single cell in parallel. The MOSFET switches (T1500 - T1517) that control the connection to the cells are controlled by the primary LTC6813-1 monitoring IC. The LTC6813-1 on the secondary unit does not support balancing. The resistor value of 2x 130 Ohm results in a balancing current of about 55mA at a cell voltage of 3.6V. This current results in a power dissipation of about 0.2W per balancing channel (at 3.6V).

Global Cell Balancing Feedback

In order to check the proper function of the balancing process or to detect a malfunction in the balancing control circuit, a global balancing feedback signal is connected to the LTC6813-1. This allows the system to check whether any balancing action is currently taking place at any time. The feedback signal is connected to the GPIO3 of the LTC6813-1. The signal remains in a logic zero state until any balancing action on any cell starts.

Temperature Sensor Measurement

The cell temperature sensors are connected to the connectors X201 (primary) and X202 (secondary). The pinout is identical for the primary and secondary unit and is described in table 18.26.

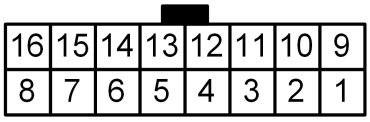


Table 18.26: Temperature sensor connector

Pin	Signal	Direction	Description
1	T-SENSOR_0	Input	NTC Sensor 0 terminal 1
2	T-SENSOR_1	Input	NTC Sensor 1 terminal 1
3	T-SENSOR_2	Input	NTC Sensor 2 terminal 1
4	T-SENSOR_3	Input	NTC Sensor 3 terminal 1
5	T-SENSOR_4	Input	NTC Sensor 4 terminal 1
6	T-SENSOR_5	Input	NTC Sensor 5 terminal 1
7	T-SENSOR_6	Input	NTC Sensor 6 terminal 1
8	T-SENSOR_7	Input	NTC Sensor 7 terminal 1
9	FUSED_VBAT-	Input	NTC Sensor 0 terminal 2
10	FUSED_VBAT-	Input	NTC Sensor 1 terminal 2
11	FUSED_VBAT-	Input	NTC Sensor 2 terminal 2
12	FUSED_VBAT-	Input	NTC Sensor 3 terminal 2
13	FUSED_VBAT-	Input	NTC Sensor 4 terminal 2
14	FUSED_VBAT-	Input	NTC Sensor 5 terminal 2
15	FUSED_VBAT-	Input	NTC Sensor 6 terminal 2
16	FUSED_VBAT-	Input	NTC Sensor 7 terminal 2

Standard 10kΩ NTC resistors (e.g. Farnell-Nr. 1299926) are recommended for use. When using other values than these, the series resistors (R100-R107) on the board may have to be adjusted. Please note that the accuracy of the internal voltage reference VREF2 decreases heavily with a load of over 3mA. Using 8x 10kΩ NTC resistors with the corresponding 10kΩ series resistors results in a current of 1.2mA (at 20°C) which is drawn from VREF2.

Each 8 temperature sensors are connected to an analog multiplexer. The analog multiplexer can be controlled via I^2C by the LTC6813-1 (7-bit address: 0x4C). In order to ensure fast settling times after switching the multiplexer input, the output signal of the multiplexer is buffered by an operational amplifier. Finally the analog voltage of the selected sensor is measured on the GPIO1 pin of the LTC6813-1.

On-board EEPROM

The primary unit as well as the secondary unit of the foxBMS BMS-Slave board is equipped with an EEPROM IC (IC801). The EEPROM for example can be used for storing data such as calibration values. Similar to the analog multiplexers, the EEPROM device is connected to the I^2C bus of the LTC6813-1 (7-bit address: 0x50).

On-board Ambient Temperature Sensor

For an additional monitoring of the ambient temperature an on-board temperature sensor is used. This temperature sensor can be read by the LTC6813-1 via the I^2C bus (7-bit address: 0x48). It is possible to program an alert temperature. Once the measured temperature reaches this alert temperature the alert pin of the IC is set to a logic low level. Currently this signal is not used on the BMS-Slave board, but it is accessible on the connector X404.

Additional Inputs and Outputs

Several additional analog and digital inputs and outputs are provided on the BMS-Slave board via pin headers. Each 16 analog inputs are provided on connector X400 (primary) and X401 (secondary). The pinout for the connectors for the primary and secondary unit is identical and is described in table 18.27.

Table 18.27: Connector for analog inputs

Pin	Signal	Direction	Description
1	ANALOG-IN_0	Input	Analog input 0
2	ANALOG-IN_1	Input	Analog input 1
3	ANALOG-IN_2	Input	Analog input 2
4	ANALOG-IN_3	Input	Analog input 3
5	ANALOG-IN_4	Input	Analog input 4
6	ANALOG-IN_5	Input	Analog input 5
7	ANALOG-IN_6	Input	Analog input 6
8	ANALOG-IN_7	Input	Analog input 7
9	ANALOG-IN_8	Input	Analog input 8
10	ANALOG-IN_9	Input	Analog input 9
11	ANALOG-IN_10	Input	Analog input 10
12	ANALOG-IN_11	Input	Analog input 11
13	ANALOG-IN_12	Input	Analog input 12
14	ANALOG-IN_13	Input	Analog input 13
15	ANALOG-IN_14	Input	Analog input 14
16	ANALOG-IN_15	Input	Analog input 15
17	+3.0V_VREF2	Output	LTC6813-1 3.0V voltage reference
18	FUSED_VBAT-	Output	GND

Each 8 analog inputs are connected to an analog multiplexer. The analog multiplexers can be controlled via I^2C by the LTC6813-1 (7-bit addresses: 0x4D and 0x4E). In order to ensure fast settling times after switching the multiplexer input, the output signals of the multiplexers are buffered by operational amplifiers. Finally the analog voltage of the selected sensor can be measured on the GPIO2 pin of the LTC6813-1.

Each 8 digital inputs/outputs are provided on the connectors X402 (primary) and X403 (secondary). The pinout for the connectors for the primary and secondary unit is identical and is described in table 18.28.

Table 18.28: Connector for digital IOs

Pin	Signal	Direction	Description
1	DIGITAL-IO_0	Input/Output	Digital input/output 0
2	DIGITAL-IO_1	Input/Output	Digital input/output 1
3	DIGITAL-IO_2	Input/Output	Digital input/output 2
4	DIGITAL-IO_3	Input/Output	Digital input/output 3
5	DIGITAL-IO_4	Input/Output	Digital input/output 4
6	DIGITAL-IO_5	Input/Output	Digital input/output 5
7	DIGITAL-IO_6	Input/Output	Digital input/output 6
8	+5.0V_VREG	Output	LTC6813-1 5.0V regulated voltage
9	FUSED_VBAT-	Output	GND

Each 8 digital inputs/outputs are connected to an I^2C controlled port expander (7-bit address: 0x20). The direction of the inputs/outputs as well as the logic levels on the pins can be selected by register settings. Each of the 8 digital inputs/outputs has a discrete pull up resistor that for example can be used for directly connecting a tactile switch.

isoSPI Daisy Chain Connection

The data transmission between the slaves and between the slaves and the basic board takes place using the isoSPI interface. The isoSPI signals are input/output on the connectors X500/X501 (primary) and X502/X503 (secondary). The isoSPI ports are bidirectional, that means they can be used in forward and reverse direction. The isoSPI connections are isolated galvanically using pulse transformers (TR1400). The voltage amplitude of the differential signal can be adjusted by setting resistors (see section *Daisy Chain Communication Current*).

The pinout of the isoSPI connectors is described in table 18.29 and table 18.30.

Table 18.29: isoSPI Daisy Chain Input Connectors

Connector Pin	Daisy Chain
1	IN+ (Primary/Secondary LTC6813-1)
2	IN- (Primary/Secondary LTC6813-1)

Table 18.30: isoSPI Daisy Chain Output Connectors

Connector Pin	Daisy Chain
1	OUT+ (Primary/Secondary LTC6813-1)
2	OUT- (Primary/Secondary LTC6813-1)

Hardware Settings / Options

Software Timer

The internal software timer of the LTC6813-1 can be enabled/disabled by a dedicated external pin (SWTEN, pin 36 of the LTC6813-1). In order to support all features, the foxBMS BMS-Slave board offers a possibility to switch the software timer. The software timer is enabled in the standard configuration, which means pin 36 is pulled to VREG via a zero-ohm resistor (R1407). The timer can be disabled by removing the resistor R1407 and placing a zero-ohm resistor to R1406.

Daisy Chain Communication Current

The daisy chain communication current can be set by the resistors R1400 and R1402. The default value is 820Ω for R1402 and 1.2kΩ for R1400. This values result in a bias current of approx. 1mA and a differential signal amplitude of 1.18V. Theses values are suitable for high noise environments with cable lengths of over 50m. For more information please have a look at the LTC6813-1 datasheet.

Status LED

The status LEDs LD1400 show the current mode of each, the primary and secondary LTC6813-1. The LED is on in STANDBY, REFUP or MEASURE mode, whereas the LED is off in SLEEP mode. The LED can be disabled by removing the resistor R1403 next to the LED.

GPIO Extension Connector

The internal GPIO lines 1 to 5 of the primary or secondary LTC6813-1 can be connected to the GPIO extension pin header X404 via optional zero-ohm resistors. In the standard configuration these resistors are not placed. Of course it is possible to place each both resistors for a parallel connection of the internal signals to the GPIO extension connector. For more information see the corresponding page of the schematics. The placement of the resistors and the resulting connection is shown in table 18.31.

Table 18.31: GPIO extension connector

GPIO	connect to pin header	connect to internal function
1	R1408	R1409 (default)
2	R1410	R1411 (default)
3	R1412	R1413 (default)
4	R1414	R1415 (default)
5	R1416	R1417 (default)

The pinout of the extension connector X404 is described in table 18.32.

Table 18.32: Extension connector

Pin	Signal	Direction	Description
1	+3.0V_VREF2_0	Output	Primary LTC6813-1 3.0V reference voltage 2
2	+3.0V_VREF2_1	Output	Secondary LTC6813-1 3.0V reference voltage 2
3	+5.0V_VREG_0	Output	Primary LTC6813-1 5.0V regulated voltage
4	+5.0V_VREG_1	Output	Secondary LTC6813-1 5.0V regulated voltage
5	PRIMARY-GPIO1-OPT	Input/Output	Primary LTC6813-1 GPIO1
6	SECONDARY-GPIO1-OPT	Input/Output	Secondary LTC6813-1 GPIO1
7	PRIMARY-GPIO2-OPT	Input/Output	Primary LTC6813-1 GPIO2
8	SECONDARY-GPIO2-OPT	Input/Output	Secondary LTC6813-1 GPIO2
9	PRIMARY-GPIO3-OPT	Input/Output	Primary LTC6813-1 GPIO3
10	SECONDARY-GPIO3-OPT	Input/Output	Secondary LTC6813-1 GPIO3
11	PRIMARY-GPIO4-OPT	Input/Output	Primary LTC6813-1 GPIO4
12	SECONDARY-GPIO4-OPT	Input/Output	Secondary LTC6813-1 GPIO4
13	PRIMARY-GPIO5-OPT	Input/Output	Primary LTC6813-1 GPIO5
14	SECONDARY-GPIO5-OPT	Input/Output	Secondary LTC6813-1 GPIO5
15	PRIMARY-WDT	Output	Primary LTC6813-1 watchdog output
16	SECONDARY-WDT	Output	Secondary LTC6813-1 watchdog output
17	PRIMARY-TEMP-ALERT	Output	Primary board temp. sensor alarm output
18	SECONDARY-TEMP-ALERT	Output	Secondary board temp. sensor alarm output
19	FUSED_VBAT-	Output	GND
20	FUSED_VBAT-	Output	GND

The GPIO lines 6 to 9 are wired to the connector X405 permanently. There is no internal function for this GPIO lines. The pinout of the extension connector X405 is described in table 18.33.

Table 18.33: Additional GPIO extension connector

Pin	Signal	Direction	Description
1	PRIMARY-GPIO6	Input/Output	Primary LTC6813-1 GPIO6
2	SECONDARY-GPIO6	Input/Output	Secondary LTC6813-1 GPIO6
3	PRIMARY-GPIO7	Input/Output	Primary LTC6813-1 GPIO7
4	SECONDARY-GPIO7	Input/Output	Secondary LTC6813-1 GPIO7
5	PRIMARY-GPIO8	Input/Output	Primary LTC6813-1 GPIO8
6	SECONDARY-GPIO8	Input/Output	Secondary LTC6813-1 GPIO8
7	PRIMARY-GPIO9	Input/Output	Primary LTC6813-1 GPIO9
8	SECONDARY-GPIO9	Input/Output	Secondary LTC6813-1 GPIO9

External Isolated DC-Supply

It is possible to supply the BMS-Slave Board by an external DC power supply with a voltage range of 8V to 24V. The DC input is protected against reverse voltage and over-current (with a 1.25A fuse). The external DC supply has to be connected on connector X1001 or X1002 (both connectors are in parallel for daisy chaining the supply). The pinout of the connectors X1001 and X1002 is shown in table 18.34.

Table 18.34: External DC supply connector

Pin	Signal	Direction	Description
1	DC+	Input	positive supply terminal
2	DC-	Input	negative supply terminal

CHAPTER 19

Design Resources

The hardware design packages for the BMS-Master Board, the BMS-Extension Board, the BMS-Interface Board and the BMS-Slave Boards are available in the common GitHub repository. The packages include:

- Altium Designer Source Files
  - schematics
  - layout
  - active bill of materials
- Assembly files
  - BOM in Excel format
  - 3D model of PCB in step format
  - schematics in PDF format
- Fabrication files
  - PCB manufacturing data in ODB++ format
  - PCB layer stack in PDF format

Note: To open the schematic and layout files, please use Altium Designer.

Note: To manufacture the printed circuit boards, the BOM (Microsoft Excel file) and the ODB++ files in each corresponding folder should be used. Before sending the board layout to a PCB manufacturer, the layout files must be checked against the design rules provided by this manufacturer, since some board layout settings may depend on its specific design rules and may cause violations (e.g., pad layout).

CHAPTER 20

Safety Components

Before using foxBMS Master Unit and the foxBMS Slave Units with lithium-ion batteries, safety must be considered very carefully. The following parts and components are recommended to be used with the foxBMS Master Unit and the foxBMS Slave Units.

20.1 DC Battery Pack Fuse

When using foxBMS with batteries, a DC fuse in the current path is mandatory. In case of external short circuit of the battery pack, this fuse must be able to break the current at the maximum high-voltage of the complete battery. This DC fuse is a critical safety component and needs to be selected very carefully. The different parameters of the specific battery pack used must be taken into account (e.g., maximum voltage, expected short circuit current, continuous current).

Fig. 20.1 shows how to set up a basic battery system using these components. In the section *Battery Junction Box*, details can be found on how to use these and other components depending on the used battery system.

20.2 Power Contactors

To switching high-current capable batteries, mechanical power contactors (power relays) shall be used. For safety reasons, contactors with auxiliary contacts providing true state feedback are strongly recommended. foxBMS was developed and tested with the contactors of type Gigavac GX14 (GX14BAB: 12V coil voltage and normally open auxiliary contacts), but will probably work with many other contactors. The most important safety parameters are the maximum break current and the maximum allowed voltage to be broken. This must carefully be checked by the user of the foxBMS research and development platform. Further, the pick-up and continuous current of the contactor coil must be checked to be within the foxBMS electrical ratings. Nevertheless, contactors matching the used battery pack specification (e.g., continuous current, maximum voltage, short circuit current) are mandatory.

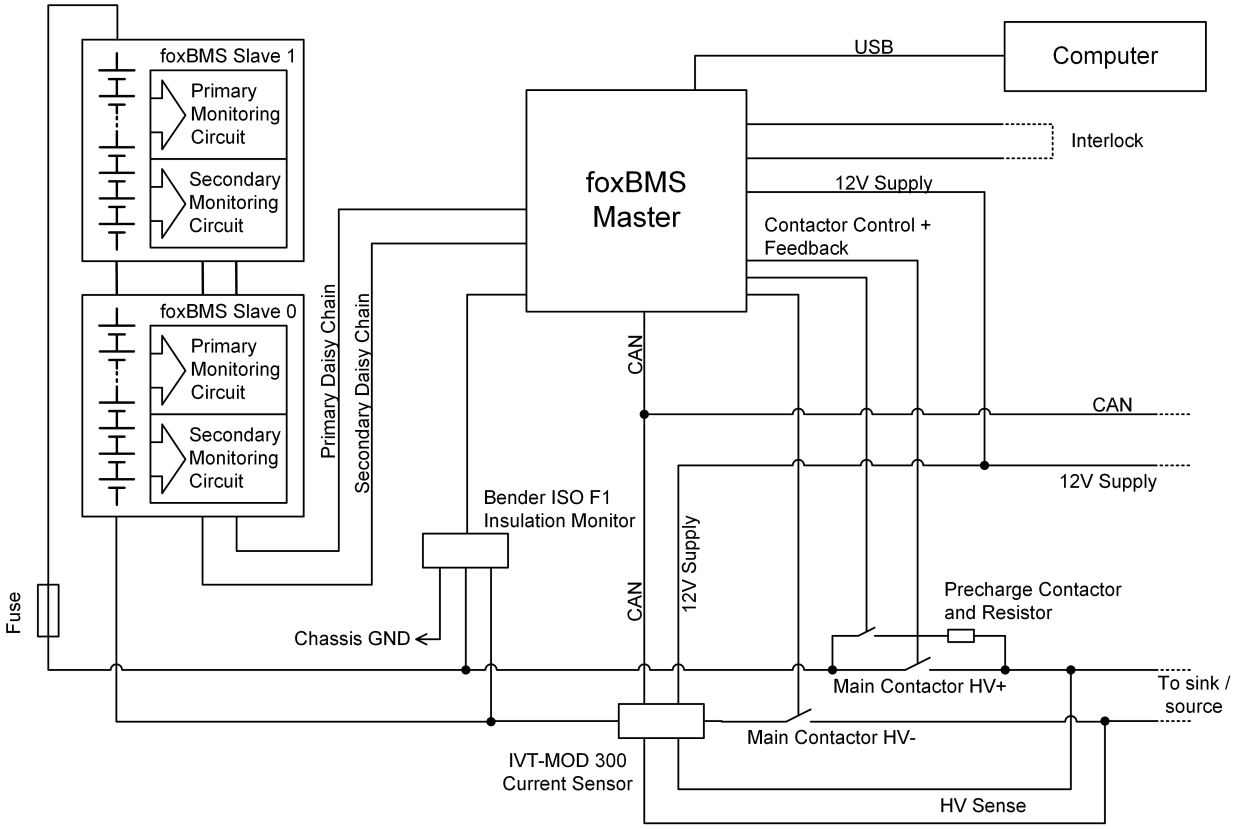


Fig. 20.1: Block diagram showing the typical topology of a battery system

20.3 Current Sensor

To measure the battery pack current, CAN based current sensors can be used. The Isabellenhütte IVT-MOD and IVT-S shunt based current sensors were tested with foxBMS. The automotive current sensor IVT-MOD 300 provides a current measurement range of +/-300A and in addition three voltage measurement inputs with a voltage measurement range of +/-600V. Current sensors with higher current measurement ranges are available. Please check the Isabellenhütte homepage and contact a salesperson to get more information. If another CAN based current sensor is used, the embedded software on the BMS-Master Board must be adapted.

20.4 Insulation Monitor

foxBMS supports an insulation monitoring device (i.e., also called a ground fault detector) to detect faulty insulation of the battery pack against chassis, ground or ~~or~~ battery pack enclosure. Bender insulation monitor IR155-3203/-3204/-3210 are supported and tested, while IR155-3204 is recommended. The Bender ISOMETER IR155-3203/-3204 monitors the insulation resistance between the insulated and active HV-conductors of an electrical drive system (up to 1000VDC) and the reference ground. The fault messages (insulation fault at the HV-system, connection or device error) will be provided at the integrated and galvanic isolated interface. The interface consists of a status output (called OKHS output) and a measurement output (called MHS/MLS output). The status output signalises errors or that the system is error free. The measurement output signalises the actual insulation resistance. Furthermore, it is possible to distinguish between different fault messages and device conditions, which are base frequency encoded. For details, consult the corresponding device datasheet or manual. The insulation resistance threshold can be factory programmed, together with ~~as well as~~ an additional undervoltage threshold. Any violation of thresholds causes an error signal on the status pin.

According to ECE R-100 the insulation resistance between the high voltage bus and the electrical chassis shall have a minimum value of 100Ohm/V of the working voltage for DC buses, and a minimum value of 500Ohm/V of the working voltage for AC buses (e.g., for a 800V system, the insulation resistance threshold should be selected for 400kOhm).

The following section describes how to test the Bender functionality.

20.4.1 Required Hardware

Before starting, following items are needed:

- BMS-Master Board
- Bender insulation monitor IR155-3204/-3210
- Lauterbach debugger
- Resistors (1kOhm, 220kOhm, 470kOhm, 1MOhm)
- Terminal block for simple wire connection
- This documentation

20.4.2 Electrical Connections

The follwong figure displays the the connectors of the Bender insulation monitor.

Connect the Bender as follwing:

- connect the bender to foxBMS as described in chapter *Specifications*
- connect line voltage L-(connector XLA-) to chassis ground (Pin 3 and 4 of connector XK1A)

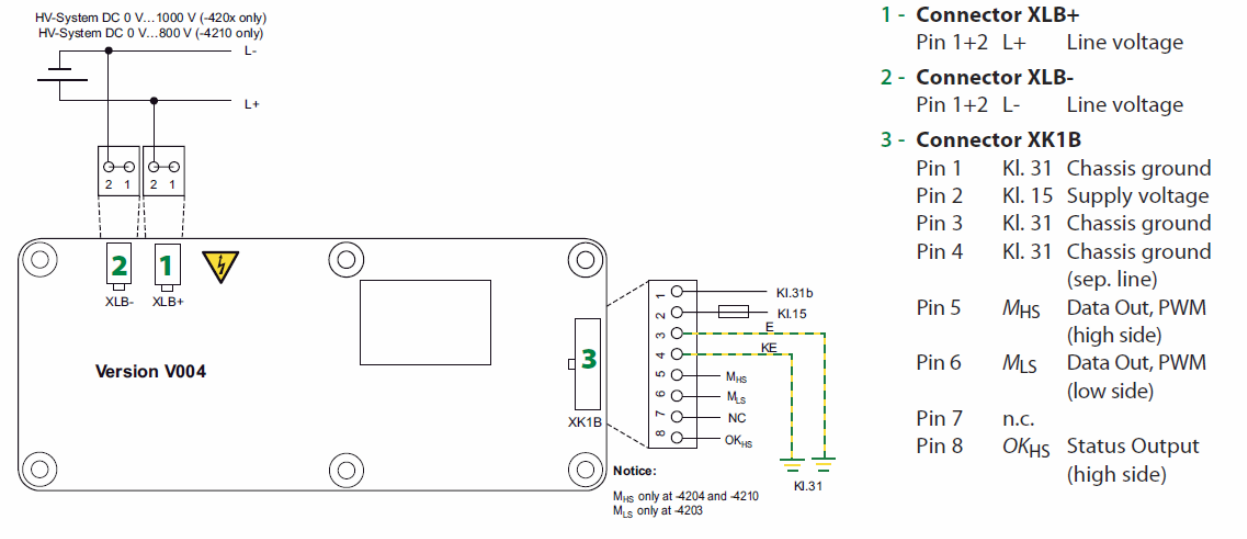


Fig. 20.2: Wiring of the Bender insulation monitor (image source: Bender Datasheet)

- connect one of the resistors between chassis ground/line voltage L-(XLA-) and line voltage L+(XLA+)

Depending on the connected resistor, different insulation values are measured.

CHAPTER 21

Battery Junction Box

This section presents an example of a typical battery junction box. The design must be adapted to the application and specific care must be taken by sizing the components to ensure the required safety level.

21.1 Introduction

21.1.1 System Specifications

The Battery Junction Box (BJB) described in this document is designed for mobile and stationary battery systems. The target system has a voltage range between 315V and 567V, while being capable to source and sink a continuous current of up to 320A. The stationary battery consists of up to 14 series connected battery modules (i.e., 28 daisy chained LTC6804). Each module is a 15s2p configuration of 2.3V 20Ah lithium-ion NMC/LTO prismatic battery cells.

21.1.2 System Overview

The BJB contains the Battery Management System (BMS) and all safety relevant components. Fig. 21.1 shows a BJB integrated into a 19" enclosure with a 4U height.

Fig. 21.2 shows the block diagram of the BJB. The battery is connected to the BJB at the left side (`Batt_Positive` and `Batt_Negative`). The source/sink (e.g., load, charger, inverter) is connected to the right side (`Out_Positive` and `Out_Negative`).

The main contactors disconnect the battery from the output terminals. These are normally off and switched on in BMS ON-Mode. The contactors are opened if the BMS detects hazardous conditions (e.g., abnormal battery cell temperature) or in BMS OFF-Mode.

Note: A precharge contactor and resistor are used to limit the inrush current into the inverter DC-link capacitor when closing the main contactors at startup.

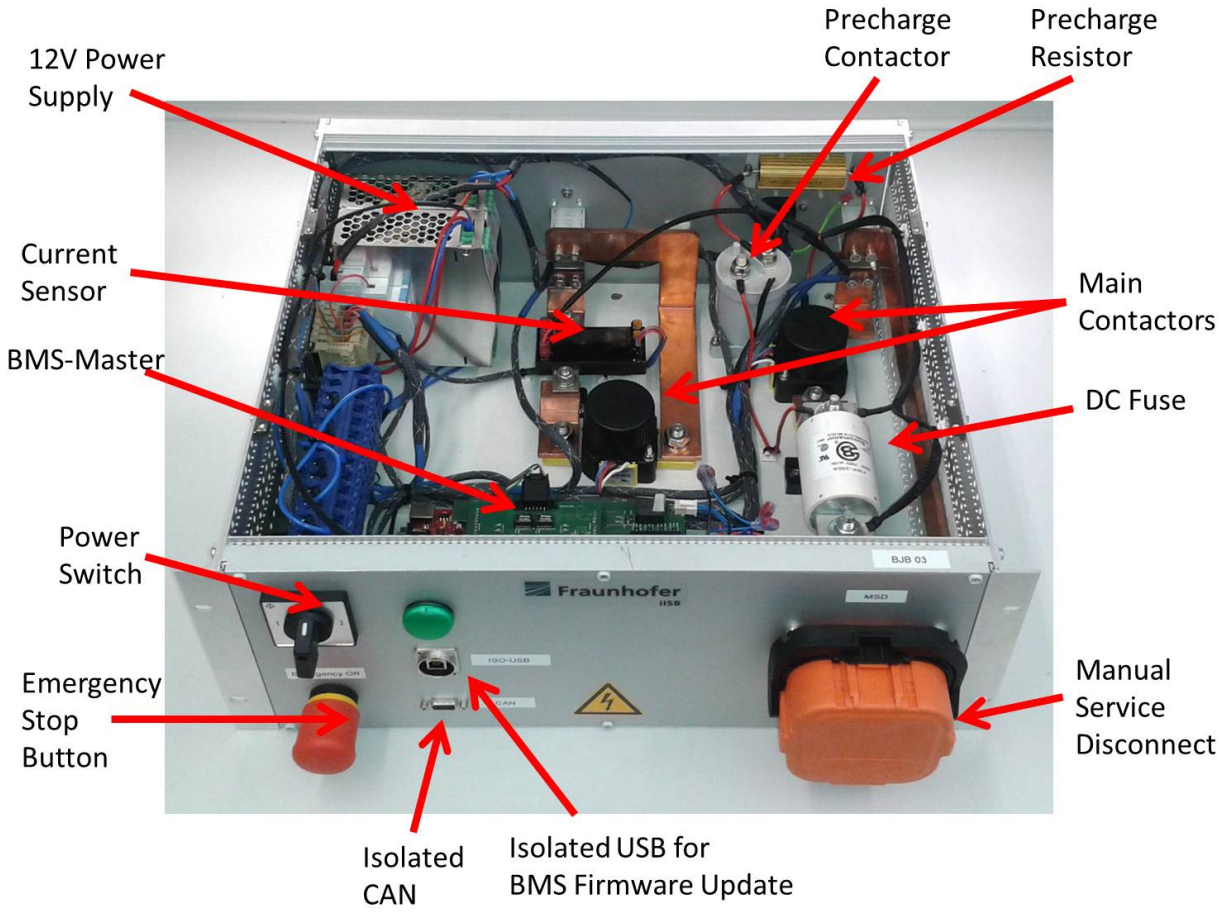


Fig. 21.1: BJB top view

To break the current flowing in case a short circuit condition occurs between the two high voltage battery system poles, a fuse is placed in the positive current path. A Cooper Bussmann FWP-Series (700Vdc) or better FWJ-Series (800Vdc) fuse was selected for this purpose.

Danger: A special high-voltage DC fuse must be used to break high short circuit currents in high voltage battery systems. This fuse must be very carefully chosen by electrically skilled engineers to ensure proper protection in the specific test environment using specific testing equipment and test conditions. In case the fuse is not appropriately chosen, its protection effect will not be provided.

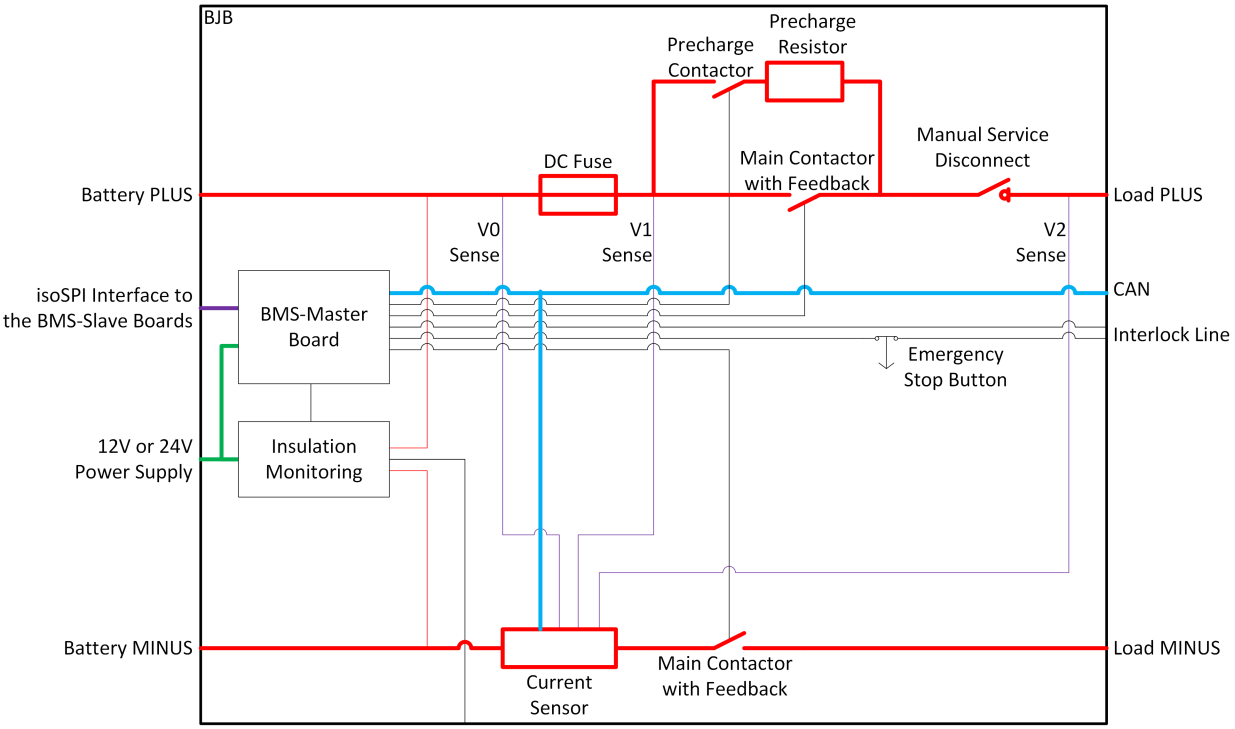


Fig. 21.2: BJB block diagram

A Manual Service Disconnect (MSD) is placed in the positive current path to ensure a manual disconnection while the system is serviced. It is mounted on the front of the BJB for easy access. The MSD used is available with an integrated fuse or as shunted version without fuse. In case the fused version is chosen, the Cooper Bussmann fuse can be omitted, but the integrated fuse version is limited to 200A.

The Battery Management System (BMS) is the main control unit of the whole battery system. It collects data from the battery modules (e.g., battery cell voltages, cell temperatures) and from the current sensor, and uses these data for battery state calculations (e.g., SOC, SOH, SOF). In addition, the BMS controls the power contactors and communicates with a superior management unit through the CAN bus.

21.2 Battery Junction Box Part List

21.2.1 Commercial Off-The-Shelf (COTS) Parts

Table 21.1 shows the components used in the BJB. All components were selected to fit the system specifications listed in *System Specifications*. If another battery configuration than specified here is used, the voltage and current ratings of

these components have to be checked and adapted.

Table 21.1: BJB part list

Part Description	Manufacturer	Part Suggested	Supplier
Power Contactors	Gigavac	GX16BEB (600A)	HVC Technologies
Precharge Contactor	TE	LEV200A4ANA	Mouser
Precharge Resistor	Vishay	LPS 300 Series (300W)	Farnell
DC Fuse*	Cooper Bussmann	FWJ-Series or FWP-Series	Mouser
Current Sensor	Isabellenhuette	IVT-MOD or IVT-S	Isabellenhuette
Emergency Stop Button	Moeller	M22-PV/K11 + M22 K01	Farnell
Insulation Monitoring	Bender	ISOMETER IR155-3204	Bender
BMS	Fraunhofer IISB	foxBMS Master Unit with foxBMS Slave Units	Fraunhofer IISB
Manual Service Disconnect	TE	AMP + Manual Service Disconnect	Power & Signal Group
12V Power Supply	Meanwell	WDR-120-12	Mouser
Power Switch	Kraus & Naimer	G20S D322-600 E	Kraus & Naimer

* rated currents and voltages are depending from the used battery cells

21.2.2 Custom Parts

In addition to the components listed in table 21.1, sundries not listed here are needed (e.g., terminal blocks). Table 21.2 shows the wires used inside the BJB. Basically only 0.5mm^2 wires are used. Wires are used for signals and low voltage only (e.g., 12V or 24V power supply). The continuous insulation rating must exceed the maximum voltage of the whole battery pack.

Table 21.2: Litz wires used inside the BJB

Color	Cross Section	Usage
Red	0.5mm^2	12V Supply Positive
Blue	0.5mm^2	12V Supply Ground
Orange	0.5mm^2	Insulation Monitoring
Blue/Yellow	0.5mm^2	Insulation Monitoring
Brown/Grey	0.5mm^2	CAN High
Brown/Orange	0.5mm^2	CAN Low
Green/Orange	0.5mm^2	Interlock Line
Green	0.5mm^2	Battery Monitoring Backup Interface
Grey/White	0.5mm^2	Battery Monitoring Backup Interface
Brown/Purple	0.5mm^2	Battery Monitoring Main Interface
Red/Green	0.5mm^2	Battery Monitoring Main Interface

For the high power DC connections, copper bus bars are used, since high current pulses can occur. The copper bus bars were fabricated with a cross section of 150mm^2 (5mm x 30mm) for the maximum continuous current specified in *System Specifications*.

21.3 Main and Precharge Contactors Wiring

The main and precharge contactors are delivered with bare wire ends. The corresponding crimps and plugs must be used to connect them to the foxBMS Master Unit. The direction of the current flow through the power contactors must

be carefully considered, since the ability of the power contactor to interrupt the current is dependant on the direction of the current flowing through them.

21.4 Insulation Monitor Wiring

The Bender insulation monitor is delivered with a socket on its PCB. The corresponding crimps and plugs must be used used to connect it to the foxBMS Master Unit.

21.5 Current Sensor Wiring

The current sensor is supplied without its crimps and housings (e.g., by JST). The current sensor has to be wired to the CAN bus and to a power supply. In addition, voltage sense wires may be connected to measure up to 3 voltages in the battery system (e.g., battery voltage, voltage over the DC fuse, voltage over the contactors, voltage on the load side). In the given BJB example, the voltage sensing inputs are monitoring the following:

- Voltage measurement 1: between fuse and main contactor
- Voltage measurement 2: between fuse and service disconnect (MSD)
- Voltage measurement 3: between battery positive and precharge contactor

---

Note: When the precharge contactor is closed after the main negative contactor has been closed, the measured voltages are used to ensure a correct precharge procedure.

21.6 Summary of the Assembly Procedure

For developing and building the BJB, the following procedure may be used:

1. Defining the specification of the battery (e.g., maximum current, maximum voltage)
2. Defining the placement of the input (from the battery) and the output (to the user) connectors and the manual service disconnect
3. Placement of the main parts of the current path (e.g., main contactors)
4. Designing of the copper bus bars (alternatively wire with appropriate cross section may be used)
5. Placement of the BJB electronic parts (e.g., BMS)
6. Wiring of the electronic parts

CHAPTER 22

Toolchain

22.1 Layout and schematics

The initial layout and schematics were developed with Cadsoft Eagle version 6.5.0 which is now owned by AUTODESK. Recently, all hardware projects were ported to Altium Designer.

22.2 Information on Debugging

Two types of debugger have been tested with foxBMS.

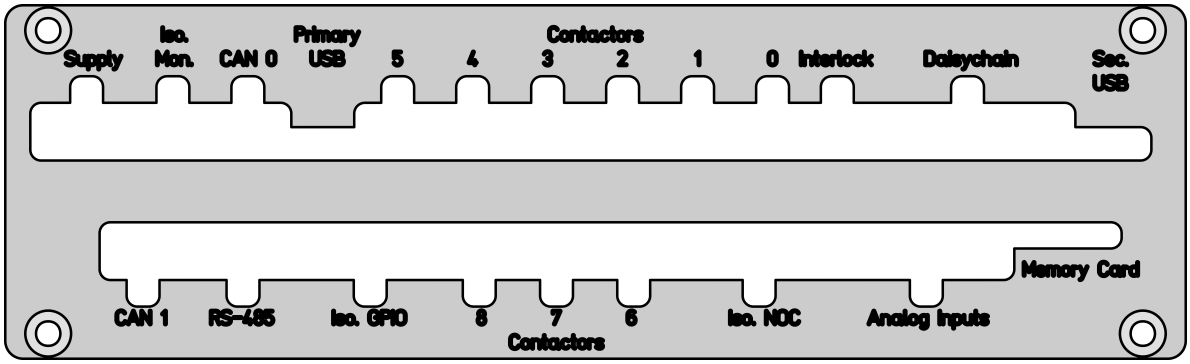
- The first one is the Segger J-Link Plus with the adapter for 19-Pin Cortex-M (the adapter is needed to connect the debugger to the foxBMS Master Unit). A cheaper debugger solution is the Segger J-Link Base which also needs the adapter.
- The second type of debugger is the Lauterbach Debugger µTrace for Cortex-M (see the "Products" section in the navigation bar on the left). The Lauterbach debugger provides more debugging functionalities, but is also more expensive than the J-Link.

CHAPTER 23

Casing

The following parts are used for the casing of the foxBMS Master Unit.

- Fischer Elektronik KO HL 0 120
- Fischer Elektronik KO HL 6 120
- Schaeffer AG foxBMS Custom Frontplate `Frontplatten Designer File, DXF`



CHAPTER 24

Software Components

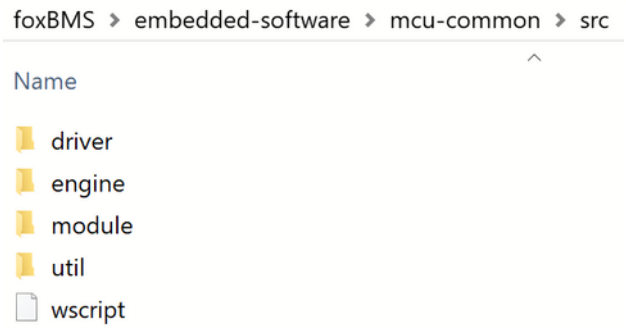
The foxBMS embedded software is made out of the following components:

- `mcu-common`
- `mcu-freertos`
- `mcu-hal`
- `mcu-primary`
- `mcu-secondary`

24.1 mcu-common

The `mcu-common` directory contains drivers that are common to MCU0 and MCU1. This means that any change made in the common directory affects both MCU0 and MCU1.

Folders and files in the mcu-common directory are as follows:

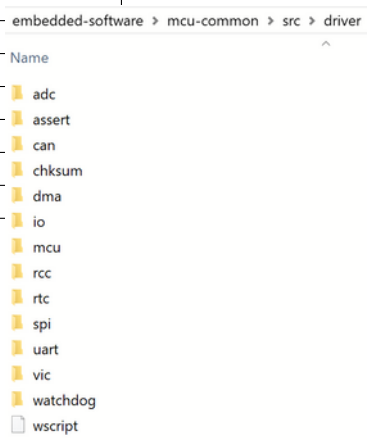


24.1.1 driver

The `driver` directory contains a foxBMS specific abstraction layer for the hardware. in the following directories

assert

Element	Description
adc	Driver for analog to digital converter, measurement of lithium backup battery voltage
can	Driver to receive/transmit CAN message
chksum	Checksum algorithms for modulo 32-bit addition and CRC32
dma	Configuration for Direct Memory Access (e.g. used for SPI Communication)
io	Driver and interfaces for I/O ports (control of output pins and read of input pins)
mcu	Driver for the interlock (Bad naming for this driver)
rcc	Configuration of the prescaler for the MCU clock system
rtc	Real time clock driver, Control and Access of Backup SRAM Registers
spi	Driver for communication via Serial Peripheral Interface (SPI bus)
uart	Driver for serial communication (UART, RS232 , RS485)
vic	Interrupt configuration
watchdog	Driver for the watchdog timer



24.1.2 engine

The `engine` directory contains all the core functions of the BMS. in the following two directories

Element	Description
database	Implementation of the asynchronous data exchange

diag

diag	With this software module, other modules can report problems

24.1.3 module

The `module` directory contains all the software modules needed by the BMS. in the following directories

Element	Description
cansignal	Definition of CAN messages and signals
hwinfo	Functions to return information about the hardware (e.g., cpu temperature)
interlock	Driver for the interlock (Should this be in the driver module?)
led	Functions for using the LEDs
ltc	Driver for battery cell monitoring IC
meas	Uses ltc module to perform measurements

tsensors

24.1.4 util

The `util` directory contains additional functions.

Element	Description
foxmath	Implementation of a math library Not a directory any more.
misc	miscellaneous functions (e.g., functions to reverse a string

Cannot find misc anymore.

24.2 mcu-freertos

The `mcu-freertos` directory contains the operating system software (FreeRTOS).

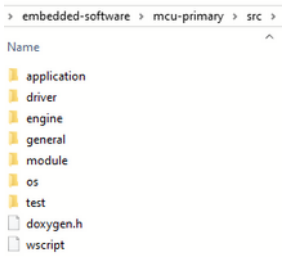
Element	Description
Source	FreeRTOS

24.3 mcu-hal

The `mcu-hal` directory contains the Hardware Abstraction Layer (HAL). It is used by the system but is provided by the MCU manufacturer, in this case ST-Microelectronics. It is used by foxBMS but not part of foxBMS.

Element	Description
CMSIS	Interface and configuration of CMSIS
STM32F4xx_HAL_Driver	STM32F4xx family Hardware Abstraction Layer drivers

24.4 mcu-primary



The `mcu-primary` directory contains the software specific to MCU0.

24.4.1 application

The `application` directory contains the user applications.

Element	Description
algo	Task for integration various algorithms
bal	Driver for balancing
bms	Decision are taken here by the BMS (e.g., open contactors in case of a problem)
com	Serial port communication layer (for debug purposes)
config	Contains the configuration for the user applications (e.g., task configuration)
plausibility	
sox	Coulomb-counter (current integrator) and State-of-Function calculator
task	User specific cyclic tasks (10ms and 100ms)

Application tasks should be used to call user-defined functions.

24.4.2 driver

The `driver` directory contains a foxBMS specific abstraction layer for the hardware.

Element	Description
config	Configuration of the drivers (from mcu-common and primary specific)
sdram	Drivers for the external SDRAM
timer	Drivers for the internal timers

24.4.3 engine

The `engine` directory contains all the core functions of the BMS.

Element	Description
config	Contains the configuration of engine components (e.g., task configuration)
diag	With this software module, other modules can report problems
nvramhandler	Handler for the non volatile memories
sys	System state machine, starts all other state machines
task	Cyclic engine tasks (1, 10 and 100ms) that call system related functions

This is removed to the engine directory of mcu-common.

24.4.4 general

The `general` directory contains the main function and configuration files.

	Element	Description
file	main	Initialization of hardware modules, of interrupts and of the operating system
folder	config	Contains the configuration for the system initialization: configuration and interface functions to HAL and FreeRTOS, global definitions, interrupt configurations and startup code
folder	includes	Contains the standard types
file	version	Sets the version number

24.4.5 module

The `module` directory contains all the software modules needed by the BMS.

Element	Description
config	Contains the configuration for the software modules
contactor	Driver to open/close contactors and read contactor feedback
isoguard	Driver for monitoring galvanic isolation in the system
nvram	Non-volatile Memory: Eeprom and button cell buffered SRAM (BKP_SRAM)

24.4.6 os

The `os` directory contains configurations for the operating system FreeRTOS.

	Element	Description
No folders here.	os	Interface to FreeRTOS (e.g., wrapper functions of cyclic application and engine tasks)

24.4.7 test

24.5 mcu-secondary

Contains the software specific to MCU1. These are the same elements as `mcu-primary`, adapted for MCU1.

CHAPTER 25

Software Architecture

This section describes the foxBMS software architecture. The architecture was chosen to enable a processor independent programming as far as possible. ~~Fig.~~ Fig. 25.1 shows the software architecture of foxBMS.

The software ~~consists~~ is developed on top of a base hardware abstraction layer of the microcontroller (HAL of STM32). The driver layer on top of it is used to configure and initialize the peripherals of the microcontroller. These hardware dependent layers resemble the base of the software architecture. The remaining modules are built on top of it to allow an easy port of the software to different microcontrollers.

25.1 Hardware abstraction layer

The hardware abstraction layer (HAL) consists of the STM32F4xx HAL Drivers `V1.7.4`. Additionaly the Cortex Microcontroller Software Interface Standard (CMSIS) `V2.6.2` is used as an abstraction layer for the Cortex M4F processor.

25.2 Drivers

The driver layer is responsible for the configuration and the initialization of the controller peripherals. These drivers are dependent on the specific used microcontroller and the used HAL. Drivers are only allowed to include files from driver or the lower hardware abstraction layer (i.e. SPI modules includes DMA module to configure the respective DMA stream).

25.3 foxBMS-Modules

foxBMS-modules initialize, configure and control hardware connected to the foxBMS Master Unit. These modules are only allowed to access the hardware through the drivers ~~and directly via~~ which use the HAL. No direct function calls to the HAL ~~should be~~ make the modules as independent as possible from the used microcontroller. They may include engine modules (i.e. database, diag), applications (com) or even the operating system if they need to ensure some runtime specific behaviour. These modules are called within the operating system to ensure a real-time behaviour.

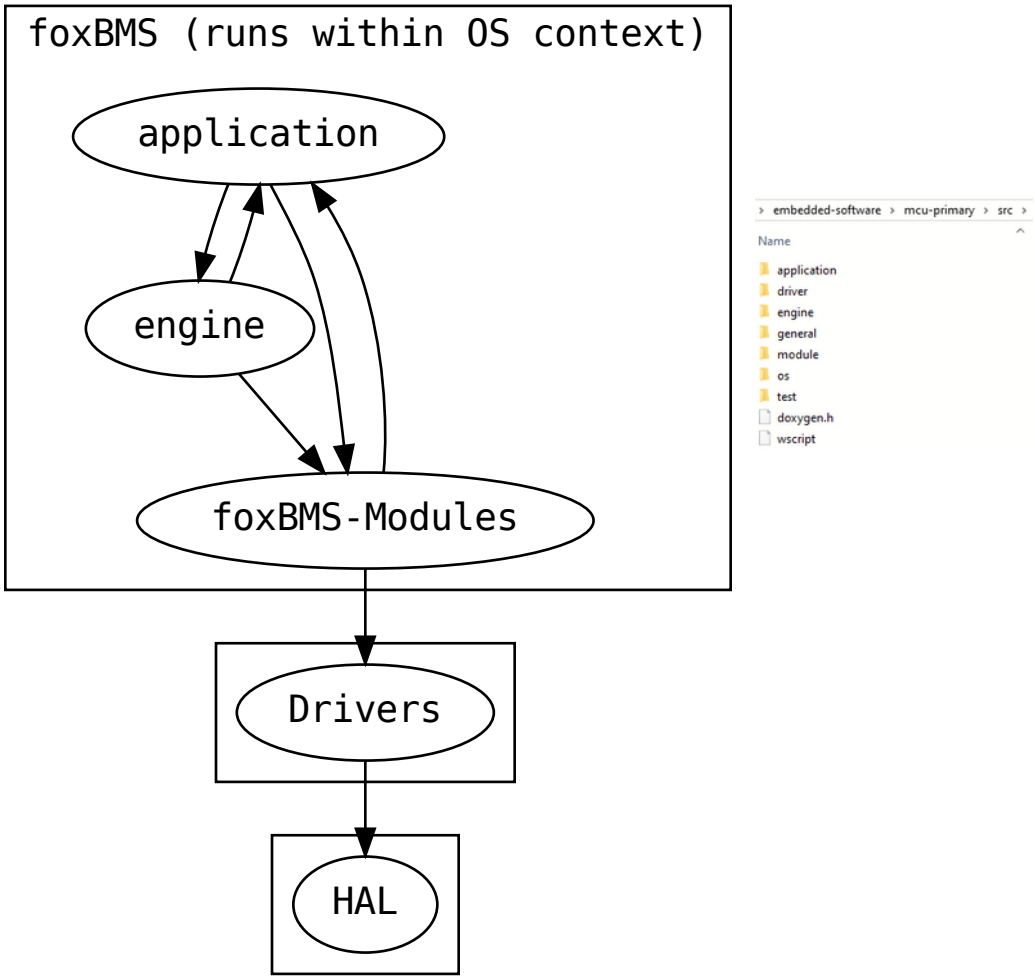


Fig. 25.1: foxBMS software architecture

25.4 Operating System

The real-time operating system FreeRTOS `V10.1.1` is used to manage the foxBMS-engine and applications. Tasks with different priorities are used to process the different algorithms and state machines and to ensure a real-time behaviour of the different modules.

25.5 foxBMS-Engine

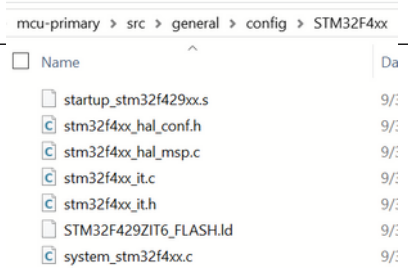
Important hardware independent software modules are bundled to the foxBMS-engine. This involves a database for storing important parameters, a diagnosis module for error detection and handling as well as the startup/initialization behaviour of the BMS. The operating system engine tasks are the tasks with the highest priorites and used to ensure a real-time behaviour of the safety-critical modules. Engine modules are allowed to include foxBMS-Modules, drivers, applications and of course the operating system.

25.6 foxBMS-Application

The application layer gathers all hardware independent modules that are responsible for the actual BMS functionality (i.e. SOC, SOH, balancing...). The most important module is the `bms` module that monitors the behaviour of the battery system to ensure an adherence of the safe-operating area and controls the BMS state (i.e. charging/discharging). The operating system applications tasks are the tasks with the lowest priorites to ensure a real-time behaviour of the safety-critical modules. Applications are allowed to include foxBMS-Modules, drivers, engine-modules and of course the operating system.

CHAPTER 26

Software Overview



26.1 Startup

The startup code begins at the label `Reset_Handler` in `general\config\startup_stm32f429xx.s`. After initialization of the main microcontroller registers, the system clock unit and the memory data (e.g., stack- and program pointer, system control block (SCB), interrupt vector, pre-initialization of data-sections, FPU settings), the C function `main()` is called. In ~~`general\main.c`~~ the second step of initializations of the microcontroller unit, peripherals and software modules are done (e.g., interrupt priorities, hardware modules like SPI and DMA). At this point, interrupts are still disabled. The steps are indicated by the global variable `os_boot`. At the end of the main function, the operating system resources (tasks, events, queues, mutex) are configured in `OS_TaskInit()` (`os/os.c`) and the scheduler is started by enabling the interrupts. Scheduling is started by invoking all configured tasks (FreeRTOS threads) regarding their priority. The activation of the scheduling is indicated by `os_boot = OS_RUNNING`. The OS-scheduler first calls the task `void ENG_TSK_Engine(void)` as the highest priority is assigned to this task. All other tasks are blocked in a while-loop until `ENG_TSK_Engine()` finishes the third initialization step and enters the periodic execution.

In general\main.c

26.2 Operating System

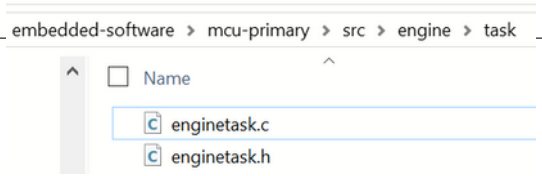
The following changes were to the FreeRTOS kernel in file port.c:

Cannot find this function.

- Function `uint8_t vPortCheckCriticalSection(void)` was added to check if a function is currently executed within a critical section of the OS.

A critical section is entered with the FreeRTOS function `vPortEnterCritical` and exited with the function `vPortExitCritical`. The operating system is configured in the file `FreeRTOSConfig.h`. A detailed explanation of the parameters is given at https://www.freertos.org/a00110.html.

26.3 Engine



Defined in "enginetask.c" –>

The task `void ENG_TSK_Engine(void)` executes the third (and last) step of system initialization with enabled interrupts in `ENG_PostOSInit()`. Then `OS_TSK_Engine()` manages the database and system monitoring via a periodic call of `DATA_Task()` and `DIAG_SysMon()` every 1ms.

After that, `os_boot` is set to `OS_SYTEM_RUNNING` and the function `void ENG_Init(void)` is run before the periodic tasks. Initializations can be made in this function. The database is already running when it is called. The system then runs the following tasks periodically:

- `void ENG_TSK_Cyclic_1ms(void)`
- `void ENG_TSK_Cyclic_10ms(void)`
- `void ENG_TSK_Cyclic_100ms(void)`

It must be noted that no changes should be made directly in these functions, since they are used as wrapper. They call the following functions:

- `void ENG_Cyclic_1ms(void)`
- `void ENG_Cyclic_10ms(void)`
- `void ENG_Cyclic_100ms(void)`

where the effective calls for system tasks are made. These three functions and `ENG_TSK_Engine()` make up the core of the system and are called `engine`. These function calls are found in `engine/task/enginetask.c`. Additionally, three users tasks run periodically. They are described in *User Applications*.

26.4 User Applications

For the user applications, three periodic tasks are available:

- `void APPL_TSK_Cyclic_1ms(void)`
- `void APPL_TSK_Cyclic_10ms(void)`
- `void APPL_TSK_Cyclic_100ms(void)`

It must be noted that no changes should be made directly in these functions, since they are used as wrapper. They call the following functions:

- `void APPL_Cyclic_1ms(void)`
- `void APPL_Cyclic_10ms(void)`
- `void APPL_Cyclic_100ms(void)`

Here, the user can implement its own functions, like new battery state estimation algorithms. The user has access to system data via the database. These function calls are found in `application/task/appltask_cfg.c`.

26.5 Startup Sequence

The start-up sequence is shown in fig. 26.1.

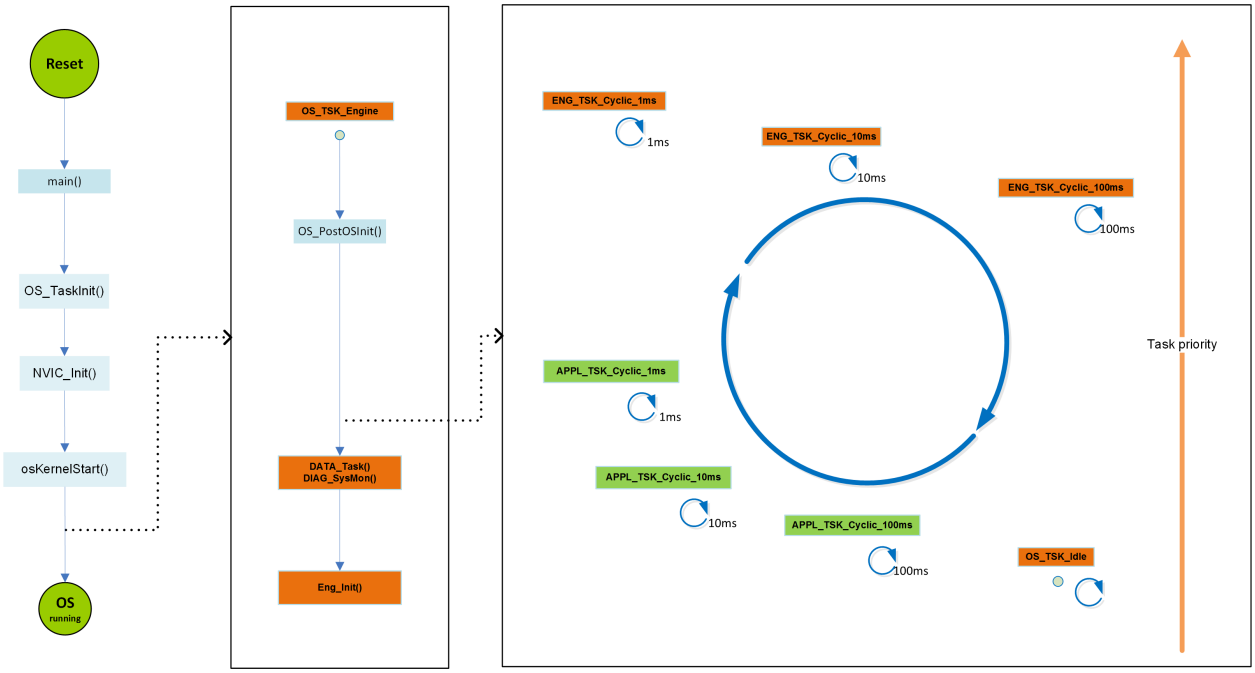


Fig. 26.1: foxBMS system start-up

26.6 Software Architecture

The database runs with the highest priority in the system and provides asynchronous data exchange for the whole system. Fig. 26.2 shows the data exchanges implemented via the database.

Fig. 26.3 shows the main structure of foxBMS.

The two key modules used are:

- `SYS`
- `BMS`

but

`SYS` has a lower priority than the database and a higher priority than `BMS`. Both modules are implemented as a state machine, with a trigger function that implements the transition between the states. The trigger functions of `SYS` is called in `void ENG_Cyclic_10ms(void)`. As `BMS` implements the user application, it is called in the user task `void APPL_Cyclic_1ms(void)`.

`SYS` controls the operating state of the system. It starts the other state machines (e.g., `CONT` for the contactors, `ILCK` for the interlock, `BMS`).

`BMS` gathers info on the system via the database and takes decisions based on this data. The BMS is driven via CAN. Requests are made via CAN to go either in STANDBY mode (i.e., contactors are open) or in NORMAL mode (i.e., contactors are closed). A safety feature is that these requests must be sent periodically every 100ms. `BMS` retrieves the state requests received via CAN from the database and analyses them. If the requests are not sent correctly, this means that the controlling unit driving the BMS has a problem and the correctness of the orders sent to the BMS may not be given anymore. As a consequence, in this case `BMS` makes a request to `CONT` to open the contactors. Currently, `BMS` checks the cell voltages, the cell temperatures and the global battery current. If one of these physical quantities show a value out of the safe operating area, `BMS` makes the corresponding state request to `CONT` to open the contactors. `BMS` is started via an initial state request made in `SYS`.

A watchdog instance is needed in case one of the aforementioned tasks hangs: in this case, the control over the contactors would not be provided anymore. This watchdog is made by the System Monitor module which monitors

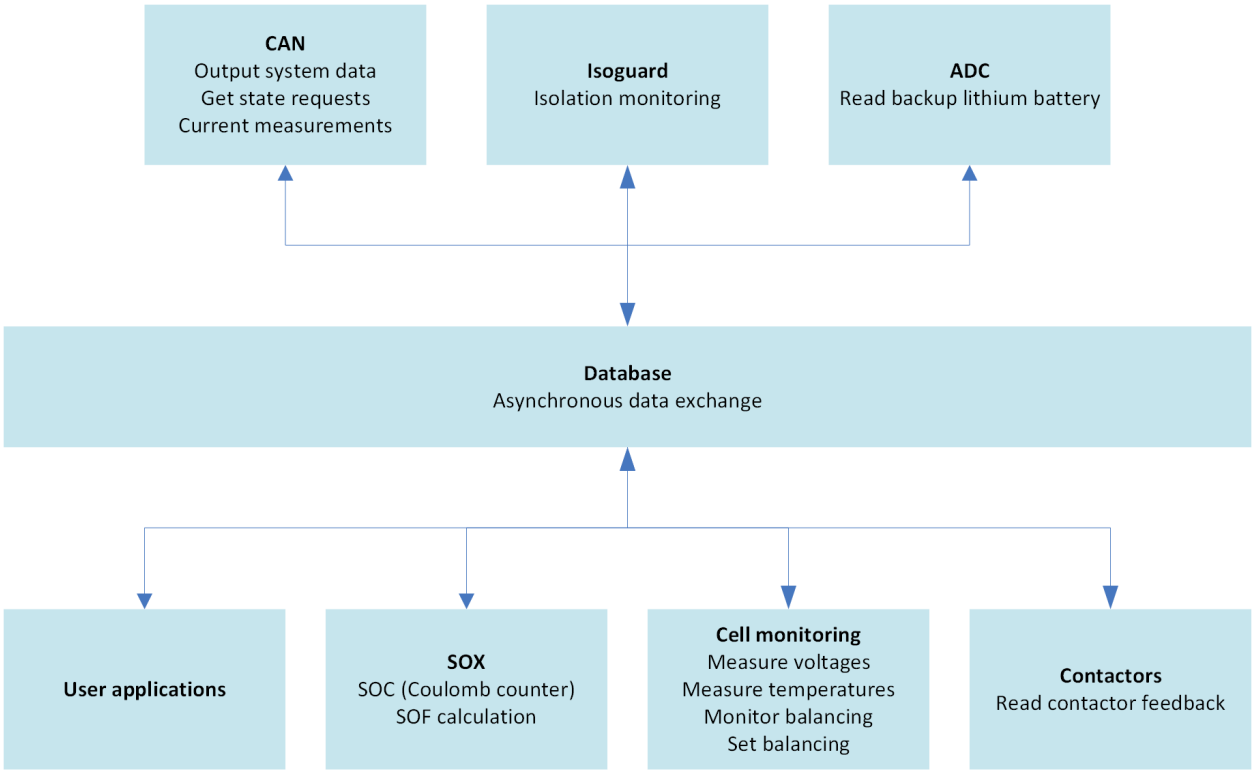


Fig. 26.2: Asynchronous data exchange with the foxBMS database

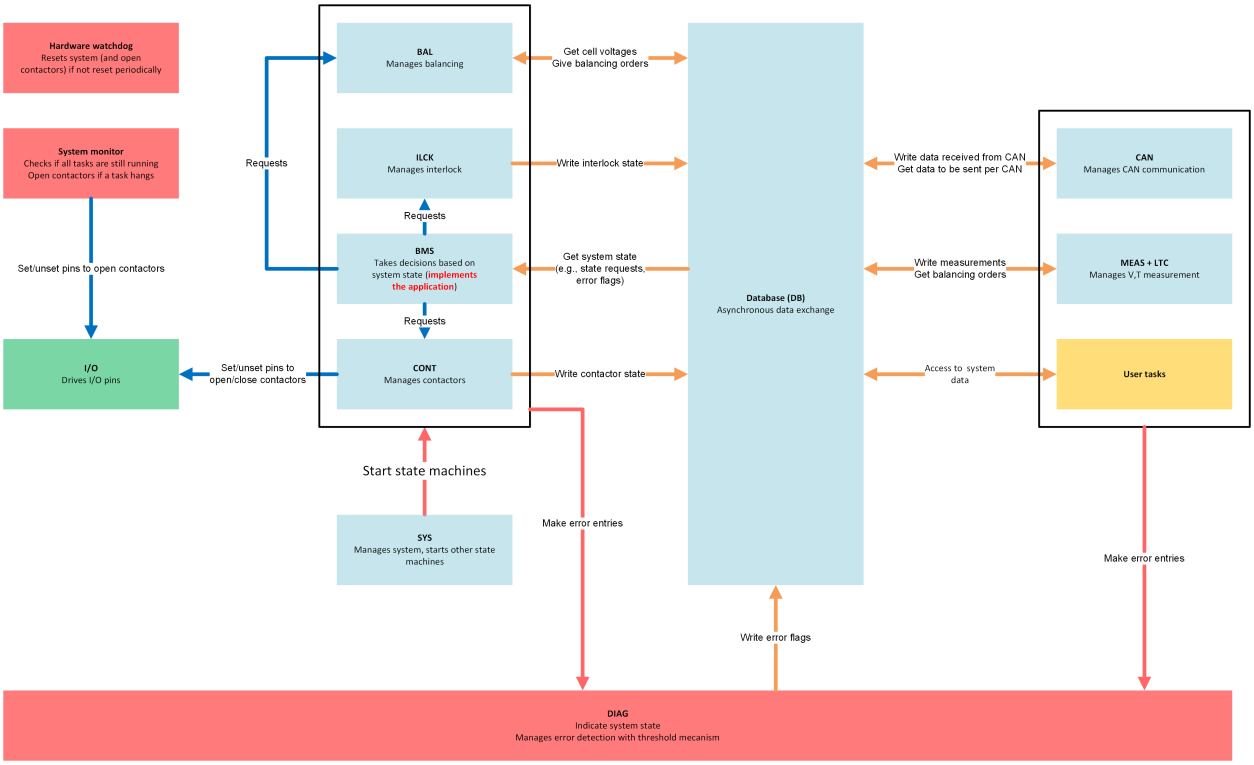


Fig. 26.3: Main tasks in foxBMS

all important tasks (e.g., `Database`, `SYS`, `BMS`): if any of the monitored tasks hangs, the contactors are opened to prevent damage and protect persons and the battery. To ensure the highest level of safety, opening the contactors is made by direct access to the `io` module.

A last barrier is present in case all the preceeding measures fail: the hardware watchdog timer. In case it is not triggered periodically, it resets the systems, provoking the opening of the contactors. Function calls (other than `SYS`) and closely related to the system are made in the engine tasks, for example:

- CAN
- Battery cells monitoring (voltages and temperatures)
- Galvanic isolation monitoring
- Lithium 3V coin cell voltage monitoring
- EEPROM
- NVRAM-Handler

26.7 Diagnostic

The `diag` module is designed to report problems on the whole system. The events that trigger the `diag` module have to be defined by the user. The event handler `DIAG_Handler(...)` has to be called when the event is detected. The way the system reacts to a `Diag` event is defined via a callback function or by the caller according to the return value.

Diagnostic events are stored in the Backup SRAM memory in the variable `DIAG_ERROR_ENTRY_s diag_memory[]`

```
typedef struct {
 uint8_t YY;
 uint8_t MM;
 uint8_t DD;
 uint8_t hh;
 uint8_t mm;
 uint8_t ss;
 DIAG_EVENT_e event;
 DIAG_CH_ID_e event_id;
 uint32_t item;
 uint32_t dummy1;
 uint32_t Val0;
 uint32_t Val1;
 uint32_t Val2;
 uint32_t Val3;
 } DIAG_ERROR_ENTRY_s;
```

26.8 Data stored in the database

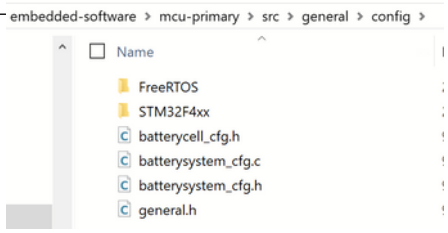
The following data is stored in the database:

- Cell voltages
- Cell temperatures
- SOX (Battery state, contains State-of-Charge)
- Balancing control

- Balancing feedback
- Current sensor measurements (includes battery current)
- Hardware information
- Last state request made to the BMS
- Minimum, maximum and average values for voltages and temperatures
- Measurement from isolation monitor
- Interface to communicate via I2C with extra funcitonalities on slave (e.g., EEPROM, port expander)
- Result from open-wire check on slaves
- LTC device parameter
- LTC accuracy
- Error state of the BMS (i.e., error flags, set to 0 or 1)
- MSL, maximum safety limits
- RSL, recommended safety limits
- MOL, maximum operating limits
- Calculated values for moving average
- Contactor feedback
- Interlock feedback
- BMS state (e.g., standby, normal, charge, error)
- Current limits calcuated for State-of-Function
- Voltages read on GPIOs of the slaves

CHAPTER 27

Important Switches in Code



27.1 System Relevant Switches

In `embedded-software\mcu-primary\src\general\config\batterysystem_cfg.h`, three important switches are defined.

The switch

```
#define CURRENT_SENSOR_PRESENT                  TRUE
```

is by default set to `TRUE`. In this configuration, the `SYS` statmachine will go into an error state during startup if no current sensor is detected. The switch must be set to `FALSE` for foxBMS to start without current sensor.

The switch

```
#define CHECK_CAN_TIMING                        TRUE
```

is set to `TRUE` by default. In this configuration, the `BMS` statemachine will go into an error state if no requests are made periodically per CAN with a period of 100ms. When the switch is set to `FALSE`, the check is not made.

The switch

```
#define BALANCING_DEFAULT_INACTIVE              TRUE
```

is set to `TRUE` by default. This prevents any balancing. In order for balancing to be possible, the switch must be set to `FALSE`.

27.2 Battery Cell Relevant Switches

In `embedded-software\mcu-primary\src\general\config\batterycell_cfg.h`, the Safe Operating Area for the cells is defined:

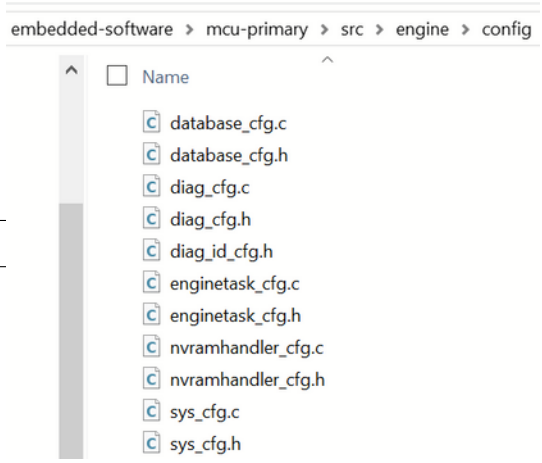
- Upper and lower voltage

- Upper and lower temperature for charging
- Upper and lower temperature for discharging
- Upper current for charging and discharging

The battery cell capacity is also defined with

```
#define BC_CAPACITY 3500
```

The unit is mAh.



27.3 Deactivating Cell Tests

In `embedded-software\mcu-primary\src\engine\config\diag_cfc.c`, the structure

```
DIAG_CH_CFG_s  diag_ch_cfg[]
```

allows disabling the checks made by the `diag` module. Disabling is made by replacing

```
DIAG_RECORDING_ENABLED, DIAG_ENABLED
```

with

```
DIAG_RECORDING_DISABLED, DIAG_DISABLED
```

Important checks are:

- `CELLVOLTAGE_OVERVOLTAGE`
- `CELLVOLTAGE_UNDERVOLTAGE`
- `OVERTEMPERATURE_CHARGE`
- `OVERTEMPERATURE_DISCHARGE`
- `UNDERTEMPERATURE_CHARGE`
- `UNDERTEMPERATURE_DISCHARGE`
- `OVERCURRENT_CHARGE`
- `OVERCURRENT_DISCHARGE`
- `LTC_PEC`

Note: Setting the cell limits is safety relevant and must be done with care.

Note: Working without configuring the right battery cell voltage limits is dangerous and should never be done when real batteries are connected, since they may burn and explode when overcharged or shorted.

27.4 Using Current Counter from Current Sensor

Current-counting can now be made using the current sensor. foxBMS will detect automatically if the corresponding data is being sent by the current sensor. If yes, the hardware current-counter from the sensor is used. If not, the software integrator is used.

27.5 Defining the Convention for the Current Direction

Two functions have been defined in `embedded-software\mcu-primary\src\general\config\batterycell_cfg.h` to test the current direction conveniently.

The first one

```
BS_CURRENT_DIRECTION_e BS_CheckCurrent_Direction(void);
```

gets the current from the database and returns `BS_CURRENT_DISCHARGE` if a discharge current is flowing through the battery, `BS_CURRENT_CHARGE` otherwise.

The second one

```
BS_CURRENT_DIRECTION_e BS_CheckCurrentValue_Direction(void);
```

this function
functions the same way. The only difference is that is does not get the current value from the database but uses the value passed to the function.

~~The define~~ `POSITIVE_DISCHARGE_CURRENT` is used to define the current direction convention:

- if set to `TRUE`, discharge currents are positive
- if set to `FALSE`, charge currents are positive

ition s
This define affect the calculation in the `SOX` module too.

27.6 Complete List of Switches

A complete list of switches can be found here:

- The online version of the latest released version is found here:

  Warning: This documentation always points to the latest realsed version on Github and the local changes can not be found in that online version.

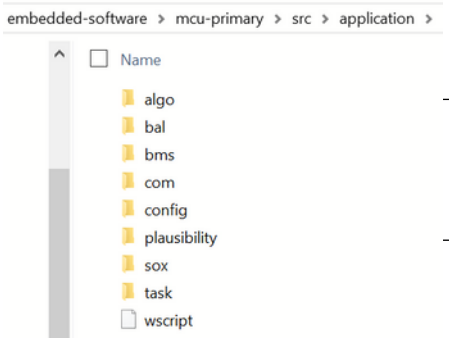
  - Online documentation: Primary switches
  - Online documentation: Secondary switches

- If the project was compiled locally, this documentation can be accesed here:

  Warning: This only exists after building the documentation locally. If this was not done, the link will not work.

  - Local documentation: Primary switches
  - Local documentation: Secondary switches

CHAPTER 28



Software Modules

This section contains the documentation of the most important software modules.

The is section is organized the same way the sources are organized (`Driver`, `foxBMS-modules`, `Engine`, `Utils` and `Application`).

Application:

28.1 Algorithm

The `algorithm` module module is part of the `Application` layer.

The `algorithm` module provides an environment for algorithms running on the BMS-Master Board. It handles the execution of algorithms and monitors execution times of the individual algorithms. Currently the current and power moving averages for 1s, 5s, 10s, 30s and 60 seconds is calculated.

28.1.1 Module Files

Driver:

- `embedded-software\mcu-primary\src\application\algo\algo.c` (algoc)
- `embedded-software\mcu-primary\src\application\algo\algo.h` (algoh)

Driver Configuration:

- `embedded-software\mcu-primary\src\application\config\algo_cfg.c` (algocfgc)
- `embedded-software\mcu-primary\src\application\config\algo_cfg.h` (algocfgc)

Data

28.1.2 Structure

of the state/variables and function pointer

The struct `ALGO_TASKS_s` contains the definition of an algorithm.

Defined in \mcu-primary\src\application\config\algo_cfg.h

```
typedef struct ALGO_TASKS {
    ALGO_STATE_e state;                /* !< current execution state */
    uint32_t cycleTime_ms;             /*!< cycle time of algorithm */
    uint32_t maxCalcDuration_ms;       /* !< maximum allowed calculation duration for␣
↪task */
    uint32_t startTime;                /* !< start time when executing algorithm */
    void (*func)(uint32_t algoIdx);    /*!< callback function */
} ALGO_TASKS_s;
```

`state` contains the current state of the algorithm:

```
typedef enum ALGO_STATE {
    ALGO_READY            = 0,
    ALGO_RUNNING          = 1,
    ALGO_WAIT_FOR_OTHER   = 2,
    ALGO_RDY_BUT_WAITING  = 3,
    ALGO_EXECUTE_ASAP     = 4,
    ALGO_BLOCKED          = 5,
} ALGO_STATE_e;
```

`cycleTime_ms` defines the periodic execution time of the algorithm. `maxCalcDuration_ms` specifies the maximum execution time of the algorithm. If an algorithm takes longer to execute than specified, it is set to the `ALGO_BLOCKED` state and will not be executed again. An error will be set in the `diag` module. `startTime` is set when the execution of an algorithm starts. `void (*func)(uint32_t algoIdx)` is the function pointer to the implementation of the algorithm.

Defined in \mcu-primary\src\application\algo\algo.c

`ALGO_MainFunction` loops over the different algorithms and executes them one after another if the cycleTime_ms is elapsed. Before calling the algorithm, the algorithm state is set to `ALGO_RUNNING` and the start time for the execution time monitoring is set:

```
/* Set state to running -> reset to READY before leaving algo function */
algo_algorithms[i].state = ALGO_RUNNING;
algo_algorithms[i].startTime = OS_getOSSysTick();
algo_algorithms[i].func(i);
```

Note: After finishing the execution of the algorithm and before leaving the function, the algorithm state needs to be set back to `ALGO_READY`. Otherwise it will not be executed again.

```
// Only set task to ready state if it isn't blocked by the monitoring unit because of␣
↪a runtime violation
if (algo_algorithms[algoIdx].state != ALGO_BLOCKED) {
    algo_algorithms[algoIdx].state = ALGO_READY;
}
```

28.2 Balancing

The `balancing` module is part of the `Application` layer.

28.2.1 Module Files

Driver:

- embedded-software\mcu-primary\src\application\bal\bal.c (balc)
- embedded-software\mcu-primary\src\application\bal\bal.h (balh)

Driver Configuration:

- embedded-software\mcu-primary\src\application\config\bal_cfg.c (balcfgc)
- embedded-software\mcu-primary\src\application\config\bal_cfg.h (balcfgh)

Functionality

28.2.2 Structure

The balancing module takes care of the voltage or charge equalization of the battery cells. Balancing is deactivated by default by the switch BALANCING_DEFAULT_INACTIVE in embedded-software\mcu-primary\src\general\config\batterysystem_cfg.h, which is set to TRUE, to prevent automatic start of the balancing when foxBMS is used in a laboratory environment for example. The switch must be manually set to FALSE to allow the automatic balancing and charge equalization process.

The default balancing method is voltage-based balancing. This is set by the switch BALANCING_VOLTAGE_BASED in embedded-software\mcu-primary\src\application\config\bal_cfg.h, which is then set to TRUE. A more advanced balancing method implemented in foxBMS is based on the SOC history balancing. As this method needs a look-up table of the used battery cell behavior, it is not the default method. It can be used by setting the switch BALANCING_VOLTAGE_BASED to TRUE. FALSE?

When the current flowing through the battery is below the limit defined by BS_REST_CURRENT_mA in embedded-software\mcu-primary\src\general\config\batterysystem_cfg.h, the balancing module waits BAL_TIME_BEFORE_BALANCING_S seconds before starting to perform balancing. The waiting time is re-initialized every time the current exceeds BS_REST_CURRENT_mA. No balancing takes place if the voltage of the cells in the battery pack goes below BAL_LOWER_VOLTAGE_LIMIT_MV or the maximum temperature of the cells in the pack goes above BAL_UPPER_TEMPERATURE_LIMIT_DEG.

28.2.3 Voltage-based balancing

In voltage-based balancing, the balancing module takes the minimum battery cell voltage of the complete battery pack and activates balancing for all the cells whose voltage is above the minimum + BAL_THRESHOLD_MV. Once all cells have been balanced, the threshold is set to BAL_THRESHOLD_MV + BAL_HYSTERESIS_MV to avoid an oscillating behavior between balancing and not balancing.

In the bms module, when entering the STANDBY state, voltage-based balancing is allowed. When entering PRECHARGE or ERROR, voltage-based balancing is not allowed.

Fig. 28.1 shows the state machine managing voltage-based balancing in foxBMS.

28.2.4 SOC history-based balancing

The SOC history-based balancing works as follows: at one point in time, when no current is flowing and the cell voltages have fully relaxed (e.g., after 3 hours rest time), the voltages of all cells are measured. With a suitable SOC versus voltage look-up table, the voltages are converted to their respective SOCs. The SOCs are then translated to Depth-of-Discharge (DOD) using the nominal capacity, with:

```
DOD = Capacity * (1-SOC)
```

The cell with the highest DOD is taken as a reference, since it is the most discharged cell in the battery pack. Its charge difference is set to 0. For all other cells, the charge difference is computed via:

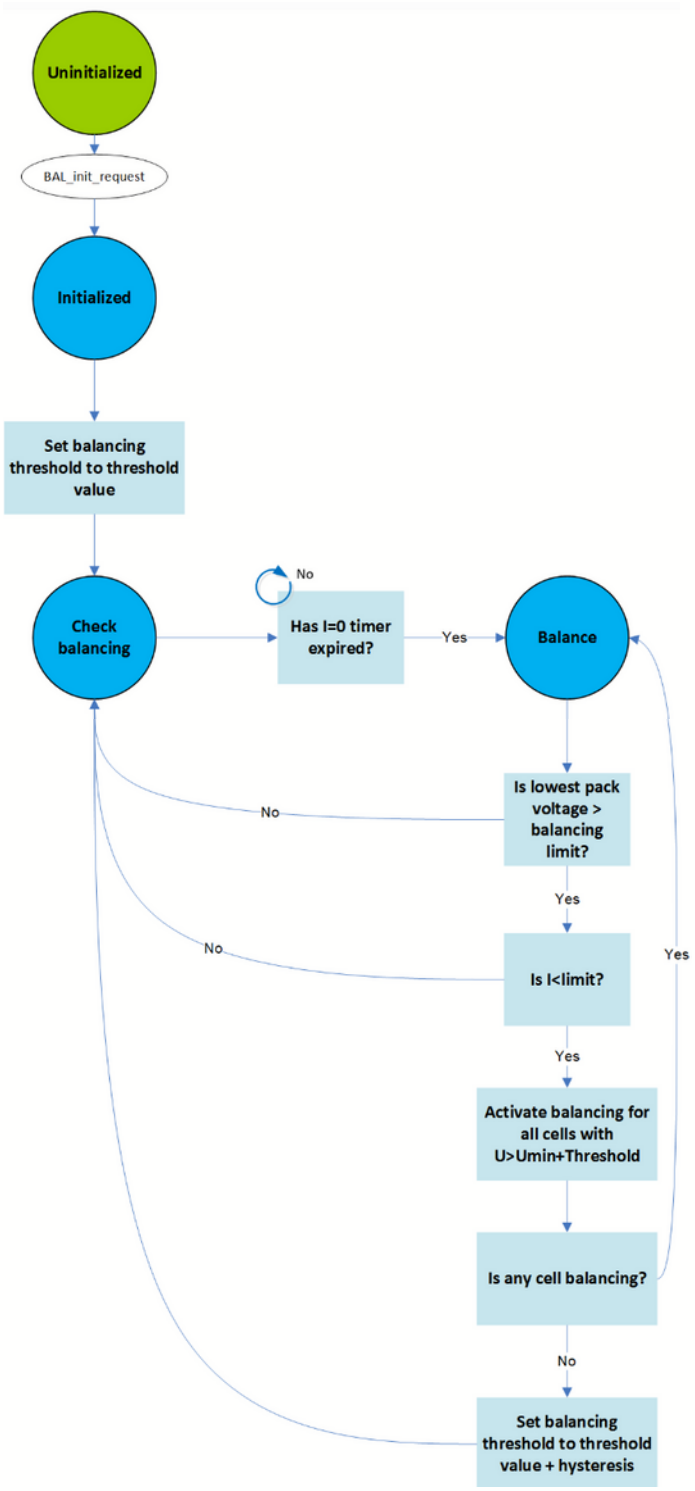

Uninitialized
BAL_init_request
Initialized
Set balancing threshold to threshold value
Check balancing
No
Has I=0 timer expired?
Yes
Balance
Is lowest pack voltage > balancing limit?
No
Yes
Is I<limit?
No
Yes
Activate balancing for all cells with U>Umin+Threshold
Is any cell balancing?
Yes
No
Set balancing threshold to threshold value + hysteresis

Fig. 28.1

```
Charge difference(considered cell) = DOD(reference cell) - DOD(considered
cell)
```

Balancing is then switched on for all cells. Every second, for each cell, the voltage is taken and the balancing current computed with:

```
current = cell voltage / balancing resistance
```

The balancing quantity:

```
current * 1s
```

is subtracted from the charge difference. Balancing is stays turned on until the charge difference reaches 0.

In SOC history-based balancing, `BS_BALANCING_RESISTANCE_OHM` must be defined identically to the balancing resistances soldered on the BMS-Slave Board. When the imbalances are computed, they are set to a non-zero value to balance each specific cell only if its cell voltage is above the minimum cell voltage of the battery pack plus a threshold. The threshold is set in this case to `BAL_THRESHOLD_MV` + `BAL_HYSTERESIS_MV`. It is not simply set to `BAL_THRESHOLD_MV` for compatibility reasons with the code shared with the voltage-based balancing.

The correspondence between cell voltage and SOC must be defined by the user depending on the specific battery cells used. Currently, it is done in the function `SOC_GetFromVoltage()` in `sox.c`. This function gets a voltage in V and return an SOC between 0 and 1.

Note: The SOC to voltage correspondence is specific to the cell used. The user must define the look-up table, or the SOC history-based balancing will not perform as expected.

Fig. 28.2 shows the state machine managing the SOC history-based balancing in foxBMS.

28.3 BMS

The `bms` module is part of the `Application` layer.

The `bms` module contains the application running on the BMS-Master Board. It handles requests of superior control units via CAN messages and checks the system state via error flags read from the database.

28.3.1 Module Files

It may be a better idea to combine the primary and secondary code and use the flag to distinguish the primary and secondary targets.

Driver:

- `embedded-software\mcu-primary\src\application\bms\bms.c` (bmsprimaryc)
- `embedded-software\mcu-primary\src\application\bms\bms.h` (bmsprimaryh)
- `embedded-software\mcu-secondary\src\application\bms\bms.c` (bmssecondaryc)
- `embedded-software\mcu-secondary\src\application\bms\bms.h` (bmssecondaryh)

Driver Configuration:

- `embedded-software\mcu-primary\src\application\config\bms_cfg.c` (bmscfgprimaryc)
- `embedded-software\mcu-primary\src\application\config\bms_cfg.h` (bmscfgprimaryh)
- `embedded-software\mcu-secondary\src\application\config\bms_cfg.c` (bmscfgsecondaryc)

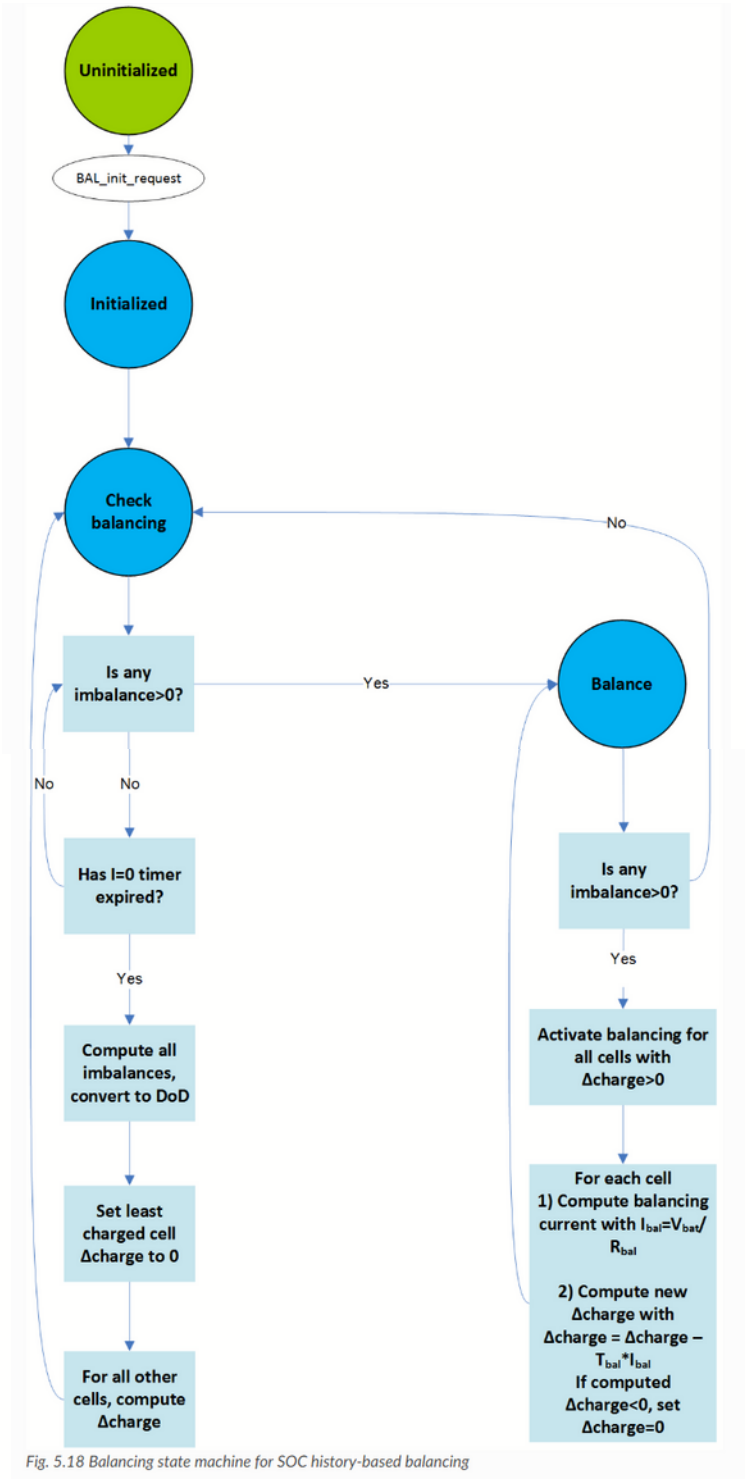


Fig. 5.18 Balancing state machine for SOC history-based balancing

Fig. 28.2

- embedded-software\mcu-secondary\src\application\config\bms_cfg.h (bmscfgsecondaryh)

State machine

28.3.2 Structure

Fig. 28.3 shows the statemachine corresponding to the default application implemented in foxBMS.

In the default application, the requests made by CAN and the error flags are read from the database. If no error is detected, the bms module makes request to the contactor module and interlock module to drive the contactors. In case one or more errors are detected, a request is made to the contactor module to open the contactors.

Three states are implemented:

- STANBDY STANDBY
- NORMAL
- CHARGE
* ENGINE

STANDBY corresponds to the state where all the contactors are open. NORMAL and CHARGE correspond to a state where the contactors ~~of one~~ of the powerlines are closed to allow current flowing.

BS = battery system

The CHARGE state is available only if the switch BS_SEPARATE_POWERLINES in embedded-software\mcu-primary\src\general\config\batterysystem_cfg.h is set to 1. It corresponds to the use of a separate powerline compared to the powerline used in the normal state.

(for charging)

The transition between the states is made in response to CAN request read from the database. From STANDBY, the state machine can transition to NORMAL or CHARGE, or the opposite. No transition is possible directly between NORMAL and CHARGE.

Any more complex application (e.g., that does not open the contactors immediately in case of aerospace applications) can be implemented here.

28.3.3 Module Files for the primary MCU

Driver:

- embedded-software\mcu-primary\src\application\bms\bms.c
- embedded-software\mcu-primary\src\application\bms\bms.h

Driver Configuration:

- embedded-software\mcu-primary\src\application\config\bms_cfg.c
- embedded-software\mcu-primary\src\application\config\bms_cfg.h

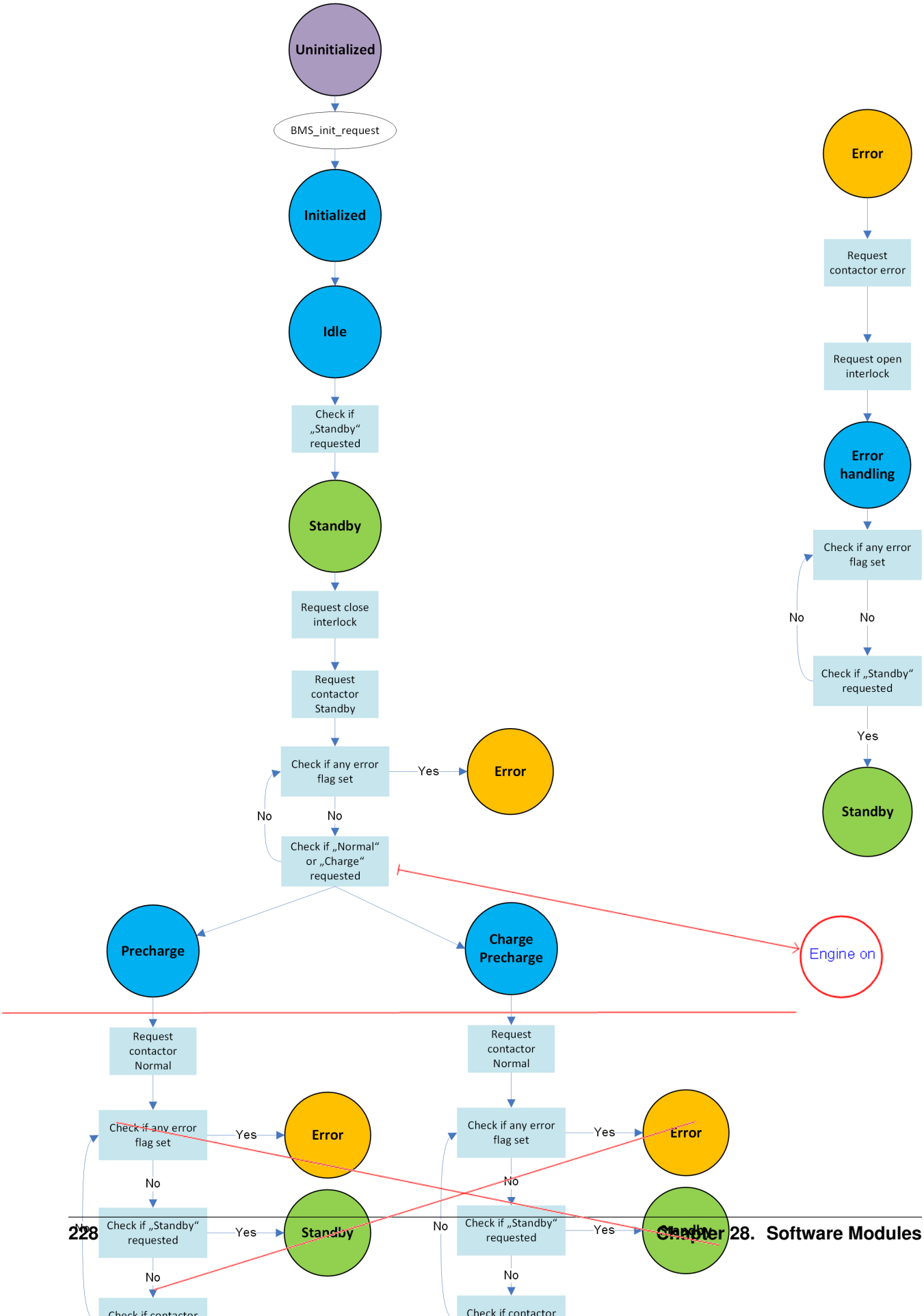
28.3.4 Configuration of BMS

The Safe Operating Area (SOA) comprising the cell voltage and temperature limits are not defined in the bms module. They can be found in the file embedded-software\mcu-primary\src\general\config\batterycell_cfg.h.

The following switches are defined:

NAME	LEVEL	DESCRIPTION	default value
BMS_CAN_TIMING_TEST	user	CAN timing test enable	TRUE
BMS_TEST_CELL_SOF_LIMITS	user	SOF limits test enable	FALSE

F = function


Uninitialized
BMS_init_request
Initialized
Idle
Check if „Standby“ requested
Standby
Request close interlock
Request contactor Standby
Check if any error flag set
Yes
Error
No
No
Check if „Normal“ or „Charge“ requested
Precharge
Charge Precharge
Engine on
Request contactor Normal
Check if any error flag set
Yes
Error
No
Check if „Standby“ requested
Yes
Standby
No
Check if contactor
Request contactor Normal
Check if any error flag set
Yes
Error
No
Check if „Standby“ requested
Yes
Standby
No
Check if contactor
Error
Request contactor error
Request open interlock
Error handling
Check if any error flag set
No
No
Check if „Standby“ requested
Yes
Standby

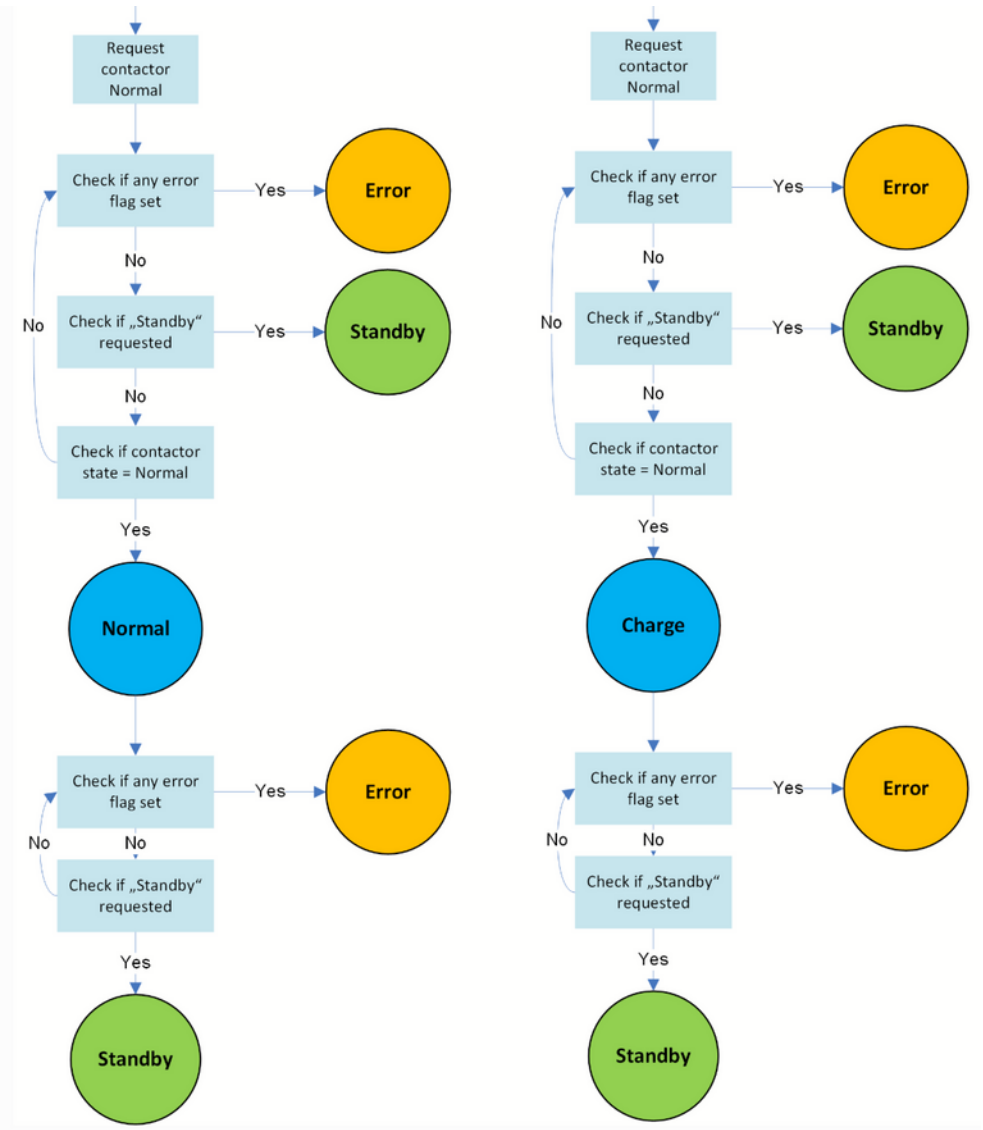


Fig. 5.19 BMS state machine

Fig. 28.3

The IDs of the requests receivable via CAN signal are configured with:

NAME	LEVEL	DESCRIPTION	default value
BMS_REQ_ID_NORMAL	user	ID to request for NORMAL state	3
BMS_REQ_ID_CHARGE	user	ID to request for CHARGE state	4
BMS_REQ_ID_STANDBY	user	ID to request for STANDBY state	8
BMS_REQ_ID_ENGINE	user		Ask Aaron

28.4 COM

The `com` module is part of the `Application` layer.

The `com` module handles communication between the MCU and external devices. It makes use of different interfaces (e.g., `uart` module) for this purpose.

28.4.1 Module Files

Driver:

- `embedded-software\mcu-primary\src\application\com\com.h` (comc)
- `embedded-software\mcu-primary\src\application\com\com.c` (comh)

28.4.2 Establishing Connection with PC

1. Connect PC with USB cable to the primary USB interface
2. Start a terminal program (e.g., HTerm)
3. Select correct COM Port
4. Select following settings:

- Baudrate: 115200 Bd
- Databits: 8, Stopbits: 1, Parity: None
- DTR: disabled, RTS: disabled
- Send on enter: CR

28.4.3 Functionality

The `com` module provides possibilities for communication with external devices using UART. For incoming requests, it provides some kind of basic command parser/handler and access to the `sys` module. Furthermore, it provides the user with some information about the system and a testmode where the system settings can be alternated and basic system tests can be performed.

The `com` module can be enabled by using the BUILD_MODULE_ENABLE_COM define. Its handler needs to be called periodically (e.g., by using `ENG_TSK_Cyclic_10ms()`). Currently the COM_Decoder supports following commands:

Command	Description
help	get available command list
gettime	get system time
getruntime	get runtime since last reset
printdiaginfo	get diagnosis entries of DIAG module (entries can only be printed once)
printcontactor-info	get contactor information (number of switches/hard switches) (entries can only be printed once)
teston	enable testmode, testmode will be disabled after a predefined timeout of 30s when no new command is sent

Following commands are only available during enabled testmode:

Command	Description
testoff	disable testmode
settime YY MM DD HH MM SS	set mcu time and date (YY-year, MM-month, DD-date, HH-hours, MM-minutes, SS-seconds)
reset	enforces complete software reset using HAL_NVIC_SystemReset()
watchdogtest	performs watchdog test, watchdog timeout results in system reset (predefined 1s)
setsoc xxx.xxx	set SOC value (000.000% - 100.000%)
ceX	enables contactor number X (only possible if BMS is in no error state)
cdX	disables contactor number X (only possible if BMS is in no error state)

28.5 SOX

This section describes where and how to implement state estimation algorithms (e.g., SOC, SOH, SOF). The basic SOC calculation is done by a simple Coulomb counter. Its implementation is shown here. Current derating depending on cell voltages, temperatures and SOC is performed to compute the SOF.

28.5.1 Module Files

Driver:

- `embedded-software\mcu-primary\src\application\sox\sox.h` (soxc)
- `embedded-software\mcu-primary\src\application\sox\sox.c` (soxh)

Driver Configuration:

- `embedded-software\mcu-primary\src\application\config\sox_cfg.h` (soxcfgc)
- `embedded-software\mcu-primary\src\application\config\sox_cfg.c` (soxcfgh)

28.5.2 Detailed Description

The `sox` module gets the relevant measurement minimum and maximum values from the database and stores the current derating values in the database.

SOC - State of Charge

The state of charge estimation (SOC) is implemented in the form of a simple Coulomb counter. The SOC initialization is done after startup by reading the value from the non-volatile memory. Not implemented right now is a

initialization of the SOC by VOLTAGE-SOC relation (lookup table), but configuration placeholders are already in *SOX Configuration*. These placeholders define the constraints at which the initialization with lookup table is valid.

SOF - State of Function

The state of function estimation (SOF) consists of current derating values. These charge and discharge derating values are calculated according to battery cell specific constraints. For this, three parameters are taken into account:

- temperature
- voltage
- state-of-charge

Four different curves are calculated:

- recommended operating limit (ROL) (this is the recommended operating current transmitted on the CAN bus)
- maximum operating limit (MOL) (if the floating current exceeds this threshold, the overcurrent warning flag is set)
- recommended safety limit (RSL) (if the floating current exceeds this threshold, the overcurrent alarm flag is set)
- (absolut) maximum safety limit (MSL) (if the floating current exceeds this threshold, the overcurrent error flag is set and the contactors are opened)

Specific points in the following derating curves have to be defined by configurable defines:

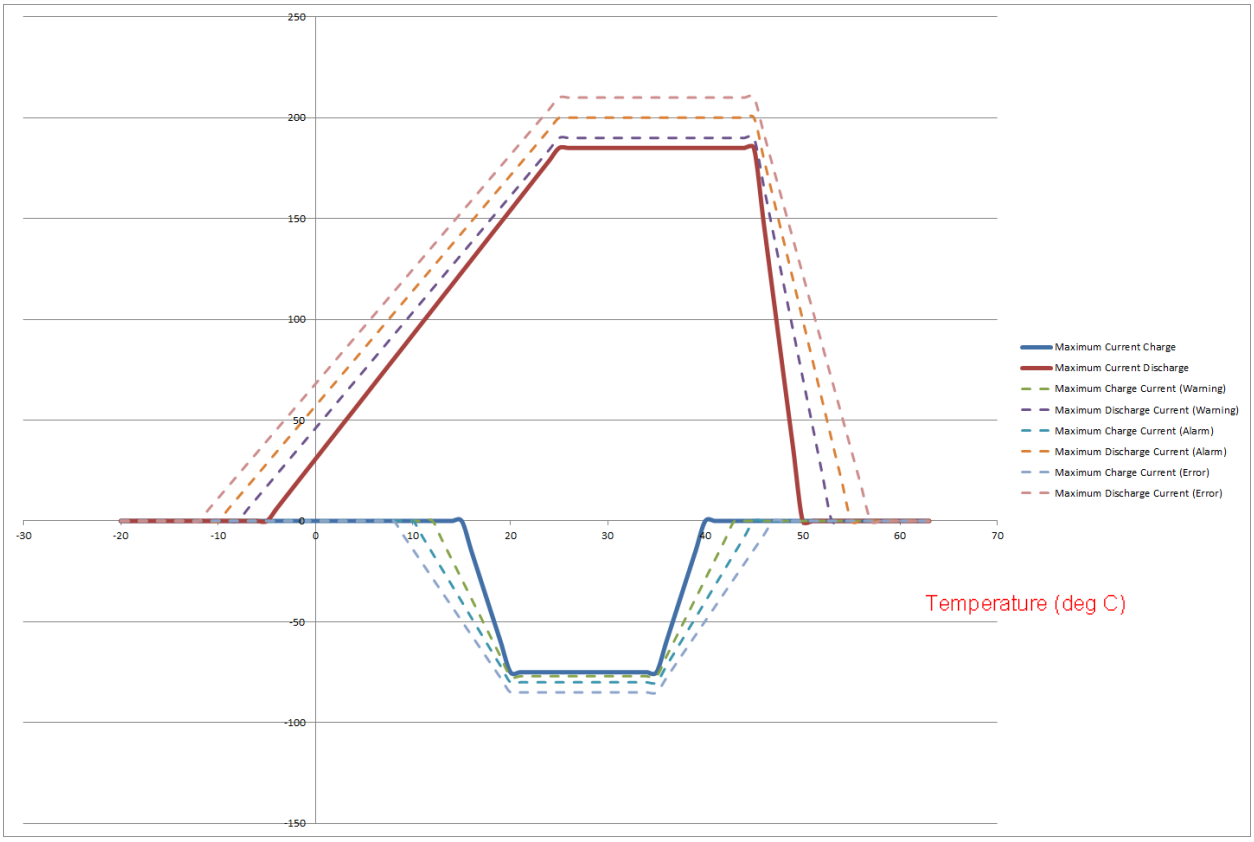


Fig. 28.4: Temperature dependent current derating with three alarm levels (warning, alarm, error)

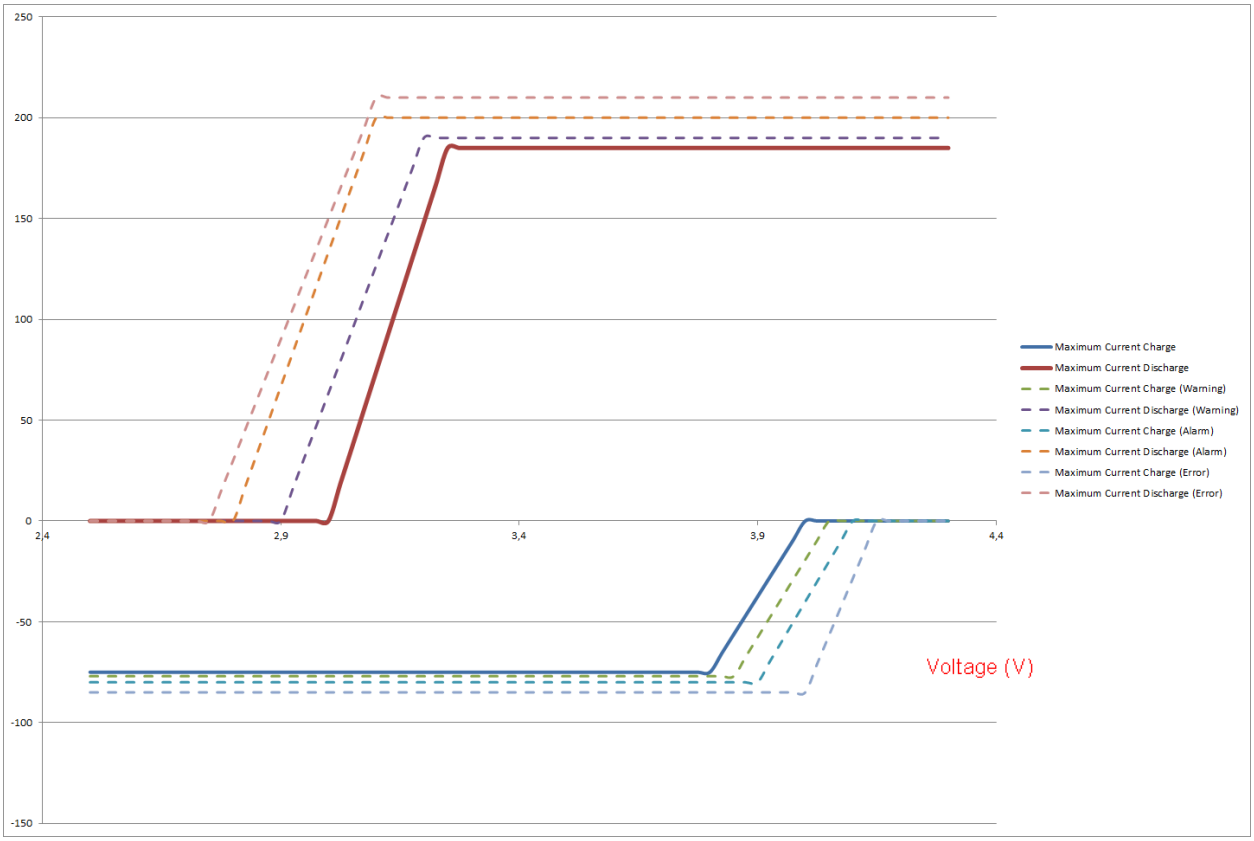


Fig. 28.5: Voltage dependent current derating with three alarm levels (warning, alarm, error)

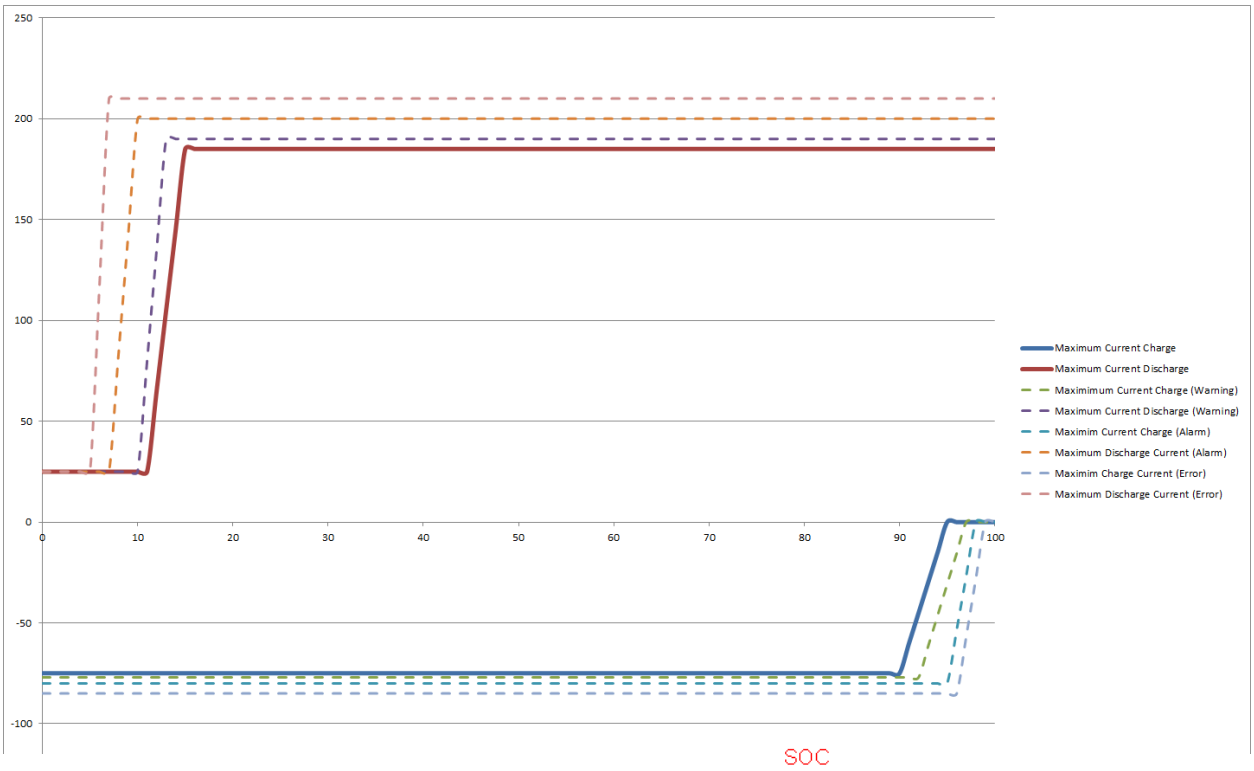


Fig. 28.6: SOC dependent current derating with three alarm levels (warning, alarm, error)

These specific points are the values where derating starts and where it is at full extent. They are described in *Configuration Variables* and battery specific values for the maximum allowed current can be seen for example in *Configuration Example for Lithium-Ion NMC/LTO Chemistry* and *Configuration Example for Lithium-Ion NCA/Graphite or NMC/Graphite Chemistries*.

28.5.3 SOX Configuration

Configuration Variables

For the Coulomb counting method the cell capacity (in case of parallel cell configuration, it is the sum of the parallel connected cells) has to be given here:

NAME	LEVEL	TYPE	UNIT	DESCRIPTION	DEFAULT
SOX_CELL_CAPACITY	devel	float	mAh	cell capacity in SOC formula coulomb counter	20000.0

Currently there is only placeholder for the initialization by a Voltage-SOC relation. The following configuration can be used after implementation:

NAME	TYPE	UNIT	DESCRIPTION	DEFAULT
SOX_SOC_INIT_CURRENT_LIMIT	int	mA	at initialization the current must be below	100
SOX_DELTA_MIN_LIMIT	int	mV	see source code	10
SOX_DELTA_MAX_LIMIT	int	mV	see source code	10

These are the configuration variables of the ROL, MOL, RSL and MSL:

NAME	LEVEL	TYPE	UNIT	DESCRIPTION	VALIDATOR
SOX_CURRENT_MAX_CONTINUOUS_CHARGE	devel	float	A	maximum current continuous charge	1<x<240
SOX_CURRENT_MAX_CONTINUOUS_DISCHARGE	devel	float	A	maximum current continuous discharge	1<x<240
SOX_CURRENT_LIMP_HOME	devel	float	A	discharge current in limp home emergency mode	1<x<40
SOX_TEMP_LOW_CUTOFF_DISCHARGE	devel	float	°C	low temperature discharge derating start	-40.0<x<80.0
SOX_TEMP_LOW_LIMIT_DISCHARGE	devel	float	°C	low temperature discharge derating full	-40.0<x<80.0
SOX_TEMP_LOW_CUTOFF_CHARGE	devel	float	°C	low temperature charge derating start	-40.0<x<80.0
SOX_TEMP_LOW_LIMIT_CHARGE	devel	float	°C	low temperature charge derating full	-40.0<x<80.0
SOX_TEMP_HIGH_CUTOFF_DISCHARGE	devel	float	°C	low temperature discharge derating start	-40.0<x<80.0
SOX_TEMP_HIGH_LIMIT_DISCHARGE	devel	float	°C	low temperature discharge derating full	-40.0<x<80.0
SOX_TEMP_HIGH_CUTOFF_CHARGE	devel	float	°C	low temperature charge derating start	-40.0<x<80.0
SOX_TEMP_HIGH_LIMIT_CHARGE	devel	float	°C	low temperature charge derating full	-40.0<x<80.0
SOX_SOC_CUTOFF_CHARGE	devel	int	0.01%	high SOC derating starts	0<=x<=10000
SOX_SOC_LIMIT_CHARGE	devel	int	0.01%	high SOC derating full extent	0<=x<=10000
SOX_SOC_CUTOFF_DISCHARGE	devel	int	0.01%	low SOC derating starts	0<=x<=10000
SOX_SOC_LIMIT_DISCHARGE	devel	int	0.01%	low SOC derating full extent	0<=x<=10000
SOX_VOLT_CUTOFF_CHARGE	devel	int	mV	high voltage derating starts	0<=x<=5000
SOX_VOLT_LIMIT_CHARGE	devel	int	mV	high voltage derating full extent	0<=x<=5000
SOX_VOLT_CUTOFF_DISCHARGE	devel	int	mV	low voltage derating starts	0<=x<=5000
SOX_VOLT_LIMIT_DISCHARGE	devel	int	mV	low voltage derating full extent	0<=x<=5000

These configuration values are building the main safety feature and are therefore considered highly safety-relevant.

Configuration Example for Lithium-Ion LFP/Graphite Chemistry

This configuration is very conservative and the limits are defensive. It is the default standard configuration. These values must be adapted to the specific battery cells used.

NAME	UNIT	VALUE	DESCRIPTION
SOX_CURRENT_MAX_CONTINUOUS_CHARGE	A	10.00	maximum current continuous charge
SOX_CURRENT_MAX_CONTINUOUS_DISCHARGE	A	10.00	maximum current continuous discharge
SOX_CURRENT_LIMP_HOME	A	3.00	discharge current in limp home emergency mode
SOX_TEMP_LOW_CUTOFF_DISCHARGE	°C	5.0	low temperature discharge derating start
SOX_TEMP_LOW_LIMIT_DISCHARGE	°C	-5.0	low temperature discharge derating full
SOX_TEMP_LOW_CUTOFF_CHARGE	°C	10.0	low temperature charge derating start
SOX_TEMP_LOW_LIMIT_CHARGE	°C	0.0	low temperature charge derating full
SOX_TEMP_HIGH_CUTOFF_DISCHARGE	°C	45.0	low temperature discharge derating start
SOX_TEMP_HIGH_LIMIT_DISCHARGE	°C	55.0	low temperature discharge derating full
SOX_TEMP_HIGH_CUTOFF_CHARGE	°C	30.0	low temperature charge derating start
SOX_TEMP_HIGH_LIMIT_CHARGE	°C	37.0	low temperature charge derating full
SOX_SOC_CUTOFF_CHARGE	0.01%	8500	high SOC derating starts
SOX_SOC_LIMIT_CHARGE	0.01%	9500	high SOC derating full extent
SOX_SOC_CUTOFF_DISCHARGE	0.01%	1500	low SOC derating starts
SOX_SOC_LIMIT_DISCHARGE	0.01%	500	low SOC derating full extent
SOX_VOLT_CUTOFF_CHARGE	mV	3300	high voltage derating starts
SOX_VOLT_LIMIT_CHARGE	mV	3550	high voltage derating full extent
SOX_VOLT_CUTOFF_DISCHARGE	mV	2700	low voltage derating starts
SOX_VOLT_LIMIT_DISCHARGE	mV	2300	low voltage derating full extent

Configuration Example for Lithium-Ion NMC/LTO Chemistry

NAME	UNIT	VALUE	DESCRIPTION
SOX_CURRENT_MAX_CONTINUOUS_CHARGE	A	120.00	maximum current continuous charge
SOX_CURRENT_MAX_CONTINUOUS_DISCHARGE	A	120.00	maximum current continuous discharge
SOX_CURRENT_LIMP_HOME	A	20.00	discharge current in limp home emergency mode
SOX_TEMP_LOW_CUTOFF_DISCHARGE	°C	0.0	low temperature discharge derating start
SOX_TEMP_LOW_LIMIT_DISCHARGE	°C	-10.0	low temperature discharge derating full
SOX_TEMP_LOW_CUTOFF_CHARGE	°C	0.0	low temperature charge derating start
SOX_TEMP_LOW_LIMIT_CHARGE	°C	-10.0	low temperature charge derating full
SOX_TEMP_HIGH_CUTOFF_DISCHARGE	°C	45.0	low temperature discharge derating start
SOX_TEMP_HIGH_LIMIT_DISCHARGE	°C	55.0	low temperature discharge derating full
SOX_TEMP_HIGH_CUTOFF_CHARGE	°C	45.0	low temperature charge derating start
SOX_TEMP_HIGH_LIMIT_CHARGE	°C	55.0	low temperature charge derating full
SOX_SOC_CUTOFF_CHARGE	0.01%	8500	high SOC derating starts
SOX_SOC_LIMIT_CHARGE	0.01%	9500	high SOC derating full extent
SOX_SOC_CUTOFF_DISCHARGE	0.01%	1500	low SOC derating starts
SOX_SOC_LIMIT_DISCHARGE	0.01%	500	low SOC derating full extent
SOX_VOLT_CUTOFF_CHARGE	mV	2400	high voltage derating starts
SOX_VOLT_LIMIT_CHARGE	mV	2550	high voltage derating full extent
SOX_VOLT_CUTOFF_DISCHARGE	mV	2000	low voltage derating starts
SOX_VOLT_LIMIT_DISCHARGE	mV	1750	low voltage derating full extent

Configuration Example for Lithium-Ion NCA/Graphite or NMC/Graphite Chemistries

NAME	UNIT	VALUE	DESCRIPTION
SOX_CURRENT_MAX_CONTINUOUS_CHARGE	A	80.00	maximum current continuous charge
SOX_CURRENT_MAX_CONTINUOUS_DISCHARGE	A	200.00	maximum current continuous discharge
SOX_CURRENT_LIMP_HOME	A	40.00	discharge current in limp home emergency mode
SOX_TEMP_LOW_CUTOFF_DISCHARGE	°C	25.0	low temperature discharge derating start
SOX_TEMP_LOW_LIMIT_DISCHARGE	°C	-10.0	low temperature discharge derating full
SOX_TEMP_LOW_CUTOFF_CHARGE	°C	20.0	low temperature charge derating start
SOX_TEMP_LOW_LIMIT_CHARGE	°C	10.0	low temperature charge derating full
SOX_TEMP_HIGH_CUTOFF_DISCHARGE	°C	45.0	low temperature discharge derating start
SOX_TEMP_HIGH_LIMIT_DISCHARGE	°C	55.0	low temperature discharge derating full
SOX_TEMP_HIGH_CUTOFF_CHARGE	°C	35.0	low temperature charge derating start
SOX_TEMP_HIGH_LIMIT_CHARGE	°C	45.0	low temperature charge derating full
SOX_SOC_CUTOFF_CHARGE	0.01%	8500	high SOC derating starts
SOX_SOC_LIMIT_CHARGE	0.01%	9500	high SOC derating full extent
SOX_SOC_CUTOFF_DISCHARGE	0.01%	1500	low SOC derating starts
SOX_SOC_LIMIT_DISCHARGE	0.01%	500	low SOC derating full extent
SOX_VOLT_CUTOFF_CHARGE	mV	4000	high voltage derating starts
SOX_VOLT_LIMIT_CHARGE	mV	4100	high voltage derating full extent
SOX_VOLT_CUTOFF_DISCHARGE	mV	3100	low voltage derating starts
SOX_VOLT_LIMIT_DISCHARGE	mV	2750	low voltage derating full extent

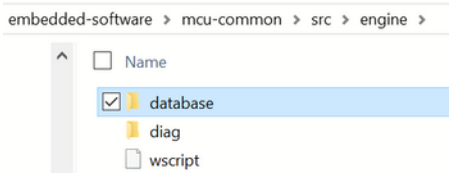
Engine:

28.6 Database

The `database` module is part of the `Engine` layer.

The `database` module allows the user to store and retrieve data. The database along with the `diag` module runs in the highest priority. This **must** not be changed.

A quick introduction on how to add additional information to the database is found *here*.



28.6.1 Module Files

Driver:

- `embedded-software\mcu-common\src\engine\database\database.c` (databasec)
- `embedded-software\mcu-common\src\engine\database\database.h` (databaseh)

Driver Configuration:

- `embedded-software\mcu-primary\src\engine\config\database_cfg.c` (databasecfgprimaryc)
- `embedded-software\mcu-primary\src\engine\config\database_cfg.h` (databasecfgprimaryh)
- `embedded-software\mcu-secondary\src\engine\config\database_cfg.c` (databasecfgsecondaryc)
- `embedded-software\mcu-secondary\src\engine\config\database_cfg.h` (databasecfgsecondaryh)

28.6.2 Detailed Description

The `DATA_Task()` task is running with the highest priority inside the `ENG_TSK_Engine`.

To store and read data the database introduces two functions:
, the database module

- `DB_WriteBlock(..)` and
- `DB_ReadBlock(..)`.

Block Diagram

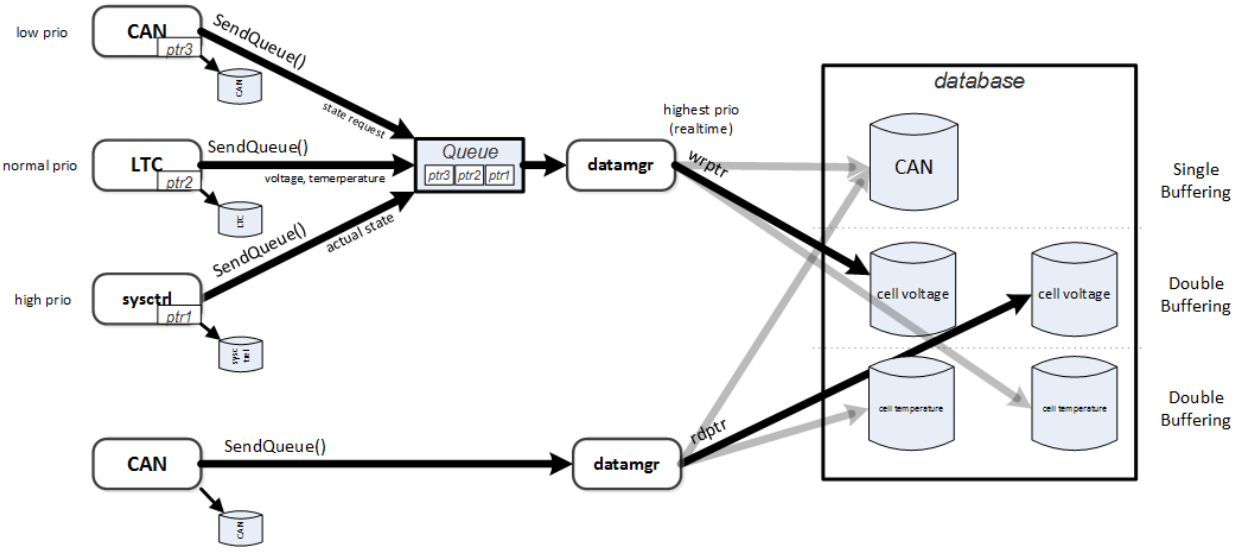


Fig. 28.7: SendQueue

28.6.3 Configuration

The configuration of the database consists of the following parts:

`database_cfg.h`

- Number of **data blocks** in `DATA_BLOCK_ID_TYPE_e`
- Human understandable alias for the datablocks (e.g., `DATA_BLOCK_ID_ALLGPIOVOLTAGE` for `DATA_BLOCK_2`3). (See line 171.)
- Typedefed struct of the actual data. This always consists of a timestamp, the previous timestamp and then the arbitrary data.

`database_cfg.c`

- A variable for the database block `DATA_BLOCK_ALLGPIOVOLTAGE_s data_block_ltc_allgpiovoltages[DOUBLE_BUFFERING];` (See line 181.)
- The variable must be introduced to the database header variable `DATA_BASE_HEADER_s data_base_header[] = {..}`.

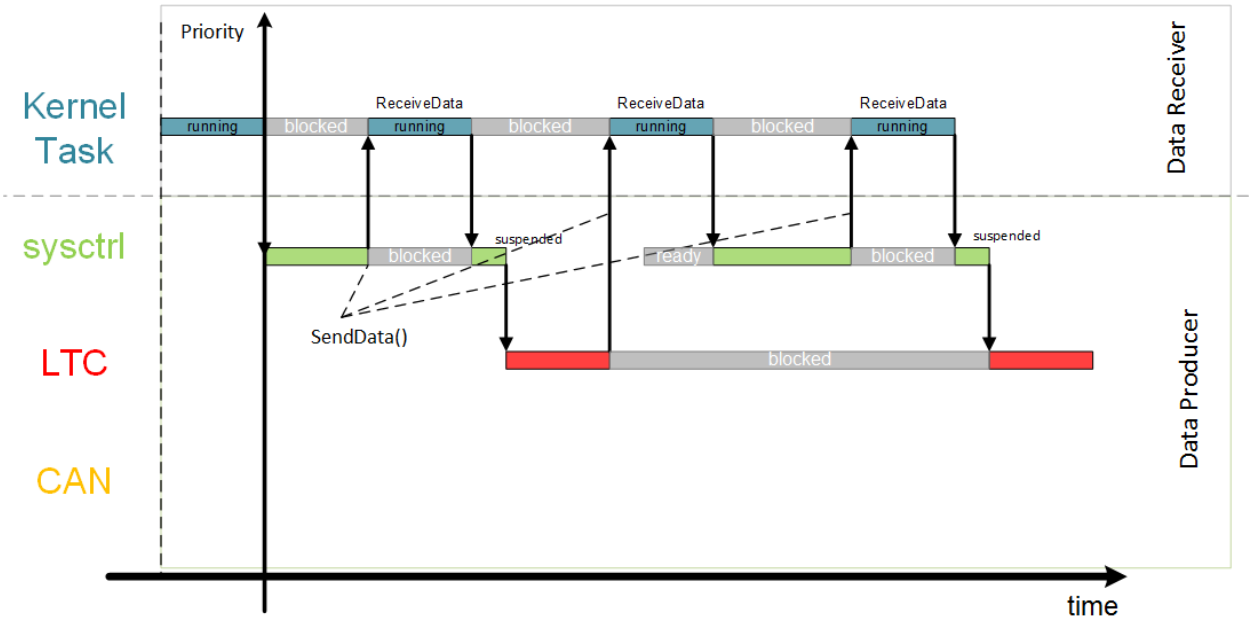


Fig. 28.8: Timing

28.6.4 Usage/Examples

For examples of usage and adaptions see:

- function `BAL_Activate_Balancing_Voltage()` in `embedded-software\mcu-primary\src\application\bms.c`
- *Add/read/write a database entry*.

28.7 Diagnosis

The `uart` module is part of the `Engine` layer.

The diagnose module is responsible for error handling, error reporting and system monitoring.

28.7.1 Module Files

Driver:

- `embedded-software\mcu-common\src\engine\diag\diag.c` (diagc)
- `embedded-software\mcu-common\src\engine\diag\diag.h` (diagh)

Driver Configuration:

- `embedded-software\mcu-primary\src\engine\config\diag_cfg.c` (diagcfgprimaryc)
- `embedded-software\mcu-primary\src\engine\config\diag_cfg.h` (diagcfgprimaryh)
- `embedded-software\mcu-primary\src\engine\config\diag_id_cfg.h` (diagidcfgprimaryh)

- `embedded-software\mcu-secondary\src\engine\config\diag_cfg.c` (diagcfgsecondaryc)
- `embedded-software\mcu-secondary\src\engine\config\diag_cfg.h` (diagcfgsecondaryh)
- `embedded-software\mcu-secondary\src\engine\config\diag_id_cfg.h` (diagidcfgsecondaryh)

28.7.2 Description

The `diag` module consists of 2 independent main parts, diagnosis handling and system monitoring. When configured, reported errors are logged into the global diagnosis memory and a callback function can be triggered.

The handler counts errors and calls a callback function when the configured error threshold is exceeded. The callback function is called again only when the error counter go back to zero after the error threshold was reached.

The initialization of the diagnosis module has to be done during start-up after the diagnosis memory is available (e.g. after Backup-SRAM is accessible). The Backup-SRAM Flag `DIAG_DATA_IS_VALID` indicates the data validity in diagnosis memory.

Two types of handling are defined:

- the general handler with debounce filter and thresholds for entering and exiting error state
- the contactor handler which counts all switching actions and reports when contactors are opened while the current flowing through the battery is above the configured threshold.

28.7.3 Usage

Diagnosis Handling

For using the diagnosis handler for a specific check in a module, a free diagnosis id has to be defined in `diag_cfg.h` and included as an additional diagnosis channel in `diag_cfg.c`:

```
diag_cfg.h:
#define     DIAG_ISOMETER_ERROR                 DIAG_ID_37          // Device error,␣
↪invalid measurement result

diag_cfg.c:
DIAG_CH_CFG_s  diag_ch_cfg[]= {
  ...
  {DIAG_ISOMETER_ERROR, DIAG_ERROR_SENSITIVITY_MID,  DIAG_RECORDING_ENABLED, DIAG_
↪ENABLED, callbackfunction},
  ...
};
```

Where error counting is needed, the diagnosis handler has to be called in the following way:

```
if( <error detected>)
{
     retVal = DIAG_Handler(DIAG_ISOMETER_ERROR, DIAG_EVENT_OK, 0, 0);
     if (retVAl != DIAG_HANDLER_RETURN_OK)
     {
          /* here implement local (directly) diagnosis handling */
     }
}
```

(continues on next page)

(continued from previous page)

```
else
{
     DIAG_Handler(DIAG_ISOMETER_ERROR, DIAG_EVENT_NOK, 0, 0);
}
```

The callback function

```
/**
 * @brief  dummy callback function of diagnosis events
 */
void callbackfunction(DIAG_CH_ID_e ch_id, DIAG_EVENT_e event) {
    /* Dummy function -> empty */
}
```

is called when the error threshold is reached or when the counter goes back to zero after the threshold was reached. Typically, an error flag is set or unset in the callback function. This database entry is updated periodically in the 1ms engine task.

System Monitoring

For using the system monitor for a specific task or function, a free monitoring channel ID has to be defined in `diag_cfg.h`:

```
typedef enum {
    DIAG_SYSMON_DATABASE_ID         = 0,   /*!< diag entry for database
↪*/
    DIAG_SYSMON_SYS_ID              = 1,   /*!< diag entry for sys
↪*/
    DIAG_SYSMON_BMS_ID              = 2,   /*!< diag entry for bms
↪*/
    DIAG_SYSMON_CONT_ID             = 3,   /*!< diag entry for contactors
↪*/
    DIAG_SYSMON_ILCK_ID             = 4,   /*!< diag entry for contactors
↪*/
    DIAG_SYSMON_LTC_ID              = 5,   /*!< diag entry for ltc
↪*/
    DIAG_SYSMON_ISOGUARD_ID         = 6,   /*!< diag entry for ioguard
↪*/
    DIAG_SYSMON_CANS_ID             = 7,   /*!< diag entry for can
↪*/
    DIAG_SYSMON_APPL_CYCLIC_1ms     = 8,   /*!< diag entry for application 10ms task
↪*/
    DIAG_SYSMON_APPL_CYCLIC_10ms    = 9,   /*!< diag entry for application 10ms task
↪*/
    DIAG_SYSMON_APPL_CYCLIC_100ms   = 10,  /*!< diag entry for application 100ms task
↪*/
    DIAG_SYSMON_MODULE_ID_MAX       = 11   /*!< end marker do not delete
↪*/
} DIAG_SYSMON_MODULE_ID_e;
```

and a new channel configured in `diag_cfg.c`:

```
DIAG_SYSMON_CH_CFG_s  diag_sysmon_ch_cfg[] = {
   ...
   {DIAG_SYSMON_ISOGUARD_ID,   DIAG_SYSMON_CYCLICTASK,  400, DIAG_RECORDING_ENABLED,
↪DIAG_ENABLED, callbackfunction},
```

(continues on next page)

(continued from previous page)

```
    ...
};
```

In this example, a timeout of 400 ms is defined. In the corresponding task or cyclic called function, a notification to the system monitor has to be done within the configured timeout by passing the state value, here 0 (ok),

Example:

```
my_isoguardfunction() {
    ...
    DIAG_SysMonNotify(DIAG_SYSMON_ISOGUARD_ID,0);
    ...
}
```

28.8 System

The (`sys` module) takes care of all system related tasks. Periodic system checks can be implemented here. In the default configuration, it starts other important statemachines (e.g., `ILCK`, `CONT`, `BMS`)

28.8.1 Module Files

Driver:

- `embedded-software\mcu-primary\src\engine\sys\sys.c` (sysprimaryc)
- `embedded-software\mcu-primary\src\engine\sys\sys.h` (sysprimaryh)
- `embedded-software\mcu-secondary\src\engine\sys\sys.c` (syssecondaryc)
- `embedded-software\mcu-secondary\src\engine\sys\sys.h` (syssecondaryh)

Driver Configuration:

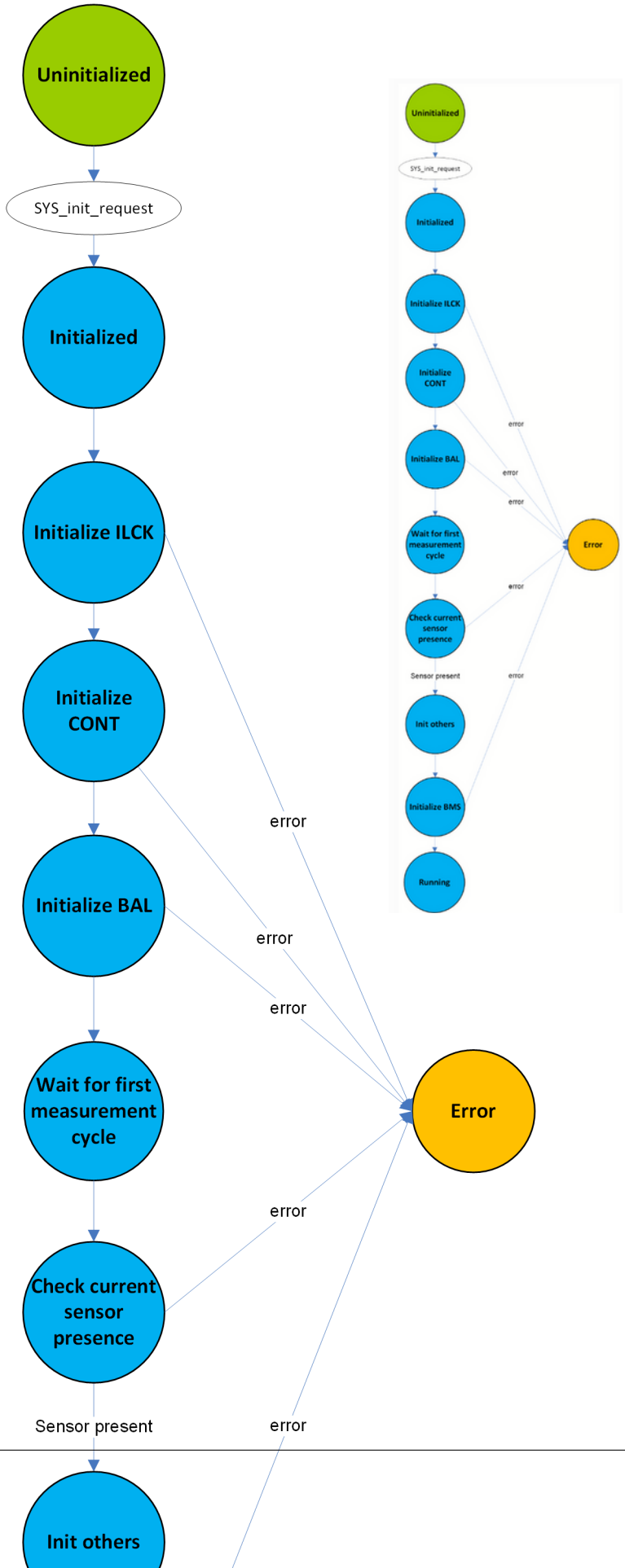
- `embedded-software\mcu-primary\src\engine\config\sys_cfg.c` (syscfgprimaryc)
- `embedded-software\mcu-primary\src\engine\config\sys_cfg.h` (syscfgprimaryh)
- `embedded-software\mcu-secondary\src\engine\config\sys_cfg.c` (syscfgsecondaryc)
- `embedded-software\mcu-secondary\src\engine\config\sys_cfg.h` (syscfgsecondaryh)

28.8.2 Structure

Fig. 28.9 shows the statemachine implementing the `sys` module.

First, the following statemachines are initialized with state requests:

- `ILCK`
- `CONT`
- `BAL`


Uninitialized
SYS_init_request
Initialized
Initialize ILCK
Initialize CONT
Initialize BAL
Wait for first measurement cycle
Check current sensor presence
Sensor present
Init others
Initialize BMS
Running
Error
error

The state of these modules is checked to ensure that the initialization was successful. A timeout mechanism is used: the SYS statemachine goes into an error state goes in case one of the initializations goes wrong.

The next step is to wait until a complete measurement cycle of the cell voltages and temperatures has taken place before enabling the periodic sending of values per CAN, to avoid the transmission of invalid data at startup.

The presence of a current sensor is then checked.

The last step is the initialization of BMS state machine with a state request.

After all these steps, the system is fully running. If necessary, additional periodic system checks can be implemented in the SYS state machine.

28.9 NVRAM-Handler

The nvramhandler module is part of the Engine layer.

The nvramhandler module provides an interface to store or read data in the non-volatile memory. The data can either by stored cyclic (i.e. every 120s) or triggered (i.e. when opening the contacors while current is floating). The nvramhandler module is an abstraction layer to the actual implementation of the non-volatile ram (i.e. EEPROM or memory card).

28.9.1 Module Files

Driver:

- embedded-software\mcu-primary\src\engine\nvramhandler\nvramhandler.c (nvramhandlerc)
- embedded-software\mcu-primary\src\engine\nvramhandler\nvramhandler.h (nvramhandlerh)

Driver Configuration:

- embedded-software\mcu-primary\src\engine\config\nvramhandler_cfg.c (nvramhandlercfgc)
- embedded-software\mcu-primary\src\engine\config\nvramhandler_cfg.h (nvramhandlercfgh)

28.9.2 Structure

The struct NVRAM_BLOCK_s contains the definition of an data block that is stored in the non-volatile ram..

```
typedef struct NVRAM_BLOCK {
    NVRAM_state_e state;                /*!< state of datahandler block          */
    uint32_t lastUpdate;                /*!< time of last nvram update           */
    NVRAM_UpdateType_e mode;            /*!< update mode (cyclic or triggered)   */
    uint32_t updateCycleTime_ms;        /*!< cycle time of algorithm             */
    uint32_t phase_ms;                  /*!< start time when executing algorithm */
    STD_RETURN_TYPE_e (*funcRD)(void);  /*!< read callback function              */
    STD_RETURN_TYPE_e (*funcWR)(void);  /*!< write callback function             */
} NVRAM_BLOCK_s;
```

state contains the current state of the nvramhandler:

```
typedef enum NVRAM_state {
    NVRAM_wait     = 0,
    NVRAM_write    = 1,
    NVRAM_read     = 2,
} NVRAM_state_e;
```

Each block should be configured with the state `NVRAM_wait`. `lastUpdate` holds the timestamp when the data block was updated (written) the last time. `mode` is either `NVRAM_Cyclic` or `NVRAM_Triggered`. `updateCycleTime_ms` is the cyclic update time of the data block when configured in cyclic mode. `phase_ms` is a constant timing offset to prevent multiple write/read requests to the non-volatile storage at the same time. `(*funcRD)(void)/(*funcWR)(void)` is the callback pointer to the specific read/write function of the data block which are dependent on the specific implementation of the non-volatile ram.

The function `NVRAM_dataHandlerInit` intializes the struct `NVRAM_BLOCK_s nvram_dataHandlerBlocks`. It sets all states to `NVRAM_wait` and the lastUpdate to 0.

`NVRAM_dataHandler` loops over the different data blocks and updates the respective datablocks if either the cycle time is elapsed or a trigger has been received via the function `NVRAM_setWriteRequest(NVRAM_BLOCK_ID_TYPE_e blockID)`. If a block is configured in cyclic mode it can still be triggered. It is then updated after receiving the trigger and afterwards again is updated every `updateCycleTime_ms`. Furthermore the function `NVRAM_dataHandler` can execute asynchronous read requests from the non-volatile storage over the function `NVRAM_setReadRequest(NVRAM_BLOCK_ID_TYPE_e blockID)`.

`NVRAM_BLOCK_ID_TYPE_e blockID` is used to indicate which data block should be accessed. This implementation structure is analog the implementation of the database.

```
#define NVRAM_BLOCK_ID_OPERATING_HOURS          NVRAM_BLOCK_00
#define NVRAM_BLOCK_ID_CELLTEMPERATURE          NVRAM_BLOCK_01
#define NVRAM_BLOCK_ID_CONT_COUNTER             NVRAM_BLOCK_02
```

Defines are mapped to the enum and the enum corresponds to the order of the configured non-volatile data blocks in strcut `NVRAM_BLOCK_s nvram_dataHandlerBlocks`.

foxBMS-Modules:

28.10 CANSIGNAL

The `cansignal` module is part of the `Module` layer.

The `cansignal` module is a software module used to handle the conversion from data providers like a database or measurement modules to the `can` module. It works similar to a typical IPO (input-processing-output) pattern.

28.10.1 Module Files

Driver:

- `embedded-software\mcu-common\src\module\cansignal\cansignal.h` (cansignalh)
- `embedded-software\mcu-common\src\module\cansignal\cansignal.c` (cansignalc)

Driver Configuration:

- `embedded-software\mcu-primary\src\module\config\cansignal_cfg.h` (cansignalcfgh)
- `embedded-software\mcu-primary\src\module\config\cansignal_cfg.c` (cansignalcfgc)

28.10.2 Detailed Description

File Structure and Interfaces

The `cansignal` module is a simple one-file module with one-file configuration:

- The module itself consists of one source file `cansignal.c` and its associated `cansignal.h`
- The configuration is given in `cansignal_cfg.c` and its associated `cansignal_cfg.h`

The external interface to the `cansignal` module is very easy and consists of just two functions:

- CANS_Init
- CANS_MainFunction

Message vs signal:
* Message is the entire CAN message
* Signal is the user data in the message

CANS_Init is for parameter and configuration checking of the `cansignal` module.

CANS_MainFunction is for data processing and should be called periodically.

Data Flow

The data flow is generally divided in two different domains, one for reception of CAN messages and signal value distribution, the other for transmission of CAN message and signal value assembling/message composition.

When a CAN message is received physically, it is stored in a buffer which is polled periodically by `CANS_MainFunction()`. In case of a match, the corresponding signals of the message are extracted, scaled and handed over to a data consumer via the callback setter function.

When a CAN message needs to be transmitted physically, the signals data belonging to this CAN message are collected via their getter callback function. From this signals data, the CAN message is assembled. If everything worked fine, it is stored in a buffer and sent out over the CAN peripheral hardware.

fig. 28.10 shows the data flow used to assemble a CAN message.

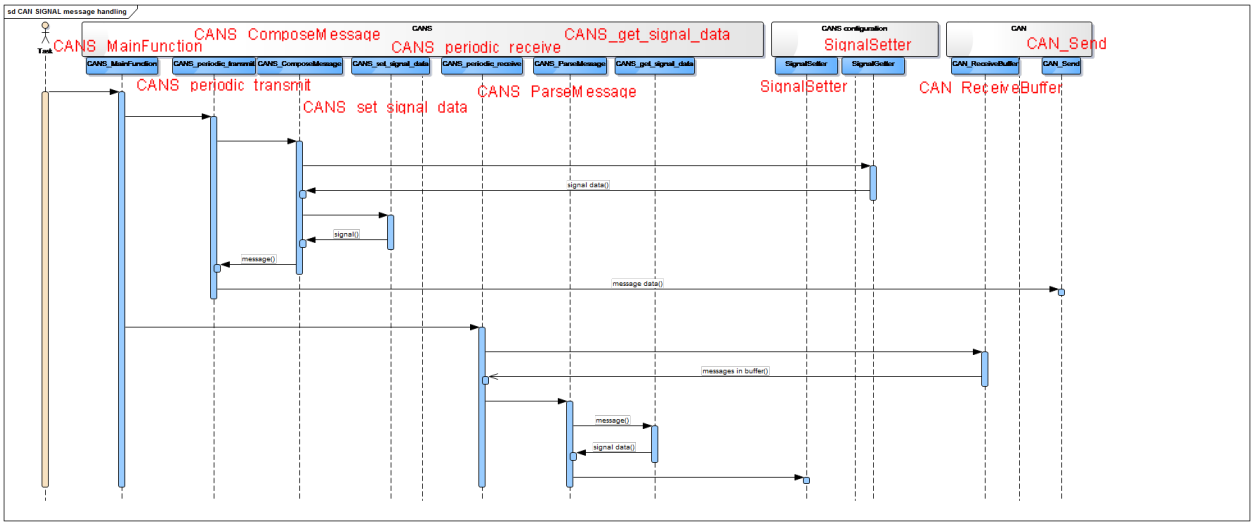


Fig. 28.10: CAN data flow to assemble messages

Control Flow

The module operates from a single function call to `CANS_MainFunction()`.

First the periodic message transmission is handled in this function. If correspondence of an internal tick counter to a message repetition time and phase is detected (i.e., the periodic time expired) the CAN message is composed from its signals. Therefore all signals, which are included in a message, are collected via their getter callback and written to the message data block at the right position in the right length. This message data block together with the ID, Data Length Code and so on, is handed over to the `can` module, which handles the low level transmission to the CAN specific peripheral registers.

The message reception in turn is done by reading out the buffer of the `can` module. Then the signals configuration is searched for this message(s). If one signal is represented in these received messages it is extracted and handed over to the setter callback function configured for this signal.

28.10.3 CAN Configurations

Default Configuration

The configuration comprises:

- enums for both receive and transmit message names (`CANS_messagesTx_e` respectively `CANS_messagesRx_e`)
- enums for both receive and transmit signal names (`CANS_CANx_signalsTx_e` respectively `CANS_CANx_signalsRx_e`)
- arrays for signal definition, for both receive and transmit signals
- callback function implementations for getting and setting of signals
- callback function implementations for post-processing after transmission and reception of messages

Custom Configuration Examples

See FAQ section on how to manually add/delete a CAN message.

28.10.4 Usage/Examples

To use `can` module and `cansignal` module with a correct *CAN Configurations*, just call `CANS_MainFunction` in the cyclic tasks timeslot, that is configured in `CANS_TICK_MS`.

28.11 Contactor

The `contactor` module is part of the `foxBMS-Modules` layer.

The `contactor` module switches the contactors according to the requests that are made to it. It checks the feedback line of each contactors periodically.

The foxBMS Master Unit can control up to 9 contactors: 6 on the BMS-Master Board and 3 on the BMS-Extension Board. The according control and feedback circuit is exemplarily shown for contactor 0 in Fig. 17.11. The contactor is switched on and off by an AQV25G2S photoMOS (IC1001) by the primary microcontroller MCU0. Every contactor channel is protected with slow blowing fuse (F1001) type Schurter UMT-250 630mA (3403.0164.xx). The free wheeling diode D1001 is not populated. It has to be inserted when contactors are used, that do not provide an internal free wheeling diode. The contactor interface also supports a feedback functionality for contactors with auxiliary contacts. The contactor status can be read back by MCU0 via an ADUM3300 (IC1103).

28.11.1 Module Files

Driver:

- `embedded-software\mcu-primary\src\module\contactor\contactor.c` (contactorc)
- `embedded-software\mcu-primary\src\module\contactor\contactor.h` (contactorh)

Driver Configuration:

- `embedded-software\mcu-primary\src\module\config\contactor_cfg.c` (contactorcfgc)
- `embedded-software\mcu-primary\src\module\config\contactor_cfg.h` (contactorcfgh)

The CONT pins used in the contactor_cfg.h file are defined in the "Contactors" section of mcu-primary\src\driver\config\io_mcu_cfg.h. Also defined there are Interlock control and feedback pins.

```
148   *Contactors' Controll and Feedback Pins
149   */
150   #define IO_PIN_INTERLOCK_CONTROL              IO_PD_4
151   #define IO_PIN_INTERLOCK_FEEDBACK             IO_PD_5
152   #define IO_PIN_CONTACTOR_0_CONTROL            IO_PI_2
153   #define IO_PIN_CONTACTOR_0_FEEDBACK           IO_PH_13
154   #define IO_PIN_CONTACTOR_1_CONTROL            IO_PI_1
155   #define IO_PIN_CONTACTOR_1_FEEDBACK           IO_PH_14
156   #define IO_PIN_CONTACTOR_2_CONTROL            IO_PI_0
157   #define IO_PIN_CONTACTOR_2_FEEDBACK           IO_PH_15
158   #define IO_PIN_CONTACTOR_3_CONTROL            IO_PA_9
159   #define IO_PIN_CONTACTOR_3_FEEDBACK           IO_PC_6
160   #define IO_PIN_CONTACTOR_4_CONTROL            IO_PA_8
161   #define IO_PIN_CONTACTOR_4_FEEDBACK           IO_PC_7
162   #define IO_PIN_CONTACTOR_5_CONTROL            IO_PC_9
163   #define IO_PIN_CONTACTOR_5_FEEDBACK           IO_PC_8
```

28.11.2 Structure

Fig. 28.11 shows the statemachine managing the contactors in foxBMS.

Three states are implemented:

- `STANBDY`
- `NORMAL`
- `CHARGE`

Need to add another state called ENGINE, which stands for engine start. In this state, the Engine relay is closed.

`STANDBY` corresponds to the state where all the contactors are open. `NORMAL` and `CHARGE` correspond to a state where the contactors of one of the powerlines are closed to allow current flowing.

The `CHARGE` state is available only if the switch `BS_SEPARATE_POWERLINES` in `embedded-software\mcu-primary\src\general\config\batterysystem_cfg.h` is set to 1. It corresponds to the use of a separate powerline ~~compared~~ in addition to the powerline used in the normal state.

Also, in the same file, we have BS_NR_Of_CONTACTORS defined as 6.

The `PRECHARGE` state performs the transition between `STANDBY` and `NORMAL` /`CHARGE`: the `plus precharge contactor` is closed before the `plus main contactor`, to avoid shorting the battery when closing the contactors.

The transition between the states is made through state request. These are made by the `bms` module. From `STANDBY`, the state machine can transition to `NORMAL` or `CHARGE`, or the opposite. No transition is possible directly between `NORMAL` and `CHARGE`. The transition to/from ENGINE is the same.

28.11.3 Switching Counter

Contactors have a specified number of switching cycles. The number of switching cycles varies depending on if the contactor has been opened under load or not. The values have to be taken from the manufacturers manual.

foxBMS uses the following strategy to allow ~~it~~ the user to store the number of switching cycles each contactor has done and track their usage.

The counters for **closing**, **opening** and **opening under load** are stored in the variable `bkpsram_contactors_count`. The counters are backup into the the second non-volatile memory (EEPROM), when the foxBMS Master Unit is power cycled. This ensures, that even if the RTC battery is at some point completely discharged, the second last counter number of all contactors is still safely stored and available for diagnostic purposes.

> **Warning: Limitations of the contactor swichting counter**
>
> If the RTC battery is removed or empty AND the mcu is power cycled, all countings since the last saving (therefore a valid power cycle with charged RTC battery) on the EEPROM are lost.

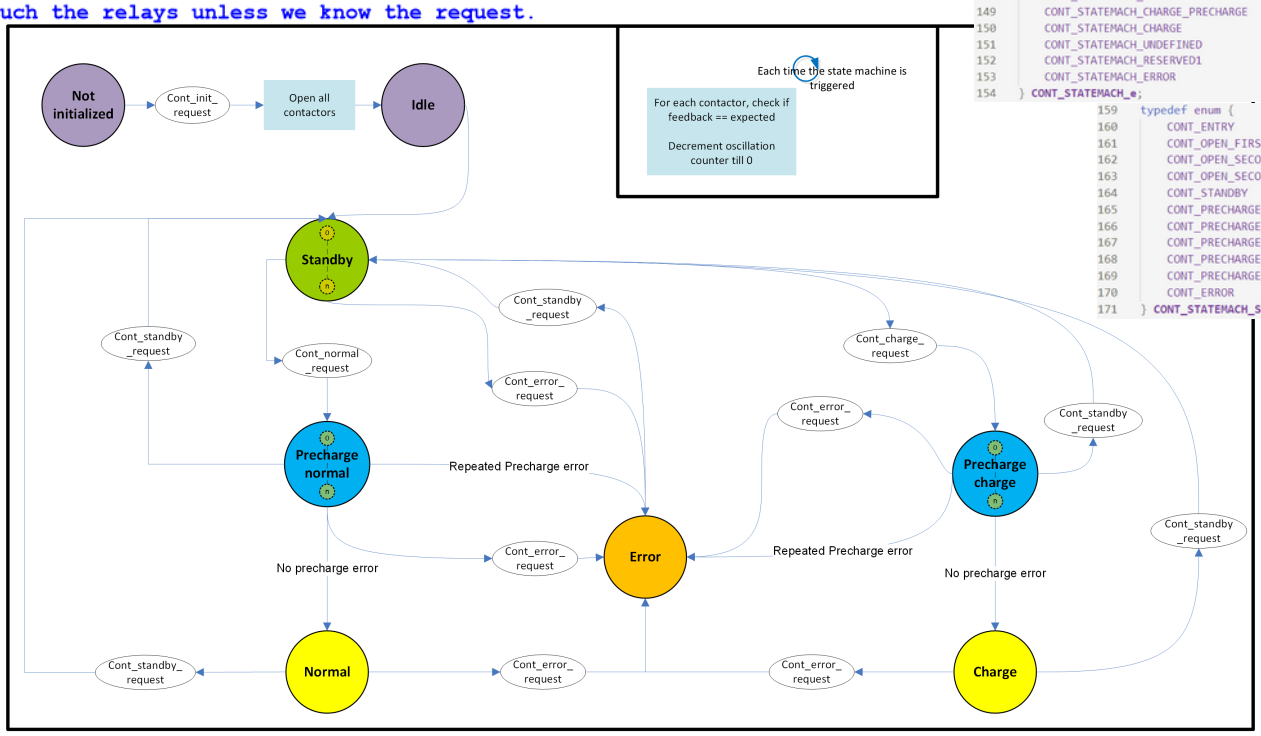
For details of the counter implementation see the modules `diag` and the `bkpsram`.

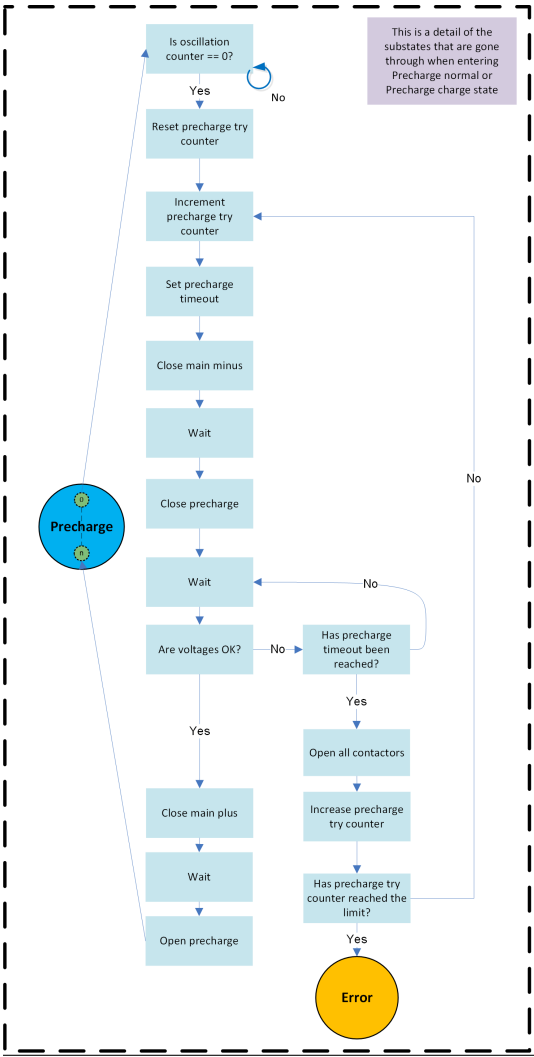
back up SRAM

Note: for the Phx project, we should check the request first before opening all the contactors. This can prevent the case of loosing the power due to the restart of the system. Due to the usage of the latching type relays, we need not touch the relays unless we know the request.


```
typedef enum {
    /* Init-Sequence */
    CONT_STATEMACH_UNINITIALIZED            = 0,
    CONT_STATEMACH_INITIALIZATION           = 1,
    CONT_STATEMACH_INITIALIZED              = 2,
    CONT_STATEMACH_IDLE                     = 3,
    CONT_STATEMACH_STANDBY                  = 4,
    CONT_STATEMACH_PRECHARGE                = 5,
    CONT_STATEMACH_NORMAL                   = 6,
    CONT_STATEMACH_CHARGE_PRECHARGE         = 7,
    CONT_STATEMACH_CHARGE                   = 8,
    CONT_STATEMACH_UNDEFINED                = 20,
    CONT_STATEMACH_RESERVED1                = 0x80,
    CONT_STATEMACH_ERROR                    = 0xF0,
} CONT_STATEMACH_e;
```

```
typedef enum {
    CONT_ENTRY
    CONT_OPEN_FIRST_CONTACTOR
    CONT_OPEN_SECOND_CONTACTOR_MINUS
    CONT_OPEN_SECOND_CONTACTOR_PLUS
    CONT_STANDBY
    CONT_PRECHARGE_CLOSE_MINUS
    CONT_PRECHARGE_CLOSE_PRECHARGE
    CONT_PRECHARGE_CLOSE_PLUS
    CONT_PRECHARGE_CHECK_VOLTAGES
    CONT_PRECHARGE_OPEN_PRECHARGE
    CONT_ERROR
} CONT_STATEMACH_SUB_e;
```





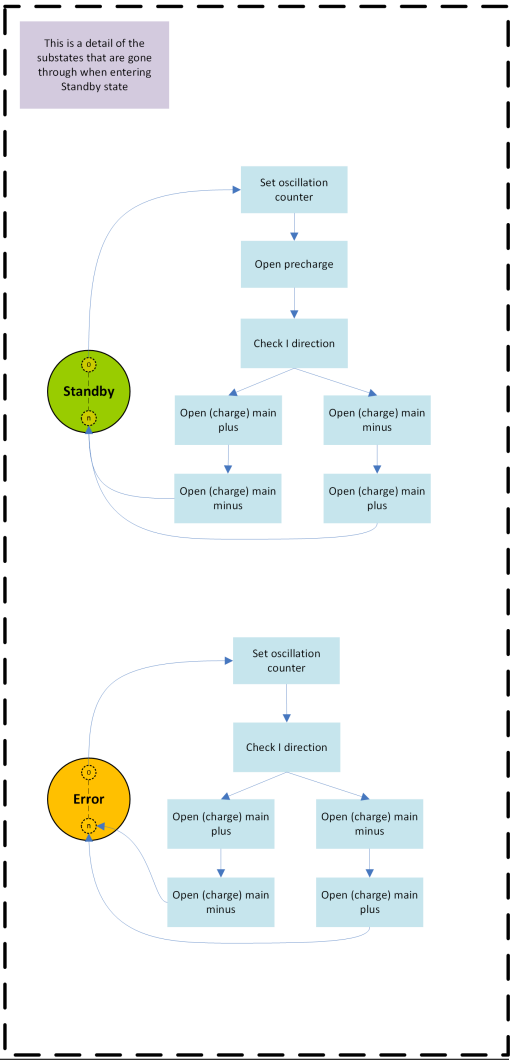
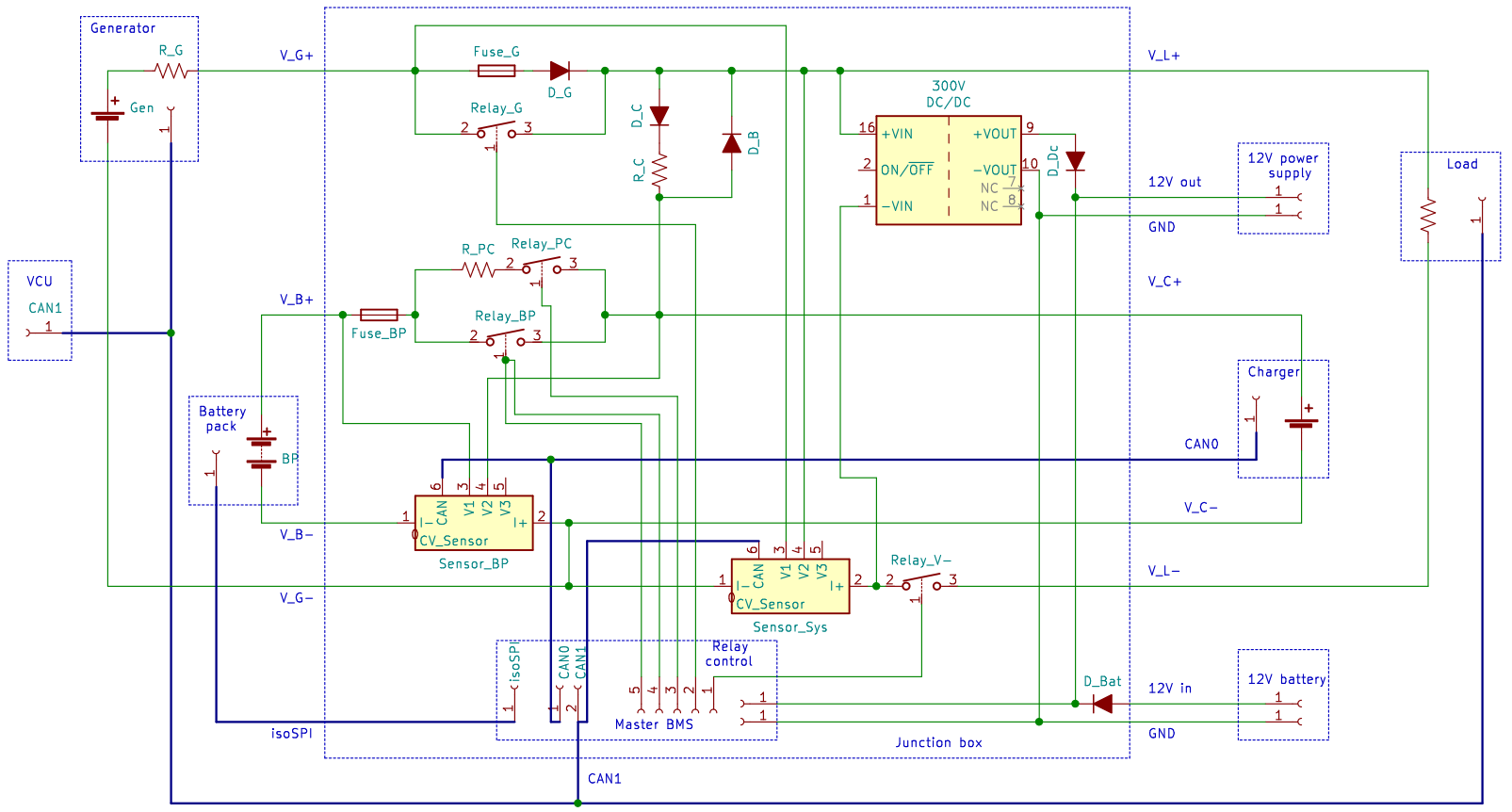


Fig. 28.11: Contactor state machine



Notes:
1. The load can be the CAV load or the active load or the Xing Mobility battery packs.
2. The VCU, engine/generator, master BMS, a current/voltage sensor, and the load will be on a CAN bus.
3. There will be another CAN bus connecting the master BMS, another current/voltage sensor, and the charger.
4. isoSPI is a communication link between the master and slave BMSs.
5. We may not need to have Relay_V−.
6. Relay_BP is a latch relay for added safety.
7. The casing of the junction box is connected to the negative terminal of the 12 V power source, which is used to power the master BMS to control the relays.
8. The 12 V input power source can be replaced by the 24/28 V power source as needed. In this case, a DC/DC converter is needed to get the 12 V power.

We have the following relays:
* Relay_BP = CONT_MAIN_PLUS (latching)
* Relay_PC = CONT_PRECHARGE_PLUS (non-latching)
* Relay_V- = CONT_MAIN_MINUS (latching)
* Relay_G = CONT_GENERATOR (non-latching)

Note:
* To go to charge state: (1) close Relay_PC, (2) close Relay_BP, and (3) open Relay_PC.
* To go to normal state: (1) close Relay_V-, (2) close Relay_PC, (3) close Relay_BP, and (4) open Relay_PC.
* To go to engine state: (1) close Relay_G, (2) close Relay_PC, (3) close Relay_BP, and (4) open Relay_PC.

Sheet: /		
File: System_diagram_v02.sch		
Title:		
Size: USLetter	Date:	**Rev:**
KiCad E.D.A. eeschema (5.1.2)−1		Id: 1/1

28.11.4 Module Files

Driver:

- embedded-software\mcu-primary\src\module\contactor\contactor.h
- embedded-software\mcu-primary\src\module\contactor\contactor.c

Driver Configuration:

- embedded-software\mcu-primary\src\module\config\contactor_cfg.h
- embedded-software\mcu-primary\src\module\config\contactor_cfg.c

28.11.5 Configuration of the Contactors

A contactor physically consists of an control pin, an optional feedback pin, which can be configured as normally open or normally closed. This hardware condition is mapped in the software in the contactor set-up files in `contactor_cfg.c` and `contactor_cfg.h`.

The entry point to change the contactor configuration is the configuration header file `contactor_cfg.h`. At first, the general naming of the contactors is mapped to something meaningful for the contactors (such as IO_PIN_CONTACTOR_0_FEEDBACK.

Then, they are further mapped to something more meaningful, such as CONT_MAIN_PLUS_FEEDBACK in contactor_cfg.h, as shown below.

```
#define CONT_PRECHARGE_PLUS_CONTROL         PIN_MCU_0_CONTACTOR_0_CONTROL
#define CONT_PRECHARGE_PLUS_FEEDBACK        PIN_MCU_0_CONTACTOR_0_FEEDBACK
```

Note: if the contactor had no feedback one would define

```
#define CONT_PRECHARGE_PLUS_CONTROL         PIN_MCU_0_CONTACTOR_0_CONTROL
#define CONT_PRECHARGE_PLUS_FEEDBACK        CONT_HAS_NO_FEEDBACK
```

Next, the contactors are identified by setting up the typedefed enumeration `CONT_NAMES_e`. If there are three contactors, two in the positive path (one for precharge and a main contactor) and a main contactor in the negative path, the configuration of the `CONT_NAMES_e` looks like this (the names are free of choice):

```
typedef enum {
    CONT_MAIN_PLUS              = 0,
    CONT_PRECHARGE_PLUS         = 1,
    CONT_MAIN_MINUS             = 2,
} CONT_NAMES_e;
```

Need to add three more relays.

At this point the configuration of the header is done and the configuration is finished in `contactor_cfg.c`.

The pins where the contactors are connected to the MCU and the hardware configuration are now composed to a contactor object/struct. By using the example above, the contactors must now be configured accordingly. It is assumed that the contactors have a feedback and are of type normally open. The setup summed up would look like this:

Contactor	Control pin	Feedback pin	Hardware feedback configuration
CONT_MAIN_PLUS	CONT_MAIN_PLUS_CONTROL	CONT_MAIN_PLUS_FEEDBACK	CONT_FEEDBACK_NORMALLY_OPEN
CONT_PRECHARGE_PLUS	CONT_PRECHARGE_PLUS_CONTROL	CONT_PRECHARGE_PLUS_FEEDBACK	CONT_FEEDBACK_NORMALLY_OPEN
CONT_MAIN_MINUS	CONT_MAIN_MINUS_CONTROL	CONT_MAIN_MINUS_FEEDBACK	CONT_FEEDBACK_NORMALLY_OPEN

This would result in the following configuration in the source code for the hardware configuration:

Note that some of the IO related are defined in the following file:

- embedded-software\mcu-common\src\driver\io\io.h (ioc)

```
CONT_CONFIG_s cont_contactors_config[BS_NR_OF_CONTACTORS] = {
     {CONT_MAIN_PLUS_CONTROL,         CONT_MAIN_PLUS_FEEDBACK,         CONT_FEEDBACK_
↪NORMALLY_OPEN},
     {CONT_PRECHARGE_PLUS_CONTROL,    CONT_PRECHARGE_PLUS_FEEDBACK,    CONT_FEEDBACK_
↪NORMALLY_OPEN},
     {CONT_MAIN_MINUS_CONTROL,        CONT_MAIN_MINUS_FEEDBACK,        CONT_FEEDBACK_
↪NORMALLY_OPEN}
};                                              Need to add three more relays.
```

The corresponding feedback state configuration would look like this:

```
 CONT_ELECTRICAL_STATE_s cont_contactor_states[BS_NR_OF_CONTACTORS] = {
     {0,     CONT_SWITCH_OFF},
     {0,     CONT_SWITCH_OFF},
     {0,     CONT_SWITCH_OFF},          Need to add three more relays.
};
            CONT_SWITCH_OFF
```

```
typedef struct {
    CONT_ELECTRICAL_STATE_TYPE_s set;
    CONT_ELECTRICAL_STATE_TYPE_s feedback;
} CONT_ELECTRICAL_STATE_s;
```

Note: The configuration in `cont_contactors_config[]` must have the same order as defined in `CONT_NAMES_e`.

The parameters after the feedback type parameter display the state in which the contactor is. The initial state of the contactor is always switched off. The state variables store the set value (`TRUE` or `FALSE`), the expected feedback (`CONT_SWITCH_OFF` or `CONT_SWITCH_ON`), the measured feedback (`CONT_SWITCH_OFF` or `CONT_SWITCH_ON`) and the according timestamp to each (`os_timer`).

At this point the setup of the contactors is finished.

Fig. 28.12 gives a visualization of the configuration.

```
309  typedef enum {
310      CONT_SWITCH_OFF     = 0,
311      CONT_SWITCH_ON      = 1,
312      CONT_SWITCH_UNDEF   = 2,
313  } CONT_ELECTRICAL_STATE_TYPE_s;
                                  e
```

28.11.6 Interaction

The `bms` module uses the `contactor` module APIs.

28.12 Interlock

The `interlock` module switches the interlock according to the requests that are made to it. It checks the feedback line of the interlock periodically.

28.12.1 Module Files

Driver:

- `embedded-software\mcu-common\src\module\interlock\interlock.h` (interlockh)
- `embedded-software\mcu-common\src\module\interlock\interlock.c` (interlockc)

Driver Configuration:

- `embedded-software\mcu-primary\src\module\config\interlock_cfg.h` (primary-interlockcfgh)
- `embedded-software\mcu-primary\src\module\config\interlock_cfg.c` (primary-interlockcfgc)

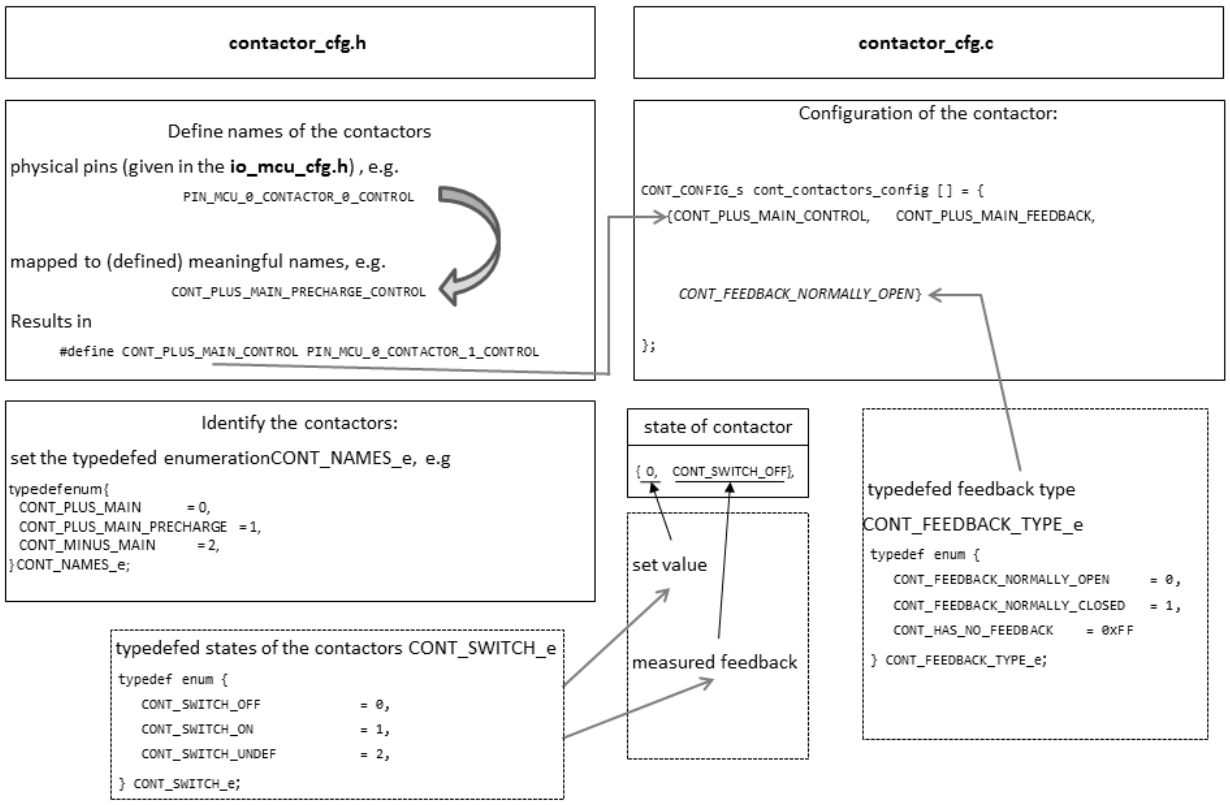


Fig. 28.12: Contactor configuration

- embedded-software\mcu-secondary\src\module\config\interlock_cfg.h (secondaryinterlockcfgh)
- embedded-software\mcu-secondary\src\module\config\interlock_cfg.c (secondaryinterlockcfgc)

28.12.2 Structure

Fig. 28.13 shows the statemachine managing the interlock in foxBMS.

Two states are implemented:

- CLOSED
- OPEN

Requests are made to the interlock statemachine by BMS. BMS closes the interlock when entering STANDBY and opens the interlock when entering the ERROR state.

The interlock feedback is checked periodically. If the set value dose not match the measured feedback (e.g., the interlock was opened outside of foxBMS), the corresponding error flag will be set. The application implemented in BMS will then get the information and react accordingly.

28.12.3 Interaction

The bms module uses the contactor module APIs.

28.13 Isoguard

The isoguard module is part of the foxBMS-Module layer.

The isoguard module measures the insulation resistance between the insulated and the active high-voltage conductors of the battery and the reference earth (e.g., chassis ground/Kl.31 in automotive applications).

28.13.1 Module Files

Driver:

- embedded-software\mcu-primary\src\module\isoguard\isoguard.h (isoguardh)
- embedded-software\mcu-primary\src\module\isoguard\isoguard.c (isoguardc)
- embedded-software\mcu-primary\src\module\isoguard\ir155.h (ir155h)
- embedded-software\mcu-primary\src\module\isoguard\ir155.c (ir155c)

Driver Configuration:

- embedded-software\mcu-primary\src\module\config\isoguard_cfg.h (isoguardcfgh)
- embedded-software\mcu-primary\src\module\config\isoguard_cfg.c (isoguardcfgc)

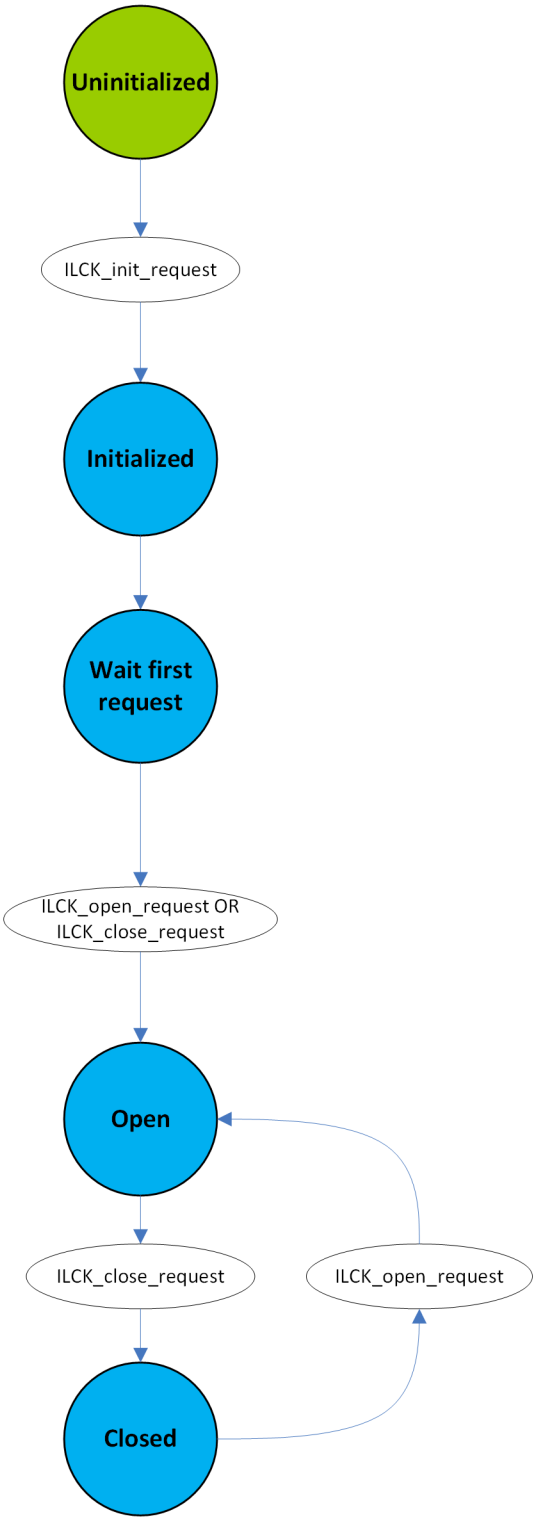


Fig. 28.13: Interlock state machine

28.13.2 Detailed Description

File Structure and Interfaces

The `isoguard` module is separated into two different layers, a bottom and a top layer. The top layer consists of the `isoguard.c` and its associated `isoguard.h` file. It provides an easy interface to initialize the `isoguard` module and to access the measurement data. The insulation threshold to differentiate between a good and bad insulation can be set in the `isoguard_cfg.h` file. The bottom layer contains the `ir155.c` and `ir155.h` files. They are both dedicated to the Bender IR155-3204[1] hardware (IR155-3203/-3204/-3210 are supported) and handle the connection of the Bender to the MCU, as well as the evaluation and interpretation of the measurement data.

The external interface to the `isoguard` module is simple and consists of three functions and is viewable in fig. 28.14:

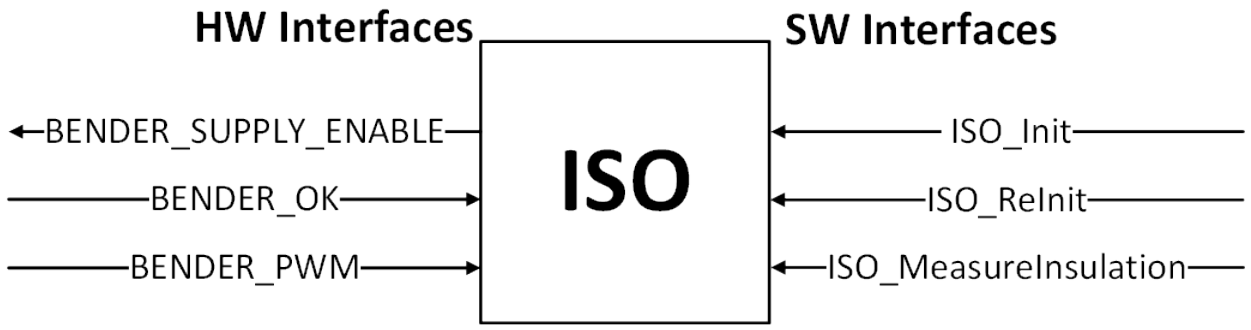


Fig. 28.14: External isoguard driver interface

`ISO_Init(void)` Initializes software `isoguard` module and enables the hardware Bender module

`ISO_ReInit(void)` Resets software `isoguard` module and hardware Bender module

`ISO_MeasureInsulation(void)` Measures and evaluates the insulation

Configuration

The configuration is done in the `isoguard_cfg.h` file and consists of two defines:

```
#define ISO_CYCLE_TIME                  200
```

`ISO_CYCLE_TIME` is the periodic calling time of the `ISO_MeasureInsulation(void)` function and must not be chosen lower than 150ms due to the lowest possible frequency of the Bender of 10Hz. The default value is 200ms.

```
#define ISO_RESISTANCE_THRESHOLD        400
```

`ISO_RESISTANCE_THRESHOLD` specifies the resistance threshold to differentiate between good and bad insulation. This value has no impact if the threshold is set lower than the intern resistance threshold of the Bender insulation monitor. The default value is 400kOhm.

Initialization

The initialization is done via the interface function `ISO_Init(void)`, which is forwarded to the Bender insulation monitor specific initialization functions, as shown in fig. 28.15.

The most important steps in the initialization process are:

[1] Datasheet ISOMETER IR155-3204 [PDF] https://www.bender-de.com/fileadmin/products/doc/IR155-42xx-V004_DB_en.pdf

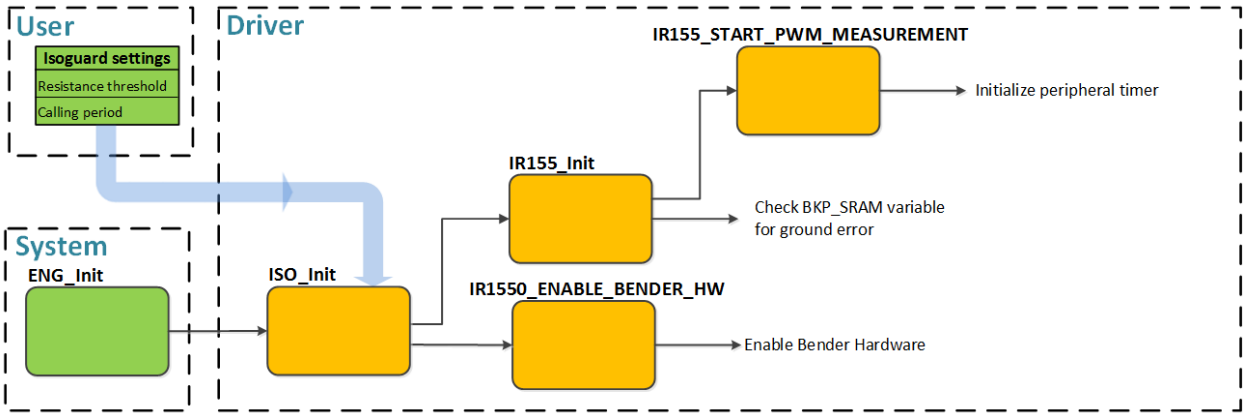


Fig. 28.15: Initialization of the `isoguard` module

- Enabling of PWM input measurement
- Reading of `BKP_SRAM` variable for previous Ground Error
- Set start-up time before measurement results are trustworthy (dependent on the previous state)
- Enabling of Bender hardware module

The function `ISO_ReInit(void)` resets the whole `isoguard` module and the initialization process is done again, including the waiting time until the measurement values are declared as trustworthy.

28.13.3 Usage

After initializing the `isoguard` module, the `ISO_MeasureInsulation(void)` function needs to be called periodically according to the set cycle time. The Bender insulation monitor measurement values are evaluated and then written into the database where further modules can operate on this data. Following measurement values are saved:

```
typedef struct {
    uint8_t valid;          // 0 -> valid, 1 -> resistance unreliable
    uint8_t state;          // 0 -> resistance/measurement OK , 1 -> resistance too␣
↪low or error
    uint8_t resistance;
    uint32_t timestamp;
    uint32_t previous_timestamp;
}DATA_BLOCK_ISOMETER_s;
```

The measured insulation is split into intervals according to the array `uint16 const static ir155_ResistanceInterval[7]`. For more detailed information see source code.

28.13.4 Observable SW-Behavior

The `data_block_isometer.timestamp`-variable must be always running, otherwise the measured values are not written into the database (Check if the define `ISO_ISOGUARD_ENABLE` is defined, otherwise the `isoguard` module is not activated). Depending on the used resistor, the following behavior can be observed:

Variable	Behavior
ir155_DC.resistance	Should equal the resistance value
ir155_DC.dutycycle	Depending on the resistance, see table below for more information
ir155_DC.OKHS_state	0 if measured resistance below factory set resistance threshold, otherwise 1
ir155_DC.mode	IR155_NORMAL_MODE
ir155_DC.state	IR155_RESIST_MEAS_GOOD if resistance greater than factory set resistance threshold, otherwise IR155_RESIST_MEAS_BAD
data_ data_block_isometer.valid	0; can be 1 if g rounderror occurred before reset. Should be 0 again after 25s
data_ data_block_isometer.state	0 if measured r resistance > ISO_RESISTANCE_THRESHOLD and no error, otherwise 1
data_ data_block_isometer.resistance	value between 0 and 7, see table below for more information

28.13.5 References

28.14 LTC

The driver communicates with the LTC6811-1 monitoring ICs in daisy-chain configuration. The ICs are used to:

- measure the battery cell voltages
- measure the voltages directly on the GPIOs
- measure the voltage of up to 32 inputs via I2C-driven multiplexer
- enable passive balancing of the connected battery cells

board

The slave version is configured in `src\module\config\ltc_cfg.h` by the define `SLAVE_BOARD_VERSION`.

- `SLAVE_BOARD_VERSION` must be set to `1` if version 1.xx of the foxBMS Slaves is used.
- `SLAVE_BOARD_VERSION` must be set to `2` if version 2.xx of the foxBMS Slaves is used. Version 2.xx is the default configuration.

28.14.1 Module Files

Driver:

- `embedded-software\mcu-common\src\module\ltc\ltc_defs.h` (ltc_defsh)
- `embedded-software\mcu-common\src\module\ltc\ltc_pec.c` (ltc_pecc)
- `embedded-software\mcu-common\src\module\ltc\ltc_pec.h` (ltc_pech)
- `embedded-software\mcu-common\src\module\ltc\ltc.c` (ltcc)
- `embedded-software\mcu-common\src\module\ltc\ltc.h` (ltch)

Driver Configuration:

- `embedded-software\mcu-primary\src\module\config\ltc_cfg.c` (primaryltccfgc)
- `embedded-software\mcu-primary\src\module\config\ltc_cfg.h` (primaryltccfgh)
- `embedded-software\mcu-secondary\src\module\config\ltc_cfg.c` (secondaryltccfgc)
- `embedded-software\mcu-secondary\src\module\config\ltc_cfg.h` (secondaryltccfgh)

28.14.2 Detailed Description

State Machine

The `ltc` module is implemented as a state machine. The operation of the state machine is described in fig. 28.16.

After initialization, the state machine goes in a measurement loop:

- measure voltages
- read measured voltages
- select multiplexer input
- measure selected input
- read multiplexer input
- check state requests
- balance cells

The function `LTC_SetStateRequest()` is used to make these state requests to the LTC6811-1 state machine.

If more than one multiplexer input is configured, only one is measured per measurement cycle. The next one is measured during the next cycle. When the last configured multiplexer input is reached, the sequence starts over.

Results Retrieval and Balancing Orders

The results of the measurements are written in the database. The balancing is made according to the control variable set in the database. The corresponding variables are:

```
DATA_BLOCK_CELLVOLTAGE_s          ltc_cellvoltage; //cell voltages
DATA_BLOCK_CELLTEMPERATURE_s      ltc_celltemperature; //cell temperature
DATA_BLOCK_MINMAX_s               ltc_minmax; //minimum and maximum values at battery␣
↪pack
                                              //level for voltages and temperatures
DATA_BLOCK_BALANCING_FEEDBACK_s   ltc_balancing_feedback; //result from balancing␣
↪feedback
                                                    //(is at least one cell␣
↪being balanced)
DATA_BLOCK_BALANCING_CONTROL_s    ltc_balancing_control; //balancing orders read from␣
↪database
DATA_BLOCK_SLAVE_CONTROL_s        ltc_slave_control;    //features on slave controlled␣
↪by I2C
```

Possible state requests

Following actions can be requested:

- write to IO port-expander
- read from IO port-expander
- read external temperature sensor on slave
- read global balancing feedback
- read from external slave EEPROM
- write to external slave EEPROM

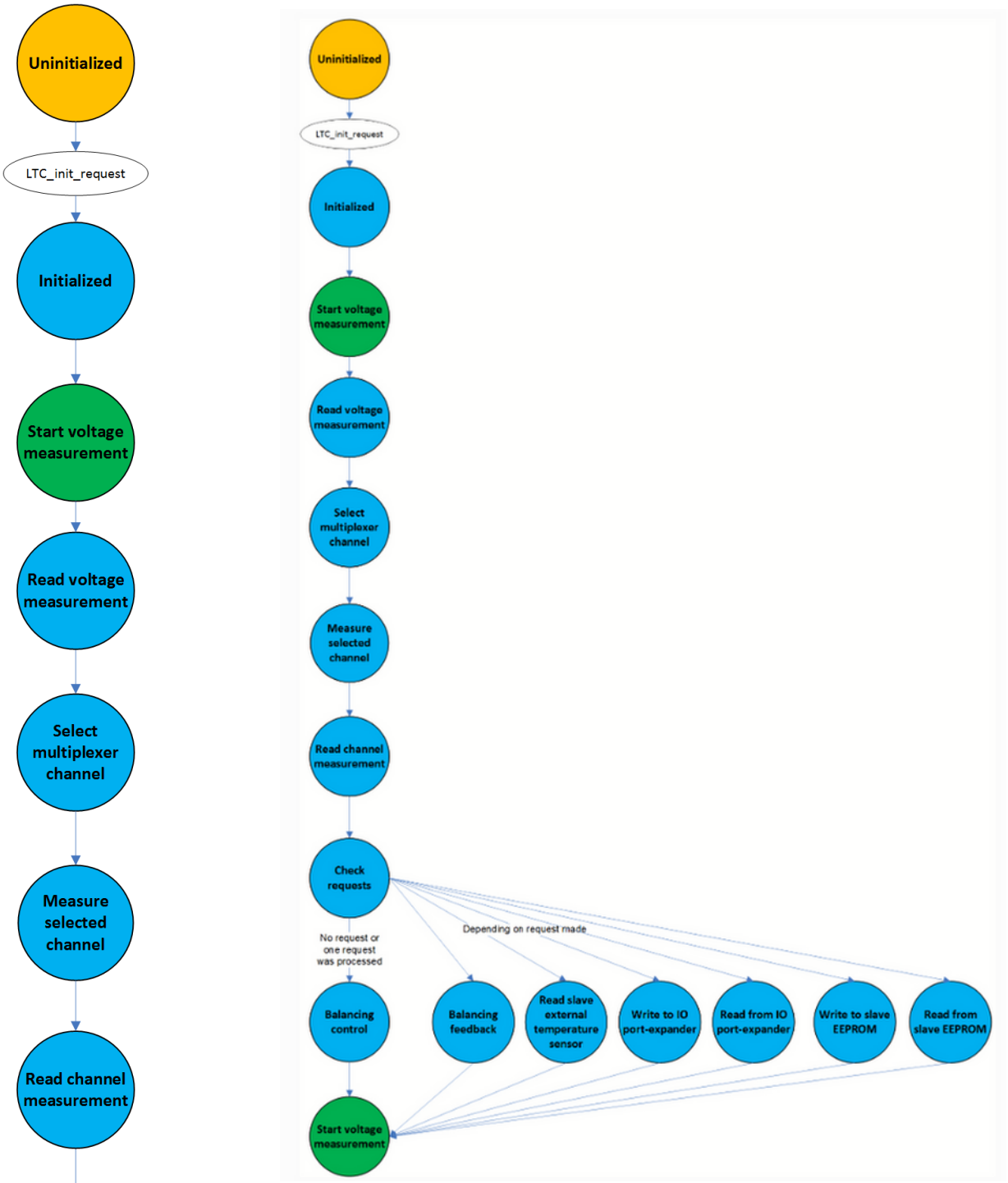

Uninitialized
LTC_init_request
Initialized
Start voltage measurement
Read voltage measurement
Select multiplexer channel
Measure selected channel
Read channel measurement
Check requests
No request or one request was processed
Depending on request made
Balancing control
Balancing feedback
Read slave external temperature sensor
Write to IO port-expander
Read from IO port-expander
Write to slave EEPROM
Read from slave EEPROM

Fig. 28.16

The corresponding requests are made with the following functions from the `measurement` module.

```
uint8_t MEAS_Request_IO_Write(void);
uint8_t MEAS_Request_IO_Read(void);
uint8_t MEAS_Request_Temperature_Read(void);
uint8_t MEAS_Request_BalancingFeedback_Read(void);
uint8_t MEAS_Request_EEPROM_Read(void);
uint8_t MEAS_Request_EEPROM_Write(void);
```

Before reading from the EEPROM, the address must be written to `eeprom_read_address_to_use` in the database block `DATA_BLOCK_SLAVE_CONTROL_s`. The result is stored in `eeprom_value_read` in the database block `DATA_BLOCK_SLAVE_CONTROL_s`. Once the read operation has been performed, the value `0xFFFFFFFF` is stored in `eeprom_read_address_to_use`.

Before writing to the EEPROM, the address must be written to `eeprom_write_address_to_use` in the block `DATA_BLOCK_SLAVE_CONTROL_s`. The data to be written must be stored in `eeprom_value_write` in the database block `DATA_BLOCK_SLAVE_CONTROL_s` before issuing the write request. Once the write operation has been performed, the value `0xFFFFFFFF` is stored in `eeprom_write_address_to_use`.

Measurement frequency

When no requests are made, with 8 LTC6811-1 monitoring ICs in the daisy-chain and 12 cell voltages, in normal measurement mode, a measurement cycle takes no more than 20ms. As a consequence, a measurement frequency of 50Hz can be achieved for the voltages.

If requests are made, the measurement cycle can last longer (e.g., access to the EEPROM on the slaves needs more time).

28.14.3 Configuration

SPI Interface

Line 152

In `ltc_cfg.h`, the SPI devices used by the LTC6811-1 is defined by the macro `SPI_HANDLE_LTC`. The SPI handle must be chosen from the list `SPI_HandleTypeDef spi_devices[]` in `spi_cfg.c`. The frequency of the used SPI must be adjusted with the configuration in `spi_cfg.c`, so that the frequency is not higher than 1MHz. This is the maximum allowed frequency for the LTC6811-1 IC. The function `LTC_SetTransferTimes()` sets the waiting times used for the `ltc` module automatically at startup. It uses `LTC_GetSPIClock()` to get the SPI clock frequency automatically.

Measurement Mode and Channel Selection

Three measurement modes are available for the voltage measurements:

- Normal
- Filtered
- Fast

These modes are defined in the LTC6811-1 datasheet [ltc_datasheet]. For cell voltage measurements, the macro `VOLTAGE_MEASUREMENT_MODE` is defined in `ltc_cfg.h`. For multiplexers measurement, the macro `GPIO_MEASUREMENT_MODE` is defined in `ltc_cfg.h`.

Changing the number of cell voltages

This number is changed in `batterysystem_cfg.h` with the define `BS_NR_OF_BAT_CELLS_PER_MODULE`.

In addition, the variable

```
const uint8_t ltc_voltage_input_used[BS_MAX_SUPPORTED_CELLS]
```

must be adapted, too, in `ltc_cfg.c`.

It has the size of `BS_MAX_SUPPORTED_CELLS`. If a cell voltage is connected to the LTC IC input, `1` must be written in the table. Otherwise, `0` must be written. More details can be found in the software FAQ: *How to configure the voltage inputs?*.

Multiplexer Sequence

In case of single GPIO measurements, the multiplexer sequence to be read is defined in `ltc_cfg.c` with the variable `LTC_MUX_CH_CFG_t ltc_mux_seq_main_ch1[]`. It is a list of elements of the following form:

```
{
    .muxID     = 1,
    .muxCh     = 3,
}
```

where the multiplexer to select (muxID, can be 0, 1, 2 or 3) and the channel to select (muxCh, from 0 to 7) are defined. Channel 0xFF means that the multiplexer is turned off. This is used to avoid two or more multiplexers to have their outputs in a low-impedance state at the same time.

Temperature Sensor Assignment

For temperature sensors, the following variable is used to give the correspondence between the channel measured and the index used:

```
uint8 ltc_muxsensortemperaturmain_cfg[8] =
{    8-1 ,  //channel 0
    7-1 ,  //channel 1
    6-1 ,  //channel 2
    5-1 ,  //channel 3
    4-1 ,  //channel 4
    3-1 ,  //channel 5
    2-1 ,  //channel 6
    1-1   //channel 7
}
```

In the above example, channel 0 of the multiplexer corresponds to temperature sensor 8. If muxseqptr is the multiplexer sequence of type `LTC_MUX_CH_CFG_t` as defined above, the sensor index is retrieved in the variable `sensor_idx` via:

```
sensor_idx = ltc_muxsensortemperaturmain_cfg[muxseqptr->muxCh];
```

Further information on the configuration of the temperature sensors can be found in the software FAQ: *How to add/remove temperature sensors?*.

28.14.4 References

28.15 CAN

The `can` module is part of the `Driver` layer.

The driver interfaces the CAN networks and supports the CAN protocols version 2.0A and B. It has been designed to manage incoming messages and transmit messages with the help of message buffers. It is highly customizable and works efficiently with a high number of messages and minimum CPU load.

28.15.1 Module Files

Driver:

- `embedded-software\mcu-common\src\driver\can\can.h` (canh)
- `embedded-software\mcu-common\src\driver\can\can.c` (canc)

Driver Configuration:

- `embedded-software\mcu-primary\src\driver\config\can_cfg.h` (cancfgh)
- `embedded-software\mcu-primary\src\driver\config\can_cfg.c` (cancfgc)

28.15.2 Detailed Description

Main Features

- Two CAN nodes (CAN0 and CAN1)
- CAN protocol version 2.0A, B active
- Bit rates up to 1Mbit/s

Transmission:

- Configurable transmit priority
- Configurable transmit message buffer
- Possible bypassing of transmit message buffer
- Periodic message transmitting

Reception:

- Identifier list and mask mode featured
- Configurable software receive message buffer
- Possible bypassing of receive message buffer

For a detailed overview of the CAN peripheral see source[1].

[1] Datasheet RM0090 [PDF] http://www.st.com/web/en/resource/technical/document/reference_manual/DM00031020.pdf

File Structure and Interfaces

The `can` module is a one-file module with one-file configuration: The module itself consists of one `can.c` and its associated `can.h` file. The configuration is given in `can_cfg.c` and its related `can_cfg.h` file.

The external interface to the can module is easy and consists of seven functions and is viewable in fig. 28.17.

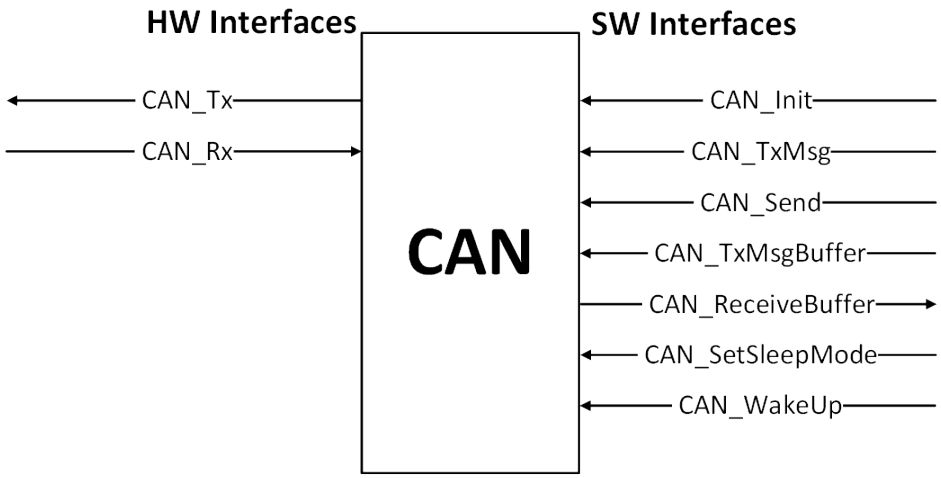


Fig. 28.17: External CAN Driver interface

`CAN_Init(void)` Initializes CAN settings and message filtering

`CAN_TxMsg(CAN_NodeTypeDef_e canNode, uint32_t msgID, uint8_t* ptrMsgData, ... )` Transmits message directly on the CAN bus

`CAN_Send(CAN_NodeTypeDef_e canNode, uint32_t msgID, uint8_t* ptrMsgData, ... )` Adds message to transmit buffer

`CAN_TxMsgBuffer(CAN_NodeTypeDef_e canNode)` Transmits a CAN message from transmit buffer

`CAN_ReceiveBuffer(CAN_NodeTypeDef_e canNode, Can_PduType* msg)` Reads a CAN message from RxBuffer

`CAN_SetSleepMode(CAN_NodeTypeDef_e canNode)` Set CAN node to sleep mode

`CAN_WakeUp(CAN_NodeTypeDef_e canNode)` Wake CAN node up from sleep mode

Data/Control Flow

Transmission There are two different possibilities to transmit a message on the CAN bus. Both possibilities are equally correct and transmit the messages via interrupts. The first possibility is to transmit the message directly through the `CAN_TxMsg` function. The second possibility is to add the message first with a call of `CAN_Send` to the message transmit buffer and then later transmit it with the `CAN_TxMsgBuffer` function from the buffer on the CAN bus.

Reception After a successful reception a messages is either stored in the receive message buffer or it bypasses the buffer and is directly interpreted during the ISR via callback functions. The buffered messages can be accessed through the `CAN_ReceiveBuffer` function.

A detailed overview over how the CAN driver software interacts with the hardware is shown in the following flow chart in fig. 28.18.

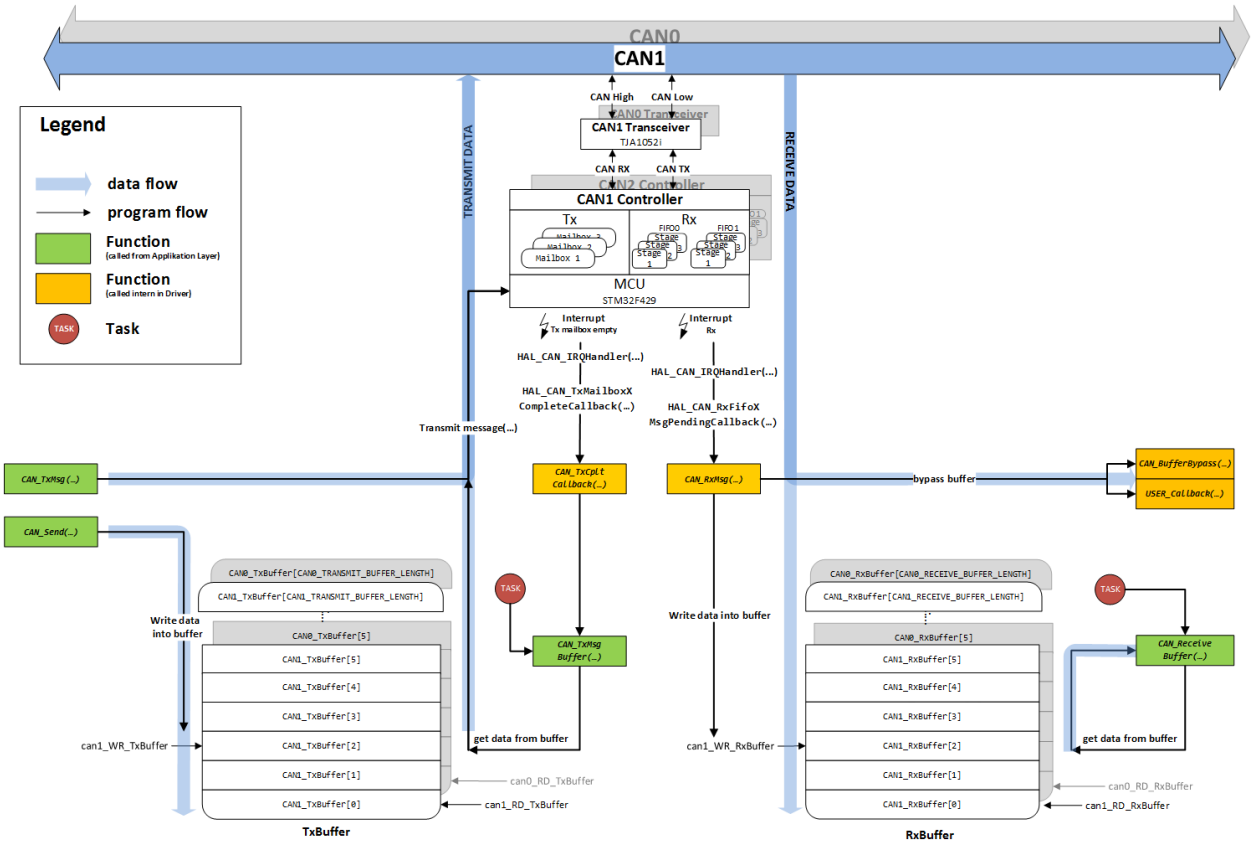


Fig. 28.18: CAN flow chart of the interaction between hardware and software

28.15.3 Configuration

The activation of CAN0 and CAN1 is set in the `can_cfg.h` file. Additionally are in this file the message buffer options and the message reception options to be configured.

Additional configurations in this file are

The `can_cfg.c` file is responsible for the general CAN options (CAN bit-rate, sending behavior). The received messages are to be defined here.

The interrupt priority of the CAN interrupts is set in the `nvic_cfg.c` file.

can_cfg.h

The configurable options are

```
 /* CAN bus baudrate */
 //#define CAN_BAUDRATE 1000000
 #define CAN_BAUDRATE 500000
 //#define CAN_BAUDRATE 250000
 //#define CAN_BAUDRATE 125000

 /* CAN options */
 #define CAN_USE_CAN_NODE0               1

 /* transmit buffer */
 #define CAN0_USE_TRANSMIT_BUFFER        1
 #define CAN0_TRANSMIT_BUFFER_LENGTH     16

 /* receive buffer */
 #define CAN0_USE_RECEIVE_BUFFER         1
 #define CAN0_RECEIVE_BUFFER_LENGTH      16

/* Number of messages that will bypass the receive buffer and will be interpreted␣
↪right
 * on reception. Set the respective IDs and implement the wished functionality either␣
↪in
 * an individual callback function or in the default STD_RETURN_TYPE_e CAN_
↪BufferBypass(...)
 * function in the can.c file. Use bypassing only for important messages because of␣
↪handling
 * during ISR */
#define CAN0_BUFFER_BYPASS_NUMBER_OF_IDs 1
```

There are four predefined values for the CAN bus baud-rate and only one define must be enabled at the same time. If another baud-rate is wished, the time quants need to be calculated and then this option can be added in the configuration. If the define `CAN_USE_CAN_NODE1` is disabled (i.e., set to zero), all other options of the chosen CAN bus (i.e., CAN0 or CAN1) have no influence on the program execution.

Warning: The bypass possibility should be used carefully because the message interpreting is then done in the context of the corresponding ISR-Handler and can lead to a violation of the timing constraints. The recommended setting is to use both buffers and to bypass as little messages as possible.

can_cfg.c

The `CAN_HandleTypeDef hcan` configures the CAN bus transfer settings. The default setting is a bit rate of 0.5MHz, automatic bus-off management, automatic wake-up mode, automatic retransition mode, locked receive FIFOs

against overrun and an identifier driven transmission of messages. The bitrate is set through the prescaler and two bit segments (BS) and is calculated with the following formula:

$$bitrate = \frac{\frac{CAN\_CLK}{Prescaler}}{SyncSeg + BS1 + BS1}$$

With the STM32F429 this calculates to a default bitrate of:

$$bitrate = \frac{\frac{42MHz}{6}}{1 + 6 + 7} = 0.5MHz$$

The periodic transmit messages are defined in the `const CAN_MSG_TX_TYPE_s can_CAN1_messages_tx[]` array structs:

```
typedef struct  {
    uint32_t ID;                    /*!< CAN message id */
    uint8_t DLC;                    /*!< CAN message data length code */
    uint32_t repetition_time;       /*!< CAN message cycle time */
    uint32_t repetition_phase;      /*!< CAN message startup (first send) offset */
    can_callback_funcPtr cbk_func;  /*!< CAN message callback after message is sent␣
↪or received */
} CAN_MSG_TX_TYPE_s;   S = struct
```

Messages that are transmitted asynchronously don't need to be declared in this struct. Moreover the settings for message reception need to be set in the source file. The received messages are defined in the `CAN_MSG_RX_TYPE_s can_RxMsgs[CAN_NUMBER_OF_RX_IDs]` array structs:

```
typedef struct CAN_MSG_RX_TYPE {
    uint32_t ID;    /*!< message ID */
    uint32_t mask;  /*!< mask or 0x0000 to select list mode */
    uint8_t DLC;    /*!< data length */
    uint8_t RTR;    /*!< rtr bit */
    uint32_t fifo;  /*!< selected CAN hardware (CAN_FILTER_FIFO0 or CAN_FILTER_
↪FIFO1) */
    STD_RETURN_TYPE_e (*func)(uint32_t ID, uint8_t*, uint8_t, uint8_t);  /*!<␣
↪callback function */
} CAN_MSG_RX_TYPE_s;
```

Each message can get an individual callback function assigned to. If a message shall bypass the reception buffer and be interpreted right on reception, the message identifier needs to be additionally defined in the `can_bufferBypass_RxMsgs[]` array. The callback function of the bypassed message is then automatically called on reception of the message. If no callback function is defined (`NULL` in the callback parameter), then the default function `STD_RETURN_TYPE_e CAN_BufferBypass(...)` in the `can.c` file is called and the interpretation needs to be implemented there.

More detailed information is provided in the comments in the source code and the STM32F429 microcontroller reference manual[1].

28.15.4 Usage

The following sections describe the usage of the CAN driver.

Initialization

The calling hierarchy and the control/data flow of the initialization process is visible in fig. 28.19. The user data input is made in the `can_cfg.c/h` files.

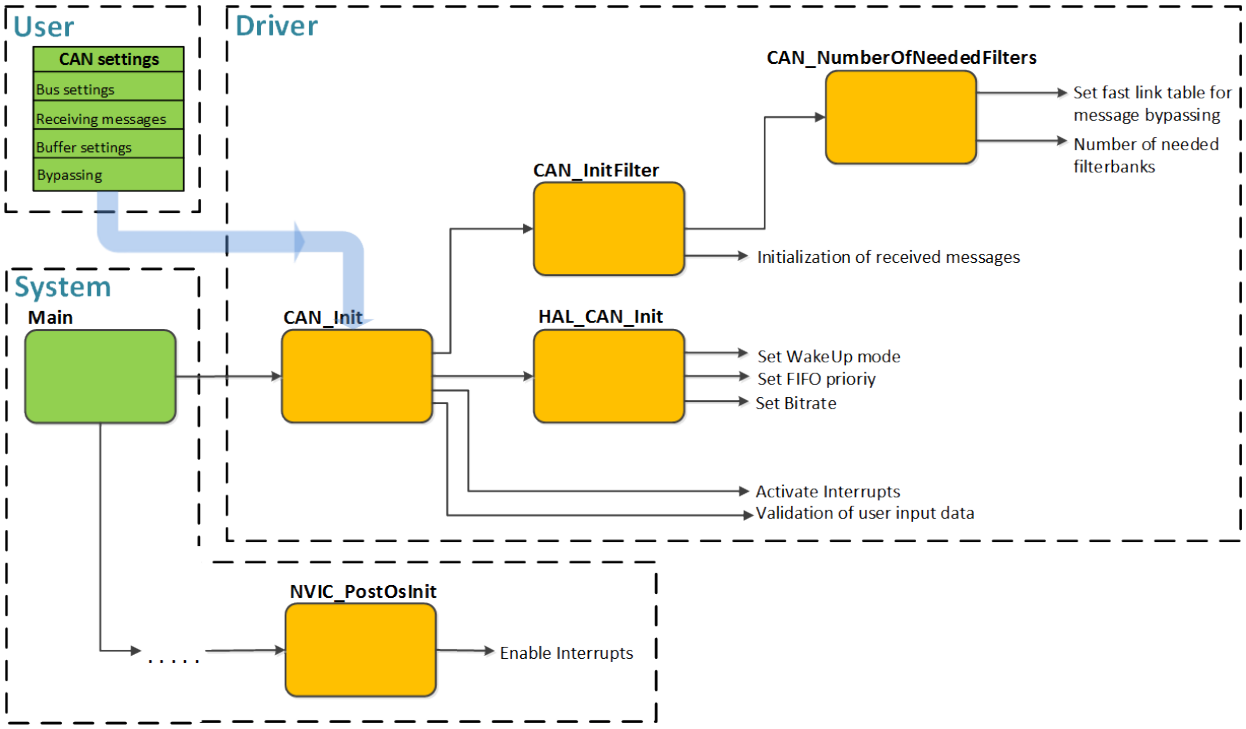


Fig. 28.19: CAN driver initialization calling hierarchy

Send Messages

There are two different possibilities to transmit a message on the CAN bus. Both possibilities are equally correct and transmit the message via interrupt. The first possibility is to transmit the message directly with the `CAN_TxMsg(...)` function. The second possibility is to add the message first, with `CAN_Send(...)`, to the transmit buffer and then later transmit it buffer with the `CAN_TxMsgBuffer(...)` function on the CAN bus.

The recommended transmission possibility via message buffer is visible in the following sequence diagram in fig. 28.20:

Receive Messages

What are those filter banks?

CAN messages are received after configuring the hardware filter banks. As mentioned above, the reception initialization is based on the receiving messages and generated automatically. After a successful reception the messages are either stored in the receive message buffer or corresponding to the configuration, they bypass the buffer and are directly interpreted. The bypassing is executed during the ISR and therefore, to avoid a violation of timing constraints, as little messages as possible should be bypassed. The buffered messages are interpreted asynchronous to their reception by the function `CAN_ReceiveBuffer(...)` from the application layer. If the receive buffer is disabled, then all messages are interpreted right on reception during the ISR.

The interpreting mechanism is shown in the sequence diagram in fig. 28.21.

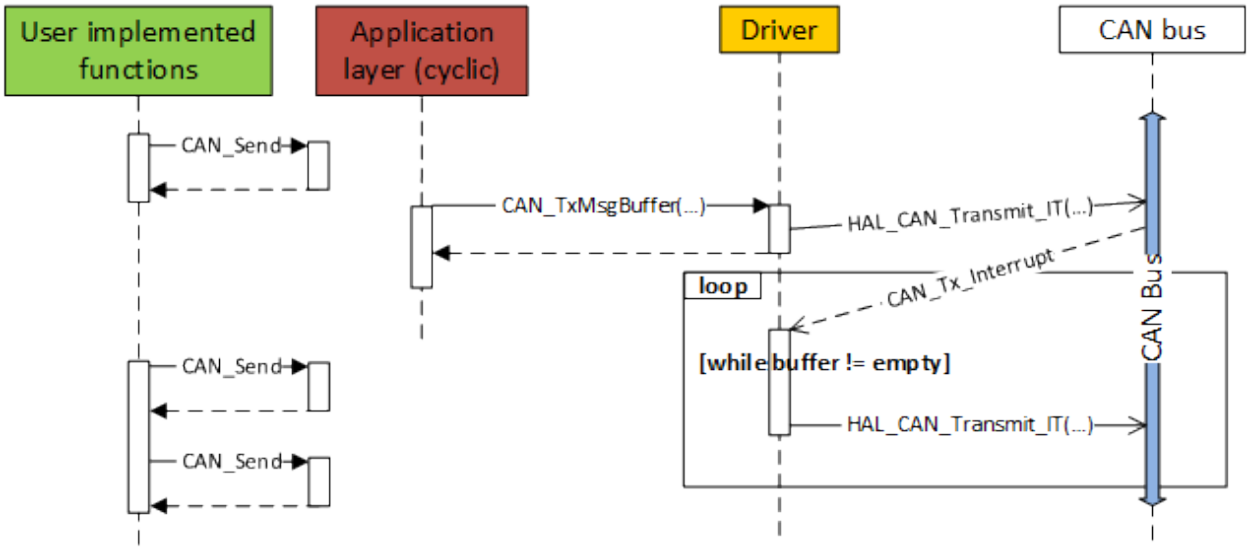


Fig. 28.20: CAN driver transmission sequence diagram with buffer usage

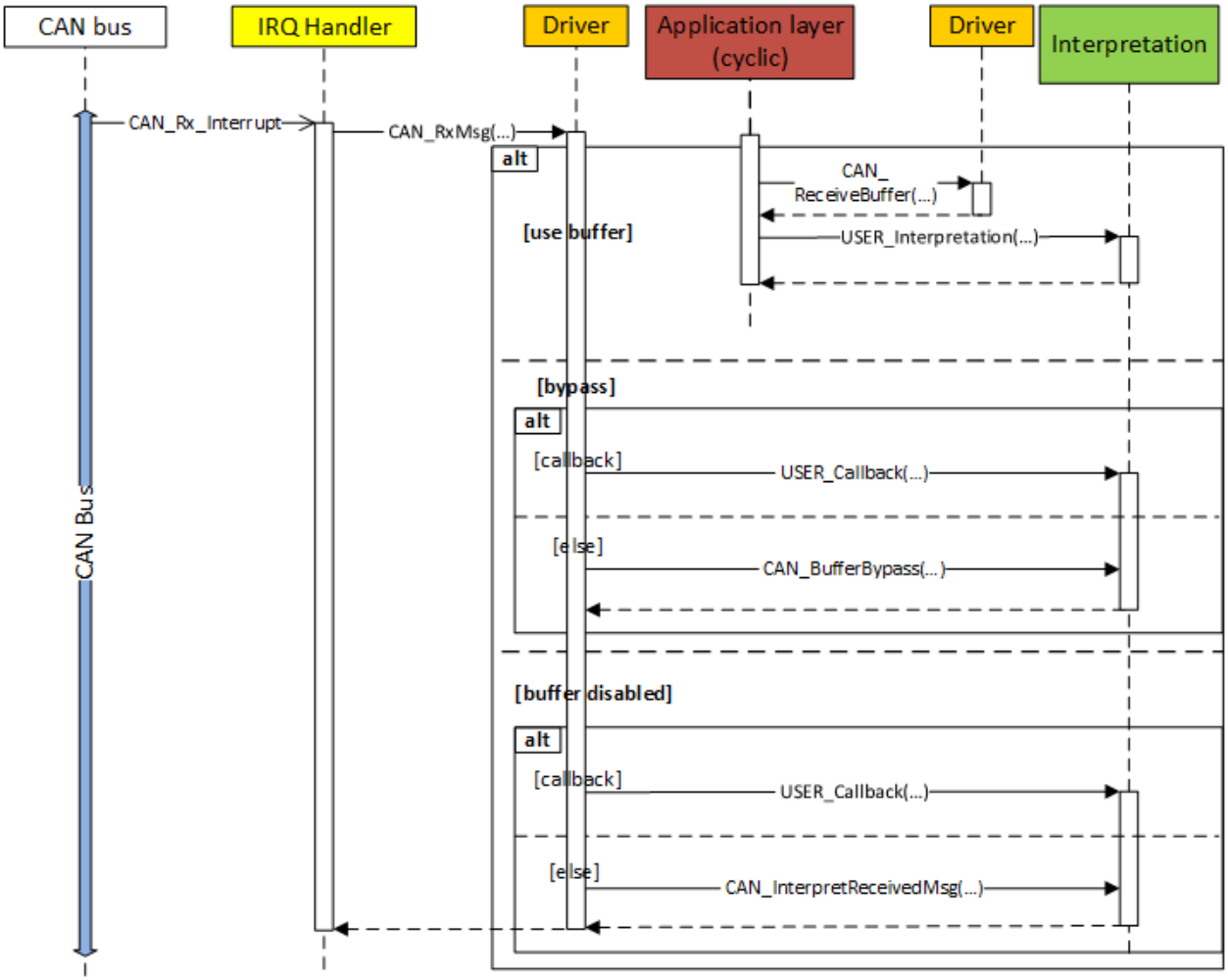


Fig. 28.21: CAN driver reception sequence diagram

28.15.5 References

28.16 Checksum

The chksum module is part of the foxBMS-Modules layer.

This section describes the embedded part of the checksum feature in foxBMS. For the buildprocess documentation on the checksum feature, see *Build Process*.

The chksum module provides the capability to verify data integrity during runtime for multiple purposes (e.g., firmware validation and verification of sent/received data).

28.16.1 Module Files

Source:

- embedded-software\mcu-common\src\driver\chksum\chksum.h (chksumh)
- embedded-software\mcu-common\src\driver\chksum\chksum.c (chksumc)

28.16.2 Detailed Description

The chksum module offers 2 possibilities to check data integrity, Modulo32BitAddition and a hardware based, slightly modified CRC32 implementation. The following short examples demonstrate the usage of the chksum module:

```
uint32_t chksum = CHK_crc32((uint8_t*)0x08000000, 0x1000) -> hashes 4kB of
code, starting at 0x08000000 using CRC32 algorithm

uint32_t chksum = CHK_modulo32addition((uint8_t*)0x08000000, 0x1000) ->
hashes 4kB of code, starting at 0x08000000 using Modulo32BitAddition
algorithm
```

During the startup, the CHK_crc32 function is used to verify the integrity of the flashed firmware image. After the calculation of the CRC32 checksum, it is compared to a hardcoded value within the flashheader struct. When both values match, the firmware is valid and execution continues. If both values do not match, an error is reported.

The flashheader contains the validation and checksum information with memory areas of separate software partitions, it gets generated during compilation by an external checksum tool, written in python. Furthermore it also provides the possibility for software versioning.

28.16.3 Checksum Configuration

Enabling/disabling of the checksum verification at startup is set by the following macro:

```
// deactivating the chechsum
#define BUILD_MODULE_ENABLE_FLASHCHECKSUM           0

// activating the chechsum
#define BUILD_MODULE_ENABLE_FLASHCHECKSUM           1
```

28.16.4 Checksum Default Configuration

The default value enables checksum verification at startup.

28.17 IO

This section describes the configuration of the pins of the foxBMS microcontrollers on the BMS-Master Board using the `io` module. The `io` module allows an easy way to configure all pins of the microcontroller at a central position, reading the input signals and writing signals to the output pins of the microcontroller.

28.17.1 Module Files

Driver:

- `embedded-software\mcu-common\src\driver\io\io.c` (ioc)
- `embedded-software\mcu-common\src\driver\io\io.h` (ioc)

Driver Configuration:

- `embedded-software\mcu-primary\src\driver\config\io_cfg.c` (primaryiocfgc)
- `embedded-software\mcu-primary\src\driver\config\io_cfg.h` (primaryiocfgh)
- `embedded-software\mcu-primary\src\driver\config\io_mcu_cfg.h` (primaryio_mcu_cfgh)
- `embedded-software\mcu-primary\src\driver\config\io_package_cfg.h` (primaryio_package_cfgh)
- `embedded-software\mcu-secondary\src\driver\config\io_cfg.c` (secondaryiocfgc)
- `embedded-software\mcu-secondary\src\driver\config\io_cfg.h` (secondaryiocfgh)
- `embedded-software\mcu-secondary\src\driver\config\io_mcu_cfg.h` (secondaryio_mcu_cfgh)
- `embedded-software\mcu-secondary\src\driver\config\io_package_cfg.h` (secondaryio_package_cfgh)

28.17.2 Configuration of the GPIOs

Initialization of the GPIOs

The following example shows how to configure the pins of the microcontrollers. The starting point is the following pin configuration:

Package Pin	Port-Pin	Signal name	Alternative Function
...			
16	F-0	PIN_MCU0_FMC_RAM_A0	flexible memory controller (FMC)
17	F-1	PIN_MCU0_FMC_RAM_A1	flexible memory controller (FMC)
...			
128	D-13	PIN_MCU0_ISO_GPIO_IN1	Digital Input
133	G-9	PIN_MCU0_ISO_GPIO_OUT0	Digital Output
...			

To define a signal, a `#define` has to be set in `io_cfg_foxbms.h` describing the signal:

```
...
#define PIN_MCU0_FMC_RAM_A0                 IO_PF0
#define PIN_MCU0_FMC_RAM_A1                 IO_PF1
...
#define PIN_MCU0_ISO_GPIO_IN1               IO_PD13
#define PIN_MCU0_ISO_GPIO_OUT0              IO_PG9
...
```

The initialization of this configuration on the hardware is executed through the `io` module function `IO_Init(*io_cfg)`. The `const IO_PIN_CFG_s io_cfg[]` has to be configured in `io_cfg.h`. In the given example, this looks like:

```
IO_PIN_CFG_s io_cfg[] = {
{PIN_MCU0_FMC_RAM_A0,       IO_MODE_AF_PP,        IO_PIN_NOPULL,    IO_SPEED_HIGH,    ␣
↪IO_ALTERNATE_AF12_FMC,         IO_PIN_LOCK_ENABLE},
{PIN_MCU0_FMC_RAM_A1,       IO_MODE_AF_PP,        IO_PIN_NOPULL,    IO_SPEED_HIGH,    ␣
↪IO_ALTERNATE_AF12_FMC,         IO_PIN_LOCK_ENABLE},
...
{PIN_MCU0_ISO_GPIO_IN1,    IO_MODE_INPUT,        IO_PIN_PULLDOWN,  IO_SPEED_FAST,    IO_
↪ALTERNATE_NO_ALTERNATE,     IO_PIN_LOCK_ENABLE},
{PIN_MCU0_ISO_GPIO_OUT0,   IO_MODE_OUTPUT_PP,    0,                0,                IO_
↪ALTERNATE_NO_ALTERNATE,     IO_PIN_LOCK_ENABLE},
}
```

The configuration options of each pin are documented in[1]. The naming conventions of foxBMS for setting the pin (alternate) function is found in `io_cfg.h` at `IO_PIN_ALTERNATE_e`. The clocks of the ports are enabled automatically when the pins are initialized through `IO_Init(&io_cfg)` as this function calls `IO_ClkInit(void)`. In order to prevent other modules and functions to change the configuration of the pins, the macro `IO_PIN_LOCKING` in `io_cfg.h` has to be defined. This macro automatically calls `IO_LockPin(pin)` for every pin which has defined `IO_PIN_LOCK_ENABLE` in `io_cfg[]` and locks the configuration registers of the corresponding pin.

Reading and Writing the Pins

- Reading the digital input of `PIN_MCU0_ISO_GPIO_IN1`

```
...
IO_PIN_STATE_e pinstate;
pinstate = IO_ReadPin(PIN_MCU0_ISO_GPIO_IN1);
...
```

- Writing the digital output of `PIN_MCU0_ISO_GPIO_OUT0`

```
...
IO_PIN_STATE_e pinstate = IO_PIN_SET;
IO_WritePin(PIN_MCU0_ISO_GPIO_OUT0, pinstate);
...
```

Sources:

[1] Datasheet DM00071990 [PDF] http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00071990.pdf

28.18 NVM

The `nvm` module provides a non-volatile memory for the data by using an external EEPROM with integrated transparent hardware error correcting code (ECC). It handles the backup and recovery features between the participating submodules and depends on the implementation (EEPROM, BKSSRAM or SRAM). The user has to group the data in channels which are taped with a checksum and handled as one entity. Usually, the data are first written into the SRAM (or BKPSRAM if available) and then backed up into the EEPROM. Alternatively, the data can be written directly into the EEPROM without a buffer. When a buffer is used, it can be located in SRAM or in BKPSRAM (Backup SRAM) with following different reset characteristics:

- **BKPSRAM:** no data loss in case of any kind of reset (button cell buffered SRAM)
- **SRAM (cleared memory section):** data loss in case of any kind of reset
- **SRAM (non-cleared memory section):** data loss only in case of power down, no data loss in case of software reset or warm startup

28.18.1 Module Files

Driver:

- `src\module\nvram\eepr.h`
- `src\module\nvram\eepr.c`
- `src\module\rtc\bkpsram.h`
- `src\module\rtc\bkpsram.c`

Driver Configuration:

- `src\module\config\eepr_cfg.h`
- `src\module\config\eepr_cfg.c`
- `src\module\config\bkpsram_cfg.h`
- `src\module\config\bkpsram_cfg.c`

28.18.2 Data Management

Data Structure

The data storage of the EEPROM is organized in coherent data entities called channels. Each channel has its own storage area reserved in the EEPROM. EEPROM channels can be used as a non-volatile backup for corresponding BKPSRAM channels. Furthermore, every channel has a checksum, an ID, and optionally a software write protection and a pointer to the location of the corresponding channel in the SRAM or BKPSRAM. Additionally, every channel has a dirty flag, which is set to indicate a modification of data to initiate a backup process to the EEPROM. EEPROM channels can be configured to be updated automatically during startup, if data is not up-to-date or in case it is corrupted.

Example: EEPROM-channels A, B and C have coherent BKPSRAM channels. E has a coherent SRAM channel. D does not have a coherent channel: the data are written directly into this channel.

Statemachine of the EEPROM Driver

The statemachine is capable of handling the requests to write, read or to go into an idle mode. The states of the statemachine are:

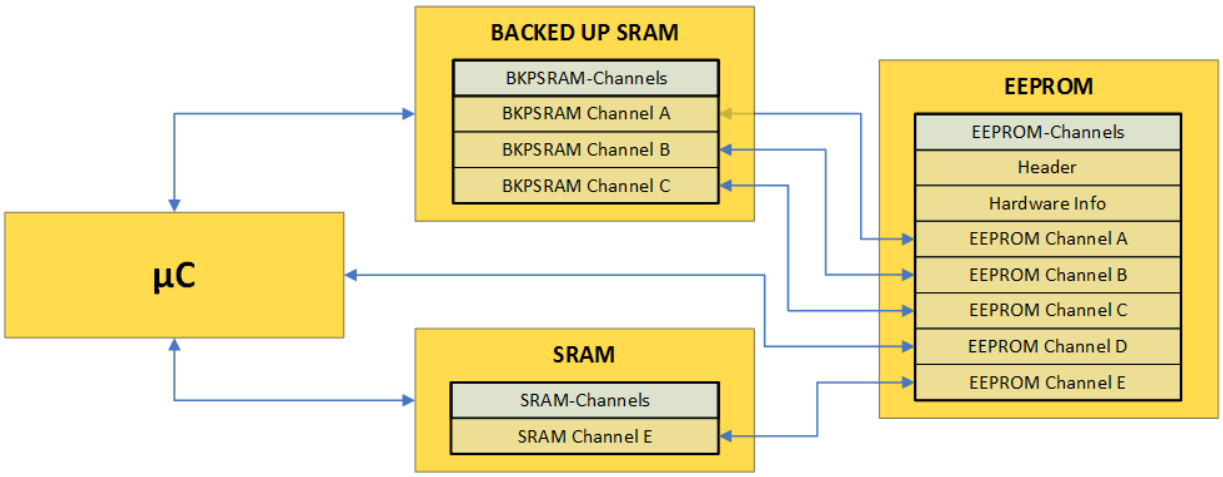


Fig. 28.22: Data organisation of the NV-Memory

State	Description
DISABLED	Initial state at startup
UNINITIALIZED	First state which will be entered after startup: the initialization sequence is started here
INIT_IN_PROCESS	Checking of EEPROM availability and actual initialization
IDLE	Idle mode, ready to receive commands
WRITE_MEMORY	Start of write process
WRITE_ENABLE_INPROCESS	Activation and verification of the write mode of the EEPROM
WRITE_MEMORY_ENABLED	Page length and timers are set, actual write command is being sent
WRITE_MEMORY_INPROCESS	Verification of the write process
READ_MEMORY	Page length and timers are set, read command is being sent
READ_MEMORY_INPROCESS	Verification of the write process
CHECK_DATA	Checksum verification of the written/read data
EEPR_INITFAILED EEPR_READFAILED EEPR_WRITEFAILED	Handling of an initialization error Handling of read errors Handling of write errors

Startup

Before any data can be written or read, the EEPROM is being initialized by sending the request to go to idle. If the initialization was successful, the version number of the software and the EEPROM Header Information is checked, to ensure an equal memory layout software and NVM. Afterwards the data channels will be initialized.

Channel Initialization

During the initialization, the driver will check and update the data of the selected channels as shown in Fig. 28.24. The specific control mechanisms to identify corrupt data are:

(1) Hardware-Check of Brownout Reset (BOR)

(2) Software-Check of BKPSRAM checksum

(3) Software-Check of EEPROM checksum

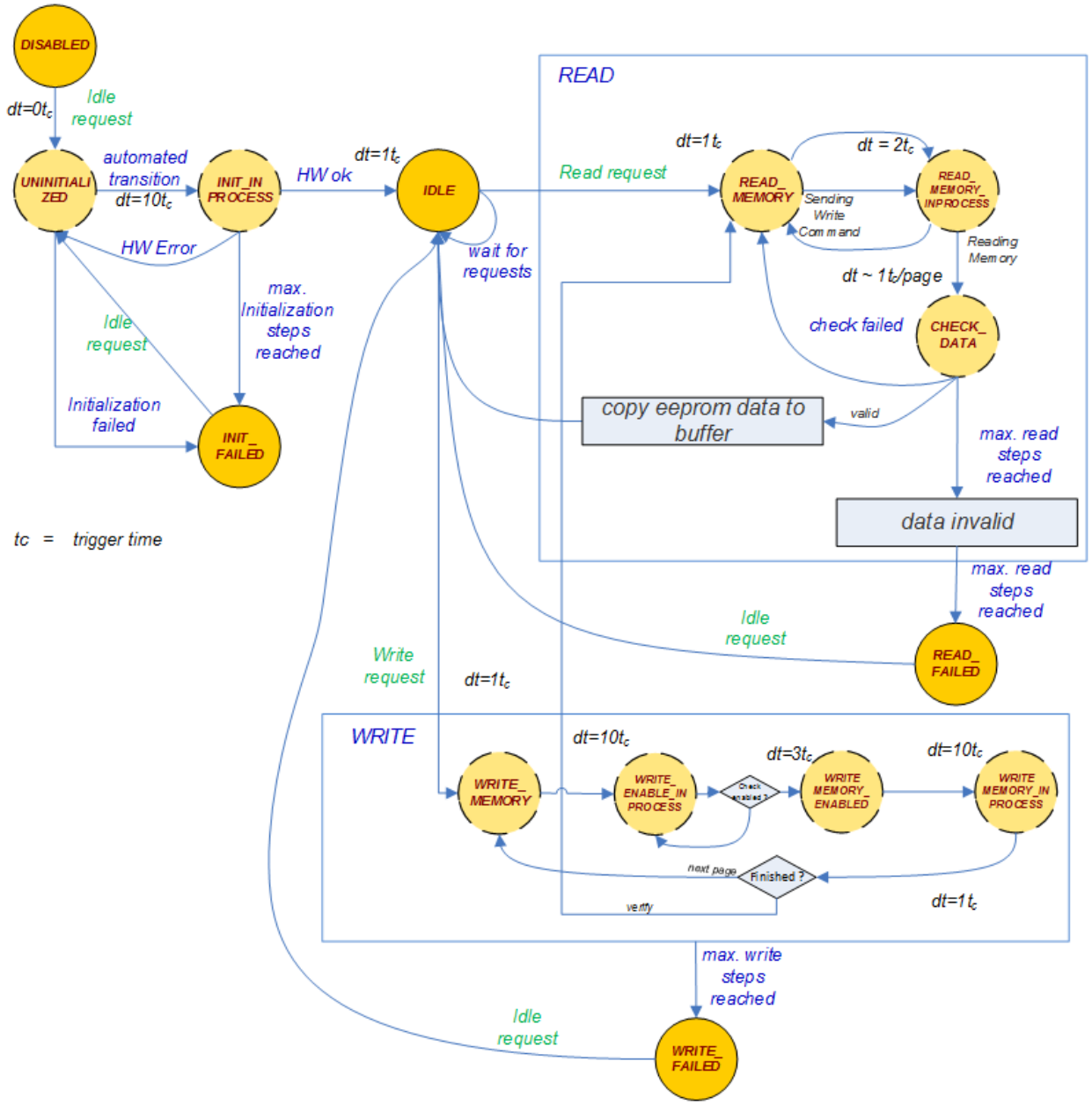


Fig. 28.23: Statemachine of the EEPROM

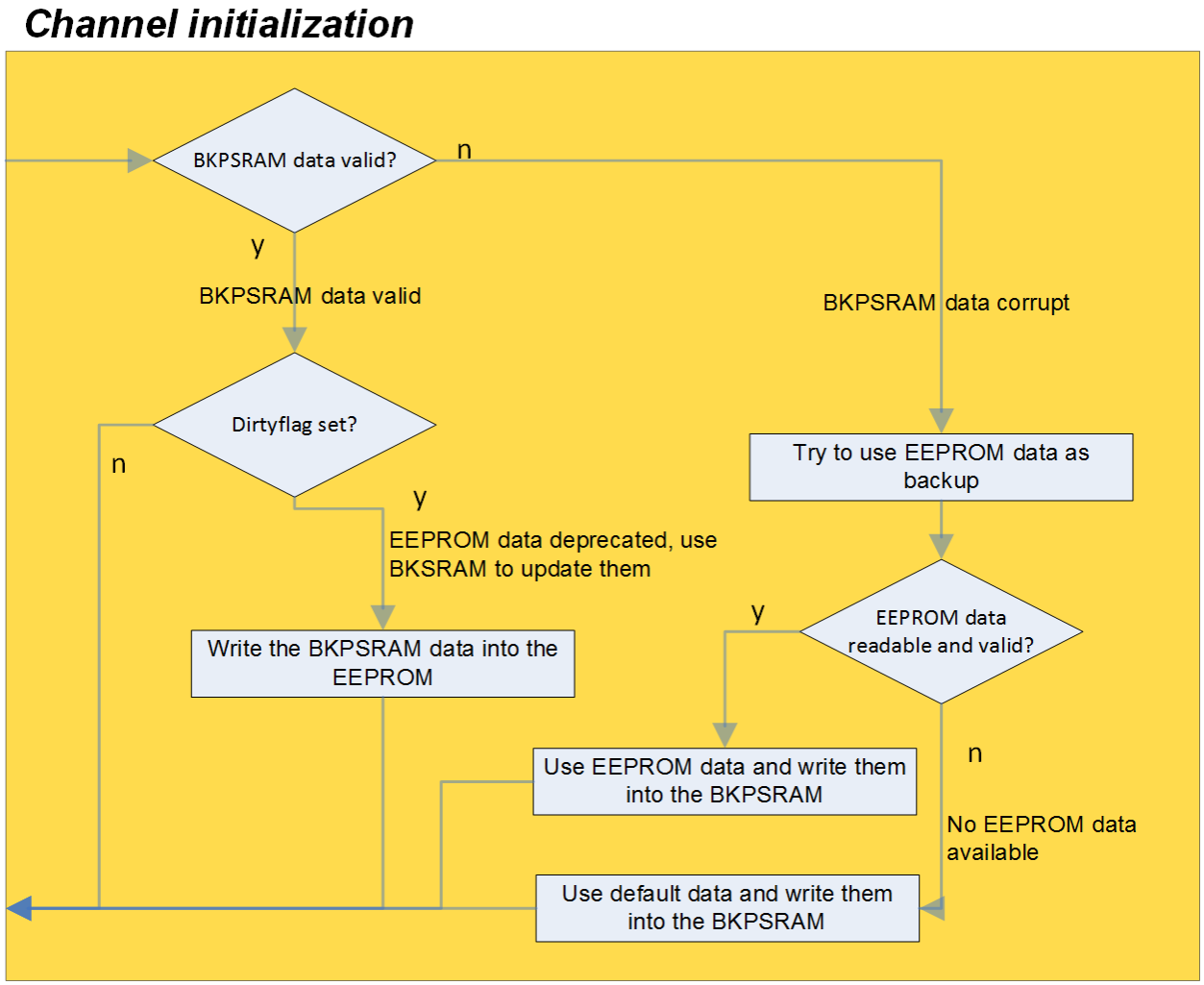


Fig. 28.24: Initialization of the NVM channels

28.18.3 NVM Configuration

Channel Configuration

Users can create new channels by declaring them in the array `eepr_ch_cfg[]`.

When creating channels, attention should be paid to the following aspects:

1. It is advisable to create a structure to work with the correct data format.
2. When creating the structure, users have to pay attention to the correct data alignment. Data should be stored in blocks of 4 byte (32 bit) to avoid errors when reading or writing data. Example: Do not store a 32 bit number directly behind a 16 bit number. Instead, insert a 16 bit dummy in between.
3. Make sure channels do not overlap.
4. The checksum must always be stored in the last 4 bytes of the channel.
5. Make sure that the channel ID matches the position in the array.
6. Setting the write protection is optional and should only be set if the channel data in the EEPROM does not have to be changed during the runtime.
7. Setting the pointer to the location of the corresponding channel (BKPSRAM or SRAM) is also optional, but necessary if an automatic update of the data should be done during the initialization.
8. To support the automatic update function, the user has to make two function calls in the function `EEPR_InitChannelData()`, just like the other channels that are handled there.
9. In the function `EEPR_SetDefaultValue(EEPR_CHANNEL_ID_TYPE_e eepr_channel)` the default values of the new channel must be set.
10. It is advisable to implement a GET and SET function for the new channel in the `bkpsram_cfg.c`, similar to the implemented examples, to ensure data consistency.

Write and Read during Runtime

To write or read an EEPROM channel during runtime, the user should use the functions `EEPR_SetChannelData(EEPR_CHANNEL_ID_TYPE_e eepr_channel, uint8_t *dest_ptr)` or `EEPR_GetChannelData(EEPR_CHANNEL_ID_TYPE_e eepr_channel, uint8_t *dest_ptr)` and trigger the state machine until it is either in the idle mode again or in an error state. To use the BKPSRAM pointer defined in the channel config, the source/destination pointer can be declared to be a NULL-pointer. To write or read the data of a BKPSRAM channel, the implemented GET and SET functions can be used.

28.18.4 Usage

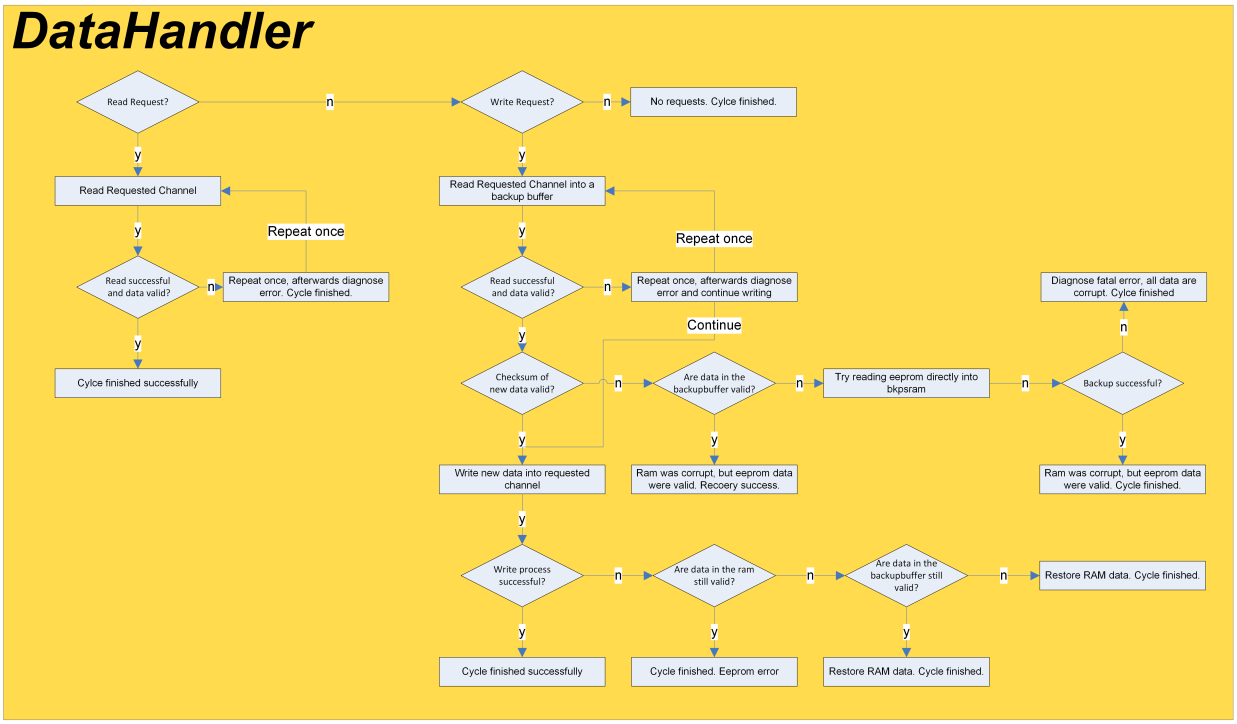
Initialization

The EEPROM can be initialized manually, by setting the request to go to IDLE and triggering the state machine until the desired state is reached. Alternatively, the EEPROM can be initialized automatically by calling the function `EEPR_ERRORTYPES_e EEPR_Init(void)`, which will initialize the EEPROM first and the channels afterwards. The BKPSRAM module can be initialized by calling `void BKP_SRAM_Init(void)`.

Usage during runtime

During runtime a datahandler will manage the write and read process of the channels. It continuously checks if there is a read request or a write request for every channel. The read/write requests can be set with `void`

`EEPR_SetChReadReqFlag(EEPR_CHANNEL_ID_TYPE_e eepr_channel)` for read requests and `void EEPR_SetChDirtyFlag(EEPR_CHANNEL_ID_TYPE_e eepr_channel)` for write requests. Afterwards the datahandler will take care of the write-process itself. How the write/read-process works can be seen in Fig. **??**.



After the write/read process is complete, the datahandler will diagnose the errors in `EEPR_DATAHANDLING_ERROR_e EEPR_ch_dataerrors`.

Other Functions

`void EEPR_Trigger(void)` triggers the statemachine of the EEPROM. `EEPR_RETURNTYPE_e EEPR_SetStateRequest(EEPR_STATE_e state_req, EEPR_CHANNEL_ID_TYPE_e channel, uint8_t* ramaddress)` is used to request the desired state and returns if the request was accepted. `EEPR_ERRORTYPES_e EEPR_Init(void)` initializes the EEPROM.

28.18.5 Sources

-

28.19 UART

The `uart` module is part of the `Driver` layer.

The `uart` module provides the capability to send and receive data using the RS232/UART interface.

28.19.1 Module Files

Driver:

- `embedded-software\mcu-common\src\driver\uart\uart.c` (uartc)
- `embedded-software\mcu-common\src\driver\uart\uart.h` (uarth)

Driver Configuration:

- `embedded-software\mcu-primary\src\driver\config\uart_cfg.c` (primaryuartcfgc)
- `embedded-software\mcu-primary\src\driver\config\uart_cfg.h` (primaryuartcfgh)

28.19.2 Detailed Description

The `uart` module uses a user-defined buffer for transmitting data. A custom IRQ handler is responsible for handling all receive/transmit processes. Those operations are dispatched to sub-functions according to the set flags in the status register.

28.19.3 UART Configuration

The default configuration is done within the `UART_Init()` function, which gets called during startup. Below is an example showing the UART configuration which uses usart3 with a baud rate of 115200Bd:

```
UART_HandleTypeDef uart_cfg[UART_NUMBER_OF_USED_UART_CHANNELS] = {
       {
            .Instance = USART3,
            .Init.BaudRate = 115200,
            .Init.WordLength = UART_WORDLENGTH_8B,
            .Init.StopBits = UART_STOPBITS_1,
            .Init.Parity = UART_PARITY_NONE,
            .Init.Mode = UART_MODE_TX_RX,
            .Init.HwFlowCtl = UART_HWCONTROL_NONE,
            .Init.OverSampling = UART_OVERSAMPLING_16,
       }
};
```

28.19.4 Usage

The initialization has to be done during startup for subsequent function calls to succeed. `UART_IntRx` and `UART_IntTx` are interrupt driven and do not need to be called by the user. Assuming initialization was done properly, the user only needs to copy content from the internal ring buffer for reading purpose, or call `UART_vWrite` for writing purpose.

CHAPTER 29

Software Tools

This section contains a documentation of the tools used with foxBMS.

29.1 Flashtool

The flashtool is located in `/path/to/foxbms/tools/flashtool`.

With the command line tool, the flash memory of the foxBMS (or generally STM32F4) microcontroller can be programmed with a binary file. Also a dump read or a full erase of the flash memory can be done.

Note: Make sure that CAN0 connector is either disconnected or that there is no traffic on CAN0 bus while flashing `primary` MCU. If this is not the case, the internal bootloader of the controller will select a wrong boot source and the flashing fails.

29.1.1 Module Files

Source:

- `tools/flashtool/__init__.py`
- `tools/flashtool/detect.py`
- `tools/flashtool/flash.py`
- `tools/flashtool/flashconfig.yaml`
- `tools/flashtool/foxflasher.py`
- `tools/flashtool/mcuconfig.ini`
- `tools/flashtool/README.md`
- `tools/flashtool/stm32flasher.py`

- tools/flashtool/stm32interface.py

29.1.2 Flash Memory Areas

Program Flash: * starting at 0x08000000 * variable size * contains the whole foxbms firmware itself * binary flash file from build process: foxbms_flash.bin

Program Header Flash * starting at 0x080FFF00 * 256 Byte size * contains flash area checksum and serial numbers etc. * binary flash file from build process: foxbms_flashheader.bin

Caution: when flash programming, do not erase for each flash area, because the full flash memory is erased and everything that was written before, is erased too!

29.1.3 Procedure

1. Connect the USB slot of the primary or secondary microcontroller to the computer.
2. Check to see which serial interface it is connected (Windows: COMxx, Linux: /dev/ttyUSBx)
3. The tool must be called from the commandline:

```
usage: foxflasher.py [-h] [-v] [--erase] [--read] [--write] [--verify]
                     [--bytes BYTES] [--bauds BAUDS] [--port PORT]
                     [--address ADDRESS] [--goaddress GOADDRESS]
                     FIRMWARE FILE

foxBMS---STM32 flash tool

positional arguments:
FIRMWARE FILE         firmware binary

optional arguments:
  -h, --help            show this help message and exit
  -v, --verbosity       increase output verbosity
  --erase, -e           erase firmware
  --read, -r            read and store firmware
  --write, -w           writes firmware
  --verify, -y          verify the firmware
  --bytes BYTES, -s BYTES
                        bytes to read from the firmware
  --bauds BAUDS, -b BAUDS
                        transfer speed (bauds)
  --port PORT, -p PORT  ttyUSB port
  --address ADDRESS, -a ADDRESS
                        target address
  --goaddress GOADDRESS, -g GOADDRESS
                        start address (use -1 for default)

Example:
foxflasher.py --port COM3 --erase --write --verify build/src/general/foxbms_flash.bin

Copyright (c) 2015, 2016 Fraunhofer IISB.
All rights reserved.
This program has been released under the conditions of the BSD 3-Clause License.
```

Commandline examples:

- read 512 bytes from start address 0x08000000 to file asdf.bin when windows serial interface COM13 is connected to foxBMS `python foxflasher.py -r -s 512 -p COM13 asdf.bin`
- erase flash, write and verify content of file asdf.bin at start address 0x08000100 to ~~when~~ when windows serial interface COM13 is connected to foxBMS `python foxflasher.py -e -w -v -a 0x08000100 -p COM13 asdf.bin`

29.1.4 For Developers

Class Diagram

STM32Interface: - serial interface abstraction of foxBMS interface to send according to the protocol specified by ST Microelectronics

- for detailed insight to the USART protocol refer to STM32 appnote AN3155[1]
- for detailed insight to the device bootloader refer to STM32 appnote AN2606[2]

STM32flasher: - easy-to-use flash tool for STM32f4 microcontrollers (similar to popular ~~other~~ other flashtools like avrdude)

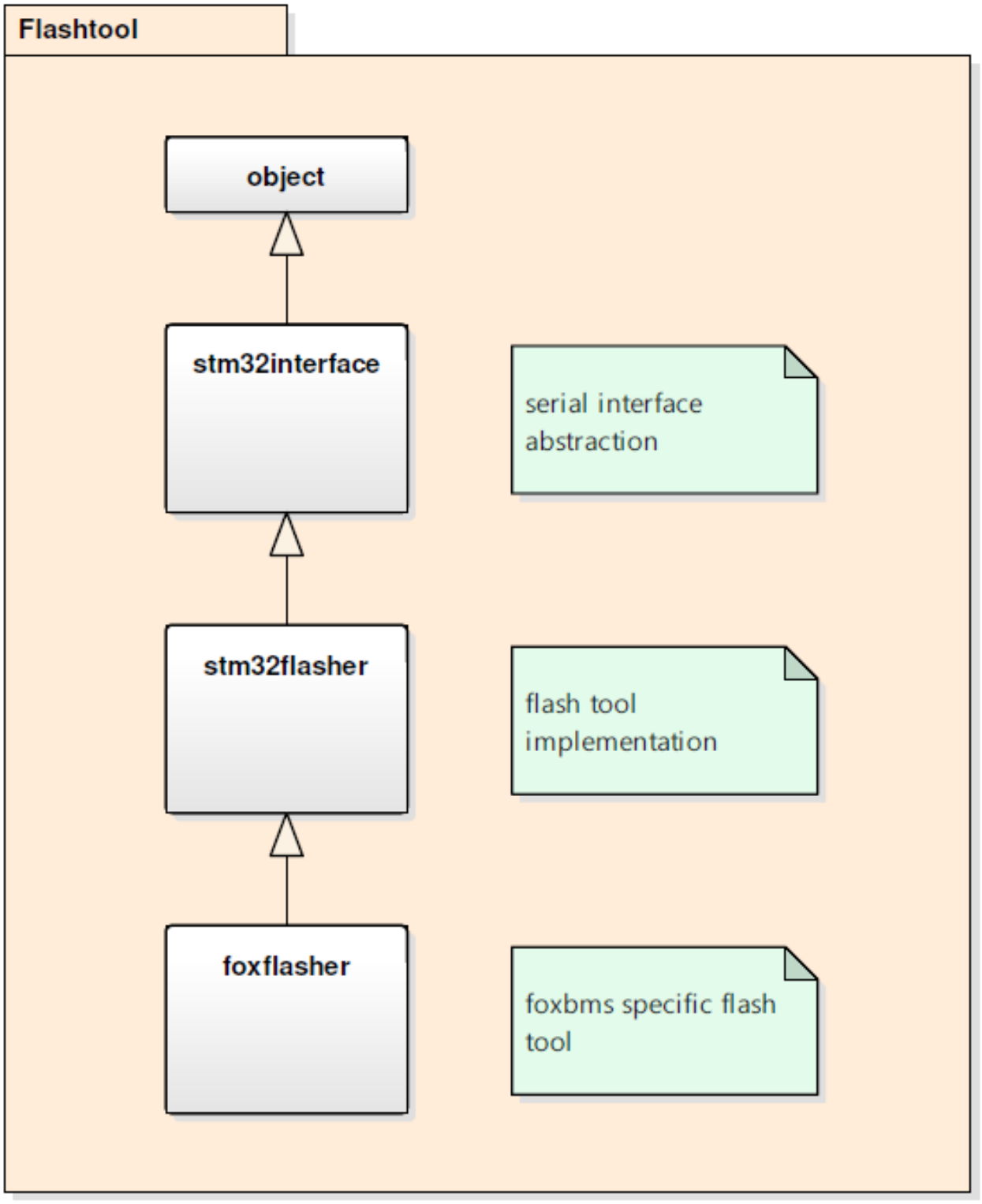
FoxFlasher: - specific implementation of the STM32F4 microcontroller flashtool, which has following specialisations:

- sets bootpin and resets microcontroller by the use of the serial interface DTR and RTS pins after the serial port is opened
- TODO: read out CTS and DSR serial interface input pins to distinguish between the primary and secondary microcontroller

29.1.5 References

[1] ST Microelectronics Application Note 3155 [PDF], 'USART protocol used in the STM32 bootloader' http://www.st.com/content/ccc/resource/technical/document/application_note/51/5f/03/1e/bd/9b/45/be/CD00264342.pdf

[2] ST Microelectronics Application Note 2606 [PDF], 'STM32 microcontroller system memory boot mode' http://www.st.com/content/ccc/resource/technical/document/application_note/b9/9b/16/3a/12/1e/40/0c/CD00167594.pdf


Flashtool
object
stm32interface
serial interface abstraction
stm32flasher
flash tool implementation
foxflasher
foxbms specific flash tool

CHAPTER 30

Build Process

foxBMS uses `waf The meta build system` for building binaries and the documentation.

For detailed information on `waf` see waf.io. A short introcution to `waf` is given at waf.io/apidocs/tutorial. The more detailed version of how to use `waf` is found at waf.io/book.

30.1 General

30.1.1 Where to find the toolchain?

The `waf` toolchain is located in the directory `foxbms\tools`, in the binary `waf`. This archive is automatically unpacked in a directory named something like `waf-{{X}-{{some-hash-value}}` containing the waf library, where `{{X}}` is the dash-separated version number of `waf`. It is unpacked into `foxbms\tools`. It is generally assumed that all commands are run from directory `foxbms`. Therefore `waf` has to be always called by `python tools\waf some-command` where `some-command` is an argument defined in the `wscript`.

Additional build tools are located in `foxbms\tools\waftools`. These are the tools needed for building the documentation, i.e., `doxygen` and `sphinx`.

30.1.2 Where are the build steps described?

The build process is described in files named `wscript`, that can be found nearly everywhere inside the different directories inside the foxBMS project. Later in this documentation this is explained in detail.

30.1.3 General

`waf` needs always to be run form the top level of the repository and the path to waf must be given relative to that directory. This path is `tools\waf`.

Listing 30.1: How to call waf

```
cd path\to\foxbms
python tools\waf {{some-command}}
```

30.1.4 What commands can be used?

To get an overview of support ed commands run `--help` or `-h` on the `waf` binary:

Listing 30.2: How to call help on waf

```
python tools\waf --help
```

This is the output in foxBMS version `1.6.3`

Listing 30.3: Waf help in foxBMS

```
waf [commands] [options]

Main commands (example: ./waf build -j4)
  build                 : executes the build  build what?
  build_all             : builds all parts of the project (binaries and documentation)
  build_libs            : executes the build of libs
  build_primary         : executes the build of primary
  build_primary_bare    : executes the build of primary_bare
  build_secondary       : executes the build of secondary
  build_secondary_bare  : executes the build of secondary_bare
  clean                 : cleans the project
  clean_all             : cleans all parts of the project
  clean_libs            : cleans the project libs
  clean_primary         : cleans the project primary
  clean_primary_bare    : cleans the project primary_bare
  clean_secondary       : cleans the project secondary
  clean_secondary_bare  : cleans the project secondary_bare
  configure             : configures the project
  cpplint               : configures cpplint
  dist                  : creates an archive containing the project source code
  distcheck             : creates tar.bz form the source directory and tries to run a␣
↪build
  distclean             : removes build folders and data
  doxygen               : creates doxygen documentation
  doxygen_libs          : creates the doxygen documentation of libs
  doxygen_primary       : creates the doxygen documentation of primary
  doxygen_primary_bare  : creates the doxygen documentation of primary_bare
  doxygen_secondary     : creates the doxygen documentation of secondary
  doxygen_secondary_bare: creates the doxygen documentation of secondary_bare
  flake8                : runs flake8 on the foxBMS repository
  list                  : lists the targets to execute
  list_libs             : lists the targets to execute for libs
  list_primary          : lists the targets to execute for primary
  list_primary_bare     : lists the targets to execute for primary_bare
  list_secondary        : lists the targets to execute for secondary
  list_secondary_bare   : lists the targets to execute for secondary_bare
  sphinx                : creates the sphinx documentation of the project
  step                  : executes tasks in a step-by-step fashion, for debugging
```

(continues on next page)

(continued from previous page)

```
  step_libs             : executes tasks in a step-by-step fashion, for debugging of␣
↪libs
  step_primary          : executes tasks in a step-by-step fashion, for debugging of␣
↪primary
  step_primary_bare     : executes tasks in a step-by-step fashion, for debugging of␣
↪primary_bare
  step_secondary        : executes tasks in a step-by-step fashion, for debugging of␣
↪secondary
  step_secondary_bare   : executes tasks in a step-by-step fashion, for debugging of␣
↪secondary_bare

Options:
  --version             show program's version number and exit
  -c COLORS, --color=COLORS
                        whether to use colors (yes/no/auto) [default: auto]
  -j JOBS, --jobs=JOBS  amount of parallel jobs (8)
  -k, --keep            continue despite errors (-kk to try harder)
  -v, --verbose         verbosity level -v -vv or -vvv [default: 0]
  --zones=ZONES         debugging zones (task_gen, deps, tasks, etc)
  -h, --help            show this help message and exit
  --cpplint-conf-file=CPPLINT_CONF
                        cpplint configuration file (default: cpplint.yml)
  -t TARGET, --target=TARGET
                        build target: debug (default)/release
  -l LIBS, --libs=LIBS  name of the library to be used

  Configuration options:

  Build and installation options:
    -p, --progress      -p: progress bar; -pp: ide output

  Step options:
    --files=FILES       files to process, by regexp, e.g. "*/main.c,*/test/main.o"

  Installation and uninstallation options:
    --distcheck-args=ARGS
                        arguments to pass to distcheck
```

30.1.5 The `configure` command

Before building any binaries or documentation is possible, the project needs to be configured. A successfull `configure` command and its ouput is shown below:

Listing 30.4: Configuration of a the project

```
python tools\waf configure
(...)
'configure' finished successfully (0.340s)
```

30.1.6 The `build` commands

After the project has been configured, a build can be triggered and it is generally exectued by the `build` commands. As foxBMS requries building variants, one has to use e.g., `build_primary` in order to build binaries for the primary MCU.

Listing 30.5: Example of a wrong and a correct build command.

```
python tools\waf build
Waf: Entering directory `.\foxbms\build'
A build variant must be specified, run 'python tools\waf --help'
python tools\waf build_primary
(...)
'build_primary' finished successfully (8.800s)
```

The possible `build` commands, the definition of the and corresponding targets and the targets itself is listet below:

- **Primary MCU**

  Listing 30.6: Build primary binaries

  ```
  python tools\waf build_primary
  ```

  The targets are defined at:

  - `foxbms\embedded-software\mcu-primary\src\application\wscript`
  - `foxbms\embedded-software\mcu-common\src\engine\wscript`
  - `foxbms\embedded-software\mcu-common\src\module\wscript`
  - `foxbms\embedded-software\mcu-primary\src\engine\wscript`
  - `foxbms\embedded-software\mcu-primary\src\module\wscript`
  - `foxbms\embedded-software\mcu-freertos\wscript`
  - `foxbms\embedded-software\mcu-hal\STM32F4xx_HAL_Driver\wscript`
  - `foxbms\embedded-software\mcu-primary\src\general\wscript`

  Listing 30.7: Primary targets

  ```
  foxbms-application
  foxbms-common-driver
  foxbms-common-engine
  foxbms-common-module
  foxbms-common-util
  foxbms-driver
  foxbms-engine
  foxbms-module
  foxbms-os
  foxbms-stmhal
  foxbms_primary.elf
  'list_primary' finished successfully (0.052s)
  ```

- **Secondary MCU**

  Listing 30.8: Build secondary binaries

  ```
  python tools\waf build_secondary
  ```

  The targets are defined at:

  - `foxbms\embedded-software\mcu-secondary\src\application\wscript`
  - `foxbms\embedded-software\mcu-common\src\engine\wscript`

  - `foxbms\embedded-software\mcu-common\src\module\wscript`
  - `foxbms\embedded-software\mcu-secondary\src\engine\wscript`
  - `foxbms\embedded-software\mcu-secondary\src\module\wscript`
  - `foxbms\embedded-software\mcu-freertos\wscript`
  - `foxbms\embedded-software\mcu-hal\STM32F4xx_HAL_Driver\wscript`
  - `foxbms\embedded-software\mcu-secondary\src\general\wscript`

Listing 30.9: Secondary targets

```
foxbms-application
foxbms-common-driver
foxbms-common-engine
foxbms-common-module
foxbms-common-util
foxbms-engine
foxbms-os
foxbms-stmhal
foxbms_secondary.elf
'list_secondary' finished successfully (0.100s)
```

- **General documentation** The general documenation is build by

Listing 30.10: Build the general foxBMS documentation

```
python tools\waf sphinx
```

- **API documentation** The API documentation is build using the `doxygen_{{variant}}`, therefore

Listing 30.11: Build the general foxBMS documentation

```
python tools\waf doxygen_primary
python tools\waf doxygen_secondary
```

- **Cleaning** It is also possible to clean the binaries and Doxygen documentation. This step is performed by the `clean` command.

  As seen from `--help` the possible `clean` commands are

  - `clean_all`
  - `clean_libs`
  - `clean_primary`
  - `clean_primary_bare`
  - `clean_secondary`
  - `clean_secondary_bare`
  - `distclean`

  Each command cleans the specified option, except for `distclean`. However it is possible to make a complete clean by `distclean`. After `distclean` the entire build directory and all lock files etc. are deleted and the project needs to be configured again. Cleaning the general sphinx documentation alone is currently not supported, but it can be achivied by running `distclean`.

30.2 Targets

As stated above the targets and sub targets of the build process are shown by `list_x` where `x` is the specified target. The main target is the `*.elf.unpatched` file. The final targets are build afterwards. After successfully linking the map file is generated.

These logging files are found in `build` and `build\{{target}}`. Additional to the `*.elf.unpatched` and `*.elf` files a `*.hex` and two `*.bin` files of the binary are generated. The `*.bin` files are separated into the flash and the flashheader. The size of each object/binary is written to a log file.

The targets are build as follows (final targets are filled gray):

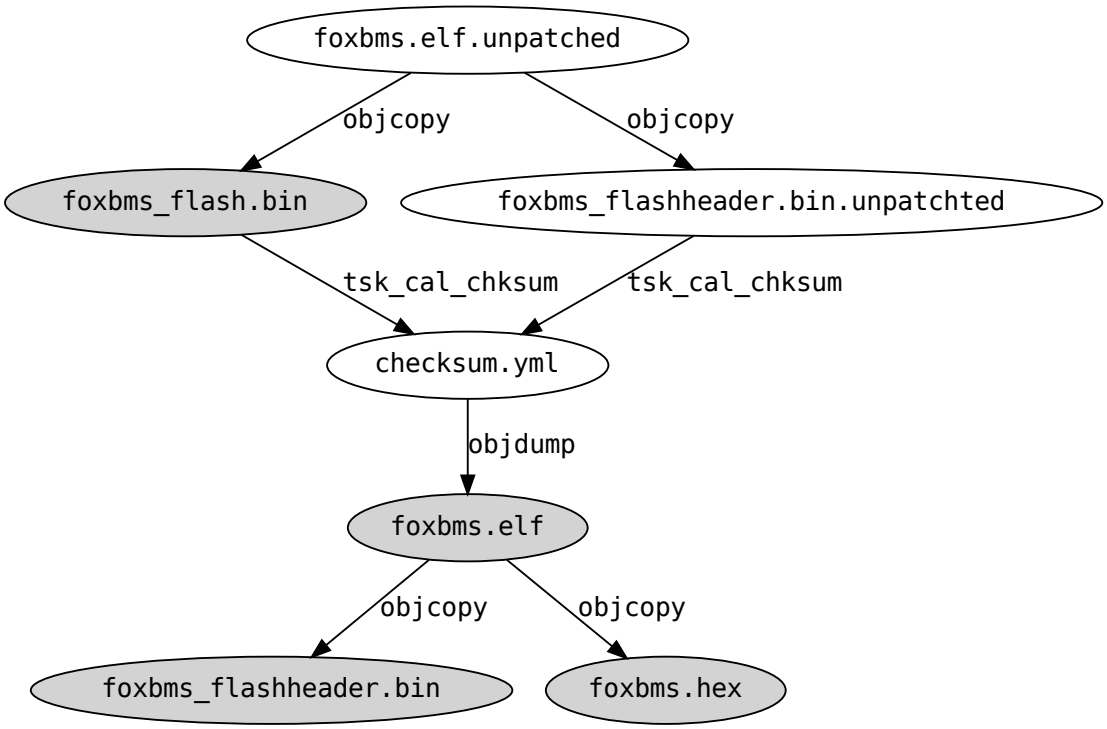


Fig. 30.1: foxBMS build process targets

30.3 Build Process

Note: For testing the following explanations it is assumed that `python tools\waf configure` has been run.

This sections gives an overview how the build process is defined. All features are generally defined in the top `wscript` located at `foxbms\wscript`.

The minimum functions that are needed to be defined a build in `waf` are:

- `configure` and

- `build`.

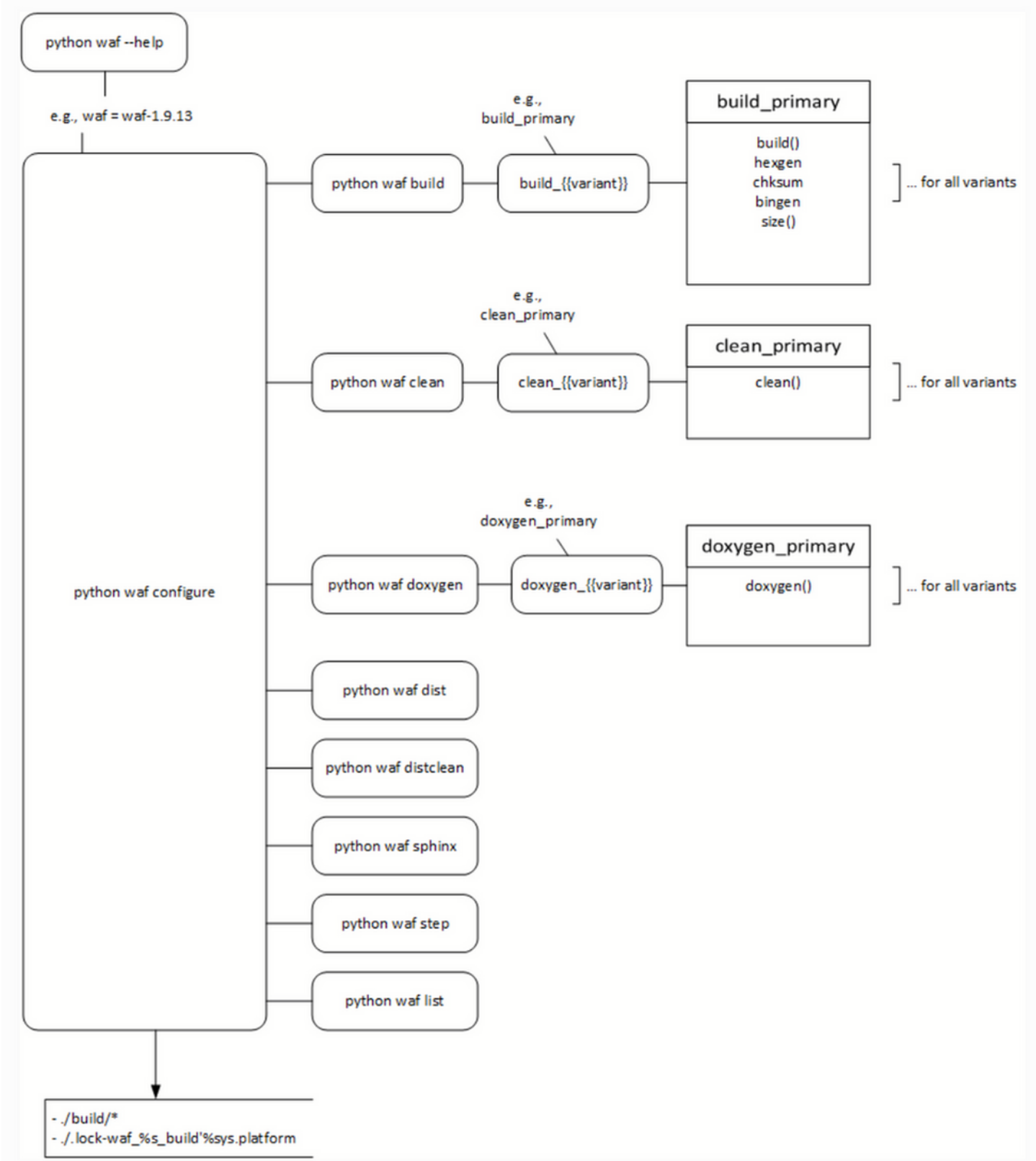
As the toolchain needs more targets the following functions need to be implemented: `doxygen` and `sphinx`.

Furthermore the following features are needed:

- for calculating the checksum based on the `*.elf.unpatched` file the class `tsk_cal_chksum` and for creating the `*.elf` file the `tsk_wrt_chksum` class and for adding these features the function `add_chksum_task`,
- for stripping the debug symbols in release mode the class `strip` and the function `add_strip_task`,
- for creating a `hex` file from the `elf` file the class `hexgen` and the function `add_hexgen_task`,
- for generating `bin` files from `elf` files the classes `tsk_binflashheadergen`, `tsk_binflashgen` and the function `add_bingen_task` and the class `tsk_binflashheaderpatch` and the function `add_patch_bin_task`,
- for generating size information of the objects and binaries the class `size` and the function `process_sizes`,
- for copying the libraries build by `build_libs` into the correct directories the class `copy_libs` and the function `add_copy_libs`,
- for compiling assembler files `*.s` the class `Sasm` and the function `asm_hook`.

For implementation details see the `wscript` itself.

Some of these functionalities require scripts from `foxbms\tools`.


python waf --help
e.g., waf = waf-1.9.13
python waf configure
python waf build
e.g., build_primary
build_{{variant}}
build_primary
build()
hexgen
chksum
bingen
size()
... for all variants
python waf clean
e.g., clean_primary
clean_{{variant}}
clean_primary
clean()
... for all variants
python waf doxygen
e.g., doxygen_primary
doxygen_{{variant}}
doxygen_primary
doxygen()
... for all variants
python waf dist
python waf distclean
python waf sphinx
python waf step
python waf list
- ./build/*
- ./.lock-waf_%s_build'%sys.platform

30.3.1 General Documentation

This build target uses the function `def sphinx(bld)`. Since this definition of a function called `sphinx`, it is accepted as command to waf.

This general documentation is generated by running

Listing 30.12: Generate general foxBMS documentation

```
python tools\waf sphinx
```

The implementation details of the `sphinx` command can be found in `foxbms\tools\waftools\sphinx_build.py`.

30.3.2 Primary and Secondary Binaries and Doxygen Documentation

In order to have different build *variants*, these variants have to be defined. This is done at the top of the main `wscript` at `foxbms\wscript`. The variants have to be defined for the binary build and Doxygen documentation.

Listing 30.13: Implementation of the variant build

```
from waflib.Build import BuildContext, CleanContext, InstallContext, \
    UninstallContext, ListContext, StepContext
for x in variants:
    for y in (
        BuildContext,
        CleanContext,
        InstallContext,
        UninstallContext,
        ListContext,
        StepContext
        ):
        name = y.__name__.replace('Context','').lower()
        class tmp(y):
            if name == 'build':
                __doc__ = '''executes the {} of {}'''.format(name, x)
            elif name == 'clean':
                __doc__ = '''cleans the project {}'''.format(x)
            elif name == 'install' or name == 'uninstall':
                __doc__ = '''CURRENTLY NOT SUPPORTED:{}s the project {}'''.
↪format(name, x)
            elif name == 'list':
                __doc__ = '''lists the targets to execute for {}'''.format(x)
            elif name == 'step':
                __doc__ = '''executes tasks in a step-by-step fashion, for \
debugging of {}'''.format(x)
            cmd = name + '_' + x
            variant = x

    dox = 'doxygen'
    class tmp(BuildContext):
        __doc__ = '''creates the {} documentation of {}'''.format(dox, x)
        cmd = dox + '_' + x
        fun = dox
        variant = x
```

In the function build and doxygen the build variant is checked, the the correct sources are selected. If no build variant

is specified, an error message is displayed, telling to specify a variant. This is generally implemented something like this:

Listing 30.14: Implementation to ensure a variant build

```
def build(bld):
    import sys
    import logging
    from waflib import Logs
    if not bld.variant:
        bld.fatal('A {} variant must be specified, run \'{} {} --help\'\
'.format(bld.cmd, sys.executable, sys.argv[0]))

    bld.env.__sw_dir = os.path.normpath('embedded-software')

    src_dir = os.path.normpath('mcu-{}'.format(bld.variant))
    ldscript = os.path.join(bld.env.__sw_dir, src_dir, 'src', bld.env.ldscript_
↪filename)
```

For doxygen it is implemented very similar:

Listing 30.15: Implementation to ensure a variant doxgen API documentation

```
def doxygen(bld):
    import sys
    import logging
    from waflib import Logs

    if not bld.variant:
        bld.fatal('A build variant must be specified, run \'{} {} --help\'\
'.format(sys.executable, sys.argv[0]))

    if not bld.env.DOXYGEN:
        bld.fatal('Doxygen was not configured. Run \'{} {} --help\'\
'.format(sys.executable, sys.argv[0]))

    _docbuilddir = os.path.normpath(bld.bldnode.abspath())
    doxygen_conf_dir = os.path.join('documentation', 'doxygen')
    os.makedirs(_docbuilddir, exist_ok=True)
    conf_file = 'doxygen-{}.conf'.format(bld.variant)
    doxygenconf = os.path.join(doxygen_conf_dir, conf_file)
```

30.4 wscripts

As mentioned above, the build process is described in `wscripts`, which are itself valid python scripts. The top is `foxbms\wscript` which defines the functions needed for the build, e.g., `configure`, `build` etc.

From the top `wscript` the other `wscript` s are called recursive by `bld.recurse(..)`.

To get a detailed view on the single build steps, see these files.

30.5 Building and Linking with a Library

The toolchain enables to build a library and then links against the library. It is possible to build and link multiple libraries into the binaries.

The `wscript` in `embedded-software\libs` lists the libraries to be build. Libraries that should be build have to be listed here. Based on the example library `testlib` it is shown how to include a library in foxBMS.

Note: In fact the `libs` directory contains two test libraries (`foxbms-user-lib` and `my-foxbms-library`) in order to show how multiple libraries can be used. The first example shows how to build one single library and in the second example it is shown, how to build and use more than one library.

General Setup

The `wscript` in `embedded-software\libs` lists in the function `bld.recurse(...)` the directories containing the sources for the to be build library (see line 11 in Listing 30.16).

Listing 30.16: The `wscript` in `embedded-software\libs`

```
1  def build(bld):
2      header_files_src = bld.path.ant_glob('**/*.h')
3      header_filenames = [n.name for n in header_files_src]
4      if not len(header_filenames) == len(set(header_filenames)):
5          duplicates = list(set([x for x in header_filenames
6                                 if header_filenames.count(x) > 1]))
7          err_msg = 'There are headers with the same file name recursively ' \
8                    ' inside directory \'{}\':\n' \
9                    '{}'.format(bld.path.abspath(), '\n'.join(duplicates))
10         bld.fatal(err_msg)
11     bld.recurse('testlib myfoxbmslibrary')
```

Note: For every additional library that should be build, the directory containing the library must be added to this line, e.g., if the library sources are in a directory called `advancedalgorithms` this lines needs to look like this:

Listing 30.17: Adding the library source directory `advancedalgorithms`

```
1  def build(bld):
2      bld.recurse('testlib myfoxbmslibrary advancedalgorithms')
```

The actual build of the library is defined in the `wscript` in `embedded-software\libs\testlib`. All source and header files have to be in the library directory, for this example these are are `testlib.c` and `testlib.h`. The library is then build by the `wscript` in `embedded-software\libs\testlib`.

Listing 30.18 explained in detail:

- All source files that should be build have to be listed in the `srcs` list (see line 5-6.).
- The name of the library is set to `foxbms-user-lib` (see line 11).

Note: For later on further expanding the `advanced-algorithms` example, the library name `my-advanced-algorithm` is assumed.

Listing 30.18: wscript

```
1  def build(bld):
2      header_files_src = bld.path.ant_glob('*.h')
3      header_files_tar = [os.path.join(bld.env.INCLUDE_DIR_LIBS,
4                          x.path_from(bld.path)) for x in header_files_src]
5      srcs = ' '.join([
6          os.path.join('testlib.c')])
7
8      includes = os.path.join(bld.bldnode.abspath()) + ' '
9      includes += ' '.join([bld.path.get_src().abspath()])
10
11     bld.stlib(target='foxbms-user-lib',
12               source=srcs,
13               includes=includes,
14               cflags=bld.env.CFLAGS_foxbms,
15               features=['size', 'copy_libs', 'check_includes'])
16     bld(features='subst',
17         source=header_files_src,
18         target=header_files_tar,
19         is_copy=True)
```

- The object files (`*.o`) and the library (`*.a`) are found in `build\libs\embedded-software\libs\testlib\`.
- The libraries (the `*.a`-files) are copied in `build\lib\*.a`. When building the default dummy libraries these are `build\lib\libfoxbms-user-lib.a` and `build\lib\libmy-foxbms-library.a`. The *lib*-prefix is generated automatically. This task is generated by the `copy_lib` feature (see line 14).
- The headers are copied to `build\include` (see line 15-17).

Warning: The header names for all library headers are checked for uniqueness. Header files with the same name recursively inside the `libs` directory will lead to a build error. This check needs to be performed, as all headers get copied to include directory at `build\include`.

The Library

The library declaration of `super_function(uint8_t a, uint8_t b)` is in `testlib.h`:

Listing 30.19: testlib.h

```
1  #ifndef TESTLIB_H_
2  #define TESTLIB_H_
3
4  /*================== Includes ============================================*/
5  #include <stdint.h>
6
7  /*================== Macros and Definitions ==============================*/
8
9  /*================== Constant and Variable Definitions ===================*/
10 extern uint8_t super_variable;
11
12 /*================== Function Prototypes =================================*/
13 extern uint16_t super_function(uint8_t a, uint8_t b);
14
15 /*================== Function Implementations ============================*/
16
```

(continues on next page)

(continued from previous page)

```
#endif /* TESTLIB_H_ */
```

The library defines a function `super_function(uint8_t a, uint8_t b)` in `testlib.c`:

Listing 30.20: testlib.c

```
uint16_t super_function(uint8_t a, uint8_t b) {
    return a + b;
}
```

Building

1. Build the library (or libraries):

   Listing 30.21: Build the libraries

   ```
   python tools\waf build_libs
   ```

   Now all libraries are present in `build\libs` and the headers are in `build\include`.

2. Configure the foxBMS project to work with a library, in the first example it is the `foxbms-user-lib` library.

   Listing 30.22: Configuration with library useage

   ```
   python tools\waf configure --libs=foxbms-user-lib
   ```

Note: For including the hypothetical `my-advanced-algorithm` library the command would be:

```
python tools\waf configure --libs=my-advanced-algorithm
```

---

1. Include the headers needed for the functions in the sources and use the functions as needed.

   Listing 30.23: Include header and use a function from the library

   ```
   /*================== Includes ============================================*/
   /* some other includes  */
   #include "testlib.h"

   /*================== Function Prototypes ==================================*/

   /*================== Function Implementations ============================*/
   int main(void) {
       uint16_t a = 0;
       /* Use the function super_function from the library */
       a = super_function(2,2);
       /* other code */
   }
   ```

2. Build the foxBMS binary as usual.

Listing 30.24: Build the foxBMS binary

```
python tools\waf build_primary
```

Building with Multiple Libraries

A project may want to use multiple libraries. For this example the two provided dummy libraries are assumed (`foxbms-user-lib` and `my-foxbms-library`).

1. The project is configured to work with both libraries. The library names are given as command line argument separated by comma (**no additional whitespace**).

Listing 30.25: Configuration with multiple library useage

```
python tools\waf configure --libs=foxbms-user-lib,my-foxbms-library
```

2. Build the library (or libraries):

Listing 30.26: Build the libraries

```
python tools\waf build_libs
```

3. Include the headers needed for the functions in the sources and use the functions as needed.

Listing 30.27: Include header and use a function from the library

```
/*================== Includes ============================================*/
/* some other includes  */
#include "testlib.h"
#include "myfoxbmsalgorithms.h"

/*================== Function Prototypes ==================================*/

/*================== Function Implementations =============================*/
int main(void) {
    uint16_t a = 0;
    /* Use the function super_function from the library foxbms-user-lib */
    a = super_function(2,2);
    uint16_t b = 0;
    /* Use the function another_super_function from the library my-foxbms-library
    ↪*/
    a = another_super_function(2,2);
    /* other code */
}
```

4. Build the foxBMS binary as usual.

Listing 30.28: Build the foxBMS binary

```
python tools\waf build_primary
```

30.6 Building the Test

Note: The test builds described in this section are not mandatory. They can be used as a simple check that the software architecture is kept (see *foxBMS software architecture*).

In order to verify that low level drivers (i.e., the drivers in `embedded-software\mcu-common\driver`) do not relay on higher level modules (e.g., FreeRTOS, database, etc.) two tests are included. These can be build by

Listing 30.29: Build bare tests for primary and secondary mcu

```
python tools\waf configure
python tools\waf configure build_primary_bare
python tools\waf configure build_secondary_bare
```

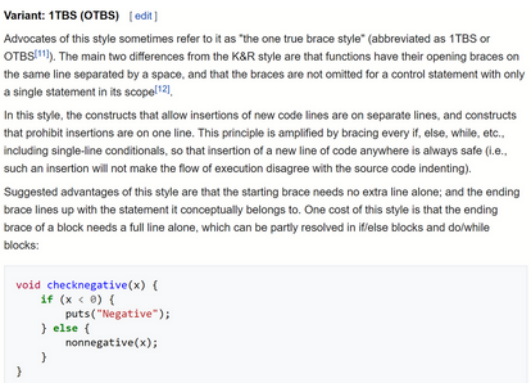
```
Indentation of the Switch statement.

One thing to keep in mind is to use as less level of
indention as possible in the code. To this end and to be
consistent with the indentation of the If and Else If
statetments, we prefer to use the following format:

switch (cond) {
case 1:
    handle_case1;
    break;
case 2:
    handle_case2;
    break;
default:
    handle_default;
}
```

CHAPTER 31

Software Style Guide

31.1 General

All plain text files must end with a single empty line (POSIX, 3.206 Line).



Variant: 1TBS (OTBS) [edit]

Advocates of this style sometimes refer to it as "the one true brace style" (abbreviated as 1TBS or OTBS[11]). The main two differences from the K&R style are that functions have their opening braces on the same line separated by a space, and that the braces are not omitted for a control statement with only a single statement in its scope[12].

In this style, the constructs that allow insertions of new code lines are on separate lines, and constructs that prohibit insertions are on one line. This principle is amplified by bracing every if, else, while, etc., including single-line conditionals, so that insertion of a new line of code anywhere is always safe (i.e., such an insertion will not make the flow of execution disagree with the source code indenting).

Suggested advantages of this style are that the starting brace needs no extra line alone; and the ending brace lines up with the statement it conceptually belongs to. One cost of this style is that the ending brace of a block needs a full line alone, which can be partly resolved in if/else blocks and do/while blocks:

```
void checknegative(x) {
    if (x < 0) {
        puts("Negative");
    } else {
        nonnegative(x);
    }
}
```

31.2 File Templates

All file templates are found in `/tools/styleguide/file-templates`

31.2.1 Source Code

In short: For foxbms we use for the embedded code 1TBS (check by *cpplint*) and *flake8* for the python code.

Listing 31.1: Template for C files

```
/**
 *
 * @copyright &copy; 2010 - 2019, Fraunhofer-Gesellschaft zur Foerderung der
 *  angewandten Forschung e.V. All rights reserved.
 *
 * BSD 3-Clause License
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 * 1.  Redistributions of source code must retain the above copyright notice,
 *     this list of conditions and the following disclaimer.
 * 2.  Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in the
 *     documentation and/or other materials provided with the distribution.
 * 3.  Neither the name of the copyright holder nor the names of its
 *     contributors may be used to endorse or promote products derived from
 *     this software without specific prior written permission.
```

(continues on next page)

(continued from previous page)

```
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *
 * We kindly request you to use one or more of the following phrases to refer
 * to foxBMS in your hardware, software, documentation or advertising
 * materials:
 *
 * &Prime;This product uses parts of foxBMS&reg;&Prime;
 *
 * &Prime;This product includes parts of foxBMS&reg;&Prime;
 *
 * &Prime;This product is derived from foxBMS&reg;&Prime;
 *
 */

/**
 * @file    template.c
 * @author  foxBMS Team
 * @date    00.00.0000 (date of creation)
 * @ingroup X_INGROUP
 * @prefix  XXX
 *
 * @brief   abc
 *
 * @details def
 *
 */

/*================== Includes ==============================================*/
#include "template.h"

/*================== Macros and Definitions ================================*/

/*================== Static Constant and Variable Definitions ==============*/

/*================== Extern Constant and Variable Definitions ==============*/

/*================== Static Function Prototypes (all should be listed here for better reference)*/

/*================== Static Function Implementations =======================*/

/*================== Extern Function Implementations (Prototypes are listed in the .h file)=======*/
                    Global function
```

Listing 31.2: Template for h files

```
/**
 *
 * @copyright &copy; 2010 - 2019, Fraunhofer-Gesellschaft zur Foerderung der
 *  angewandten Forschung e.V. All rights reserved.
 *
 * BSD 3-Clause License
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are met:
 * 1.  Redistributions of source code must retain the above copyright notice,
 *     this list of conditions and the following disclaimer.
 * 2.  Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in the
 *     documentation and/or other materials provided with the distribution.
 * 3.  Neither the name of the copyright holder nor the names of its
 *     contributors may be used to endorse or promote products derived from
 *     this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 *
 * We kindly request you to use one or more of the following phrases to refer
 * to foxBMS in your hardware, software, documentation or advertising
 * materials:
 *
 * &Prime;This product uses parts of foxBMS&reg;&Prime;
 *
 * &Prime;This product includes parts of foxBMS&reg;&Prime;
 *
 * &Prime;This product is derived from foxBMS&reg;&Prime;
 *
 */

/**
 * @file    template.h
 * @author  foxBMS Team
 * @date    00.00.0000 (date of creation)
 * @ingroup X_INGROUP
 * @prefix  XXX
 *
 * @brief   abc
 *
 * @details def
 *
 */

#ifndef TEMPLATE_H_
```

(continues on next page)

(continued from previous page)

```
#define TEMPLATE_H_

/*================== Includes ================================================*/

/*================== Macros and Definitions ==================================*/

/*================== Extern Constant and Variable Declarations ===============*/

/*================== Extern Function Prototypes ==============================*/

#endif /* TEMPLATE_H_ */
```

Listing 31.3: Template for python scripts

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

# @copyright &copy; 2010 - 2019, Fraunhofer-Gesellschaft zur Foerderung der
#   angewandten Forschung e.V. All rights reserved.
#
# BSD 3-Clause License
# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions are met:
# 1.  Redistributions of source code must retain the above copyright notice,
#     this list of conditions and the following disclaimer.
# 2.  Redistributions in binary form must reproduce the above copyright notice,
#     this list of conditions and the following disclaimer in the documentation
#     and/or other materials provided with the distribution.
# 3.  Neither the name of the copyright holder nor the names of its
#     contributors may be used to endorse or promote products derived from this
#     software without specific prior written permission.
#
# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
# AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
# IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
# ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
# LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
# CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
# SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
# INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
# CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
# ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
# POSSIBILITY OF SUCH DAMAGE.
#
# We kindly request you to use one or more of the following phrases to refer to
# foxBMS in your hardware, software, documentation or advertising materials:
#
# &Prime;This product uses parts of foxBMS&reg;&Prime;
#
# &Prime;This product includes parts of foxBMS&reg;&Prime;
#
# &Prime;This product is derived from foxBMS&reg;&Prime;

"""Example docstring

MUST BE GIVEN
```

(continues on next page)

(continued from previous page)

```
"""
import os
import sys
import argparse
import logging

__version__ = 0.1
__date__ = '2017-12-05'
__updated__ = '2017-12-05'


def main():
    """Use google style docstrings
from http://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html
    """
    program_name = os.path.basename(sys.argv[0])
    program_version = '{}'.format(__version__)
    program_build_date = str(__updated__)
    program_version_message = '{} {}'.format(
        program_version, program_build_date)
    program_shortdesc = __import__('__main__').__doc__.split('\n')[1]
    program_license = '''{}
{}

    Created by the foxBMS Team on {}.
    Copyright 2018 foxBMS. All rights reserved.

    Licensed under the BSD 3-Clause License.

    Distributed on an "AS IS" basis without warranties
    or conditions of any kind, either express or implied.

USAGE
'''.format(program_name, program_shortdesc, str(__date__))

    parser = argparse.ArgumentParser(
        description=program_license,
        formatter_class=argparse.RawDescriptionHelpFormatter)
    parser.add_argument(
        '-v',
        '--verbosity',
        dest='verbosity',
        action='count',
        default=0,
        help='set verbosity level')
    parser.add_argument(
        '-V',
        '--version',
        action='version',
        version=program_version_message)

    args = parser.parse_args()

    if args.verbosity == 1:
        logging.basicConfig(level=logging.INFO)
    elif args.verbosity > 1:
        logging.basicConfig(level=logging.DEBUG)
```

(continues on next page)

(continued from previous page)

```
    else:
        logging.basicConfig(level=logging.ERROR)


if __name__ == '__main__':
    main()
```

31.2.2 Documentation

Listing 31.4: Template for reStructuredText

```
.. include:: ../../macros.rst (RELATIVE LINK, MUST BE ADAPTED)

.. _DEFINE_A_LABEL_WHICH_STARTS_WITH_UNDERSCORE:

========
HEADINGS
========

TEXT TEXT

.. _DEFINE_Another_LABEL_WHICH_STARTS_WITH_UNDERSCORE:

SMALLER HEADING
---------------

TEXT TEXT
```

31.3 Details

31.3.1 C Sources

Doxygen

Every function has doxygen style documentation. For extern function these are in the header file, for static function these are in the c file at the implementation.

```
/**
 * @brief   sets the current state request of the state variable cont_state.
 *
 * @details This function is used to make a state request to the state machine,e.g,␣
↪start voltage
 *          measurement, read result of voltage measurement, re-initialization.
 *          It calls CONT_CheckStateRequest() to check if the request is valid. The␣
↪state request
 *          is rejected if is not valid. The result of the check is returned␣
↪immediately, so that
 *          the requester can act in case it made a non-valid state request.
 *
 * @param   state request to set
 *
```

(continues on next page)

(continued from previous page)

```
 * @return  CONT_OK if a state request was made, CONT_STATE_NO_REQUEST if no state␣
↪request was made
 */
CONT_RETURN_TYPE_e CONT_SetStateRequest(CONT_STATE_REQUEST_e statereq) {
    CONT_RETURN_TYPE_e retVal = CONT_STATE_NO_REQUEST;

    taskENTER_CRITICAL();
    retVal = CONT_CheckStateRequest(statereq);

    if (retVal == CONT_OK) {
            cont_state.statereq    = statereq;
    }
    taskEXIT_CRITICAL();

    return retVal;
}
```

Include Guard

The include guard must have the following form (considerng the filename is `template.h`)

```
#ifndef TEMPLATE_H_
#define TEMPLATE_H_

#endif /* TEMPLATE_H_ */
```

Naming conventions

Variables

- The module acronym must be used as prefix.
- All variable names must be in lower case.
- No prefix or suffix is required to distinguish global from static variables.
- Underscores may be used in the name of variables.

Functions

- The module acronym must be used as prefix.
- All function names starts with its acronym in capital letters followed by a concatenation of words where first letter of each word should start with capital letters.
- No prefix or suffix is required to distinguish extern from static functions.
- Underscores may be used in the name of functions.

Macros

- The module acronym must be used as prefix.
- All macro names must be in uppercase letters.
- Underscores may be used in the name of macros.

Examples

Module	Acronym	Example
spi.c/h	SPI	SPI_Init() spi_state SPI_MACRO
measurement.c/h	MEAS	MEAS_Init() meas_state MEAS_MACRO

Typedefs, Enums and Structs

- The module acronym must be used as prefix.
- All enum/struct names must be in uppercase letters.
- Underscores may be used in the name of enums/structs.
- Every enum/struct must be typedefed.
- Every enum/struct must have a general doxygen documentation comment in the form of `/** documentation comment */`
- Every enum/struct entry must be documented by a doxygen comment in the form of `/*!< documentation comment */`.

```
/** diagnosis handler return types */
typedef enum {
    DIAG_HANDLER_RETURN_OK                  = 0,            /*!<  error not occured␣
↪or occured but threshold not reached */
    DIAG_HANDLER_RETURN_ERR_OCCURED         = 2,            /*!<  error occured and␣
↪enabled */
    DIAG_HANDLER_RETURN_WARNING_OCCURRED    = 3,            /*!<  warning occured␣
↪(error occured but not enabled) */
    DIAG_HANDLER_RETURN_ERR_OCCURRED        = 4,            /*!<  error occured and␣
↪enabled */
    DIAG_HANDLER_RETURN_WRONG_ID            = 5,            /*!<  wrong diagnosis id␣
↪*/
    DIAG_HANDLER_RETURN_UNKNOWN             = 6,            /*!<  unknown return type␣
↪*/
    DIAG_HANDLER_INVALID_TYPE               = 7,            /*!<  invalid diagnosis␣
↪type, error in configuration */
    DIAG_HANDLER_INVALID_DATA               = 8,            /*!<  invalid data,␣
↪dependent of the diagHandler */
    DIAG_HANDLER_RETURN_NOT_READY           = 0xFFFFFFFF,   /*!<  diagnosis handler␣
↪not ready */
} DIAG_RETURNTYPE_e;
```

31.3.2 Includes

There are two scenarios to be considered for file includes. These are shown below.

Scenario 1

Normally a every software part has the following structure, considering a driver called `abc`:

- `*\config\abc_cfg.c`
- `*\config\abc_cfg.h`
- `*\abc\abc.c`
- `*\abc\abc.h`

The include strcture has then always to be like this:

Listing 31.5: abc_cfg.h

```
/* THE DRIVER CONFIGURATION HEADER abc_cfg.h ALWAYS INCLUDES general.h FIRST */
/* IF ADDITIONAL HEADERS ARE NEEDED FOR TYPES ETC. THESE ARE INCLUDED BELOW */
#include "general.h"
#include "additionally_needed_header_1.h"
```

Listing 31.6: abc_cfg.c

```
/* THE DRIVER CONFIGURATION IMPLEMENTATION abc_cfg.c ALWAYS INCLUDES THE␣
↪CONFIGURATION HEADER abc_cfg.h FIRST */
/* NEVER INCLUDE general.h IN THE DRIVER CONFIGURATION IMPLEMENTATION FILE abc_cfg.c␣
↪*/
/* IF ADDITIONAL HEADERS ARE NEEDED FOR TYPES ETC. THESE ARE INCLUDED BELOW */
#include "abc_cfg.h"
#include "additionally_needed_header_2.h"
```

Listing 31.7: abc.h

```
/* THE DRIVER HEADER abc.h ALWAYS INCLUDES THE DRIVER CONFIGURATION HEADER abc_cfg.h␣
↪FIRST */
/* NEVER INCLUDE general.h IN THE DRIVER HEADER FILE abc.h */
/* IF ADDITIONAL HEADERS ARE NEEDED FOR TYPES ETC. THESE ARE INCLUDED BELOW */
#include "abc_cfg.h"
#include "additionally_needed_header_3.h"
```

Listing 31.8: abc.c

```
/* THE DRIVER IMPLEMENTATION abc.c ALWAYS INCLUDES THE DRIVER HEADER abc.h FIRST */
/* NEVER INCLUDE general.h IN THE DRIVER IMPLEMENTATION FILE abc.c */
/* IF ADDITIONAL HEADERS ARE NEEDED FOR TYPES ETC. THESE ARE INCLUDED BELOW */
#include "abc.h"
#include "additionally_needed_header_4.h"
```

Scenario 2

If a software only has a driver header and a driver implementation

- `*\abc\abc.c`
- `*\abc\abc.h`

the include rules changed to

Listing 31.9: abc.h

```
/* THE DRIVER HEADER abc.h ALWAYS INCLUDES general.h FIRST */
/* IF ADDITIONAL HEADERS ARE NEEDED FOR TYPES ETC. THESE ARE INCLUDED BELOW */
#include "general.h"
#include "additionally_needed_header_1.h"
```

Listing 31.10: abc.c

```
/* THE DRIVER IMPLEMENTATION abc.c ALWAYS INCLUDES THE DRIVER HEADER abc.h FIRST */
/* NEVER INCLUDE general.h IN THE DRIVER IMPLEMENTATION FILE abc.c */
/* IF ADDITIONAL HEADERS ARE NEEDED FOR TYPES ETC. THESE ARE INCLUDED BELOW */
#include "abc.h"
#include "additionally_needed_header_2.h"
```

Use designated initializers whenever possible for initializations for the structures.

CHAPTER 32

Software FAQ and HOWTOs

This sections shows different kind of actions related to the software.

p.327

- *How to simply trigger events via CAN?*
- *How to start/stop balancing the battery cells?*
- *How to set the initial SOC value via CAN?*
- *How to reset deep-discharge flag via CAN?*
- *How to configure the voltage inputs?*
- *How to add/remove temperature sensors?*
- *How to change the resolution of the temperature values stored?*
- *How to enable and disable the checksum?*
- *How to call a user function to implement a specific algorithm?*

32.1 How to start foxBMS if no messages are sent by CAN?

If no current sensor is connected to foxBMS, it will not start. This default behavior can be changed by setting the switch

```
#define CURRENT_SENSOR_PRESENT                TRUE
```

to `FALSE` as explaned in *Important Switches in Code*.

32.2 How to create a task and change its priority and period?

First, declare a new task configuration in the `appltask_cfg.h` file:

```
/**
* Task configuration of the 10ms application task
*/
extern BMS_Task_Definition_s appl_tskdef_10ms;
```

The task configuration is a struct of type `BMS_Task_Definition_s` and defined as follows:

```
/**
* struct for FreeRTOS task definition
*/
typedef struct {
    uint32_t Phase;          /*!< (ms)                                    */
    uint32_t CycleTime;      /*!< (ms)                                    */
    OS_PRIORITY_e Priority;  /*!<                                         */
    uint32_t Stacksize;      /*!<  Defines the size, in words, of the
                                   stack allocated to the idle task.     */
} BMS_Task_Definition_s;
```

- Phase: phase offset of the task in ms
- CycleTime: cycle time of the task in ms
- Stacksize: stack size allocated to the task
- Priority: task priority for scheduling

The task priorities are defined by CMSIS and consist of seven priorities:

```
typedef enum  {
  osPriorityIdle          = -3,         // /< priority: idle (lowest)
  osPriorityLow           = -2,         // /< priority: low
  osPriorityBelowNormal   = -1,         // /< priority: below normal
  osPriorityNormal        =  0,         // /< priority: normal (default)
  osPriorityAboveNormal   = +1,         // /< priority: above normal
  osPriorityHigh          = +2,         // /< priority: high
  osPriorityRealtime      = +3,         // /< priority: realtime (highest)
  osPriorityError         =  0x84       // /< system cannot determine priority
                                        //    or thread has illegal priority
} osPriority;
```

The priorities `osPriorityRealtime`, `osPriorityHigh`, `osPriorityAboveNormal` and `osPriorityNormal` are already used by the foxBMS engine and therefore should not be used. In conlusion the priorities `osPriorityBelowNormal` and `osPriorityLow` shall be used.

Second, add the task configuration in the `appltask_cfg.c` file:

```
/**
 * predefined 10ms task for user code
 */
BMS_Task_Definition_s appl_tskdef_10ms = { 0,   10, osPriorityBelowNormal, 512/4 };
```

Third, declare a task handle and a task function in the `appltask.c/h` file:

```
 /**
  *  Definition of task handle 10ms task
  */
 xTaskHandle appl_handle_tsk_10ms;

/**
  * @brief   10ms engine application task
  */
 extern void APPL_TSK_Cyclic_100ms(void);
```

The task initialization and creation is done in the function `APPL_CreateTask()` in the `appltask.c` file. Before assigning the task handle to the newly created task, a new thread needs to be defined for the operating system. This is done by a call of the function `osThreadDef(name, thread, priority, instances, stacksz)`. The function parameters are:

- name: name of the function that represents the task
- thread: os_pthread-pointer to the function that represents the task
- priority: initial priority of the thread function
- instances: number of possible thread instances (0 -> only one instance)
- stacksize

The task handle is now, with a call of `osThreadCreate`, assigned to the task.

```
// Cyclic Task 10ms
osThreadDef(APPL_TSK_Cyclic_10ms, (os_pthread )APPL_TSK_Cyclic_10ms,
        appl_tskdef_10ms.Priority, 0, appl_tskdef_10ms.Stacksize);
appl_handle_tsk_10ms = osThreadCreate(osThread(APPL_TSK_Cyclic_10ms), NULL);
```

The implementation of the task should be done like shown in the following example:

```
void APPL_TSK_Cyclic_10ms(void) {
    while (os_boot != OS_SYSTEM_RUNNING)
    {
        ;
    }

    osDelayUntil(os_schedulerstarttime, appl_tskdef_10ms.Phase);

    while(1)
   {
        uint32_t currentTime = osKernelSysTick();

        APPL_Cyclic_10ms();

        osDelayUntil(currentTime, appl_tskdef_10ms.CycleTime);
    }
}
```

Note:

- Every task should have the while loop `while(os_boot != OS_SYSTEM_RUNNING)` at the beginning of the funtion. This prevents the task from being executed before foxBMS is completely initialized.
- `osDelayUntil(os_schedulerstarttime, appl_tskdef_10ms.Phase)` sets the wished phase offset of the task
- Tasks in FreeRTOS should never finish. Therefore, the actual task implementation is done in a `while(1)`-loop
- `APPL_TSK_Cyclic_10ms` serves as wrapper function for task implementation in `APPL_Cyclic_10ms()`
- The call of `osDelayUntil(currentTime, appl_tskdef_10ms.CycleTime)` sets the task in blocked state until the cycle time until the next period arrives
- Every task should be secured by the system monitoring module (`DIAG_SysMonNotify()`)
- foxBMS provides two default tasks for user applications with a periods of 10ms and 100ms

```
void APPL_Cyclic_10ms(void) {
    DIAG_SysMonNotify(DIAG_SYSMON_APPL_CYCLIC_10ms, 0);        // task is running,␣
↪state = ok

    /* User specific implementations:   */
    /*   ...                            */
    /*   ...                            */
}
```

32.3 How to add a software module and take it into account with WAF?

The steps to follow are:

- Creation of a new subfolder (for example `mymodule`) in one of the existing source folders `application`, `engine`, `general`, `module`, `module/utils`)
- Copy of all source files to the newly created subfolder

- Modification of the wscript file located in the chosen existing source folder, to add the new module. For example, to add a software module in the `application` folder, in the `includes` section of the wscript, the following new line has to be added:

```
os.path.join(bld.srcnode.abspath(), 'src', 'application', 'mymodule'),
```

In case the new software module has to be used in another existing module, the same line has to be added in the wscript file corresponding to the existing module where it is imported.

32.4 How to change the multiplexer measurement sequence for the LTC driver?

The sequence is defined in `module\config\ltc_cfg.c`, via the array `LTC_MUX_CH_CFG_t ltc_mux_seq_main_ch1[]`, which contains a concatenation of elements like:

```
{
    .muxID     = 1,
    .muxCh     = 0xFF,
},
{
   .muxID     = 2,
   .muxCh     = 0,
},
{
   .muxID     = 2,
   .muxCh     = 1,
},
```

There are 4 multiplexers with IDs from 0 to 3 on the foxBMS Slave Units. The multiplexer is chosen with the variable `muxID`. Each multiplexer has 8 channels, chosen with the variable `muxCh` (between 0 and 7). Channel 0xFF means that the multiplexer is disabled (i.e., high-impedance mode: none of the 8 inputs is connected to the output). With the code sequence shown above, multiplexer is first disabled, then channel 0 of multiplexer 2 is read, then channel 1 of multiplexer 2 is read.

Typically, multiplexer 0 and 1 are used for temperature measurement, and multiplexer 2 and 3 are used for balancing feedback (i.e., monitor if a cell is being balanced or not). As a consequence, by default, measurements of multiplexer 0 and 1 are stored in a database structure of type `DATA_BLOCK_CELLTEMPERATURE_s` and measurement of multiplexer 2 and 3 are stored in a database structure of type `DATA_BLOCK_BALANCING_FEEDBACK_s`.

For temperature measurements, the variable `uint8_t ltc_muxsensortemperatur_cfg[6]` contains the look-up table between temperature sensors and cells: the first entry defines the temperature sensor number assigned to the first cell, the second entry defines the temperature sensor number assigned to the second cell, and so on. If no look-up table is needed, this array should simply be filled with integers increasing from 0 to number of temperature sensors minus 1. In this example, `muxsensortemperaturmain_cfg[6]` has a size of 6 because it is the default number of temperature sensors supported by the foxBMS Slave Units. This must be adapted at two places:

- In `module\config\ltc_cfg.c`, where the variable is defined
- In `module\config\ltc_cfg.h`, in the declaration `extern uint8_t ltc_muxsensortemperatur_cfg[6]`

32.5 How to change the relation between voltages read by multiplexer via LTC6811-1 and temperatures?

The function `float LTC_Convert_MuxVoltages_to_Temperatures(float v_adc)` is defined in `ltc_cfg.c`. It gets a voltage in V as input and returns a temperature. It can simply be changed to meet the application needs.

To get the function converting the measured voltage to temperature, the following procedure can be followed if a Negative Temperature Coefficient resistor (NTC) is used :

1. Create a spreadsheet (e.g., in Microsoft Excel)
2. In the datasheet of the NTC, take the table giving the resistance versus the temperature
3. In the spreadsheet, define three columns:
    - Temperature value (read from the NTC datasheet)
    - Corresponding resistance value (read from the NTC datasheet)
    - Voltage value provided by the voltage divider calculated with the NTC resistance for each temperature value. On the latest foxBMS Slave Unit, the voltage divider is formed by a 10kOhm resistor in series with the NTC, with a 3V power supply, as shown in *Temperature Sensor Measurement*
    - Different slave versions may have different voltage dividers. Have a look a the *Slaves* documentation to select correct voltage divider for your slave.
4. Plot the temperature versus the corresponding measured voltage value (i.e., first column versus third column in the spreadsheet)
5. Fit a polynomial function to the plotted curve (e.g., by using Microsoft Excel)
6. Implement the polynom in `float LTC_Convert_MuxVoltages_to_Temperatures(float Vout)`

The function `float LTC_Convert_MuxVoltages_to_Temperatures(float Vout)` gets a voltage as input and outputs the corresponding temperature.

32.6 How to configure the MCU clock?

The configuration is defined in `module/config/rcc_cfg.c` via two structures:

```
RCC_OscInitTypeDef RCC_OscInitStruct = {
    .OscillatorType = RCC_OSCILLATORTYPE_HSE,
    .HSEState = RCC_HSE_ON,
    .PLL.PLLState = RCC_PLL_ON,
    .PLL.PLLSource = RCC_PLLSOURCE_HSE,
    .PLL.PLLM = RCC_PLL_M,     // Oscillator Clock: 8MHz -> (8Mhz / 8) * 336 / 2 ->␣
↪168MHz
    .PLL.PLLN = RCC_PLL_N,
    .PLL.PLLP = RCC_PLL_P,
    .PLL.PLLQ = RCC_PLL_Q      // Oscillator Clock: 8MHz -> (8Mhz / 8) * 336 / 7   ->␣
↪48MHz
};

RCC_ClkInitTypeDef RCC_ClkInitStruct = {
    .ClockType = RCC_CLOCKTYPE_SYSCLK|RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_PCLK1|RCC_
↪CLOCKTYPE_PCLK2,
```

(continues on next page)

(continued from previous page)

```
    .SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK,     // System Clock Source: PLL-Clock
                                                 // (Cortex-Core, AHB-Bus, DMA, memory)
    .AHBCLKDivider = RCC_AHBCLKDivider,          // Div=1 , AHB  CLOCK: 168MHz
    .APB1CLKDivider = RCC_APB1CLKDivider,        // Div=4 , APB1 CLOCK:  42MHz
    .APB2CLKDivider = RCC_APB2CLKDivider         // Div=2 , APB2 CLOCK:  84MHz
};
```

Fig. 32.1 shows a summary of the system clocks, with the variables defined via the structures and their effect (either as divider or multiplier).

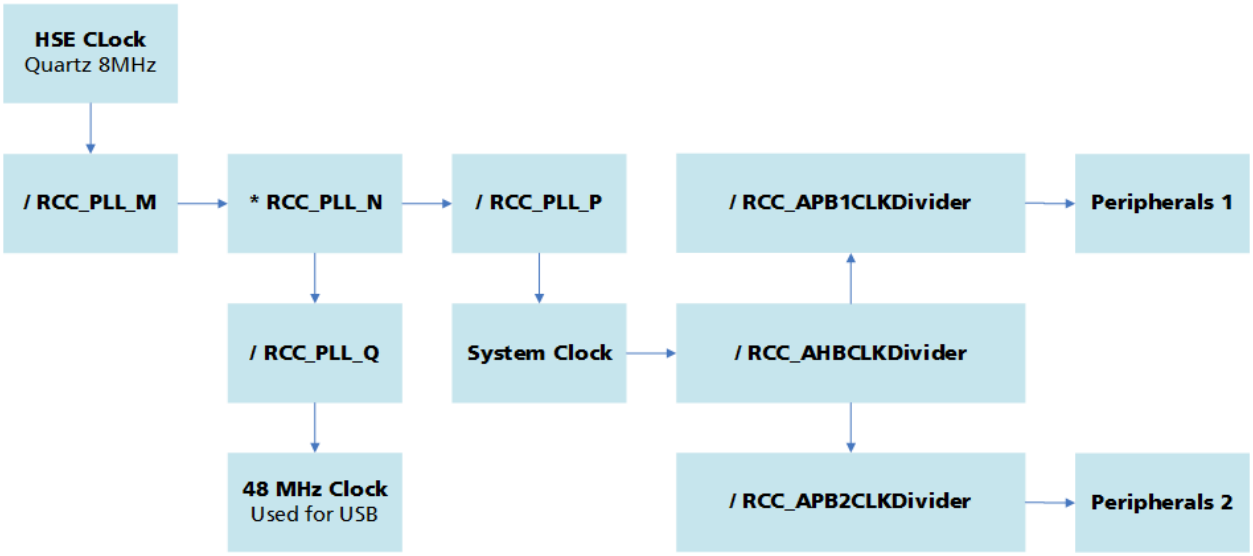


Fig. 32.1: Clock system of the microcontroller used in foxBMS

On the BMS-Master Board, a 8MHz oscillator is used as clock source. It should be noted that some of the multipliers/dividers can take all integers values in a certain range, while others can only take a specific set of values. The values must be defined so that the clock values are within the allowed ranges. These are defined in the microcontroller datasheet.

32.7 How to configure the LTC6811-1 and SPI clocks?

In `module/config/ltc_cfg.h`, the macro `#define SPI_HANDLE_LTC &spi_devices[0]` is defined. It points to a SPI handle: this SPI device will be used for the communication with the LTC6811-1. SPI handles are defined in `module/config/spi_cfg.c`. Depending on the SPI device chosen, the clock used will be peripheral clock 1 or 2 (this information is found in the STM32F4 datasheet). The clock used by the SPI device is obtained after division of the peripheral clock frequency. In the SPI handle, the value for the divider is defined via `.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_128`. In the LTC6811-1 driver, the SPI frequency is read directly via a HAL function and all timings are adapted automatically. The user must only ensure that the SPI frequency used for the LTC6811-1 is not higher than 1MHz. This is the maximal frequency allowed for the LTC6811-1 communication (as defined in the LTC6811-1 datasheet).

32.8 How to configure the CAN clock?

If the APB1 (Peripheral) clock changes, the CAN timing has to be adapted according to the following formula:

$$\text{clock}_{\text{CAN}} = \frac{\text{APB1}}{(\text{prescaler} + \text{timequantums})}$$

$$\text{timequantums} = 1 + \text{timequantumsBS1} + \text{timequantumsBS2}$$

The timequantums (TQ) are constrained to specific discrete values by the STM32 microcontroller.

Sample Configurations

CAN clock	APB1	Prescaler	BS1	BS2
1.0 MHz	42 MHz	3	6 TQ	7 TQ
1.0 MHz	32 MHz	4	5 TQ	2 TQ
0.5 MHz	42 MHz	6	6 TQ	7 TQ
0.5 MHz	32 MHz	8	5 TQ	2 TQ

Example:

change the relevant low level driver handle in file /modules/config/can_cfg.c

`CAN_HandleTypeDef hcan1` or `CAN_HandleTypeDef hcan2`

```
.Init.Prescaler = 3,          // CAN_CLOCK = APB1 = 42 MHz
                              // resulting CAN speed: APB1/prescaler/sumOfTimequants
                              // sum: 1tq for sync + BS1 + BS2
.Init.BS1 = CAN_BS1_6TQ,      // --> CAN = 42 MHz/(3*14) = 1.0 MHz
.Init.BS2 = CAN_BS2_7TQ,
```

$$\text{timequantums} = 1 + 6 + 7 = 14$$

$$\text{clock}_{CAN} = \frac{42.0}{(3 \cdot 14)} = 1.0$$

Further details can be found in STM32F4 datasheet.

32.9 How to configure and drive I/O ports?

The pin configuration of the hardware is is defined in `const IO_PIN_CFG_s io_cfg[]` in the file `module/config/io_cfg.c` with entries like:

```
{PIN_MCU_0_BMS_INTERFACE_SPI_MISO, IO_MODE_AF_PP, IO_PIN_NOPULL,
 IO_SPEED_HIGH, IO_ALTERNATE_AF5_SPI1, IO_PIN_LOCK_ENABLE}
```

The parameters are:

- Pin: Defines the pin name (defined in `module/config/io_mcu0_cfg.h`).
- Mode: The possibilities of the mode are defined in `module/config io_cfg.c` via the enum type `IO_PIN_MODES_e`. Often used modes are `IO_MODE_AF_PP` for use of one alternate function of the pin, `IO_MODE_INPUT` to use the pin as an input, `IO_MODE_OUTPUT_PP` to use the pin as an output with push-pull functionality.
- Pinpull: Defines wether the pin is used without pull-up or pull-down (`IO_PIN_NOPULL`), to use the pin with pull-up (`IO_PIN_PULLUP`) or to use the pin with pull-down (`IO_PIN_PULLDOWN`).

- Speed: Defines the speed of the pin (`IO_SPEED_LOW`, `IO_SPEED_MEDIUM`, `IO_SPEED_FAST` or `IO_SPEED_HIGH`).
- Alternate: Defines if the signal/pin uses an alternate function or not. If no alternate function is used this is set to `IO_ALTERNATE_NO_ALTERNATE`. If the signal/pin uses a alternate function one can choose from the possibilities from the enumeration `IO_PIN_ALTERNATE_e` in `module/config/io_cfg.h`.
- Pinlock: `IO_PIN_LOCK_DISABLE` or `IO_PIN_LOCK_ENABLE` to disable or enable pin locking.
- Initvalue: Sets the initial state of the pin in case of an output pin; The pin is set to `IO_PIN_RESET` for 0/low and to `IO_PIN_SET` for 1/high. If no value is given for a output pin it is set to low.

As explained above, the signal names are to be defined in `module/config/io_mcu0_cfg.h` with macros like:

```
#define PIN_MCU_0_BMS_INTERFACE_SPI_MISO IO_PA_6
```

where `IO_Px_y` corresponds to the physical pin on the MCU, with `x` the port (e.g., A,B,C) and `y` the pin number on the port (e.g., 0,1,2).

This configuration is initialized in `main.c` with the function call `IO_Init(&io_cfg[0])`.

Pins configured as output are driven with the function with `IO_PIN_RESET` or `IO_PIN_SET` to set the pin to low or high. Example (The signal/pin name corresponds to the one defined in `module/config/io_mcu0_cfg.h`).:

```
IO_WritePin(PIN_MCU_0_BMS_INTERFACE_SPI_MISO, IO_PIN_RESET);     // set pin low
IO_WritePin(PIN_MCU_0_TO_MCU_1_INTERFACE_SPI_MISO, IO_PIN_SET);  // set pin high
```

The states of pins configured as input are read with the `IO_ReadPin(<Signalname>)` function. The function returns `IO_PIN_RESET` or `IO_PIN_SET` (0 or 1). Example with the signal/pin name corresponding to the one defined in `module/config/io_mcu0_cfg.h`:

```
IO_PIN_STATE_e pinstate = IO_PIN_RESET;
pinstate = IO_ReadPin(PIN_MCU_0_BMS_INTERFACE_SPI_NSS);
```

32.10 How to add and configure interrupts?

The interrupt configuration can be found in `general/config/nvic_cfg.c` via the variable `NVIC_InitStruct_s nvic_interrupts[]`, which contains entries of the form: `{ DMA2_Stream2_IRQn, 2, NVIC_IRQ_LOCK_ENABLE, NVIC_IRQ_ENABLE }`

The configuration parameters are:

- Symbolic name of interrupt source (as defined in the system file stm32f429xx.h)
- Interrupt priority: number between 0 and 15, a lower number means a higher priority
- Parameter irqlock: if set to `NVIC_IRQ_LOCK_ENABLE`, the interrupt is locked according to the initial state and cannot be modified by the interface functions `NVIC_EnableInterrupts()` or `NVIC_DisableInterrupts()`
- Initial state of interrupt source: set to `NVIC_IRQ_ENABLE` to get the interrupt enabled by the initialization function. In case of `NVIC_IRQ_DISABLE`, the interrupt must be activated by calling `NVIC_EnableInterrupts()` after the initialization

In `general/config/stm32f4xx_it.c`, a corresponding callback function must be defined (for example `void DMA2_Stream2_IRQHandler(void)` for DMA stream 2 of DMA device 2). It will be called when the interrupt is triggered.

For a proper operation, the interrupt handling (callback function) has to execute the following steps:

- Clear the pending interrupt with for example `HAL_NVIC_ClearPendingIRQ(DMA2_Stream2_IRQn)`
- Call the HAL IRQ handler (or a custom handler) with for example: `HAL_DMA_IRQHandler(&dma_devices[0])`

Note: Interrupt routines with interrupt priority above the maximum FreeRTOS configuration level (`configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY`) must not call FreeRTOS API functions. These interrupts are real-time interrupts, which are bypassing the operating system. Interrupt routines with interrupt priority equal or lower than this maximum level must call the corresponding FreeRTOS API functions with ending `...._FROM_ISR()`.

Check the documentation (datasheet, reference manual) of the interrupt source for additional steps.

32.11 How to add a database entry and to read/write it?

The example of the entry for cell voltages is taken. In `engine\config\database_cfg.h`, the definition of a block must be added `#define DATA_BLOCK_ID_CELLVOLTAGE DATA_BLOCK_1` with one available block. The blocks are defined via the following enumeration:

```
typedef enum {
    DATA_BLOCK00        = 0,
    DATA_BLOCK01        = 1,
    DATA_BLOCK02        = 2,
    DATA_BLOCK03        = 3,
    DATA_BLOCK04        = 4,
    DATA_BLOCK05        = 5,
    DATA_BLOCK06        = 6,
    DATA_BLOCK07        = 7,
    DATA_BLOCK08        = 8,
    DATA_BLOCK09        = 9,
    DATA_BLOCK10        = 10,
    DATA_BLOCK_MAX      = DATA_MAX_BLOCK_NR,
} DATA_BLOCK_ID_TYPE_e;
```

If more than `DATA_BLOCK_MAX` blocks are needed, it must be changed in the defines:

```
#define DATA_MAX_BLOCK_NR               11
```

A structure must then be declared for the block ID created:

```
/*  data structure declaration of DATA_BLOCK_ID_CELLVOLTAGE */
typedef struct {
 /* Timestamp info needs to be at the beginning. Automatically written on DB_
↪WriteBlock      */
    uint32_t timestamp;                     /*!< timestamp of database entry
↪        */
    uint32_t previous_timestamp;            /*!< timestamp of last database entry
↪        */
    /* data block entries
↪        */
    uint16_t      voltage[NR_OF_BAT_CELLS];   //unit: mV
    uint8_t       state;
} DATA_BLOCK_CELLVOLTAGE_s;
```

This structure needs to contain a variable timestamp and previous_timestamp at the beginning of the struct. These timestamp are automatically updated each time new values are stored in the database. The reamaing struct consists of all the data needed for the entry. In `mcu-primary\engine\config\database_cfg.c`, a variable with the structure type must be declared

```
DATA_BLOCK_CELLVOLTAGE_s data_block_cellvoltage[DOUBLE_BUFFERING];
```

The user can choose `SINGLE_BUFFERING` or `DOUBLE_BUFFERING`. The last step is to add an entry in the structure `DATA_BASE_HEADER_s data_base_header[]`:

```
{
(void*)(&data_block_cellvoltage[0]),
sizeof(DATA_BLOCK_CELLVOLTAGE_s),
DOUBLE_BUFFERING,
},
```

With either `SINGLE_BUFFERING` or `DOUBLE_BUFFERING` (the same as in the structure declaration). When access to the created database entry is needed, a local variable with the corresponding type must be created in the module where it is needed: `DATA_BLOCK_CELLVOLTAGE_s cellvoltage;`

Access to a data field is made with the usual C-syntax:

```
cellvoltage.voltage[i]
```

Getting the data from the database in the local variable is made via:

```
DB_ReadBlock(&cellvoltage ,DATA_BLOCK_ID_CELLVOLTAGE)
```

Storing data from the local variable to the database is made via:

```
DB_WriteBlock(&cellvoltage, DATA_BLOCK_ID_CELLVOLTAGE)
```

During each call of `DB_WriteBlock()` the timestamps will automatically be updated.

32.12 How to store data in the backup SRAM of the MCU?

The STM32F4 has 4kB Backup SRAM. Variables can be stored there with the keywork `MEM_BKP_SRAM` which is defined in `src/general/config/global.h`.

Example:

```
#define DIAG_FAIL_ENTRY_LENGTH (50)
DIAG_ERROR_ENTRY_s MEM_BKP_SRAM diag_memory[DIAG_FAIL_ENTRY_LENGTH];
```

32.13 How to manually add CAN entries (transmit and receive) and to change the transmit time period?

are discussed

Several steps have to be done to add a transmit message and signal (message and signal in receive direction in parentheses):

File: `module/config/cansignal_cfg.h`

1. The message name must be added. For TX, i.e., transmit data, a message entry must be added in the enumeration `CANS_messagesTx_e`: (RX, i.e., receive, `CANS_messagesRx_e`)

2. Then one or more signal names must be added in the enumeration `CANS_CANx_signalsTx_e` (`CANS_CANx_signalsRx_e`)

File: `module/config/can_cfg.c`

3. The CAN message must be defined

3.1. In case a TX message should be defined - add the message for CAN0 or CAN1 in the respective array (const CAN_MSG_TX_TYPE_s can_CANx_messages_tx). The message looks like `{ 0x123, 8, 100, 20, NULL_PTR }`

The parameters are:

- Message ID (11bit standard identifier used in foxBMS)
- Data Length Code (i.e., number of bytes), usually 8
- Transmit period in ms: must be multiple of `CANS_TICK_MS`
- Delay in ms for sending the message the first time: must be multiple of `CANS_TICK_MS`
- A function pointer to be called after transmission, `NULL_PTR` if nothing needs to be done

3.2 If a RX message needs to be configured, the message for CAN0 or CAN1 must be added in the respective array (CAN_MSG_RX_TYPE_s can0_RxMsgs[]). The message looks like `{ 0x123, 0xFFFF, 8, 0, CAN_FIFO0, NULL }`

The parameters are:

- Message ID (11bit standard identifier used in foxBMS)
- Mask for the CAN hardware filter (Select mask or use 0x0000 to select list mode)
- Data Length Code (i.e., number of bytes), usually 8
- Hardware receive FIFO (CAN_FIFO0 or CAN_FIFO1)
- function pointer to be called after reception, usually NULL_PTR to do nothing because signals with respective function pointers are used

File: `module/config/cansignal_cfg.c`

4. Then the signals added in the header must be added in the .c file and in `const CANS_signal_s cans_CANx_signals_tx[]` (`const CANS_signal_s cans_CANx_signals_rx[NR_SIGNALS_RX]`).

   A signal looks like:

```
{CANS_MSG_Name}, 0, 8, 0, 255, 1, 0, NULL_PTR, &cans_getsignaldata
```

   The parameters are:

   - Symbolic name of the message containing the signal (defined in `CANS_messagesTx_e`, in the header file)
   - Start bit of signal
   - Signal length in bits
   - Minimum value (float)
   - Maximum value (float)
   - Scaling factor (float)
   - Scaling offset (float)

- Callback function for setter (when CAN msg is received) (`NULL_PTR` if no function needed, e.g., for transmit)
- Callback function gor getter (when CAN msg is transmitted) (`NULL_PTR` if no function needed, e.g., for receive only)

5. The callback functions must be declared and implemented in `cansignal_cfg.c`.

32.14 How to adapt the CAN module when less than 18 battery cells or 12 temperature sensors per module are used?

Five adaptions are necessary when removing unused battery cell voltages or temperatures This procedure is executed exemplarily for 9 cell voltages and 5 temperature sensors:

1. Adapt struct `const CAN_MSG_TX_TYPE_s can_CAN0_messages_tx[]` in file `module/config/can_cfg.c`:

```
{ 0x200, 8, 200, 20, NULL_PTR },  //!< Cell voltages module 0 cells 0 1 2
{ 0x201, 8, 200, 20, NULL_PTR },  //!< Cell voltages module 0 cells 3 4 5
{ 0x202, 8, 200, 20, NULL_PTR },  //!< Cell voltages module 0 cells 6 7 8
{ 0x203, 8, 200, 20, NULL_PTR },  //!< Cell voltages module 0 cells 9 10 11
{ 0x204, 8, 200, 20, NULL_PTR },  //!< Cell voltages module 0 cells 12 13 14
{ 0x205, 8, 200, 20, NULL_PTR },  //!< Cell voltages module 0 cells 15 16 17

{ 0x210, 8, 200, 30, NULL_PTR },  //!< Cell temperatures module 0 cells 0 1 2
{ 0x211, 8, 200, 30, NULL_PTR },  //!< Cell temperatures module 0 cells 3 4 5
{ 0x212, 8, 200, 30, NULL_PTR },  //!< Cell temperatures module 0 cells 6 7 8
{ 0x213, 8, 200, 30, NULL_PTR },  //!< Cell temperatures module 0 cells 9 10 11
```

Remove all unused CAN message depending on the number of used cells/sensors. In the example case remove cell voltage message 0x203 because we transmit only cell voltages 0-8. Additionally remove cell temperature message 0x212 and 0x213 because we only use temperatures 0-4. Repeat this process for all modules.

2. Adapt enum `CANS_messagesTx_e` in file `module/config/cansignal_cfg.h`:

```
CAN0_MSG_Mod0_Cellvolt_0,  //!< Module 0 Cell voltages 0-2
CAN0_MSG_Mod0_Cellvolt_1,  //!< Module 0 Cell voltages 3-5
CAN0_MSG_Mod0_Cellvolt_2,  //!< Module 0 Cell voltages 6-8
CAN0_MSG_Mod0_Cellvolt_3,  //!< Module 0 Cell voltages 9-11
CAN0_MSG_Mod0_Cellvolt_4,  //!< Module 0 Cell voltages 12-14
CAN0_MSG_Mod0_Cellvolt_5,  //!< Module 0 Cell voltages 15-17

CAN0_MSG_Mod0_Celltemp_0,  //!< Module 0 Cell temperatures 0-2
CAN0_MSG_Mod0_Celltemp_1,  //!< Module 0 Cell temperatures 3-5
CAN0_MSG_Mod0_Celltemp_2,  //!< Module 0 Cell temperatures 6-8
CAN0_MSG_Mod0_Celltemp_3,  //!< Module 0 Cell temperatures 9-11
```

Remove all unused enmus depending on the number of used cells/sensors. In the example case remove enum `CAN0_MSG_Mod0_Cellvolt_3` because we transmit only cell voltages 0-8. Additionally remove cell temperature enum `CAN0_MSG_Mod0_Celltemp_2` and `CAN0_MSG_Mod0_Celltemp_3` because we only use temperatures 0-4. Repeat this process for all modules.

3. Adapt enum `CANS_CAN0_signalsTx_e` in file `module/config/cansignal_cfg.h`:

```
CAN0_SIG_Mod0_volt_valid_0_2,
CAN0_SIG_Mod0_volt_0,
```

(continues on next page)

(continued from previous page)

```
CAN0_SIG_Mod0_volt_1,
CAN0_SIG_Mod0_volt_2,
CAN0_SIG_Mod0_volt_valid_3_5,
CAN0_SIG_Mod0_volt_3,
CAN0_SIG_Mod0_volt_4,
CAN0_SIG_Mod0_volt_5,
CAN0_SIG_Mod0_volt_valid_6_8,
CAN0_SIG_Mod0_volt_6,
CAN0_SIG_Mod0_volt_7,
CAN0_SIG_Mod0_volt_8,
CAN0_SIG_Mod0_volt_valid_9_11,
CAN0_SIG_Mod0_volt_9,
CAN0_SIG_Mod0_volt_10,
CAN0_SIG_Mod0_volt_11,
CAN0_SIG_Mod0_volt_valid_12_14,
CAN0_SIG_Mod0_volt_12,
CAN0_SIG_Mod0_volt_13,
CAN0_SIG_Mod0_volt_14,
CAN0_SIG_Mod0_volt_valid_15_17,
CAN0_SIG_Mod0_volt_15,
CAN0_SIG_Mod0_volt_16,
CAN0_SIG_Mod0_volt_17,

CAN0_SIG_Mod0_temp_valid_0_2,
CAN0_SIG_Mod0_temp_0,
CAN0_SIG_Mod0_temp_1,
CAN0_SIG_Mod0_temp_2,
CAN0_SIG_Mod0_temp_valid_3_5,
CAN0_SIG_Mod0_temp_3,
CAN0_SIG_Mod0_temp_4,
CAN0_SIG_Mod0_temp_5,
CAN0_SIG_Mod0_temp_valid_6_8,
CAN0_SIG_Mod0_temp_6,
CAN0_SIG_Mod0_temp_7,
CAN0_SIG_Mod0_temp_8,
CAN0_SIG_Mod0_temp_valid_9_11,
CAN0_SIG_Mod0_temp_9,
CAN0_SIG_Mod0_temp_10,
CAN0_SIG_Mod0_temp_11,
```

Remove all unused enmus depending on the number of used cells/sensors. In the example case remove enums `CAN0_SIG_Mod0_volt_valid_9_11` to `CAN0_SIG_Mod0_volt_11` because we transmit only cell voltages 0-8. Additionally remove cell temperature enums `CAN0_SIG_Mod0_temp_5` to CAN0_SIG_Mod0_temp_11 because we only use temperatures 0-4. Repeat this process for all modules.

4. Adapt struct `const CANS_signal_s cans_CAN0_signals_tx[]` in file `module/config/cansignal_cfg.c`:

```
/* Module 0 cell voltages */
{ {CAN0_MSG_Mod0_Cellvolt_0}, 0, 8, 0, UINT8_MAX, 1, 0, NULL_PTR, &cans_getvolt },  /
↪*!< CAN0_SIG_Mod0_volt_valid_0_2 */
{ {CAN0_MSG_Mod0_Cellvolt_0}, 8, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt }, ␣
↪/*!< CAN0_SIG_Mod0_volt_0 */
{ {CAN0_MSG_Mod0_Cellvolt_0}, 24, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt },␣
↪ /*!< CAN0_SIG_Mod0_volt_1 */
{ {CAN0_MSG_Mod0_Cellvolt_0}, 40, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt },␣
↪ /*!< CAN0_SIG_Mod0_volt_2 */
```

(continues on next page)

(continued from previous page)

```
{ {CAN0_MSG_Mod0_Cellvolt_1}, 0, 8, 0, UINT8_MAX, 1, 0, NULL_PTR, &cans_getvolt },  /
↪*!< CAN0_SIG_Mod0_volt_valid_3_5 */
{ {CAN0_MSG_Mod0_Cellvolt_1}, 8, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt }, ␣
↪/*!< CAN0_SIG_Mod0_volt_3 */
{ {CAN0_MSG_Mod0_Cellvolt_1}, 24, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt },␣
↪ /*!< CAN0_SIG_Mod0_volt_4 */
{ {CAN0_MSG_Mod0_Cellvolt_1}, 40, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt },␣
↪ /*!< CAN0_SIG_Mod0_volt_5 */
{ {CAN0_MSG_Mod0_Cellvolt_2}, 0, 8, 0, UINT8_MAX, 1, 0, NULL_PTR, &cans_getvolt },  /
↪*!< CAN0_SIG_Mod0_volt_valid_6_8 */
{ {CAN0_MSG_Mod0_Cellvolt_2}, 8, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt }, ␣
↪/*!< CAN0_SIG_Mod0_volt_6 */
{ {CAN0_MSG_Mod0_Cellvolt_2}, 24, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt },␣
↪ /*!< CAN0_SIG_Mod0_volt_7 */
{ {CAN0_MSG_Mod0_Cellvolt_2}, 40, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt },␣
↪ /*!< CAN0_SIG_Mod0_volt_8 */
{ {CAN0_MSG_Mod0_Cellvolt_3}, 0, 8, 0, UINT8_MAX, 1, 0, NULL_PTR, &cans_getvolt },  /
↪*!< CAN0_SIG_Mod0_volt_valid_9_11 */
{ {CAN0_MSG_Mod0_Cellvolt_3}, 8, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt }, ␣
↪/*!< CAN0_SIG_Mod0_volt_9 */
{ {CAN0_MSG_Mod0_Cellvolt_3}, 24, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt },␣
↪ /*!< CAN0_SIG_Mod0_volt_10 */
{ {CAN0_MSG_Mod0_Cellvolt_3}, 40, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt },␣
↪ /*!< CAN0_SIG_Mod0_volt_11 */
{ {CAN0_MSG_Mod0_Cellvolt_4}, 0, 8, 0, UINT8_MAX, 1, 0, NULL_PTR, &cans_getvolt },  /
↪*!< CAN0_SIG_Mod0_volt_valid_12_14 */
{ {CAN0_MSG_Mod0_Cellvolt_4}, 8, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt }, ␣
↪/*!< CAN0_SIG_Mod0_volt_12 */
{ {CAN0_MSG_Mod0_Cellvolt_4}, 24, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt },␣
↪ /*!< CAN0_SIG_Mod0_volt_13 */
{ {CAN0_MSG_Mod0_Cellvolt_4}, 40, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt },␣
↪ /*!< CAN0_SIG_Mod0_volt_14 */
{ {CAN0_MSG_Mod0_Cellvolt_5}, 0, 8, 0, UINT8_MAX, 1, 0, NULL_PTR, &cans_getvolt },  /
↪*!< CAN0_SIG_Mod0_volt_valid_15_17 */
{ {CAN0_MSG_Mod0_Cellvolt_5}, 8, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt }, ␣
↪/*!< CAN0_SIG_Mod0_volt_15 */
{ {CAN0_MSG_Mod0_Cellvolt_5}, 24, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt },␣
↪ /*!< CAN0_SIG_Mod0_volt_16 */
{ {CAN0_MSG_Mod0_Cellvolt_5}, 40, 16, 0, UINT16_MAX, 1, 0, NULL_PTR, &cans_getvolt },␣
↪ /*!< CAN0_SIG_Mod0_volt_17 */

/* Module 0 cell temperatures */
{ {CAN0_MSG_Mod0_Celltemp_0}, 0, 8, 0, UINT8_MAX, 1, 0, NULL_PTR, &cans_gettemp },  /
↪*!< CAN0_SIG_Mod0_volt_valid_0_2 */
{ {CAN0_MSG_Mod0_Celltemp_0}, 8, 16, -128, 527.35, 100, 128, NULL_PTR, &cans_gettemp }
↪,  /*!< CAN0_SIG_Mod0_temp_0 */
{ {CAN0_MSG_Mod0_Celltemp_0}, 24, 16, -128, 527.35, 100, 128, NULL_PTR, &cans_gettemp␣
↪},  /*!< CAN0_SIG_Mod0_temp_1 */
{ {CAN0_MSG_Mod0_Celltemp_0}, 40, 16, -128, 527.35, 100, 128, NULL_PTR, &cans_gettemp␣
↪},  /*!< CAN0_SIG_Mod0_temp_2 */
{ {CAN0_MSG_Mod0_Celltemp_1}, 0, 8, 0, UINT8_MAX, 1, 0, NULL_PTR, &cans_gettemp },  /
↪*!< CAN0_SIG_Mod0_volt_valid_3_5 */
{ {CAN0_MSG_Mod0_Celltemp_1}, 8, 16, -128, 527.35, 100, 128, NULL_PTR, &cans_gettemp }
↪,  /*!< CAN0_SIG_Mod0_temp_3 */
{ {CAN0_MSG_Mod0_Celltemp_1}, 24, 16, -128, 527.35, 100, 128, NULL_PTR, &cans_gettemp␣
↪},  /*!< CAN0_SIG_Mod0_temp_4 */
{ {CAN0_MSG_Mod0_Celltemp_1}, 40, 16, -128, 527.35, 100, 128, NULL_PTR, &cans_gettemp␣
↪},  /*!< CAN0_SIG_Mod0_temp_5 */
```

(continues on next page)

(continued from previous page)

```
{ {CAN0_MSG_Mod0_Celltemp_2}, 0, 8, 0, UINT8_MAX, 1, 0, NULL_PTR, &cans_gettemp },  /
↪*!< CAN0_SIG_Mod0_volt_valid_6_8 */
{ {CAN0_MSG_Mod0_Celltemp_2}, 8, 16, -128, 527.35, 100, 128, NULL_PTR, &cans_gettemp }
↪,  /*!< CAN0_SIG_Mod0_temp_6 */
{ {CAN0_MSG_Mod0_Celltemp_2}, 24, 16, -128, 527.35, 100, 128, NULL_PTR, &cans_gettemp␣
↪},  /*!< CAN0_SIG_Mod0_temp_7 */
{ {CAN0_MSG_Mod0_Celltemp_2}, 40, 16, -128, 527.35, 100, 128, NULL_PTR, &cans_gettemp␣
↪},  /*!< CAN0_SIG_Mod0_temp_8 */
{ {CAN0_MSG_Mod0_Celltemp_3}, 0, 8, 0, UINT8_MAX, 1, 0, NULL_PTR, &cans_gettemp },  /
↪*!< CAN0_SIG_Mod0_volt_valid_9_11 */
{ {CAN0_MSG_Mod0_Celltemp_3}, 8, 16, -128, 527.35, 100, 128, NULL_PTR, &cans_gettemp }
↪,  /*!< CAN0_SIG_Mod0_temp_9 */
{ {CAN0_MSG_Mod0_Celltemp_3}, 24, 16, -128, 527.35, 100, 128, NULL_PTR, &cans_gettemp␣
↪},  /*!< CAN0_SIG_Mod0_temp_10 */
{ {CAN0_MSG_Mod0_Celltemp_3}, 40, 16, -128, 527.35, 100, 128, NULL_PTR, &cans_gettemp␣
↪},  /*!< CAN0_SIG_Mod0_temp_11 */
```

Remove all unused struct entries depending on the number of used cells/sensors. Repeat this process for all modules.

5. Adapt functions `cans_getvolt` and `cans_gettemp` in file `module/config/cansignal_cfg.c`:

Remove all case-statements for which the signals have been deleted in the previous steps.

32.15 How to adapt the CAN module when less or more than 8 battery modules are used?

When less than 8 modules are used, the same procedure as described in *How to adapt the CAN module when less or more than 8 battery modules are used?* must be used but only the unused modules instead of unused cell voltages/temperatures must be deleted. If additional modules should be added also follow these instructions but instead of deleting these signals and CAN messages add the wished number of new messages.

Morover an adaption of the functions `cans_getvolt` and `cans_gettemp` in file `module/config/cansignal_cfg.c` is necessary:

```
// Determine module and cell number
if (sigIdx - CAN0_SIG_Mod0_volt_valid_0_2 < CANS_MODULSIGNALS_VOLT) {
    modIdx = 0;
    cellIdx = sigIdx - CAN0_SIG_Mod0_volt_valid_0_2;
} else if (sigIdx - CAN0_SIG_Mod1_volt_valid_0_2 < CANS_MODULSIGNALS_VOLT) {
    modIdx = 1;
    cellIdx = sigIdx - CAN0_SIG_Mod1_volt_valid_0_2;
} else if (sigIdx - CAN0_SIG_Mod2_volt_valid_0_2 < CANS_MODULSIGNALS_VOLT) {
    modIdx = 2;
    cellIdx = sigIdx - CAN0_SIG_Mod2_volt_valid_0_2;
} else if (sigIdx - CAN0_SIG_Mod3_volt_valid_0_2 < CANS_MODULSIGNALS_VOLT) {
    modIdx = 3;
    cellIdx = sigIdx - CAN0_SIG_Mod3_volt_valid_0_2;
} else if (sigIdx - CAN0_SIG_Mod4_volt_valid_0_2 < CANS_MODULSIGNALS_VOLT) {
    modIdx = 4;
    cellIdx = sigIdx - CAN0_SIG_Mod4_volt_valid_0_2;
} else if (sigIdx - CAN0_SIG_Mod5_volt_valid_0_2 < CANS_MODULSIGNALS_VOLT) {
    modIdx = 5;
    cellIdx = sigIdx - CAN0_SIG_Mod5_volt_valid_0_2;
} else if (sigIdx - CAN0_SIG_Mod6_volt_valid_0_2 < CANS_MODULSIGNALS_VOLT) {
```

(continues on next page)

(continued from previous page)

```
    modIdx = 6;
    cellIdx = sigIdx - CAN0_SIG_Mod6_volt_valid_0_2;
} else if (sigIdx - CAN0_SIG_Mod7_volt_valid_0_2 < CANS_MODULSIGNALS_VOLT) {
    modIdx = 7;
    cellIdx = sigIdx - CAN0_SIG_Mod7_volt_valid_0_2;
}
```

This if statement performs a module detection to allocate the correct cell voltage to the respective CAN message. This statement needs to be shortened/extended depending on the number of used modules. The same method is used to allocate the cell temperatures and thus needs to be adapted as well.

Note: The define CANS_MODULSIGNALS_TEMP is calculated as followed:

```
#define CANS_MODULSIGNALS_TEMP  (CAN0_SIG_Mod1_volt_valid_0_2 - CAN0_SIG_Mod0_temp_
↪valid_0_2)
```

When only one module is used this define needs to be redefined on the next signal used after the last temperature signal of module 0. If no signal is defined after the module 0 temperature signals use the last defined temperature signal. For the default configuration of foxBMS this could look like this:

```
#define CANS_MODULSIGNALS_TEMP  ((CAN0_SIG_Mod0_temp_11 - CAN0_SIG_Mod0_temp_valid_0_
↪2) + 1)
```

32.16 How to change the blink period of the indicator LEDs?

It is defined via the three variables in `module/utils/led.c`:

- `led_0_nbr_of_blink`
- `led_1_nbr_of_blink`
- `led_blink_time`

Currently the three variables are set directly in the function `LED_Ctrl()`. If desired or needed, the three values can be read from the database and so set elsewhere in the code.

The time is set in ms via `led_blink_time`. The LED cycle is divided in two periods, T0 and T1. Each of both periods lasts `led_blink_time` ms. During T0, LED0 is on or blinks and LED1 is off. During T1, LED0 is off and LED1 is on or blinks. The behavior described as ''is on or blinks" is defined via `led_0_nbr_of_blink` for LED0 and via `led_1_nbr_of_blink` for LED1. If `led_0_nbr_of_blink` is set to 1, LED0 is on during T0. If `led_0_nbr_of_blink` is set to n greater than 1, LED0 will blink n times during T1. For example, if the period is set to 3000ms and `led_0_nbr_of_blink` set to 3, LED0 will have the following behavior during T0:

- On for 500ms
- Off for 500ms
- On for 500ms
- Off for 500ms
- On for 500ms
- Off for 500ms

The behavior is the same for LED1 with `led_1_nbr_of_blink`. With this functionality, each LED can blink independently n times. If each LED can blink up to 4 times, the LEDs can visually give 16 different messages.

The adjustment of the blink period is provided to make reading by the user easier: for higher number of blinks, the blink period can be made longer: for example, it is easier to count 4 blinks in 3s than 4 blinks in 1s).

32.17 How to add an entry for the diag module?

Adding a new sensitivity level is done in `src/engine/config/diag_cfg.h` by adding a new `#define` in:

```
#define DIAG_ERROR_SENSITIVITY_HIGH         (0)   // logging at first event
#define DIAG_ERROR_SENSITIVITY_MID          (5)   // logging at fifth event
#define DIAG_ERROR_SENSITIVITY_LOW          (10)  // logging at tenth event
#define DIAG_ERROR_SENSITIVITY_CUSTOM       (100) // logging at 100th event
```

and errors can be defined:

```
#define     DIAG_CH_CUSTOM_FAILURE              DIAG_ID_XX
```

and the error is then added in `DIAG_CH_CFG_s diag_ch_cfg[]` in file `src/engine/config/diag_cfg.c`:

```
DIAG_CH_CFG_s  diag_ch_cfg[] = {
 {DIAG_CH_CUSTOM_FAILURE,         DIAG_GENERAL_TYPE, DIAG_ERROR_SENSITIVITY_HIGH,
  DIAG_RECORDING_ENABLED, DIAG_ENABLED, dummyfu},
```

The number of logged error events can be changed by modifying the value of the macro:

```
#define DIAG_FAIL_ENTRY_LENGTH                (50)    // Number of errors that can be␣
↪logged
```

Note: The diag module is a powerful module for general error handling. The user has to be aware of timings when using custom diag entries. As example how to use this module correct syscontrol is choosen. - The function `SYSCTRL_Trigger()` is called in the 10ms task (`ENG_TSK_Cyclic_10ms()`), meaning every 10ms this function must be executed. - In the diagnosis-module header `diag_cfg.h` there is the enum `DIAG_SYSMON_MODULE_ID_e` for the different error types that are handeled by the diagnosismodule. For syscontrol errors there is `DIAG_SYSMON_SYSCTRL_ID`. - In the diagnosis-module source `diag_cfg.c` there is the `diag_sysmon_ch_cfg[]` array assigning timings to this error, in this case 20ms.

```
{DIAG_SYSMON_SYSCTRL_ID, DIAG_SYSMON_CYCLICTASK, 20,
DIAG_RECORDING_ENABLED, DIAG_ENABLED, dummyfu2},
```

This means every time `SYSCTRL_Trigger()` is called, the function indicating *syscontrol is running* has to be exectued. If this is not done, the diagnosis module will set the syscontrol to the error state. Therefore, the user must set up functions, which are wanted to be supervised by the diagnosis module, and that they are still running, in this way:

```
void SYSCTRL_Trigger(SYSCTRL_TRIG_EVENT_e event) {
    DIAG_SysMonNotify(DIAG_SYSMON_SYSCTRL_ID, 0);    // task is running, therefore
                            // reset state to 0
    /* user code */
}
```

32.18 How to configure contactors without feedback?

In the file `src/module/config/contactor_cfg.c`, the contactors are defined with:

```
CONT_CFG_s cont_contactors_cfg[NR_OF_CONTACTORS] = {
        {CONT_MAIN_PLUS_CONTROL,        CONT_MAIN_PLUS_FEEDBACK,        CONT_FEEDBACK_
↪NORMALLY_OPEN},
        {CONT_PRECHARGE_PLUS_CONTROL,   CONT_PRECHARGE_PLUS_FEEDBACK,   CONT_FEEDBACK_
↪NORMALLY_OPEN},
        {CONT_MAIN_MINUS_CONTROL,       CONT_MAIN_MINUS_FEEDBACK,       CONT_FEEDBACK_
↪NORMALLY_OPEN}
};
```

The following change must be made to configure a contactor without feedback. In this case, the precharge contactor is taken as example:

```
{CONT_PRECHARGE_PLUS_CONTROL,   CONT_PRECHARGE_PLUS_FEEDBACK,   CONT_HAS_NO_FEEDBACK}
```

With this configuration, the feedback pin (`CONT_PRECHARGE_PLUS_FEEDBACK`) is ignored and the feedback value will always be equal to the expected value for the precharge contactor. This configuration should not be used without precautions since the safety level will be reduced.

32.19 How to simply trigger events via CAN?

The CAN message with ID 0x100 is considered as a *debug message*. If received, the function `cans_setdebug()` defined in `cansignal_cfc.c` is called. There, a switch case is used on the first byte (byte0) of the message to define what to do. This means that 256 actions are possible, and that 7 bytes remain to transfer data in the debug message.

32.20 How to start/stop balancing the battery cells?

When the `#define BALANCING_DEFAULT_INACTIVE` is set to `FALSE`, foxBMS starts the balancing process automatically if the balancing requirements are met. The balancing behavior is described in details in the documentation of the *Balancing* module.

This behavior can be influenced via a debug message sent per CAN (*How to simply trigger events via CAN?*).

When the data `0E 01 00 00 00 00 00 00` is sent, the balancing state machine goes to the `BAL_STATEMACH_INACTIVE_OVERRIDE` state. No balancing takes place.

When the data `0E 03 00 00 00 00 00 00` is sent, the balancing state machine goes to the `BAL_STATEMACH_ACTIVE_OVERRIDE` state. Balancing takes place without taking into account the state of the `bms` module, the current flowing through the battery and the minimum cell voltage in the battery pack.

When the data `0E 02 00 00 00 00 00 00` is sent, the balancing state machine goes out of the override state. Balancing takes place taking into account the state of the `bms` module, the current flowing through the battery and the minimum cell voltage in the battery pack.

32.21 How to set the initial SOC value via CAN?

This is done via the debug message with the first byte (byte0) equal to 11 (0x0B). The SOC is defined via the next two bytes (i.e., on 16bit: byte1 byte2). The SOC is given in 0.01% unit, which means that the 16bit number should

be comprised between 0 and 10000. If a smaller value is given, the SOC will be set to 0%. If a greater value is given, SOC will be set to 100%.

32.22 How to reset deep-discharge flag via CAN?

Note: This message should only be sent after replacing the affected cell as charging a deep-discharged cell is highly dangerous.

The deep-discharge flag can be reset by transmitting debug message with the first byte (byte0) equal to 170 (0xAA).

32.23 How to configure the voltage inputs?

This number is changed in `batterysystem_cfg.h` with the define `BS_NR_OF_BAT_CELLS_PER_MODULE`. In addition, the variable

```
const uint8_t ltc_voltage_input_used[BS_MAX_SUPPORTED_CELLS]
```

must be adapted, too, in `ltc_cfg.c`.

It has the size of `BS_MAX_SUPPORTED_CELLS`. If a cell voltage is connected to the LTC IC input, `1` must be written in the table. Otherwise, `0` must be written.

For instance, if 5 cells are connected to inputs `0`, `2`, `5`, `7`, `11`,

```
#define BS_NR_OF_BAT_CELLS_PER_MODULE               5
```

must be used and

```
const uint8_t ltc_voltage_input_used[BS_MAX_SUPPORTED_CELLS] = {
    1 ,
    0 ,
    1 ,
    0 ,
    0 ,
    1 ,
    0 ,
    1 ,
    0 ,
    0 ,
    0 ,
    1 ,
};
```

must be defined. The number of `1` in the table must be equal to `BS_NR_OF_BAT_CELLS_PER_MODULE`.

32.24 How to add/remove temperature sensors?

To add one temperature sensor, the first step is to change:

```
#define BS_NR_OF_TEMP_SENSORS_PER_MODULE            6
```

to:

```
#define BS_NR_OF_TEMP_SENSORS_PER_MODULE            7
```

in `batterysystem_cfg.h`.

However, this obvious step is not sufficient.

In `ltc_cfg.c`, the variable `LTC_MUX_CH_CFG_s ltc_mux_seq_main_ch1[]` which indicates the channel sequence to read on the multiplexer has to be modified. One sensor channel on one multiplexer has to be added. In this example, on the multiplexer with the ID 0 and channel 6 is read additionally.

Concretely:

```
{
    .muxID     = 1,
    .muxCh     = 0xFF,
},
{
    .muxID     = 2,
    .muxCh     = 0xFF,
},
{
    .muxID     = 3,
    .muxCh     = 0xFF,
},
{
    .muxID     = 0,
    .muxCh     = 0,
},
{
    .muxID     = 0,
    .muxCh     = 1,
},
{
    .muxID     = 0,
    .muxCh     = 2,
},
{
    .muxID     = 0,
    .muxCh     = 3,
},
{
    .muxID     = 0,
    .muxCh     = 4,
},
{
    .muxID     = 0,
    .muxCh     = 5,
},
{
    .muxID     = 0,       //configure the multiplexer to be used
    .muxCh     = 6,       //configure input to ne used on the selected multiplexer
},
```

must be used instead of:

```
{
    .muxID     = 1,
```

(continues on next page)

(continued from previous page)

```
    .muxCh     = 0xFF,
},
{
    .muxID     = 2,
    .muxCh     = 0xFF,
},
{
    .muxID     = 3,
    .muxCh     = 0xFF,
},
{
    .muxID     = 0,
    .muxCh     = 0,
},
{
    .muxID     = 0,
    .muxCh     = 1,
},
{
    .muxID     = 0,
    .muxCh     = 2,
},
{
    .muxID     = 0,
    .muxCh     = 3,
},
{
    .muxID     = 0,
    .muxCh     = 4,
},
{
    .muxID     = 0,
    .muxCh     = 5,
},
```

Then the look-up table must be modified: it allows defining the correspondence between multiplexer channel and sensor order in the temperature array. By default, the mapping keeps the same order. In the previous example:

```
const uint8_t ltc_muxsensortemperatur_cfg[6] = {
    1-1 ,       /*!< index 0 = mux 0, ch 0 */
    2-1 ,       /*!< index 1 = mux 0, ch 1 */
    3-1 ,       /*!< index 2 = mux 0, ch 2 */
    4-1 ,       /*!< index 3 = mux 0, ch 3 */
    5-1 ,       /*!< index 4 = mux 0, ch 4 */
    6-1 ,       /*!< index 5 = mux 0, ch 5 */
    //7-1 ,     /*!< index 6 = mux 0, ch 6 */
    //8-1 ,     /*!< index 7 = mux 0, ch 7 */
    //9-1 ,     /*!< index 8 = mux 1, ch 0 */
    //10-1 ,    /*!< index 9 = mux 1, ch 1 */
    //11-1 ,    /*!< index 10 = mux 1, ch 2 */
    //12-1 ,    /*!< index 11 = mux 1, ch 3 */
    //13-1 ,    /*!< index 12 = mux 1, ch 4 */
    //14-1 ,    /*!< index 13 = mux 1, ch 5 */
    //15-1 ,    /*!< index 14 = mux 1, ch 6 */
    //16-1      /*!< index 15 = mux 1, ch 7 */
};
```

must be changed to:

```
const uint8_t ltc_muxsensortemperatur_cfg[7] = {
    1-1 ,        /*!< index 0 = mux 0, ch 0 */
    2-1 ,        /*!< index 1 = mux 0, ch 1 */
    3-1 ,        /*!< index 2 = mux 0, ch 2 */
    4-1 ,        /*!< index 3 = mux 0, ch 3 */
    5-1 ,        /*!< index 4 = mux 0, ch 4 */
    6-1 ,        /*!< index 5 = mux 0, ch 5 */
    7-1 ,      /*!< index 6 = mux 0, ch 6 */
    //8-1 ,      /*!< index 7 = mux 0, ch 7 */
    //9-1 ,      /*!< index 8 = mux 1, ch 0 */
    //10-1 ,     /*!< index 9 = mux 1, ch 1 */
    //11-1 ,     /*!< index 10 = mux 1, ch 2 */
    //12-1 ,     /*!< index 11 = mux 1, ch 3 */
    //13-1 ,     /*!< index 12 = mux 1, ch 4 */
    //14-1 ,     /*!< index 13 = mux 1, ch 5 */
    //15-1 ,     /*!< index 14 = mux 1, ch 6 */
    //16-1       /*!< index 15 = mux 1, ch 7 */
};
```

A last step has to be done in `ltc_cfg.h`: adjust the size of the array, by changing:

```
extern const uint8_t ltc_muxsensortemperatur_cfg[6];
```

into:

```
extern const uint8_t ltc_muxsensortemperatur_cfg[7];
```

The same procedure must be used to remove a temperature sensor.

32.25 How to change the resolution of the temperature values stored?

The scaling of the raw values read from the LTC6811-1 is made in `ltc.c`, in the function `static void LTC_SaveTemperatures_SaveBalancingFeedback(void)`. First, the raw data are read from the SPI buffer with:

```
val_ui=*((uint16_t *)(&LTC_MultiplexerVoltages[2*((LTC_NUMBER_OF_LTC_PER_MODULE*i*LTC_
↪N_MUX_CHANNELS_PER_LTC)+muxseqptr->muxID*LTC_N_MUX_CHANNELS_PER_MUX+muxseqptr->
↪muxCh)]));
```

Then the raw data are scaled and are available in Volt:

```
val_fl = ((float)(val_ui))*100e-6;
```

Then the read voltage is converted into a temperature with a look-up table:

```
val_si = (int16_t)(LTC_Convert_MuxVoltages_to_Temperatures(val_fl));
```

The last step is to store the temperature into a `sint16` variable:

```
ltc_celltemperature.temperature[i*(NR_OF_TEMP_SENSORS_PER_MODULE)+sensor_idx]=val_si;
```

At this step, the resolution can be changed. For instance, m°C could be stored instead of °C.

32.26 How to enable and disable the checksum?

The checksum mechanism is enabled via the following define in the file `general.h`:

```
#define BUILD_MODULE_ENABLE_FLASHCHECKSUM 1
```

32.27 How to call a user function to implement a specific algorithm?

In `embedded-software\mcu-primary\src\application\config\appltask_cfg.c`, the user function must simply be called in one of the periodic user function:

```
void APPL_Cyclic_1ms(void)
void APPL_Cyclic_10ms(void)
void APPL_Cyclic_100ms(void)
```

depending on the periodicity needed. The user function can access system data via the database.

For instance, to access the voltages, a structure must be declared with

```
DATA_BLOCK_CELLVOLTAGE_s voltages;
```

The data is retrieved from the database with

```
DB_ReadBlock(&voltages, DATA_BLOCK_ID_CELLVOLTAGE);
myvoltage = voltages.voltage[i];
```

The field `voltages.timestamp` is updated with the system timestamp (in ms) everytime new voltages are written in the database. This timestamp can be used to check if the voltages have been updated.

To store the results, an entry must be created in the database, as explained in *How to add a database entry and to read/write it?*.

Once the results are stored in the database, they can for instance be sent by CAN.

Bibliography

[ltc_datasheet6804] LTC6804 Datasheet http://cds.linear.com/docs/en/datasheet/680412fb.pdf

[ltc_datasheet6811] LTC6811 Datasheet http://cds.linear.com/docs/en/datasheet/68111f.pdf

[ltc_datasheet] LTC6804 Datasheet http://www.linear.com/product/LTC6804-1