Application Report

SLUA640B–August 2012–Revised May 2016

Host System Calibration Method

Jared Casey

ABSTRACT

To add improvements to the Impedance Track™ algorithm, changes have been made to free up firmware code space. In the process of freeing up code space, the calibration calculations are no longer automatically performed by the gauge but must be performed by the host and the results written back to the gauge. This application note provides a flow on how to implement the calibration algorithm on a host device.

1 Gauges That Use the Host System Calibration Method

The host system calibration method is utilized by gauges that implement the Impedance Track (IT) algorithm as well as Impedance Track with Dynamic Voltage Correlation (IT–DVC) algorithm. Note that gauges that implement the IT-DVC algorithm only need to perform the voltage and temperature calibrations.

1.1 Impedance Track Gauges (as of time of writing)

- bq27542-G1
- bq27545-G1
- bq27546-G1
- bq27510-G3
- bq27520-G4
- bq27530-G1
- bq27531-G1
- bq27532-G1
- bq27742-G1

1.2 Impedance Track with Dynamic Voltage Correlation (as of time of writing)

- bq27621-G1

2 General I^{2}C Command Information

In the following flow charts, all I^{2}C functions take 3 arguments.

Write command arguments:

1. Address
2. Data
3. Wait time in ms

Read command arguments:

1. Address
2. Number of bytes read
3. Wait time in ms

Impedance Track is a trademark of Texas Instruments.

3 Calibration Method

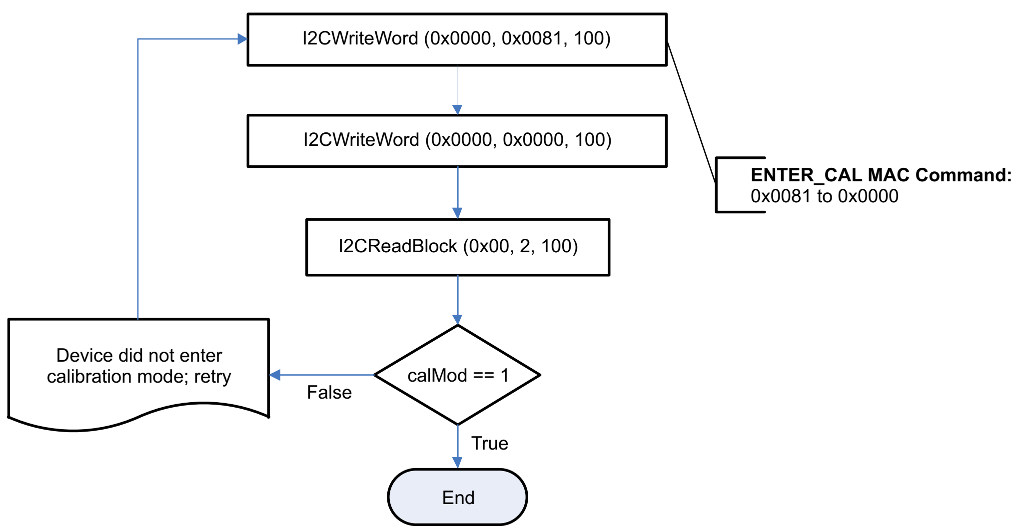
The calibration method is broken up into the following sections:

1. Enter Calibration Mode
2. Exit Calibration Mode
3. CC Offset
4. Board Offset
5. Obtain Raw Calibration Data
6. Current Calibration
7. Voltage Calibration
8. Temperature Calibration
9. Floating Point Conversion

Each section includes a sample code excerpt for an implementation of the calibration step.

4 Enter Calibration Mode

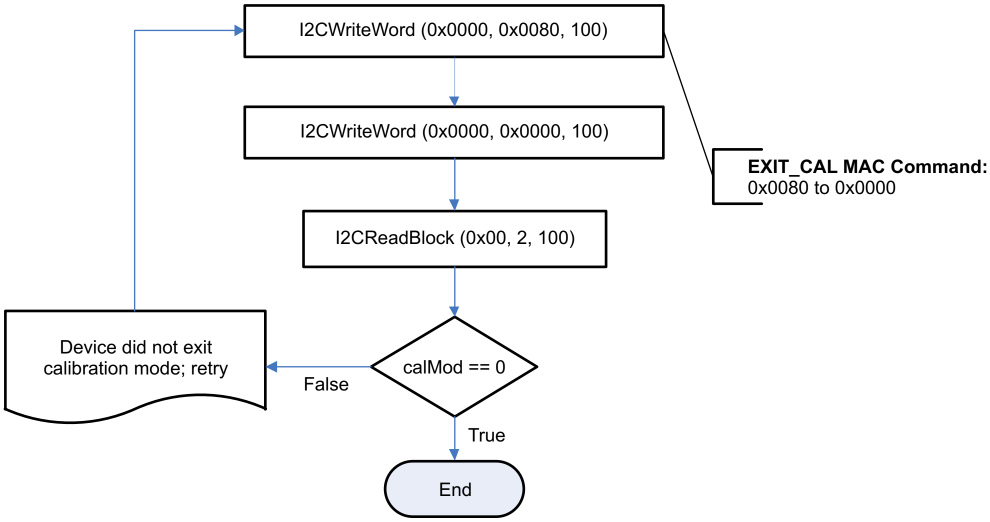
This sequence puts the gauge into CALIBRATION mode. These steps must be performed when the gauge is in the UNSEALED mode.




```
void enterCalibrationMode(void) {
    printf("Entering Calibration Mode.\n");

    send_subCommand(0x00, 0x2D); //Enable Calibration mode
    send_subCommand(0x00, 0x81); //Enter Calibration mode
    send_subCommand(0x00, 0x00);
    send_Command(0x00);
    unsigned char buffer[2] = { 0x00, 0x00 };
    _delay_cycles(10000);
    I2C_read(buffer, 2);

    do {
        printf("%x\n", buffer[1]);
        printf("Gauge is not in Calibration mode.\n");

    } while ((buffer[1] & 0x10) != 0x10);

    printf("Gauge is in Calibration mode.\n");

}
```

5 Exit Calibration Mode

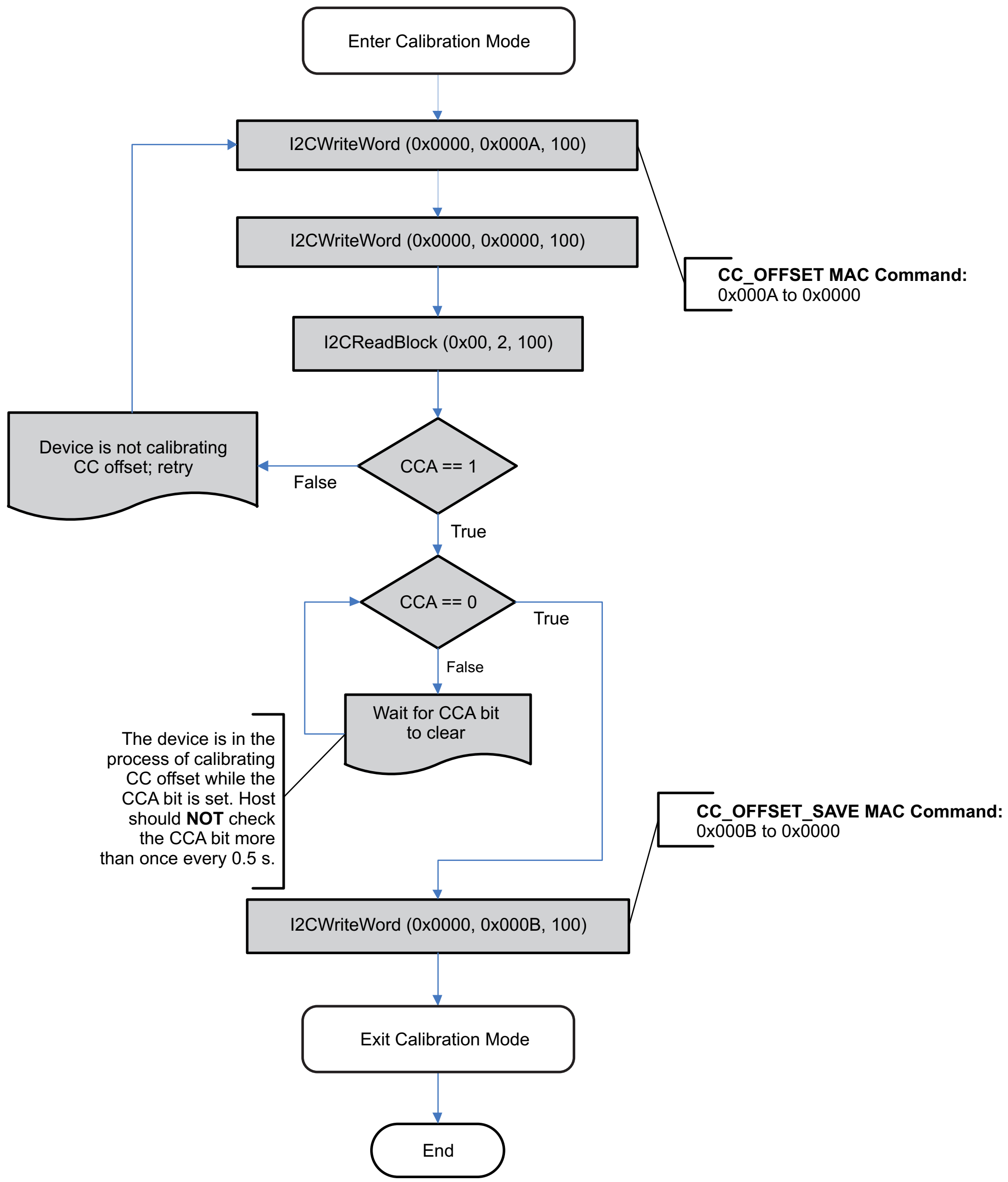
This sequence takes the gauge out of CALIBRATION mode. These steps must be performed when the gauge is in the UNSEALED mode.



6 CC Offset

Use MAC commands for CC Offset calibration. The host system does not need to write information to the Data Flash (DF). Refer to the fuel gauge data sheet for the location of the *[CCA]* bit. The host system needs to ensure the fuel gauge is unsealed.

NOTE: While the device is calibrating the CC Offset, the host system must not read the *ControlStatus()* Register at a rate greater than once every 0.5 seconds.



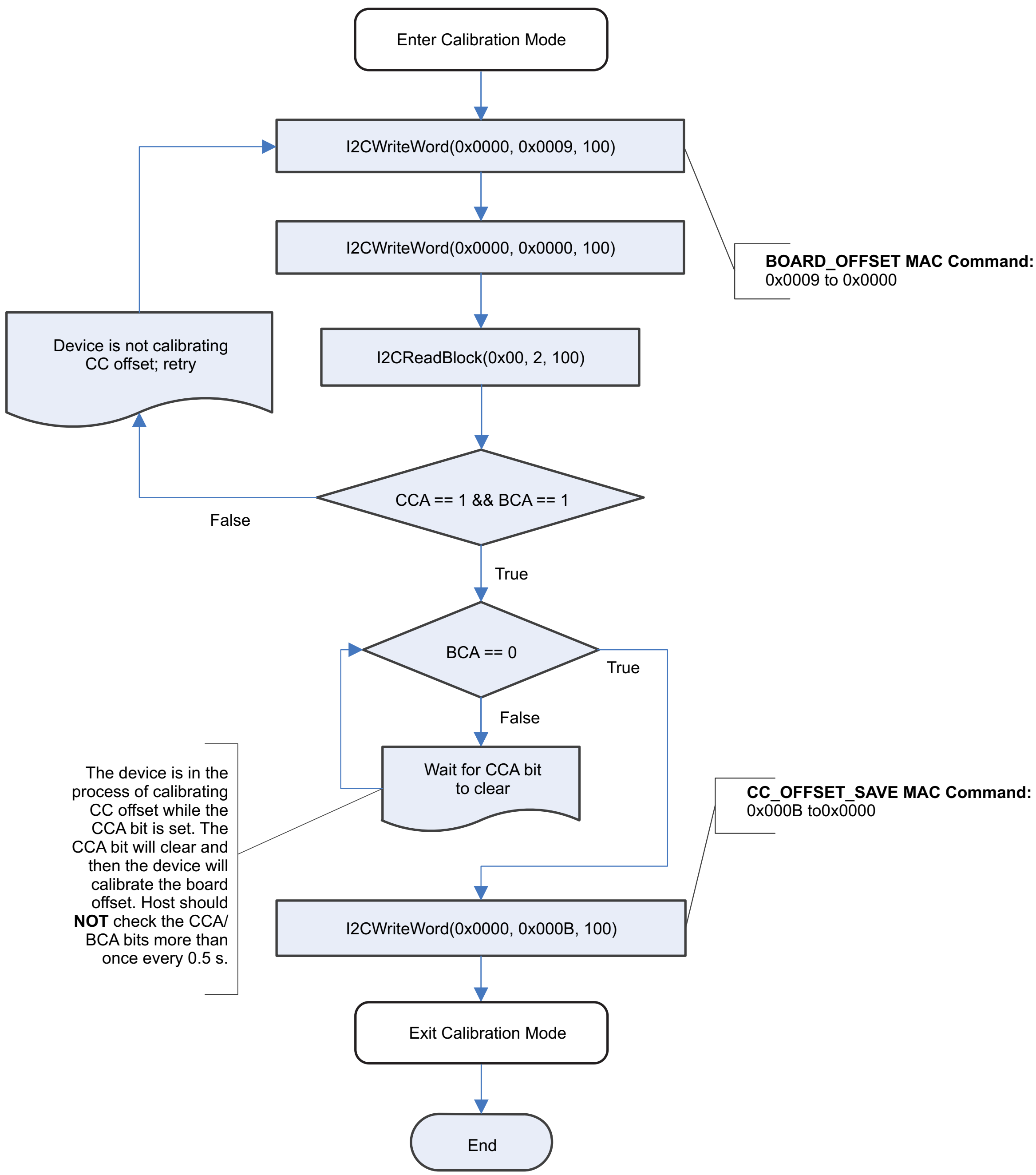
```
void calibrate_CC_Offset(void) {
    enterCalibrationMode(); //Enter Calibration Mode

    unsigned char buffer[2] = { 0x00, 0x00 };

    //Perform CC Offset calibration
    do {
        send_subCommand(0x00, 0x0A);
        send_subCommand(0x00, 0x00);
        send_Command(0x00);
        _delay_cycles(10000);
        I2C_read(buffer, 2);

    } while ((buffer[1] & 0x08) != 0x08);

    printf("Calibrating...\n");

    while ((buffer[1] & 0x08) == 0x08) {
        printf("%x\n", buffer[1]);
        send_subCommand(0x00, 0x00);
        send_Command(0x00);
        _delay_cycles(6000000); //0.5s delay
        I2C_read(buffer, 2);
        if ((buffer[1] & 0x08) == 0x00) {
            printf("%x\n", buffer[1]);
            printf("Done Calibrating CC offset.\n");
            break;
        }

    }

    send_subCommand(0x00, 0x80); //Exit Calibration mode

}
```

7 Board Offset

Use MAC commands for Board Offset calibration. The host system does not need to write information to the DF. The host system needs to ensure the fuel gauge is unsealed. Refer to the fuel gauge data sheet for the location of the *[CCA]* and *[BCA]* bits.

NOTE: Calculating the **Board Offset** also calculates the **CC Offset**; therefore, it is not necessary to go through the **CC Offset** calibration process if the **Board Offset** calibration process is implemented.

NOTE: While the device is calibrating the CC Offset, the host system should not read the *ControlStatus()* Register at a rate greater than once every 0.5 seconds.

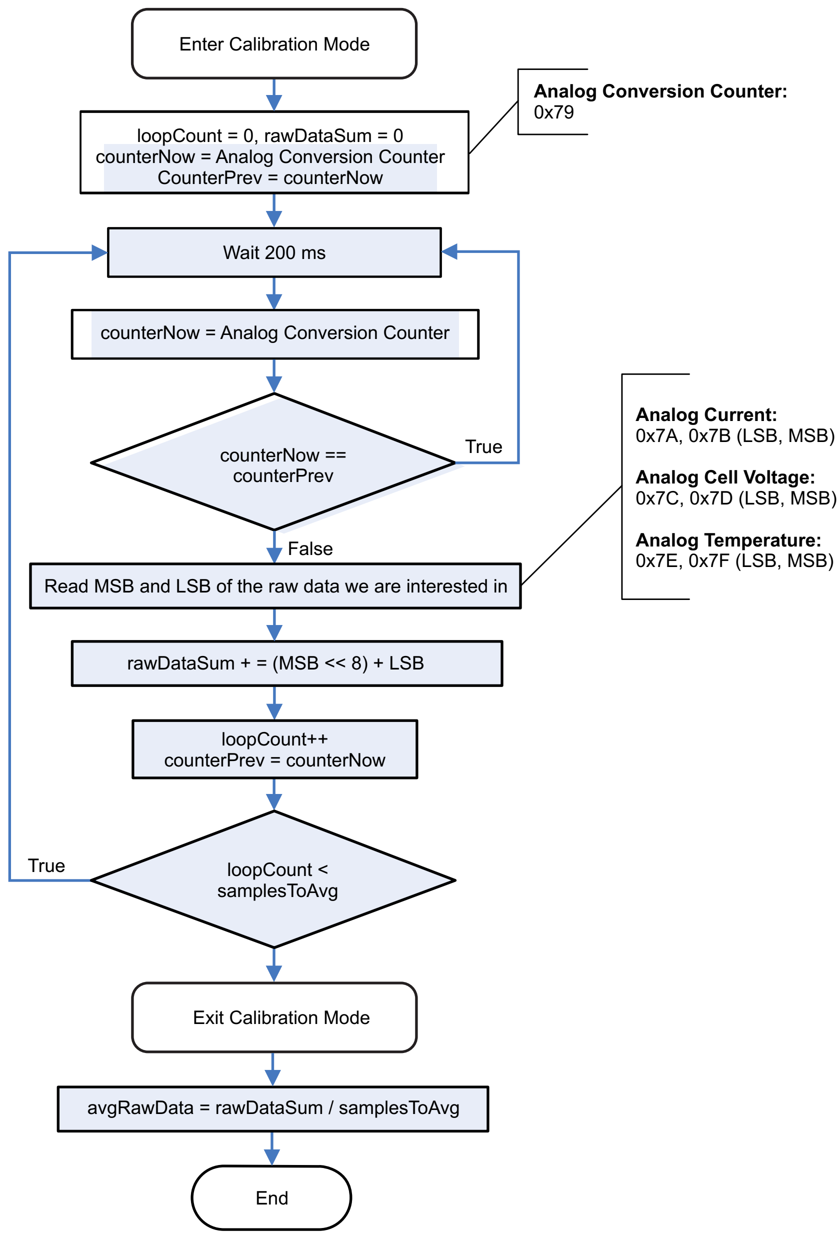


```
void calibrate_Board_Offset(void) {

    enterCalibrationMode(); //Enter Calibration Mode

    unsigned char buffer[2] = { 0x00, 0x00 };

    //Perform Board Offset calibration
    do {
        send_subCommand(0x00, 0x09);
        send_subCommand(0x00, 0x00);
        send_Command(0x00);
        _delay_cycles(10000);
        I2C_read(buffer, 2);

    } while ((buffer[1] & 0x0C) != 0x0C);

    printf("Calibrating...\n");

    while ((buffer[1] & 0x0C) == 0x0C) {
        printf("Calibrating...\n");
        printf("%x\n", buffer[1]);
        send_subCommand(0x00, 0x00);
        send_Command(0x00);
        _delay_cycles(10000000); //0.5s delay
        I2C_read(buffer, 2);
        if ((buffer[1] & 0x0C) == 0x00) {
            printf("%x\n", buffer[1]);
            printf("Done Calibrating Board offset.\n");
            break;
        }

    }

    send_subCommand(0x00, 0x80); //Exit Calibration mode

}
```

8 Obtain Raw Calibration Data

The following flow chart demonstrates how the host system obtains the raw data to calibrate current, voltage, and temperature. The host system uses this flow in conjunction with the Section 8, Section 9, Section 10, and Section 11 flows described in this application report. It is recommended that the host system samples the raw data multiple times, at a rate of once per second, to obtain an average of the raw current, voltage, and temperature. The host system needs to ensure the fuel gauge is unsealed.

The extended command, 0x79, returns a counter that the host system can use to determine if the raw data sample is a newer sample than the previous sample read. If the *Analog Conversion Counter* has not increased by at least 1 count in between reads, then the host should wait approximately 200 ms, until the counter is checked again. The counter can be larger than just a single count. The loop should exit when the number of averaged samples has been obtained, but the host should not read from the fuel gauge until the *Analog Conversion Counter* has increased by at least one count.



```
/**
 * Use this function to obtain raw calibration data from the gauge
 * when calibrating voltage, current or temperature.
 * dataWanted = 1 for voltage, 2 for current, 3 for temperature
 */
void obtainRawCalibrationData(int dataWanted) {

    enterCalibrationMode();
    unsigned char analogConversionCounter[1] = { 0x79 }; //Analog Conversion Counter extended
command must be a direct single-byte write to the gauge.
    unsigned char buffer[1] = { 0x00 };
    int loopCount = 0, counterNow = 0, counterPrev = 0, samples = 15;

    //Obtain the desired raw Data
    uint16_t rawDataSum = 0;

    switch (dataWanted) {
    case 1:

        I2C_write(analogConversionCounter, 1);
        _delay_cycles(10000);
        I2C_read(buffer, 1);
        counterNow = (int) buffer[0]; //Initialize ADC conversion counter
        counterPrev = counterNow;

        for (loopCount = 0; loopCount < samples;) {

            if (counterNow != counterPrev) {
                read_Register(0x7C);
                rawDataSum +=
                (((uint16_t) block[1] << 8) + (uint16_t) block[0]);
                loopCount++;
                counterPrev = counterNow;
            } else {
                _delay_cycles(10000);
                I2C_write(analogConversionCounter, 1);
                _delay_cycles(10000);
                I2C_read(buffer, 1);
                counterNow = (int) buffer[0];

            }

        }

        avgRawData = rawDataSum / samples;
        printf("avgRawData is %X\n", avgRawData);
        break;
```

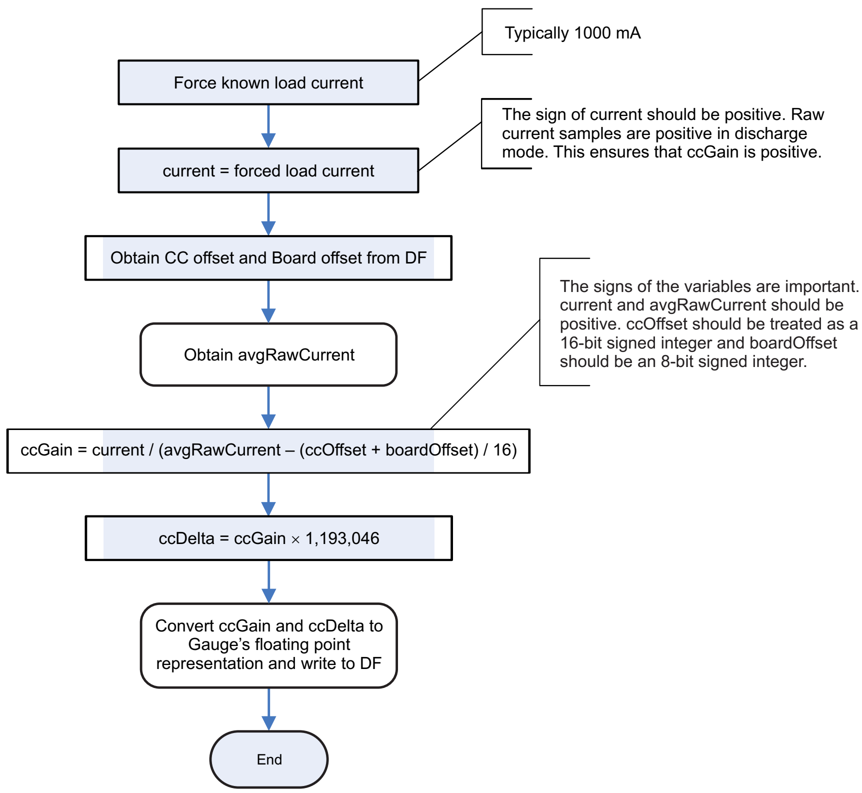
```
case 2:
        I2C_write(analogConversionCounter, 1);
        _delay_cycles(10000);
        I2C_read(buffer, 1);
        counterNow = (int) buffer[0]; //Initialize ADC conversion counter
        counterPrev = counterNow;

        for (loopCount = 0; loopCount < samples;) {

            if (counterNow != counterPrev) {
                read_Register(0x7A);
                rawDataSum +=
                (((uint16_t) block[1] << 8) + (uint16_t)block[0]);
                loopCount++;
                counterPrev = counterNow;
            } else {
                _delay_cycles(10000);
                I2C_write(analogConversionCounter, 1);
                I2C_read(buffer, 1);
                counterNow = (int) buffer[0];

            }
        }
        avgRawData = rawDataSum / samples;
        break;
    case 3:
        I2C_write(analogConversionCounter, 1);
        _delay_cycles(10000);
        I2C_read(buffer, 1);
        counterNow = (int) buffer[0]; //Initialize ADC conversion counter
        counterPrev = counterNow;

        for (loopCount = 0; loopCount < samples;) {

            if (counterNow != counterPrev) {
                read_Register(0x7E);
                rawDataSum +=
                (((uint16_t) block[1] << 8) + (uint16_t)block[0]);
                loopCount++;
                counterPrev = counterNow;
            } else {
                _delay_cycles(10000);
                I2C_write(analogConversionCounter, 1);
                I2C_read(buffer, 1);
                counterNow = (int) buffer[0];
            }
        }

        avgRawData = rawDataSum / samples;
        break;
    }

    send_subCommand(0x00, 0x80); //Exit Calibration mode
}
```

9 Current Calibration

The **CC Gain** and **CC Delta** are two calibration parameters of concern for current calibration. A known load, typically 1000 mA, is applied to the device during this process. Details on converting the **CC Gain** and **CC Delta** to floating point format are in *Obtain Raw Calibration Data*. The host system must ensure the fuel gauge is unsealed.

NOTE: The step labeled **Obtain avgRawCurrent** refers to *Obtain Raw Calibration Data*.

The step labeled **Convert ccGain and ccDelta to Gauge's floating point representation and write to DF** refers to *Floating Point Conversion*.

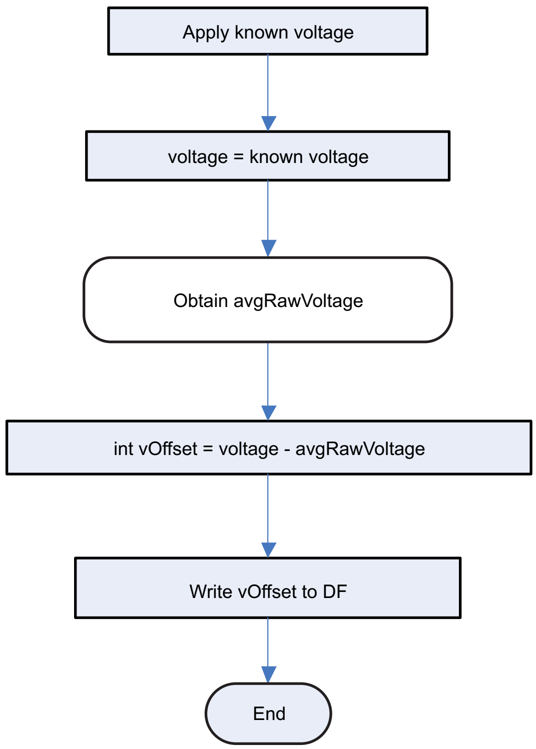


```
void calibrateCurrent(int knownCurrent) {
    printf("Begin Calibrating Current.\n");
    obtainRawCalibrationData(2);

    send_extendedCommand(0X3E, 0x00, 0x68);
    read_Block();
    int ccOffset = (((uint16_t) block[8] << 8) + (uint16_t) block[9]);
    signed char boardOffset = block[10];
    float ccGain = (float) (knownCurrent
            / (float) ((int) avgRawData - ((ccOffset + boardOffset) / 16)));
    printf("ccGain is %3.5f\n", ccGain);

    float ccDelta = ccGain * 1193046;
    printf("ccDelta is %3.5f\n", ccDelta);

    floating2Byte(ccGain);
    send_extendedCommand(0X3E, 0x00, 0x68);
    read_Block();
    read_Checksum();
    printf("oldChk is %x\n", checksum);
    unsigned char newChecksum = calculate_New_Checksum_4B(0, rawData[0],
            rawData[1], rawData[2], rawData[3]); //Calculate new checksum based on 4 bytes
calculated from ccGain.
    printf("newChk is %x\n", newChecksum);

    send_extendedCommand_1B((0x40 + (0 % 32)), rawData[0]);
    send_extendedCommand_1B((0x40 + (1 % 32)), rawData[1]);
    send_extendedCommand_1B((0x40 + (2 % 32)), rawData[2]);
    send_extendedCommand_1B((0x40 + (3 % 32)), rawData[3]);

    send_extendedCommand_1B(0x60, newChecksum);
    _delay_cycles(100000);

    floating2Byte(ccDelta);
    send_extendedCommand(0X3E, 0x00, 0x68);
    read_Block();
    read_Checksum();
    printf("oldChk is %x\n", checksum);
    newChecksum = calculate_New_Checksum_4B(4, rawData[0], rawData[1],
            rawData[2], rawData[3]); //Calculate new checksum based on 4 bytes calculated from
ccDelta.
    printf("newChk is %x\n", newChecksum);

    send_extendedCommand_1B((0x40 + (4 % 32)), rawData[0]);
    send_extendedCommand_1B((0x40 + (5 % 32)), rawData[1]);
    send_extendedCommand_1B((0x40 + (6 % 32)), rawData[2]);
    send_extendedCommand_1B((0x40 + (7 % 32)), rawData[3]);

    send_extendedCommand_1B(0x60, newChecksum);
    _delay_cycles(100000);

    send_extendedCommand(0X3E, 0x00, 0x68);
    read_Block();
    printf("Done Calibrating Current.\n");
}
```

10 Voltage Calibration

A known voltage must be applied to the device for voltage calibration. The calculated voltage offset must be written to the corresponding location in DF. The voltage offset is represented by an integer that is a single byte in size and can be written to the appropriate location in DF without any intermediate steps. The host system must ensure the fuel gauge is unsealed.

NOTE: The step labeled **Obtain avgRawVoltage** refers to the Obtain Raw Calibration Data section.

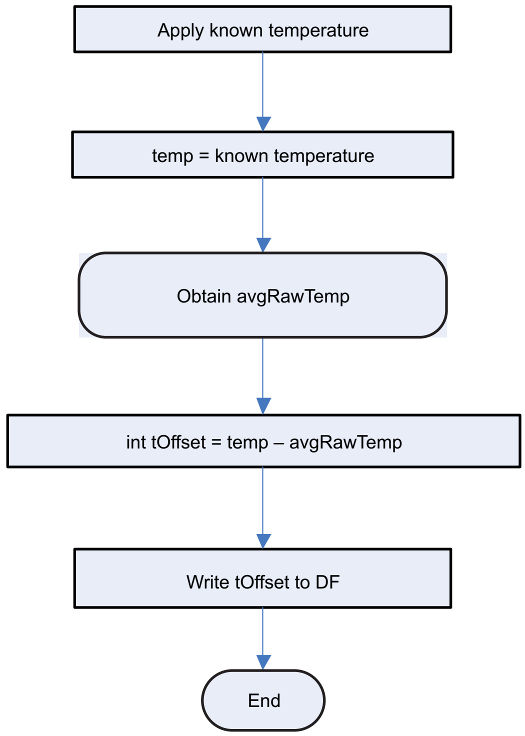



```
void calibrateVoltage(uint16_t knownVoltage) {
    printf("Begin Calibrating Voltage.\n");
    obtainRawCalibrationData(1);
    int vOffset = knownVoltage - avgRawData;
    char offset = (char) vOffset;
    printf("vOffset is %X\n", offset);

    send_extendedCommand(0X3E, 0x00, 0x68);
    read_Block();
    read_Checksum();
    printf("oldChk is %x\n", checksum);
    unsigned char newChecksum = calculate_New_Checksum_1B(13, (char) vOffset);
    printf("newChk is %x\n", newChecksum);

    send_extendedCommand_1B((0x40 + (13 % 32)), offset);
    send_extendedCommand_1B(0x60, newChecksum);
    _delay_cycles(100000);
    send_extendedCommand(0X3E, 0x00, 0x68);
    read_Block();
    printf("Done Calibrating Voltage.\n");
}
```

11 Temperature Calibration

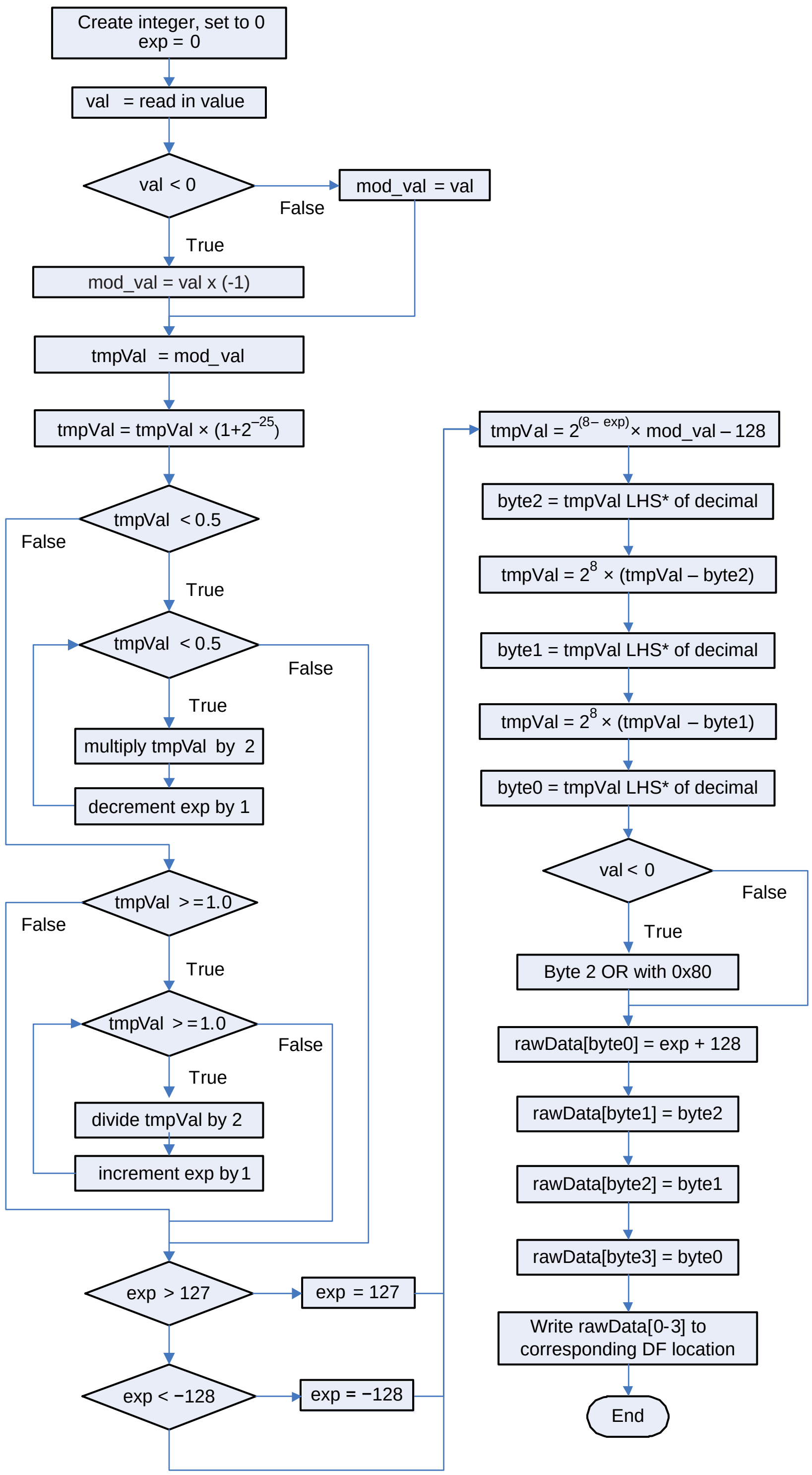
A known temperature must be applied to the device for temperature calibration. The calculated temperature offset must be written to the corresponding location in DF. The temperature offset is represented by an integer that is a single byte in size and can be written to the appropriate location in DF without any intermediate steps. The host system must ensure the fuel gauge is unsealed.

NOTE: The step labeled **Obtain avgRawTemp** refers to the Obtain Raw Calibration Data section.



12 Floating Point Conversion

This section details how to convert the floating point **CC Gain** and **CC Delta** values to the format understood by the gauge. The output should be 4 bytes that need to be written to the corresponding DF location.



* LHS is an abbreviation for Left Hand Side. This refers to truncating the floating point value by removing anything to the right of the decimal point.

```
void floating2Byte(float value) {
    float CC_value = value; //Read CC_gain or CC_delta value from the gauge.
    int exp = 0; //Initialize the exponential for the floating to byte conversion
    float val = CC_value;
    float mod_val;

    if (val < 0) {
        mod_val = val * -1;
    } else {
        mod_val = val;
    }

    float tmpVal = mod_val;
    tmpVal = tmpVal * (1 + pow(2, -25));

    if (tmpVal < 0.5) {
        while (tmpVal < 0.5) {
            tmpVal *= 2;
            exp--;
        }
    } else if (tmpVal < = 1.0) {
        while (tmpVal >= 1.0) {
            tmpVal = tmpVal / 2;
            exp++;
        }
    }

    if (exp > 127) {
        exp = 127;
    } else if (exp < -128) {
        exp = -128;
    }

    tmpVal = pow(2, 8 - exp) * mod_val - 128;
    unsigned char byte2 = floor(tmpVal);

    tmpVal = pow(2, 8) * (tmpVal - (float) byte2);
    unsigned char byte1 = floor(tmpVal);

    tmpVal = pow(2, 8) * (float) (tmpVal - (float) byte1);
    unsigned char byte0 = floor(tmpVal);

    if (val < 0) {
        byte2 = (byte2 | 0x80);
    }
    rawData[0] = exp + 128;
    rawData[1] = byte2;
    rawData[2] = byte1;
    rawData[3] = byte0;

    int i = 0;
    for (; i < 4; i++) {
        printf("rawData[%d] is %x\n", i, rawData[i] & 0xff);
    }
}
```

Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from A Revision (January 2015) to B Revision **Page**