Industrial Battery Front End API Software

This software manual provides a description of the Application Programming Interface intended for use with the industrial Battery Font End ICs from the RAA489xxx series. The document contains a complete description of the elements of the interface, application guidelines, and examples.

Contents

1. Introduction 1
1.1 Assumptions and Advisory Notes 2
2. Battery Front End Software Structure 2
3. Battery Front End Application Programming Interface 4
3.1 Interface Structure 4
3.2 BFE Interface Instance Structure 13
3.3 Using the BFE Interface 14
3.3.1 Preparation 14
3.3.2 Configuration 14
3.3.3 Use of API Functions 16
3.3.4 Examples 17
4. Revision History 18

1. Introduction

The Application Programming Interface (API) described in this manual facilitates the implementation of firmware for Battery Management Systems (BMSs) that feature industrial-grade Battery Front Ends (BFEs). It provides a common interface to configure, control, and use BFE features, helping accelerate the development of firmware by enabling the reuse of code, facilitating system-level integration, and improving code scalability.

The compatible BFE devices are from the RAA489xxx family, which targets a wide range of applications from low voltage range with a couple of cells up to powerful designs with large stackable battery packs. Due to the diversity of applications supported by this BFE family, the features of some BFEs can differ significantly from each other. For example, some feature an integrated FET driver, external cell balancing, built-in diagnostic features, and configurable thresholds, whereas others only offer basic monitoring functions and fixed thresholds. Nevertheless, protecting the battery pack integrity is common to all of them, so they can be controlled using a common firmware API that provides scalable and universal functionalities to the BMS. Unique features offered exclusively by specific BFEs can be included in the API as additional interface implementations so that no functionality is hidden behind the universality of the common API.

The interface described in this manual is designed for Renesas 32-bit MCUs with ARM Cortex-M Core from the RA MCU family. It uses the Flexible Software Package (FSP) of the e^{2}studio integrated development environment, which generates Hardware Abstraction Layer drivers. However, the API can be ported to other MCU architectures by adapting the driver modules generated by FSP.

The industrial battery management product portfolio of Renesas contains two basic architectures. The first uses a Fuel Gauge Integrated Circuit (FGIC), which has a co-packaged Analog Front End and MCU. The second architecture contains a separate Battery Front End device and external MCU. Figure 1 shows the separate BFE-MCU architecture for which the API is designed.

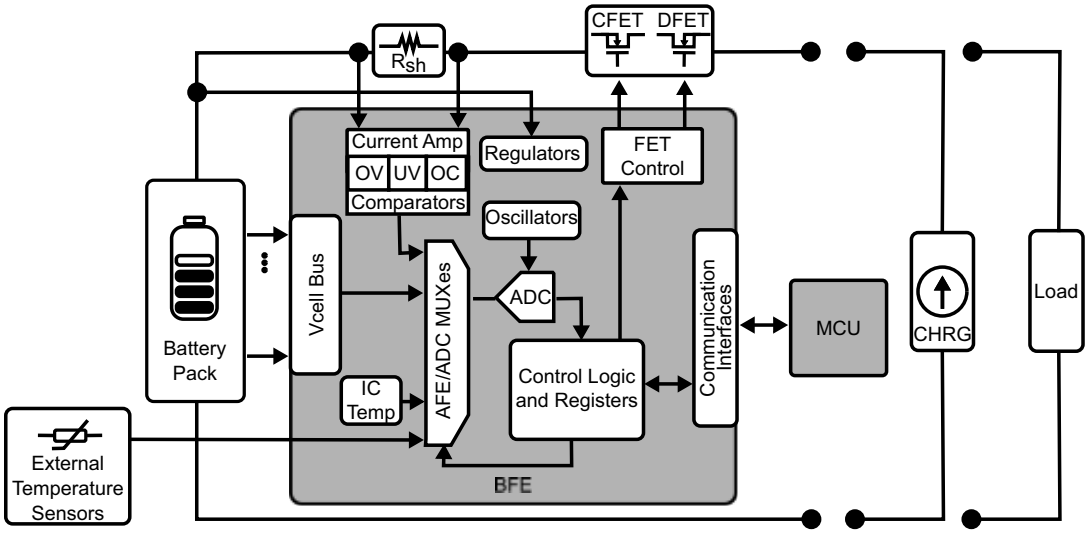

Figure 1. General Block Diagram of Battery Management System

1.1 Assumptions and Advisory Notes

- It is assumed that you possess basic understanding of microcontrollers, embedded systems hardware, battery management systems and Li-based battery cells.
- It is assumed that you have prior experience working with Integrated Development Environments (IDEs) such as e2studio and Flexible Software Package (FSP).
- It is assumed that you are familiar with the Renesas RA family ARM Cortex-M microcontrollers.
- It is assumed that you are familiar with the Renesas industrial Battery Front Ends from the RAA489xxx series.

2. Battery Front End Software Structure

At the software architecture level, modules provide functionalities using interfaces, which can be thought of as contracts between the module providing functionality and the module requiring it. The API allows modules to be swapped in and out by instances that implement the same interface; therefore, defining BFE functionalities as functions of the API of an abstraction layer, referred to as Battery Abstraction Layer (BAL), allows applications such as BMS State Machine, Cell Balancing, State-of-Charge or etc. to use potentially any module that implements it. Figure 2 shows the software structure of the industrial BFE API.

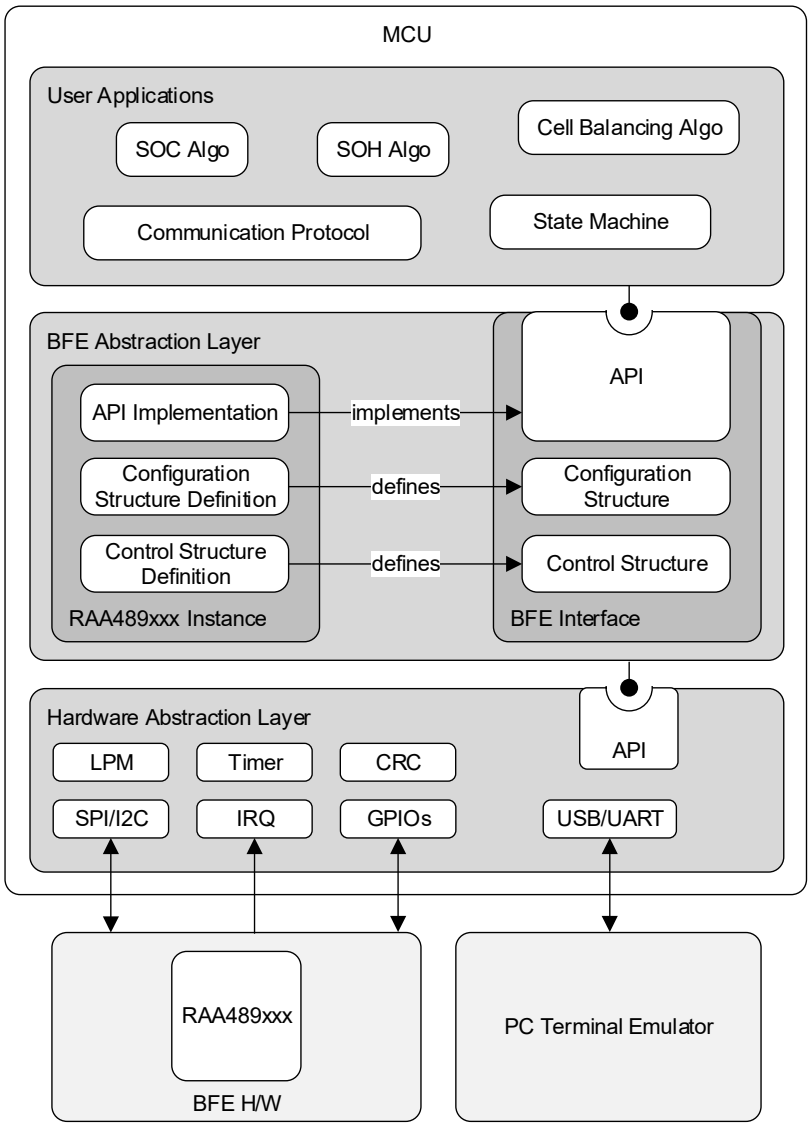

Figure 2. General Software Structure with BFE API

The software structure of Battery Front End Application Programming Interface has multiple layers and sub-components. They are the following:

- Hardware layer comprises the Battery Management System (BMS). It contains the following hardware:
  - Single Battery Front End IC or multiple, connected in a stack
  - MCU that controls the entire BMS system
  - External HMI device or remote management unit
  - In some hardware configurations other components could be available like external current monitor, MOSFET driver, etc.
- Software layer for driving the MCU (Hardware Abstraction Layer drivers) with the following APIs to interact with the BFE device and the other hardware:
  - Inter-chip communication module between the BFE and the MCU. It is typically SPI or I2C.
  - General Purpose Input/Output (GPIO) to access and configure I/O ports that control, configure and reset the BFE.
  - External Interrupt Request (IRQ) interface to detect events like Fault pin assertion and other generated by the BFE.

  - USB/ UART interface to communicate with the HMI device
  - Timer module used for critical timings tracking.
  - Hardware Cyclic Redundancy Check calculator to verify communication packed content.
  - Additional interfaces used in the particular implementation.
- Battery Front End Abstraction Layer (BAL). It is a middleware module that is placed between the higher-level applications and the HAL drivers. It comprises the Battery Front End API and the instance of the particular BFE. You can find there the implementations of all API functions providing access to the BFE resources, diagnostic functions, and communication drivers.
- Software layer comprising the high-level user applications such as:
  - State Machine that is managing the BMS states and modes.
  - Fault Management Algorithm for handling the BFE faults and the returned API interface error codes.
  - State-of-Charge algorithm
  - State-of-Health algorithm
  - Cell Balancing algorithm
  - High-level communication protocol for displaying information to the user of connecting the BMS to other external systems.

Table 1 shows the file structure of BFE API implementation. The listed files are mandatory but more could be available in the particular BFE implementation. Keep in mind that the additional files, related to a system and applications are not on focus. The current document reviews the content of **src/r_bfe/r_bfe_api.h**.

Table 1. Directory Structure of the Sample Code

Directory		Filename	Description
src	r_bfe	r_bfe_api.h	Contains the interface type definitions and everything related to it.
		r_bfe_cfg.h	Contains the interface settings.
		r_raa489xxx.c	Contains the actual code for the interface implementation.
		r_raa489xxx.h	Contains definitions, structures, enumerations and declarations of the API functions for the interface implementation.
	apps	bal_data.c	Contains the BFE instance and declarations of the major structures.
		bal_data.h	Contains the exported global variables of the interface.

3. Battery Front End Application Programming Interface

3.1 Interface Structure

The BAL API declares the signature of the functions that applications can use. Their actual implementation is provided by modules, namely instances, which define the body of the functions with code that execute the required functionality. Table 2 describes the members of the BFE API structure. All members are pointers to functions, whose parameters are constants or pointers to variables (structures) used for input or output of values. Some of the variable types are of type void, so each instance of the API must define them, such as structures, according to the BFE features. These generic types are referred to as instance-defined parameters. For example, they are used for the returned temperatures, cell voltages or faults but also for specific configuration or control options for cell balancing, auto-scan. The parameter of data type **st_bfe_ctrl_t** is a control structure. It is common to all API functions and works as a unique identifier of the BFE instance. All API functions return error codes defined by the enumeration **e_bfe_err_t** (Table 3). These codes are used to indicate successful execution or appearance of an error.

Table 2. BFE API Structure and its Fields

typedef struct st_bfe_api		
Member	**Parameters**	**Description**
* p_initialize	*st_bfe_ctrl_t* * **const** p_ctrl *st_bfe_cfg_t* **const** * const p_cfg	Initializes the BFE interface by enabling and configuring the necessary peripheral modules of the MCU, identifying the BFE and other device-specific actions.
* p_deinitialize	*st_bfe_ctrl_t* * **const** p_ctrl	Deinitializes the BFE interface. It disables the used peripheral modules of the MCU.
* p_setup	*st_bfe_ctrl_t* * **const** p_ctrl *st_bfe_cfg_t* **const** * const p_cfg	Configures the BFE device (stack) by writing into all configuration registers. It extracts the necessary data from the control **p_ctrl** and configuration **p_cfg** structures.
* p_reset	*st_bfe_ctrl_t* * **const** p_ctrl *e_bfe_reset_type_t* type	Resets the BFE. Several predefined reset options can be set with the **type** input parameter.
* p_modeSet	*st_bfe_ctrl_t* * **const** p_ctrl *e_bfe_mode_t* mode	Forces the BFE to enter a mode or state set with the **mode** input parameter from a predefined list of modes.
* p_modeRead	*st_bfe_ctrl_t* * **const** p_ctrl *e_bfe_mode_t* * **const** p_mode	Reads the current BFE mode. The result is returned into a variable pointed by **p_mode**.
* p_commTest	*st_bfe_ctrl_t* * **const** p_ctrl	Tests communication between the MCU and single or multiple BFE devices. If communication cannot be established, an error is returned accordingly.
* p_selfDiagnostic	*st_bfe_ctrl_t* * **const** p_ctrl *e_bfe_diag_option_t* option	Runs a self-diagnostic test for the BFE. Several predefined diagnostic options can be set with the **option** input parameter.
* p_memoryCheck	*st_bfe_ctrl_t* * **const** p_ctrl *e_bfe_mem_check_option_t* option	Runs memory tests inside a BFE for corrupted registers and data. Several predefined memory test options can be set with the **option** input parameter.
* p_vPackGet	*st_bfe_ctrl_t* * **const** p_ctrl *uint32_t* * **const** p_value *uint8_t* trigger *uint8_t* read	Acquires the battery pack voltage with the BFE. The result is returned into a variable pointed by **p_value**. The returned data is converted into voltage. When using the Boolean input parameters **trigger** and **read**, the function can be configured to trigger a measurement or read the results only or both.
* p_iPackGet	*st_bfe_ctrl_t* * **const** p_ctrl *bfe_i_pack_meas_t* * **const** p_values *uint8_t* trigger *uint8_t* read	Acquires the battery pack current with the BFE. The result is returned into a structure pointed by **p_values**. The returned data is converted into current. When using the Boolean input parameters **trigger** and **read**, the function can be configured to trigger a measurement or read the results only or both.
* p_vCellsGet	*st_bfe_ctrl_t* * **const** p_ctrl *bfe_vcell_meas_t* * **const** p_values *uint8_t* trigger *uint8_t* read	Acquires the voltages of all cells in the battery pack. The result is returned into a structure pointed by **p_values**. The returned data is converted into voltage. When using the Boolean input parameters **trigger** and **read**, the function can be configured to trigger a measurement or read the results only or both.

Table 2. BFE API Structure and its Fields (Cont.)

typedef struct st_bfe_api		
Member	**Parameters**	**Description**
* p_temperaturesGet	*st_bfe_ctrl_t* * **const** p_ctrl *bfe_temp_meas_t* * **const** p_values *uint8_t* trigger *uint8_t* read	Acquires the temperatures with the BFE. The result is returned into a structure pointed by **p_values**. The returned data is converted into temperature and voltage. When using the Boolean input parameters **trigger** and **read**, the function can be configured to trigger a measurement or read the results only or both.
* p_allGet	*st_bfe_ctrl_t* * **const** p_ctrl *bfe_all_meas_t* * **const** p_values *uint8_t* trigger *uint8_t* read	Acquires all measurable parameters with the BFE. The returned data is converted into the relevant units. When using the Boolean input parameters **trigger** and **read**, the function can be configured to trigger a measurement or read the results only or both.
* p_otherGet	*st_bfe_ctrl_t* * **const** p_ctrl *bfe_other_meas_t* * **const** p_values *uint8_t* trigger *uint8_t* read	Acquires custom parameters with the BFE. The result is returned into a structure pointed by **p_values**. The returned data is converted into the relevant units. When using the Boolean input parameters **trigger** and **read**, the function can be configured to trigger a measurement or read the results only or both.
* p_faultsAllRead	*st_bfe_ctrl_t* * **const** p_ctrl *bfe_faults_t* * **const** p_faults	Reads all fault registers of the BFE. The fault data is returned into a structure pointed by **p_faults**.
* p_faultsCheck	*st_bfe_ctrl_t* * **const** p_ctrl	Checks the BFE for faults. It monitors the relevant fault pin or checks a fault status register.
* p_faultsAllClear	*st_bfe_ctrl_t* * **const** p_ctrl *uint8_t* * **const** p_success	Attempts to clear all faults in the BFE. The result is returned into a Boolean variable pointed by **p_success**.
* p_cellBalanceControl	*st_bfe_ctrl_t* * **const** p_ctrl *bfe_cb_cfg_t* * **const** p_bal_cfg *e_bfe_process_ctrl_t* ctrl_option	Configures and controls cell balancing process in the BFE. The cell balancing configuration parameters are set in a structure pointed by **p_bal_cfg**. The process is controlled by the input parameter **ctrl_option**.
* p_isCellBalancing	*st_bfe_ctrl_t* * **const** p_ctrl	Checks if cell balancing is in progress in the BFE.
* p_continuousScanControl	*st_bfe_ctrl_t* * **const** p_ctrl *bfe_scan_cont_cfg_t* * **const** p_scan_cfg *e_bfe_process_ctrl_t* ctrl_option	Controls scan continuous function of the BFE. The scan continuous configuration parameters are set in a structure pointed by **p_scan_cfg**. The process is controlled by the input parameter **ctrl_option**.
* p_wdControl	*st_bfe_ctrl_t* * **const** p_ctrl *bfe_watchdog_ctrl_t* * **const** p_options	Controls watchdog timer function in the BFE. The watchdog timer parameters are set in a structure pointed by **p_options**.
* p_fetControl	*st_bfe_ctrl_t* * **const** p_ctrl *uint8_t* group_num *e_bfe_fet_state_t* c_fet_state *e_bfe_fet_state_t* d_fet_state	Controls the power FETs of the BFE. The set of power FETs is selected with the input parameter **group_num**. The charge and discharge FETs state is controlled with the input parameters **c_fet_state** and **d_fet_state**.
* p_fetRead	*st_bfe_ctrl_t* * **const** p_ctrl *uint8_t* group_num *e_bfe_fet_state_t* * **const** p_c_fet_state *e_bfe_fet_state_t* * **const** p_d_fet_state	Reads the states of the power FETs of the BFE. The set of power FETs is selected with the input parameter **group_num**. The states of the charge and discharge FETs are returned in the variables, pointed by **p_c_fet_state** and **p_d_fet_state**.
* p_gpioControl	*st_bfe_ctrl_t* * **const** p_ctrl *bfe_gpio_ctrl_t* * **const** p_options	Controls the GPIO pins of the BFE. The pin parameters are set in a structure pointed by **p_options**.

Table 2. BFE API Structure and its Fields (Cont.)

typedef struct st_bfe_api		
Member	**Parameters**	**Description**
* p_registerRead	*st_bfe_ctrl_t* * **const** p_ctrl *bfe_register_t* * **const** p_register	Reads a register in the BFE. The register address, value and other device specific parameters are contained inside the structure pointed by **p_register**.
* p_registerWrite	*st_bfe_ctrl_t* * **const** p_ctrl *bfe_register_t* * **const** p_register	Writes in a register of the BFE. The register address, value and other device specific parameters are contained inside the structure pointed by **p_register**.

The error code enumeration contains multiple error codes, systemized into groups. When no error is present, the functions return BFE_SUCCESS. Any other constant indicates an unnormal behavior and can be used for debugging of the code during development or for advanced fault management.

Table 3. BFE Error Codes Enumeration

typedef enum e_bfe_err			
Group	**Constant**	**Value**	**Description**
No error	BFE_SUCCESS	0	No error was returned.
General Errors	BFE_ERR_ASSERTION	1	A critical assertion has failed.
	BFE_ERR_INVALID_POINTER	2	The pointer points to invalid memory location.
	BFE_ERR_INVALID_ARGUMENT	3	There is an invalid input parameter.
	BFE_ERR_INVALID_NUMARGUMENTS	4	Invalid number of arguments.
	BFE_ERR_INVALID_REGISTER	5	The target register does not exist.
	BFE_ERR_READ_ONLY_REGISTER	6	The register cannot be written.
	BFE_ERR_INVALID_STATE	7	The selected state is invalid.
	BFE_ERR_UNMATCHED_REGISTERS	8	MCU and device registers do not match.
	BFE_ERR_UNSUPPORTED_FEATURE	9	Feature or function is not supported by the BFE.
	BFE_ERR_DEVICE_BUSY	10	The device is currently busy.
	BFE_ERR_DEVICE_NOT_INITIALIZED	11	The device has not been initialized.
	BFE_ERR_NOT_ENOUGH_MEMORY	12	The caller has not assigned enough memory.
	BFE_ERR_REGISTER_RESET_UNMATCHED	13	The expected reset value does not match the actual register value.
	BFE_ERR_INV_REG_SIZE	14	The register size is invalid.
	BFE_ERR_NONSUPPORTED_MODE	15	The selected device mode is unsupported.
	BFE_ERR_INVALID_OPERATION	16	The operation is not allowed.
	BFE_ERR_INVALID_CELL	17	Invalid cell number.
	BFE_ERR_WRITE_VERIFY	18	Write command verification error.
	BFE_ERR_SCAN_CNTR	19	The scan command was not received.

Table 3. BFE Error Codes Enumeration (Cont.)

typedef enum e_bfe_err			
Group	**Constant**	**Value**	**Description**
Comm. Specific Errors	BFE_ERR_COMM_TIMEOUT	21	The communication timeout exceeded.
	BFE_ERR_MSG_BUF	22	Message buffer overflow.
	BFE_ERR_COMM_FAULT	23	Communication error.
	BFE_ERR_RESPONSE	24	The response is invalid.
	BFE_ERR_CRC_INCORRECT	25	An incorrect CRC in the data array.
	BFE_ERR_COMM_NONSUPPORTED_INTERFACE	26	The selected communication interface is unsupported.
Device Specific Errors	BFE_ERR_SLEEP	30	Device sleep error.
	BFE_ERR_WAKEUP	31	Device wake up error.
	BFE_ERR_EEPROM	32	The EEPROM data is corrupted.
	BFE_ERR_ADC	33	ADC error.
	BFE_ERR_MUX	34	Multiplexer error.
	BFE_ERR_CELL_BALANCE	35	Cell balancing error.
	BFE_ERR_INVALID_DEV_ADDRESS	36	The device address is invalid.
	BFE_ERR_INVALID_CONF	37	The BFE configuration is invalid.
	BFE_ERR_MEAS_TIMEOUT	38	Measurement timeout is exceeded.
	BFE_ERR_FET_CONTROL	39	Power FETs control error.
Daisy-Chain Specific Errors	BFE_ERR_NA_STANDALONE	50	Not available in stand-alone operation.
	BFE_ERR_DAISY_CHAIN_COMM	51	Error in daisy chain communication.
	BFE_ERR_RESPONSE_TIMEOUT	52	Response is not received within the expected timeout.
	BFE_ERR_ACK	53	Acknowledge was not received.
	BFE_ERR_NAK	54	Not Acknowledge is received.
	BFE_ERR_IDENTIFICATION	55	The stack identification has failed.
FSP Specific Errors	BFE_ERR_FSP	61	There is an error in the FSP layer.

All API functions use the control structure **st_bfe_ctrl** as a parameter (Table 4). It contains status parameters of the BMS modified by some functions but also pointers to other structures like configuration, registers container or device information. The control structure can be extended to include device-specific parameters like timings or other. The extension is defined in the BFE implementation. The pointer to the configuration structure provides access of all API functions to the BFE configuration parameters and interface settings.

The detection of a fault results in setting the flag **is_fault_detected**. It does not return any error code by the function detecting it. It is a good practice to check the state of **is_fault_detected** flag after calling a function that might detect such a condition. Refer to the description of the API function to check if is the function can set this fault flag.

Table 4. Members of the BFE Control Structure

typedef struct st_bfe_ctrl		
Member	**Type**	**Description**
is_initialized	uint8_t	Flag that indicates if the BFE is initialized
is_low_power	uint8_t	Flag that indicates if the BFE is in low power mode
is_balancing	uint8_t	Flag that indicates if the BFE is balancing cells
is_scan_continuous	uint8_t	Flag that indicates if the BFE is continuously scanning
is_fault_detected	uint8_t	Flag that indicates if the BFE has detected any fault
* p_bfe_info	void	Pointer to the BFE information structure
* p_bfe_regs	void const	Pointer to the BFE registers' container
* p_cfg	const st_bfe_cfg_t	Pointer to BFE configuration settings
* p_extend	void	Pointer to BFE specific control parameters

The device configuration derives from the structure **st_bfe_cfg** (Table 5). It contains both BFE and API interface settings, related to BFE functionality which are universal to the whole RAA489xxx family. The configuration structure is extended to include the device-specific settings, which are defined in the implementing entity. The structure is used as a parameter of the setup function, but the control structure provides a pointer to it, so that functions can access it if necessary.

Table 5. Members of the BFE Configuration Structure

typedef struct st_bfe_cfg		
Member	**Type**	**Description**
* p_cells_select	const uint32_t	Pointer to a constant that indicates which cells exists in the battery pack
* p_temps_select	const uint16_t	Pointer to a constant that indicates which temperature inputs of the BFE
cells_in_series	const uint16_t	The total number of cells in series in the battery pack.
cells_in_parallel	const uint8_t	The total number of cells in parallel in the battery pack.
stack_size	const uint8_t	The number of stacked BFEs in the battery pack.
peripheral_type	const e_bfe_comm_interface_t	Constant from a predefined list of options that denotes the communication interface used between the BFE and the MCU
driver_cfg	const e_bfe_driver_cfg_t	Constant from a predefined list of options that denotes the configuration of the power FETs driver
fet_cfg	const e_bfe_fet_cfg_t	Constant from a predefined list of options that denotes the configuration of the power FETs controlling charge and discharge
* p_extend	void const	Pointer to BFE specific configuration settings

To be compatible with multiple BFEs, the API declares some parameters as void. Table 6 shows the declaration of those arguments. The instancing implementing the API for each BFE must then define these parameters. You can find the definition of each type either in the comment above the each typedef in file **bfe/r_bfe_api.h,** or in the actual the function definition inside file **bfe/r_raa489xxx.c**. the type definition of structures containing measured values contain values representing physical values. The current values are expressed in Amperes, voltages in Volts, and temperatures in Celsius rather than raw ADC or register values.

Table 6. Typedefs Redefined in the Instance Implementation

Type	Description
bfe_register_t	Device register container type. Must be redefined as a structure type whose members correspond to all BFE registers. Used for member type in the BFE control structure.
bfe_i_pack_meas_t	Pack current measurement data structure type. Must be redefined as a structure type whose members correspond to returned measured current (charge and discharge) and other current related parameters of the BFE. Used for input parameter type in (*p_iPackGet).
bfe_vcell_meas_t	Cells voltage measurement data structure type. Must be redefined as a structure type whose members correspond to returned measured cell and pack voltages for the current BFE. Used for input parameter type in (* p_vCellsGet).
bfe_temp_meas_t	Temperatures measurement data structure type. Must be redefined as a structure type whose members correspond to returned measured temperatures for the current BFE. Used for input parameter type in (*p_temperaturesGet).
bfe_all_meas_t	All inputs measurement data structure type. Must be redefined as a structure type whose members correspond to all returned measured data for the current BFE. Used for input parameter type in (* p_allGet).
bfe_other_meas_t	Other measurements data structure type. Must be redefined as a structure type whose members correspond to device specific measurement. Used for input parameter type in (* p_otherGet).
bfe_faults_t	BFE faults' structure type. Must be redefined as a structure type whose members correspond to the current BFE fault parameters. Used for input parameter type in (* p_faultsAllRead).
bfe_watchdog_ctrl_t	Watchdog timer control structure type. Must be redefined as a structure type whose members are used for the BFE watchdog timer functionality control. Used for input parameter type in (* p_wdControl).
bfe_cb_cfg_t	Cell balancing configuration structure type. Must be redefined as a structure type whose members are used for configuration of the cell balancing functionality for the current BFE. Used for input parameter type in (* p_isCellBalancing).
bfe_scan_cont_cfg_t	Continuous scan configuration structure type. Must be redefined as a structure type whose members are used for configuration of the scan continuous functionality of the current BFE. Used for input parameter type in (* p_continuousScanControl).
bfe_gpio_ctrl_t	GPIO control structure type. Must be redefined as a structure type whose members are used for control of the GPIOs of the current BFE. Used for input parameter type in (* p_gpioControl).
* extendApi_t	API interface extension structure type. When used, must be redefined as a structure type whose members are custom functions, additional to the already existing in the interface.

Figure 3 demonstrates the use of void types and their definitions in API functions. When a function has a parameter with void type definition, a variable with the redefined type from the particular BFE implementation must be declared in advance. In the example, the parameter is an array of structures with a size matching the BFE stack size with RAA489204. Pointing to a wrong variable type or size might result in memory violation and unexpected behavior. Therefore, it is important to pass a pointer pointing to the correct variable type.

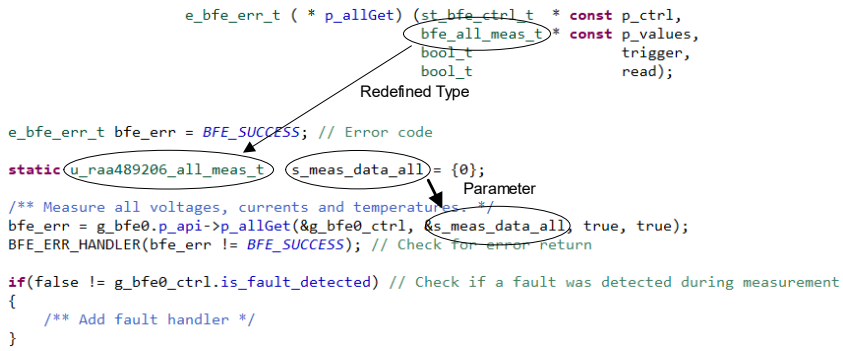

Figure 3. Using Function Parameters with Redefined Type

The API function pointed by **(*p_reset)** has fixed reset options set by the parameter type and listed in the reset types enumeration. Table 7 shows the content of the enumeration. You can select which parts of the BFE to reset. Keep in mind that you have to check which reset options are supported in the particular BFE implementation, otherwise BFE_ERR_INVALID_ARGUMENT error is returned by the function.

Table 7. BFE Reset Types Enumeration

typedef enum e_bfe_reset_type	
Constant	**Description**
BFE_RESET_TYPE_SOFT	Reset only the digital part.
BFE_RESET_TYPE_HARD	Reset both the digital and analog parts.
BFE_RESET_TYPE_TOIDLE	Reset the BFE to idle state.

The BFE modes and states are fixed and they are listed in the BFE States and Modes Enumeration (Table 8). The enumeration is used as parameters *mode* and *p_mode* types of the API functions pointed by **(*p_modeSet)** and **(*p_modeRead)** that set or return the current BFE mode. If the calling code specifies a state or mode that is not supported by the instance, **(*p_modeSet)** returns BFE_ERR_INVALID_ARGUMENT.

Table 8. BFE States and Modes Enumeration

typedef enum e_bfe_mode	
Constant	**Description**
BFE_STATE_RESET	Initial state when all circuits and oscillators are off.
BFE_MODE_POWER_UP	BFE power-up state.
BFE_MODE_IDLE	The device is ready waiting for a task to be executed.
BFE_STATE_SCAN	Single execution of measurements.
BFE_MODE_AUTOSCAN	Periodic execution of measurements.
BFE_MODE_LOW_POWER	The BFE low power mode.
BFE_MODE_SHIP	Lowest power consumption suitable for long-term storage.
BFE_MODE_POWER_DOWN	BFE shut down state.

The API function pointed by **(*p_selfDiagnostic)** has fixed diagnostic options set by the parameter *option* and listed in the BFE Diagnostic Options Enumeration. Table 9 shows the content of the enumeration. The function returns BFE_ERR_INVALID_ARGUMENT if the diagnostic options are not supported by the instance.

Table 9. BFE Diagnostic Options Enumeration

typedef enum e_bfe_diag_option	
Constant	**Description**
BFE_FULL_TEST	Run a complete self-test.
BFE_TEST_ADC	Test the ADC.
BFE_TEST_MUX	Test the multiplexer.
BFE_TEST_CB	Test cell balancing circuit.
BFE_TEST_OW	Check for open wires.
BFE_TEST_CUSTOM	Device specific test option.

The API function pointed by **(*p_memoryCheck)** has fixed memory test options set by the parameter *option* and listed in the BFE Memory Check Options Enumeration. Table 10 shows the content of the enumeration. You can select between different memory tests of BFE. *Note:* Check which memory test options are supported in the particular BFE implementation, otherwise BFE_ERR_INVALID_ARGUMENT error is returned by the function.

Table 10. BFE Memory Check Options Enumeration

typedef enum e_bfe_mem_check_option	
Constant	**Description**
BFE_CHECK_CONF_REGS	Verify content of configuration registers.
BFE_CHECK_EEPROM	Verify content of EEPROM memory.
BFE_CHECK_DEF_VALS	Check registers for default values.
BFE_CHECK_CUSTOM	Device specific memory check option.

The API functions pointed by **(*p_cellBalanceControl)** and **(*p_continuousScanControl)** have fixed process control options set by the parameter *ctrl_option* and listed in the BFE Process Control Enumeration. Table 11 shows the content of the enumeration.

Table 11. BFE Process Control Enumeration

typedef enum e_bfe_process_ctrl	
Constant	**Description**
BFE_PROCESS_ENABLE	Start a specific process.
BFE_PROCESS_INHIBIT	Stop a specific process.
BFE_PROCESS_RECONFIGURE	Reconfigure a specific process

The API function pointed by **(* p_fetControl)** controls the battery pack power FETs using the parameters *c_fet_state* and *d_fet_state* having fixed values, listed in the BFE FET State Options Enumerations (Table 12).

Table 12. BFE FET State Options Enumeration

typedef enum e_bfe_fet_state	
Constant	**Description**
BFE_FET_ON	The FET is conducting.
BFE_FET_OFF	The FET is not conducting.

The BFE configuration structure **st_bfe_cfg** has the following members with fixed values: *peripheral_type*, *driver_cfg*, and *fet_cfg*. The BFE interface supports multiple communication interface options which are listed in BFE Communication Interface Options Enumeration (Table 13). If the selected communication interface is not supported in the particular BFE implementation, the initialization function pointed by **(* p_initialize)** returns BFE_ERR_COMM_UNSUP_INTERFACE error message.

Table 13. BFE Communication Interface Options Enumeration

typedef enum e_bfe_comm_interface	
Constant	**Description**
BFE_COMMUNICATION_INTERFACE_SPI	Serial Peripheral Interface is used.
BFE_COMMUNICATION_INTERFACE_SCI	Serial Communication Interface is used.
BFE_COMMUNICATION_INTERFACE_I2C	Inter-Integrated Circuit communication interface is used.
BFE_COMMUNICATION_SINGLE_WIRE	Single wire communication interface is used.

In the battery pack, there are multiple options about the power FETs position and interconnection, depending on the battery power lines architecture and the BFE in use. The available ones are listed in Table 14 and Table 15. If an unsupported option is selected for the particular BFE implementation, the initialization function pointed by **(*p_initialize)** returns a BFE_ERR_UNSUPPORTED_MODE error message.

Table 14. BFE Power FETs Driver Configuration Enumeration

typedef enum e_bfe_driver_cfg	
Constant	**Description**
BFE_DRIVER_HIGH_SIDE	FETs are disconnecting the positive terminal of the battery pack.
BFE_DRIVER_LOW_SIDE	FETs are disconnecting the negative terminal of the battery pack.
BFE_DRIVER_NONE	The BFE is not driving any power FETs.

Table 15. BFE power FETs Configuration Enumeration

typedef enum e_bfe_fet_cfg	
Constant	**Description**
BFE_FET_CONFIG_SERIES	Battery pack has one set of terminals for charge and discharge.
BFE_FET_CONFIG_PARALLEL	Battery pack has separate terminals for charge and discharge.
BFE_FET_NONE	The BFE is not driving any power FETs.

3.2 BFE Interface Instance Structure

The instance structure encapsulates all the structures necessary to use a module implementation:

- Pointer to the control structure
- Pointer to the configuration structure
- Pointer to the API structure

In the Battery Abstraction Layer, the interface instance structure is defined in **bfe/r_bfe_api.h** and the instance itself is declared in **bal_data.c**. By using the name of the declared instance, you can easily access any API function or member of the control and configuration structures (such as *g_bfe0.p_api->p_setup* or *g_bfe0.p_ctrl->is_initialized*).

```
/** This structure comprises everything that is needed to use an instance of this
 * interface. */
```

```
typedef struct st_bfe_instance
{
    st_bfe_ctrl_t       * p_ctrl;     ///< Pointer to the control structure for
                        ///< this instance

    st_bfe_cfg_t  const * p_cfg;      ///< Pointer to the configuration structure
                        ///< for this instance

    bfe_api_t     const * p_api;      ///< Pointer to the API structure for this
                        ///< instance
} bfe_instance_t;
```

3.3 Using the BFE Interface

3.3.1 Preparation

To operate a Battery Front End device, the Middleware Battery Abstraction Layer module is needed. It contains both the API instance and its implementation for the particular BFE. You must begin with connection of the Middleware Battery Abstraction Layer module with the low-level Hardware Abstraction Layer of the MCU. In e^{2}studio when working with the 32-bit ARM MCUs from the Renesas RA Family, this process uses the benefits of the Flexible Software Package by connecting to the HAL drivers through their API. You must make sure that all required peripheral modules are available.

3.3.2 Configuration

Then the BFE API itself must be configured in **r_bfe/r_bfe_cfg.h**. The file contains pre-processor macros used for controlling certain features by enabling/disabling parts of the code. The following example demonstrates a part of such code configuration for the stackable BFE RAA48206.

```
/** Functions check input parameters: '0' - Disable/ '1' - Enable (Recommended). */
#define BFE_CFG_PARAM_CHECKING_EN           (1)

/** Register verification after write command: '0' - Disable/ '1' - Enable
(Recommended). */
#define BFE_REG_WRITE_VERIFY_ENABLE         (1)

/** Configuration Register verification after write command: '0' - Disable/ '1' -
Enable (Recommended). */
#define BFE_CFG_REG_WRITE_VERIFY_EN         (1)
```

For the available options, refer to the comment sections and keep in mind that some of the definitions accept Boolean values (such as check the arguments of the functions or automatically read back and verify register content after write command).

The next step before calling the API functions is to enter the settings for the BFE in the configuration structures, declared in **bal_data.c**. *Note:* The members of **st_bfe_cfg** and its extension are constant variables that are initialized during declaration and cannot be further modified in the code. It is considered that those settings are related to the hardware and for safety reasons do not change them during execution of the code (such as number and position of the battery cells in the pack or open-wire scan current duration that depends on the input filters time-constant). On other hand, the control structure is also declared in the same file but only initial values are assigned to its members as the can be further modified. You can find instructions about what values can be assigned to all structures' members in the comment sections of the type definitions of those structures in **src/r_bfe/r_bfe_api.h** and **src/r_bfe/r_bfe_raa489xxx.h**. Some of the variable types are enumerations with fixed constants. This approach facilitates the device configuration and minimizes the risk of errors by providing a list of

options from which you can select. However, keep in mind that assigning a value outside those constants lead to a fault return or unexpected behavior.

```
/** Extended configuration structure */
const st_raa489206_ext_cfg_t g_bfe0_ext_cfg =
{
/* Shunt Resistors */
/** Shunt resistance for charge and discharge currents [uohm] */
.r_ipack_shunt_uohm = 5000,
/** Shunt resistance for the external voltage regulator [mohm] */
.r_ireg_shunt_mohm = 3300,

/* Fault Thresholds and Delays */
/** Short-circuit threshold [mA] (range: ((-40.176...-321.33mV) /
r_ipack_shunt_uohm) * 1000000) */
.th_short_circuit_a = -40000,
/** Short-circuit delay [us] (range: 250...4219us) */
.delay_short_circuit_us = 1000U,
/** Discharge overcurrent threshold [mA] (range: ((-0.005...-342.95mV) /
r_ipack_shunt_uohm) * 1000000) */
.th_discharge_overcurrent_a = -20000,
/** Charge overcurrent threshold [mA] (range: ((0.005...342.95mV) /
r_ipack_shunt_uohm) * 1000000) */
.th_charge_overcurrent_a = 4000,

....

};

/* Configuration structure */
const st_bfe_cfg_t g_bfe0_cfg =
{
.p_cells_select = &g_bfe0_cells_cfg,     ///> Do not modify!!!
.p_temps_select = &g_bfe0_ext_temps_cfg, ///> Do not modify!!!
.cells_in_series = 10U,                                ///> Set the number of cells
in series
 .cells_in_parallel = 1U,                                   ///> Set the number of
cells in parallel
.stack_size = RAA489206_MAX_STACK_SIZE,  ///> Do not modify!!!
.peripheral_type = BFE_COMMUNICATION_INTERFACE_SPI, ///> Select an interface
.driver_cfg = BFE_DRIVER_HIGH_SIDE, ///> Select a FET driver configuration
.fet_cfg = BFE_FET_CONFIG_SERIES,                       ///> Select a FET
configuration
.p_extend = &g_bfe0_ext_cfg,             ///> Do not modify!!!*/
};

/* Control structure */
st_bfe_ctrl_t g_bfe0_ctrl =
{
.is_initialized = false,              ///> Do not modify!!!
.is_low_power = false,                ///> Do not modify!!!
.is_balancing = false,                ///> Do not modify!!!
.is_cont_scanning = false,            ///> Do not modify!!!
.is_fault_detected = false,           ///> Do not modify!!!
```

```
.p_bfe_info = &g_bfe0_info,            ///> Do not modify!!!
.p_bfe_regs = &g_raa489206_registers,  ///> Do not modify!!!
.p_cfg = &g_bfe0_cfg,                  ///> Do not modify!!!
.p_extend = &g_bfe0_ext_ctrl,          ///> Do not modify!!! };
```

3.3.3 Use of API Functions

Figure 4 show an example of using the API function for cell balancing pointed by **(* p_cellBalanceControl)**. The code has the following parts:

- API function which is called in the high-level application code. The error return of the function is assigned to the variable bfe_err. Then its value is checked to verify the successful execution.
- Before calling the API function a structure, used for parameter, is declared. It contains the cell balancing settings for the particular BFE implementation. You can refer to **src/r_bfe/r_bfe_raa489xxx.h** to find more information about its members.
- The cell balancing process is controlled with a parameter having predefined constants (Table 11). They are universal and can be found in the interface file **src/r_bfe/r_bfe_api.h**.
- Some members of the cell balancing configuration structure have predefined values. They are defined in enumerations that can be found in **src/r_bfe/r_bfe_raa489xxx.h**.
- The body of the API function contains control procedures, communication drivers, and other dedicated static functions. They can be used to create own purpose-specific function.

Figure 4. Using the API interface functions

3.3.4 Examples

This is a basic example for initializing the interface, running setup and measuring cell voltages with RAA489206:

```
e_bfe_err_t bfe_err = BFE_SUCCESS; // Error code

static u_raa489206_vcells_meas_t s_meas_data_cells = {0};

/** Initialize the Battery Front End interface. */
bfe_err = g_bfe0.p_api->p_initialize&g_bfe0_ctrl, &g_bfe0_cfg);
BFE_ERR_HANDLER(bfe_err != BFE_SUCCESS); // Check for error return

/* Configure the Battery Front End. */
bfe_err = g_bfe0.p_api->p_setup(&g_bfe0_ctrl, &g_bfe0_cfg);
BFE_ERR_HANDLER(bfe_err != BFE_SUCCESS); // Check for error return

/* Measure all cell voltages. */
bfe_err = g_bfe0.p_api->p_vCellsGet(&g_bfe0_ctrl, &s_meas_data_cells, true, true);
```

```
BFE_ERR_HANDLER(bfe_err != BFE_SUCCESS); // Check for error return
```

This example demonstrates the change of state with RAA489206:

```
e_bfe_err_t bfe_err = BFE_SUCCESS; // Error code

/** Put the BFE in Low Power Mode. */
bfe_err = g_bfe0.p_api->p_modeSet (&g_bfe0_ctrl, BFE_MODE_LOW_POWER);
BFE_ERR_HANDLER(bfe_err != BFE_SUCCESS); // Check for error return

/* Send a wake-up command to the BFE. */
bfe_err = g_bfe0.p_api->p_modeSet(&g_bfe0_ctrl, BFE_MODE_IDLE);
BFE_ERR_HANDLER(bfe_err != BFE_SUCCESS); // Check for error return
```

This example demonstrates the fault detection and faults read with RAA489206:

```
e_bfe_err_t bfe_err = BFE_SUCCESS; // Error code

static st_raa489206_faults_t     s_faults_data = {0};

/** Check for faults. */
bfe_err = g_bfe0.p_api->p_faultsCheck(&g_bfe0_ctrl);
BFE_ERR_HANDLER(bfe_err != BFE_SUCCESS); // Check for error return

/** Check if a fault was detected */
if(false != g_bfe0_ctrl.is_fault_detected)
{
    /** Read the faults. */
    g_bfe0.p_api->p_faultsAllRead (&g_bfe0_ctrl, &s_faults_data);
    BFE_ERR_HANDLER(bfe_err != BFE_SUCCESS); // Check for error return
}
```

For more examples, please, refer to the Software Manuals of the API Implementations for the specific BFE devices.

4. Revision History

Revision	Date	Description
1.01	Mar 20, 2025	Updated Tables 1, 2, 4, 5, 6, 12, and 13. Updated Figure 3. Removed Table 11 and associated text. Consolidated Tables 13 and 14. Updated Configuration section. Updated Examples section.
1.00	Oct 18, 2022	Initial release.

IMPORTANT NOTICE AND DISCLAIMER

RENESAS ELECTRONICS CORPORATION AND ITS SUBSIDIARIES (“RENESAS”) PROVIDES TECHNICAL SPECIFICATIONS AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT OF THIRD-PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for developers who are designing with Renesas products. You are solely responsible for (1) selecting the appropriate products for your application, (2) designing, validating, and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. Renesas grants you permission to use these resources only to develop an application that uses Renesas products. Other reproduction or use of these resources is strictly prohibited. No license is granted to any other Renesas intellectual property or to any third-party intellectual property. Renesas disclaims responsibility for, and you will fully indemnify Renesas and its representatives against, any claims, damages, costs, losses, or liabilities arising from your use of these resources. Renesas' products are provided only subject to Renesas' Terms and Conditions of Sale or other applicable terms agreed to in writing. No use of any Renesas resources expands or otherwise alters any applicable warranties or warranty disclaimers for these products.

(Disclaimer Rev.1.01)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact Information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit www.renesas.com/contact-us/.

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.