48V Mobility Solution

1. What is the US069-EVK?

The US069-EVK 48V mobility solution is a proof of concept (POC) that has all the electrical building blocks of a higher voltage (36V-48V) power train system. The solution consists of 4 boards. One motherboard that has the MCU (RX23T) used as the main controller for the system. The RX23T is responsible for setting up and monitoring the battery front end (BFE) (ISL94216), monitoring and controlling the chargers (ISL81801 & P9415) and driving the inverter board that drives the motor. The inverter board mates with the motherboard to make the motor control block. This is the 2nd board for the solution. The 3rd board of the solution is a 60W wireless power receiver that uses two 30W P9415 receivers connected in parallel. The board with transmitter sources 60W continuous power to the input of the charger. The wireless power board is an optional feature to the solution. The 4th board, which is optional, enables Bluetooth connectivity to the solution. A RX23W Bluetooth MCU target board communicates with the motherboard by way of an I2C port. The RX23W communicates with a client (a phone or tablet) to control and to receive updates from the system. The mobile app is provided and is coded for Android systems. The figure below is the block diagram for the system described.

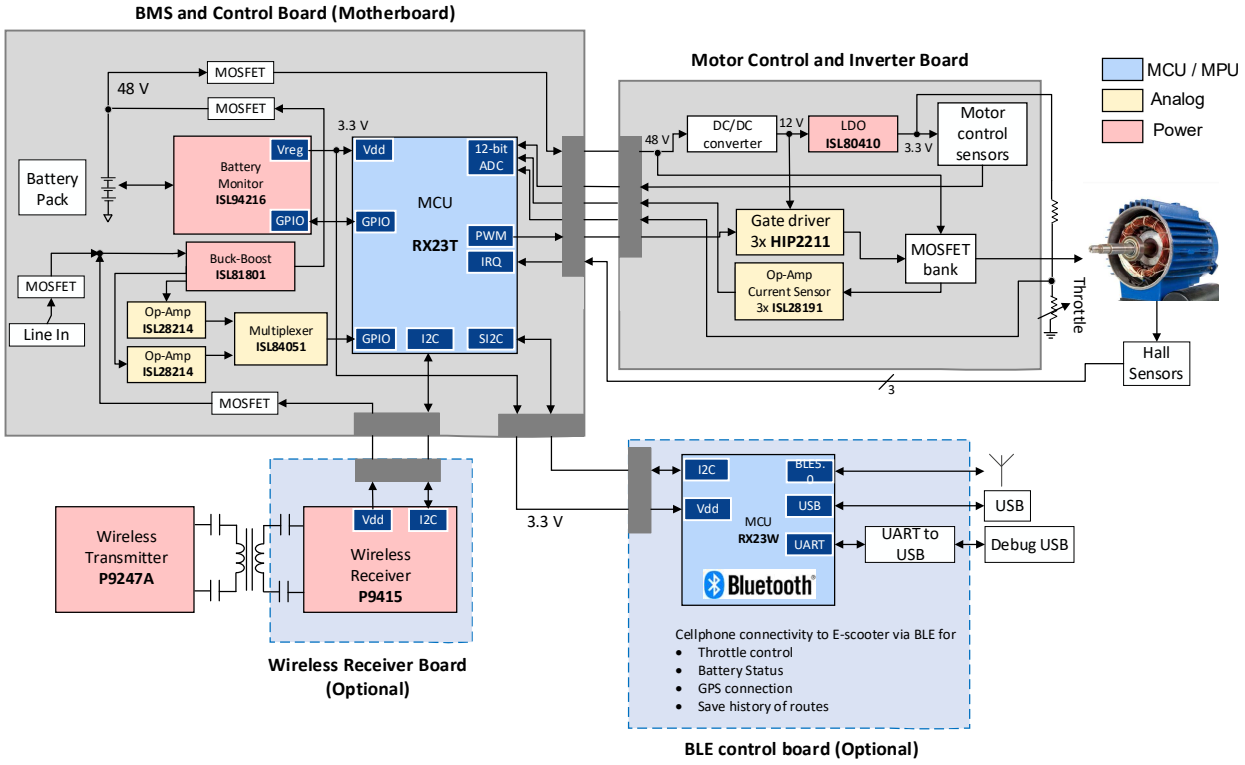

Figure 1-1: US069 System Block Diagram System Flow

2. Table of Contents

3. Objective

This solution is a starting platform that provides the user with hardware and preconfigured software to begin developing the end solution whether it be a scooter or an HVAC solution. The code is modularized like the hardware to be able to add and subtract features with relative ease. The code provided is preconfigured to spin a motor using a resistive throttle. The table below is the starting configuration for US069.

PARAMETER	SPECIFICATION/ CONFIGURATION
BATTERY PACK	14 – Cell 48V
WIRED CHARGER	12V-65V input / 2A Continuous*; 10A for fast charging; **Use Fan or Heat sink for higher charging current*
WIRELESS CHARGER	20V Input / 1A @60V output
MOTOR CONTROL	Powers 1600W BLDC with Hall Sensors
RX23T TO RX23W COMMUNICATIONS	I2C
WIRELESS CLIENT APPLICATION	Android Application

Table 1 US069 Starting Hardware and Software Configuration

The US069 supports other hardware configurations. These configurations and the changes needed are discussed within this document.

This document discusses the solution in terms of a system, hardware, and software. Controls in the software are explained by actions or responses in the hardware and how it impacts the overall system.

Hover over and click over title and pages to maneuver around the document.

What should US069 be used for?

US069 is to be used as a starting platform for users to develop their end use case. The user should experiment and utilize the base code and routines to control the end application. This solution works with the Renesas RX family of microcontrollers with e^2 Studio (Eclipse) IDE.

What is US069 NOT!

The US069 is not a **turnkey** solution to production. The solution does not cover all errors cases. The user needs to take ownership of the hardware and software to determine which error cases need coverage.

4. The Boards

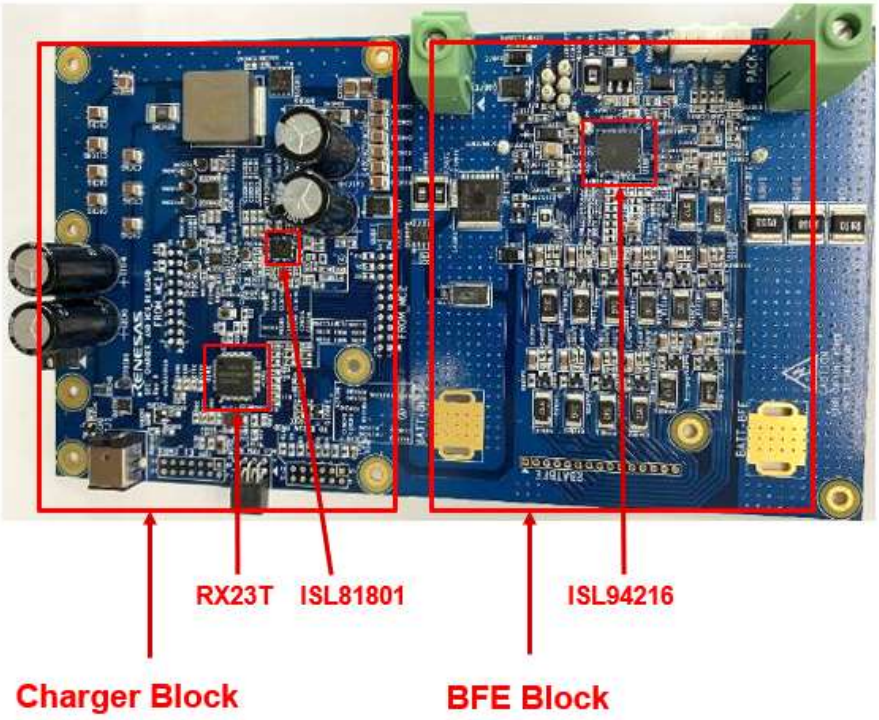

Figure 4-1: Motherboard (BFE, Charger, And MCU)

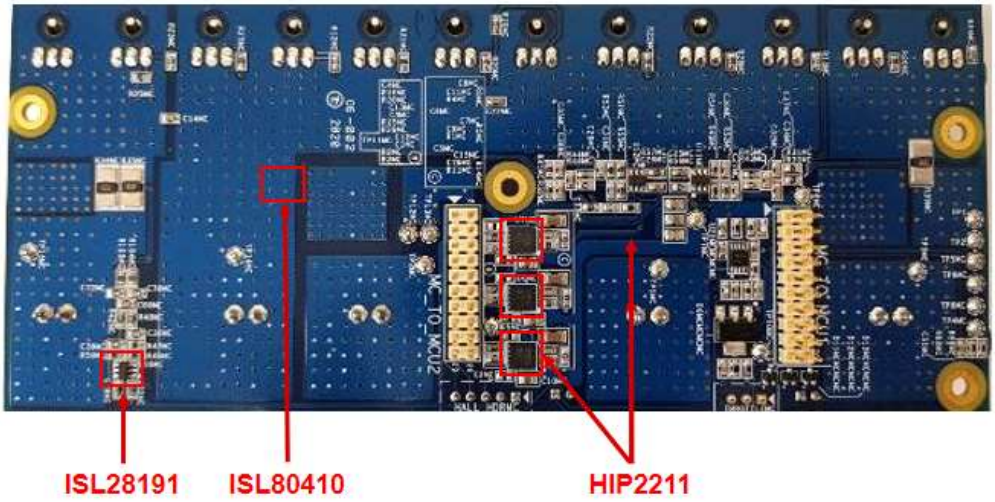

Figure 4-2: Inverter Board (Motor Control)

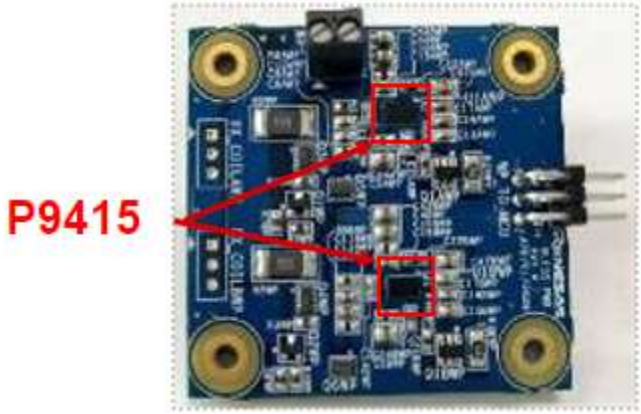

Figure 4-3: Wireless Power Board

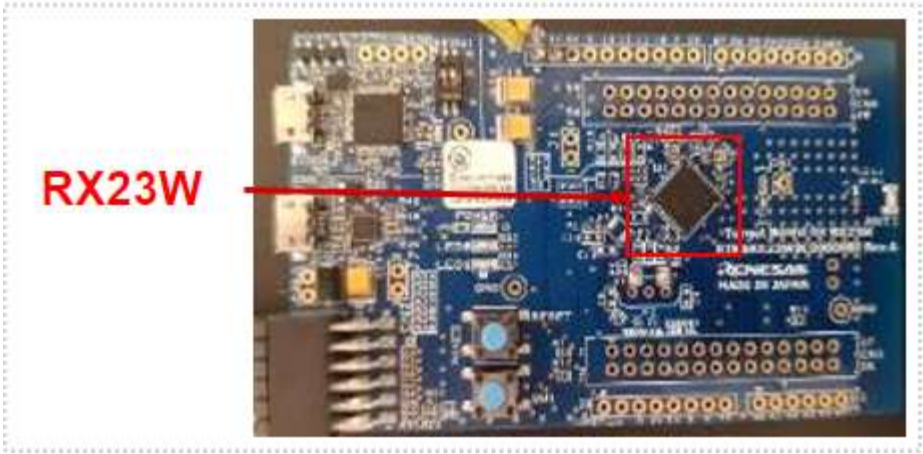

Figure 4-4: RX23W Target Board (Bluetooth)

5. What is needed to begin?

E2 Emulator – https://www.renesas.com/us/en/software-tool/e2-emulator-programming-function for debugging the motherboard and programming the MCU.

Mini-USB cable to debug the RX23W target board

Download e^2 Studios for RX - https://www.renesas.com/us/en/software-tool/e-studio This is the programming and debugging environment for both RX23T and RX23W.

A power supply that can source 3.3V ~100mA

Table 2 are wired connections needed for the system to operate.

WIRED CONNECTIONS	RECOMMENDATION
MC_PACK+ TO PACK_+, MC_PACK- TO PACK- (INVERTER BOARD TO MOTHER BOARD)	Low Gauge Wire
WP TO CHR (WIRELESS POWER TO MOTHER BOARD)	Wire that can pass 3A
WIRE AND CONNECTOR FOR WIRED CHARGER INPUT	Wire that can pass the charge current to the charger.
WIRE AND CONNECTOR FOR BATT+BFE AND BATT-BFE	XT60 connector Low gauge wire
WIRE AND CONNECTOR FOR 2BATTBFE ***CELL 14 CONNECTS TO PIN 15 (2BATTBFE)*** ***CELL0 CONNECTS TO PIN 1 (2BATTBFE)***	https://www.jst-mfg.com/product/detail_e.php?series=199 Or a standard 0.1in pitch header If using wire 1A max
WIRE THAT CONNECTS THE I2C OF THE RX23W TO THE *TO_RX23W_PMOD* CONNECTOR	General sized wire is good

Table 2: Hardware connections and components required

6. Setting up the programming environment

Install e^2 Studio in the windows PC. A free time-based evaluation license is provided for a first time install.

Download the latest motherboard source code from www.renesas.com.

Import the archived project (File → Open Project from File System) to the workspace. A dialog box appears Figure 6-1.

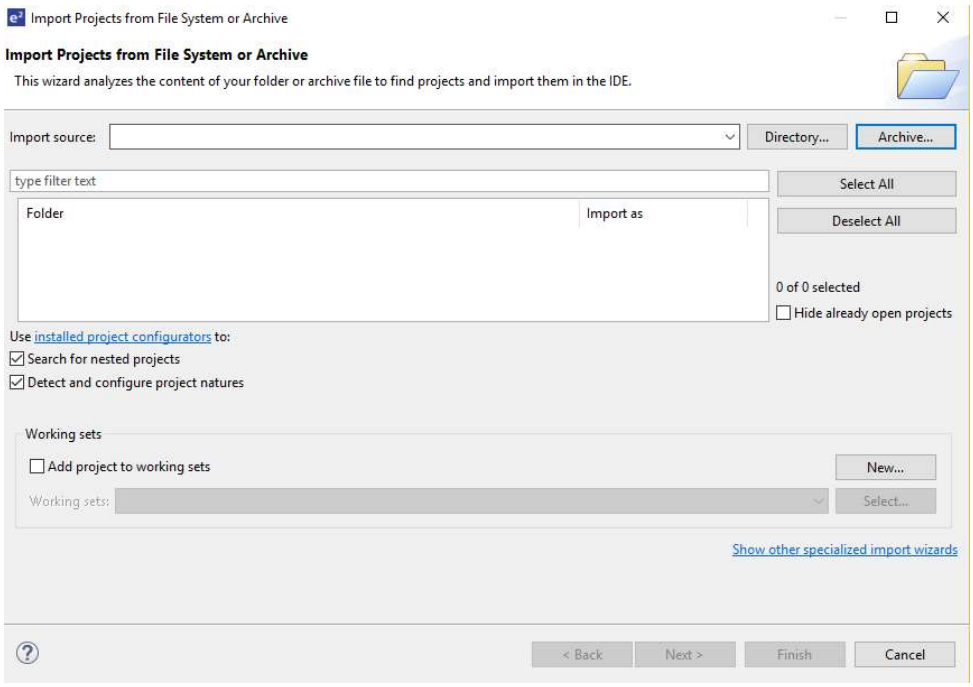

Figure 6-1: Dialog box to import an archive project

After importing, the project is shown in the Project Explorer tab in the main window. Right-click on the active project in Project Explorer, choose Properties. In the window, select C/C++ Build → Settings. Choose the Toolchain tab. Choose the correct Toolchain compiler (CCRX version xx.xx). Press Apply. See Figure 6-2.

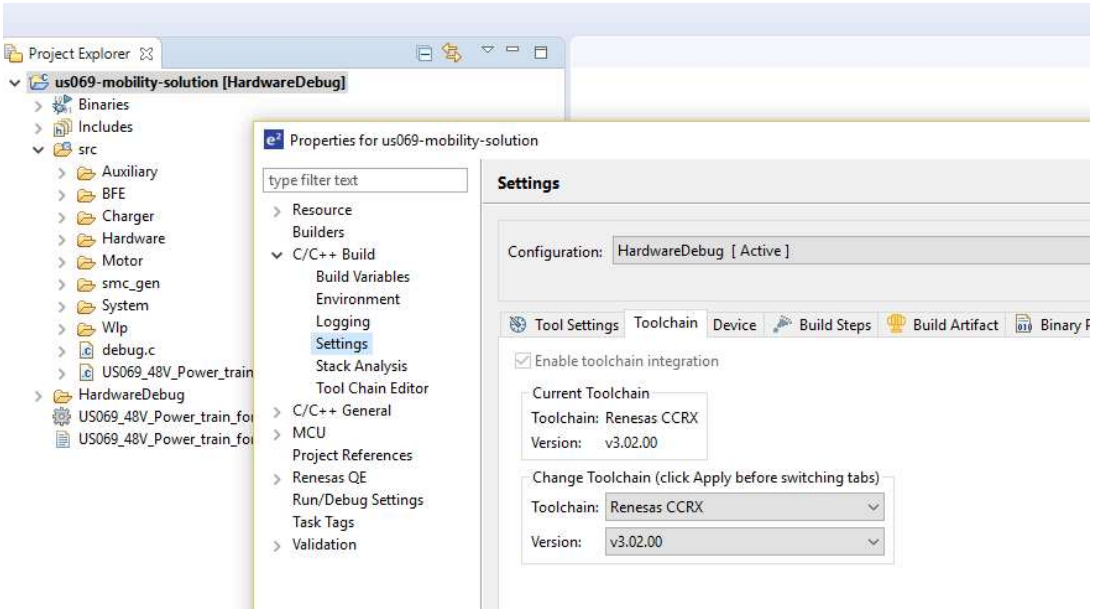

Figure 6-2: Setting the compiler (toolchain)

Then click on the Tool Settings tab. Select Converter → Output. Check the Output hex file box. Then choose Intel Hex Format File for output file type. (Figure 6-3)

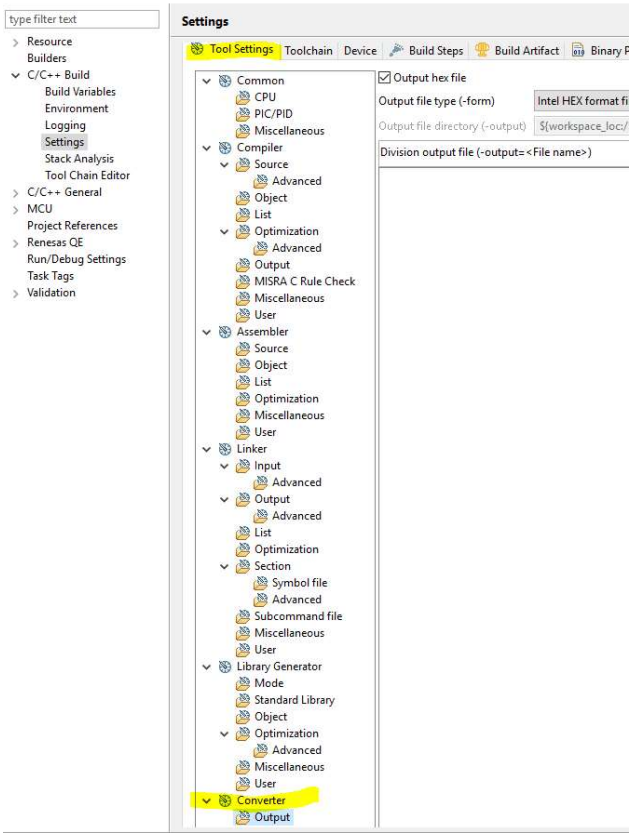

Figure 6-3: Set the format of the compiled code

In the left-most column of the window, choose C/C++ General → Indexer. The top 5 boxes starting with Enable Project Specific Settings should be checked. Press Apply and Close. (Figure 6-4)

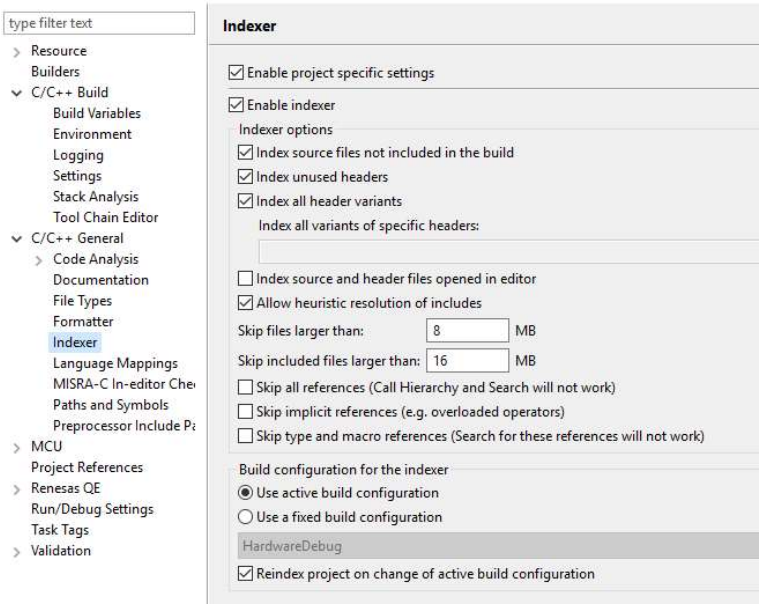

Figure 6-4: Setting up the indexer

Under the Run menu, choose Debug Configurations. Select the .x file under the Renesas GDB Hardware Debugging. Select the Debugger tab followed by the connection settings. Under the Power drop down, the *Power Target from The Emulator* should read a NO setting. The Target device is the R5F523T5. The Debug Hardware is E1 (RX) or E2. Apply and Close window. (Figure 6-5)

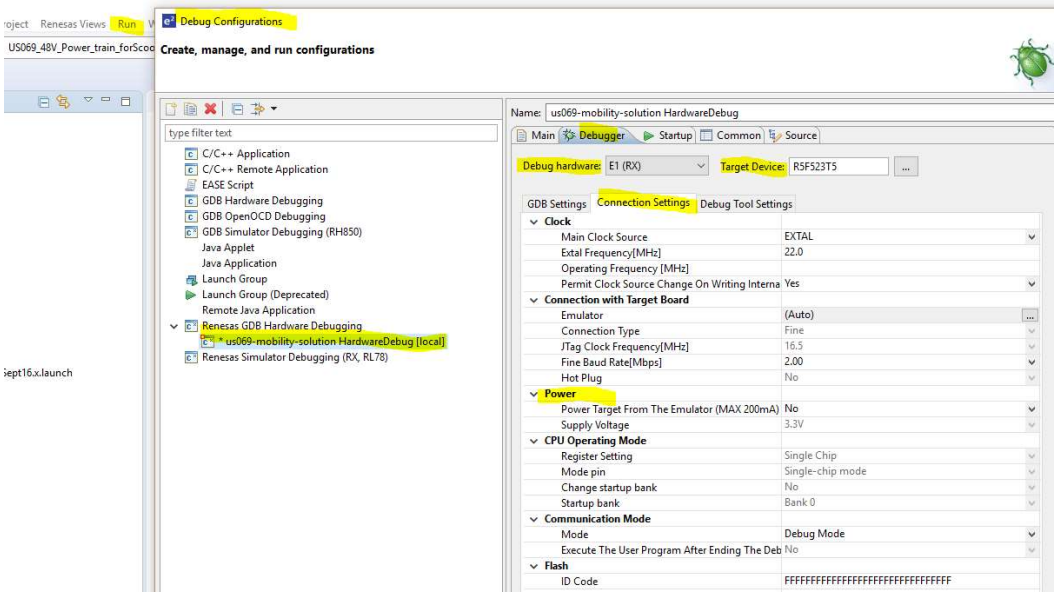

Figure 6-5: Setup the debug environment

Under the Project menu in the main e^2 Studios window, choose Clean. This does a clean build.

The sequence of actions described above quickly enables the debug environment. More details are found in Renesas Application notes that exist outside the document.

6.1. Debugging Environment Shortcut Keys and Useful tips

SHORTCUT KEYS	ACTION
SELECT A LINE THEN CTRL +R	The code will run to line without requiring a break point.
F6	Step Over a Routine
F5	Step Into a Routine
F8	Run Code or Run to next breakpoint
F11	Start Debugger
SELECT A VARIABLE OR ROUTINE THEN F3	Goes to the definition
ALT+LEFT	To back out of a routine
CTRL + H THEN FILE SEARCH TAB	Find all the instances where a routine or phrase is called. Displayed in console window. Double click the result to go the location.

Table 3: e^2 Studio Shortcut Keys for Debugging

6.2. Debugger Hardware Setup

With BFE, Charger, and MCU (motherboard) disconnected from the battery, connect the inverter board to the motherboard. Using a low gauge wire connect PACK- to MC_PACK- and PACK+ to MC_PACK+.

Connect 2BAT2BFE header to the battery pack. The header connects the individual cells of the pack to the monitoring and balancing circuitry of the ISL94216. Next, connect BAT+_BFE and BAT-_BFE to the positive and negative connection of the packs. This connection powers the BFE and allows large currents to flow to and from the pack.

Without communication with the MCU, the BFE's strong regulator will turn ON for 5s before the device places itself into LOW POWER mode. After which, the device periodically turns on for 20ms every 2s to make some system voltage measurements. The strong regulator is activated when the measurements are occurring. The mode is observed by probing the VDDBFE signal. The VDDBFE (the regulator that powers the RX23T) rises to 3.3V while measurements are being made. While the chip is in LOW POWER mode the regulation voltage reduces to around 2V. If there is too much of a load on the regulator, the pulse is not observed, and the measurements are not made.

Connect the emulator to a computer and the 14-pin header to the E1_EMRX header on the motherboard.

A 3.3V power supply is required to connect to the VDDBFE signal with the initial connection between the battery and the BFE present. Connect the battery first, then make the connection between VDDBFE and the external 3.3V. The system grounds are located at test points DGNDBFE and PACK-BFE.

An unpowered emulator results in a heavy load to the 3.3V power supply.

The 3.3V power supply can be removed after the RESET state has been executed in code (see Reset State pg.20).

6.3. Quick Start Up

A Battery Emulator (resistor divider; MCB_PS3_Z) board has been provided to simulate the battery. The board allows for execution of the code through *sys_update()*. Connect the emulator board (red) to the motherboard as shown in Figure 6-6.

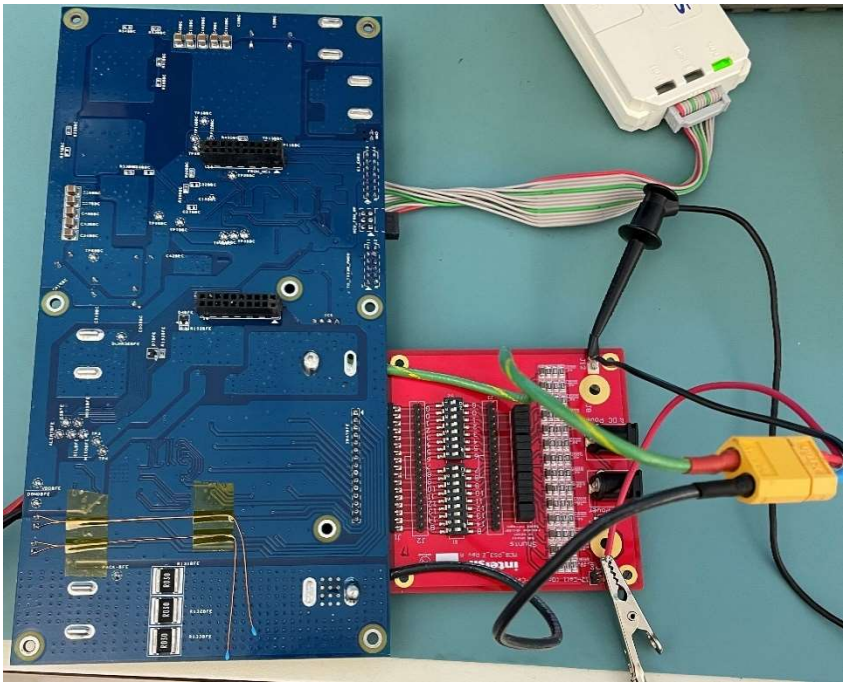

Figure 6-6: Battery Emulator Board Connection to the Motherboard.

The board is connected 180degrees from the motherboard component side. Connect the emulator board such that pin 17 and pin 1 are open. See Figure 6-7. Connect the positive terminal of the power supply to J11 of the emulator board and BATT+BFE of the motherboard. Connect the negative terminal of the power supply to J12 of the emulator board and BATT-BFE of the motherboard.

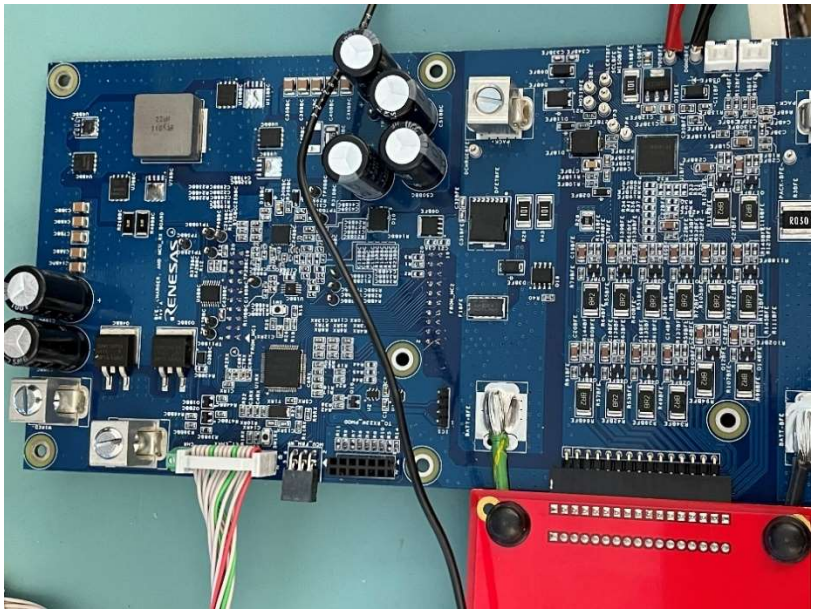

Figure 6-7: Top Side Motherboard Connected to Emulator Board.

Program the supply from 48V to 58.8V. Connect a 3.3V voltage source to the VDDBFE and DGNDBFE test points on the motherboard. Power the supplies and launch the debug environment.

7. System Code Organization

The code and hardware are largely modularized. The high-level structure of the code is shown in Figure 7-1

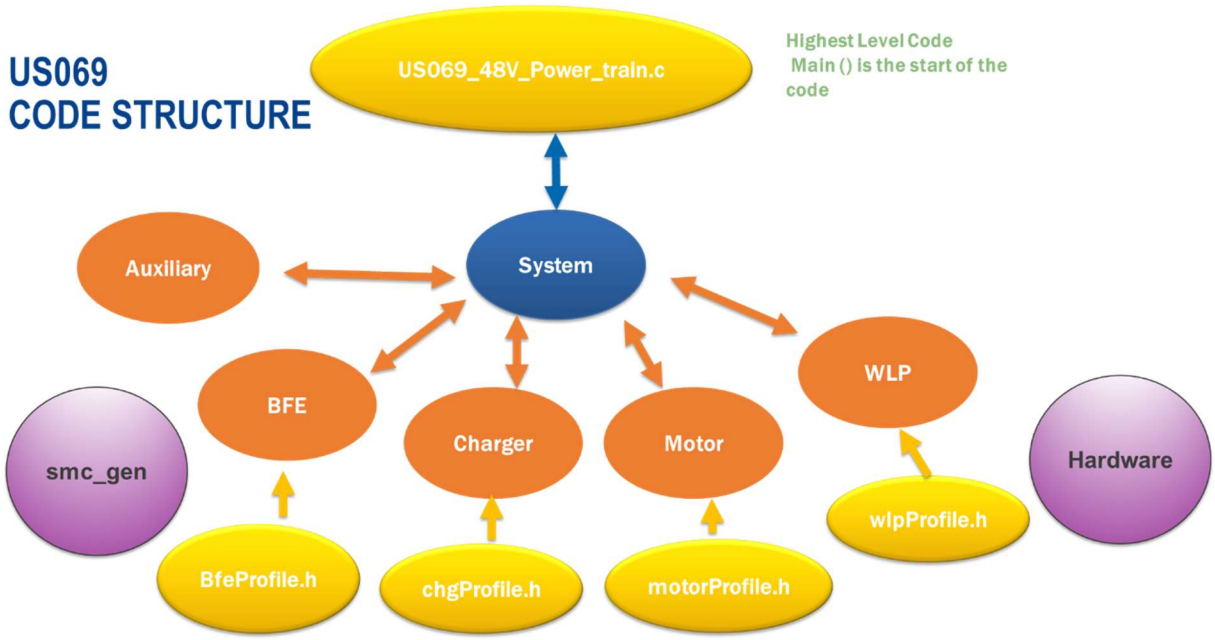

Figure 7-1: US069: High Level code structure

The code mostly follows the structure in the illustration above. The blue, orange and purple bubbles are folders that contain header (.h) and program (.c) files supporting the folder description.

FOLDER	DESCRIPTION OF FUNCTION CALLS
AUXILIARY	The detection and the I2C communication between the motherboard and the Bluetooth device.
BFE	The routines to control and read from the ISL94216
CHARGER	The routines that detect the wired charger and control the ISL81801 charger
MOTOR	Routines that control and monitor the motor
WLP	Routines that control and detect the wireless power receiver
HARDWARE	Low-level code used to set up and read from the ADC of the MCU
SMC_GEN	Low level code for MCU peripheral setup

Table 4: A Brief Description of Each coding folder

The code in the low-level folders is accessible throughout the project. The arrows that indicate the interaction between modules are largely true. There are routines called that directly access the modules that arrows do not show in the illustration.

The profile header files are used to configure each module to the end application. Within these files, there are thresholds setting and characterization parameters that determine the behavior of the module within the system.

US069_48V_Power_train_xxx.c is where the program begins. It is the top level of the code.

8. System Overview

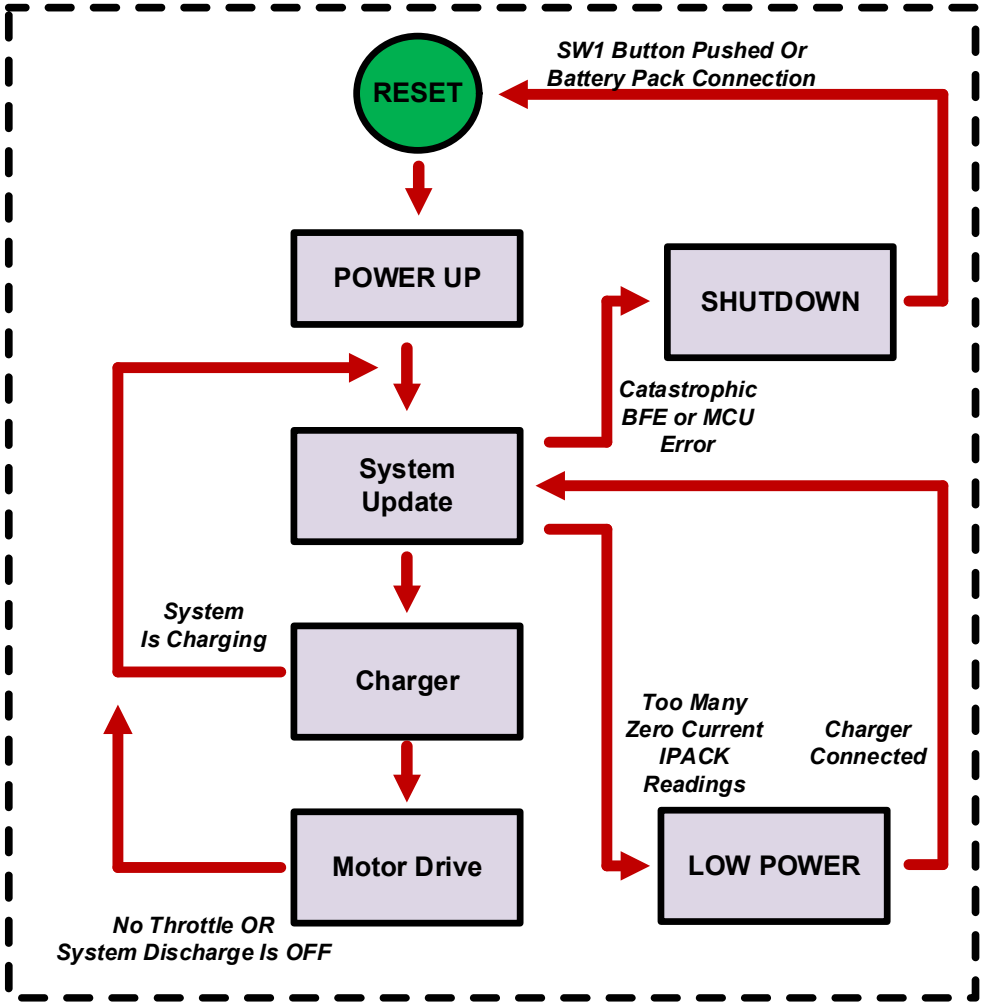

Figure 8-1: High-Level System Flow

The system is coded for a scooter use case or an application with a throttle and a motor.

8.1. RESET State

The RESET state is entered when either the battery pack completes the connection to the electronics or the battery front end (BFE) (ISL94216) is reset or the MCU (RX23T) is reset. The BFE is reset by pressing the SW1 button or sending a soft reset via a serial communication or by asserting the $\overline{RESET}$ GPIO pin that connects to the BFE. The MCU is reset by pressing the SW1RX button.

After the MCU either resets the BFE or the battery connection is reestablished the MCU configures the BFE regulator to strong. This configuration allows the sourcing of regulation currents larger than 80mA. The BFE strong regulator powers the MCU and the BFE. The regulation voltage is 3.3V.

The MCU begins the boot process when the BFEs regulation voltage rises above 2.7V.

8.2. POWER UP State.

The POWER UP state configures the MCU peripherals to a ready state. This state performs checks to test the health of the system while initializing the system.

At the entrance to the state, the MCU sends the BFE a soft reset to clear past BFE configurations. The BFE is then configured with the strong regulator always on. The strong regulator, which powers the MCU, is always enabled for all BFE modes including RESET and LOW POWER.

The state then initializes the global system variables, the BFE IC and the charger circuitry. The system checks the presence of the charger (wireless and wired) connections and the RX23W Bluetooth module. Status checks are made with each module.

8.3. SYSTEM UPDATE

The SYSTEM UPDATE state is called within a continuous loop that is the main loop for the system. The state measures and updates all system variables that are defined as part of the system register map. If the RX23W is present, a serial packet with updated system values is sent to the device.

Any errors while updating will transition the state to LOW POWER or SHUTDOWN.

8.4. CHARGER

The CHARGER state follows the SYSTEM UPDATE. The state monitors the charging ports and the charging status of the battery. It detects the connection of a charger to the system. The state prioritizes wired over wireless charging. The state detects and connects the charger while utilizing the autonomous charging features of the BFE.

While charging the battery, any faults that may occur or maintenance with the charger are serviced. The removal of a charger, or a battery full status from the BFE, or a charger error results in the charging circuit turning off and being reinitialized.

8.5. MOTOR DRIVE

The MOTOR DRIVE state is enabled when the g_sys_data.Sys_control.Bit.SystemONOFF = 1. The variable is either hard coded or set by Bluetooth communication from the mobile device (client) to the RX23W (server). During a system update, the RX23W communicates to the RX23T to change the bit state. The state energizes the motor inverter with the connection of the battery pack via the discharge FET of the BFE.

After connecting the inverter to the battery pack, the BFE is set to continuous scan mode where the IC periodically measures battery pack and system measurements. Any measurements that exceeds the testing threshold stops the scan. Voltage checks are made at each side of the discharge FET to validate the connection.

After initializing the motor setup, the system enters a continuous loop that exits when a flag (gExitMotorLoop) is set to TRUE or if the motor variable (g_sys_data.Sys_Status.Bit.MCPresent) state is set to NOT PRESENT. A throttle reading below the minimum voltage threshold sets the gExitMotorLoop flag to TRUE. The low throttle voltage results in the exit of the motor loop.

8.6. SYSTEM SHUTDOWN

The SYSTEM SHUTDOWN state is entered from the SYSTEM UPDATE state. The state is entered from any catastrophic faults from either the BFE or communications between the RX23T and the RX23W. A fault places the system in SHUTDOWN status. This is the lowest powered state of the system.

Reconnecting the battery pack and pressing the SW1 button exits the state to RESET state.

8.7. LOW POWER State

The LOW POWER state occurs when there have been too many successive no current IPACK readings by the BFE. The state is entered from the SYSTEM UPDATE state. The BFE FETs are turned off and the IC is placed in the LOW POWER mode. The MCU is also placed in a LOW POWER state.

The system exits the state when a charger is connected to the system.

9. The States in Relation to Hardware and Software

This section describes each state in more detail. Each state's description references key variables and routines used in the software to control the hardware.

9.1. Reset State

The *sys_startup()* in the US069_48V_Power_train_xxx.c file calls the RESET state. The routine is called from the main routine of the system. The MCU hardware is initialized followed by the activation of the SPI port.

A soft reset is sent to the BFE via a SPI command. This command initializes the register and state machines of the ISL94216. After a small wait, the strong regulator of the BFE is enabled full time from a SPI command. The strong regulator is able to source currents >80mA. For this application, the weak regulator (sourcing currents of ~1mA) is not applicable.

The weak/ strong regulator output can be read between test points VDDBFE and DGNDBFE or PACK-BFE). PACK-BFE is the ground for the system.

It is safe to have a 3.3V power supply connected to this regulator. It may be needed to start up the debug process (pg.10). Prior to stepping into the startup routine, the current is roughly 2mA to 4mA sourcing from the 3.3V supply connected to the regulator.

After turning ON the strong regulator and confirming the command has been received, the external power supply should be disconnected from the board. The regulator should be able to sustain the load. The regulator sourcing current can be measured from the BFE.

Table 5 lists the default RX23T pin setting after a reboot of the MCU. The MCU pins are configured with the system configurator (*US069_48V_Power_train_xxx.sfg*) and through text commands. Look in the Hardware folder for the ADC configuration and the smc_gen folder for other MCU resources.

MCU PIN Name	Net Name	Pin	Pin Config	Sub Block	Default	Description	Notes
P00	WP_INT	2	Input (INT) (NEDG)	Wireless Power	1	Interrupt from Wireless Power Block	IRQ02
P91	N_EN1	32	Output (N_EN1) (Open Drain)	Wireless Power	1	Enable P9415 (1) See truth table	Truth Table in WP block
P92	N_EN2	31	Output (N_EN2) (Open Drain)	Wireless Power	1	Enable P9415 (2) See truth table	Truth Table in WP block
PB2	SDA0	24	I/O (SDA) (Open Drain)	Wireless Power	1	I2C Comm pin	
PB1	SCL0	25	Output (SCL) (Open Drain)	Wireless Power	1	I2C Comm pin	

MCU PIN Name	Net Name	Pin	Pin Config	Sub Block	Default	Description	Notes
PB3	CHR_PG	23	Input	Charger	1	Power Good	
PB7	CHR_EN	17	Output Push Pull	Charger	0	Enables the charger (ISL81801)	
P11	CHR_AO	61	Input Analog	Charger		The analog input from the mux within the charger	An016/IRQ01
PA4	Ch_A_A0	64	Output Push Pull	Charger	0	Address 0 of the analog mux	
PA5	Ch_A_A1	65	Output Push Pull	Charger	0	Address 1 of the analog mux	
PB0	CHR_VO	26	Output PWM	Charger	1 (3.3V)	The pin is PWM'd to control the charger's output voltage	
PA3	CHR_IO	27	Output PWM	Charger	1 (3.3V)	The pin is PWM'd to control the charger's current clamp	

PD6	WL_CNTRL	13	Output Push Pull	Charger	0	Controls connection from Charger to Wireless Output	
PD7	W_CNTRL	12	Output Push Pull	Charger	0	Controls connection from Charger to Wired Output	

MCU PIN Name	Net Name	Pin	Pin Config	Sub Block	Default	Description	Notes
P30	N_CS	45	Output Push Pull	Battery Front End	1	SPI PORT Chip Select	
P22	MISOA	48	Input SPI	Battery Front End		SPI PORT Master In Slave Out	
P23	MOSIA	47	Output SPI	Battery Front End	0	SPI PORT Master Out Slave In	
P24	SCLA	46	Output SPI	Battery Front End	0	SPI PORT Serial Clock	
PB4	N_ALRT	21	Input INT	Battery Front End	1	Alert pin for BFE configure as IRQ3; may configure as POE in Future	
P31	N_WK	43	Output (Open Drain)	Battery Front End	HI-Z	Wake the 94216 up also used for charging	
P32	N_RST	41	Output (Open Drain)	Battery Front End	HI-Z	Reset the 94216;	
P33	FETSOFF	40	Output (Open Drain)	Battery Front End	HI-Z	Automatically turns off Power FETs when pin is asserted HI	This overrides the IC decision making for the FETs
P02	PRE_CHR	1	Output (Open Drain)	Battery Front End	HI-Z	Charges the motor control Cap prior to Lo impedance DFET turning ON	Only turn on max 200ms. 50ohm res will burn.

MCU PIN Name	Net Name	Pin	Pin Config	Sub Block	Default	Description	Notes

P01	N_CS1	4	Output Push Pull	TOP Rx23W	1	SPI PORT Chip Select for Motor Control	
PE2	NMI/Motor Stop	11	Input	TOP Rx23W/ MOTOR Control	1	Setup to stop the motor in a control way. Touch to GND exit motor loop	Non Maskable INT
PB6	SSDA5	18	I/O (SDA) (Open Drain)	TOP Rx23W	1	SDA pin for I2C comms to the Rx23W. The Rx23T is configured as a master	
PB5	SSCL5	19	Output (SCL) (Open Drain)	TOP Rx23W	1	SCL pin for I2C comms to the Rx23W. The Rx23T is configured as a master	
VCC	V3P3	10,20,42	Input	TOP Rx23W	0V	Digital power from BFE	
VSS	AGND	8,22,44,60	Input	TOP Rx23W	0V	Board ground	

MCU PIN Name	Net Name	Pin	Pin Config	Sub Block	Default	Description	Notes
P71	U_HI	38	Output MTIOC3B	Motor Control	Hi-Z	Motor Control MTIOC3B Timer	
P74	U_LO	35	Output MTIOC3D	Motor Control	Hi-Z	Motor Control MTIOC3D Timer	
P72	V_HI	37	Output MTIOC4A	Motor Control	Hi-Z	Motor Control MTIOC4A Timer	
P74	V_LO	34	Output MTIOC4C	Motor Control	Hi-Z	Motor Control MTIOC4C Timer	
P73	W_HI	36	Output MTIOC4B	Motor Control	Hi-Z	Motor Control MTIOC4B Timer	
P76	W_LO	33	Output MTIOC4D	Motor Control	Hi-Z	Motor Control MTIOC4D Timer	
P40	IU_AO	56	Input AN000	Motor Control		Analog Input Current Sense for U (AN000)	
P41	IV_AO	55	Input AN001	Motor Control		Analog Input Current Sense for V (AN001)	
P42	IW_AO	54	Input AN002	Motor Control		Analog Input Current Sense for W (AN002)	
P43	Vpk_AO	53	Input AN003	Motor Control		Analog Input Vpack sense (AN003)	

P44	VU_AO	52	Input AN004	Motor Control		Analog Input U voltage sense (AN004)	
P45	VV_AO	51	Input AN005	Motor Control		Analog Input V voltage sense (AN005)	
P46	VW_AO	50	Input AN006	Motor Control		Analog Input W voltage sense (AN006)	
P47	THERM MTR	49	Input AN007	Motor Control		Analog Input Thermistor voltage sense for motor (AN007)	
P10	V THROTTLE	62	Input AN017	Motor Control		Analog Input Throttle voltage for motor (AN017)	
AVCC0	MCU_VCC	57	Output	Motor Control	0V	Analog Power option to deliver power to the MC block	
P93	HU_IRQ	30	Input IRQ0	Motor Control	0	Hall Sensor Signal From U; IRQ0	
P94	HV_IRQ	29	Input IRQ1	Motor Control	0	Hall Sensor Signal From V; IRQ1	
PA2	HW_IRQ	28	Input IRQ4	Motor Control	0	Hall Sensor Signal From W; IRQ4	
P70	POE	39	Input Open Drain	Motor Control	1	POE; Over Current & braking detect. The motor is killed with excessive current and when either the front brake or rear brake is pressed	

Table 5: Default MCU pin settings for the RX23T

9.2. Power UP state

The routines *sys_init()* and *sys_check()* in the US069_48V_Power_train_xxx.c file defines the POWER UP state.

9.2.1. Sys_init()

Sys_init() initializes the system variables (*sys_var_init()*), the hardware (*sys_hardware_init()*) and the modules(*sys_modules_init()*). The system variables that are the default states when starting the program. The *g_sys_data* variables are part of a large structure (*sys_data_t*) that contains most of the variables in the System Register Map (pg.). The *sys_hardware_init()* configures the MCU features such as interrupts, ADCs and communication ports to the MCU default state. *Sys_modules_init()* writes the settings which include the IC's operational modes and thresholds for the IS94216.

The default settings are located in ISL94216.h. The *bfe_init()* is the routine that write setting to the IC. The Table 6 and Table 7 are the default settings for the ISL94216. More details for about registers settings are found in the ISL94216 datasheet on the Renesas website.

94216 Register Hex	Setting in Hex	Notes
0x01	0x00	
0x02	0xE0	
0x03	0xC2	
0x04	0xFC	
0x05	0xFF	
0x90	0xD1	
0x0E	0x8C	
0x11	0xC0	
0x12	0x30	
0x1B	0x51	
0x1F	0xCC	Disable Comm TO for debug
0x24	0x5C	Do a scan before turning on Cpmp
0x25	0xEF	
0x28	0x80	~504ms CBON
0x29	0xC0	~48ms CBOFF
0x2E	0x2B	LP time=8192loop; scan Delay = 256ms
0x83	0x00	
0x84	0x00	
0x85	0x3F	make 0x1E to see Busy Bit (Debug)
0x86	0xD7	
0x87	0xF0	
0x88	0x00	
0x89	0x00	

Table 6 The Default Register Settings for the ISL94216

BFE Var	ISL94216.h System Var	BFE Threshold
OV	VCELL_OV_INIT	4.2
UV	VCELL_UV_INIT	2.5
DVOV	VCELL_MAX_DELTA	0.5
Vcell Ave	VCELL_AVE	1
Vcell Flt dly	VCELL_FAULT_DELAY	2
Dut 0/1	DUT_0_1_TEMP_INIT	-20C
DOT 0/1	DOT_0_1_TEMP_INIT	55C
CUT 0/1	CUT_0_1_TEMP_INIT	0C
COT 0/1	COT_0_1_TEMP_INIT	40C
Etaux Dly	ETAUX_FAULT_DELAY	3
IOTW	IOTW_INIT	65C
IOTF	IOTF_INIT	85C
VBAT OV	VBAT_OV_INIT	59.5
VBAT UV	VBAT_UV_INIT	33.6
Other FLt Dly	OTHER_FAULT_DELAY	1
DSC Dlt	DSC_DELAY_INIT	1ms
DOC	DOC_INIT	-300mV (60A)
DOC Dly	DOC_DELAY_INIT	3 (0x20)
COC	COC_INIT	15mV (3A)
COC Dly	COC_DELAY_INIT	8 (0x7)
Ipack Ave	IPACK_AVE_INIT	1
dCb min	CB_MIN_DELTA_INIT	50mV
CbMAx	CB_MAX_INIT	4.23
CbMin	CB_MIN_INIT	3V
VEOC	V_EOC_INIT	4.18V
IEOC	I_EOC_INIT	325u (65mA)
IREG (Norm)OC	IREG_NORM_OC_INIT	165mV (50mA)
IREG (LP) OC	IREG_NORM_OC_INIT	165mV (50mA)

Table 7 Default Threshold Setting for The ISL94216

For more details about the ISL94216 (BFE) functions and settings, visit the datasheet at https://www.renesas.com.

9.2.2. Sys_check()

The *sys_check()* routine checks for the presence of the peripherals that can connect to the motherboard and does a status check of the motherboard.

Bfe_setup()

The bfe_setup(enableFaultHandle) does an initial scan of the battery pack, current and temperature. If the variable enableFaultHanldle is TRUE, a fault measured by the BFE will place the device into system shutdown. Prior to moving to system shutdown, all system variables are updated.

After a single scan is complete, the fault and status registers of the BFE (*bfeReadAllFaultsHelper()*) are reported to the Renesas Virtual Console (*Renesas View → Debug → Renesas Virtual Console*). These bit values quickly report the health of the system. The Table 8 are the bit definition for the fault and status bit of the BFE. The table is for reference. The datasheet should be referenced for true accuracy.

Register Address (Hex)	Register Name	Page	Bit Function								Power On Reset Value (Hex)	Type
			7 (MSB)	6	5	4	3	2	1	0 (LSB)		
10	EOC Current Threshold (IEOC)	82	Step 1.345mV; Range: 5.25µV to 342.95mV								00	R/W
System Faults and Indicators												
63	Priority Faults	83	VCCF	OWF	IOTF	COCF	DOCF	DSCF	UVF	OVF	00	R/W
64	ETAUX Faults	86	COT1	CUT1	DOT1	DUT1	COT0	CUT0	DOT0	DUT0	00	R/W
65	Other Faults	87	VBOVF	VBUVF	CPMP NRDY	OW xT1	OW xT0	OW V_{BAT1}	OW VSS	CRCF	20	R/W
66	CB Status	90	BAT FULL	IOTW	IEOC	VEOC	DVCF	2HI2CB	2LO2CB	NEED CB	00	R/W
67	Status	92	DCHRGI	CHRGI	CH PRESI	LD PRESI	OTHER FAULTS	IREG1	IREG2	VTMPF	00	R/W
68	Open-Wire Status (Two Bytes)	95	OW16	OW15	OW14	OW13	OW12	OW11	OW10	OW9	00	R/W
69			OW8	OW7	OW6	OW5	OW4	OW3	OW2	OW1	00	R/W

Table 8: Fault and Status bit definition of the ISL94216 (BFE)

Open wire faults (OWF) are common errors that occur while debugging. See registers 0x68 and 0x69 to determine which pin could have the open wire. If OW xT1 or OW xT2 are set, check if the thermistors are connected to THERM1BFE or THERM2BFE. If the BFE is connected to a power supply and resistor divider (cell emulator) and the voltage is too low, the UVF bit will set.

The setup routine updates all system registers measurement values after the single scan. The *bfe_update_monitor_data()* resides in debug.c. It converts the bfe measurement data to float values. Making it easier to understand the status of the system. The g_debug_data structure is where the float measurements are stored.

If there are no faults after setup, the charge pump is instructed to turn ON. This places the BFE in a state to turn on the CFET and DFET at will.

Chg_init()

The *chg_init()* routine uses the MCU GPIO pins to place the state of the charger in a non-active safe state. The net names in Table 5 are similar to the control names used to set GPIO pins. As an example, CHG_W_CNTRL control pin equals W_CNTRL in the table.

The W_CNTRL and WL_CNTRL control the PMOS ON state via and NPN resistor as shown in Figure 9-1. The charger's regulation voltage and current are controlled by PWM pin that makes a cheap DAC when connected to a low pass filter. See Figure 9-3. The pin's PWM frequency is reset, and the pins' functions stop in this routine.

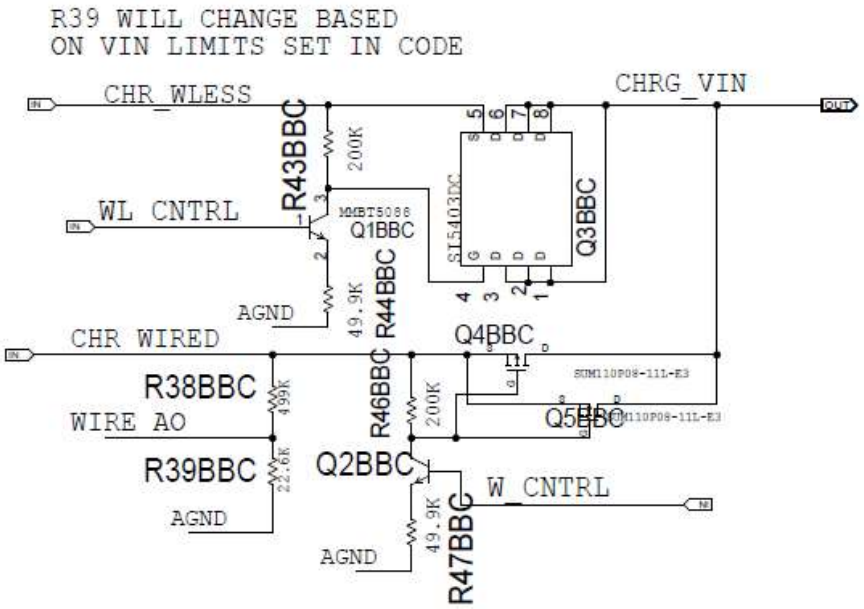

Figure 9-1: Control Circuitry That Selects Which Input Connects to the Charger

Wlp_connect_init()

The *wlp_connect_init()* routine re-initializes the wireless power port and connections to a safe and known state. Each P9415's enable pin is set to a high impedance state (1) to allow for the RX23T to read the pin's status in detecting if the P9415 is present. If the wireless transmitter and receiver are mated, the 5V regulator within the P9415 powers up and pulls up on the WLP_NENx pins (P91 and P92). When both pins read back a high (1), a connection from the wireless power receiver to MCU can proceed. CHG_WL_CNTRL is a control pin that connects the wireless power output (~20V) to the charger circuit through a PMOS. A low (0) setting turns off the NPN that connects to the PMOS gate. The result is a PMOS that is OFF. See Figure 9-1. All pins' names are similar to the net names in Table 5.

Chg_wired_charger_check()

The *chg_wired_charger_check()* routine reads the voltage at the wired charger input port (CHR_WIRED; Figure 9-1) to determine if a charger is connected to the port.

The code internally checks the voltage. The MCU, using the AN16 (P11) pin, reads the WIRE_AO (Figure 9-1) voltage by switching the multiplexer's input to it. If the voltage is greater than 10V, then the line present bit is set to PRESENT. A comparison is made to check if the wrong voltage charger is connected to the port. This sets the WPF (Table 12Table 12: System Variables) bit to high (1). The check closes out the routine by changing the pin to an interrupt and configuring the multiplexer to the original position.

Wlp_wireless_charger_check()

The system gives precedence to wired over wireless charging. The *wlp_wireless_charge_check()* is executed when the LineInPresent (wired charging) system variable (Table 12) is set to NOT_PRESENT.

The routine checks that status of the wireless receivers enable pins (WLP_NEN1 and WLP_NEN2) by placing the pins in a high impedance state and reading the digital pin back. If both pins read back a high (1), the output voltage of each P9415 is read by way of the I2C port. Each chip's voltage reading is compared to a range of output voltages defined in Wlp.h. The voltage comparison is to check if the correct version of the P9415 is soldered to the board. A failed comparison sets the WLPF_VOUT system variable to 1.

With the enable signals set to high (1), each P9415 output is not connected to the charger board. See Figure 9-2.

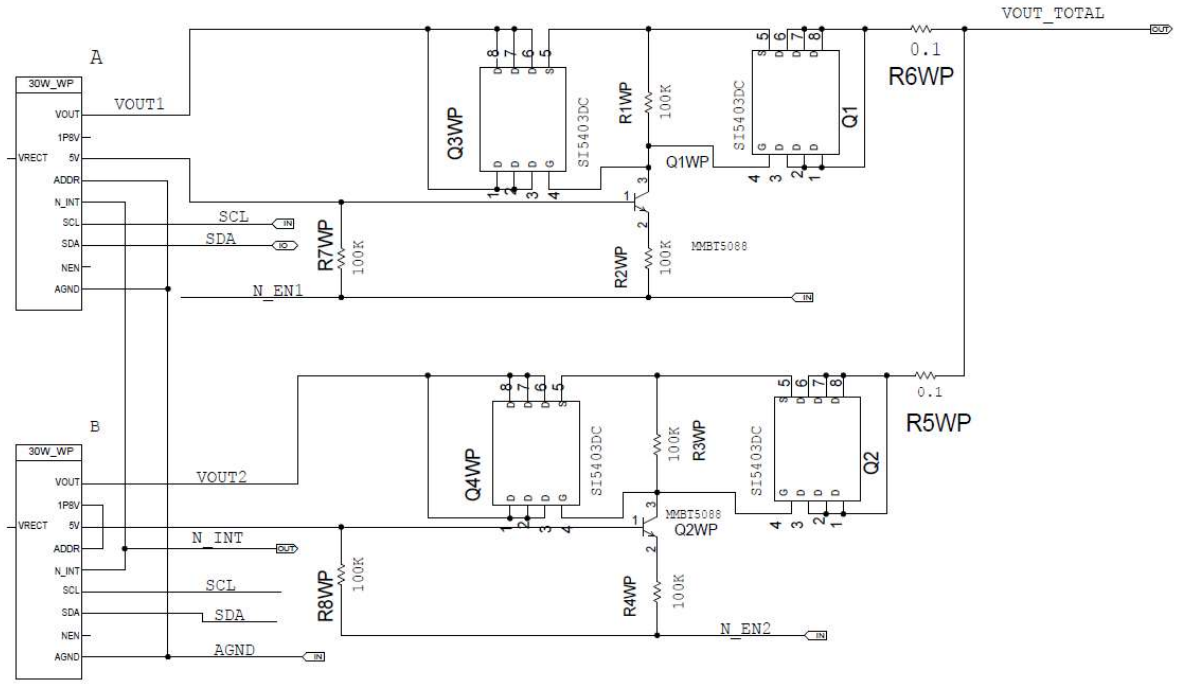

Figure 9-2: Interface Circuitry Between Wireless Power Receiver and Motherboard.

A current reading is made from each chip and compared to the max and min current threshold defined in Wlp.h. By default, there should be no current present. If the current exceeds the limits, systems variables are set with codes. Otherwise, the WLPPresent system variable is set to 1.

Auxi_check()

The *Auxi_check()* routine checks for the presence of the RX23W running the US069 code. The RX23W sets up the I2C port as a slave to the RX23T master. The slave address for the RX23W is 0xE2/3 (8-bits).

The code check of the RX23T PE2 pin is high (1). If the pin is low, this indicates an error from the RX23W. The AUXF system variable (Table 12Table 12: System Variables) is set to 1. The PE2 is set up as an input. The pin also serves to stop the motor and exit the motor routine.

If the RX23W is ok, the RX23T sends a 12-byte read command to the RX23W. The communication between the two MCUs has a Cyclical Redundancy Check (CRC) with every transaction. The CRC equation is compliant with CRC-CITT16 X25 protocol. If the slave address does not NAK and the CRC check passes and the data byte read back (byte 0) is 0x45, the RX23W is PRESENT. The system variable AuxPresent is set to PRESENT.

The code then waits for the RX23W (server) to connect to the client (Bluetooth device). The code monitors the first nibble of data byte 0 for each readback. If the value is something other than 0x5, then RX23W is connected to a client. Otherwise, the code stalls at this point.

9.2.3. Before the while loop

Before entering the endless while loop, global variables are defined. The *gCounter* is a generic unused counter used in experiments. The *gExitMotorLoop* is initiated to TRUE. The variable is toggled when the speed request from the throttle is below a set speed. The motor is stopped, and the motor code is exited. The system updates and checks are made.

The SystemOnOff system variable (Table 12) is initiated to OFF (0). With this setting, the motor is not allowed to run. If motor operation is required without the RX23W connected, set this variable to 1. If the RX23W is connected, the variable is set by pressing the power button on the mobile application (client). The client sends the status change to the server (RX23W). The RX23W changes this variable through an I2C transaction.

9.3. Endless While Loop

The endless while loop (*while(1)*) is the main function for the system. This is where system status is updated, charger port status is checked, and the motor is run. There are five routines (*sys_update()*, *sys_charging()*, *motor2BfeConnect()*, *motor_trigger_detect()* and *sys_discharging()*) that perform these tasks.

9.3.1. Sys_update()

The *sys_update()* routine updates system variables and scans the status of the battery pack.

The system can be in an IDLE state where there nothing the system is not charging or discharging. A BFE single system scan(*bfe_single_scan_start()*) is performed to monitor and update the battery pack status. Once the system scan is complete (BFE Busy bit is 0), the BFE fault and status bits are updated, the measurements are read back and the open cell voltage (OCV) fuel gauge is updated. The *bfe_update_monitor()* is used to monitor the floating values of the BFE measurements.

If either the charger or the motor is PRESENT (system variables MCPresent,, LineInPresent, WirelessPresent (Table 12)), the BFE continues to scan the battery and system parameters. After the scan is finished (BFE Busy bit = 0), the BFE variables and measurements are updated.

The code then finishes the system variables update then sends a 202-byte packet (200 data bytes and 2 bytes for CRC) to the RX23W, provided the RX23W is present (*AuxPresent*). Otherwise, the system checks for a connection to the RX23W (*aux_check()*). The data bytes sent are the system variables. Following an RX23T send command, the RX23T reads 12 bytes back. The 12 bytes are a way for the RX2W to communicate with the RX23T. Data byte 0 bit 7 of the readback is the SystemONOFF (Table 12) variable. The RX23W changes this bit to disable/enable the motor.

Any BFE faults detected while updating are binned out and system variables are updated with a series of compares.

9.3.2. Sys_Charging()

The *sys_charging()* routine does a port status update. The code services the port if allowed. Charging is not allowed while the motor is operational. If either charger flag is PRESENT while charging is not allowed, the charger that is present is programmed to shut down.

When the system is actively charging.

When charging is allowed and either charger is PRESENT, the system checks if the ISL94216's VEOC (voltage end-of-charge) status bit or if the Battery Full status bit (BATT_FULL) is set. A set VEOC bit indicates at least one cell measured a voltage close to battery full voltage. A set bit changes the BFE's charging state from monitoring a cell voltage to comparing a pack current below a set threshold (IEOC). The IEOC threshold is the end-of-current charge threshold or the tapper charge threshold. Once the pack current measures below this threshold, the BATT_FULL bit is set. Charging is complete.

If VEOC is set, the charger output voltage changes from a regulation voltage of CHG_VOUT_BULK to CHG_VOUT_ABSORPTION. These variables are defined in *charger.h*. When charging with VEOC set to low (0), the charger is in constant current mode (CC). The charging current can be as high as 10A. The current passes through a diode (D10) reducing the charging voltage to the pack. When the charger is off, the output impedance to the charger is low to ground. With CFET OFF and no blocking diode, current flows from the battery through the CFET body diode to the charger. The diode is present to prevent discharging. The bulk regulation value is above the final battery charge voltage to allow for full current charging for longer. Once VEOC is set, the charge regulation voltage is reduced to the final pack voltage.

Once charging is complete (BATT_FUL is set), the charger is disabled, and the system is placed in a safe state.

An additional check is made when the system is wirelessly charging. The system measures each P9415 for sourcing current and output voltage. This is to check the health of the wireless receiver. If the receiver and transmitter are misaligned or sourcing too much current, the readings will exceed the voltage and current limits defined in *wlp.h*. Any errors will shut down the charger system.

When the system is not charging

When the system is not charging, the routine does a *chg_wired_charger check()* (See pg.28) to determine if a line has been connected since last check. If a connection has been made, the inverter is turned OFF by way of turning DFET off and disabling interrupts. The charger is initialized (*chg_init()* pg.28), and the code proceeds to connect the charger to the battery (*chg_wired_charger_connect()* pg.32).

Wired charging takes precedence over wireless charging. The $\overline{\text{INT}}$ pin for each P9415 is OR'd and connected to IRQ2 (P00) of the MCU. Upon wireless transmission (P9247) to the receiver (P9415), the receiver powers up resulting in that interrupt $\overline{\text{INT}}$ pin rising. The positive edge interrupts the RX23T. The RX23T services the interrupt with the *r_Config_ICU_irq2_interrupt()* (smc_gen→Config_ICU_user.c) routine by setting the global variable (*g_wlp_IntPresent*) to 1.

The g_wlp_IntPresent is used to determine if a charger has been recently connected. If so, then the wireless charger is initialized (see *wlp_connect_init()* pg. 28) and a check is made (*wlp_wireless_charger_check()* page29) . If the wireless charge is present (system variable WLPresent), the discharge path is disconnected and the process of connecting the wireless receiver to the battery for charging, begins (*wlp_wireless_charger_connect()* pg. 35).

Once either charger status has been connected and confirmed (system variables LineInPresent or WirelessPresent) the motor functions are disabled (system variable DischargingAllowed is set to NOT_ALLOWED).

Any faults with the charger will shut down the charger.

Chg_wired_charger_connect()

The *chg_wired_charger_connect()* configures and connects the charger (ISL81801) to the battery. The code measures the internal reference that determines the full range and step size of the MCU ADC. The system is placed in a safe state by turning OFF any power FETs between the battery and charger or motor. The input P channel MOSFETs (Figure 9-1) that connects the charger to the wired or wireless ports is disconnected by setting the control signals (~W_CNTRL and ~WL_CNTRL) to 0. This turns OFF the NPN allowing the gate to source voltage to equal 0.

The PMOS that connects the wired connection to the charge is enabled (~W_CNTRL =1). System variables are updated and the BFE performs a signal scan to determine the unloaded pack voltage and the status of the battery pack.

Programming the charger output voltage and current.

The charger's regulation voltage is programmed to a few hundredths of the measured unloaded pack voltage. When CFET (Charge FET) is turned ON, there is less of a transient between the system. The regulation voltage and current are controlled by the pulse width modulated (PWM) pins from the MCU. See Figure 9-3.

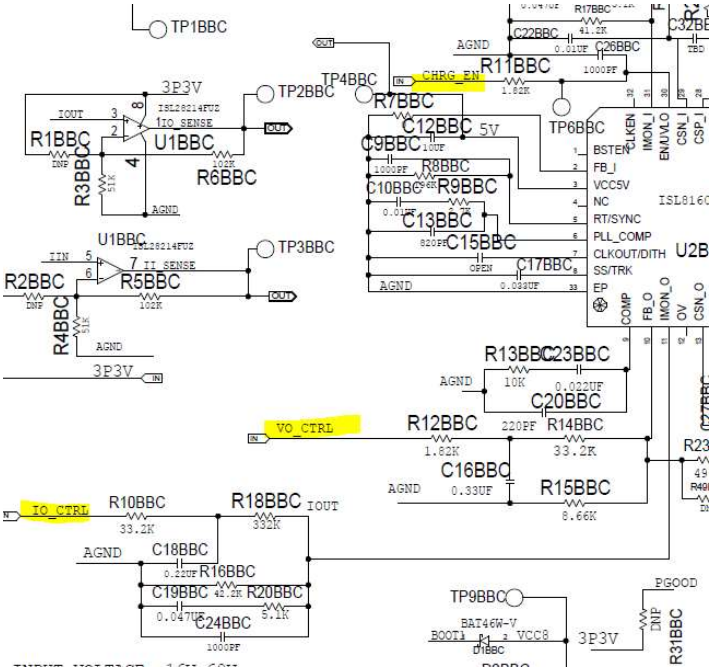

Figure 9-3: Connections to the Charger that are Reinitialized at Charger Initialization

At a high level, the Vo_Cntrl establishes a stable voltage at the R12BBC and C16BBC node (Vcntrl), to margin the regulator to the correct output voltage. VO_CNTRL is modulated by the MCU (PB0). The relationship between the charging voltage and the control voltage is expressed in Equation 1. The naming convention of the variable are similar to the names on the schematic. As an example, R23chg is the same resistor as R23BBC in the schematic and CHG_R23 macro variable in *charger.h*. The *chg_control_vout(Vout)* routine calculates and sets the PWM duty cycle such that the correct control voltage is set for the requested Vout voltage.

$$V_{out1}(V_{cntrl}) = 0.8 + R_{23chg}\left[\left(\frac{0.8}{R_{15chg}}\right) + \frac{(0.8 - V_{cntrl})}{R_{14chg}}\right]$$

$$V_{cntrl}(V_{out}) = \frac{\left[-V_{out}\cdot R_{15chg}\cdot R_{14chg} + 0.8\cdot R_{14chg}\cdot(R_{23chg} + R_{15chg}) + R_{23chg}\cdot 0.8\cdot R_{15chg}\right]}{(R_{15chg}\cdot R_{23chg})}$$

Equation 1: The mathematical relationship between control voltage and charge regulation voltage

Equation 1 can be used to configure the charger hardware for a specific range of voltages. The macro variable values need to be updated when hardware changes are made. This ensures proper operation. A 0% duty cycle does not equate to 0V for the control voltage. The lowest voltage setting for the control voltage is roughly 0.6V.

The regulation current is controlled by modulating the IO_CNTRL pin of the MCU (PA3). From the PWM signal, a voltage is established at the R10BBC/ C18BBC node (Figure 9-3) (Control Current). The voltage sets the clamping current to the battery, enabling a constant current charging.

$$I_{clamp}(IO_{cntrl}) = \frac{\left[1.2\cdot\left(R_{18chg} + R_{16chg}\right) - R_{16chg}\cdot\left[IO_{cntrl} + (20\cdot u)\cdot R_{18chg}\right]\right]}{\left[R_{18chg}\cdot R_{34chg}\cdot(200\cdot u)\cdot R_{16chg}\right]}$$

$$IO_{cntrl}(I_{clamp}) = \frac{\left[1.2\cdot\left(R_{16chg} + R_{18chg}\right) - R_{16chg}\cdot R_{18chg}\left[I_{clamp}\cdot R_{34chg}\cdot(200\cdot u) + (20\cdot u)\right]\right]}{R_{16chg}}$$

Equation 2: The mathematical relationship between control current and clamping current.

Equation 2 is the relationship between the control current (IOcntrl) and the clamp current. The resistors referenced in the equation are the resistors shown in the schematic and defined in *charger.h*. The labels are not exact matches to the mentioned documentation.

Prior to enabling the charger, the charger output current is programmed. The clamp current is programmed to a low value for safety, and the regulation voltage is programmed to be around the unloaded pack voltage. This limits the transients when the charger is connected to the battery.

The output voltage is checked to make sure the voltage is within range of the requested output voltage. Otherwise, a fault is set (WPF = 1). The Power Good pin of the controller is checked to ensure proper charger operation.

The power FET lock is removed, and the charger is connected to the battery (CFET ON). The charger regulator is ramped in steps to the charging voltage (CHG_VOUT_BULK). The value is defined in *charger.h*. This charging voltage is not the final charging voltage as explained in “When the system is actively charging.31. With each voltage iteration, the pack voltage and current are written to the Renesas Debug console. The battery pack fault and status bits are reported as well.

Next, the clamp current is incremented insteps to the final charging current. The variables CHG_START_CLAMP_I and CHG_END_CLAMP_I are the starting and ending current clamps. These variables are defined in *charger.h*. The precision of the controller’s current clamp is low. With each clamp increment, the set clamp current is compared to the desired sourcing current of the charger (CHG_IPACK_WIRED). If the measured current by the BFE is greater than the desired current, the incremental loop is exited. With each current iteration, the pack diagnostics can be written to the Renesas display console.

The charger’s input voltage is monitored with each current increment. It is possible to source more current than the wired power supply can source. If the wired power supply voltage is measured 15% below the unloaded supply, the charger connection fails and the system variable WPF is set. The charger is shut down outside the routine.

The *bfe_cb_status()* routine sets the BFE to autonomous cell balancing and charging routine. The routine is passed the threshold settings for the charger over current (COC) BFE check. The BFE is then set up for continuous scan mode.

A final safety check is made to ensure the battery and the charger are connected to each other. The voltage at the charger is measured (CHG+_S on the schematic) and compared to the pack voltage. If the difference is greater than 2V, the wired power fault bit is set (WPF). There is a blocking diode between the charger and the battery which results in a voltage drop, hence the threshold 2V.

If there is no issue with the comparison, the system variable LineInPresent is set to PRESENT and the system enters charging mode.

Wlp_wireless_charger_connect()

The *wlp_wireless_charger_connect()* configures the charger and BFE for wireless charging. The flow of the routine parallels the *chg_wire_charger_connect()* (pg. 32). Use this routine's write up as a reference.

The routine places the charger and the system in a safe state by turning off any connections between receiver, battery and charger. The receiver voltage is measured from each P9415 and compared to a voltage range (WLP_VOUT_MAX and WLP_VOUT_MIN) defined in *Wlp.h*. The nominal output voltage of the receiver is 20V. The comparison checks for receiver to transmitter alignment.

The battery pack voltage is measured. The charger and clamp are programmed to a safe current clamp value (CHG_START_CLAMP_I_WL), which is defined in *charger.h*, and the voltage is programmed to approximate the pack voltage. The charger is enabled, and the Power Good pin is checked. If the pin is low, the system variable WLPF is set and the charger shuts down outside the routine.

The charger is connected to the battery by turning CFET on. The charging voltage is ramped to the CHG_VOUT_BULK. With each step, the output voltage and sourcing current of each P9415 is written to the Renesas console.

At the end of stepping the charging voltage, each P9415 is checked for output voltage, sourcing current and the sum of current sourcing. This is done in *wlp_protection_check()*. If any check fails, the system variable WLPF is set and the charger disconnects and shuts down.

If there are no issues, the current clamp is stepped from CHG_START_CLAMP_I_WL to CHG_END_CLAMP_I_WL (defined in *charger.h*). With each current step, the P9415 output voltage and current are reported. Each P9415 is checked for sourcing excessive current, and the receiving system is checked for sourcing too much current. If any of the cases occur, the loop is exited. The thresholds are defined in *Wlp.h*.

The receivers are checked one additional time for output voltage and sourcing current. If no problems are reported, the BFE is set up for autonomous cell balancing and charging and continuous scan mode is initiated.

As in the wired charging connect routine, a check is made on the connection between charger and battery. The system variable WirelessPresent is set to PRESET if everything passes.

9.4. Motor Drive

The Motor Drive routines consists of three routines *motor2BfeConnect()*, *motor_trigger_detect()*, and *sys_discharging()*. The motor algorithm that rotates the motor and monitors for errors uses the *RX23T 120-degree conducting control of permanent magnetic synchronous motor using hall sensors* code provided by Renesas. The code and documentation are provided on the RX23T -RX24T CPU Card landing page.

This section discussed the integration between the system and the motor control code. Some motor control algorithms and key variables are discussed within the section. For in depth detail of the motor control code, visit www.renesas.com.

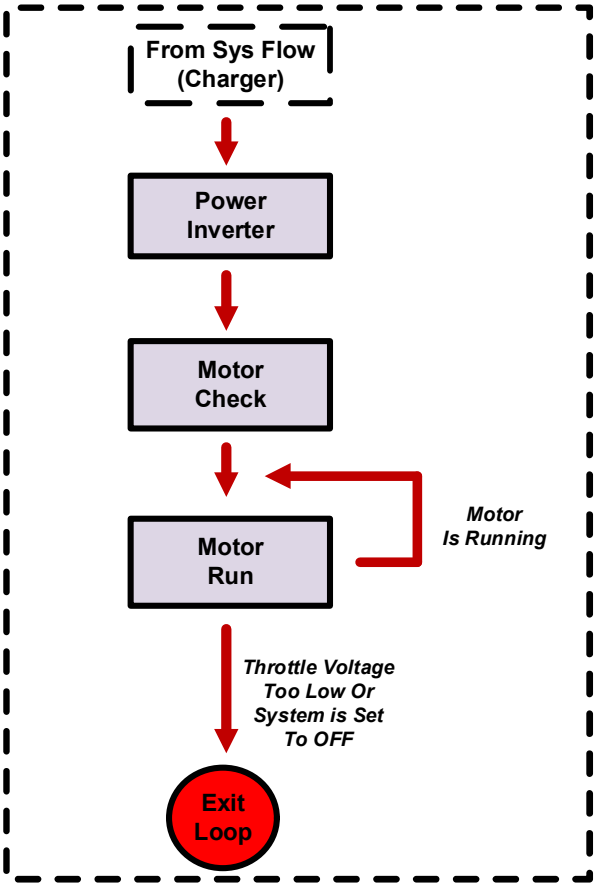

Figure 9-4: Motor Control Flow

9.4.1. POWER INVERTER

Motor2BfeConnect()

The POWER INVERTER state is a state within the MOTOR DRIVE state that electrically connects the motor to the battery pack by way of powering the Inverter / Motor Control board. The Inverter board is powered by the BFE turning ON the discharge FET (DFET), connecting the battery pack to the inverter. The *motor2BfeConnect()* routine executes the POWER INVERTER state.

The state checks the status of the system and the BFE prior to connecting the power. The MCU sets the BFE FETSOFF pin to low. This removes the safeguard that prevents the power FETs of the BFE from turning ON. A pre-charge circuit is enabled for 200ms that charges the motor control caps in a linear manner. See Figure 9-5.

Motor control capacitors are large in value and are used to stabilize the line voltage in events of instantaneous current demand or electrical kickback from the motor. Charging these large capacitors from an uncharged state results in a large inrush current sourced by the battery. A lithium battery has very low impedance and delivers nearly unlimited current instantaneously. The speed and magnitude of the current can result in unwanted transients due to inductive charging. The transients can electrically overstress the components within the signal path.

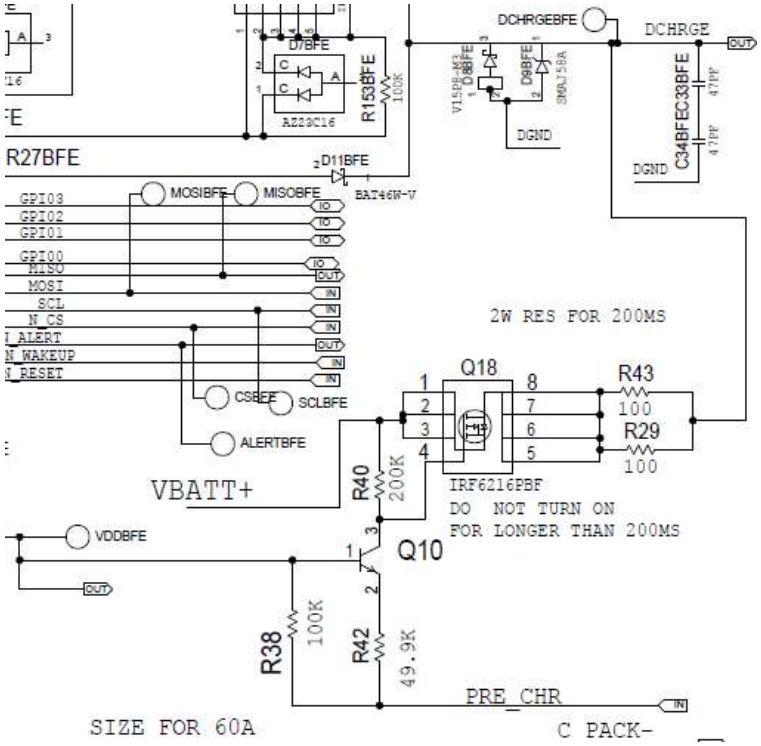

Figure 9-5: BFE Pre-charge Cap Circuit

The MCU sets PRE_CHR (MCU pin P02) to 0V. This turns ON the PMOS (Q18BFE) that connects the inverter and battery pack via current limiting resistors (R43BFE and R29BFE). After 200ms has expired, the motor control caps are significantly charged to allow for the BFE to turn ON the DFET.

In this state, the DFET can only turn ON and power the inverter board if the system is not actively charging, A single scan is performed and the BFE variables are updated. The system checks if DFET is already ON. If not, it turns on DFET by unlocking the FET safeguard, followed by a cap pre-charge and turning on the FETs. The *motor_precharge()* routine pre-charges the Inverter board capacitors. **DO NOT SINGLE STEP THROUGH THE ROUTINE**. The *bfe_DFET_ON_OFF(on_off)* routine turn DFET ON and OFF. **DO NOT SINGLE STEP THROUGH THE ROUTINE.** Run these two routines together up till the *bfe_read_all_register()* routine.

9.4.2. MOTOR CHECK

Motor_trigger_detect()

The *motor_trigger_detect()* routine runs the MOTOR_CHECK state. The MOTOR_CHECK state runs the routine *motor_check()*, which checks the connection between the battery pack and inverter supply voltage. A pack measurement is made by the BFE and compared to the pack measurement made by the RX23T (P43 of the MCU) through a resistor divider. If the two measurements are within 20% of each other, the motor present bit (g_sys_data.Sys_Status.Bit.MCPresent) is set to PRESENT. The BFE is then placed into continuous SCAN mode.

A NON_PRESENT bit setting disables the discharge path and the BFE is placed in IDLE mode.

The voltage at the battery pack is compared to the voltage that is supplied to the inverter (motor control) board(P43). The comparison is only made if a start_up request has been set (*motor_startup_request*) and discharged is allowed. The check ensures there is a low impedance connection between battery and inverter.

9.4.3. MOTOR RUN

Sys_discharging()

The *sys_discharging()* routine is a wrapper function that runs the motor. This routine interfaces the system to the *RX23T 120-degree conducting control of permanent magnetic synchronous motor using hall sensors* code through the *motor_run() routine*. The routine runs continuously, provided that the exit motor loop (gExitMotorLoop) flag is set to FALSE and the motor is PRESENT (MCPresent). The exit motor loop flag is set to TRUE when a motor stop is requested. The power to the inverter is disconnected (DFET is OFF) when exiting the routine.

There are three profiles the motor is programmed to run. Throttle Control controls the speed of the motor based on the voltage reading from the throttle. If the requested speed by way of a voltage reading is lower than the minimum throttle speed (MIN_SPEED_THROTTLE_RPM_P), the motor will not start. Above the threshold, the motor will spin at the requested speed.

The Single Speed setting programs the motor to turn at the requested speed set by the variable SpeedRpm. The Step Speed Up and Down setting programs the motor to spin at a set speed for a period of time, then sets the motor speed to 0. The next step turns the motor from 0 to a higher rpm for a period of time. The stepping continues until the max speed is reached (maxSpeed), then the speed lowers with each step until the start speed is reached. This setting is useful for tuning the motor.

To stop the motor in a controlled way, touch pin 2 of the RX23W_PMOD header (motherboard) to ground. The motor will stop as a result of this. The ground pin connection asserts the pin PE2 (AUX_NMI) of the MCU to ground. Within the *While* loop the pin is checked in each iteration. A low pin reading executes the *Stop Motor* command. It is safe to place a breakpoint after the command has been executed.

Motor_run()

The *motor_run()* routine drives the motor based on a requested speed, either from a throttle measurement or a programmed speed. The *motor_run()* routine first determines the state of the motor. A motor state of INACTIVE with a motor *start* command requested, re-initializes the motor and variables, then tries to rotate the motor. A *stop* or *reset* command stops the motor. The routine updates the desired speed of the motor when the motor is in ACTIVE state without a *stop* command. The *motor_run() routine* executes with every iteration of the while loop. Motor status and controls are updated, except for timer and hardware interrupts. In this operation the system is checked to make sure the halls signals are sequencing in the correct order and in a timely way and the inverter voltage and leg currents are within range. Any errors measured result in resetting the motor and the state, followed by a motor *start* command. Table 9 contains useful variables to monitor while operating the motor.

WATCH VARIABLES OF INTEREST	DESCRIPTION
ST_G.F4_VDC_AD	Inverter voltage at MC_PACK
ST_G.F4_IU_AD	U leg current
ST_G.F4_IV_AD	V leg current
ST_G.F4_IW_AD	V leg current
ST_G.U2_RUN_MODE	What mode is the motor in [INIT -0, BOOT--1; DRIVE--2 (closed loop)]
ST_G.U2_ERROR_STATUS	Reports errors detected in mtr_carrier_interrupt
ST_G.F4_SPEED_RAD	Measured speed in radians; rads→rpm = 9.55/(number of pole pairs)

Table 9: Useful Variables to Monitor While Running the Motor

DO NOT PAUSE THE MOTOR CODE WHILE RUNNING THE MOTOR. Doing so can damage the hardware and potentially ignite the motor or battery pack. Safe places to stop the code while running the motor are in motor error if statements, after a *motor_run* stop request or whenever the motor has been re-initialized or reset prior to starting.

ERROR DESCRIPTION	ERROR CODE
NO ERROR	0x00
OVER VOLTAGE (INVERTER)	0x02
OVER SPEED	0x04
HALL TIMEOUT	0x08
HALL PATTERN ERROR	0x20
UNDER VOLTAGE (INVERTER)	0x80
OVER CURRENT U	0xA1
OVER CURRENT V	0xA2
OVER CURRENT W	0xA4
ALL 3 PHASES OVER CURRENT	0xA7

Table 10 Error Modes for The Motor

Table 10 are the error codes for the st_g.u2_error_status variable. When watching the variable, set the variable to real time refresh.

When debugging the motor run code, there are some routines of interest.

Mtr_speed_calc(mtr_st_hall_120 * st_m)

The routine calculates the speed of the motor in radians. It uses the last 6 hall interrupt timer readings to determine the motor speed. The routine is located in mtr_spm_hall_120.c.

Mtr_1ms_interrupt ()

A timer-based interrupt that does the transition between initiate motor start, boot state (open loop) and drive state (closed PI loop). The boot state waits for the motor to reach a minimum speed before transitioning to a close loop system (drive state). The routine sets the duty cycle and reference voltage based on speed and voltage of the system. The routine is located in mtr_interrupt.c.

Mtr_carrier_interrupt()

The routine compares the motor readings to the respective limits. The motor is checked for inverter over/under voltage, phase(s) over current, over speed and hall timeout errors. **DO NOT SET A BREAKPOINT IN THIS ROUTINE**. Printf statements do not work within the routine. The printf command is a slow statement to execute. Do a real time watch on the variable in Table 9 or store the variable(s) of interest in memory (pointer or array) and print the value while outside the *motor_run()* routine.

For more information with the motor code, refer to the *RX23T 120-degree conducting control of permanent magnetic synchronous motor using hall sensors* code documentation on the Renesas web site.

10. Connecting and Tuning the Motor

The code provided with the solution drives a y-rotated brushless motor with hall sensors. This section should be used as a guideline for connecting the electronics to the motor.

For motors that do not have the U, V, W or A, B, C wires labels, a quick test to determine the phase sequence to connect each wire to a three-resistor bridge is shown below.

10.1. Inverter board Testing.

Prior to mating the Inverter to the MCU, Charger, BFE board for the first time, power the inverter board with an independent supply. Apply 16V or greater between MC_Pack+ and MC_Pack-. Monitor the supply current to the inverter board. The current sourced by the power supply should not exceed a couple mA.

Measure the regulated voltage connected to the FET driver circuit (TP7MC). A reading within 20% of 12V is expected.

Measure the regulation voltage that connects to the current sense and Hall circuitry (TP6MC). The measured voltage should be within 20% of the nominal regulation voltage. The nominal regulation voltage is 5V. The hall sensor operating voltage determines the regulation voltage for the board.

3.3V can also be programmed by changing the feedback resistor divider circuit of the LDO. Please look at the notes in the schematic for changing other resistor values if the system is running off 3.3V.

The current sense circuit is bidirectional. The output current of the current sense should measure roughly 1.65V or half the ADC measurement range. This is the zero-current voltage. The analog measurement range for RX23T is 0V to 3.3V. The ADC measurement is 12 bits.

10.2. Hall Sensors

Most hall sensors are powered to 5V. The hall output is an open drain requiring a pull up. Each hall sensor phase is pulled to a 3.3V by way of a low pass filter. The 3.3V pull up makes the signal compatible to the RX23T logic range. The Rx23T is powered at 3.3V.

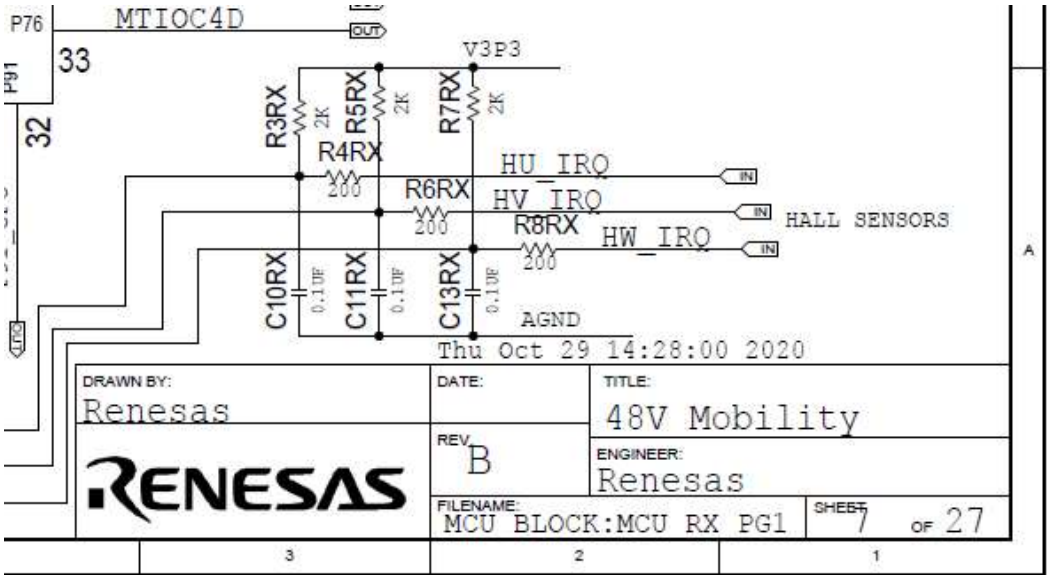

Figure 10-1 Hall Sensor Connection to the RX23t

Design Note:

At motor startup, the instantaneous current drawn by the motor may result in errant pulses being coupled to the hall signal lines. It is important to filter the hall signal to the MCU appropriately such that the slew rate of the signal is maintained while suppressing any electrical noise. Be mindful of the VIL and VIH levels of the MCU.

Powering the Inverter Board to determine the Hall Sensor Order.

Read through "Setting up the programming environment" (pg.10) if this is the first time running the debug environment. The Inverter board requires 16V or more for proper operation.

The Inverter/Motor Control board is powered by the battery pack from the motherboard. Stop the code (software break point) after the *motor2BfeConnect()* routine to power the inverter board without trying to drive a motor.

Connect the hall sensor wires to the hall header (HALL_HDRMC) located at the edge the Inverter board (Figure 10-2).

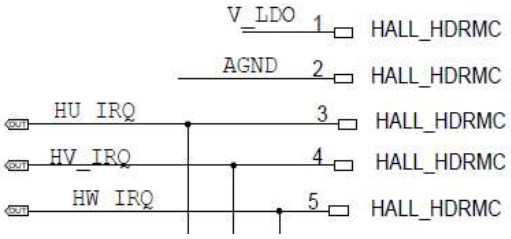

Figure 10-2 Hall Header on the Inverter Board

When the Inverter board is powered, probe the voltage at HU, HV or HW (TP11MC, TP12MC and TP13MC). The output of the halls should be between 1 and 6. HU, HV and HW can never equal each other. Record the starting value of the hall signals. Assume the starting position of the hall is 6 (HU =0, HV=1, HW =1). Arrange the hall signal connections to sequence the hall signal in the following order 6→2→3→1→5→4→6 when rotating the motor clockwise. The rotation direction is from the perspective of looking into the motor shaft that connects to the motor. This ensures the hall connection to the board is correct.

Note:

Use an oscilloscope for easier viewing. Some motors do not have smooth transitions between cogs, resulting in fast transition of halls reading that may lead to visually missing a signal transition. There should never be a reading of 0 or 7.

10.3. Determining the Motor Parameters.

Motors bought from a distributor or a third-party vendor often do not have detailed specifications describing the electrical parameters or the number of poles pairs. This section discusses a series of experiments to determine the motor's mechanical and electrical parameters

10.3.1. Phase Connection Order and Phase Determination

For motors that do not have the U, V, W or A, B, C wires labels, a test to determine the phase sequence is to connect each wire to a three-resistor bridge. As shown in the image below.

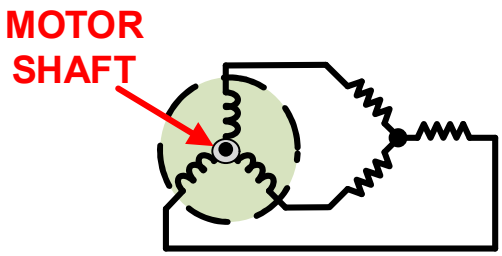

Figure 10-3: A three-resistor bridge connection to determine phase order

Prior to rotating the motor, connect differential probes to each motor wire and reference the probe to the center of the resistor bridge. Connect the shaft of the motor to a power tool or some equipment that can rotate the shaft clockwise while looking into the motor shaft. Rotate the motor shaft at a relatively constant speed.

While spinning the motor, monitor the measured signals on the oscilloscope. Each phase connection should have a sine wave like response, as shown.

The oscilloscope screen capture in the Figure 10-4 shows the measured response from the motor when it is **driven** by the inverter. The screen capture can be used to visualize the response from the motor when the inverter is **not** driving the motor.

Performing this experiment yields the sequence of the motor connection. Powering the hall sensors while spinning the motor yields the specific phase for each motor wire.

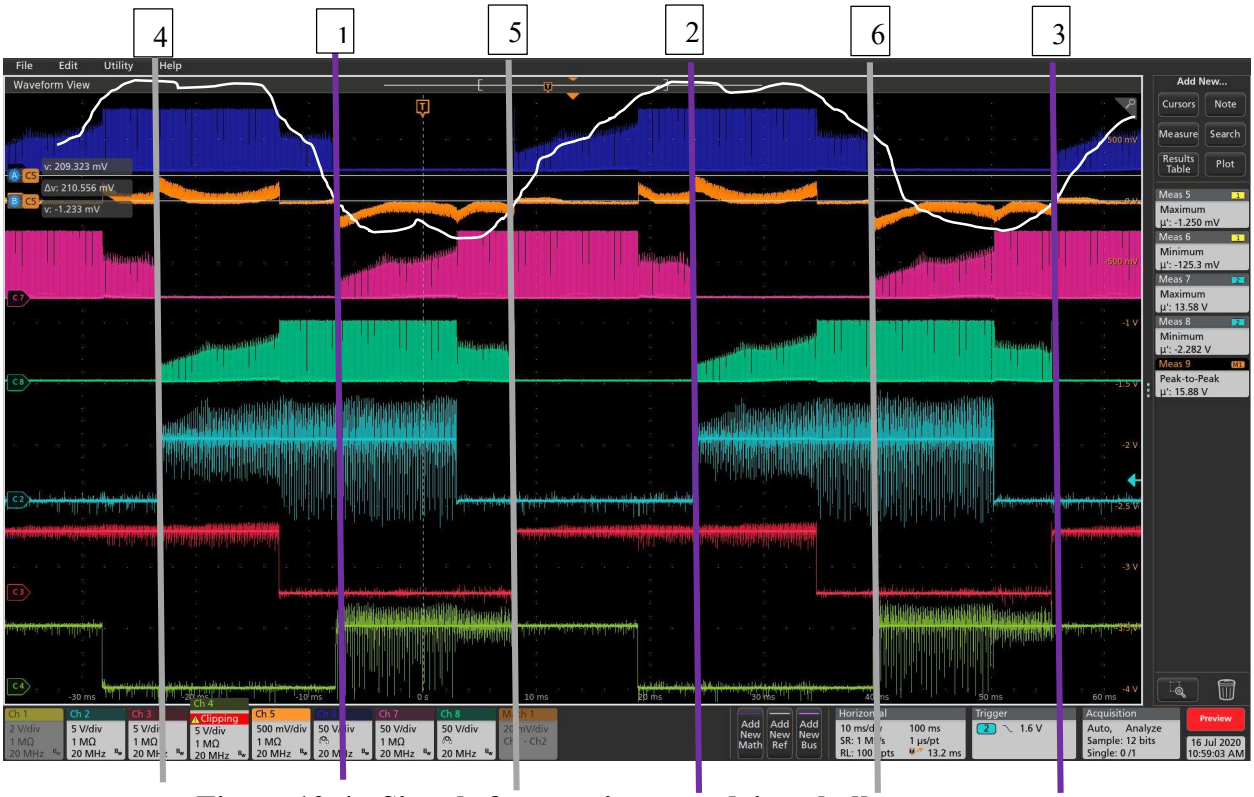

Figure 10-4: Signals from an inverter driven hall sensor motor

In the picture above and from top to bottom, the signal definition are as follows; C6(Blue) – phase voltage U (A), C7(Pink) – phase voltage V (B), C8(Green) – phase voltage W (C), C2(Cyan) – Hall Sensor U, C3(Red) – Hall Sensor V , C4(Yellow) – Hall Sensor W and C5(orange) – Current measurement of (1A/10mV) phase W. The white line is a drawn trace that represents the back EMF of U phase.

The white line is roughly the signal shape from each phase while spinning the motor without the inverter. To determine the order of the motor wires (A, B, C), arbitrarily choose the motor wire and look for the negative slope zero crossing (label 1 in the figure). At the same instance of time, look for a rising slope zero crossing at the other two wires. This is the next phase the inverter would drive. The third wire is roughly at a signal peak at the same instance. When the second phase negative slope zero crossing occurs, the third phase positive slope zero crossing occurs (label 2 in the figure).

Once the U, V, and W hall signals are known (pg.41), the specific phase of each wire is determined by spinning the motor while monitoring the phase voltage and hall signals. The rising edge of each phase hall signal occurs when the phase voltage is roughly at its positive peak. Labels 4 through 6 show the transition for each hall signal.

Design Note:
The physical placement of the hall sensor to the motor may not be equidistant. It is possible that one hall signal is present for a longer time than another.

The hall signals shown above are noisy. Noisy hall signals may lead to drive signals being applied when they are not desired. The signal path should use hardware and software filtering. When measuring these signals, make sure the probes are properly grounded.

10.3.2. Motor Pole Pairs

There are several methods of determining the number of poles pairs a motor has. Assume the same setup was used to determine phase order and phase assignment.

Spin the motor at a set speed or measure the speed the motor is spinning with a tachometer. Capture an oscilloscope screen shot of the hall signals.

Measure the frequency of a halls signal. The frequency of all the hall signals should be roughly the same. The mathematical relationship between hall frequency and the number of pole pairs is shown below

$$F_{Hall} = \frac{N_{poles} * \frac{N_{rad/s}}{2 * \pi}}{2}$$

Equation 10-1: The relationship between Hall Frequency, Poles and Speed.

$N_{rad/s}$ is the speed of the motor in radians per second.

N_{poles} is the number of pole pairs of the motor.

F_{Hall} is the frequency of the hall signal.

The motor speed is either known or measured by a tachometer. The frequency of the hall is measured. Solve for the number of pole pairs. The value should be a whole number. Round if necessary.

10.3.3. Motor Resistance

The motor resistances can be measured by using an ohm meter and measuring the resistance between the phases.

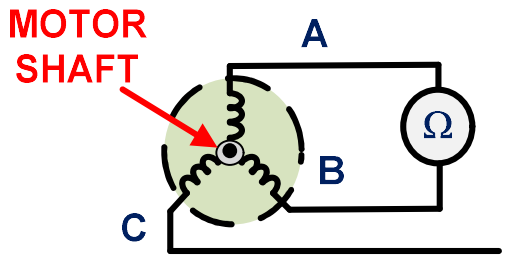

Figure 10-5: The Motor Resistance is Found by Measuring the Resistance between Phases

Measure the resistance between phases A and B, A and C and finally B and C. The value of each measurement should be roughly the same. If the values are significantly different, this could signify that the motor is not a Y-rotate motor.

$$R_{winding} = \frac{R_{A\ to\ B} + R_{A\ to\ C} + R_{B\ to\ C}}{3 * 2}$$

Equation 10-2: Motor Winding Resistance

$R_{Winding}$ is the average wind resistance.

$R_{ph1\ to\ ph2}$ is the resistance measurement between phases.

10.3.4. Motor Inductance

Performing the same experiment using an LCR meter instead of a resistance meter, the inductance of each motor winding is solved by using equation below.

$$L_{winding} = \frac{L_{A\ to\ B} + L_{A\ to\ C} + L_{B\ to\ C}}{3 * 2}$$

Equation 10-3: Motor Winding Resistance

$L_{Winding}$ is the average winding inductance.

$L_{ph1\ to\ ph2}$ is the inductance measurement between phases.

10.4. Connecting the Motor Parameters to the System Code

After all the motor parameters and phase locations are known, connect the hall sensor and phase wires to the inverter. **MAKE SURE THE POWER IS DISCONNECTED**. Phase A of the motor connects to UMC of the inverter. Phase B connects to VMC of the inverter. Phase C connects to WMC of the inverter.

10.4.1. Motor parameters configurations

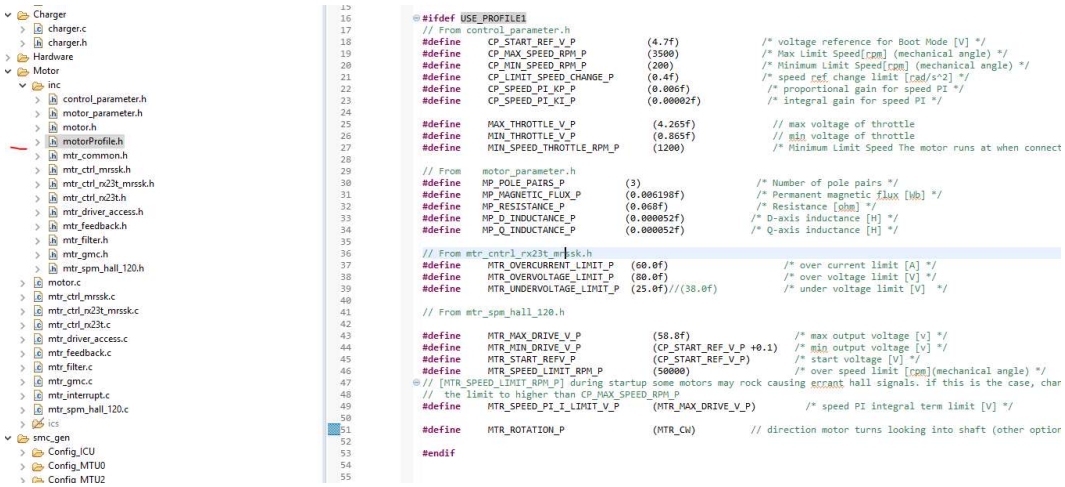

Figure 10-6: Motor Parameters Definition

The *motorProfile.h* header located in the includes folder of Motor section (See **Error! Reference source not found.**). The file extracts the important parameter and thresholds from the underlying code to define the motor that is connected. Either create a switch or change the values for each parameter to describe the motor connected.

The scooter motor is an induction motor. The Q inductance value is the same as D inductance value.

10.4.2. Motor and Control Loop Limitations

Parameters:

CP_START_REF_V_P is the starting voltage to initiate motor movement. Set a starting voltage between 1.5V and 1.9V and increase the voltage until the motor starts to move. Monitor all phases and hall signal when testing.

CP_MAX_SPEED_RPM_P is the maximum speed (RPMs) the motor can be driven to.

CP_MIN_SPEED_RPM_P is the speed the motor turns while the system is open looped and in either in BOOT or INIT states. While in BOOT state, if the hall speed readings equal or are greater than this value the state changes to DRIVE. The DRIVE state closes the system and uses a PI control routine to correct the system error (*mtr_feedback.c*).

CP_LIM_SPEED_CHANGE_P is how quickly the close loop system responds to a change in speed.

CP_SPEED_PI_KP_P is the proportional coefficient for the PI control system. Set value to 0.002 and increase. Increasing the parameter value overdamps the motor response.

CP_SPEED_PI_KI_P is the integration coefficient for the PI control system. Set the value to 0.00001 and increase. Increasing the parameter quickens the motors response to a change in speed step. Increasing the value too much may result in an underdamped condition or an oscillation within the control system.

MAX_THROTTLE_V_P is the max voltage from the throttle. This parameter is used in Throttle Control mode.

MIN_THROTTLE_V_P is the min voltage from the throttle. This parameter is used in Throttle Control mode.

MIN_SPEED_THROTTLE_RPM_P is the minimum motor speed when the MCU throttle reading is above this value. A voltage reading below the threshold stops the motor. This parameter is used in Throttle Control mode.

10.4.3. Motor Control Limits

Parameters:

MTR_OVERCURRENT_LIMIT_P is the maximum current limit measured at each leg. Each leg's current is compared to this value.

MTR_OVERVOLTAGE_LIMIT_P is the maximum voltage limit measured for the supply of the inverter.

MTR_UNDERVOLTAGE_LIMIT_P is the minimum voltage limit measured for the supply of the inverter.

10.4.4. Hall Controls and Limits

Parameters:

MTR_SPEED_LIMIT_RPM_P is the speed limit measured from the hall signals. The speed limit should be equal to the forced speed limit (**CP_MAX_SPEED_RPM_P)**. Increase the value if errant hall pulses result in errant speed values. Look for errant hall pulses at motor startup. If there is rocking of the motor at start up, this could lead to a bad speed reading.

MTR_SPEED_PI_I_LIMIT_V_P is the maximum voltage the PI loop can request. This value should equal the maximum voltage of the pack.

MTR_MAX_DRIVE_V_P is the maximum voltage the system can command.

MTR_MIN_DRIVE_V_P is the minimum command voltage. The value is typically a couple tenths of a volt higher than the starting voltage (**CP_START_REF_V_P)**. Some motors may contain a high startup voltage and a lower voltage to maintain momentum.

MTR_ROTATION_P is the rotation of the shaft from the perspective of looking into the shaft. An MTR_CW is a clockwise rotation. An MTR_CCW is a counterclockwise rotation.

10.5. Testing the motor

Before testing the motor, change the value of the SystemONOFF variable to 1. This allows the motor to discharge without a Bluetooth connection. Change the gMotorSpinProfile within *sys_discharging()* to SINGLE_SPEED. Change the SpeedRpm variable to the desired speed. In *motorProfile.*h, change the CP_START_REF_P value to 1.9 to start. Compile the program.

With an oscilloscope, measure the phase voltage and halls signal for each leg. Measuring a phase current is a bonus.

Without a charger or Bluetooth module connected, step to the *motor_run()* within the *sys_discharging()* routine.

Note the hall starting location. Set the scope to Single Sequence and capture the first pulse transition sent from the inverter to the motor. Using the inverter signal shown in Figure 10-4 pg.43 of a motor rotating clockwise as a reference, determine if the PWM pulses are being applied to the correct phase of the motor. If the starting hall signal is 110b (W=1, V=1, U=0), the PWM signal should be on the U(A) phase.

In the figure, locate label 2. Locate hall signal 110b to the left of label 2. The system changes which PWM is pulsed when a hall signal transitions negative. At approximately 20ms, the hall transitions from 110b to 010b. At this time, the U(A) phase is being driven.

If the PWM is not being applied to the correct phase, change the connections and repeat the experiment. If certain that the motor connections to the inverter are accurate and the motor is not moving (Hall Time Out s*st_g.u2_error_status()* = 0x08), change CP_START_REF_P by a few tenths. Continue raising the voltage until the motor turns. Do not raise the value too much; the system could be damaged.

NOTE:

DO NOT set breakpoints within the motor control routine after the motor has executed a motor start command. Doing so could damage the hardware and motor. It may even start a fire. To monitor the motor parameters in real time, see Mtr_carrier_interrupt() (pg.40).

10.6. Tuning the motor (PI Loop)

For this section, the motor should be spinning at a set speed. In the *sys_discharge()* routine, change mode to STEP_SPEED_UP_DOWN. The mode accelerates the motor for a stop to a set speed. For each speed step, the speed step increases by a *stepSpeed* until the *maxSpeed* is reached. Once the *maxSpeed* step has been performed, the speed step is decreased by a *stepSpeed*. This continues until the speed is below the *startSpeed*.

After setting the variables and compiling, launch the real time chart from the Renesas Views menu [Renesas View → Debug → Real-time Chart]. A graphical window appears.

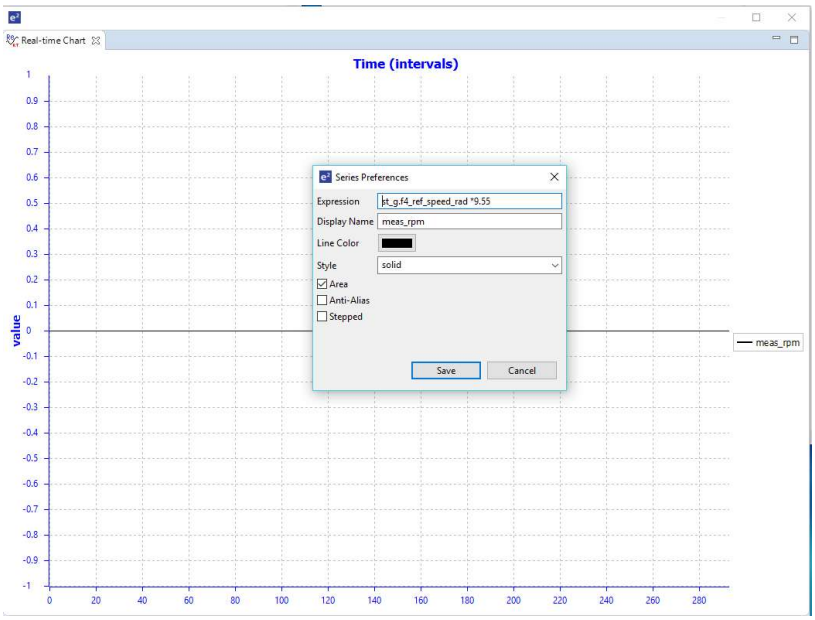

Figure 10-7: Add Series to the Real-time Chart.

Within the chart, right-click and choose new series. The measure speed variable is *st_g.f4_ref_speed_rad*. Change the output units to rpm by multiplying the variable by 9.55/number of pole pairs. Add another series equal to *SpeedRpm*. This is the requested speed of the motor.

To increase the responsiveness of the motor, increase the value of CP_SPEED_PI_KP_P (proportional gain) first then CP_SPEED_PI_KI_P (integral gain). The CP_LIMIT_SPEED_CHANGE_P is the responsiveness of the motor once it receives a change in speed request.

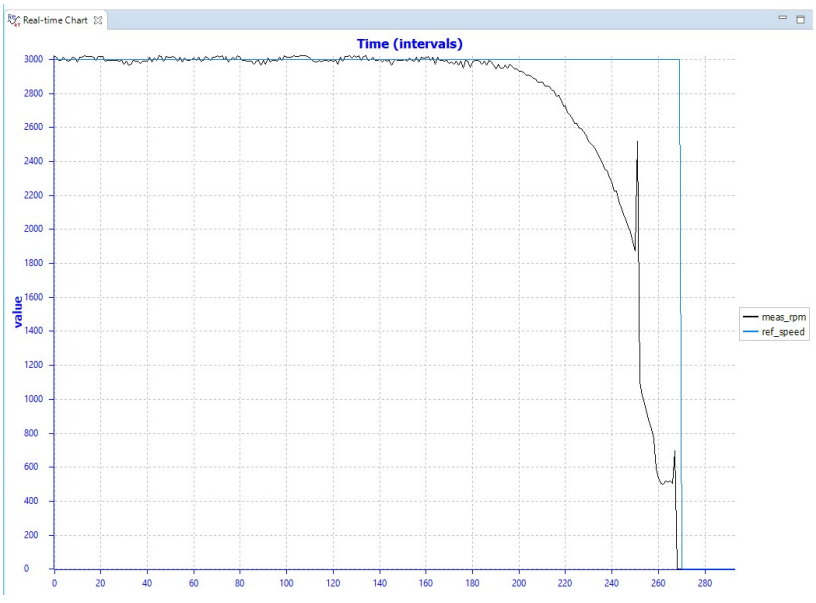

Figure 10-8: Changing Control Loop Parameters Changes the Responsiveness of the Motor

Figure 10-8 shows at the response of the motor from a single step response. The old data points are on the right side of the graph and the newest measurements are on the left.

NOTE:

The chart clears itself upon terminating the debug session. To capture the graph, pause the code while the motor is stopped or place a breakpoint after the motor stop command.

11. Changing the Hardware

11.1. BFE (ISL94216)

The BFE can support battery packs less than the 14 cells. There are a series of Do Not Place resistor footprints that have been added to the board to support packs from 7 cells (See Figure 11-1).

For every cell that is reduced from 14 cells, populate the cell balancing and Vcell resistor closest to vcell 8 (VC8). A 13-cell pack would populate R9BFE (CB10) and R2BFE (VC10) with 0ohm resistors. A 12-cell pack would populate the 13-cell resistor and R33 (CB7) and R32 (VC7).

Table 11 are the hardware changes need for system less than 14 cells. With the hardware changes, software changes are needed for threshold limits and cell counts. Threshold limits and count are found in the *BfeProfile.h*. The register value for the variable NUM_OF_CELLS is defined in Table 11. The NUM_OF_CELLS value for 14 cells is 0xFE7F.

NUMBER OF CELLS	RESISTOR TO POPULATE	CB RESISTOR TO REMOVE	CONNECT VC8 TO 2BATBFE PINS	NUM OF CELLS VAL
13	R9BFE, R2BFE	R66BFE	8 to 9	0xFC7F
12	13 cell res + R33, R32	R66BFE	7 to 8	0xFC3F
11	12 cell res + R21BFE, R19BFE	R64BFE	7 to 8	0xF83F
10	11 cell res + R35, R31	R64BFE	6 to 7 or 6 to 8	0xF81F
09	10 cell res + R37, R36	R42BFE	6 to 7 or 6 to 8	0xF01F
08	09 cell res + R39, R30	R42BFE	5 to 6 or 5 to 8	0xF00F
07	08 cell res + R34, R41	R40BFE	5 to 6 or 5 to 8	0xE00F

Table 11: BFE System Configuration for Cells Less than 14.

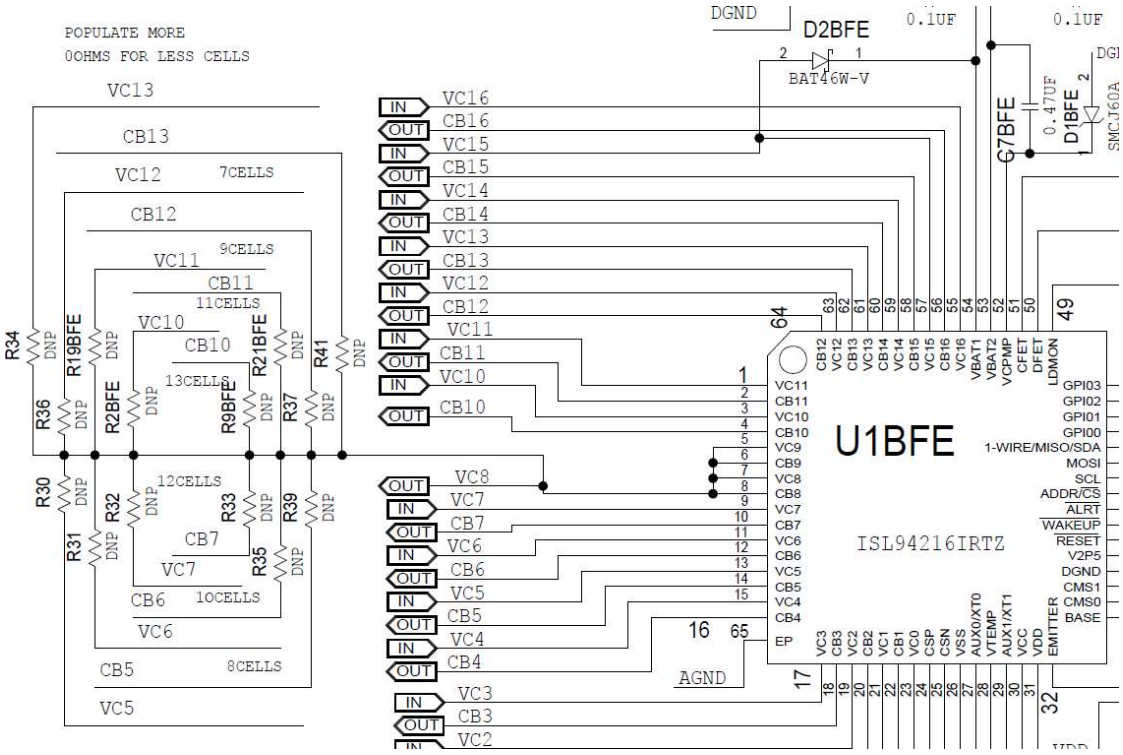

Figure 11-1: USE DNP Resistors to reduce the cell count of the pack

11.2. Charger

For charger regulation current and the voltage configurability, see section “Programming the charger output voltage and current.” (pg.32). The *chargerProfile.h* has the thresholds of the battery pack to be charged. Change regulation voltage and current to suit the pack to charge. Any component value changes made in hardware should be reflected in the *charger.h* header file. The *charger.h* also has the component values for the attenuator (*WIRE_AO*).

11.3. Wireless Charger

The *wlpProfile.h* head is the file that stores the thresholds specific to the wireless charger. All other profiles charging profiles (regulation voltage and charging current) are stored in *chargerProfile.h*.

11.4. Motor

See “Connecting the Motor Parameters to the System Code” (pg.45) to change the *MotorProfile.h* thresholds and parameters for a specific motor. In addition to connecting the motor, the regulation voltage that powers the hall sensors and throttle are configurable.

The V_LDO regulator is variable (Figure 11-2). The default configuration for the V_LDO regulator is 5.0V. A 3.3V V_LDO output requires R66MCMCMC to change to 118kΩ. Changing the regulator value to 3.3V requires R47MC to be removed from C54MC and the MC_THROTTLE_ATTN (*control_parameter.h)* to equal 1.

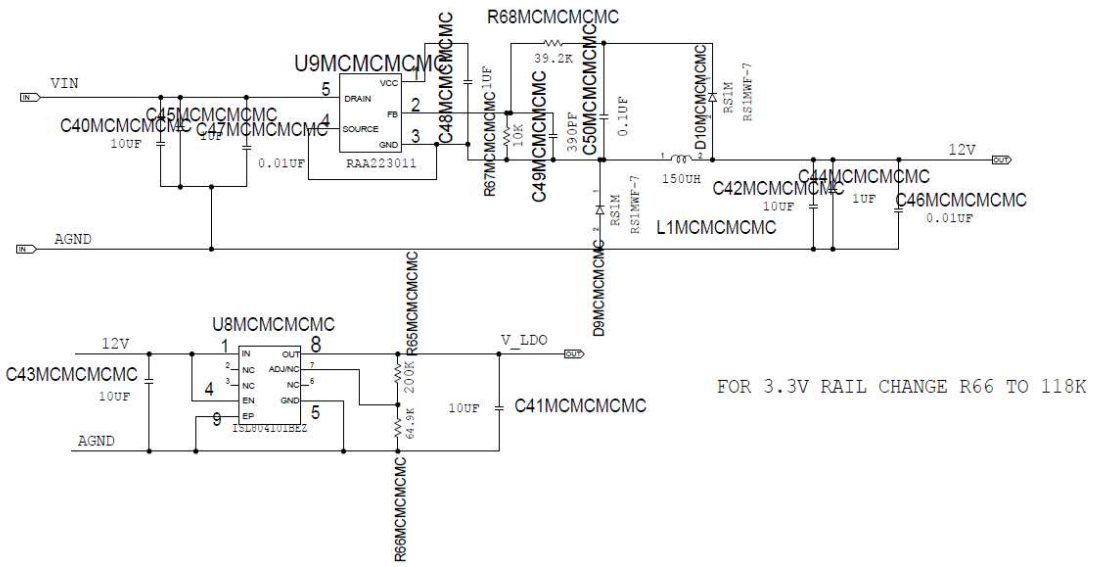

Figure 11-2: Inverter Board Regulator

The change in regulator voltage changes the center voltage of the bidirectional current sense amplifiers (U10MC, U11MC, U12MC). These amplifiers combined with the ADCs inside the MCU measure the leg current. The max current should be within the measurement range of the ADC (0V to 3.3V).

For a regulator voltage of 3.3V, change R46MC, R49MC and R51MC to 20kΩ.

The hardware OCP requires changes for the 3.3V operation. Change R70MC to 8.87kΩ and R74MC to 9.53kΩ to equal the threshold levels for 5V.

12. Bluetooth Connection (RX23W)

The US069 uses the RX23W Target Board (RTK5RX23W0C00000BJ) for connection between the mobile app and the system. More information is found on the RX23W Bluetooth system in Appendix C on page 93 of this document.

The operational guide that sets up the RX23W target boards is provided in “Appendix B BLE Operation Guide” pg.60.

The device composition for this project is shown in Figure 12-1.

A Host Board (RX23T), which behaves as an I^2C Master, is connected to the RX23W, which acts as an I^2C Slave. The RX23W connects to a Smart Phone via BLE, where the RX23W behaves as the client (master) and the Smart Phone acts as the server (slave). The Smart Phone is an Android or iOS device. The RX23W can be optionally connected to a Serial Emulator via USB connection to a Host PC.

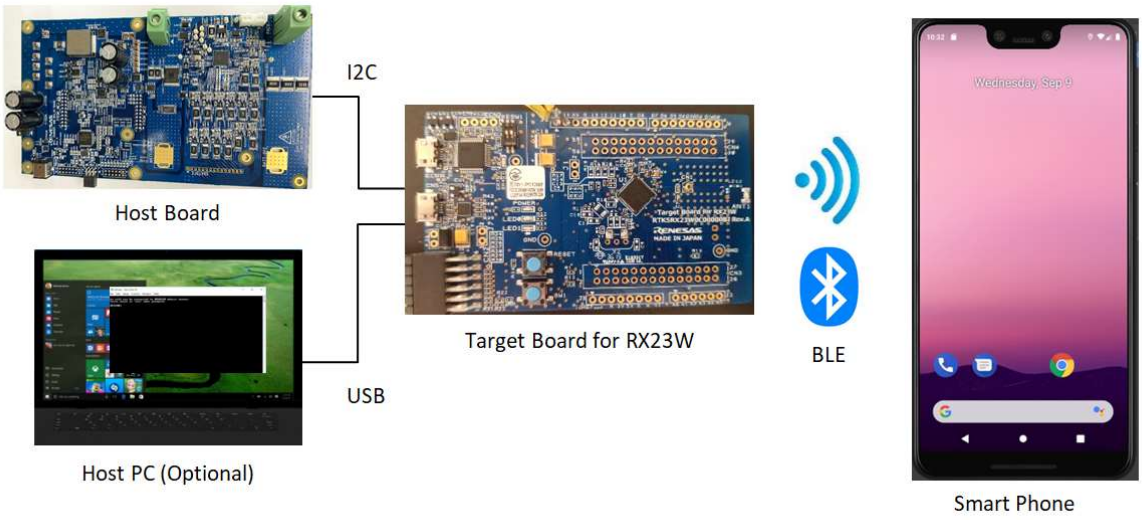

Figure 12-1: Bluetooth System makeup

12.1. Initialization

When the device is powered on, the MCU will initialize and enable the LED, the I²C module, and the BLE module. The LED is used to indicate data sent from the mobile app. The I²C module sets the slave address of the RX23W (0x71) and sets up two I²C buffers. The first is a receive buffer with a size of 202 bytes, and the other is a send buffer with a size of 12 bytes. Then, it enables I²C communication and prepares for slave transfer/reception. A transmission request from the master will now trigger an interrupt linked to an I²C callback function which handles I²C transmissions.

The BLE module initializes all the common protocol stack elements, such as the host stack, the Generic Attribute Profile (GATT) database, the GATT server/client, and the custom service, the System Data Transmission service. GATT describes how data is transferred between peripheral and host devices, which are defined as servers and clients. Servers hold attribute lookup data and service definitions in a database, and the client sends requests to this server. Services break data up into logic entries called characteristics that can be individually accessed. The BLE module also initializes the optional Command Line Interface that can be accessed via a Serial Emulator to show BLE events such as advertising and connection.

After initialization, the BLE module is ready, and the device is set to an idle state where it waits for I²C transmissions and BLE events to process.

12.2. I²C Execution

When an I²C transmission is sent from the Scooter host device (master) with the correct slave address, the RX23W device recognizes the start of a transmission and throws an interrupt. The interrupt handler function handles incoming I²C communications by checking to see if the transmission was completed or timed out. If the transmission is completed successfully (with no NACK or timeout), a Cyclic Redundancy Check (CRC) is performed on the receive buffer, which is an error-detecting code, to ensure that the contents are correct. A CRC-16 check appends 2 check bytes to the buffer, and these bytes are checked at the end of every I²C transmission to confirm that no noise was present in the transmission.

In this system, the Scooter host device makes an I²C write and then a read approximately every second to send its system data and check for input from the mobile app. After data is sent to or received from the host device, the RX23W resets the parameters for the send and receive buffers to their initial state to prepare for the next transmission (this does not delete the buffer contents, but they will be overwritten by the next transmission) and performs a CRC-16 check on the receive buffer. When data is received from the Smart Phone via BLE, the device places the contents into the I²C send buffer and appends CRC-16 check bytes to the send buffer to be read by the Scooter host device.

12.3. BLE Execution

When an I²C transmission is completed successfully, the contents of the receive buffer are split into a static buffer, which is a buffer for static data such as device IDs and firmware versions, and a continuous data buffer, which includes battery voltage, scooter motor speed, etc. The System Data Transmission service takes the 7-byte static buffer and 120-byte continuous data buffer and encodes them into BLE packets to be sent to the Smart Phone via a Notify operation, which is a write operation that does not require acknowledgement from the client. When data is received via BLE from the Smart Phone, the data packet is decoded and CRC-16 check bytes are appended to the contents, which are placed in the I²C send buffer to be read by the I²C master.

12.4. Mobile App

A custom mobile app was developed specifically for this project to interpret and output the data received by the RX23W, as well as input data to the RX23W. Screenshots of the mobile app can be seen in Figure 12-2. In its default state, the app is idle, but when the ‘Connect’ button is pressed, the device will begin scanning for Bluetooth devices, and initiate a connection if the Device Name (which was set to ‘SCOOTER’) and Bluetooth address matches with the RX23W. The RX23W then begins to send data to the app, which is handled by the app and used to update the display. Voltage and current data are constantly updated, and the app indicates if the device is Idle, Charging, or Discharging. Battery level data can also be seen at the bottom of the Connect tab, showing in green if the battery level is in a range up to 25%, 50%, 75%, and 100% (full charge). There is a separate Register tab which allows the user to view individual register data as well.

If the user presses the ‘Powered Off’ button, the phone will send data to the RX23W to indicate a power ON state, which is decoded and forwarded via I²C to the host device to be interpreted. This allows the user to engage the throttle and control the motor.

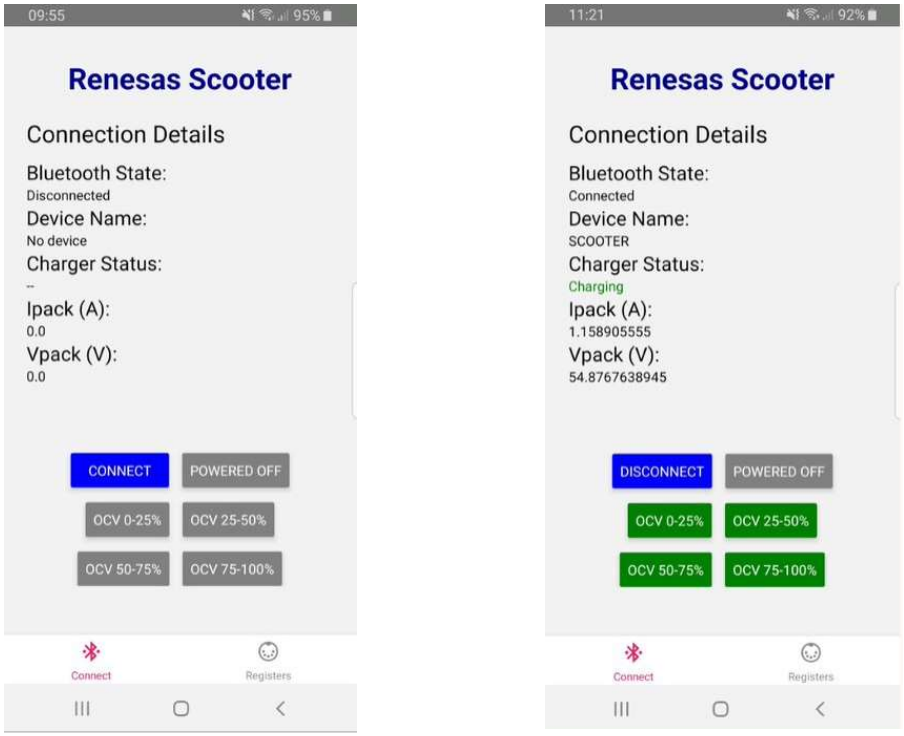

Figure 12-2: BLE Mobile App

13. Appendix A

Attached is the list of System variables for the project. These variables are actively updated and reside in the RX23T. These variables are pushed to the RX23W when the device is connected.

Struct Variable	Sub-Member	Type	Description	Default Value
Sys_Device_ID		uint8_t	Device ID for the system	0x23
Sys_Firmware		uint8_t	System firmware version	0x00
Sys_Fault		Sys_fault_t		0x00
	WLPF	uint8_t	1 = Wireless Power Fault detected	
	WPF	uint8_t	1 = Wired Power Fault detected	
	CHGF	uint8_t	1 = Charger Fault detected	
	BFEF	uint8_t	1 = BFE (ISL94216) Fault detected	

	MTRF	uint8_t	1 = Motor Fault detected	
	AUXF	uint8_t	1 = Auxiliary Fault detected	
Sys_ErrorCodeLowByte		Sys_error_code _low_t		0x00
	WPF_VIN	uint8_t	1 = Wired Power input voltage is out of range	
	CHGF_PG	uint8_t	1 = PG level of the ISL81801 is incorrect	
	CHGF_VOUT	uint8_t	1 = Charger output voltage is out of range of target	
	CHGF_II_IO	uint8_t	1 = Output current is greater than 0.5A when CFET is off	
	WLPF_DEVICE	uint8_t	1 = P9415 LDO output current is incorrect	
	WLPF_VOUT	uint8_t	1 = P9415 output voltage is out of range	
	WLPF_BFE	uint8_t	1 = Battery pack voltage is lower than 35V	
	WLPF_PROTECTION	uint8_t	1 = Wireless Power check failed and was protected	
Sys_ErrorCodeHighByte		uint8_t	Not used in current version	N/A
Sys_Status		Sys_status_t		0x30
	WirelessPresent	uint8_t	1 = Wireless Power was confirmed to be connected	
	LineInPresent	uint8_t	1 = Wired Power was confirmed to be connected	
	ChargingAllowed	uint8_t	1 = Charging the device is allowed	
	DischargingAllowed	uint8_t	1 = Discharging (running the motor) is allowed	
	MCPresent	uint8_t	1 = Motor was confirmed to be connected	
	AuxPresent	uint8_t	1 = RX23W device was confirmed to be connected	
	DFETstatus	uint8_t	1 = Discharge FET is enabled	
	CFETstatus	uint8_t	1 = Charge FET is enabled	
Sys_OCVFuelGauge		OCVFuelGauge_t		0x00
	Percent_100	uint8_t	1 = Minimum battery cell value is at 100% (full)	
	Percent_100to75	uint8_t	1 = Minimum battery cell value is between 75 and 100%	

	Percent_75to50	uint8_t	1 = Minimum battery cell value is between 50 and 75%	
	Percent_50to25	uint8_t	1 = Minimum battery cell value is between 25 and 50%	
	Percent_25to0	uint8_t	1 = Minimum battery cell value is between 0 and 25%	
Sys_control		Sys_control_t		0x00
	SystemONOFF	uint8_t	1 = Motor is enabled by Bluetooth app	
	ResetSys2Default	uint8_t	Not used in current version	N/A
	ClearAllFaults	uint8_t	Not used in current version	N/A
	BFEScanMode	uint8_t	Not used in current version	N/A
	DFTONorOFF	uint8_t	Not used in current version	N/A
	CFETONorOFF	uint8_t	Not used in current version	N/A
Bfe_OWStatus		uint16_t	Reg 0x68 Not used in current version	N/A
Bfe_Temp		uint16_t	Measures maximum temperature (Etaux1 or Etaux0)	0x0000
Bfe_VcellMax		uint16_t	Maximum cell voltage across cells	0x0000
Bfe_VcellMin		uint16_t	Minimum cell voltage across cells	0x0000
Bfe_MCellnum		uint16_t	Maximum and minimum number of cells	0x0000
Bfe_Vcell1		uint16_t	Reg 0x30 Measures voltage of cell 1	0x0000
Bfe_Vcell2		uint16_t	Reg 0x32 Measures voltage of cell 2	0x0000
Bfe_Vcell3		uint16_t	Reg 0x34 Measures voltage of cell 3	0x0000
Bfe_Vcell4		uint16_t	Reg 0x36 Measures voltage of cell 4	0x0000
Bfe_Vcell5		uint16_t	Reg 0x38 Measures voltage of cell 5	0x0000
Bfe_Vcell6		uint16_t	Reg 0x3A Measures voltage of cell 6	0x0000
Bfe_Vcell7		uint16_t	Reg 0x3C Measures voltage of cell 7	0x0000
Bfe_Vcell8		uint16_t	Reg 0x3E Measures voltage of cell 8	0x0000

Bfe_Vcell9		uint16_t	Reg 0x40 Measures voltage of cell 9	0x0000
Bfe_Vcell10		uint16_t	Reg 0x42 Measures voltage of cell 10	0x0000
Bfe_Vcell11		uint16_t	Reg 0x44 Measures voltage of cell 11	0x0000
Bfe_Vcell12		uint16_t	Reg 0x46 Measures voltage of cell 12	0x0000
Bfe_Vcell13		uint16_t	Reg 0x48 Measures voltage of cell 13	0x0000
Bfe_Vcell14		uint16_t	Reg 0x4A Measures voltage of cell 14	0x0000
Bfe_Vcell15		uint16_t	Reg 0x4C Measures voltage of cell 15	0x0000
Bfe_Vcell16		uint16_t	Reg 0x4E Measures voltage of cell 16	0x0000
Bfe_DeltaVcell		uint16_t	Reg 0x50 Difference between maximum and minimum cell voltages	0x0000
Bfe_Vpack		uint16_t	Reg 0x5C Measures total battery pack voltage	0x0000
Bfe_Timer		unsigned long	Reg 0x54 Measures time stamps for current measurements	0x0000
Bfe_Ipack		uint16_t	Reg 0x52 Measures the current across the sense resistor	0x0000
Bfe_Ireg		uint16_t	Reg 0x61 Measures reference voltage across the sense resistor between EMITTER and VDD	0x0000
Bfe_Etaux0		uint16_t	Reg 0x58 External temperature monitor for battery pack	0x0000
Bfe_Etaux1		uint16_t	Reg 0x5A External temperature monitor for battery pack	0x0000
Bfe_InternalTemp		uint8_t	Reg 0x5E Reports internal temperature of the ISL94216	0x0000
Bfe_VTEMP		uint8_t	Reg 0x5F Reference voltage for Etaux pins to monitor battery pack temperature	0x0000
Bfe_VCC		uint8_t	Reg 0x60 Internal measurement of VCC pin voltage	0x0000

Bfe_0to25Threshold		uint16_t	Threshold voltage to indicate charge level of battery pack (for 0-25% charge)	0xACE1
Bfe_25to50Threshold		uint16_t	Threshold voltage to indicate charge level of battery pack (for 25-50% charge)	0xBDE0
Bfe_50to75Threshold		uint16_t	Threshold voltage to indicate charge level of battery pack (for 50-75% charge)	0xD1CA
Bfe_75to100Threshold		uint16_t	Threshold voltage to indicate charge level of battery pack (for 75-100% charge)	0xE5B5
Wlp_ChipID		uint16_t	Reg 0x00 Chip ID for the wireless power device	0x9415
Wlp_HardwareID		uint8_t	Reg 0x02 Hardware ID for the wireless power device	0x0000
Wlp_CustomerID		uint8_t	Reg 0x04 Customer ID	0x0000
Wlp_StatusRegister_A		uint16_t	Reg 0x2C Status of LDO, OTP, OVP, OCP, etc.	0x0000
Wlp_StatusRegister_B		uint16_t	Reg 0x2D Status of ID and device authentication, etc.	0x0000
Wlp_InterruptRegister_A		uint16_t	Reg 0x30 Notifies of interrupts for LDO, OTP, OVP, OCP, etc.	0x0000
Wlp_InterruptRegister_B		uint16_t	Reg 0x31 Notifies of interrupts for ID and device authentication, etc.	0x0000
Wlp_InterruptRegister		uint16_t	Not used in current version	N/A
Wlp_IntEnRegister		uint16_t	Reg 0x34 Enables/disables the above interrupt registers	0x0000
Wlp_ReadVout		uint16_t	Reg 0x3C 12-bit ADC reading of main LDO Vout value	0x0000
Wlp_ReadIout		uint16_t	Reg 0x44 Current reading of Iout	0x0000
Wlp_ReadTemperature		uint16_t	Reg 0x46 Temperature reading of the die in 12-bit ADC value	0x0000
Wlp_RxFlag		uint16_t	Not used in current version	N/A
Chg_info		Chg_info_t		0x00

	LineInPresent	uint8_t	Confirms that RX23W auxiliary device is connected	0x00
	WLPPresent	uint8_t	Confirms that P9415 wireless power device is connected	0x00
	WL_Cntrl	uint8_t	Enables/disables the wireless power device	0x00
	W_Cntrl	uint8_t	Enables/disables the wired power input	0x00
	AnalogMux	uint8_t	Not used in current version	N/A
Chg_Vin		uint16_t	Voltage reading for the charger voltage input ADC	0x0000
Chg_Vout		uint16_t	Voltage reading for the charger voltage output ADC	0x0000
Chg_Iin		uint16_t	Current reading for the charger current input ADC	0x0000
Chg_Iout		uint16_t	Current reading for the charger current output ADC	0x0000
Mtr_throttle		uint16_t	Not used in current version	N/A
Mtr_speed		uint16_t	Not used in current version	N/A
Mtr_rotate_speed		uint16_t	Not used in current version	N/A
Mtr_temp		uint16_t	Not used in current version	N/A

Table 12: System Variables

14. Appendix B BLE Operation Guide

This section is a step-by-step guide on how to set up and run the project, as well as how to set up the phone app for communication. This section assumes that you have a Target Board for RX23W, Smart Phone, 2 micro-USB cables and have installed e^2 studio.

The phone app described in this example will be the Renesas GATTBrowser app. Refer to the GATTBrowser Application Note in Section 3 for more details.

14.1. 1.1. Setup

For a detailed guide on how to use e^2 studio and install projects onto the Target Board for RX23W, refer to RX Family QE for BLE[RX] R_BLE Script sample and dedicated program Application Notes in Section 3.

- Open e^2 studio, and import and compile the project
- There are 2 DIP switches (ESW1) on the RX23W board which are currently switched OFF. Turn Switch 2 to ON to enable debugging/program loading
- There are 2 micro-USB connectors on the RX23W board which can both be used to power the board

- Connect a micro-USB cable between the Host PC and the ECN1 connector (for connecting to the on-board emulator (E2 J-link Lite)), and select Debug in e^2 studio to load the program onto the board
- The RX23W board can now be disconnected from the Host PC (or used to power the board) and Switch 2 can be turned OFF so that the program runs automatically when powered on
    - Alternatively, the program can be run from e^2 studio in Debug mode
- The program will start automatically, but pressing the RESET button at any time will reset the device and restart the program
- (Optional) Connect a micro-USB cable between the Host PC and the CN5 connector to allow for serial connection and open a serial emulator (terminal) such as Tera Term to view Bluetooth events
    - The terminal must be configured according to Section 2.8.6 of the RX23W Group BLE Module Firmware Integration Technology Application Note
- This concludes the setup. Once the program has been loaded, power the board with a micro-USB, launch a terminal (optional) and open the GATTBrowser app on the Smart Phone to begin

14.2. 1.2. How to Use

- When power is supplied to the RX23W board, it begins Bluetooth advertising automatically
- Select ‘Scan’ from the GATTBrowser app to view all of the Bluetooth devices in range that are advertising
- The RX23W board is advertised as ‘SCOOTER’, with the Bluetooth address ‘74:90:50:FF:FF:FF’
- Select the ‘SCOOTER’ device from GATTBrowser by tapping the ‘play’ button
- Once connected, the services are discovered and UUIDs for the available services can be seen.
- There are 3 custom characteristics which can be selected by their UUIDs at the bottom:
    - The first one (35ada…) contains continuous data (which would be battery, motor speed, etc.) which can be notified to the app from the RX23W board
    - The second (68e…) contains static data (device ID, firmware, etc.) which can be also sent to the app from the RX23W board
    - The third (5f5…) allows you to write data to the board with hex values such as ’00 01 03 05’ and read those values back
- Select the second service, tap the button to turn notifications on, and press the switch (SW1) on the RX23W board to send static data
    - Alternatively, data can be sent via I^{2}C from a Host Device if it is written to the device
- Press the back button on the Smart Phone and select the first service, turn notifications on and press the switch on the RX23W board again. Confirm that it is also sending continuous data that is being incremented every second (only the first 20 bytes can be viewed)
    - Alternatively, data can be sent via I^{2}C from a Host Device if it is written to the device
- Pressing the switch repeatedly will turn the timer off and on again
- Press the back button on the Smart Phone and select the third service. If a value of 80 is written in the first byte of the third service, LED1 on the RX23W board will turn on. Sending any other value will turn the LED off

- To reset the program, disconnect the device from the Smart Phone and press the RESET switch on the RX23W board, or terminate the program in e^2 studio and select Debug again

For the custom mobile app, refer to the Mobile App section for an overview of features and how to use the app with the complete setup. The switches on the RX23W are not needed, as the system will automatically begin to send data once connected.

In e^2 studio, all of the data that is sent and received via I^{2}C/BLE, as well as error and control messages, are printed to the Renesas Virtual Debug Console and can be viewed if the device is being run by the on-board debugger.

15. Appendix C: Motherboard (RX23T) System Routines

The folder and file configurations of the project programs are given below.

Project scr folder	Sub Folder	Sub Folder	Files	Description
src of US069_48V_Power_train_for_Scooter	Auxiliary		Auxi.c	Communication source file with RX23W
			Auxi.h	Communication header file with RX23W
	BFE		ISL94216.c	BFE related routine
			ISL94216.h	Header file of BFE module
	Charger		charger.c	Source file of charger related functions
			charger.h	Header file of charger module
	Hardware	Config_ S12AD0	Config_S12AD0_ chg.c	Source file of 12-bit A/D converter for charger module
			Config_S12AD0_ chg.h	Header file of 12-bit A/D converter for charger module
		Config_ICU	Config_ICU_chg. c	Source file of interrupt controller for charger module
			Config_ICU_chg. h	Header file of interrupt controller for charger module
	Motor	inc	motor.h	Header file of motor control
			--other h files are omitted--	Header file of internal c files of motor control
		--	motor.c	Top file of motor control, includes motor initialization setting, motor check, motor start/stop etc.
		--	--other c files are omitted--	Internal c files of motor control
	smc_gen	--omitted		MCU peripherals settings by FIT
	System		system.c	System process source file including system initialization, system check, system fault process etc
			system.h	Definition for system and variable for system
	Wlp		Wlp.c	Source file of wireless charger related functions
			Wlp.h	Header file of wireless charger module
	--		US069_48V_Pow er_train_for_Sco oter.c	main routine file
	--		debug.c	For debugging use

15.1. System

System Hardware Initialization Routine

sys_hardware_init()	
Outline	System hardware initialization
Declaration	void sys_hardware_init(void)
Description	Initialize all the hardware that charger and BFE used and start them
Call Function	R_Systeminit_user() bfe_hw_start() Auxi_hw_start()
Argument	None
Return Value	None

System Variable Initialization Routine

sys_var_init ()	
Outline	System variable initialization
Declaration	void sys_var_init(void)
Description	Initialize all the global variables
Call Function	None
Argument	None
Return Value	None

System Module Initialization Routine

sys_modules_init ()	
Outline	System initialization
Declaration	void sys_modules_init (void)
Description	System initialization charger module,BFE module and auxillary (RX23W) module
Call Function	bfe_init () chg_init () Auxi_init()
Argument	None
Return Value	None

System Initialization Routine

sys_init()	
Outline	System initialization
Declaration	void sys_init(void)
Description	System initialization which include hardware initialization, variable initialization and modules initialization
Call Function	sys_hardware_init () sys_var_init () sys_modules_init()
Argument	None
Return Value	None

System Check Routine

sys_check()	
Outline	System check
Declaration	void sys_check(void)
Description	System check wireless charger, wired charger and Auxillary (RX23W) to confirm they are present or not.
Call Function	bfe_setup() bfe_charge_pump_on() wlp_wireless_charger_check() chg_wired_charger_check() auxi_check()
Argument	None
Return Value	None

System Update

sys_update()	
Outline	System update data
Declaration	void sys_update(void)
Description	System update data for charger, BFE
Call Function	chg_reg_update() wlp_reg_update() bfe_update_bfe_variables() Auxi_send_data()
Argument	None
Return Value	None

System Charging

sys_charging()	
Outline	System charge procedure
Declaration	void sys_charging(void)
Description	System execute charging operation
Call Function	chg_charger_shut_down() chg_charging_procedure() wlp_wireless_charger_monitor() wlp_charging_procedure() chg_wired_charger_connect() wlp_wireless_charger_connect()
Argument	None
Return Value	None

System Fault Process

sys_FaultProcess()	
Outline	System fault process
Declaration	void sys_FaultProcess(void)
Description	System fault process which is called by ISL94216.c
Call Function	bfe_single_scan_start() bfe_update_bfe_variables() bfe_update_bfe_variables()
Argument	None
Return Value	None

System Low Power Mode

sys_LowPowerEnter()	
Outline	System low power mode
Declaration	void sys_LowPower_Enter(void)
Description	System, BFE, Rx23W are all enter to low power mode
Call Function	None
Argument	None
Return Value	None

System Normal Mode

sys_LowPowerExit()	
Outline	System normal mode
Declaration	void sys_LowPowerExit(void)
Description	System exit low power mode enter to normal mode
Call Function	bfe_cfet_on_off() bfe_single_scan_start() bfe_update_bfe_variables() Auxi_enable() Auxi_send_data()
Argument	None
Return Value	None

15.2. Auxillary

Auxillary Initialization

Auxi_init ()	
Outline	Auxillary initialization
Declaration	void Auxi_init (void)
Description	Initialize control pin to connect to RX23W
Call Function	None
Argument	None
Return Value	None

Auxillary Hardware Start

Auxi_hw_start ()	
Outline	Auxillary hardware start
Declaration	void Auxi_hw_start (void)
Description	Start hardware to connect to RX23W
Call Function	R_Config_SCI5_Start()
Argument	None
Return Value	None

Auxillary Enable

Auxi_enable ()	
Outline	Auxillary enabled
Declaration	void Auxi_enable (void)
Description	Set N_CS1 to “0” to enable RX23w communication
Call Function	None
Argument	None
Return Value	None

Auxillary Disable

Auxi_disable ()	
Outline	Auxillary disabled
Declaration	void Auxi_disable (void)
Description	Set N_CS1 to “1” to disable RX23w communication
Call Function	None
Argument	None
Return Value	None

Auxillary Check

Auxi_check ()	
Outline	Auxillary check
Declaration	void Auxi_check (void)
Description	Check RX23W is present or not
Call Function	Auxi_Write()
Argument	None
Return Value	None

Auxillary Receive Data

Auxi_receive_data ()	
Outline	Receive data from RX23W
Declaration	void Auxi_receive_data (void)
Description	Receive data from RX23W
Call Function	Auxi_read()
Argument	None
Return Value	None

Auxillary Send Data

Auxi_send_data ()	
Outline	Send system variables to RW23W
Declaration	void Auxi_send_data (void)
Description	Send data to RX23W
Call Function	Auxi_write()
Argument	None
Return Value	None

15.3. BFE

Routine Name	Sub Routine and Function Included	Description
bfe_init()	Reset BFE Module	Power down RESET pin of 94216 for 300ms, then release RESET pin for 100ms.
	Get System Information of chip, store part ID and product revision to system variable (sys_data_t)	Implement first communication with 94216 using global operation register read command.
	bfe_set_reg_threshold()	Set threshold to voltage, current and temperature limit register.
	bfe_set_reg_init()	Initial other registers except threshold setting registers.
	crc_fault_get()	Check if there is any CRC error in communication.
bfe_setup()	Unmask (0) the charge pump not ready mask bit	Read out the value in Other Faults Mask(0x85) register and set the CPMP NRDY MASK to 0, then write it back to 0x85 register.
	bfe_clear_all_faults()	Clear all faults through setting Clear Faults and Status bit in V_{CELL} Operation register(0x02) to 1.
	bfe_single_scan_start()	Start a single scan
	bfe_read_all_faults_status()	Read all Faults and Status (0x63 to 0x69)
	bfe_fault_handle()	Call faults handling routine
bfe_cb_setup()	bfe_insert_fetsoff()	Assert FETSOFF to low to allow the power FETs to turn on.
	bfe_cal_cur_vol() bfe_vol_cnt() bfe_write_reg_value()	Change COC Threshold to 2.2A.
	bfe_read_reg_one_value() bfe_write_reg_value()	Set CB_EN, AUTO_CB and CB_CHRG, IEOC EN and CB EOC to 1.
bfe_update_bfe_variables()	Check if the chip is not busy.	Read busy bit in Global Operation register(0x01) to confirm current status of chip
	bfe_read_all_faults_status() bfe_read_all_measurement()	If the chip is not busy, read out all faults and status register's value, and all measured value stored in related registers.
	bfe_read_c_dfet_status()	Update the current status of CFET and DFET.
	bfe_vmax_vin_get()	Get maximal and minimal Vcell while recording its position. Then update the g_sys_data variable.
	bfe_max_etaux_get()	Get the register value corresponding to the higher temperature in two Etaux. Then update the g_sys_data variable.
	bfe_battery_level_get()	Get current battery level: 0%~25%, 25%~50%, 50%~75%, 75%~100%

Initial routine

bfe_init()

Outline	Initial BFE module	
Declaration	uint8_t bfe_init()	
Description	Reset chip, confirm connection by reading system information. Set register's threshold, and set register initialization value	
Call Function	bfe_read_reg_one_value() bfe_set_reg_threshold() bfe_set_reg_init() crc_fault_get()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_set_reg_threshold()

Outline	Set threshold value to chip's register	
Declaration	uint8_t bfe_set_reg_threshold()	
Description	Implement threshold setting to related registers.	
Call Function	bfe_write_reg_value() bfe_vol_cnt_integer() bfe_cal_res_deg() bfe_cal_vol_res_simple() bfe_cal_vol_deg() bfe_read_reg_one_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_set_reg_init()

Outline	Set initial value to chip's register	
Declaration	uint8_t bfe_set_reg_init()	
Description	Implement initial setting to related registers.	
Call Function	bfe_write_reg_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

Initial value setting table:

94216 Register Addr	Setting in Hex	Note
0x01	0x00	
0x02	0xE0	
0x03	0xC2	
0x04	0xFC	
0x05	0xFF	
0x09	0x51	
0x0E	0x8C	
0x11	0xC0	
0x12	0x30	
0x1B	0x52	Set LP REG for strong regulator
0x1F	0xCC	Disable Comm TO for debug (bit1)
0x24	0x5C	
0x25	0x2F	
0x28	0x80	~512ms CBON
0x29	0x0C	~48ms CBOFF
0x2E	0x2B	
0x83	0x00	
0x84	0x00	
0x85	0x3F	
0x86	0xD7	
0x87	0xF0	
0x88	0x00	
0x89	0x00	

bfe_setup()		
Outline	BFE setup routine	
Declaration	uint8_t bfe_setup()	
Description	Clean all faults, trigger a single scan, handle faults.	
Call Function	bfe_read_reg_one_value() bfe_write_reg_value() bfe_clear_all_faults() bfe_single_scan_start() bfe_read_all_faults_status() bfe_fault_handle()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_cb_setup()		
Outline	Set up BFE cell balancing function	
Declaration	void bfe_cb_setup()	
Description	Assert FETSOFF, change COC threshold and set cell balancing register.	
Call Function	bfe_insert_fetsoff()	
	bfe_cal_cur_vol()	
	bfe_vol_cnt()	
	bfe_read_reg_one_value()	
	bfe_write_reg_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

Operation routine

bfe_insert_fetsoff()		
Outline	Control FETSOFF pin	
Declaration	void bfe_insert_fetsoff(bool low_high)	
Description	Control GPIO1 pin.	
Call Function	None	
Argument	bool low_high	Level output to GPIO1 pin
Return Value	None	

bfe_insert_wakeup()		
Outline	Wake up chip from ship or low power to idle mode	
Declaration	void bfe_insert_wakeup(bool low_high)	
Description	Pull down WAKEUP pin for changing mode from ship or low power to idle.	
Call Function	None	
Argument	bool low_high	Level output to WAKEUP pin
Return Value	None	

bfe_clear_all_faults()		
Outline	Clear faults and status	
Declaration	uint8_t bfe_clear_all_faults()	
Description	Set Vcell operation register(0x02) to clear faults and status.	
Call Function	bfe_read_reg_one_value()	
	bfe_write_reg_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_single_scan_start()

Outline	Start a single scan
Declaration	void bfe_single_scan_start()
Description	Set SYS Scan SEL and SYS Scan Trigger to start a single scan.
Call Function	bfe_read_reg_one_value() bfe_write_reg_value() bfe_change_mode()
Argument	None
Return Value	None

bfe_continue_scan_start()

Outline	Start a continuous scan procedure
Declaration	void bfe_continue_scan_start()
Description	Set SYS Scan SEL and SYS Scan Trigger to start a continuous scan.
Call Function	bfe_read_reg_one_value() bfe_write_reg_value()
Argument	None
Return Value	None

bfe_scan_start()

Outline	Trigger a system scan	
Declaration	void bfe_scan_start()	
Description	Start scan with its original mode(single or continue).	
Call Function	bfe_read_reg_one_value() bfe_write_reg_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_scan_finish()

Outline	Check if the scan completed.	
Declaration	void bfe_scan_finish()	
Description	Check BUSY bit in Global Operation register and confirm if the scan completed.	
Call Function	bfe_read_reg_one_value()	
Argument	None	
Return Value	bool	Scan finished(0), Scan is going on(1)

bfe_charge_pump_on()

Outline	Enable the charge pump	
Declaration	void bfe_charge_pump_on()	
Description	Enable the charge pump, which provides a biasing voltage to CB16. Wait for charge pump not ready bit cleared.	
Call Function	bfe_read_reg_one_value()	
	bfe_write_reg_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_cfet_on_off()

Outline	Turn on or off CFET	
Declaration	uint8_t bfe_cfet_on_off()	
Description	Because CFET and DFET cannot be opened simultaneously, before turning on CFET, turn off DFET first.	
Call Function	bfe_read_reg_one_value()	
	bfe_write_reg_value()	
	bfe_read_c_dfet_status()	
Argument	bool on_off	Turn on or off CFET
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_dfet_on_off()

Outline	Turn on or off DFET	
Declaration	uint8_t bfe_dfet_on_off()	
Description	Because CFET and DFET cannot be opened simultaneously, before turning on DFET, turn off CFET first.	
Call Function	bfe_read_reg_one_value()	
	bfe_write_reg_value()	
	bfe_read_c_dfet_status()	
Argument	bool on_off	Turn on or off CFET
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_read_c_dfet_status()

Outline	Read current CFET and DFET gate's status	
Declaration	uint8_t bfe_read_c_dfet_status()	
Description	Read out the current C/DFET gate's status and update related status bit in g_sys_data variable.	
Call Function	bfe_read_reg_one_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_change_mode()

Outline	Mode change routine	
Declaration	void bfe_fault_handle()	
Description	Change to a specified mode by modifying Scan Operation register(0x2E).	
Call Function	bfe_read_reg_one_value()	
	bfe_write_reg_value()	
Argument	uint8_t cur_mode	Mode needs to be changed to
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_fault_handle()

Outline	BFE fault handling routine
Declaration	void bfe_fault_handle()
Description	Using different handling process according to different fault.
Call Function	clear_fault_bit()
	bfe_dfet_on_off()
	bfe_change_mode()
Argument	None
Return Value	None

bfe_update_bfe_variables()

Outline	Update the current faults and status, register value after measurement, while updating g_sys_data variable.	
Declaration	uint8_t bfe_update_bfe_variables()	
Description	If BUSY bit in Global Operation register is 0 which means measurement completed, read out all faults and status and register value used for storing measured voltage, current and temperature. After related maximum and minimum are calculated, update g_sys_data variable.	
Call Function	bfe_read_reg_one_value()	
	bfe_read_all_faults_status()	
	bfe_read_all_measurement()	
	bfe_read_c_dfet_status()	
	bfe_vmax_vin_get()	
	bfe_max_etaux_get()	
	bfe_battery_level_get()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_cb_auto()

Outline	Enable auto cell balancing or not	
Declaration	uint8_t bfe_cb_auto(bool on_off)	
Description	Turn on or off auto cell balancing function.	
Call Function	bfe_read_reg_one_value()	
	bfe_write_reg_value()	
Argument	bool on_off	Auto cell balancing turn on or off
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_cb_enable_disable()		
Outline	Enable cell balancing function or not	
Declaration	uint8_t bfe_cb_enable_disable(bool on_off)	
Description	Enable or disable cell balancing function.	
Call Function	bfe_read_reg_one_value()	
	bfe_write_reg_value()	
Argument	bool on_off	Cell balancing turn on or off
Return Value	uint8_t status	MD_OK(0), MD_ERROR

clear_fault_bit()		
Outline	Clear fault bit routine	
Declaration	uint8_t clear_fault_bit(uint8_t addr, uint8_t fault_bit)	
Description	Clear a specified fault bit.	
Call Function	bfe_read_reg_one_value()	
	bfe_write_reg_value()	
Argument	uint8_t address	The address of the fault bit
	uint8_t fault_bit	Fault bit
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_aux_open_wire_test()		
Outline	Etaux open-wire test routine	
Declaration	uint8_t bfe_aux_open_wire_test()	
Description	Connect internal switch of Sow which pull up xTx pin to VCC (through RETAUX) to test if open-wire occurs.	
Call Function	bfe_read_reg_one_value()	
	bfe_write_reg_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR_XT_OPEN_W

Note: This routine is not called in this project which means there is no Etaux open-wire function in this application. Please confirm if this is necessary for this application.

Get measurement and status routine

crc_fault_get()

Outline	Read CRC Fault from chip	
Declaration	bool crc_fault_get()	
Description	Read out CRCF from Other Faults register(0x65).	
Call Function	bfe_read_reg_one_value()	
Argument	None	
Return Value	bool status	No fault(0), Has fault(1)

bfe_read_all_faults_status()

Outline	Read out all faults and status value	
Declaration	uint8_t bfe_read_all_faults_status()	
Description	All faults and status(0x63~0x69) are read out.	
Call Function	bfe_read_reg_multi_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

Communicating routine

bfe_read_reg_one_value()

Outline	Read one value from one register	
Declaration	uint8_t bfe_read_reg_one_value(uint8_t addr, uint8_t* rcv_data)	
Description	Read one register's value from a specified address.	
Call Function	bfe_crc_calculate()	
	bfe_spi_read()	
Argument	uint8_t addr	Register's address
	uint8_t * rcv_data	The address of the reception buffer
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_read_reg_multi_value()

Outline	Read multiple register value by command	
Declaration	uint8_t bfe_read_reg_multi_value(uint8_t cmd, uint8_t* rcv_data, uint8_t rcv_num)	
Description	Read multiple register's value by different command.	
Call Function	bfe_crc_calculate()	
	bfe_spi_read()	
Argument	uint8_t cmd	Sequential read command code
	uint8_t* rcv_data	The address of the reception buffer
	uint8_t rcv_num	Number needs to be read out
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_spi_read()		
Outline	Read data from register using SPI	
Declaration	uint8_t bfe_spi_read(uint32_t* const snd_buf, uint16_t const length, uint32_t* const rcv_buf)	
Description	Implement the timing of reading one value from a specified register.	
Call Function	R_Config_RSPI0_Send_Receive()	
	R_BSP_SoftwareDelay()	
Argument	uint32_t* const snd_buf	the address of sending buffer
	uint16_t const length	the number of bytes needs to be sent
	uint32_t* const rcv_buf	the address of the reception buffer
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_write_reg_value()		
Outline	Write one value to one register	
Declaration	uint8_t bfe_write_reg_value(uint8_t addr, uint8_t* rcv_data)	
Description	Write one value to a specified register.	
Call Function	bfe_crc_calculate()	
	bfe_spi_write()	
Argument	uint8_t addr	Register's address
	uint8_t snd_data	Register's value
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_spi_write()		
Outline	Read data from register using SPI	
Declaration	uint8_t bfe_spi_write(uint32_t* const snd_buf, uint16_t const length, uint32_t* const rcv_buf)	
Description	Implement the timing of writing one value to a specified register.	
Call Function	R_Config_RSPI0_Send_Receive()	
	R_BSP_SoftwareDelay()	
Argument	uint32_t* const snd_buf	the address of sending buffer
	uint16_t const length	the number of bytes needs to be written
	uint32_t* const rcv_buf	the address of the reception buffer
Return Value	uint8_t status	MD_OK(0), MD_ERROR

Calculation routine

bfe_vol_cnt()		
Outline	Calculate register's value from voltage	
Declaration	uint8_t bfe_vol_cnt(float init, double offset, double step)	
Description	Calculated integer value from a specified voltage, offset, and step.	
Call Function	None	
Argument	float init	Voltage
	double offset	The fixed offset or minimum value
	double step	Step voltage for one counter
Return Value	uint8_t	Calculated register's value

bfe_cnt_vol()

Outline	Calculate voltage according to register's value	
Declaration	float bfe_cnt_vol(uint16_t reg_val, double step)	
Description	Calculated voltage from register's value and step.	
Call Function	None	
Argument	uint16_t reg_val	Register's value
	double step	Step voltage for one counter
Return Value	float	Calculated voltage

bfe_cal_deg_res()

Outline	Calculate resistor from temperature	
Declaration	double bfe_cal_deg_res(double cur_deg)	
Description	Calculated resistor from external NTC's Degree Celsius.	
Call Function	None	
Argument	double cur_deg	Degree celsius
Return Value	double	Calculated resistor's value

bfe_cal_res_vol()

Outline	Calculate voltage from resistor	
Declaration	double bfe_cal_res_vol(double cal_res)	
Description	Calculate voltage from resistor value of external NTC.	
Call Function	None	
Argument	double cal_res	Resistance value
Return Value	double	Calculated voltage

bfe_cal_int_deg_cnt()

Outline	Calculate register's value from internal temperature	
Declaration	uint8_t bfe_cal_int_deg_cnt(double deg_val)	
Description	Calculate register's value from Degree Celsius of internal NTC resistor.	
Call Function	None	
Argument	double deg_val	Degree Celsius
Return Value	uint8_t	Calculated register value

bfe_cal_int_cnt_deg()

Outline	Calculate internal temperature from register's value	
Declaration	float bfe_cal_int_cnt_deg(uint8_t reg_val)	
Description	Calculate degree celsius of internal NTC resistor from register's value.	
Call Function	None	
Argument	uint8_t reg_val	Register's value
Return Value	float	Calculated degree Celsius

bfe_cal_ext_cnt_deg()

Outline	Calculate external NTC's temperature from register's value	
Declaration	float bfe_cal_int_cnt_deg(uint8_t reg_val)	
Description	Calculate external NTC's Degree Celsius from register's reading value.	
Call Function	None	
Argument	uint16_t reg_val	Register's value
Return Value	float	Calculated external NTC's degree Celsius

bfe_cal_cur_vol()

Outline	Calculate voltage from current	
Declaration	double bfe_cal_cur_vol(double cal_res)	
Description	Calculate voltage from current.	
Call Function	None	
Argument	float cur_val	current value
	float res_val	resistance value
Return Value	double	Calculated voltage

bfe_vmax_vin_get()

Outline	Get maximum and minimum in 16 Vcell
Declaration	void bfe_vmax_vin_get()
Description	Compare all Vcell, get the maximum and minimum, and store in g_sys_data.
Call Function	None
Argument	None
Return Value	None

bfe_max_etaux_get()

Outline	Get the register value corresponding to the higher temperature in two Etaux
Declaration	void bfe_max_etaux_get()
Description	Compare Etaux0 and Etaux1, get the maximal(temperature maximum), then store in g_sys_data varialbe.
Call Function	None
Argument	None
Return Value	None

bfe_battery_level_get()

Outline	Get current battery power of 16 cells
Declaration	void bfe_battery_level_get()
Description	Get current battery power of 16 cells, display its position in g_sys_data variable.
Call Function	None
Argument	None
Return Value	None

bfe_ipack_current_get()		
Outline	Calculate Ipack from counter in register	
Declaration	float bfe_ipack_current_get()	
Description	According to Ipack count in g_sys_data variable, calculates actual current of Ipack.	
Call Function	None	
Argument	None	
Return Value	float	Calculated Ipack value

Routine only for debug

bfe_read_all_register()		
Outline	Read out all register value	
Declaration	uint8_t bfe_read_all_register()	
Description	All register(0x00~0x89) will be read out.	
Call Function	bfe_read_reg_multi_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_read_ow_status()		
Outline	Read out 16 Vcell open-wire status	
Declaration	uint8_t bfe_read_ow_status()	
Description	Get responsible for reading out open-wire status of 16 Vcell(0x68~0x69).	
Call Function	bfe_read_reg_one_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_read_all_vcell_vol()		
Outline	Read out Vcell voltage and Vcell max delta voltage	
Declaration	uint8_t bfe_read_all_vcell_vol()	
Description	Get responsible for reading out 16 Vcell and Vcell max delta voltage (0x31~0x51).	
Call Function	bfe_read_reg_multi_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_read_vpack_value()		
Outline	Read out VBAT1 voltage	
Declaration	uint8_t bfe_read_vpack_value()	
Description	Get responsible for reading out VBAT1 voltage(0x5C~0x5D).	
Call Function	bfe_read_reg_one_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_read_ipack_value()

Outline	Read out Ipack voltage and Ipack Timer	
Declaration	uint8_t bfe_read_ipack_value()	
Description	Get responsible for reading out Ipack voltage(0x52~0x53) and Ipack Timer(0x54~0x57).	
Call Function	bfe_read_reg_multi_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_read_other_value()

Outline	Read out other voltage	
Declaration	uint8_t bfe_read_other_value()	
Description	Get responsible for reading out other voltage(0x58~0x62) which includes ETAUX0/1, BVAT1, Internal Temperature, VTEMP, Vcc and Ireg voltage.	
Call Function	bfe_read_reg_multi_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_read_ireg_value()

Outline	Read out Ireg voltage	
Declaration	uint8_t bfe_read_ireg_value()	
Description	Get responsible for reading out Ireg voltage(0x61~0x62).	
Call Function	bfe_read_reg_one_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_read_itemp_value()

Outline	Read out Internal Temperature and VTEMP Voltage	
Declaration	uint8_t bfe_read_itemp_value()	
Description	Get responsible for reading out internal temperature(0x5E) and VTEMP voltage(0x5F).	
Call Function	bfe_read_reg_one_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

bfe_read_all_measurement()

Outline	Read out all register value	
Declaration	uint8_t bfe_read_all_measurement()	
Description	Get responsible for reading out all measurement result(0x30~0x62) registers.	
Call Function	bfe_read_reg_multi_value()	
Argument	None	
Return Value	uint8_t status	MD_OK(0), MD_ERROR

15.4. Wired Charger

Charger Initialization

chg_init()	
Outline	Charger initialization
Declaration	void chg_init(void)
Description	Initialize the charger related ports.
Called Function	chg_wired_charger_check() chg_charger_shut_down() sys_module_init()
Argument	None
Return Value	None

PWM Output Start

chg_pwm_start()	
Outline	Start PWM output
Declaration	void chg_pwm_start(void)
Description	Start PWM output
Called Function	chg_wired_charger_connect() wlp_wireless_charger_connect()
Argument	None
Return Value	None

PWM Output Stop

chg_pwm_stop()	
Outline	Stop PWM output
Declaration	void chg_pwm_stop(void)
Description	Stop PWM output
Called Function	chg_init()
Argument	None
Return Value	None

Multiplexer Input Switch

chg_mux_in_switch()		
Outline	Switch analog inputs	
Declaration	void chg_mux_in_switch(chg_mux_switch_t addr_ai)	
Description	Switch analog inputs to measure wired voltage, charger output voltage, input current or output current.	
Called Function	chg_wired_charger_check() chg_wired_charger_connect() chg_reg_update() wlp_wireless_charger_connect() wlp_protection_check()	
Argument	addr_ai	CHG_IN_CHRGPLUS CHG_IN_II_SENSE CHG_IN_IO_SENSE CHG_IN_WIRED_AO
Return Value	None	

A/D and IRQ1 Switch

chg_mux_out_switch()		
Outline	Switch P11 functions	
Declaration	void chg_mux_out_switch(chg_mux_switch_t pin_function)	
Description	Select P11 as A/D analog input port or IRQ1 detection port, or select internal reference for A/D conversion.	
Called Function	chg_wired_charger_check() chg_wired_charger_connect() chg_charger_shut_down() chg_reg_update() chg_charging_procedure() wlp_wireless_charger_connect() wlp_protection_check()	
Argument	pin_function	CHG_OUT_AN016 CHG_OUT_IRQ1 CHG_INTERNAL
Return Value	None	

IRQ1 Detection Type Set and IRQ1 Start

chg_irq1_detection_type_set()		
Outline	Set IRQ1 detection type and start IRQ1	
Declaration	void chg_irq1_detection_type_set(chg_irq_detection_type_t type)	
Description	Select falling edge or rising edge as IRQ1 detection type, and start IRQ1	
Called Function	chg_wired_charger_check()	
	chg_charger_shut_down()	
	r_Config_ICU_irq1_interrupt()	
Argument	type	CHG_IRQ_EDGE_FALLING
		CHG_IRQ_EDGE_RISING
Return Value	None	

Measure MCU VCC

chg_internal_measure()	
Outline	Measure VCC of MCU
Declaration	void chg_internal_measure(void)
Description	Measure VCC voltage by A/D internal reference
Called Function	chg_wired_charger_check()
	chg_wired_charger_connect()
	chg_reg_update()
	chg_charging_procedure()
	wlp_wireless_charger_connect()
	wlp_protection_check()
Argument	None
Return Value	None

Measure VIN

chg_sense_vin()		
Outline	Measure VIN	
Declaration	void chg_sense_vin (float *vin)	
Description	Measure VIN through AN016	
Called Function	chg_wired_charger_check()	
	chg_wired_charger_connect()	
	chg_reg_update()	
Argument	*vin	Store the voltage value of VIN
Return Value	None	

Measure VOUT

chg_sense_vout()		
Outline	Measure VOUT	
Declaration	void chg_sense_vout (float *vout)	
Description	Measure VOUT through AN016	
Called Function	chg_wired_charger_check()	
	chg_wired_charger_connect()	
	chg_reg_update()	
Argument	*vout	Store the voltage value of VOUT
Return Value	None	

Measure IIN

chg_sense_iin ()		
Outline	Measure IIN	
Declaration	void chg_sense_iin (float *iin)	
Description	Measure IIN through AN016	
Called Function	chg_wired_charger_check()	
	chg_wired_charger_connect()	
	chg_reg_update()	
Argument	*iin	Store the current value of IIN
Return Value	None	

Measure IOUT

chg_sense_iout()		
Outline	Measure IOUT	
Declaration	void chg_sense_iout (float *iout)	
Description	Measure IOUT by AN016	
Called Function	chg_wired_charger_check()	
	chg_wired_charger_connect()	
	chg_reg_update()	
Argument	*iout	Store the current value of IOUT
Return Value	None	

Set VOUT

chg_control_vout()		
Outline	Set VOUT	
Declaration	void chg_control_vout (float vout)	
Description	Control VOUT by PWM	
Called Function	chg_wired_charger_connect()	
	chg_charging_procedure()	
	wlp_wireless_charger_connect()	
Argument	vout	VOUT set value
Return Value	None	

Set IOUT

chg_control_iout()	
Outline	Set IOUT
Declaration	void chg_control_iout (float iout)
Description	Control IOUT by PWM
Called Function	chg_wired_charger_connect() wlp_wireless_charger_connect()
Argument	iout IOUT set value
Return Value	None

Wired Charger Check

chg_wired_charger_check()	
Outline	Wired charger check
Declaration	void chg_wired_charger_check(void)
Description	Check whether the wired power is plugged in after reset
Called Function	sys_check()
Argument	None
Return Value	None

Wired Charger Connect

chg_wired_charger_connect()	
Outline	Wired charger connect routine
Declaration	void chg_wired_charger_connect(void)
Description	Control ISL81801 and turn on CFET to charge the battery pack
Called Function	sys_charging
Argument	None
Return Value	None

Charger Shut Down

chg_charger_shut_down()	
Outline	Charger shut down routine
Declaration	void chg_charger_shut_down(void)
Description	Turn off CFET and clear charger status flag.
Called Function	chg_charging_procedure() sys_charging() wlp_charging_procedure()
Argument	None
Return Value	None

Wired Charger Register Update

chg_reg_update()	
Outline	Update the wired charger register for RX23W
Declaration	void chg_reg_update(void)
Description	Update VIN, VOUT, IIN, and IOUT of the charger
Called Function	sys_update()
Argument	None
Return Value	None

Wired Charger Charging Procedure

chg_charging_procedure()	
Outline	Wired charger charging procedure
Declaration	void chg_charging_procedure(void)
Description	When VEOC is set, change regulation voltage from 59.4V to 58.8V. When BATFULL is set, shut down charging process.
Called Function	sys_charging()
Argument	None
Return Value	None

15.5. Wireless Charger

Wireless Charger Check

wlp_wireless_charger_check()	
Outline	Wireless charger check
Declaration	void wlp_wireless_charger_check (void)
Description	Check whether the wireless is present after reset.
Called Function	sys_check()
Argument	None
Return Value	None

Wireless Charger Monitor

wlp_wireless_charger_monitor()	
Outline	Wireless charger monitor
Declaration	void wlp_wireless_charger_monitor (void)
Description	Monitor whether the wireless power transmitter is removed.
Called Function	sys_charging()
Argument	None
Return Value	None

Wireless Charger Connect

wlp_wireless_charger_connect()	
Outline	Wireless charger connect routine
Declaration	void wlp_wireless_charger_connect (void)
Description	Set the P9415-R, control ISL81801 and turn on CFET to charge the battery.
Called Function	sys_charging()
Argument	None
Return Value	None

Read P9415

wlp_read_p9415()		
Outline	Read P9415	
Declaration	void wlp_read_p9415(uint8 _t device_addr, uint16_t reg, uint32_t bytes_num, unsigned int * reg_val)	
Description	Communicate with P9415 to get the register values by IIC.	
Called Function	wlp_protection_check() wlp_reg_update() wlp_wireless_charger_connect()	
Argument	device_addr	WLP_P9415_A_ADDR WLP_P9415_B_ADDR
	reg	P9415 registers address
	bytes_num	Transmit data length
	*reg_val	Store the received register data
Return Value	None	

Set P9415

wlp_set_p9415()		
Outline	Set P9415	
Declaration	void wlp_set_p9415(uint8 _t device_addr, uint16_t reg, uint32_t bytes_num, uint32_t reg_val)	
Description	Communicate with P9415 to set register value by IIC.	
Called Function	wlp_wireless_charger_connect()	
Argument	device_addr	WLP_P9415_A_ADDR WLP_P9415_B_ADDR
	reg	P9415 registers address
	bytes_num	Transmit data length
	reg_val	Register set data
Return Value	None	

Wireless Charger Protection

wlp_protection_check ()	
Outline	Check wireless charger protection signals
Declaration	sys_err_code_t wlp_protection_check (void)
Description	If any protection is triggered, WLPF_PROTECTION is returned.
Called Function	sys_charging() wlp_wireless_charger_connect()
Argument	None
Return Value	SUCCESS WLPF_PROTECTION

Wireless Charger Register Update

wlp_reg_update()	
Outline	Update the wireless charger registers for RX/23W
Declaration	void wlp_reg_update (void)
Description	Update the chip ID, hardware ID, customer ID, status register, interrupt status register, VOUT register, IOUT register, and temperature register.
Called Function	sys_update()
Argument	None
Return Value	None

Wireless Charger Charging Procedure

wlp_charging_procedure()	
Outline	Wireless charger charging procedure
Declaration	void wlp_charging_procedure (void)
Description	Measure battery pack voltage, and battery charging current. If the voltage is fully charged (the voltage is close to 59.4V, within +/-1%, the current is lower than 0.1A), shut down the charging process.
Called Function	sys_charging()
Argument	None
Return Value	None

15.6. Motor

Pre-charge motor

motor_precharge ()	
Outline	Pre-charge the load capacitor of motor for not more than 200ms
Declaration	void motor_precharge (void)
Description	Pre-charge the load capacitor of motor for no more than 200ms before turning on DFET of BFE.
Called Function	None
Argument	None
Return Value	None

Initialize motor

motor_init ()	
Outline	Initialize motor
Declaration	void motor_init (void)
Description	Initialize peripherals for motor operation, variables, set initial sequence, execute reset event of motor, and set variables for board_ui mode.
Called Function	R_MTR_InitHardware() software_init() R_MTR_InitSequence() R_MTR_ExecEvent() R_MTR_AutoSetVariables()
Argument	None
Return Value	None

Check motor status

motor_check ()	
Outline	Check motor status by reading inverter bus voltage and phase voltage
Declaration	void motor_check (void)
Description	Check whether Vpk is more than 80% of battery voltage, and phase voltages are close to 0V before motor running
Called Function	R_MTR_get_vpk_AO () bfe_cnt_vol () R_MTR_check_phase_V ()
Argument	None
Return Value	None

Execute reset event for motor

motor_reset ()	
Outline	Execute reset event for motor
Declaration	void motor_reset (void)
Description	Execute reset event for motor when motor error occurs.
Called Function	R_MTR_ExecEvent()
Argument	None
Return Value	None

Control motor running

motor_run ()	
Outline	Control motor running
Declaration	void motor_run(void)
Description	Control motor running, includes motor conducting control with hall sensor, speed adjust according to the throttle input, stop running according to the brake input by POE function.
Called Function	R_MTR_GetStatus() R_MTR_ExecEvent() get_throttle() R_MTR_Limit_abs() R_MTR_SetSpeed()
Argument	None
Return Value	None

16. Appendix D: RX23W System Routines and Application Guide

16.1. API Functions

Table 1 below lists the public API functions that were used in this project and a description of their usage.

Item	Contents
R_RIIC_Open()	The function initializes the RIIC FIT module. This function must be called before calling any other API functions
R_RIIC_SlaveTransfer()	Performs I^2C slave transmission and reception. Changes the transmit and receive patterns according to the parameters
R_RIIC_Close()	This function completes the RIIC communication and releases the RIIC used.
R_GPIO_PinDirectionSet()	This function sets a given GPIO port to be either an input or an output, such as for LED pins
R_GPIO_PinWrite()	This function writes a logical HIGH or LOW signal to a given port
R_IRQ_Open()	This function initializes an interrupt channel, giving a callback function to be processed on an interrupt, such as a switch press
R_BSP_RegisterProtectDisable()	Removes write protection from a given register so that its contents can be altered
R_BSP_RegisterProtectEnable()	Re-enables write protection for a given register
R_CMT_CreatePeriodic()	Creates a periodic timer which will throw an interrupt after the amount of time specified and execute the callback function given
R_CMT_Stop()	Stops the timer specified

app_main()	The main function for the BLE module, which initializes the stack and other application library elements, and then processes BLE events
R_BLE_Open()	Enables the BLE protocol stack, and must be called once before the BLE stack is used
ble_app_init()	Initializes all of the common protocol stack elements, such as the host stack, the GATT database, the GATT server/client, and the System Data Transmission server. Other user-required APIs can be called from this function
R_BLE_CLI_Init()	Initializes the Command Line Interface so that BLE events can be processed and sent to the command line
R_BLE_CLI_RegisterCmds()	This function takes command line parameters and enables them to be called in the command line
R_BLE_CMD_SetResetCb()	Sets the reset callback for the BLE protocol stack
R_BLE_CLI_Process()	Takes input from the console and processes it one character at a time to run user commands
R_BLE_Execute()	Handles all tasks queued in the BLE protocol stack internal task queue and should be called repeatedly in the main loop
R_BLE_Close()	Closes the BLE protocol stack when BLE functionality is no longer required. This function should not be reached during operation
R_BLE_STDS_NotifySd()	Sends the data in the static data buffer via BLE Notify
R_BLE_STDS_NotifyCd()	Sends the data in the continuous data buffer via BLE Notify

Table 13 Public API Functions

Table 2 below lists the user-defined functions at the application level that typically make use of the above functions.

Item	Contents
Auxi_crc_calculate()	This function takes an input buffer and data length and calculates the CRC-16 check bytes for that buffer and returns the result, used to append CRC-16 bytes to a buffer or confirm CRC errors for a given buffer
BLE_crc_calculate()	Same as the function above, but applied to incoming BLE data, which is passed to the I^2C send buffer
board_init()	This function initializes the LED GPIO and switch IRQ, registering a callback function to be processed on an interrupt
buf_init()	This is an optional function for initializing dummy static and continuous data in the I^2C buffers to be sent via Bluetooth that can be used for debugging

i2c_init()	This function sets the I²C parameters to be used such as send/receive buffer name and size, channel number, and callback function, as well as calculating and appending CRC-16 check bytes to the initial send buffer
i2c_open()	This function is called before every new I²C transmission to enable I²C communication and prepare for slave reception/transfer
i2c_run()	This function handles incoming I²C communications by checking to see if/when the transmission was completed or timed out, prints the buffer contents, and then closes, resets, and re-opens I²C communications
i2c_close()	Closes the I²C communications after a transfer has been completed
i2c_reset()	Resets the receive and send buffers to their initial state, performs a CRC-16 check on the receive buffer, and then initiates BLE transmission
ble_notify()	Sets the contents of the receive buffer to the static/continuous buffers and then calls the BLE Notify functions (R_BLE_STDS_NotifySd()/R_BLE_STDS_NotifyCd()) to send the 2 buffers as BLE data packets (static buffer is only sent once)
callbackSlave()	This function is the interrupt callback for I²C, it simply calls the i2c_run() function
timer_cb()	If the switch was pressed instead of using I²C transmissions, a 1-second timer is used to initiate BLE transmission, and this timer callback will call the BLE Notify function and send dummy continuous data
irq_cb()	This callback function responds to a switch press. On first press, it sends static data only, every subsequent press will either initiate or stop the timer that will increment and send dummy continuous data
main()	This is the main user function, it will initialize the board and I²C, and then call app_main(), which runs the BLE module
sdts_cb()	The custom BLE service is called Standard Data Transmission Service, and this function is the callback for the service. This callback function will check for a write request event, which occurs when the phone sends data to the device. The packet received is decoded and placed in the send buffer with CRC-16 check bytes appended to be sent via I²C to the host device. If the power ON button was pressed on the mobile app, the RX23W board's LED will light up, or turn off if the button was pressed to turn it OFF
encode_st_ble_stds_cd_t()	This function will encode the contents of the continuous data buffer 1 byte at a time into a BLE packet to be sent to the phone (this function is automatically generated but the contents are user-defined)
encode_st_ble_stds_sd_t()	This function will encode the contents of the static data buffer 1 byte at a time into a BLE packet to be sent to the phone (this function is automatically generated but the contents are user-defined)

decode_st_ble_stds_fbc_t()	This function will decode the BLE packet received from the mobile app 1 byte at a time so that the device can interpret the received data (this function is automatically generated but the contents are user-defined)

Table 14 User-defined API Functions

For the complete list of I²C and BLE public API functions, refer to the documents in Section 3.

16.2. RX23W (Bluetooth) Code

In this section, an overview of the flow of code execution will be given with reference to the 2 main source files – the app_main.c file and the user application file.

16.2.1. Initialization

- Execution begins in the main() function of the user application file, where board_init() is called to initialize the LED and switch
- The buf_init() function has been commented out as it is used for bypassing I²C transmissions, but this function will initialize the continuous and static data buffers with dummy data and print their contents to be sent via BLE when the switch is pressed
- The i2c_init() function is then called to initialize the parameters for I²C transmission and set up the send/receive buffers and callback function. CRC-16 check bytes are also calculated and appended to the end of the send buffer slaveSend using auxi_crc_calculate()
- The i2c_init() function ends by calling i2c_open(), which enables I²C communication and prepares for a slave transfer/reception. A transmission request from the master will now trigger an interrupt and call the function callbackSlave(), which handles I²C transmissions
- The main() function then calls app_main(), which is the beginning of the BLE source file
- At the beginning of app_main(), the R_BLE_Open() function is called to initialize the BLE module
- The ble_app_init() function is called afterwards, which initializes all of the common protocol stack elements, such as the host stack (R_BLE_ABS_Init()), the GATT database (R_BLE_GATTS_SetDbInst()), the GATT server/client (R_BLE_SERVS_Init()/R_BLE_SERVC_Init()), and the custom service that was created by the user, the System Data Transmission server (R_BLE_STDS_Init())
- After this initialization, the Command Line is initialized with R_BLE_CLI_Init(), a set of commands are registered for the Command Line to recognize with R_BLE_CLI_ RegisterCmds(), and sets the BLE reset callback with R_BLE_CMD_SetResetCb()
- The initialization is now finished, and the program enters an idle state where it is waiting for an interrupt from a switch or I²C, or for a Bluetooth command/event to process using R_BLE_CLI_Process() and R_BLE_Execute()
- Events which are processed and printed on the Command Line include events such as scanning for a device, connection to a device, and disconnecting

16.2.2. Execution of Switch Interrupt (Debugging Only)

- This section assumes that buf_init() was enabled to create dummy data (although it is not required) and that a Smart Phone was paired with the device
- When the RX23W board switch (SW1) is pressed, the code enters and executes irq_cb()
- If this is the first time the switch was pressed, the counter variable will be 0, and the code will call R_BLE_SDTS_NotifySd() to send the contents of the static data buffer via BLE Notify to the Smart Phone and increment the counter
- On the second switch press, the callback function will create a periodic timer which throws an interrupt every second and enters timer_cb()
- When a second has passed and timer_cb() is called, the data in the continuous data buffer is incremented and R_BLE_STDS_NotifyCd() to send the continuous data buffer contents via BLE
- If the switch is pressed again, irq_cb() will use R_CMT_Stop() to stop the timer that causes the continuous data to be incremented and sent to the Smart Phone
- Subsequent switch presses will restart and stop the timer

16.2.3. Execution of I^2C Interrupt

- This section assumes that a Smart Phone was paired with the device and that the Host Device (I^2C Master) is connected to the device
- When an I^2C transmission is sent from the master with the correct slave address, the device recognizes the start of a transmission and throws an interrupt
- The interrupt causes the code to enter callbackSlave(), which simply calls i2c_run() to handle the transaction
- The i2c_run() function handles incoming I^2C communications by checking to see if/when the transmission was completed or timed out
- If the transaction was not completed and there was not a NACK, but the transaction did not time out, the function returns and no action is taken (error state)
- If the transaction was not completed and there was not a NACK, but the transaction timed out, the code prints a timeout message, as well as the contents of the receive buffer, and then closes, resets, and re-opens I^2C communications (timeout error)
- If the transaction is completed successfully, the code prints the contents of the receive buffer, and then closes, resets, and re-opens I^2C communications
- When I^2C communication is closed, the i2c_close() function is called, which waits for a transfer to finish and then stops I^2C communications
- After I^2C communication is closed, the i2c_reset() function is called, which resets the parameters for the receive and send buffers to their initial state to prepare for the next transmission (this does not delete the buffer contents, but they will be overwritten by the next transmission), performs a CRC-16 check on the receive buffer, and then initiates BLE transmission using ble_notify()
- BLE transmission will still occur even if the CRC-16 check fails, but the code prints out a message to indicate success or failure so that the user is aware of a potential error, noise, or lack of CRC-16 check from the Host Device

- When ble_notify() is called, the function will check to see if the device has been written to correctly at least once by checking the contents of the I^2C receive buffer and comparing them to a known, non-default value (the R_RIIC_SlaveTransfer() function is used for both read and write transmissions)
- If a write transmission has occurred, the I^2C receive buffer has its contents split into static and continuous data buffers and sent via BLE Notify. The static data buffer will only be sent once (because these values do not change) using R_BLE_SDTS_NotifySd()
- The continuous data buffer contents are sent every time using R_BLE_SDTS_NotifyCd()
- The Notify functions include an encode function which must be set by the user to tell the device how to encode BLE packets. In this project, all packets are created by encoding the buffer contents 1 byte at a time
- When ble_notify() has completed, I^2C communications are re-enabled using i2c_open() and the device returns to an idle state

16.2.4. Execution of BLE Write Request Interrupt (Packet from Smart Phone)

- This section assumes that a Smart Phone was paired with the device
- When data is received via BLE from the Smart Phone, the code enters the stds_cb() callback function that checks to see if a write request was made (which is always true)
- The data packet is decoded and CRC-16 check bytes are appended to the contents, which are placed in the I^2C send buffer to be read by the I^2C master
- If the 'power ON' byte was set (i.e. the button was pressed in the mobile app), this function will also turn on an LED on the RX23W board to indicate this status. The LED can be turned off and on again by this byte being cleared and set

16.3. System Operation Guide

This section is a step-by-step guide on how to set up and run the project, as well as how to set up the phone app for communication. This section assumes that you have a Target Board for RX23W, Smart Phone, 2 micro-USB cables and have installed e^2 studio.

The phone app described in this example will be the Renesas GATTBrowser app. Refer to the GATTBrowser Application Note in Section 3 for more details.

16.3.1. Setup

For a detailed guide on how to use e^2 studio and install projects onto the Target Board for RX23W, refer to RX Family QE for BLE[RX] R_BLE Script sample and dedicated program Application Notes in Section 3.

- Open e^2 studio, and import and compile the project
- There are 2 DIP switches (ESW1) on the RX23W board which are currently switched OFF. Turn Switch 2 to ON to enable debugging/program loading

- There are 2 micro-USB connectors on the RX23W board which can both be used to power the board
- Connect a micro-USB cable between the Host PC and the ECN1 connector (for connecting to the on-board emulator (E2 J-link Lite)), and select Debug in e^2 studio to load the program onto the board
- The RX23W board can now be disconnected from the Host PC (or used to power the board) and Switch 2 can be turned OFF so that the program runs automatically when powered on
    - Alternatively, the program can be run from e^2 studio in Debug mode
- The program will start automatically, but pressing the RESET button at any time will reset the device and restart the program
- (Optional) Connect a micro-USB cable between the Host PC and the CN5 connector to allow for serial connection and open a serial emulator (terminal) such as Tera Term to view Bluetooth events
    - The terminal must be configured according to Section 2.8.6 of the RX23W Group BLE Module Firmware Integration Technology Application Note
- This concludes the setup. Once the program has been loaded, power the board with a micro-USB, launch a terminal (optional) and open the GATTBrowser app on the Smart Phone to begin

16.3.2. How to Use

- When power is supplied to the RX23W board, it begins Bluetooth advertising automatically
- Select ‘Scan’ from the GATTBrowser app to view all of the Bluetooth devices in range that are advertising
- The RX23W board is advertised as ‘SCOOTER’, with the Bluetooth address ‘74:90:50:FF:FF:FF’
- Select the ‘SCOOTER’ device from GATTBrowser by tapping the ‘play’ button
- Once connected, the services are discovered and UUIDs for the available services can be seen.
- There are 3 custom characteristics which can be selected by their UUIDs at the bottom:
    - The first one (35ada…) contains continuous data (which would be battery, motor speed, etc.) which can be notified to the app from the RX23W board
    - The second (68e…) contains static data (device ID, firmware, etc.) which can be also sent to the app from the RX23W board
    - The third (5f5…) allows you to write data to the board with hex values such as ’00 01 03 05’ and read those values back
- Select the second service, tap the button to turn notifications on, and press the switch (SW1) on the RX23W board to send static data
    - Alternatively, data can be sent via I^{2}C from a Host Device if it is written to the device
- Press the back button on the Smart Phone and select the first service, turn notifications on and press the switch on the RX23W board again. Confirm that it is also sending continuous data that is being incremented every second (only the first 20 bytes can be viewed)
    - Alternatively, data can be sent via I^{2}C from a Host Device if it is written to the device
- Pressing the switch repeatedly will turn the timer off and on again

- Press the back button on the Smart Phone and select the third service. If a value of 80 is written in the first byte of the third service, LED1 on the RX23W board will turn on. Sending any other value will turn the LED off
- To reset the program, disconnect the device from the Smart Phone and press the RESET switch on the RX23W board, or terminate the program in e^2 studio and select Debug again

In e^2 studio, all of the data that is sent and received via I^{2}C/BLE, as well as error and control messages, are printed to the Renesas Virtual Debug Console and can be viewed if the device is being run by the on-board debugger.

16.4. Reference Documents

Document Name	Document No.
Bluetooth® Low Energy Protocol Stack	
Bluetooth Low Energy Protocol Stack Basic Package User's Manual	R01UW0205EJ0101
RX23W Group Bluetooth Low Energy Profile Developer's Guide Application Note	R01AN4553EJ0100
RX23W Group BLE Module Firmware Integration Technology Application Note	R01AN4860EJ0200
RX Family QE for BLE[RX] R_BLE Script sample and dedicated program Application Notes	R01AN4872EJ0100
GATTBrowser for Android Application Note	R01AN3802EJ0101
RX23W	
RX23W Group User's Manual: Hardware	R01UH0823EJ0100
Target Board for RX23W	
Target Board for RX23W User's Manual	R20UT4634EJ0102
I^{2}C Bus Interface	
RX Family I2C Bus Interface (RIIC) Module Using Firmware Integration Technology	R01AN1692EJ0246

Table 15: Useful Resources for Programming the RX23W