Application Report

SLUA801–January 2017

Battery Management Solutions
Battery Gauges

Gauge Communication

Dominik Hartl

ABSTRACT

Battery gauges have a rich set of parameters which enable compatibility with a wide range of battery types and applications. Programming these parameters requires access to data memory, which together with examples of how to read gauging results, is explained in this application note.

Contents

List of Tables

1 Gauge Commands

The host controller communicates with the gauge through *gauge commands*.

A gauge command is the equivalent to a register. For example, reading the state of charge is accomplished through the *StateOfCharge()* gauge command, which has the command codes 0x1C and 0x1D. So if the interface is I^2C, reading from device register 0x1C and 0x1D returns the current state of charge of the battery as calculated by the gauge.

The TRM for the gauge lists the available gauge commands. The commands are organized in two groups:

- Standard commands (which allow access to common gauging information).
  - Standard command *Control()* is used to execute various functions (sub commands).
- Extended commands (which mainly support access to the of the gauge's proprietary configuration parameters in data memory).

2 Gauge Configuration

Battery gauges have a rich set of parameters which enable compatibility with a wide range of battery types and applications. Some gauges require very little configuration, while others require a significant number of parameters to adjust performance for a specific battery and system.

All gauges store configuration in data memory using an indirect access method. Data memory is organized in subclasses and subclasses are organized in data blocks, which are a sequence of bytes. Each data block contains various parameters at specific offsets, lengths and data types.

Configuring the gauge means setting specific parameters, which is accomplished by writing the applicable bytes within a data block of a subclass within the data memory of the gauge.

TI provides a software tool, bqStudio, which allows easy access to all parameters through a GUI. After creating and validating the configuration with bqStudio, the configuration can be exported in a FlashStream® file. See Section 5 for information about the format of this file.

3 Accessing the Gauge From a Host Controller

TI provides system independent ANSI-C source code which is intended as a basis to develop a device driver for a high-level operating system or directly in a system without an operating system.

The API consists of functions which can configure the gauge and read gauging results.

3.1 Abstract End-System Dependent APIs

Because the code is system independent, the customer must implement three functions which abstract the low-level communications interface for the host controller, as follows:

```
int gauge_read(void *pHandle, unsigned char nRegister, unsigned char *pData, unsigned char
nLength);
int gauge_write(void *pHandle, unsigned char nRegister, unsigned char *pData, unsigned char
nLength);


pHandle is a pointer to a communications adapter (e.g. I2C driver)
nRegister is the target register (command code)
pData is a pointer to data that will be written starting at nRegister
nLength is the total number of bytes for pData
Function returns number of read/written bytes or 0 (error)


void gauge_address(void *pHandle, unsigned char nAddress);
pHandle is a pointer to a communications adapter (e.g. I2C driver)
nAddress is the I2C device address
```

FlashStream is a registered trademark of BP Microsystems LP.
Linux is a registered trademark of Linus Torvald.
All other trademarks are the property of their respective owners.

These three functions must implement the following functionality:

```
gauge_read: Read [nLength] bytes from the gauge starting at [nRegister].
gauge_write: Write [nLength] bytes to the gauge starting at [nRegister].
gauge_address: Set the I2C device address [nAddress].
```

These three function prototypes are not specific to the underlying interface (I^{2}C, SMBUS, or HDQ). However, the implementation is specific to the interface. See Section A.3 for an example using the Linux® user space I^{2}C interface.

[nRegister] in the nomenclature of a TI gauge is the equivalent to a command.

[pHandle] is a (void) pointer to a data structure which identifies the communications interface. This pointer can be NULL if there is a non-ambiguous single communications interface or it can point to data which identifies the communications interface. The example code does not access this pointer – it is intended as a handle only.

3.2 Configuration

Configuration of the gauge is supported by the following APIs.

3.2.1 FlashStream® Parser

The FlashStream Parser is the easiest way to configure the gauge by using the Golden Image FlashStream file from bqStudio.

After calling this function, the gauge is completely configured. This is a convenient way to reinitialize ROM gauges from a host uC after a power cycle or reset. This API can also program data memory for flash gauges (which keeps the configuration in persistent memory, so it is not necessary to repeat writing the configuration after a power cycle or reset).

```
// gauge_execute_fs: Execute a Flash Stream File
// pHandle = handle to communications adapter
// pFS = pointer to NULL terminated 8-bit string with complete Flash Stream File content
// return value = NULL if successful or pointer to text which failed (e.g. compare or syntax
error)
char *gauge_execute_fs(void *pHandle, char *pFS);
```

If the host uC supports a file system, the FlashStream file from bqStudio can be stored on this file system, and the host uC can open the content and provide it to this function as a zero-terminated string of characters.

If the host uC does not support a file system, the content of the FlashStream file can be copied into a constant zero-terminated C-string (for example, in a header file which is then compiled into the firmware image).

Example (Linux):

```
struct stat st;
char *pFileBuffer;
int nFSFile;
void *pI2C;

pI2C = … //assign communications interface handle

stat(FLASH_STREAM_FILE, &st);
if ((nFSFile = open(FLASH_STREAM_FILE, O_RDONLY)) <0) exit(1);

pFileBuffer = malloc(st.st_size);
if (!pFileBuffer) exit(1);
read(nFSFile, pFileBuffer, st.st_size);
close(nFSFile);

pDataClasses = gauge_execute_fs(pI2C, pFileBuffer, st.st_size);
free(pFileBuffer);
```

3.3 Gauge Standard Commands

The following APIs issue standard commands to the gauge.

Standard commands are used to obtain gauging results and to change select configuration parameters.

```
// gauge_cmd_read: Read the results from a gauge standard command
// pHandle = handle to communications adapter
// nCmd = gauge standard command (e.g. CMD_STATE_OF_CHARGE)
// pData = holds returned data from the gauge
// return value = 0 if successful or error code
int gauge_cmd_read(void *pHandle, unsigned char nCmd, unsigned int *pData)
// gauge_cmd_write: Write data for a gauge standard command
// pHandle = handle to communications adapter
// nCmd = gauge standard command (e.g. CMD_AT_RATE)
// nData = data to be written to the gauge standard command
// return value = 0 if successful or error code
int gauge_cmd_write(void *pHandle, unsigned char nCmd, unsigned int nData)
```

Examples:

```
int nVoltage = gauge_cmd_read(pI2C, CMD_VOLTAGE); //voltage in [mV]
int nSOC = gauge_cmd_read(pI2C, CMD_STATE_OF_CHARGE); //SOC in [%]
gauge_cmd_write(pI2C, CMD_AT_RATE, nAtRate); // set load value
```

3.4 Gauge Sub Commands

Sub commands are commands that the gauge executes as part of the control command.

The sub commands are used to read secondary information from the gauge, for example firmware version, ChemID (read only) or to trigger functions, for example gauge reset, clearing interrupts or enabling modes.

```
// gauge_control: Issue a control sub-command
// pHandle = handle to communications adapter
// nSubCmd = gauge sub-command (e.g. BAT_INSERT)
// return value = result from sub-command
unsigned int gauge_control(void *pHandle, unsigned int nSubCmd)
```

Examples:

```
gauge_control(pI2C, SUBCMD_ RESET); // reset the gauge
int nFWVersion = gauge_control(pI2C, SUBCMD_FW_VERSION); // read FW version
int nChemID = gauge_control(pI2C, SUBCMD_CHEMID); // read ChemID
```

3.5 Gauge Data Memory Access

Configuration data is stored in data memory (either flash or RAM). Data memory access is fairly complex because the gauge uses indirect addressing with check-sums to ensure data integrity. These APIs take care of reading and writing data classes within the data memory. See Section 4 for further information about data memory and data classes.

```
// gauge_read_data_class: Read the results from a gauge standard command
// pHandle = handle to communications adapter
// nDataClass = Subclass
// pData = holds returned data from the gauge
// return value = number of bytes read from the gauge
int gauge_read_data_class(void *pHandle, unsigned char nDataClass, unsigned char *pData, unsigned
char nLength);
// gauge_write_data_class: Write data for a gauge standard command
// pHandle = handle to communications adapter.
// nDataClass = Subclass
// nData = data to be written to the gauge
// return value = number of bytes written to the gauge
int gauge_write_data_class(void *pHandle, unsigned char nDataClass, unsigned char *pData,
unsigned char nLength);
```

Example: change design capacity and design energy for the bq27421.

To change parameters, first identify the data class (also called subclass) from the TRM. In this example, the two parameters reside in the data class *State* at offset 10 (design capacity) and offset 12 (design energy). Read the whole data class into a byte buffer with a size equal to integer multiples of 32. The size must be greater or equal of the largest offset (plus the data type length) within the data class. This information is in the TRM. In this example, the largest offset in data class *State* is 39 with a data type length of two bytes (data type = I2) so the whole data class size is 2 × 32 = 64. Change the parameter in the byte buffer followed by writing the whole data class:

```
char pData[DC_STATE_LENGTH]; //DC_STATE_LENGTH = 64
gauge_read_data_class(pI2C, DC_STATE, pData, DC_STATE_LENGTH);

pData[10] = (DESIGN_CAPACITY & 0xFF00) >>8;
pData[11] = DESIGN_CAPACITY & 0xFF;
pData[12] = (DESIGN_ENERGY & 0xFF00) >>8;
pData[13] = DESIGN_ENERGY & 0xFF;

gauge_cfg_update(pI2C); // required for ROM gauge
gauge_write_data_class(pI2C, DC_STATE, pData, DC_STATE_LENGTH);
gauge_exit(pI2C, SOFT_RESET); // required for ROM gauge
```

4 Data Memory

Data memory stores the configuration of the gauge. There are two basic types of gauges: flash and ROM (some ROM gauges have one time programmable (OTP) memory).

Writing data memory of flash gauges stores the configuration in persistent memory so even if the gauge is reset or power cycled, it keeps the configuration. Flash gauge data memory can be written without changing the operating mode of the gauge.

ROM gauges contain default configuration (including ChemID) in read only memory. During gauge boot up, the default configuration is copied automatically from ROM to data memory RAM. Updating gauge configuration for a ROM gauge means changing the content of the data memory in RAM. Because this is volatile memory, its contents are lost after a power cycle or reset. ROM gauge data memory in RAM require a change of operating mode of the gauge (configuration update mode).

Some ROM gauges have OTP memory which the user can program during production. The gauge copies the contents of the OTP memory to data memory RAM during gauge boot.

TI provides production tools (bqStudio and SmartFlash) to program flash or OTP memory during production. After the gauge configuration is completed during development using bqStudio and a gauge EVM, bqStudio generates a *Golden Image* with the gauge configuration. TI production tools use this Golden Image file to program flash or OTP memory.

4.1 Subclasses and Data Blocks

Data memory is organized in groups of parameters called subclasses. Each subclass consists of one or more data blocks which are up to 32 bytes long. Parameters are located within a data block. Each parameter has an offset address, length, and data type (see Table 1).

Table 1. Data Type Examples

TYPE	DESCRIPTION	MINIMUM	MAXIMUM
I1	Signed 1-byte integer	–128	127
I2	Signed 2-byte integer	–32768	32767
I4	Signed 4-byte integer	–2147483648	2147483647
U1/H1	Unsigned 1-byte integer	0	255
U2/H2	Unsigned 2-byte integer	0	65535
U4/H4	Unsigned 4-byte integer	0	4294967295
F4	4-byte floating point	± 9.8603 * 10-39	± 5.707267 * 1037
Sx	X-byte long string	1 byte	X bytes

Some gauges have additional data types. See the TRM of a particular gauge for more information.

To change a configuration parameter, follow these steps:

1. Locate the parameter in the gauge TRM.
2. Copy the subclass data for this parameter into a local byte buffer using the gauge data memory access function gauge_read_data_class (see Section 3.5).
3. Change the data in the local buffer (use the offset and data type information from the TRM for this parameter).
4. For ROM gauges only, enable configuration update mode with the function gauge_cfg_update.
5. Write the local buffer to the subclass using the function gauge_write_data_class.
6. For ROM gauges only, exit configuration update mode with the function gauge_exit.

Example:

The bq27421 has a subclass *State* (subclass ID: 81) which holds parameters associated with the state of the gauge and the cell, among them:

- Design capacity (offset: 10, length: 2 bytes, data type = I2), units = mAh
- Avg I Last Run (offset: 35, length: 2 bytes, data type = I2), units = mA

Design capacity is in data block #0 for subclass 81, so to write this parameter the two bytes at offset 10 must be changed (I2 → signed 2-byte integer).

If design capacity is 1500 mAh, the two bytes at offset 10, data block #0 are:

1500 decimal = 0x05DC → write 0x05 to offset 10 and 0xDC to offset 11.

Integer parameters are stored in big endian format within a data block. Avg I Last Run is at offset 35 in subclass *State* (data type I2). This parameter is within data block #1 because it exceeds the length of data block #0 (32 bytes). To read these parameters, combine the two bytes starting at offset 3 (35 – 32) in data block #1, and interpret the result as a signed integer.

For example, if the byte at offset 3 is 0x01 and the byte at offset 4 is 0xF4, the 2-byte signed integer is 0x1F4, which is 500 decimal so the average current during the last discharge was 500 mA.

4.2 Addressing Data Memory

This section explains the low-level access to the data memory. This section is intended as supplemental information – the APIs from the gauge example code implement this access method so it is not necessary to implement this from scratch. Data memory is addressed indirectly (it is not possible to read or write to parameters directly through the control interface).

The gauge has extended commands which allow access to parameters within data memory using the subclass, data block, and offset in the block data scheme.

- Subclass: Extended command 0x3E determines the subclass.
- Data block: Extended command 0x3F determines the data block.
- Block data: The data block (up to 32 bytes long) starts from extended command 0x40 and ends at extended command 0x5F.
- Block data check sum: Extended command 0x60 holds the block check sum.

Reading and writing data memory follows this sequence:

1. Write the subclass number to the subclass command (0x3E).
2. Write the data block number to the data block command (0x3F).
3. Read or write the data block (up to 32 bytes) starting at block data command (0x40).
4. (Write only): Calculate the check sum and write it to the check sum command (0x60).

The check sum is the sum of all 32 data bytes within the current data block, truncated to 8-bits and complemented. Example code to calculate the check sum follows:

```
pData    = a pointer to the data that was changed in the data block.
nLength    = length of the data block.

unsigned char check_sum(unsigned char *pData, unsigned char nLength)
{
    unsigned char nSum = 0x00;
    unsigned char n;

    for (n = 0; n <nLength; n++)
        nSum += pData[n];

    nSum = 0xFF - nSum;
    return nSum;
}
```

Example:

Write design capacity for the bq27421 is in subclass state (81 decimal = 0x51), offset 10 decimal = 0x0A (in data block 0 = 0x00), 1500 mAh = 0x05DC. This example uses the abstract API from Section 3.1 to read from and write to the gauge.

1. Write subclass (0x51) and block number (0x00): gauge_write(pI2C, 0x3E, “\x51\x00”, 2);
2. Read data block into buffer: gauge_read(pI2C, 0x40, pBuffer, 32);
3. Write design capacity:
   - pBuffer[0x0A] = 0x05;
   - pBuffer[0x0B] = 0xDC;
4. Write buffer to data block: gauge_write(pI2C, 0x40, pBuffer, 32);
5. Calculate check sum: unsigned char nCheckSum = check_sum(pBuffer, 32);
6. Write check sum: gauge_write(pI2C, 0x60, &nCheckSum, 1);
7. Wait 10 ms.
8. Read check sum: gauge_read(pI2C, 0x60, &nVerifyCheckSum, 1);
9. Verify that check sum matches (nCheckSum == nVerifyCheckSum).

4.3 ROM Gauge vs FLASH Gauge

ROM gauges must be placed in *configuration update* mode before writing to data memory is possible. This step is not required for flash gauges. The gauge C code has a function which selects configuration update mode as follows:

```
// gauge_cfg_update: select Configuration Update mode
// pHandle = handle to communications adapter
// return value: error = 0
int gauge_cfg_update(void *pHandle);
```

After one (or more) data classes have been written, the gauge must exit configuration update mode to resume gauging.

```
// gauge_exit: exit Configuration Update mode
// pHandle = handle to communications adapter
// nCmd = exit command (e.g. SOFT_RESET or EXIT_RESIM)
// return value: error = 0
int gauge_exit(void *pHandle, unsigned int nCmd);
```

Example (write Data Class *State* for bq27421):

```
gauge_cfg_update(pI2C); // required for ROM gauge
gauge_write_data_class(pI2C, DC_STATE, pData, DC_STATE_LENGTH);
gauge_exit(pI2C, SOFT_RESET); // required for ROM gauge
```

5 FlashStream® File Format

Software tools for gauges use flashstream files for gauge configuration. There are several different sub-types of flashstream files, which follow:

- Golden Image files: *.gm.fs (contain configuration for ROM gauge – for data memory in RAM).
- Configuration files for flash gauges: *.df.fs
- Firmware and configuration files for flash gauges: *.bq.fs
- Configuration for one time programmable gauges: *.ot.fs

Every flash stream type is a text file and shares the same syntax, as follows:

- Comments – start with a semicolon → ;This is an example of a comment.
- Write command – W: [device address] [command] [data][data]…[data]

  Example: Write 0x02 to command 0x3E and write 0x00 to command 0x3F for a gauge with device address 0xAA

  W: AA 3E 02 00
- Compare command – C: [device address] [command] [data][data]…[data]

  Example: Compare the contents from a gauge with device address 0xAA, starting at command 0x3E with byte sequence 0x02, 0x00, 0x02, 0x20, 0x00, 0x03

  C: AA 3E 02 00 02 20 00 03

  If the byte sequence matches, continue, otherwise stop (error).
- Delay command – X: [delay in milliseconds]

  Example: Wait 10 ms

  X: 10

Source Code

A.1 gauge.c

```
//Battery Gauge Library
//V1.0
//© 2016 Texas Instruments Inc.

#include <string>
#include "gauge.h"

#define SET_CFGUPDATE    0x0013

#define CMD_DATA_CLASS    0x3E
#define CMD_BLOCK_DATA    0x40
#define CMD_CHECK_SUM     0x60
#define CMD_FLAGS         0x06

#define CFGUPD            0x0010

//gauge_read: read bytes from gauge (must be implemented for a specific system)
//pHandle:    handle to communications adapater
//nRegister:    first register (=standard command) to read from
//pData:    pointer to a data buffer
//nLength:    number of bytes
//return value:    number of bytes read (0 if error)
extern int gauge_read(void *pHandle, unsigned char nRegister, unsigned char *pData, unsigned char
nLength);
//gauge_read: write bytes to gauge (must be implemented for a specific system)
//pHandle:    handle to communications adapater
//nRegister:    first register (=standard command) to write to
//pData:    pointer to a data buffer
//nLength:    number of bytes
//return value:    number of bytes written (0 if error)
extern int gauge_write(void *pHandle, unsigned char nRegister, unsigned char *pData, unsigned
char nLength);
//gauge_address: set device address for gauge (must be implemented for a specific system; not
required for HDQ)
//pHandle:    handle to communications adapater
//nAddress:    device address (e.g. 0xAA)
extern void gauge_address(void *pHandle, unsigned char nAddress);

//gauge_control: issues a sub command
//pHandle: handle to communications adapter
//nSubCmd: sub command number
//return value:    result from sub command
```

```
unsigned int gauge_control(void *pHandle, unsigned int nSubCmd)
{
    unsigned int nResult = 0;

    char pData[2];

    pData[0] = nSubCmd & 0xFF;
    pData[1] = (nSubCmd >> 8) & 0xFF;

    gauge_write(pHandle, 0x00, pData, 2); // issue control and sub command

    gauge_read(pHandle, 0x00, pData, 2); // read data

    nResult = (pData[1] <<8) | pData[0];

    return nResult;
}

//gauge_cmd_read:    read data from standard command
//pHandle: handle to communications adapter
//nCmd:    standard command
//return value:    result from standard command
unsigned int gauge_cmd_read(void *pHandle, unsigned char nCmd)
{
    unsigned char pData[2];

    gauge_read(pHandle, nCmd, pData, 2);

    return (pData[1] << 8) | pData[0];
}

//gauge_cmd_write:    write data to standard command
//pHandle: handle to communications adapter
//nCmd:    standard command
//return value:    number of bytes written to gauge
unsigned int gauge_cmd_write(void *pHandle, unsigned char nCmd, unsigned int nData)
{
    unsigned char pData[2];

    pData[0] = nData & 0xFF;
    pData[1] = (nData >> 8) & 0xFF;

    return gauge_write(pHandle, nCmd, pData, 2);
}

//gauge_cfg_update:    enter configuration update mode for rom gauges
//pHandle: handle to communications adapter
//return value:    true = success, false = failure
#define MAX_ATTEMPTS 5
bool gauge_cfg_update(void *pHandle)
{
    unsigned int nFlags;
    int nAttempts = 0;
    gauge_control(pHandle, SET_CFGUPDATE);

    do
    {
        nFlags = gauge_cmd_read(pHandle, CMD_FLAGS);
        if (!(nFlags & CFGUPD)) usleep(500000);
    } while (!(nFlags & CFGUPD) && (nAttempts++ < MAX_ATTEMPTS));

    return (nAttempts < MAX_ATTEMPTS);
}
```

```
//gauge_exit:    exit configuration update mode for rom gauges
//pHandle: handle to communications adapter
//nSubCmd: sub command to exit configuration update mode
//return value:    true = success, false = failure
bool gauge_exit(void *pHandle, unsigned int nSubCmd)
{
    unsigned int nFlags;
    int nAttempts = 0;
    gauge_control(pHandle, nSubCmd);

    do
    {
        nFlags = gauge_cmd_read(pHandle, CMD_FLAGS);
        if (nFlags & CFGUPD) usleep(500000);
    } while ((nFlags & CFGUPD) && (nAttempts++ <MAX_ATTEMPTS));

    return (nAttempts < MAX_ATTEMPTS);
}

//gauge_read_data_class:    read a data class
//pHandle: handle to communications adapter
//nDataClass:    data class number
//pData:    buffer holding the whole data class (all blocks)
//nLength:    length of data class (all blocks)
//return value:    0 = success
int gauge_read_data_class(void *pHandle, unsigned char nDataClass, unsigned char *pData, unsigned
char nLength)
{
    unsigned char nRemainder = nLength;
    unsigned int nOffset = 0;
    unsigned char nDataBlock = 0x00;
    unsigned int nData;

     if (nLength < 1) return 0;

    do
    {

        nLength = nRemainder;
        if (nLength > 32)
        {
            nRemainder = nLength - 32;
            nLength = 32;
        }
        else nRemainder = 0;

        nData = (nDataBlock << 8) | nDataClass;
        gauge_cmd_write(pHandle, CMD_DATA_CLASS, nData);

        if (gauge_read(pHandle, CMD_BLOCK_DATA, pData, nLength) != nLength) return -1;

        pData += nLength;
        nDataBlock++;
    } while (nRemainder > 0);

    return 0;
}
```

```
//check_sum:    calculate check sum for block transfer
//pData:    pointer to data block
//nLength:    length of data block
unsigned char check_sum(unsigned char *pData, unsigned char nLength)
{
    unsigned char nSum = 0x00;
    unsigned char n;

    for (n = 0; n < nLength; n++)
        nSum += pData[n];

    nSum = 0xFF - nSum;

    return nSum;
}

//gauge_write_data_class:    write a data class
//pHandle: handle to communications adapter
//nDataClass:    data class number
//pData:    buffer holding the whole data class (all blocks)
//nLength:    length of data class (all blocks)
//return value:    0 = success
int gauge_write_data_class(void *pHandle, unsigned char nDataClass, unsigned char *pData,
unsigned char nLength)
{
    unsigned char nRemainder = nLength;
    unsigned int nOffset = 0;
    unsigned char pCheckSum[2] = {0x00, 0x00};
    unsigned int nData;
    unsigned char nDataBlock = 0x00;

    if (nLength < 1) return 0;

    do
    {
        nLength = nRemainder;
        if (nLength < 32)
        {
            nRemainder = nLength - 32;
            nLength = 32;
        }
        else nRemainder = 0;

        nData = (nDataBlock << 8) | nDataClass;
        gauge_cmd_write(pHandle, CMD_DATA_CLASS, nData);

        if (gauge_write(pHandle, CMD_BLOCK_DATA, pData, nLength) != nLength) return -1;
        pCheckSum[0] = check_sum(pData, nLength);
        gauge_write(pHandle, CMD_CHECK_SUM, pCheckSum, 1);

        usleep(10000);

        gauge_cmd_write(pHandle, CMD_DATA_CLASS, nData);
        gauge_read(pHandle, CMD_CHECK_SUM, pCheckSum + 1, 1);
        if (pCheckSum[0] != pCheckSum[1]) return -2;

        pData += nLength;
        nDataBlock++;
    } while (nRemainder > 0);

    return 0;
}
```

```
//gauge_execute_fs:    execute a flash stream file
//pHandle: handle to communications adapter
//pFS:    zero-terminated buffer with flash stream file
//return value:    success: pointer to end of flash stream file
//error: point of error in flash stream file
char *gauge_execute_fs(void *pHandle, char *pFS)
{
    int nLength = strlen(pFS);
    int nDataLength;
    char pBuf[16];
    char pData[32];
    int n, m;
    char *pEnd = NULL;
    char *pErr;
    bool bWriteCmd = false;
    unsigned char nRegister;

    m = 0;
    for (n = 0; n < nLength; n++)
        if (pFS[n] != ' ') pFS[m++] = pFS[n];
    pEnd = pFS + m;
    pEnd[0] = 0;

    do
    {
        switch (*pFS)
        {
            case ';':
                break;
            case 'W':
            case 'C':
                bWriteCmd = *pFS == 'W';
                pFS++;
                if ((*pFS) != ':') return pFS;
                pFS++;

                n = 0;
                while ((pEnd - pFS > 2) && (n < sizeof(pData) + 2) &&(*pFS != '\n'))
                {
                    pBuf[0] = *(pFS++);
                    pBuf[1] = *(pFS++);
                    pBuf[2] = 0;

                    m = strtoul(pBuf, &pErr, 16);
                    if (*pErr) return (pFS - 2);

                    if (n == 0)    gauge_address(pHandle, m);
                    if (n == 1) nRegister = m;
                    if (n > 1) pData[n - 2] = m;
                    n++;
                }

                if (n < 3) return pFS;
                nDataLength = n - 2;

                if (bWriteCmd)
                    gauge_write(pHandle, nRegister, pData, nDataLength);
                else
                {
                    char pDataFromGauge[nDataLength];

                    gauge_read(pHandle, nRegister, pDataFromGauge, nDataLength);

                    if (memcmp(pData, pDataFromGauge, nDataLength)) return pFS;
                }
                break;
            case 'X':
                pFS++;
                if ((*pFS) != ':') return pFS;
                pFS++;
                n = 0;
                while ((pFS != pEnd) && (*pFS != '\n') &&(n <sizeof(pBuf) - 1))
                {
                    pBuf[n++] = *pFS;
                    pFS++;
                }
                pBuf[n] = 0;
                n = atoi(pBuf);
                usleep(n * 1000);
                break;
            default: return pFS;
        }

        while ((pFS != pEnd) && (*pFS != '\n')) pFS++; //skip to next line
        if (pFS != pEnd) pFS++;

    }    while (pFS != pEnd);

    return pFS;
}
```

A.2 gauge.h

```
//Battery Gauge Library
//V1.0
//© 2016 Texas Instruments Inc.

#ifndef __GAUGE_H
#define __GAUGE_H

#include <stdbool.h>
#define SOFT_RESET 0x0042

//gauge_control:    issues a sub command
//pHandle: handle to communications adapter
//nSubCmd: sub command number
//return value:    result from sub command
unsigned int gauge_control(void *pHandle, unsigned int nSubCmd);

//gauge_cmd_read:    read data from standard command
//pHandle: handle to communications adapter
//nCmd:    standard command
//return value:    result from standard command
unsigned int gauge_cmd_read(void *pHandle, unsigned char nCmd);

//gauge_cmd_write:    write data to standard command
//pHandle: handle to communications adapter
//nCmd:    standard command
//return value:    number of bytes written to gauge
unsigned int gauge_cmd_write(void *pHandle, unsigned char nCmd, unsigned int nData);

//gauge_cfg_update:    enter configuration update mode for rom gauges
//pHandle: handle to communications adapter
//return value:    true = success, false = failure
bool gauge_cfg_update(void *pHandle);

//gauge_exit:    exit configuration update mode for rom gauges
//pHandle: handle to communications adapter
//nSubCmd: sub command to exit configuration update mode
//return value:    true = success, false = failure
bool gauge_exit(void *pHandle, unsigned int nSubCmd);

//gauge_read_data_class:    read a data class
//pHandle: handle to communications adapter
//nDataClass:    data class number
//pData:    buffer holding the whole data class (all blocks)
//nLength:    length of data class (all blocks)
//return value:    0 = success
int gauge_read_data_class(void *pHandle, unsigned char nDataClass, unsigned char *pData, unsigned
char nLength);

//gauge_write_data_class:    write a data class
//pHandle: handle to communications adapter
//nDataClass:    data class number
//pData:    buffer holding the whole data class (all blocks)
//nLength:    length of data class (all blocks)
//return value:    0 = success
int gauge_write_data_class(void *pHandle, unsigned char nDataClass, unsigned char *pData,
unsigned char nLength);

//gauge_execute_fs:    execute a flash stream file
//pHandle: handle to communications adapter
//pFS:    zero-terminated buffer with flash stream file
//return value:    success: pointer to end of flash stream file
//error: point of error in flashstream file
char *gauge_execute_fs(void *pHandle, char *pFS);

#endif
```

A.3 Example Implementation of Gauge APIs for Linux® User Space I²C/ dev Interface

```
//Example for battery gauge communication using Linux User Space I²C /dev interface
//V1.0
//© 2016 Texas Instruments Inc.

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <linux/i2c-dev.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include "gauge.h"

#define GAUGE_DEVICE_ADDRESS 0xAA

typedef struct
{
    int nI2C;
    unsigned char nAddress;
} TI2C;

int gauge_read(void *pHandle, unsigned char nRegister, unsigned char *pData, unsigned char
nLength)
{
    TI2C *pI2C = (TI2C *) pHandle;
    int n;

    if (nLength < 1) return 0;

    pData[0] = nRegister;
    n = write(pI2C->nI2C, pData, 1);    // write register address

    n = read(pI2C->nI2C, pData, nLength);    // read data from register

    usleep(100);

    return n;
}

int gauge_write(void *pHandle, unsigned char nRegister, unsigned char *pData, unsigned char
nLength)
{
    TI2C *pI2C = (TI2C *) pHandle;
    unsigned char pWriteData[nLength + 1];
    int n;

    if (nLength < 1) return 0;

    pWriteData[0] = nRegister;    // write register address before writing data
    memcpy(pWriteData + 1, pData, nLength);

    n = write(pI2C->nI2C, pWriteData, nLength + 1);

    usleep(100);

    return n - 1;
}

void gauge_address(void *pHandle, unsigned char nAddress)
{
    TI2C *pI2C = (TI2C *) pHandle;
    if (nAddress != pI2C->nAddress)
        ioctl(pI2C->nI2C, I2C_SLAVE, nAddress >>1);
    pI2C->nAddress = nAddress;
}
```

```
void print_data(unsigned char *pData, unsigned int nLength)
{
    unsigned int n;

    printf("  ");
    for (n = 0; n < nLength; n++)
    {
        printf("%02X ", pData[n]);
        if (!((n + 1) % 16)) printf("\n\r  ");
    }

    printf("\n\r");
}

#define SOURCE_FILE "test.gm.fs"

#define CMD_VOLTAGE                0x04

#define SUB_CMD_FW_VERSION        0x0002
#define SUB_CMD_CONTROL_STATUS    0x0000

#define DC_STATE                0x52
#define DC_STATE_LENGTH            64
#define DESIGN_CAPACITY            3210    //[mAh]
#define NOMINAL_VOLTAGE            3.7        //[V]
#define DESIGN_ENERGY            ((unsigned int) (DESIGN_CAPACITY * NOMINAL_VOLTAGE))
#define TERMINATE_VOLTAGE        3000    //[mV]
#define TAPER_CURRENT            115        //[mA]
#define TAPER_RATE                ((unsigned int) (DESIGN_CAPACITY / (0.1 * TAPER_CURRENT)))

int main()
{
    TI2C i2c;
     void *pHandle = (void *) & i2c;
    int nSourceFile;
    struct stat st;
    long n;
    int nSeconds;
    unsigned int nResult;
    char *pFileBuffer;

    unsigned char pData[DC_STATE_LENGTH];

    printf("gauge test\n\r");

    if ((i2c.nI2C = open("/dev/i2c-1", O_RDWR)) <0)
    {
        printf("cannot open I2C bus\n\r");
        exit(1);
    }

    printf("openend I2C bus\n\r");

    gauge_address(pHandle, GAUGE_DEVICE_ADDRESS);

    nResult = gauge_control(pHandle, SUB_CMD_FW_VERSION);
    printf("  FW_VERSION = 0x%04X\n\r", nResult);

    nResult = gauge_cmd_read(pHandle, CMD_VOLTAGE);
    printf("  VOLTAGE = %04d [mV]\n\r", nResult);

    nResult = gauge_control(pHandle, SUB_CMD_CONTROL_STATUS);
    printf("  CONTROL_STATUS = 0x%04X\n\r", nResult);

    stat(SOURCE_FILE, &st);
```

```
        printf("source file '%s', size = %d\n\r", SOURCE_FILE, st.st_size);
        if ((nSourceFile = open(SOURCE_FILE, O_RDONLY)) <  0)
        {
            printf("cannot open data classes source file\n\r");
            exit(1);
        }
        pFileBuffer = malloc(st.st_size);
        if (!pFileBuffer) exit(1);
        read(nSourceFile, pFileBuffer, st.st_size);
        close(nSourceFile);

        printf(gauge_execute_fs(pHandle, pFileBuffer));

        free(pFileBuffer);

    //read data class DC_STATE:
        n = gauge_read_data_class(pHandle, DC_STATE, pData, DC_STATE_LENGTH);
        if (n) printf("Error reading data class, %d\n\r", n);

        printf("Data Class 'State' (0x52):\n\r");
        print_data(pData, DC_STATE_LENGTH);

    // this was for bq2742x - change offsets for your gauge
        pData[10] = (DESIGN_CAPACITY & 0xFF00) >> 8;
        pData[11] = DESIGN_CAPACITY & 0xFF;
        pData[12] = (DESIGN_ENERGY & 0xFF00) >> 8;
        pData[13] = DESIGN_ENERGY &  0xFF;
     pData[16] = (TERMINATE_VOLTAGE & 0xFF00) >>8;
     pData[17] = TERMINATE_VOLTAGE & 0xFF;
     pData[27] = (TAPER_RATE & 0xFF00) >> 8;
     pData[28] = TAPER_RATE & 0xFF;

    //write data class DC_STATE:
        gauge_cfg_update(pHandle);
        n = gauge_write_data_class(pHandle, DC_STATE, pData, DC_STATE_LENGTH);
        if (n) printf("Error writing data class, %d\n\r", n);
        gauge_exit(pHandle, SOFT_RESET);

        close(i2c.nI2C);
        printf("closed I2C bus\n\r");

        return 0;
    }
```