Application Report
SLVA100 - May 2001

Advanced Gas Gauge Host Firmware Guide for the TI Battery Monitors ICs

Battery Management

ABSTRACT

TI advanced battery monitoring ICs, such as the bq2018, bq2019 and bq2023, are designed to accurately measure the charge and discharge currents in rechargeable battery packs. Intended for pack integration, these devices contain all the necessary functions to form the basis of a comprehensive battery capacity management system in applications such as cellular phones, PDA's, internet appliances, or other portable products. TI battery monitors work with the host controller in the portable system to implement the battery gas gauging and management system. The host controller is responsible for interpreting the battery monitor data and communicating meaningful battery data to the end-user or power management system.

This document, written around the bq2019, is designed to assist the firmware engineer engaged in development of advanced gas gauging routines in the host controller. A strategy and procedure is presented for implementing the gas gauging function in the host firmware. The various tasks along with suggested constant and variable values are outlined and described with notes pertaining to accuracy enhancement features.

System Components

To measure battery charge and discharge current and report state of charge to an end user, requires several components (see Figure 1). These include:

- A battery monitor such as the bq2018, bq2019, or bq2023: Battery monitor ICs contain a high accuracy coulomb counter, temperature monitor, communication interface and other functions.
- A current sense resistor: The low-value (typically between 10 to 20 mΩ) sense resistor provides a means for the battery monitor to measure the current flowing into or out of the battery. The battery monitor senses the voltage across this resistor.
- A host controller with one free general-purpose I/O for communication: The host controller is required to provide intelligence to the battery monitoring setup. The host controller handles all communication with the battery monitor IC. Since the HDQ and SDQ™ communication protocols are asynchronous one-wire based, a general-purpose I/O is dedicated to handle the communication flow. The host also performs calculations with data read from the battery monitor to determine the battery state of charge.

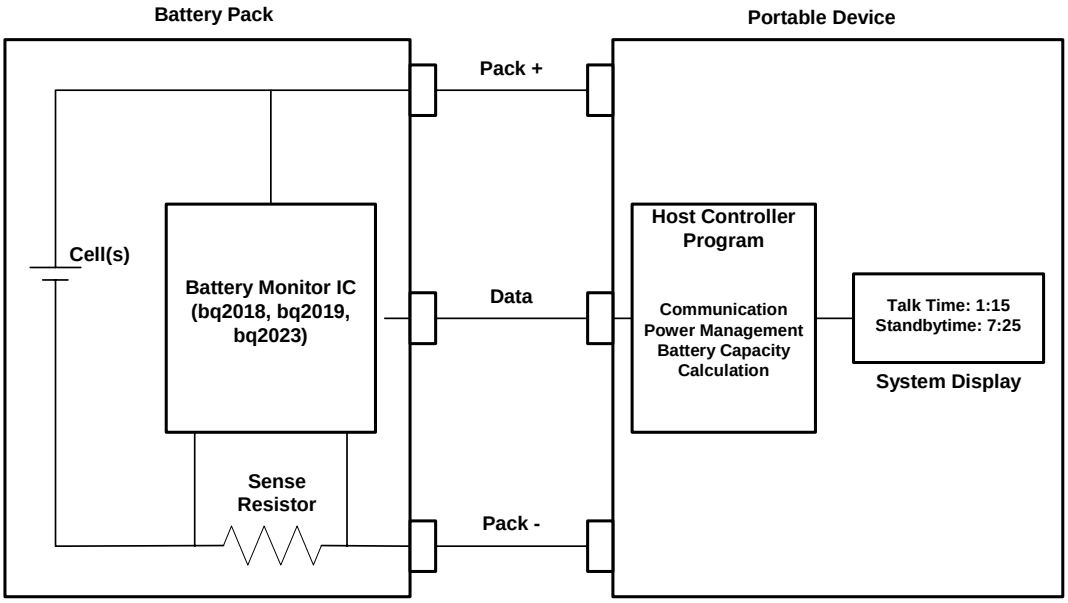

FIGURE 1: System Components

HDQ Communication-related Items

There are several items to consider when developing the lower level firmware for communicating with the bq2019 or other battery monitors with HDQ interface.

Interrupt Strategy

Normal host processing of communication with the bq2019 may encounter a high priority interrupt that needs to be serviced prior to completion of a communication transaction. The time between any bits sent from the host to the bq2019 is not time critical, and as long as the HDQ line is high, the host processor may pause the communication and service an interrupt for any extended period and then resume the communication. However, when reading bits from the bq2019, the host cannot allow any interruption, as the data is transmitted at a fixed rate and the host must know the exact timing to determine whether individual bits are ones or zeroes. Provision must be made for the host to flag any interruption while receiving data. The host should then hold the data line low long enough to cause a break (no need to wait until transmission is complete from the bq2019) and then retry the communication to avoid potential misreads of the data.

Write and Verify

It is good practice to follow all data write operations to the bq2019 with a read operation to verify that correct data was written. Repeat the write operation if the verify operation fails. In most cases, writing to the part is only done during the power shutdown sequence. However, the five register clear bits in the CLR register cannot be verified since they are automatically cleared after being set. These are used at a typical one-hour interval for register maintenance.

Rollover Protection

The six 16-bit registers TEMP, CTC, DTC, SCR, CCR, DCR are divided into high and low bytes. When reading a 16-bit value, there will be a time lapse between the two 8-bit readings. This allows the possibility that the 16-bit value may have a carry from the low byte into the high byte during the time interval between reading the two bytes.

Therefore, a strategy is required to insure there was no carry from low byte to high byte between the readings of the two bytes. One way to eliminate this possibility may include three reads instead of the normal two reads, with the high byte being read on the first and third reads and compared to insure that a carry did not occur. Of course the data could be analyzed to determine whether there was a possibility of a carry and the third read could be invoked only when a possible carry existed.

Break to Reset

If the battery pack is removed or some intermittent connection occurs on the HDQ line, the bq2019 controller may become out of sync with the host communication. A safe practice would be to issue a break prior to each communication to insure that communication was always in sync. An alternative procedure might be to always send a break at the start of a communication block and then not send it again unless there was a need to abort the communication or if the bq2019 did not respond within the expected time.

Considerations for a Gas-Gauging System

Synchronization of reported capacity

The capacity reported by the gas gauge must be synchronized with the full and/or empty condition of the battery. This is normally done by forcing the capacity to zero when the battery drops to some level just above the minimum operating voltage for the host system. Synchronization of the reported capacity can also be done when the battery is full if the host can determine that the charger has completed a normal charge termination and if the firmware has a function that will learn the true capacity of the battery in its typical use environment.

Measuring actual battery capacity

The measurement of the actual capacity is typically a measurement of a full discharge (or charge if the battery is lithium) of the battery without any intervening event that might cause the measured discharge to differ from that of a typical discharge. These events might include cold temperature operation, sitting unused for an extended period during some portion of the learning cycle, removal of the battery during the learning cycle, etc. It may also be prudent to prevent learning a capacity very much different from the previous measured capacity (in one learning cycle). A good measure of the actual capacity in the use environment will allow the full capacity to track the capacity fade of the battery and provide a good reference to synchronize the reported capacity when the battery reaches full.

Offset correction of capacity measurements

The bq2019 contains hardware for canceling the offset of the VFC and allows precision measurement of the small voltages across the sense resistor. Nevertheless, some small offset, perhaps less than 10-15 μV may exist. If a 20 mΩ sense resistor is used, the equivalent 0.5 to 0.75 mA offset can cause some error accumulation if the host remains in a full off condition (or the battery pack is removed) for an extended period of time. If a normal shutdown sequence is performed and later confirmed by the host during a power up sequence, it is possible to determine whether the DCR and CCR counts only contain counts due to offset error. The host can check the charge, discharge, and time registers to determine whether the count rate during the off interval was small enough to only be due to an offset error. If so, the counts should be ignored. If this strategy is implemented, it may be necessary to set a flag in some location in the bq2019 when a normal power down sequence is completed and reset the flag after power up. This would allow the host to distinguish a normal power down sequence from one caused by a battery removal or intermittent connection.

Charge efficiency

The charge efficiency of lithium-ion cells for coulometric charge is virtually 100% and charge efficiency can be ignored. If an algorithm for NiMH or NiCd was needed, the charge efficiency of the cells would have to be characterized and an algorithm developed to apply the efficiency appropriately. The charge efficiency for these cells changes with temperature, rate of charge, and state of charge. The detail procedure in this document assumes that charge efficiency is 100%.

Use basic bq2019 count units instead of engineering units

There is no need to accumulate the charge added or removed from the battery or to measure the full charge capacity of the battery in mAh. It will be much more convenient to make all computations in terms of the 3.05 μVh per count units reported by the CCR and DCR registers. Most computations like relative state of charge (percent of full charge) and run time (delta charge removed/charge remaining) are ratios and any unit conversions of both numerator and denominator will cancel. Any computations like standby time can be made using estimated standby current that has been converted to equivalent VFC counts to eliminate unit conversion computations by the host during normal operation.

VFC gain or sense resistor variations

There is no need to perform calibration of the VFC gain and sense resistor unless there is a requirement to provide information in engineering units to the host or user. Talk time is a ratio,

and any calibration error will cancel. If measured capacity is off by some percentage, say 20% high, the remaining capacity of the battery will be 20% high and the use rate will also be 20% high. When these are divided, the gain errors in both numerator and denominator will cancel. If an accurate value for average current drain in mA is required, then a calibration will have to be performed. The calibration factor can be stored in the bq2019 and retrieved by the host during the power up process. The host can then use this factor whenever a current computation must be reported.

Gas Gauging Firmware Example

Apart from the required general HDQ read and write functions, the gas gauge firmware may be broken down into six tasks as in table 1.

Table 1. Gas Gauge Firmware Tasks

Name	Recommended Interval	Description
GGInitialize( )	On Power Up	Qualify the battery and communication. Read and convert constants and scratchpad variables from bq2019 into the host.
GGUpdate( )	Once Per Minute (typical)	Calculate capacity, average current, run time and charge time. Manage full, empty and learning. Update display.
GGRegisterMaint( )	Hourly	Do register maintenance. Store remaining capacity in the bq2019. Clear registers.
GGSelfDischarge( )	Several Times Per Day	Make corrections to remaining capacity for self-discharge.
GGMeasBattVltg( )	Every 20 Seconds	Measure the battery voltage. Call GGUpdate( ) if battery crosses the empty voltage threshold.
GGPwrDwnSave( )	On Power Down	Provide an orderly shutdown. Update bq2019 registers.

GGInitialize ( )

On power up, the host should initialize its battery monitoring system. This usually involves testing the communication link, and reading some identification information stored in the bq2019 flash memory, such as the IDROM, that would insure the compatibility of the battery for the system.

Once the battery and battery communication has been qualified, the initialization routine should retrieve all appropriate constants from the bq2019 and store them into local variables (Table 2).

If your host is extremely limited in dynamic memory space, you may prefer to read these constants as they are needed, but the time penalty could be significant. You will want to retrieve a couple of constants from your own system, iTalkLd and iStbyLd, which specify your estimated talk load (for cell phone applications) and standby load in milliamps. These should probably not be kept in the bq2019, since they are not properties of the battery, but rather of the system.

Table 2. Typical Gas Gauge Constants

Class	Name	Suggested bq2019 Address [1]	Used In	Type/Units	Description/Comment
Mfg Data	ID_ROM	78~7F	GGInitialize( )	8 bytes	Factory programmed ROM
	sMFG_DATA	20~25	GGInitialize( )	(str) ASCII	Manufacturers Data
	sMODEL	26~2B	GGInitialize( )	(str) ASCII	Battery Model
	sMFG_NAME	2C~35	GGInitialize( )	(str) ASCII	Manufacturer Name
	iSERIAL_NO	36/37	GGInitialize( )	(Uint)	Serial Number
	iMFG_DATE	38/39	GGInitialize( )	(Uint) Date	Manufactured Date
Design Data	iDES_CAP	3A/3B	GGInitialize( ) GGUpdate( )	(Uint) Milliampere Hours	Pack Design Capacity
	iSNS_RES	3C/3D	GGInitialize( )	(Uint) μΩ * 327.68	Sense Resistor μΩ x 2^{15} / 100
	iSLF_DSG_RATE	3E/3F	GGInitialize( ) GGSelfDischarge	(Uint) %/Day * 105.8	Self Discharge Rate
	iEND_DSG_VLTG	40/41	GGInitialize( ) GGUpdate( ) GGMeasBattVltg( )	(Uint) Millivolts	End of Discharge Voltage
	iTERM_CURR	42/43	GGInitialize( ) GGUpdate( )	(Uint) Milliamperes	Charge Taper Termination Current
	iCAP_COMP_TE	44	GGInitialize( ) GGUpdate( )	(Byte) % of full capacity	Capacity compensation for temperature
	iCAP_COMP_LD	45	GGInitialize( ) GGUpdate( )	(Byte) % of full capacity	Capacity compensation for load current
	iTALK_LD	n/a	GGInitialize( )	(Uint) mA	Estimated Talk Load
	iSTBY_LD	n/a	GGInitialize( )	(Uint) mA	Estimated Standby Load
	iTALK_LD_CNTS	n/a	GGInitialize( ) GGUpdate( )	Counts @ 3.05 uVh	= iSNS_RES * iTALK_LD / 1000
	iSTBY_LD_CNTS	n/a	GGInitialize( ) GGUpdate( )	Counts @ 3.05 uVh	= iSNS_RES * iSTBY_LD / 1000
	iCYCL_CAP	n/a	GGInitialize( ) GGUpdate( )	(Uint) = iDES_CAP * iSNS_RES* 0.8/1000	May be used to simplify math in GGUpdate( )

[1] Addresses are in hex. n/a indicates that the values are maintained by the host since they are not a function of the battery. ID_ROM is a fixed address in the bq2019.

There are a number of variables to read from the bq2019 memory. These are listed in Table 3, where the **Used In** column includes the GGInitialize( ) function. Additionally, it is recommended that some additional local calculated constants iTALK_LD_CNTS, iSTBY_LD_CNTS, iCYCL_CAP (table 2) and several additional local variables for flags and calculations (table 3).

Table 3. Typical Gas Gauge Variables

Class	Name	Suggested bq2019 Address [2]	Used In	Type/Units	Description/Comment
Computed Values	iLastMeasDsg	00/01	GGInitialize( ) GGUpdate( ) GGSelfDischarge( )	(Uint) Counts @ 3.05 uVh	Last measured discharge. (Initial value set during mfg.)
	iRemCap	02/03	GGInitialize( ) GGUpdate( )	(Uint) Counts @ 3.05 uVh	Remaining capacity (Initial value set during mfg.)
	iCycleCnt	04/05	GGUpdate( )	(Uint) Units	Cycle Count (Typically increased if iDsgCntrCuml has reached 80% of design capacity)
	iMaxTemp	06	GGInitialize( ) GGUpdate( ) GGPwrDwnSave( )	(Byte) °K	Max temp seen by this bq2019. Update it in the host during GG_Update.
	bValidDsg	07	GGInitialize( ) GGUpdate( ) GGPwrDwnSave( )	(bool) Flag	Valid discharge flag
	iDsgCntr	08/09	GGInitialize( ) GGUpdate( ) GGSelfDischarge( ) GGPwrDwnSave( )	(Uint) Counts @ 3.05 uVh	Discharge counter for learning a new iLastMeasDsg
	iDsgCntrCuml	0A/0B	GGInitialize( ) GGUpdate( ) GGSelfDischarge( ) GGPwrDwnSave( )	(Uint) Counts @ 3.05 uVh	Cummulative discharge counter tracks partial discharges for iCycleCnt update.
	iLastRemCap	0C/0D	GGInitialize( ) GGPwrDwnSave( )	(Uint) Counts @ 3.05 uVh	Last computed remaining capacity value
	iCumlCorrectn	0E/0F	GGInitialize( ) GGUpdate( ) GGSelfDischarge( ) GGPwrDwnSave( )	(Uint) Counts @ 3.05 uVh	Tracks cumulative self-discharge corrections. Disqualifies learning cycle if it exceeds 10% of iDES_CAP.
Computed Values	iTimeSinceMaint	n/a	GGInitialize( ) GGUpdate( ) GGRegisterMaint( )	(Uint) Minutes	Minutes elapsed since last register maintenance.
	iRunTime	n/a	GGUpdate( )	(Uint) Minutes	Est. remaining run time at present current
	iTimeToFull	n/a	GGUpdate( )	(Uint) Minutes	Est. remaining time to charge to full

Class	Name	Suggested bq2019 Address [2]	Used In	Type/Units	Description/Comment
	iTalkTime	n/a	GGUpdate( )	(Uint) Minutes	Est. remaining run time at iTALK_LD
	iStbyTime	n/a	GGUpdate( )	(Uint) Minutes	Est. remaining run time at iSTBY_LD
	iRelChgPercent	n/a	GGUpdate( )	(Uint) %	Relative charge in percent of full charge. aka 'RSOC'
	bInit	n/a	GGInitialize( ) GGUpdate( )	(bool) Flag	Initial pass flag. Inhibits iRunTime and iTimeToFull calculations the first time through GGUpdate( )
	bEDV	n/a	GGInitialize( ) GGUpdate( )	(bool) Flag	Flag to notify other host process that battery is at end of discharge voltage.
	bChgFull	n/a	GGUpdate( )	(bool) Flag	Flag used by the charger or other host process to indicate that the battery is fully charged.
	iRemCapNow	n/a	GGInitialize( ) GGUpdate( ) GGRegisterMaint( ) GGSelfDischarge( ) GGPwrDwnSave( )	(int) Counts @ 3.05 uVh	Most recent calculation of remaining capacity.
	iRemCapPrev	n/a	GGUpdate( )	(int) Counts @ 3.05 uVh	Calculation of remaining capacity from the previous update.
	iRemCapTemp	n/a	GGSelfDischarge( )	(Uint) Counts @ 3.05 uVh	Temp variable for iterative self-discharge estimate.
	iSlfDsgEst	n/a	GGSelfDischarge( )	(Uint)	Temp self-discharge variable.
	iTempCorrection	n/a	GGSelfDischarge( )	(Uint)	Temp self-discharge variable.
	fElapsedTime	n/a	GGUpdate( )	(float) Minutes	Value in Minutes, with resolution to seconds since power on reset detected.
	fElapsedTimePrev	n/a	GGUpdate( )	(float) Minutes	Value in Minutes, with resolution to seconds since the power on reset detected previous to fElapsedTime.
bq2019 Gas Gauge Registers and cor-responding host vars.	iKelvin (Host) TMPL (bq2019) TMPH (bq2019)	 60 61	GGInitialize( ) GGUpdate	(Uint) °K	TMPH, TMPL in the bq2019 contain die temperature in °K.
	iRegClr (Host) CLR (bq2019)	 63	GGUpdate( ) GGRegisterMaint( ) GGSelfDischarge( )	(Uint) Bits	CLR is used to quickly clear DCR, CCR, SCR, DTC, CTC in any combination. iRegClr is used by the host to setup the next desired clearing pattern.

Class	Name	Suggested bq2019 Address [2]	Used In	Type/Units	Description/Comment
	iChgTime (Host) CTCL (bq2019) CTCH (bq2019)	 65 66	GGInitialize( ) GGUpdate( ) ??	(Uint) 4096 counts per hour	Charge Time Counter. May be used in GGUpdate( ) if no timer or time function is available in the host.
	iDsgTime (Host) DTCL (bq2019) DTCH (bq2019)	 67 68	GGInitialize( ) GGUpdate( ) ??	(Uint) 4096 counts per hour	Discharge Time Counter. May be used in GGUpdate( ) if no timer or time function is available in the host
	iSlfDsgCntr (Host) SCRL (bq2019) SCRH (bq2019)	 69 6A	GGInitialize( ) GGSelfDischarge( )	(Uint) 1 count per hour (20 – 30°C)	Self Discharge Counter. Rate varies automatically with temperature.
	iChgCntr (Host) CCRL (bq2019) CCRH (bq2019)	 6B 6C	GGUpdate( )	(Uint) Counts @ 3.05 uVh	Charge count register in bq2019 increments when voltage at the SR pin is positive.
	iDsgCntr (Host) DCRL (bq2019) DCRH (bq2019)	 6D 6E	GGUpdate( )	(Uint) Counts @ 3.05 uVh	Discharge count register in bq2019 increments when voltage at the SR pin is negative.

[2] Addresses are in hex.

GGInitialize () Steps

The steps for GGInitialize () are as follows:

1. Read some or all of the manufacturing data from the bq2019. Return if the communication is faulty or the battery is incorrect.
2. Read each of the design data class constants in table 2. Calculate the lower three constants from the formulas in the table.
3. Read iMaxTemp, iLastMeasDsg, iLastRemCap, bValidDsg, iRemCap, iDsgCntrCuml, iChgTime(CTC), and iDsgTime(DTC) from the bq2019.
4. Calculate iTimeSinceMaint as (iChgTime + iDsgTime) * 60 / 4096.
5. Set bInit true and bEDV false.
6. Set iRemCapNow to iLastRemCap.
7. If bValidDsg is true, first write false to that memory location in the bq2019, then read iDsgCntr and iCumlCorrectn from the bq2019.
8. Jump to GGUpdate()

GGUpdate ( )

GGUpdate() should be called at the desired display update interval, or whenever GGMeasVltg() detects that the voltage has fallen below the iEND_DSG_VLTG. Typically the update is performed at a 1-minute interval, but you may want to update more frequently as the voltage reaches iEND_DSG_VLTG.

This function is where all the work gets done. First you read the bq2019 registers, get the time, then determine if the battery is charging or discharging. If we are discharging: update the auxiliary discharge counters, update the cycle count, make capacity corrections for temperature & load and calculate run-time. Otherwise, if we are charging, then calculate time-to-full.

Next, a check is done to see if an hour has passed since the last register maintenance. If so, GGRegUpdate () is called to record cumulative corrections, clear the registers and, if desired, call GGSelfDischarge() several times each day.

Finally, a series of tasks handles capacity learning, end of discharge voltage, synchronization at full and empty, and calculation of talk time, standby time and remaining percent of full. The important thing to know here is that we clear the bq2019 registers when the voltage reaches end-of-discharge, and when the battery reaches *full* condition we update the remaining capacity & last-measured-discharge then clear the bq2019 registers.

GGUpdate() Steps

The steps for GGUpdate() are as follows:

1. Read the coulomb counters (CCR, DCR) into iChgCntr and iDsgCntr.
2. Read the bq2019 temperature (TMP) into iKelvin. If iKelvin > iMaxTemp then iMaxTemp = iKelvin
3. Set the previous elapsed time (fElapsedTimePrev) to current elapsed time (fElapsedTime)
4. Read the current elapsed time (fElapsedTime) from your system, or the bq2019 time registers.
5. Set the previous remaining capacity (iRemCapPrev) to current remaining capacity (iRemCapNow).
6. Calculate the remaining capacity as: iRemCapNow = iRemCap – iDsgCntr + iChgCntr.
7. Initialize iRunTime and iTimeToFull to –1. The function will return them this way the first time through to signify that the result is not applicable.
8. if iRemCapNow < iRemCapPrev then **we are discharging…**

    a. Increase the discharge counter: iDsgCntr = iDsgCntr + iRemCapPrev - iRemCapNow.

b. Increase the cumulative discharge counter: iDsgCntrCuml = iDsgCntrCuml + iRemCapPrev – iRemCapNow.

c. Manage the discharge cycle count by testing to see if iDsgCntrCuml is still less than iCYCL_CAP. If not, read iCycleCnt from the bq2019, increase it by 1 and write it back to the part. Then decrease iDsgCntrCuml by iCYCL_CAP.

d. If this is not the first time through this function (bInit is false) then: Calculate iTimeSinceMaint = iTimeSinceMaint + fElapsedTime – fElapsedTimePrev. Also, with bInit = false and iRemCapNow = 0 then iRunTime = 0. With bInit = false and iRemCapNow > 0, run time minutes are calculated as: iRunTime = [fElapsedTime – fElapsedTimePrev] * iRemCapNow / [iRemCapPrev – iRemCapNow].

e. Apply any desired corrections to iRemCapNow for temperature and load. You can use constants in the bq2019 such as iCAP_COMP_TE and iCAP_COMP_LD to model the chemistry in your battery. The current temperature is already in iKelvin. There should be no need to calculate the load current from dq/dt since your firmware knows if the system is operational or in standby and can use iTALK_LD or iSTBY_LD. This correction addresses the case where a cell phone that was charged in the ski lodge is placed in the user's jacket pocket. After half an hour at –10°C, the skier turns the phone on and looks at his predicted talk time, which will appear inflated without this correction. You will probably want to set a flag the first time this correction is made, and reset the flag when the temperature and/or load changes significantly. A future application note will address these techniques in detail. Move on to # 10 below.

9. If iRemCapNow >= iRemCapPrev and this is not the first time through this function (bInit is false) then: Calculate iTimeSinceMaint = iTimeSinceMaint + fElapsedTime – fElapsedTimePrev and check to see if iRemCapNow > iRemCapPrev + 1. If so, **we are charging…**
    a. Test if iRemCapNow < iLastMeasDsg. If false then the battery must be full and iTimeToFull = 0.

    b. If the above test is true then we calculate time-to-full minutes as: iTimeToFull = [fElapsedTime – fElapsedTimePrev] * [iLastMeasDsg – iRemCapNow] / [iRemCapNow – iRemCapPrev]. Also set bValidDsg false to indicate that it is not a valid discharge cycle. Note that while this formula is a good approximation for time-to-full, it can be improved upon by taking into account the exponential shape of the charge current after the charger transitions from constant-current to constant-voltage mode. A future application note will address this issue in detail.

10. Test if iTimeSinceMaint < 60. If false, then call GGRegMaint () since it must have been over an hour since the last register maintenance. Note that before it returns, GGRegMaint () may call GGSelfDischarge which would add a cumulative self-discharge value in iCumlCorrectn.
11. If bValidDsg = true then test If iKelvin < 283 (10°C) . If so, set bValidDsg = false since the part is too cold for a valid learning cycle.
12. If bValidDsg = true then test if iCumlCorrectn > 0.1 * iDES_CAP. If so, set bValidDsg = false since we have too much self-discharge to allow a valid learning cycle.

13. Look for a battery full condition by first testing if iRemCapNow >= iLastMeasDsg. If true then the battery is full. If false, the bChgFull flag (set by your charger?) should be checked since charging the battery to iLastMeasDsg might not be possible if the cells have had a substantial capacity fade since the last learning cycle. If the battery is full, the basic procedure is to set iRegClr to 0x1F, iTimeSinceMaint = 0, iRemCapNow = iLastMeasDsg, then test bValidDsg.

    a. If bValidDsg = True write iRegClr to the CLR register then set iRegClr = 0. This allows for learning to continue through partial charge/discharge cycles.

    b. If bValidDsg = False: Set bValidDsg = true, iDsgCntr = 0, iDsgCntrCuml = 0 and iRemCap = iRemCapNow. Then write iRemCap and iRegClr to the bq2019. Finally, set iRegClr = 0.

14. If the battery is not full there are three possibilities to consider: It could be empty due to low voltage, empty due to depleted capacity, or in a normal condition. The first test is to compare the battery voltage with iEND_DSG_VLTG. If the voltage is not above this threshold, then perform the steps in paragraph *a* below. If the voltage is at or above iEND_DSG_VLTG then test to see if iRemCapNow has dropped to 0 or negative. If so, perform the steps in paragraph *b* below. If iRemCapNow is above 0, the battery is normal – perform the steps in paragraph *c* below.

    a. Low Voltage. First check bEDV. If it is true, then we have already handled the problem in a previous update – so were done. Otherwise we have a low voltage synchronization and learning cycle to handle. First, set iRemCapNow = 0, iTimeSinceMaint = 0, bEDV = 1 and iRegClr = 0x1F. Then test bValidDsg. If bValidDsg is false then we do not have a valid learning cycle – set iRemCap = 0, write iRemCap and iRegClr to the bq2019, set iRegClr = 0 and move on to step 15.

       If the bValidDsg flag is true, however, we must perform the learning cycle. First add in the latest capacity reduction: iDsgCntr = iDsgCntr + iRemCapPrev – iRemCapNow. Then test iDsgCntr against iDES_CAP to insure that the proposed new value is within +/-10% of the design capacity. If we are not outside the 10% limit, set iLastMeasDsg to iDsgCntr, otherwise set iLastMeasDsg up or down 10% of the design value. Save the new iLastMeasDsg to the bq2019. Set iRemCap = 0, write iRemCap and iRegClr to the bq2019, set iRegClr=0 and move on to step 15.

    b. Depleted Capacity. First set iRemCapNow = 0 to handle the negative case. Set iRegClr = 0x1F and iRemCap = 0. Then write iRemCap and iRegClr to the bq2019. Finally, set iRegClr = 0. Move on to step 15.

    c. Battery Normal. Test if iRegClr = 0. If so, move on to step 15. However, if there is a value, it is because GGRegMaint( ) was called in step 10 above. Set iRemCap = iRemCapNow, write iRemCap and iRegClr to the bq2019. Finally, set iRegClr = 0.

15. Calculate talk minutes as iTalkTime = iRemCapNow * 60 / iTALK_LOAD, standby minutes as iStbyTime = iRemCapNow * 60 / iSTBY_LOAD, percent full as iRelChgPercent = iRemCapNow / iLastMeasDsg. Lastly, bInit is set false.

16. You now have several useful values – iRunTime or iTimeToFull, iTalkTime, iStbyTime, iRelChgPercent and iRemCap that can be returned or used within this function to update your display.

GGRegisterMaint ( )

This simple function is called in GGUpdate( ) if it is determined that an hour has passed since the previous register maintenance. Its job is to clear the variable that counts to an hour, request a clearing of the bq2019 registers along with an update of remaining capacity in the part. Also if calls GGSelfDischarge if self-discharge counts have reached a significant value.

Set iTimeSinceMaint = 0, and iRegClr = 0X1B. The 1B in iRegClr sets up a clearing pattern for all clearable registers in the bq2019 except for SCR. Test if iRemCapNow > 0. If false, then the function returns. Otherwise, read SCR into iSlfDsgCntr and test if iSlfDsgCntr < iSLF_DSG_RATE. If true, then the function returns. If false, call GGSelfDischarge( ) and return.

GGSelfDischarge ( )

The idea here is to read the self-discharge register and the self discharge rate from the bq2019, then perform self-discharge correction, save the updated remaining capacity, and clear bq2019 registers.

This function is called in GGRegisterMaint( ) if it is determined that self discharge counts are significant enough to be added into the discharge counters. We use three temporary variables here – iRemCapTemp, iTempCorrection and iSlfDsgEst for the iterative self-discharge estimate.

First set the temporary variable iRemCapTemp to the smaller of iRemCapNow or iLastMeasDsg. Set iTempCorrection = 0 and perform an iterative loop as shown below. Note that iSlfDsgCntr was retrieved from the bq2019 in GGRegisterMaint( ) and was either passed in or is a valid global.

Do while iSlfDsgCntr >= iSLF_DSG_RATE {

iSlfDsgEst = iRemCapTemp / 512

iTempCorrection = iTempCorrection + iSlfDsgEst

iRemCapTemp = iRemCapTemp – iSlfDsgEst

iSlfDsgCntr = iSlfDsgCntr – iSLF_DSG_RATE}

Then make the following assignments:

- iSlfDsgEst = iRemCapTemp * [iSlfDsgCntr / iSLF_DSG_RATE] / 512
- iRemCapTemp = iRemCapTemp – iSlfDsgEst
- iTempCorrection = iTempCorrection + iSlfDsgEst
- iRemCapNow = iRemCapNow – iTempCorrection
- iDsgCntr = iDsgCntr + iTempCorrection

- iDsgCntrCuml = iDsgCntrCuml + iTempCorrection
- iCumlCorrectn = iCumlCorrectn + iTempCorrection
- iRegClr = iRegClr or 0x04 //SCR Clear bit

Return ()

GGMeasBattVltg ( )

Use the system A/D converter to read the battery voltage at least every 20 seconds. Either make the value available globally, or allow GGUpdate( ) to query the voltage. Call GG_Update immediately if the voltage < = iEND_DSG_VLTG.

GGPwrDwnSave ( )

Call this routine for power down save of variables into the bq2019. It saves the last remaining capacity calculation, the cumulative discharge toward cycle count, and the maximum temperature. If the present discharge is valid, then it saves the valid discharge flag along with the discharge count toward last-measured-discharge learning and the cumulative correction toward valid-discharge disqualification.

Set iLastRemCap = iRemCapNow. Write iLastRemCap, iDsgCntrCuml, iMaxTemp to the bq2019. If bValidDsg = true then write bValidDsg, iDsgCntr and iCumCorrectn to the bq2019.

References

1. *Texas Instruments, Inc. Data sheet for bq2019 Advanced Battery Monitor IC (SLUS456A)*

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Copyright © 2002, Texas Instruments Incorporated