DEMO CIRCUIT DC2100A
SOFTWARE USERS GUIDE

Bi-Directional Cell Balancer Using the LTC3300-1 and the LTC6804-2

DESCRIPTION

This document covers the application of the source code provided for Demonstration Circuit DC2100A. Source code for the firmware programmed into the DC2100A and for the GUI for a Windows PC are provided. The source code can be obtained from http://www.linear.com/solutions/5126. Reference the demo manual for the DC2100A at http://www.linear.com/docs/44975 for a description of the hardware, and the operation of the board with the GUI.

The source code was developed to satisfy 2 requirements:

1. The firmware must communicate with the GUI software to allow evaluation of the LTC3300-1 and LTC6804-2 ICs. Note that the GUI has a special mode when attached to a SuperCap Demo System available to LTC sales personnel, which demonstrates the value of active balancing in a simple low-capacity system.

2. The firmware code must provide a reference design for a battery monitor and balancing system using the LTC3300-1 and the LTC6804-2. The algorithms in the source code provide solutions for the complexities of active balancing such that the user merely needs to interface them to SOC algorithms for their battery chemistry.

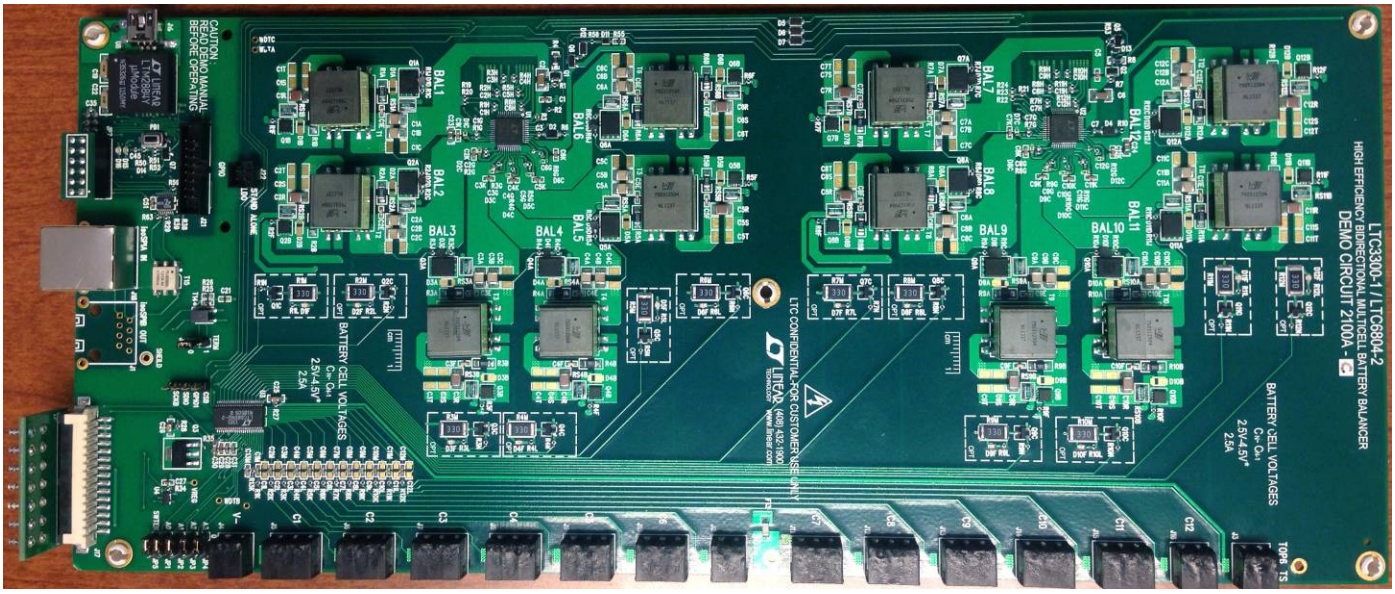


Figure 1 - DC2100A-C PCB

SOFTWARE OVERVIEW

The DC2100A source files can be downloaded from http://www.linear.com/solutions/5126. They consist of 2 collections of code for the PIC18 on the DC2100A-C, and 2 collections of code for a Windows computer (XP/7/8 supported). The interaction between this code is shown in Figure 2.

- Application Firmware (**App FW**), executing in the PIC18F47J53 microcontroller on the DC2100A-C PCB, is written in C for the CCS PCH compiler v5.027 (www.ccsinfo.com) with an MPLab v8.92 project. If the CCS compiler is purchased, the PCWH package must be used to get the RTOS library used by the DC2100A App FW.

- Bootloader Firmware (**Boot FW**), executing in the PIC18F47J53 microcontroller on the DC2100A-C PCB, is written in PIC18 assembly with MPLab v8 project. This code is a modification of the open source PIC18 bootloader from Diolan. (www.diolan.com/pic/bootloader.html).

- GUI Software (**GUI SW**), developed for Windows XP/7/8 computers with v4.5 of the .NET framework, is written in Visual Basic using Microsoft Visual Studio 2010.

- Firmware Update Software (**Update SW**), developed as a console application for Windows XP/7/8 computers, is written in Visual C++ using Microsoft Visual Studio 2010. This code is a modification of the open source update software from Diolan. (www.diolan.com/pic/bootloader.html).

.

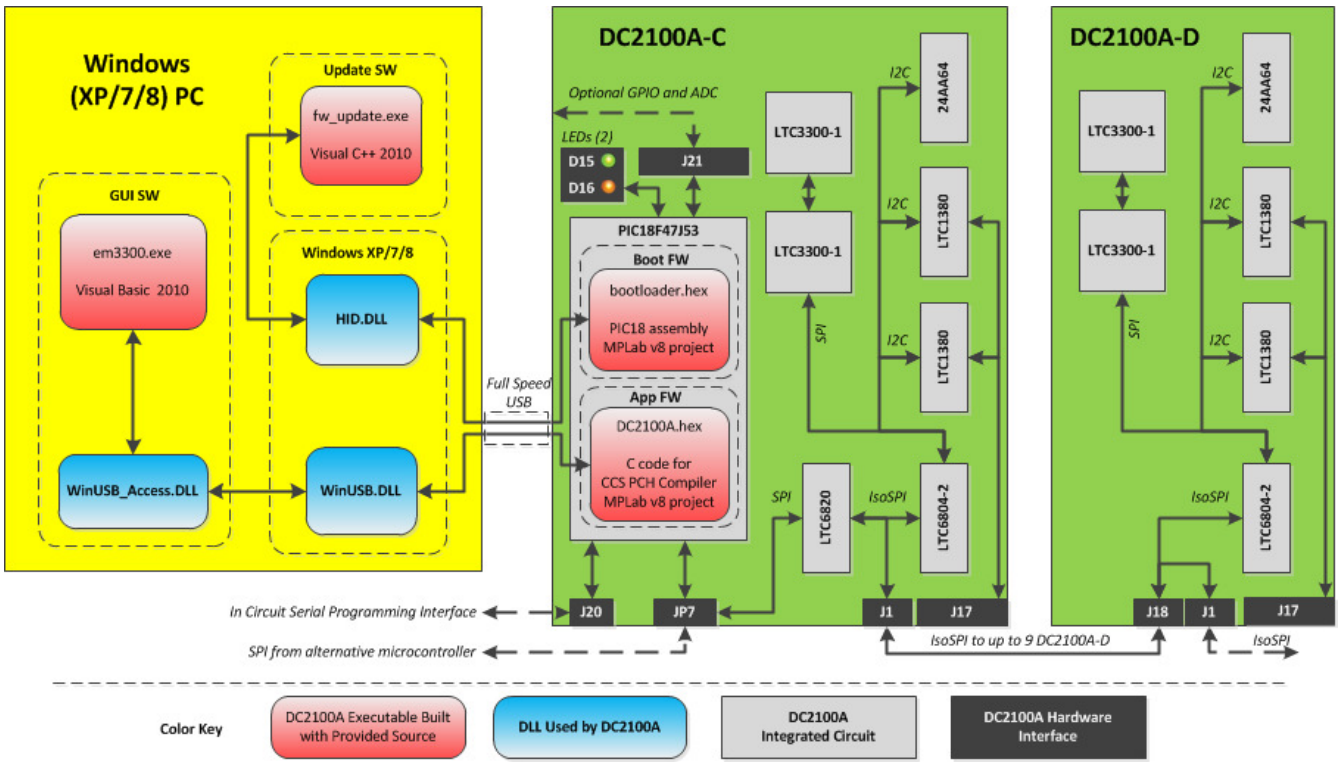


Figure 2 - DC2100A Code Communication Diagram

As shown in Figure 2, the Update SW communicates with the Boot FW through the HID Windows driver where the GUI SW communicates with the App FW through the WinUSB Windows driver.

The Boot FW uses the USB Control Transfer Interface to erase and write new App FW into the PIC18. The Boot FW does not communicate with any of the ICs on the DC2100A. It does use D15 and D16 to indicate the state of the code, however, and does use pins on JP7 to as a method to enter the bootloader as described in the BOOTLOADER FIRMWARE section.

The App FW primarily uses the USB Bulk Transfer Interface to pass commands and responses to the GUI SW. The App FW provides communication to the ICs and I/O located on the DC2100A boards and relays this information back to the GUI SW. The App FW interfaces with all of the ICs on the DC2100A as described here:

- **LTC6820**: converts SPI from PIC18 into isoSPI for the LTC6804-2s on each DC2100A.
- **LTC6804-2**: measures battery voltage and thermistor inputs. This IC also passes I2C and SPI communication to the other ICs.
- **LTC3300-1**: provides active balancing, receiving commands from the PIC18 through the LTC6804-2. The PIC18 sends the SPI commands to these ICs over the SPI commands it sends to the LTC6804-2.
- **LTC1380**: analog multiplexors connect one thermistor input to the LTC6804-2 GPIO input at a time. The PIC18 sends the I2C commands to these ICs over the SPI commands it sends to the LTC6804-2.
- **24AA64**: manufacturing and user data is stored in this 64kbit I2C EEPROM. The PIC18 sends the I2C commands to this IC over the SPI commands it sends to the LTC6804-2.

The App FW uses all of the I/O hardware interfaces on the DC2100A:

- **D15/D16**: LEDs are used to indicate the state of the DC2100A system. The Green LED indicates the FW state, where the Amber LED indicates the USB Communication state.
- **JP7**: SPI signals are passed through this connector to the LTC6820. This also offers a place to disconnect the PIC18 from the rest of the system, so that customers may use an alternative embedded system to control the ICs on the DC2100A.
- **J1/J18**: these connectors pass the isoSPI signals between each DC2100A.
- **J17**: twelve 10kΩ NTC thermistors interface to this connector. These thermistor signals pass through the two LTC1380 multiplexors to be converted by the ADC converter in the LTC6804-2. The DC2100A ships with a PCB attached to J17 (shown in Figure 3) that is populated with resistors ranging from 147Ω to 340kΩ. These simulate 10kΩ thermistors at temperatures ranging from -40°C to 160°C. It is intended for these resistors to be removed and have the turrets connected to actual 10kΩ thermistors in the customer's application.

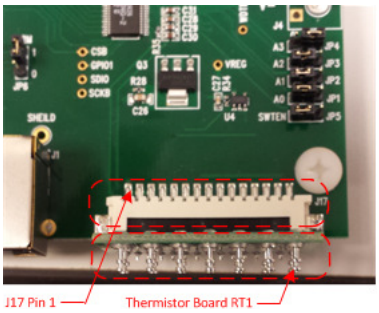


Figure 3 - Thermistor Board Connected to J17

- **J21**: General purpose I/O is available on this port, configurable to be analog, digital, or communication signals. The App FW's usage of these pins is defined in the DC2100A Application Files section as Table 6. Figure 4 shows the physical location of this I/O interface.

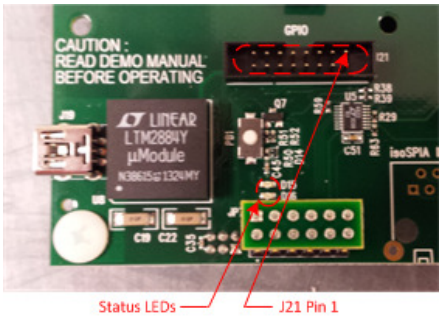


Figure 4 - PIC18 GPIO Via J21

APPLICATION FIRMWARE

BUILDING AND LOADING APP FW

The App FW for the DC2100A is built from the DC2100A.mcp project for **Microchip MPLab v8.92** and **CCS PCH compiler v5.027** or later. The DC2100A.hex file that is created by this project is optimal for loading through the Boot FW and Update SW. See BOOTLOADER FIRMWARE section for details about loading new App FW using the bootloader.

If the user wishes to use MPLab and the In-Circuit Serial Programming interface to load new App FW, this is an option through J20 on the DC2100A-C.

Pin	Function
1	N/C
2	PGC
3	PGD
4	/MCLR
5	3.3V VCC
6	GND

Table 1 - J20 Interface for ICSP

Care must be taken to not erase the memory regions defined in Table 2 to avoid overwriting the bootloader FW and manufacturing data in the DC2100A. See BOOTLOADER FIRMWARE section for details about hex image placement in the PIC18 program memory.

Memory Addresses	Memory Contents
0x00-0x7FF	Boot FW
0x1F800-0x1FFFF	Mfg Data

Table 2 – Memory Addresses to Preserve in PIC18

ALTERNATIVE MICROCONTROLLER CONNECTION

If the user wishes to use their own microcontroller to monitor and control the DC2100A hardware, JP7 offers an option to disconnect the PIC18 from the system and connect an alternative microcontroller. To do this, remove the

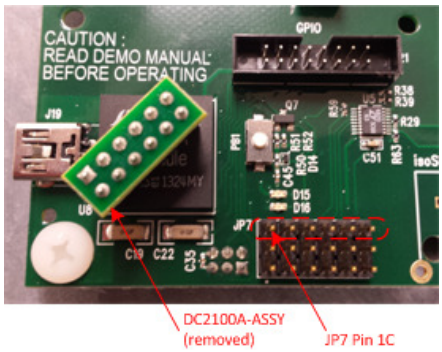


Figure 5 - JP7 Configuration for Alternative Micro

DC2100A-ASSY PCB and connect to the pins as shown in Figure 5:

Pin	Function
1C	LTC6820 ENABLE
2C	SPI MOSI
3C	SPI MISO
4C	SPI SCK
5C	SPI CS
6C	GND

Table 3 - JP7 Interface for Alternative Micro

The core components of the App FW were designed to be portable to other microcontrollers preferred by the user. They are distinguished in Figure 6 as the Reference Design Files.

App FW Architecture

Figure 6 shows the architecture of the files that comprise the DC2100A App FW. While all of the files are required to make the DC2100A function with the GUI and SuperCap Demo Systems, the outlined files form the foundation of a reference project for a battery monitor and balancing system using the LTC6804-2 and LTC3300-1. The files are collected into four groups:

- **PIC18 Driver Files** provide access to the PIC18 hardware necessary to support the other code modules. A customer using the DC2100A code in an alternative microprocessor would need to replace these files with drivers specific to their system.

- **LTC Driver Files** provide access to the services of LTC parts. They are intended to be independent of the DC2100A and the PIC18 hardware. They provide an API to the LTC6804-2, LTC3300-1, and LTC1380 ICs.

- **Reference Application Files** are examples of how the LTC6804-2/LTC3300-1 would be used to monitor and balance a customer's battery system. They are intended to be independent of the DC2100A and the PIC18 hardware. They provide Voltage monitoring, Temperature monitoring, Balancer control, and EEPROM data storage through the LTC6804-2/LTC3300-1.

- **DC2100A Application Files** are specific to the DC2100A demo board system. They satisfy the requirement to demonstrate the demo circuit functionality through the GUI SW and to operate the SuperCap Demo Systems. A customer will want to replace these files with their own scheduling, State of Charge, and communication code to interface to the reference design files.

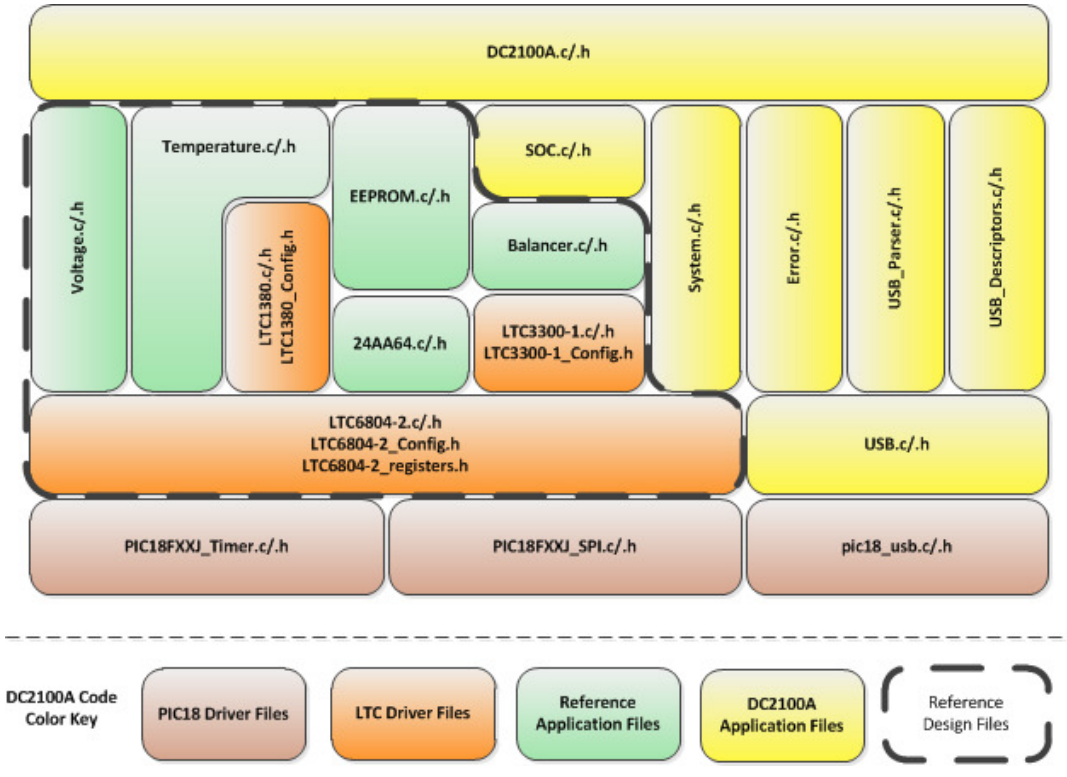


Figure 6 - App FW Architecture

PIC18 DRIVER FILES

These files are specific to the PIC18 hardware, and serve as documentation of what microprocessor resources are necessary to use the DC2100A reference code.

PIC18FXXJ_Timer.c/.h: This code uses Timer0 of the PIC18 to provide time measurements and time stamps to the other code modules:

- LTC6804-2.c uses timestamps to determine when the part has entered its STANDBY and SLEEP states, so that communication to the ICs can be preceded by the appropriate wakeup signaling.
- Voltage.c and Temperature.c attach timestamps to each group of ADC samples, so that the algorithms consuming this data can process it in real time.

PIC18FXXJ_SPI.c/.h: This code provides an interface to the SPI port on the PIC18 used to communicate with the LTC6804-2. DMA is used so that other tasks, such as CRC calculation, can be performed as bytes as sent to and received from the SPI bus. The SPI driver can be configured for different baud rates, so that communication with each IC can occur at its maximum rate:

- The PIC18 only communicates directly to the LTC6804-2 Driver. Due to the amount of processing required to calculate the PEC for the LTC6804-2, the SPI driver is buffered so that these calculations are performed as bytes are being sent and received. The other ICs that have communication passed through the LTC6804-2 have lower baud rate requirements, which are achieved by lowering the baud rate only for the portion of communication where the bytes are passed through the LTC6804-2.

pic18_usb.c/.h: This code is supplied by CCS with their PCH compiler. It is a USB driver, specific to the PIC18 hardware.

LTC DRIVER FILES

These files provide access to the services of LTC parts. They are intended to be independent of the DC2100A and the PIC18 hardware. The API to these drivers are detailed in the doxygen documentation included with the source code.

All of the LTC Driver Files are architected to allow their reuse with minimal modification. Figure 7 shows the relationship between the LTC Driver code and the other code in a system.

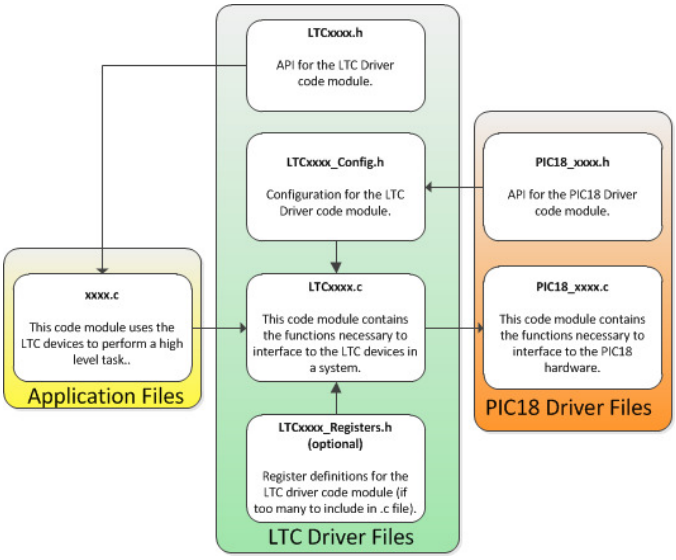


Figure 7 - LTC Driver Code Module Architecture

The LTCxxxx.c and LTCxxxx_Registers.h files contain all of the details from the LTC datasheet that are necessary to use the device, but do not need to be exposed to the Application code wishing to use the driver. Register names, bit positions of data in the registers, and data integrity checks are examples of datasheet details that would be found in these files. It is not intended for these files to require modification, in order for the LTC Driver code to be reused.

The LTCxxxx_Config.h file contains macros that configure the LTC Driver to the particular system in which it's being used. It serves as documentation of what resources are necessary to use an LTC Driver file. HW driver functions and system Timer variables are examples of configuration items that would be located in this file. Each

LTCxxxx_Config.h is configured for the DC2100A, but examples are given in the comments for how the configuration options would be changed for other systems.

The LTCxxxx.h file contains the API to the LTC Driver. It provides the details necessary for Application code to use the driver at a functional level. While some datasheet details may be in the file, they are the details necessary for Application code to use the driver. For example, the different sampling modes for the LTC6804-2 would be in this file as different Application code modules may need to sample at different rates. The register bits that need to be modified to achieve these sample rates, however, would not be in this API file.

LTC6804-2.c/.h: This code module implements a driver for the LTC6804-2 Multicell Battery Monitor. The driver performs the following operations to interface to the LTC6804-2:

- **State Management:** The LTC6804-2 enters different states in order to minimize its power consumption. The ability to perform certain functions and maintain certain data depends upon this state, and must be managed by FW. Page 20 of the LTC6804 datasheet contains the documentation for these states, with a state diagram in Figure 1 of that document.
- **Command Communication:** The LTC6804-2 has many commands used to operate the part. These commands are listed in Table 34 of the LTC6804 datasheet. Many of these commands are used to send data to and receive data from 6 byte register groups. These register groups are documented in Table 36 through Table 45 of the LTC6804 datasheet.
- **Clocking the GPIO Implemented I2C/SPI Bus:** One of the LTC6804-2 commands causes I/O ports GPIO3, GPIO4 and GPIO5 on the LTC6804-2 to act as an I2C or SPI master. Table 19 and Table 20 in the LTC6804 datasheet contain the relationship between the input IsoSPI clock for the LTC6804-2 and the output clock on the GPIO Implemented I2C/SPI bus.

Two of the more important functions using the LTC6804-2 are detailed in "**Appendix A - Voltage Monitoring using LTC6804-2**" and "**Appendix B - LTC6804-2 I²C/SPI Master Using GPIOS**" sections of this document.

LTC3300-1.c/.h: This code module implements a driver for the LTC3300-1 High Efficiency Bidirectional Multicell Battery Balancer. The driver performs the following operations to interface to the LTC3300-1:

- **Watchdog Timer Management:** The LTC3300-1 provides a means of shutting down all active balancing in the event that communication is lost. The LTC3300-1 driver provides functions to keep the IC from shutting down, when no other communication is needed.
- **Balance Command Execution:** The primary function of the LTC3300-1 is to move charge between cell groups for the purpose of achieving balance. The LTC3300-1 driver provides functions to write, execute, and suspend balance commands.
- **Balancer Status:** The LTC3300-1 status register contains information about whether the part is actively balancing, or if the part is damaged.

In the DC2100A code, the communication to the LTC3300-1 is performed through the GPIO Implemented SPI Bus of the LTC6804-2. Therefore, an LTC6804-2 identifier is input to each of the functions in this code module. The "**Appendix B - LTC6804-2 I²C/SPI Master Using GPIOS**" section of this document contains an execution diagram for a write balance command sent to the LTC3300-1 through the LTC6804-2.

LTC1380.c/.h: This code module implements a driver for the LTC1380 Single-Ended 8-Channel/Differential 4-Channel Analog Multiplexer with SMBus Interface. The driver performs the following operations to interface to the LTC1380:

- **Enable MUX Channel:** The driver provides a means of turning on one channel on the MUX.
- **Disable MUX:** The driver provides a means of turning off all channels on the MUX.

In the DC2100A code, the communication to the LTC1380-1 is performed through the GPIO Implemented I2C Bus of the LTC6804-2. Therefore, an LTC6804-2 identifier is input to each of the functions in this code module.

REFERENCE APPLICATION FILES

These files are examples of how the LTC6804-2 and LTC3300-1 could be used to monitor and balance a customer's battery system. They are intended to be independent of the DC2100A and the PIC18 hardware. They provide Voltage monitoring, Temperature monitoring, Balancer control, and EEPROM Data storage through the LTC6804-2. The API to this application code is detailed in the doxygen documentation included with the source code.

Voltage.c/.h: This code contains the Voltage Monitor Task to read the voltages in the DC2100A System at a rate set by VOLTAGE_TASK_RATE. The code module performs the following operations to monitor the voltage:

- Communication is performed with each LTC6804 in the system to measure the cell voltages.
- If the cell voltage measurements are successful, the sum-of-cells voltages are calculated.
- A timestamp is attached to each set of voltages, as well as whether any balancers were turned on. This allows algorithms consuming this information to process it appropriately.
- Each cell is monitored for UV and OV conditions. If either condition is present, then balancing operations are immediately suspended.

Temperature.c/.h: This code contains a task to read the temperature sensors in the DC2100A System at a rate set by TEMPERATURE_TASK_RATE. The code module performs the following operations to monitor the temperature:

- Communication is performed with each LTC6804 in the system to measure the one thermistor ADC value connected to GPIO2.
- If the temperature measurement is successful, the thermistor ADC value is converted to temperature by linearly interpolating from the Temperature_Table[] lookup table pictured in Figure 8. This table assumes the resistors are of the 10kΩ Curve Type 1 from Vishay.
- The raw ADC values are stored for one board selected by the GUI.
- The LTC1830 MUXes are controlled to connect the next thermistor to the LTC6804 to be read.
- Note that the large time constant that results from switching thermistors is not waited for in the task. The rate at which this task is called is expected to be longer than the settling time for the circuit.

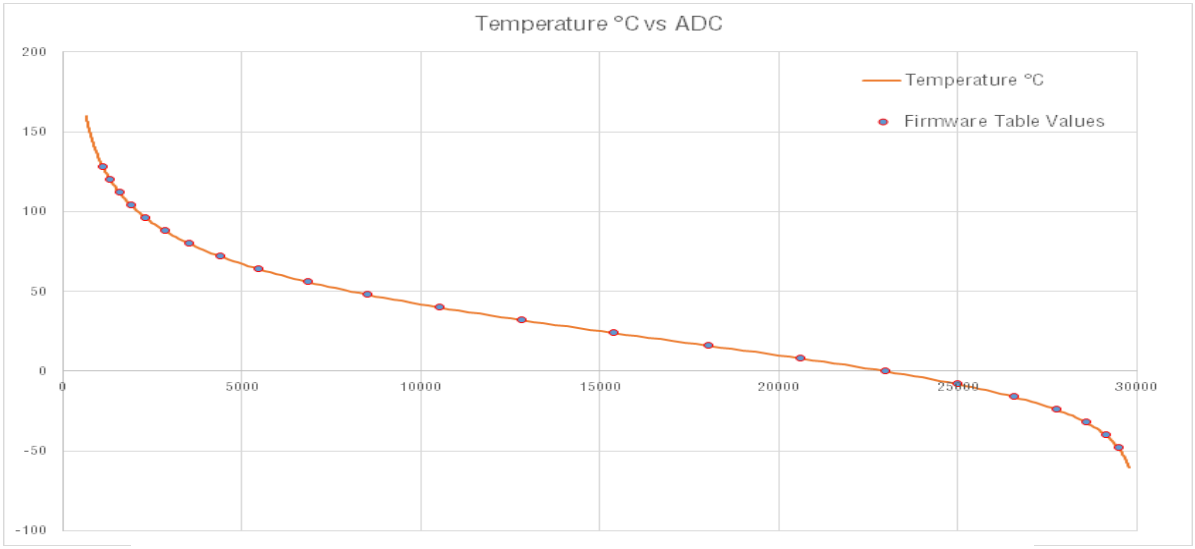


Figure 8 - Relationship between °C and ADC values stored in Temperature_Table[]

Balancer.c/.h: This code contains a task to control the balancers in the DC2100A System with a resolution set by BALANCER_TASK_RATE. The code module performs the following operations to control the balancers:

- Communication is performed with each LTC3300-1 in the system to control the active balancers, depending upon their desired state.
- Communication is performed with each LTC3300-1 to monitor their status.
- Passive balancers connected to the S-pins of the LTC6804-2 are turned on or off.

The states with which the Balancer Control Task controls the balancers is shown in Code Sample 1.

Typical usage will be for the Balancer Control Task to be in the OFF state until a higher level function determines balancing is needed. The task will then be placed in the SETUP state as the balancing times are calculated and the LTC3300s are configured for the desired balance commands. The task is then placed in the ON state in which the LTC3300s will execute the desired commands as the task counts the time for each cell's active balancing. When all of the cell balancing is complete, the task will return to the OFF state.

```
// Definition of Balancer Task States.
typedef enum
{
  BALANCER_CONTROL_OFF,      // Balancing is not active. ICs are allowed to go to sleep.
  BALANCER_CONTROL_GUI,      // Watchdog is being activated to prevent LTC3300s from going to sleep,
                             // but raw commands are being sent to ICs from the GUI.
  BALANCER_CONTROL_SETUP,    // Balancing commands are being loaded. ICs are prevented from going to sleep.
  BALANCER_CONTROL_ON,       // Balancing commands are being executed for a timed value for each cell.
                             // ICs are prevented from going to sleep.  State automatically transitions to
                             // OFF when balancing is complete.
  BALANCER_CONTROL_SUSPEND   // Balancing commands are suspended.  ICs are prevented from going to sleep.
} BALANCER_CONTROL_STATE_TYPE;

// State in which the balancer task is currently operating.
BALANCER_CONTROL_STATE_TYPE balancer_control_state;
```

Code Sample 1 - Balancer Control States

The most sophisticated Balancer Control state is the ON state, in which the task will turn on each active balancer for a period of time. The amount of time and direction for each balancer to be controlled is contained in the variable detailed in Code Sample 2.

```
// The state of each active cell balancer on each DC2100A in the system.
BALANCER_ACTIVE_STATE_TYPE Balancer_Active_State[DC2100A_MAX_BOARDS][DC2100A_NUM_CELLS];;
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COMMAND_BIT: 0 = Charge 1 = Discharge	TIME_BITS: 250ms resolution, set by BALANCER_TASK_RATE														

Code Sample 2 - Balancer Active State Variable

The functions available to set this variable are shown in Code Sample 3.

```
// Places Balancer Control Task in the BALANCER_CONTROL_SETUP state.
// Loads the desired active cell balancer states for DC2100A_NUM_CELLS cells on one board.
// return void
void Balancer_Set(
  int8 board_num,                          // The logical address for the PCB containing this Balancer.
  BALANCER_ACTIVE_STATE_TYPE* cell_state_ptr // Pointer to the desired active cell balancer states for
);                                         // DC2100A_NUM_CELLS cells.

// Places Balancer Control Task in the BALANCER_CONTROL_SETUP state.
// Calculates and loads the optimal active cell balancer states to achieve the desired amount of
// charge to move for each cell.
// Note - this function is currently only implemented for a single DC2100A board.
// return void
void Balancer_Set(
  BALANCER_DELTA_Q_TYPE* charge_target_ptr  // Pointer to the desired charge to move for
);                                         // DC2100A_NUM_CELLS cells in mAs.
```

Code Sample 3 - Functions to Set Balancer Times in ON State

The first Balancer_Set() function simply loads the variable with the times and directions specified. This is the function that is used by the standard GUI, to perform a timed balancing function via USB. The second Balancer_Set() function would be used by a reference design in which a State of Charge function determines the amount of charge imbalance in a battery, and wishes to load the optimal set of balance times for the Balancer Task to move that appropriate amount of charge. This function is used by the SuperCap Demo System version of the GUI. The details of the calculation are described in "**Appendix C - Setting Balancer Times from Charge Imbalance**."

Table 4 shows the level of control available from the Balancer Control Module when a 250ms task rate is used.

Balancer Control Characteristic	A (2A)	C (4A)	Unit
Min balance time	0.25	0.25	sec
Max balance time	2.28	2.28	hrs
Max charge moved in min balance time.	0.65	1.08	As
Min charge moved in max balance time.	5.01	7.74	Ah

Table 4 - Balancer Control with a 250ms Task Rate

EEPROM.c/.h: This code provides access to reading, writing, and setting defaults for EEPROM data stored on each DC2100A PCB. The EEPROM Code Module is not implemented as a task, but as a series of functions that are called by the other Application code.

The items that are stored in the EEPROM are shown in Table 5. When creating custom code for the DC2100A, there is much value in preserving these values as some are calibrated in the Linear factory.

Data	Description
MFG_DATA	Manufacturing Board ID Data (Model, Serial Number, SuperCap Demo Configuration).
MFG_CAP	Linear factory calibrated capacity values (used by SuperCap Demo only).
MFG_CURRENT	Linear factory calibrated balance current values.
USER_CAP	User entered capacity values.
USER_CURRENT	User entered balance current values.

Table 5 - DC2100A EEPROM Data

When values are retrieved from the EEPROM, copies are made in RAM due to the communication time required to retrieve the data. Some items have two copies in the EEPROM: The Manufacturing value (written in the Linear factory when the board is manufactured) and the User value (available for general use). This allows the factory values to be maintained, while allowing the user to enter new values if the board configuration is changed.

Each piece of data is stored with a CRC so that the data can be checked for consistency when it is retrieved from the EEPROM. The EEPROM code module loads the values in this order:

- If the User values have a valid CRC, they are loaded into RAM.
- If the User values do not have a valid CRC, but the Manufacturing values have a valid CRC, the Manufacturing values are loaded into RAM.
- If neither the User nor Manufacturing values have a valid CRC, then the nominal values from Flash are loaded into RAM.

The balance currents are used by the Balancer Code Module when calculating the optimal balance times required to move a desired amount of charge. The capacities are used by the SuperCap Demo System, or a custom State-of-Charge algorithm developed by the user.

24AA64.c/.h: This code provides an interface to the 24AA64 64 Kbit Electrically Erasable PROM located on each DC2100A PCB. It would typically be considered a driver, although it was not documented as such for the DC2100A as it is not an LTC chip.

The EEPROM Code Module writes and reads its non-volatile data through the 24AA64 code module using the functions shown in Code Sample 1.

```
// Writes a series of bytes to the 24AA64 EEPROM
// return void
void Eeprom_24AA64_Write(
   int8 board_num,     // The logical address for the PCB containing this EEPROM.
   int16 address,      // The address in the EEPROM.
   int8* data_ptr,     // Pointer to the data to write.
   int16 num_bytes     // The number of bytes to write.
);

// Reads a series of bytes to the 24AA64 EEPROM
// return void
void Eeprom_24AA64_Read(
   int8 board_num,     // The logical address for the PCB containing this EEPROM.
   int16 address,      // The address in the EEPROM.
   int8* data_ptr,     // Pointer where to store the read data.
   int16 num_bytes     // The number of bytes to read.
);

// Erases the full contents of the 24AA64 EEPROM
// return void
void Eeprom_24AA64_Erase(
   int8 board_num      // The logical address for the PCB containing this EEPROM.
);
```

Code Sample 1 - Functions to Interface to 24AA64 EEPROM

Note that these functions are blocking, so a lot of execution time will be used if long streams of data are sent to or retrieved from the EEPROM at one time.

DC2100A APPLICATION FILES

These files are specific to the DC2100A when used as an evaluation board through the GUI.

DC2100A.c/.h: The DC2100A hardware definition is contained in these files. It is intended to contain the details of the DC2100A schematic necessary for App FW code to operate. The DC2100A has GPIO available on J21 for use by the App FW. The usage of this GPIO by the App FW is shown in Table 6.

The App FW identification, App FW configuration, and Code scheduling is also located in these files. This code uses the RTOS included with the CCS compiler to schedule the App FW tasks as per Table 7. The main() function initializes the tasks and the hardware, before starting the RTOS which executes them according to their desired periodic rate. Figure 9 shows the execution of these tasks, by using the GPIO available on J21.

Note that much of the USB Parser is located in DC2100A, as the CCS compiler requires code to be located in the main() file in order to use the rtos_await() function. It is also a convention of the CCS compiler to have one file included in a project, with the called code modules included as .c files.

The only interrupt operating in the system is the USB Interrupt, managed by the CCS USB driver.

Pin	Schematic Name	FW Name	Function
1	GND	N/A	GND
2	VCC_3V3	N/A	3.3V
3	AN1	DISCHARGER_OUT_PIN	Output to control discharger in SuperCap Demo System*
4	/AN0	!CHARGER_OUT_PIN	Buffered output to control charger in SuperCap Demo System*
5	AN3	DEBUG0_OUT_PIN	Toggled in Task_Status() at same rate as Green LED D15
6	AN0	CHARGER_OUT_PIN	Output to control charger in SuperCap Demo System*
7	AN5	DEBUG1_OUT_PIN	Lowered in Task_Parser() as USB Communication is Processed
8	AN2	DEBUG2_OUT_PIN	Lowered in Task_Voltage() when voltage sampling is started.
9	AN7	DEBUG3_OUT_PIN	Lowered in Task_Temperature() when temperature sampling is started.
10	AN6	DEBUG_AIN_PIN	Monitors one A/D input referenced between VDD and VSS (future implementation).
11	SDO2	DEBUG4_OUT_PIN	Lowered in Task_Balancer() when balancer state machine is started.
12	SS2	DEBUG5_OUT_PIN	Lowered in Task_Error() when error detection is started.
13	SCL2	DISCHARGER_IN_PIN	Input to indicate discharge mode (future implementation).
14	SDI2	CHARGER_IN_PIN	Input to indicate charge mode (future implementation).
15	GND	N/A	GND*
16	VCC_5V	N/A	5V

*Pins used by SuperCap Demo System

Table 6 - J21 GPIO Configuration

Task Name	Period (ms)	Period #define	Task Function
Task_Status()	100	STATUS_TASK_RATE	Indicates the status of the DC2100A system
Task_Get_USB()	50	USB_TASK_RATE	Receives bytes from the USB bulk endpoint. Must be its own task to receive bytes while Task_Parser is pending.
Task_Parser()	50	USB_TASK_RATE	Parses bytes received from the USB bulk endpoints for commands, and sends the responses to the commands
Task_Voltage()	100	VOLTAGE_TASK_RATE	Monitors the voltages in the DC2100A system
Task_Balancer()	250	BALANCER_TASK_RATE	Controls the balancers in the DC2100A system
Task_Temperature()	600	TEMPERATURE_TASK_RATE	Monitors the temperatures in the DC2100A system
Task_Error()	1000	ERROR_TASK_RATE	Detect errors in the DC2100A system
Task_Detect()	1000	DETECT_TASK_RATE	Detect new boards in the DC2100A system

Table 7 - DC2100A RTOS Tasks

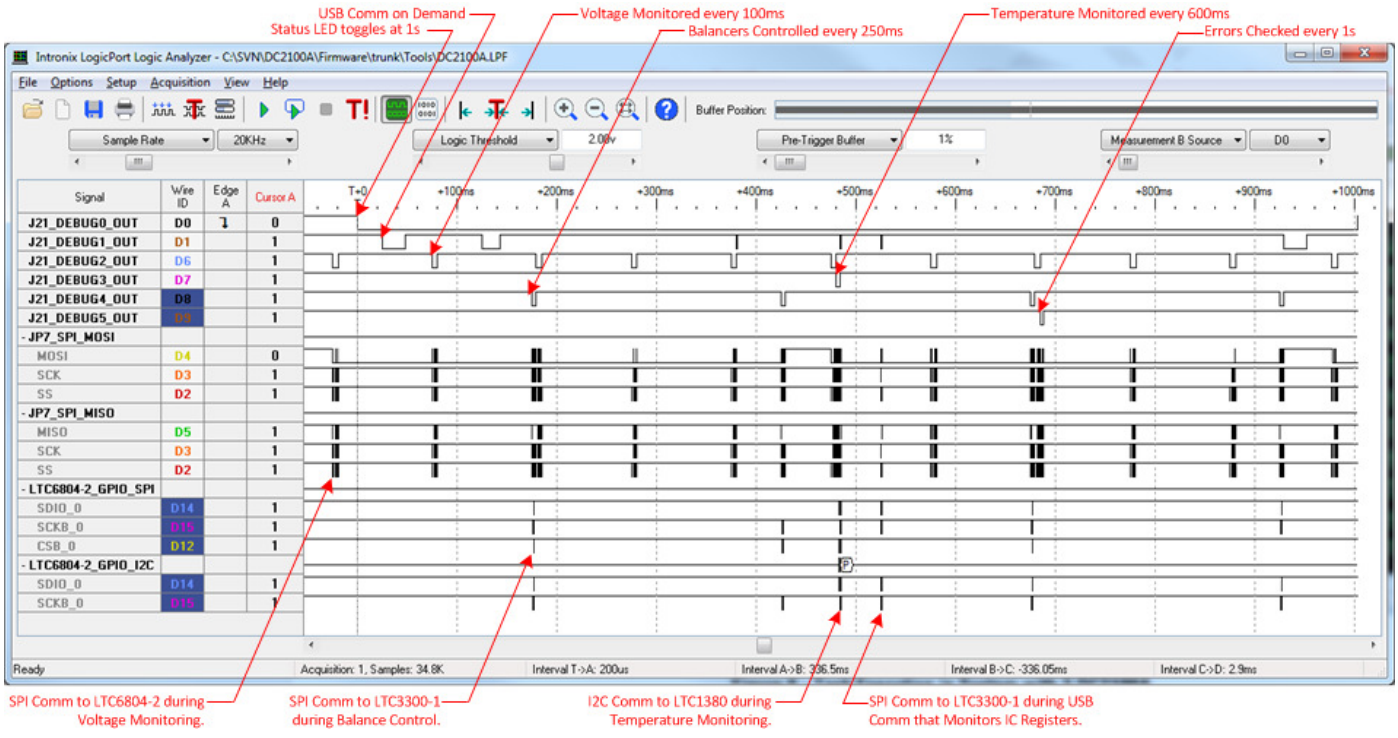


Figure 9 - Task Execution in System with 3 DC2100A

System.c/.h: This code detects and manages the state of the DC2100A system and indicates it through the Green LED D15. The states of the system are listed in Table 8. Most of the tasks in the DC2100A system will only begin to operate once the system has reached the AWAKE state.

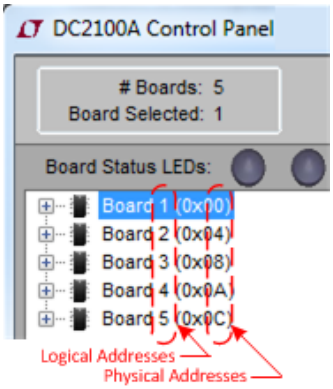


Figure 10 - Logical and Physical Addresses in 5 Board System

A DC2100A system can contain up to 8 boards, and these boards can have 16 different physical addresses set by JP1-JP4. For example, Figure 10 shows a 5 board system with physical addresses 0x00, 0x04, 0x08, 0x0A, and 0x0C. Note that the limitation of 8 boards is due to the 50Volt rating of transformer T15.

The number of boards and the mapping between the logical and physical addresses are maintained by this code module and made available to the rest of the FW and SW. Note that the DC2100A containing the PIC must have a special address as defined by DC2100A_PIC_BOARD_NUM, where the remaining physical addresses are free for the DC2100A-D boards connected to the system.

As boards are detected they are initialized and their manufacturing data is retrieved. If the boards cease communication, they are assumed to still be present but in a state where the 6804-2 does not have sufficient voltage for communication. This code module detects this condition and reinitializes the boards as they regain communication with the PIC18.

A customer using the DC2100A code as part of a reference design, would likely replace the functionality in these files with a fixed system configuration. The need to detect and configure an arbitrary number of boards at an arbitrary number of addresses is only needed when the code is used as part of a generic demonstration circuit.

System State	Description	D15 Indication
OFF	The PIC is not powered (HW state)	Off
PIC_BOARD_INIT	The PIC is powered and is being initialized. The PIC must establish comm with the LTC6804-2 on its PCB to leave this state.	Quickly Flashing (toggles every 200ms)
INIT	The PIC has initialized the LTC6804-2 on its PCB, and is now searching for attached DC2100A-D PCBs.	Quickly Flashing (toggles every 200ms)
AWAKE	The PIC has monitoring and controlling all of the DC2100A in the system.	Slowly Flashing (toggles every 1s)
SLEEP	The PIC is allowing the DC2100A system to enter a low power mode (future implementation).	Very Slowly Flashing (toggles every 10s)

Table 8 - DC2100A States

USB.c/.h: This code is supplied by CCS with their PCH compiler. It is a USB driver, abstracted away from the PIC18 hardware. The CCS driver was modified to support descriptors in RAM. Note that this USB driver operates in an interrupt service routine.

USB_Descriptors.c/.h: This code contains the USB descriptors used by the DC2100A to identify itself to the connected PC. This code is used with a modified version of the CCS USB driver as the original driver did not support strings in RAM.

USB_Parser.c/.h: This code contains the commands used to receive commands from and send responses to the DC2100A GUI.

The current implementation is minimally documented here. A future implementation of this code module will address throughput issues when many boards are in the system, as well as ensuring data consistency through atomically designed messages. This future implementation will be fully documented to allow customers to create their own software to interface to the DC2100A.

Error.c/.h: This code collects information about errors in the DC2100A system so that they can be recorded in the GUI Software's Event Log. It is very minimal, providing only an error code and 12 bytes specific to the error. It was primarily intended to expose communication issues with the ICs on the DC2100A through the GUI.

Each type of error is counted, but the full data is only collected for one error per ERROR_TASK_RATE ms. This prevents this code module from hijacking all of the USB bandwidth when a fault occurs.

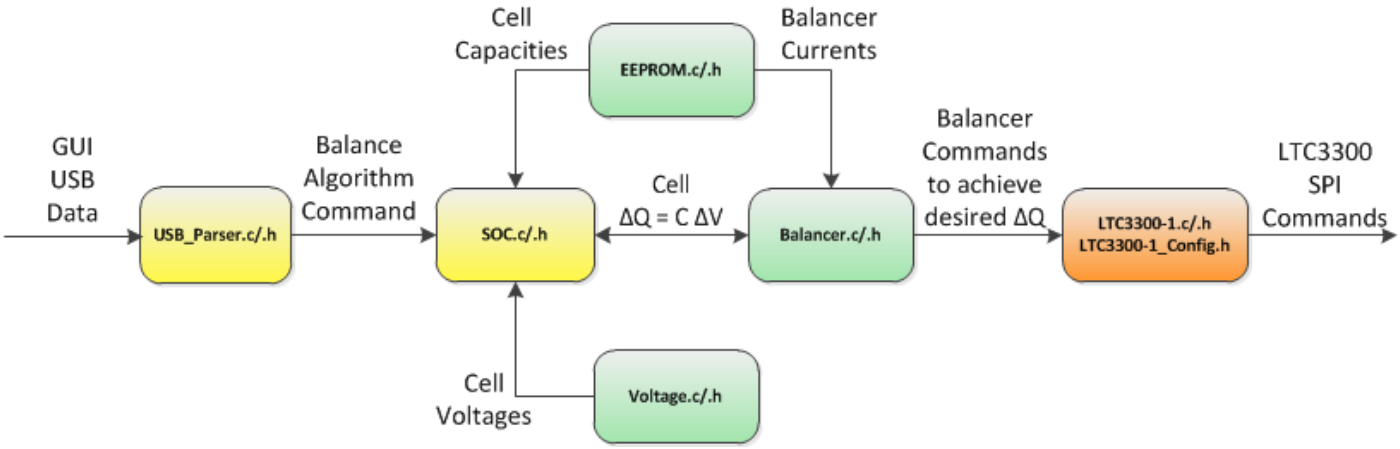


Figure 11 - SuperCap Demo Balancing Code Flow

SOC.c/.h: This code calculates the charge imbalance in the SuperCap Demo system used by the Linear Sales team. It is very simple, using the relationship between charge and voltage on a capacitor: $\Delta Q = C\,\Delta V$. The charge imbalance is then passed to the Balancer_Set() function described in "**Appendix C - Setting Balancer Times from Charge**." to calculate the optimal time for each balancer to be turned on to balance the SuperCap Demo system. Figure 11 shows the data flow as the SuperCap Demo GUI orders the SOC calculation to result in balancing activity.

When the DC2100A code is used as a reference application for the customer, the SOC calculation would contain the state of charge algorithm for the user's battery chemistry. It would likely run as its own task to estimate the ΔQ of the battery cells from voltage, current, temperature, and time information from the other code modules. When the ΔQ imbalance is unacceptable, the task would instruct the Balance Control Task to move charge for correction of the imbalance. The EEPROM has nonvolatile memory reserved for the customer's cell capacities, as the state of charge must be calculated as a percentage of the total cell capacity.

GUI SOFTWARE

The GUI SW displays data from and sends commands to the App FW. Although the source code is provided, only a high level description is given in this document.

The GUI contains 5 windows, where a different window is displayed if the DC2100A is configured as part of a SuperCap Demo System. The navigation between these windows is shown in Figure 12. The windows are briefly described here:

- **Control Panel**: this is the main form for the GUI, displaying real time data for the system as well as control of the LTC3300-1 balancers. It is also used to launch the other windows.
- **Calibration Data**: displays the User data stored in EEPROM, and allows it to be reset back to the Manufacturing values.
- **Event Log**: displays events and errors that occur in the App FW or the GUI SW.
- **Graph View/SuperCap Demo**: displays a graph of signals available in the DC2100A. The SuperCap Demo window will display if the DC2100A is part of a SuperCap Demo System.
- **Graph View Options**: allows configuration of the Graph View and SuperCap Demo windows.

See the DC2100A Demo Manual and the DC2100A SuperCap Demo Manual for documentation about the functionality in these windows.

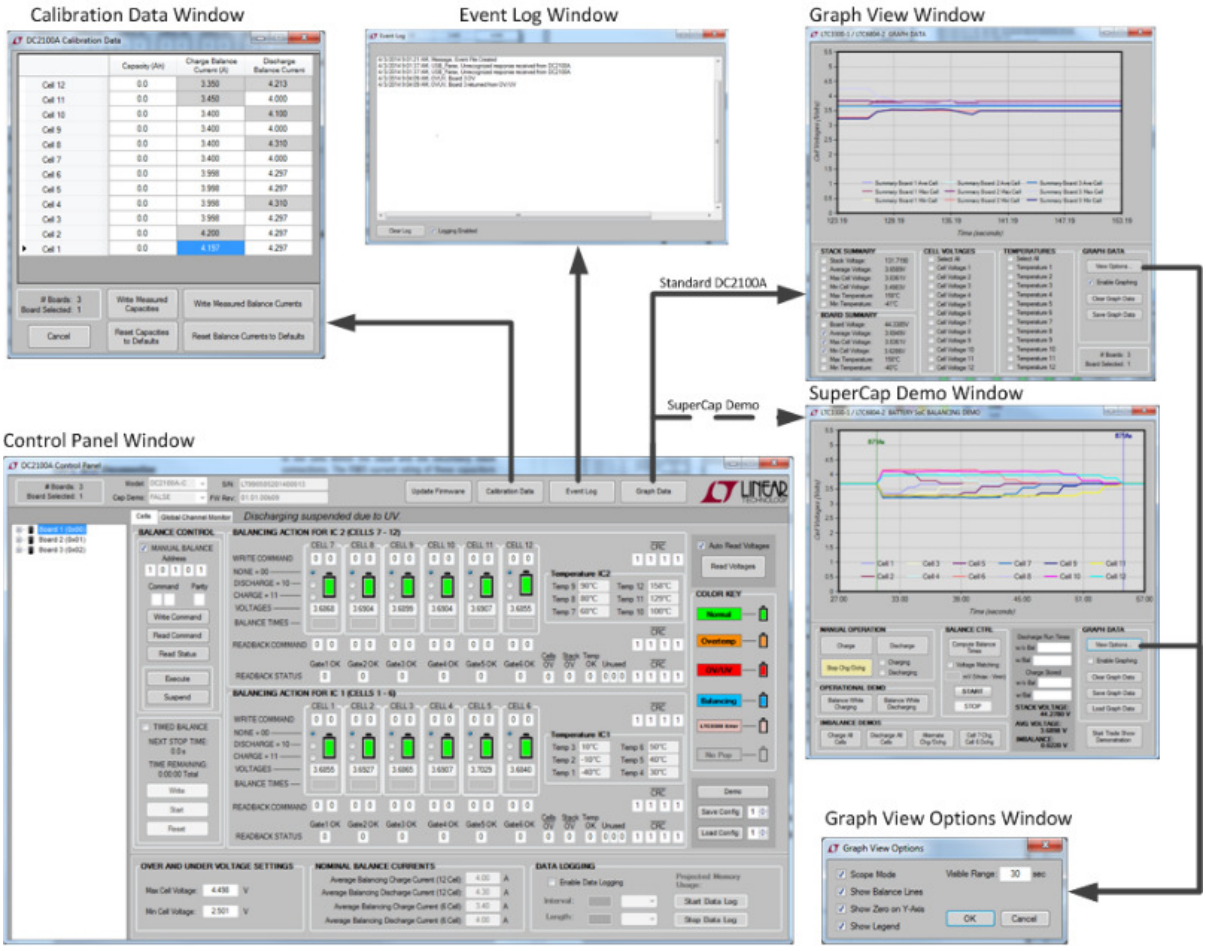


Figure 12 - DC2100A GUI Window Navigation Diagram

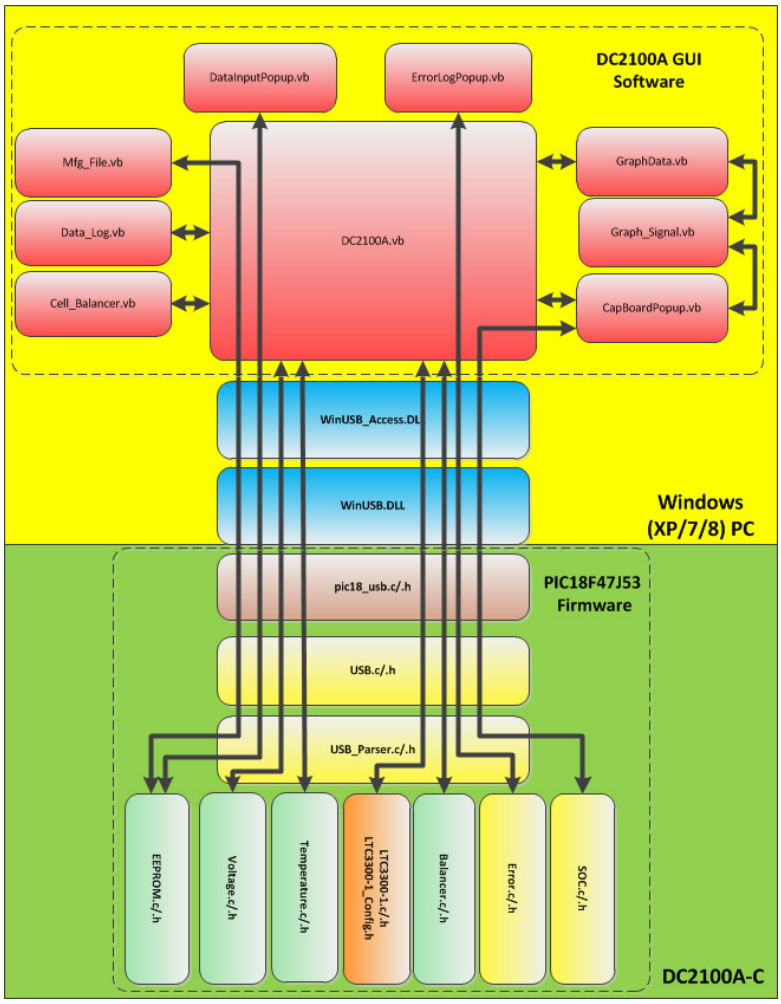


Figure 13 - GUI SW Architecture

The DC2100A GUI Software files are pictured in Figure 13, as well as the data path between the GUI and the Application FW. The files are briefly described here:

- **DC2100A.vb** – contains the control panel window and all of the USB communication between to the FW. The voltages, temperatures, LTC3300 registers, and balancing times are all regularly polled and recorded by this file.
- **Cell_Balancer.vb** - implements a class for each active cell balancer and each LTC3300 IC.
- **Data_Log.vb** – retrieves data from DC2100A.vb at a periodic rate, to save to a CSV file.
- **Mfg_File.vb** –used by Linear Technology in the factory, to configure boards as they are manufactured.
- **DataInputPopup.vb** – contains the Calibration Data Window, to provide a method of displaying and modifying the Capacity and Balance current data stored in the DC2100A.
- **ErrorLogPopup.vb** – contains the Event Log Window, to display events and errors that are reported by the GUI and App FW.
- **GraphData.vb** – contains the Graph View Window to display data retrieved from DC2100A.vb real time.
- **CapBoardPopup.vb** – Contains the SuperCap Demo Window, to send charge, discharge, and balance to a SuperCap Demo System while displaying the cell voltages real time.
- **Graph_Signal.vb** – implements a class for groups of signals displayed on a graph.

BOOTLOADER FIRMWARE

The Boot FW functions with the Update SW to load new App FW into the DC2100A-C. This code is a modification of the open source PIC18 bootloader from Diolan. Although the source code is provided, the user is directed to the documentation on the Diolan website for details (www.diolan.com/pic/bootloader.html).

The boot.inc file is used by both the Boot FW and the App FW to define the memory spaces for each piece of code. In addition, a page of program memory stores Manufacturing Data for the App FW. Figure 14 shows how the program memory is defined by the boot.inc file.

Figure 14 also shows the execution flow between the Boot FW and the App FW. When the PIC18 is powered up or resets, it immediately jumps to the Boot FW (Jump 1). If the Boot FW detects that the App FW is present, it will jump to the memory location defined as the App Reset Vector (Jump 2). The actual start of the App FW is then jumped to from the App Reset Vector (Jump 3). When the App FW wishes to re-enter the bootloader to upload FW into the PIC18, it sets a flag and resets the processor (Jump 4). When the Boot FW sees this flag set, it will enter the bootloader mode to accept new FW into the PIC18.

Note that the Boot FW does not use the PIC18 interrupts. Instead it points the PIC18 Interrupt Vectors to the App Interrupt Vectors (Interrupt Jump 1), which then jump to the actual Interrupt Service Routines in the App FW (Interrupt Jump 2).

Any new code compiled for the PIC18 on the DC2100A, must maintain the addresses in the boot.inc file to maintain compatibility with the Boot FW.

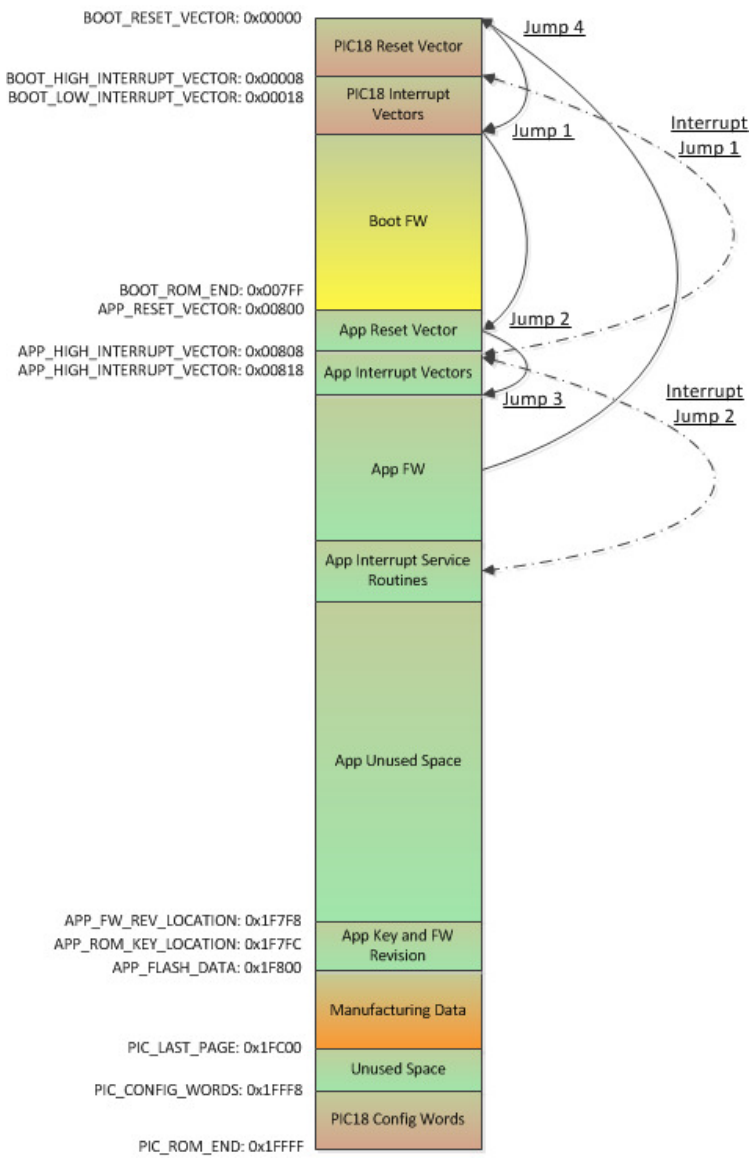


Figure 14 - DC2100A PIC18 Program Memory

FW UPDATE SOFTWARE

The Update SW functions with the Boot FW to load new App FW into the DC2100A-C. This code is a modification of the open source PIC18 bootloader from Diolan. Although the source code is provided, the user is directed to the documentation on the Diolan website for details (www.diolan.com/pic/bootloader.html).

Before the Update SW can communicate with the DC2100A-C, the PIC18 must first be switched to the bootloader mode. The "Firmware Update" button on the Control Panel of the DC2100A GUI will achieve this after it establishes communication with the DC2100A-C.

Figure 15 - JP7 Placement to Force Bootloader Mode

Alternatively, the PIC18 can be placed in bootloader mode upon power up if JP7 pins 2A and 3A are shorted together upon power up or reset. This is shown in Figure 15.

Once bootloader mode has been entered, the DC2100A will begin to flicker the Green LED with a 1 second period, indicating that it is ready to accept code. The fw_update.exe program can be run at the command line with the format:

fw_update.exe -e -w -vid 0x1272 -pid 45067 -ix DC2100A.hex

As the bootloader accepts commands to erase and write program memory, the Amber LED will light while the commands are being executed. Note that the erase operation can take several seconds as the full App FW space is erased.

Figure 16 shows the command line following a successful upload of new FW through the FW Update Software.

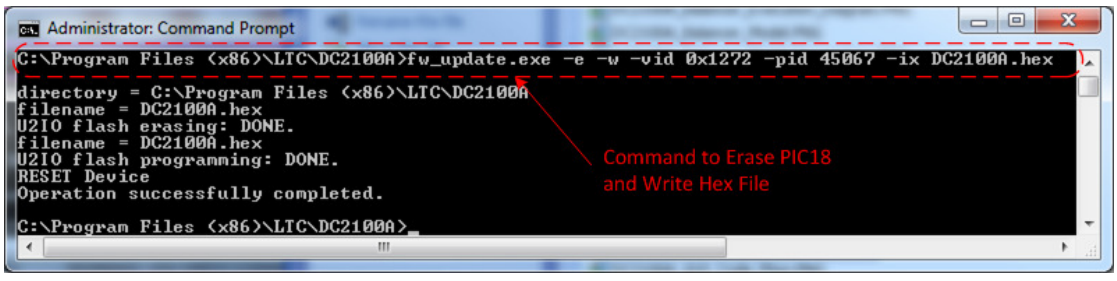


Figure 16 - Successful FW Update Command

Appendix A - Voltage Monitoring using LTC6804-2

One of the most basic functions of the LTC6804-2 is to monitor the cell voltages in a battery. This section details the DC2100A code available to perform cell voltage measurements with the LTC6804-2.

Many commands and register groups are used to monitor the voltages through the LTC6804-2. They are pictured in Figure 17.

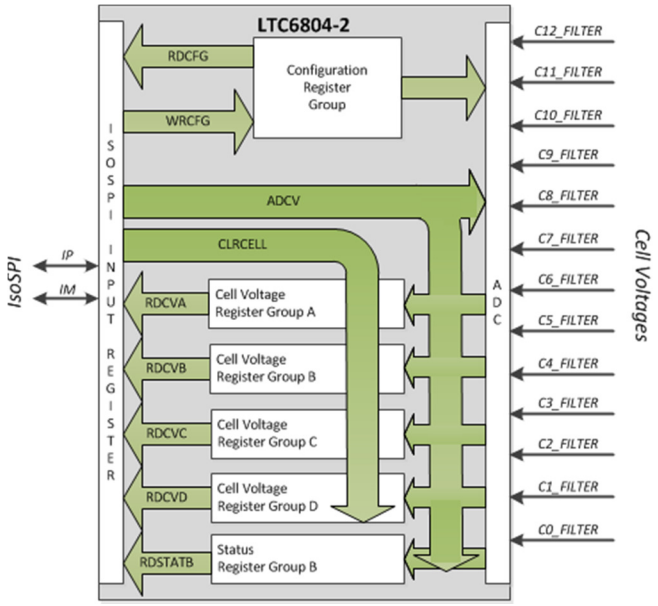


Figure 17 - Commands and Register Groups used for Voltage Monitoring

The following commands are used to perform voltage monitoring:

- RDCFG and WRCFG are used to read the Configuration Register Group, modify it, and write it back to the LTC6804-2. The following bits in this register group are pertinent to voltage monitoring:
  1. REFON – this bit turns on the ADC reference, so that the sampling and conversion time is the only delay between starting and reading ADC samples.
  2. ADCOPT – this bit partially controls the conversion mode of the ADC.
  3. VUV/VOV – these bits set the thresholds used for detecting overvoltage (OV) and undervoltage (UV) conditions for each set of ADC samples.
- CLRCELL is used to clear the last set of ADC samples, so that it can be detected when a new set of ADC samples do not start due to a failure to communicate with the device.
- ADCV begins the ADC sampling and conversion of the cell voltages. Bits in this command, along with the ADCOPT bit in the Configuration Register Group, control the mode of the ADC conversion. When the conversion is complete, the results are stored in Cell Voltage Register Groups A, B, C, and D. The OV and UV status flags are set in Status Register Group B after these ADC samples are compared to the new ADC conversion results.
- The RDCVA, RDCVB, RDCVC, and RDCVD commands retrieve the ADC results from Cell Voltage Register Groups A, B, C, and D.
- The RDSTATB command retrieves the UV and OV flags from Status Register Group B.

To aid the customer that wishes to monitor the voltage without managing the details of these commands and register groups, the DC2100A code provides several public function.

Some of the LTC6804-2 functions only need to be called once after the IC has left the SLEEP state. These functions are shown in Code Sample 5.

```
// Turns the LTC6804 ADC Reference on and off.
// return void
void LTC6804_Refon_Set(
   int8 board_num,     // The logical address for the PCB containing this LTC6804-2 IC.
   BOOLEAN refon       // TRUE to turn on the reference, FALSE to turn it off.
);

// Max reference wakeup time in us from datasheet Electrical Characteristics page 7 as tREFUP.
#define LTC6804_TREFUP      4400   // max reference wakeup time.

// Sets the LTC6804 under-voltage and over-voltage thresholds in LTC6804_UVOV_RESOLUTION units.
// return void
void LTC6804_UVOV_Thresholds_Set(
   int8 board_num,     // The logical address for the PCB containing this LTC6804-2 IC.
   int16 vuv_value,    // Under-voltage value.
   int16 vov_value     // Over-voltage value.
);
```

Code Sample 2 - Init Functions for Voltage Monitoring

The *LTC6804_RefOn_Set()* function turns on the reference for ADC conversions. The FW must wait for t_{REFUP} after turning on the reference before starting an ADC conversion. This action will bring the IC from the STANDBY state to the REFUP state. Note that reference can be left off in order for the LTC6804-2 to consume less power, but it will require the FW to wait for the t_{REFUP} delay for each ADC conversion.

The *LTC6804_UVOV_Thresholds_Set()* function configures the UV and OV thresholds for the LTC6804-2. After each ADC conversion, the cell voltages are compared to these thresholds, and the IC sets UV and OV flags if ether threshold is violated. The UV and OV thresholds remain set as long as the IC does not reenter the SLEEP state.

When cell voltage measurements are started, there are several cell combinations that can be measured with several conversion modes. These are shown in Code Sample 6. The cell combinations either begin sampling for one pair or all 12 cells. The cells are sampled simultaneously, and then converted a pair at a time. The conversion modes offer a trade-off between conversion speed and accuracy. The cell combination and conversion mode options are used as inputs to the functions available to start and complete ADC conversions shown in Code Sample 7.

When using these functions to perform a voltage monitoring task, the *LTC6804_Cell_ADC_Clear()* function would

```
// Cell selection for ADC Conversion from datasheet table 35.
typedef enum {
  LTC6804_CH_ALL = 0,                // All Cells
  LTC6804_CH_CELL_1_AND_7 = 1,  // Cell 1 and Cell 7
  LTC6804_CH_CELL_2_AND_8 = 2,  // Cell 2 and Cell 8
  LTC6804_CH_CELL_3_AND_9 = 3,  // Cell 3 and Cell 9
  LTC6804_CH_CELL_4_AND_10 = 4, // Cell 4 and Cell 10
  LTC6804_CH_CELL_5_AND_11 = 5, // Cell 5 and Cell 11
  LTC6804_CH_CELL_6_AND_12 = 6, // Cell 6 and Cell 12
  LTC6804_CH_CELL_NUM
} LTC6804_CH_CELL_TYPE;

// Conversion Modes Available in the LTC6804 from datasheet table 35.
typedef enum {
  LTC6804_CONVERSION_27KHZ_MODE = 0,  // 27kHz conversion mode
  LTC6804_CONVERSION_14KHZ_MODE = 1,  // 14kHz conversion mode
  LTC6804_CONVERSION_7KHZ_MODE = 2,   // 7kHz conversion mode
  LTC6804_CONVERSION_3KHZ_MODE = 3,   // 3kHz conversion mode
  LTC6804_CONVERSION_2KHZ_MODE = 4,   // 2kHz conversion mode
  LTC6804_CONVERSION_26HZ_MODE = 5,   // 26Hz conversion mode
```

Code Sample 6 - Cell Combinations and Conversion Modes

be first sent to clear the ADC results. This would then be followed by the *LTC6804_Cell_ADC_Start()* function would then be called to start the conversion of a set of

```
// Clears the LTC6804 Cell Voltage ADC registers.  This is useful to detect if the conversion
// was started properly when the results are read.
// return void
void LTC6804_Cell_ADC_Clear(
  int8 board_num                    // The logical address for the PCB containing this LTC6804-2 IC.
);

// Starts the LTC6804 Cell Voltage ADC conversion at the specified conversion mode.
// Note function always permits discharge.
// return void
void LTC6804_Cell_ADC_Start(
  int8 board_num,                   // The logical address for the PCB containing this LTC6804-2 IC.
  LTC6804_CONVERSION_MODE_T mode,   // The mode to use for ADC conversion
  LTC6804_CH_CELL_TYPE cell_select, // The cells to convert
  BOOLEAN discharge_permitted       // True if discharge is to be permitted during this cell voltage conversion.
);

// Reads the LTC6804 Cell Voltage ADC conversion results.
// return TRUE if the LTC6804 communication was successful.
BOOLEAN LTC6804_Cell_ADC_Read(
  int8 board_num,                   // The logical address for the PCB containing this LTC6804-2 IC.
  LTC6804_CH_CELL_TYPE cell_select, // The cells to convert.
  unsigned int16* adc_value_ptr     // Pointer where to return up to LTC6804_NUM_CELLV_ADC cell voltages.
);

// Gets the LTC6804 flags indicating under-voltage and over-voltage conditions are present.
// return TRUE if the LTC6804 communication was successful.
BOOLEAN LTC6804_UVOV_Flags_Get(
  int8 board_num,                   // The logical address for the PCB containing this LTC6804-2 IC.
  int16* vuv_flags,                 // Pointer where to return under-voltage flags in a bitmap with bit 0 = cell 0.
  int16* vov_flags                  // Pointer where to return over-voltage flags in a bitmap with bit 0 = cell 0.
);
```

Code Sample 7 - Voltage Monitoring Functions

cells using a specific conversion mode. Note that the *LTC6804_Cell_ADC_Start()* function will only modify the Configuration Register Group if necessary to change the ADC conversion mode. Otherwise, it will only send the appropriate ADCV command to the LTC6804-2 ICs. The time required for the ADC conversion to complete is specified in Code Sample 8. Since these times are not insignificant, the LTC6804-2 driver separates the *LTC6804_Cell_ADC_Start()* function from the *LTC6804_Cell_ADC_Read()* function so the application code can perform other operations while the conversion is underway. After the conversion delay is complete, the *LTC6804_Cell_ADC_Read()* function is called to retrieve the results and the *LTC6804_UVOV_Flags_Get()* function is called to return the UV and OV flags.

```
// in us, delay between sampling and reading a pair of ADC conversions
// Time for ADC results for one channel to be ready, from Table 6 (t1C).
#define LTC6804_CONVERSION_27KHZ_DELAY      201       // 27kHz conversion mode
#define LTC6804_CONVERSION_14KHZ_DELAY      230       // 14kHz conversion mode
#define LTC6804_CONVERSION_7KHZ_DELAY       405       // 7kHz conversion mode
#define LTC6804_CONVERSION_3KHZ_DELAY       501       // 3kHz conversion mode
#define LTC6804_CONVERSION_2KHZ_DELAY       754       // 2kHz conversion mode
#define LTC6804_CONVERSION_26HZ_DELAY       34000     // 26Hz conversion mode

// in us, delay between sampling and reading a all ADC conversions
// Time for ADC results for all channels to be ready, from Table 5 (t6C).
#define LTC6804_CONVERSIONS_ALL_27KHZ_DELAY 11130     // 27kHz conversion mode
#define LTC6804_CONVERSIONS_ALL_14KHZ_DELAY 1288      // 14kHz conversion mode
#define LTC6804_CONVERSIONS_ALL_7KHZ_DELAY  2335      // 7kHz conversion mode
#define LTC6804_CONVERSIONS_ALL_3KHZ_DELAY  3033      // 3kHz conversion mode
#define LTC6804_CONVERSIONS_ALL_2KHZ_DELAY  4430      // 2kHz conversion mode
#define LTC6804_CONVERSIONS_ALL_26HZ_DELAY  202000    // 26Hz conversion mode
```

Code Sample 8 - ADC Conversion Times

An example of how these functions are used to perform voltage monitoring with a single board is shown in Code Sample 3. The UV threshold was set to 2V and the OV threshold was set to 4V. A diagram of the execution of this code is shown in Figure 18, with Cell 1 set to 1.9V and each successive cell voltage increased by 0.2V.

```
// Example usage of Voltage Monitoring Functions.
{
  int8 cell_num;
  int16 adc_value[DC2100A_NUM_CELLS];
  int16 vov_flags, vuv_flags;
  BOOLEAN comm_success;

  // Set UV to 2.0V and OV to 4.0V.
  LTC6804_UVOV_Thresholds_Set(0,
                              (2.00 * UV_PER_V/LTC6804_UVOV_RESOLUTION),
                              (4.00*UV_PER_V/LTC6804_UVOV_RESOLUTION));

  // Turn on the ADC reference for board 0.
  LTC6804_Refon_Set(0, TRUE);

  // Wait for the reference to come up.
  delay_us(LTC6804_TREFUP);

  // Clear ADC results on board 0.
  LTC6804_Cell_ADC_Clear(0);

  // Start converting all of the cell ADC values for board 0 using 7kHz mode while permitting discharge.
  LTC6804_Cell_ADC_Start(0, LTC6804_CONVERSION_7KHZ_MODE, LTC6804_CH_ALL, TRUE);

  // Wait for conversion to be complete
  // Note - It is worthwhile to perform some other action while waiting for the ADC conversion to be complete.
  delay_us(LTC6804_CONVERSIONS_ALL_7KHZ_DELAY);

  // Read the ADC results for all cells.
  comm_success = LTC6804_Cell_ADC_Read(0, LTC6804_CH_ALL, adc_value);

  if(comm_success == TRUE)
  {
    // If results were successfully returned from LTC6804_Cell_ADC_Read() function,
    // verify that the ADC conversion was successfully started.
    for(cell_num = 0; cell_num < DC2100A_NUM_CELLS; cell_num++)
    {
      if(adc_value[cell_num] == LTC6804_ADC_CLEAR)
      {
        // If results were cleared, then ADC conversion did not successfully start.
        comm_success = FALSE;
      }
    }
  }

  // If ADC conversion was successfully started and results were returned from
  // LTC6804_Cell_ADC_Read() function, get the UV and OV flags.
  if(comm_success == TRUE)
  {
    comm_success = LTC6804_UVOV_Flags_Get(0, vuv_flags, vov_flags);
  }

  // If results were returned from LTC6804_UVOV_Flags_Get() function, do something with the results.
  if(comm_success == TRUE)
  {
    if(vuv_flags || vov_flags)
    {
      // do something if UV or OV is present.
    }
  }
}
```

Code Sample 3 - Example Usage of Voltage Monitoring Functions.

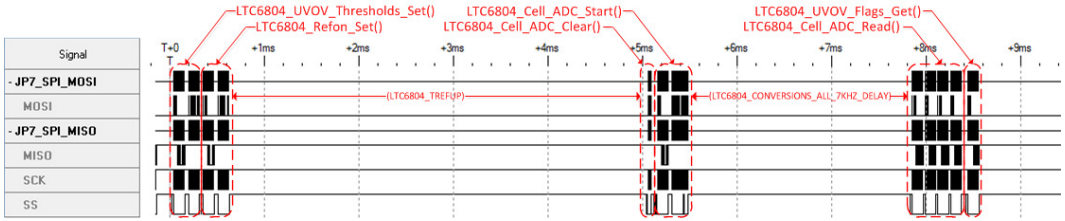


Figure 18 - Execution of Voltage Monitoring Function Usage Example

- LTC6804_UVOV_Thresholds_Set (detailed in Figure 19). This function would typically only be called during initialization, or when the thresholds are changed.
- LTC6804_Refon_Set (detailed in Figure 20). This function would typically be only called during initialization, or before and after ADC conversions if power consumption is to be minimized.
- LTC6804_TREFUP Delay. Figure 18 shows how much the ADC conversion rate is limited if the reference is continuously turned on and off, although it does come at the expense of extra power consumption.
- LTC6804_Cell_ADC_Clear (detailed in Figure 20). This function clears the ADC conversion registers, so that an incomplete conversion can be detected by the firmware.
- LTC6804_Cell_ADC_Start (detailed in Figure 24). This function begins the ADC conversion at the 7kHz mode. Note that if the ADCOPT bit does not need to be changed, only the ADCV command will be sent to the LTC6804.
- LTC6804_CONVERSIONS_ALL_7KHZ_DELAY. Figure 18 shows the required delay when converting all cells using the 7kHz mode. As this time isn't trivial, the firmware would likely wish to perform some other task during this time.
- LTC6804_Cell_ADC_Read (detailed in Figure 23). This function reads the 12 cell voltage out of the 4 register groups in which they are stored. The voltages measured are approximately 1.9V and increasing to 4.1V in 0.2V increments.
- LTC6804_UVOV_Flags_Get (Figure 22). The function reads the flags out of the auxiliary register. In this example, only Cell 1 is undervoltage and only Cell 12 is overvoltage.

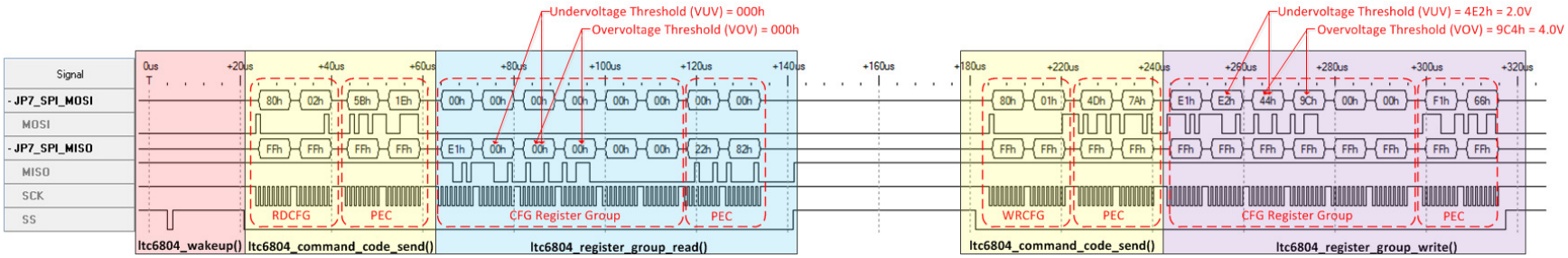


Figure 19 - LTC6804_UVOV_Thresholds_Set() with VUV = 2.0V and VOV = 4.0V

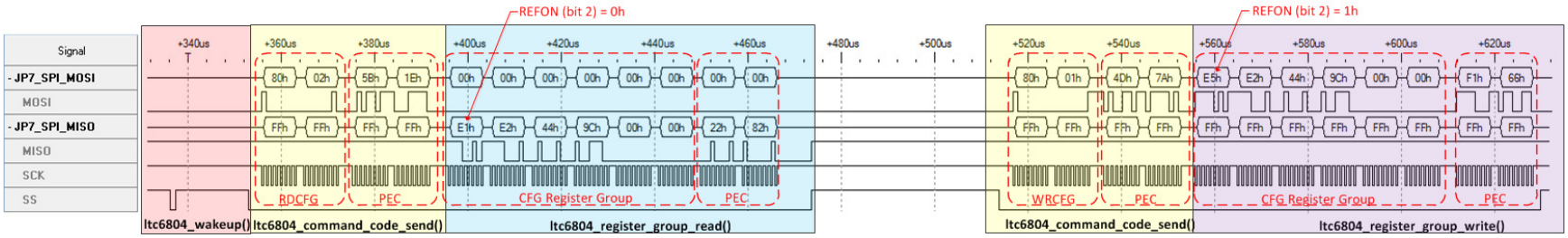


Figure 20 - LTC6804_Refon_Set() to turn on ADC reference.

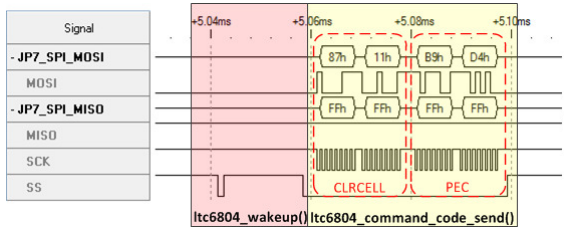


Figure 21 - LTC6804_Cell_ADC_Clear() to Clear Previous ADC Conversions

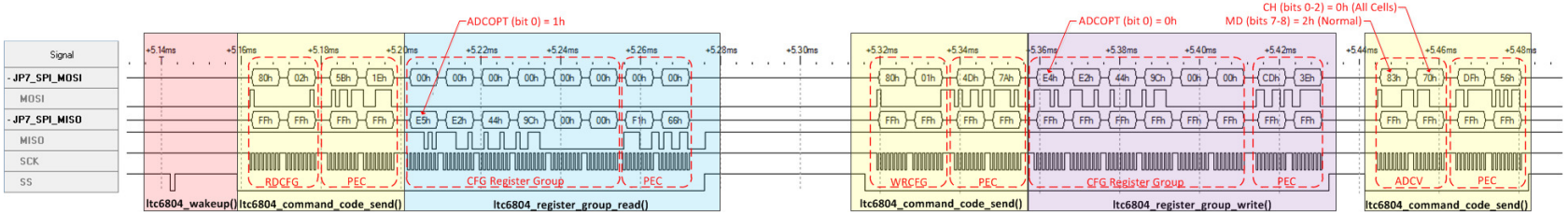


Figure 24 - LTC6804_Cell_ADC_Start() to Start ADC Conversions using 7kHz Mode

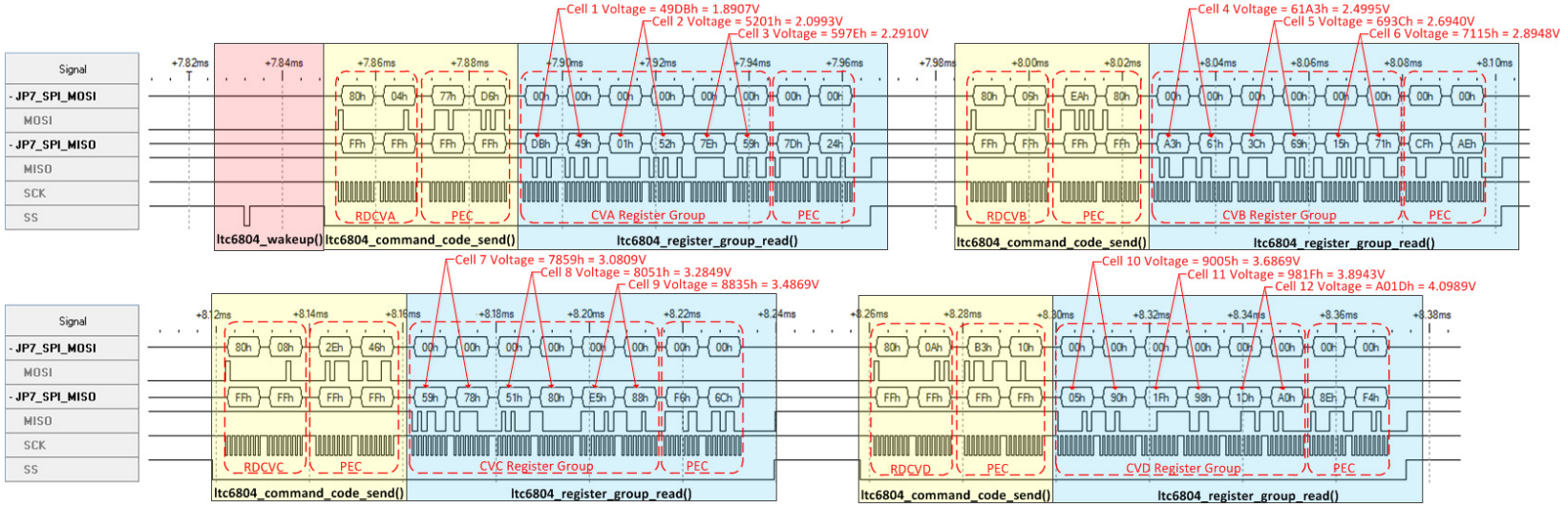


Figure 23 - LTC6804_Cell_ADC_Read() to Read 12 Cells

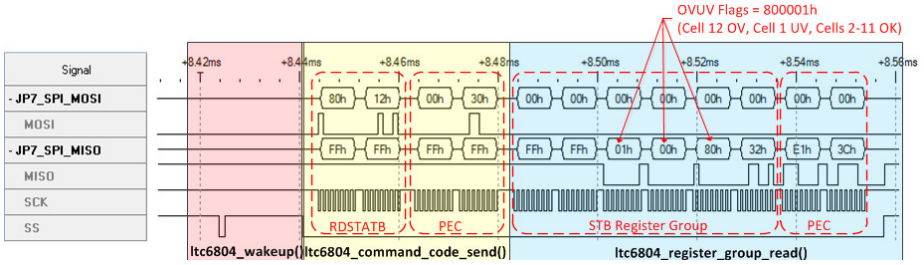


Figure 22 - LTC6804_UVOV_Flags_Get() to Received UV and OV Flags

Appendix B - LTC6804-2 I²C/SPI Master Using GPIOS

The DC2100A uses I/O ports GPIO3, GPIO4 and GPIO5 on the LTC6804-2 as an I2C and SPI master port to communicate to with the LTC3300-1, LTC1380, and 24AA64 slave devices.

The functionality is implemented through the COMM register group, using the WRCOMM, RDCOMM, and STCOMM commands. A diagram of this system is shown in Figure 25.

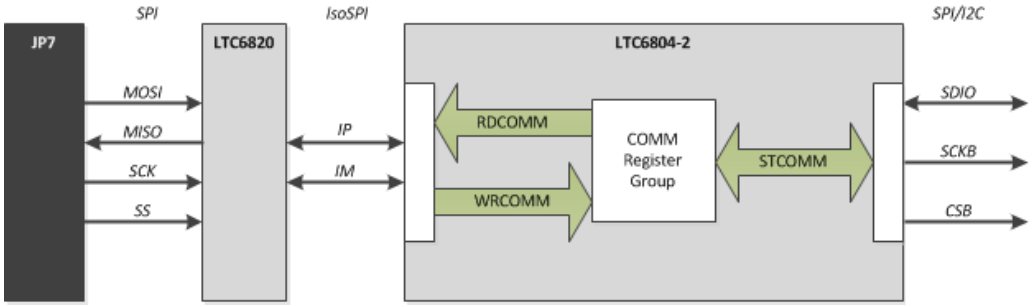


Figure 25 - Diagram of I2C/SPI Communication through LTC6804-2 on DC2100A

The implementation of this communication can be complex, due to the 6 byte COMM register group only being capable of containing 3 data bytes at a time, with one ICOM and FCOM code for each byte. In the LTC6804-2 datasheet, Table 15 shows the layout for the COMM register group where Table 16 and Table 17 show the values for ICOM and FCOM.

The following commands are used to pass SPI/I2C bytes through the COMM register group to the IsoSPI interface:

- WRCOMM is used to load data bytes, ICOM codes, and FCOM codes into the COMM register group.
- RDCOMM is used to read data bytes, ICOM codes, and FCOM from the COMM register group.
- STCOMM is used to transfer the data bytes from the COMM register group to the GPIO implemented I²C/SPI bus

To aid the customer that wishes to pass bytes through the LTC6804-2 to the GPIO implemented I²C/SPI bus, the DC2100A code provides the following functions shown in Code Sample 4.

```
// Writes a string of bytes to the LTC6804 SPI port implemented on its GPIO pins.
// return void
void LTC6804_SPI_Write(
        int8 board_num,       // The logical address for the PCB containing this LTC6804-2 IC.
        BOOLEAN start,        // TRUE if the CS should be raised at the start of SPI communication.
        BOOLEAN stop,         // TRUE if the CS should be lowered at the end of SPI communication.
        int8* data_ptr,       // Pointer to the data to write to the SPI Bus.
        int16 num_bytes,      // The number of bytes to write to the SPI Bus.
        int16 baud_khz        // The baud rate at which the SPI Bus should be clocked.
);

// Writes one byte, and then reads a string of bytes to the LTC6804 SPI port implemented on its GPIO pins.
// return void
void LTC6804_SPI_Read(
        int8 board_num,       // The logical address for the PCB containing this LTC6804-2 IC.
        BOOLEAN start,        // TRUE if the CS should be raised at the start of SPI communication.
        BOOLEAN stop,         // TRUE if the CS should be lowered at the end of SPI communication.
        int8* data_ptr,       // Pointer to a byte to first write to the SPI Bus, and where to store
                              // the data read from the SPI Bus.
        int16 num_bytes,      // The number of bytes to write to the SPI Bus.
        int16 baud_khz        // The baud rate at which the SPI Bus should be clocked.
);

// Writes a string of bytes to the LTC6804 I2C port implemented on its GPIO pins.
// return void
void LTC6804_I2C_Write(
        int8 board_num,       // The logical address for the PCB containing this LTC6804-2 IC.
        BOOLEAN start,        // TRUE if this write would be started by an I2C Start Condition.
        BOOLEAN stop,         // TRUE if this write would be ended with an I2C Stop Condition.
        int8* data_ptr,       // Pointer to the data to write to the I2C Bus.
        int16 num_bytes,      // The number of bytes to write to the I2C Bus.
        int16 baud_khz        // The baud rate at which the I2C Bus should be clocked.
);

// Writes one byte, and then reads a string of bytes to the LTC6804 I2C port implemented on its GPIO pins.
// return void
void LTC6804_I2C_Read(
        int8 board_num,       // The logical address for the PCB containing this LTC6804-2 IC.
        BOOLEAN start,        // TRUE if this read would be started by an I2C Start Condition.
        BOOLEAN stop,         // TRUE if this read would be ended with an I2C Stop Condition.
        int8* data_ptr,       // Pointer to a byte to first write to the I2C Bus, and where to
                              // store the data read from the I2C Bus.
        int16 num_bytes,      // The number of bytes to read to the I2C Bus.
        int16 baud_khz        // The baud rate at which the I2C Bus should be clocked.
);
```

Code Sample 4 - Functions for LTC6804-2 GPIO SPI/I2C Comm.

The details of operation for one of these functions are shown in Figure 26, for the LTC6804_SPI_Write() function. In order to write bytes through the LTC6804-2 to its GPIO SPI port, the WRCOMM and STCOMM commands must be used. Subroutines used by the LTC6804-2 driver are color-coded, as these are the pieces of code that are common to all of the public functions exposed through the API.

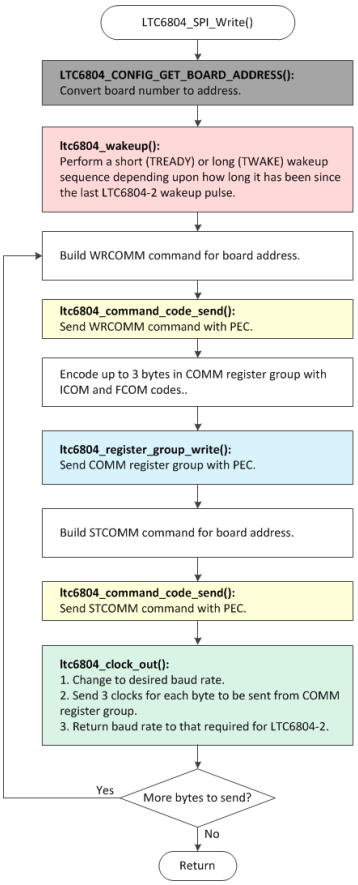


Figure 26 - LTC6804_SPI_Write() Flowchart

An example of the execution of this function is detailed in Figure 28, where 5 bytes {0xA9, 0xBB, 0xB3, 0xBB, 0xB3} are sent to board 0x00 at 1MHz. These bytes will write a balance command to the two LTC3300-1 ICs on that board.

The second function that will be detailed is the LTC6804_I2C_Read() function shown in Figure 27. As this function needs to receive data from the GPIO I²C bus, it needs to use the RDCOMM command in addition to the WRCOMM and STCOMM commands.

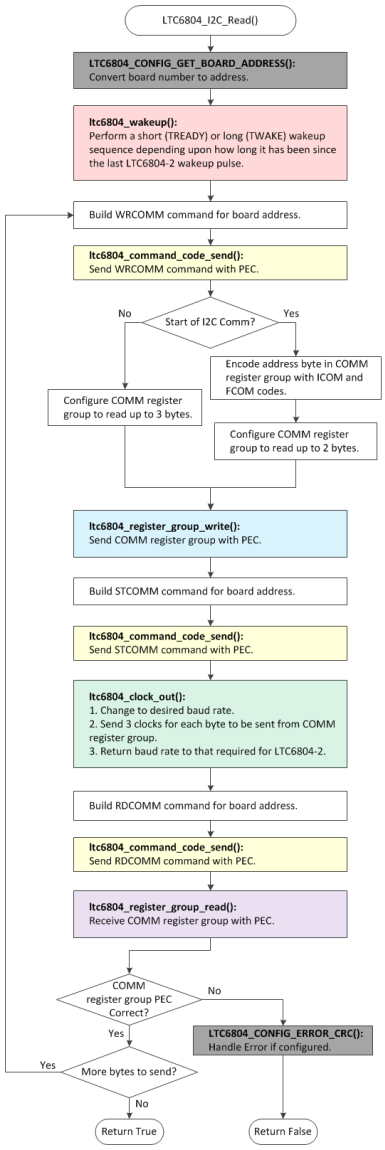


Figure 27 - LTC6804_I2C_Read() Flowchart

An example of the execution of this function is detailed in Figure 29, where 3 bytes {0x00, 0x11, 0x22} are read from I²C address 0x50 at 400kHz. Note that this function actually writes a byte as part of its execution, since it does not make sense to read from I²C without first writing an address to the bus. Also note that the acknowledgements are not checked by this function, and error detection is left to the higher level code.

The other two functions available to write to and read from the GPIO implemented I²C/SPI bus operate similarly to these two functions and will not be detailed.

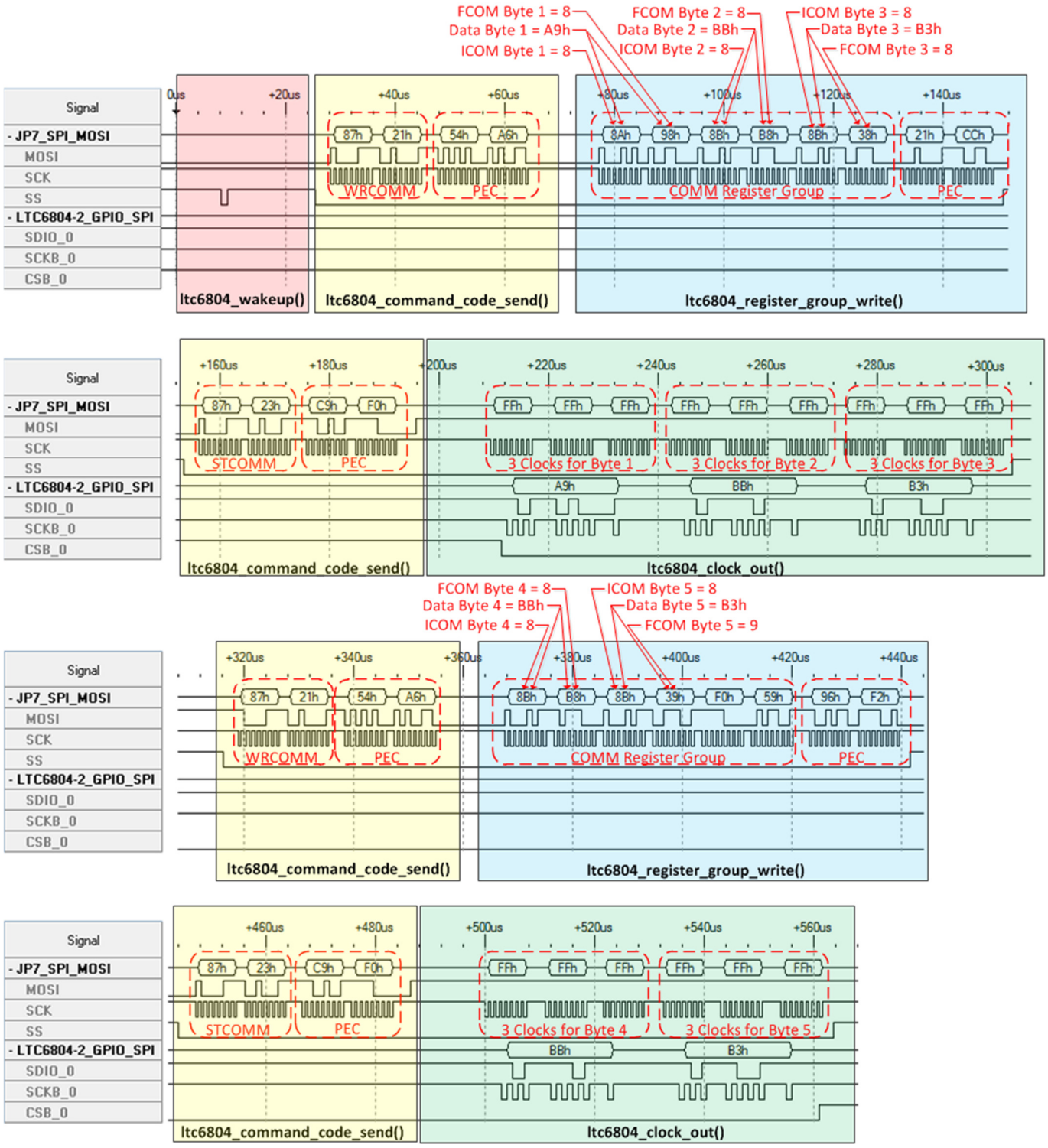


Figure 28 - LTC6804_SPI_Write() Execution

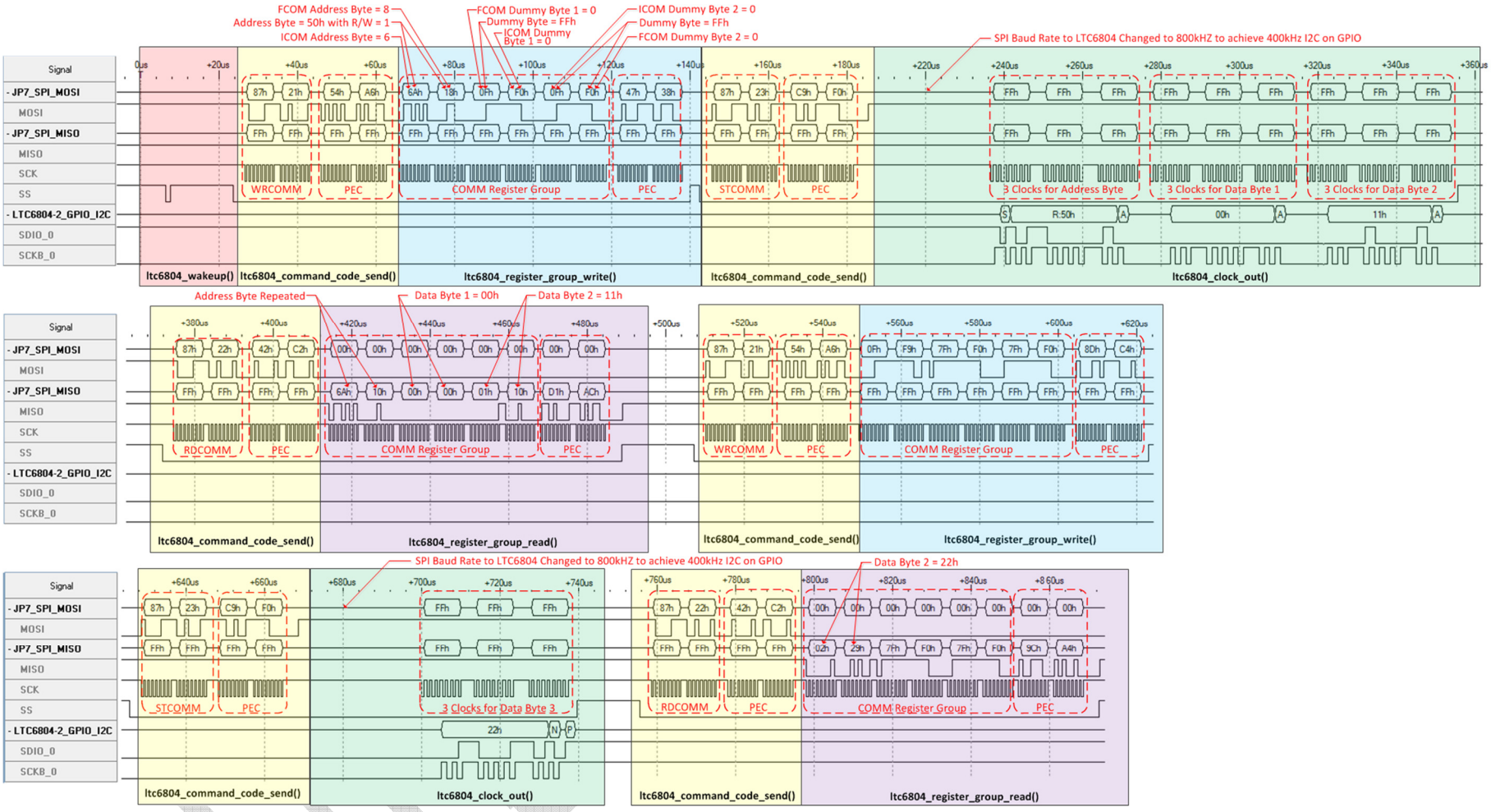
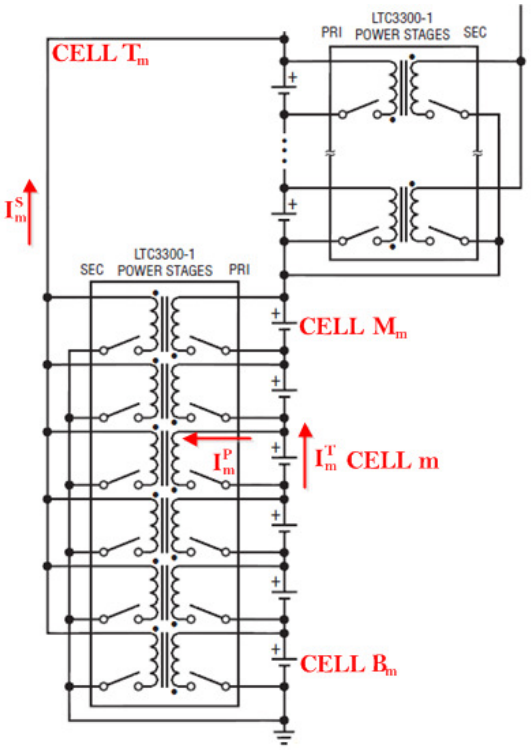


Figure 29 - LTC6804_I2C_Read() Execution

Appendix C - Setting Balancer Times from Charge Imbalance

The interactions between each of the cells in active balancing complicates the calculation for how the balancers should be controlled in order to move a desired amount of charge. To address this calculation, one of the Balancer_Set() functions available from the Balancer Code Module accepts the amount of charge to move in each cell (ΔQ*) as its input and calculates the optimal set of balancer commands to achieve this ΔQ*). At this time, this function is only written to calculate the balance times for one DC2100A balancing 12 cells.

Figure 30 shows the parameters involved for a ΔQ-to-Δt calculation to be performed for Cell m on a DC2100A. Through superposition, the effect of each balancer can be calculated separately and the combined effect can then be analyzed. The Balancer_Set(ΔQ*) function initially calculates the Δt necessary for the primary current for the cell balancer to equal ΔQ*. An iterative loop is then performed to adjust Δt until the actual amount of charge moved in each cell is as close to ΔQ* as possible. Most sets of ΔQ* do not have a set of Δt that will move the exact amount of desired charge. An obvious example is when ΔQ* equals zero for all cells except one. Since the DC2100A circuit must move charge from one cell to another, it is impossible to move charge from one cell but not move any charge to the others. In addition, there will be charge losses due to the balancing operation not being 100% efficient. To account for these discrepancies, the Balancer_Set(ΔQ*) will iterate to a solution as close as possible to the desired ΔQ* for each cell, and the difference between the desired ΔQ* and actual ΔQ is spread equally amongst all of the cells.

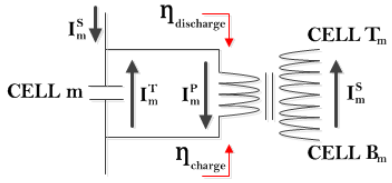


N = Total Number of Cells in Battery Stack
CELL T_m = Highest Potential Cell Receiving Secondary Currents from Cell m
CELL M_m = Highest Potential Cell Supplying Secondary Currents to Cell m
CELL B_m = Lowest Potential Cell Supplying Secondary Currents to Cell m
I^T_m = Current Discharging from Cell m
I^P_m = Primary Balance Current Discharging from Cell m
I^S_m = Secondary Balance Current Resulting from I^P_m

$\Delta Q^P_m = t_m \times I^P_m$ = Charge Discharged from Cell m due to I^P_m Balancing for Δt_m seconds

$\Delta Q^T_m = \Delta Q^P_m - \sum_{n=B_m}^{M_m} \Delta Q^S_n$ = Charge Discharged from Cell m due to All Primary and Secondary Balance Currents

Figure 30 - Parameters for Balance Time CalcIculation



Discharge

$$\frac{V_m \times \Delta Q^P_m}{\eta_{discharge}} = \sum_{n=B_m}^{T_m} V_n \times \Delta Q^S_m$$

Charge

$$V_m \times \Delta Q^P_m = \frac{\sum_{n=B_m}^{T_m} V_n \times \Delta Q^S_m}{\eta_{charge}}$$

Equation 1 - Balancer Energy Balance Equations

For the Balancer_Set(ΔQ*) function to estimate the effect of a balance operation, it must calculate the secondary currents for a given primary current. Equation 1 shows the energy balance equations that are used to derive the equation for the balancer secondary currents. Since $V_m = V_n$ for all n when the cells are balanced, and it's assumed that the Balancer Control Task is used to prevent the cells from ever becoming too imbalanced, these equations simplify to those in Equation 2.

Discharge

$$\Delta Q^S_m = \frac{\Delta Q^P_m \times \eta_{discharge}}{T_m - B_m + 1}$$

Charge

$$\Delta Q^S_m = \frac{\Delta Q^P_m \div \eta_{charge}}{T_m - B_m + 1}$$

Equation 2 - Balancer Secondary Current Equations

With each iteration of the Balancer_Set(ΔQ*) function, the total charge moved is calculated and used to better estimate the balancer commands in the next iteration. Equation 3 shows how each guess [k] for the primary charge moved by the balance operation is calculated from the previous guess [k-1] of the primary charge plus two feedback terms: the Cell Charge Error and the Half-Stack Charge Error Feedback Terms.

$$\Delta Q_m^P[k] = \Delta Q_m^P[k-1] + \frac{\left(\Delta Q_m^* - \sum_{n=1}^{N}\frac{(\Delta Q_n^* - \Delta Q_n^T)}{N} - \Delta Q_m^T[k-1]\right)}{\eta_c} + \frac{\sum_{n=B_m}^{M_m}\frac{(\Delta Q_n^* - \Delta Q_n^T[k-1])}{M_m - B_m + 1} - \sum_{n=M_m+1}^{T_m}\frac{(\Delta Q_n^* - \Delta Q_n^T[k-1])}{M_m - B_m}}{\eta_s}$$

Cell Charge Error Feedback Term

Half Stack Charge Error Feedback Term

Equation 3 - Balancer_Set(ΔQ*) Iteration Equation

The Cell Charge Error Feedback Term, accounts for the difference between ΔQ^* and the actual ΔQ^T moved in the cell. The difference between ΔQ^* and the actual ΔQ^T moved in all of the cells is averaged and added to ΔQ^* to account for most sets of ΔQ^* not having a set of Δt that will move the exact amount of desired charge.

For many situations the Cell Charge Error Feedback Term is adequate for the Balancer_Set(ΔQ*) function to converge to a value. In some instances, however cells receiving secondary currents from but not supplying secondary currents to Cell m will not be able to achieve balance unless Cell m adjusts its balance time. The Half Stack Charge Error Feedback Term accounts for this condition to ensure that the cells in the bottom half of the stack will continue to adjust their balance times if the cells in the top half of the stack have not converged to a solution. It achieves this by continuing the iteration as long as the average difference between the desired ΔQ^* and the actual ΔQ^T moved in the cells of each half stack are not equal.

Each Feedback Term includes a damping factor η to control the rate at which the iterations settle upon a set of Δt. The effect of these damping factors is shown in Figure 31.

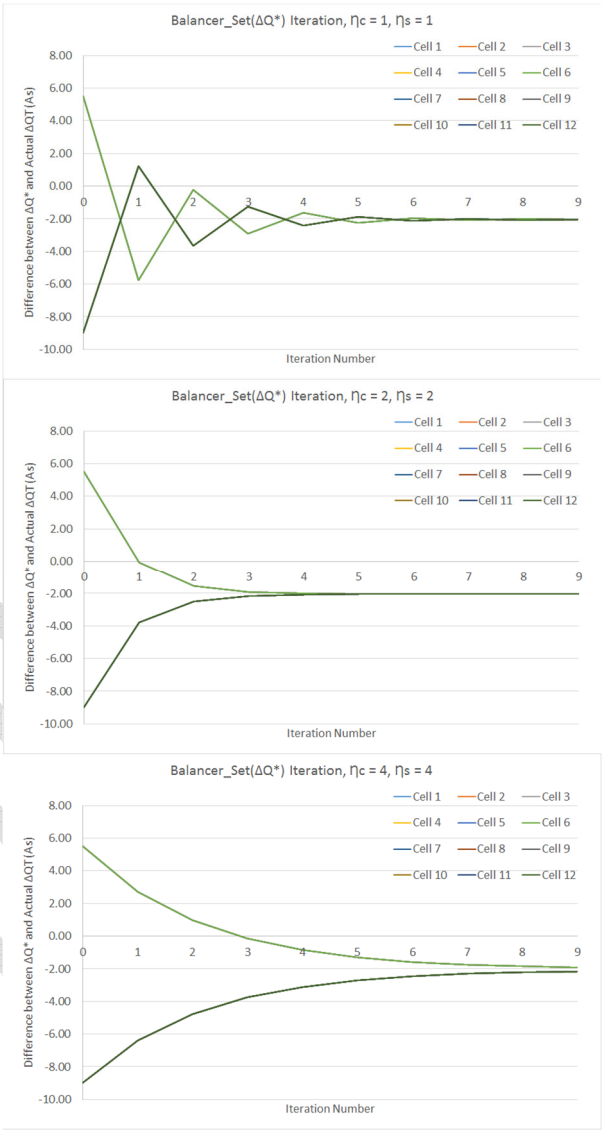


Figure 31 - Effect of Damping Factor in Balanscer_Set(ΔQ*)

Once the Balancer_Set(ΔQ*) function has converged on a set of Δt, the actual ΔQ^T values are passed back to the calling function. The purpose of this is to allow the higher level function the option of redistributing the difference between ΔQ^* and the actual ΔQ^T due to differences in cell capacity.

An example of where this is done is the SuperCap Demo System where two of the cells are purposely given different capacities than the other ten. The different capacities cause the difference between ΔQ^* and the actual ΔQ^T to represent a different % SOC in each cell if they are spread evenly across each cell. Figure 33 shows that even with a

huge variation in the cell voltages, the Balancer_Set(ΔQ*) function quickly converges to a set of balance times that moves the desired charge in each cell with the difference between ΔQ* and the actual ΔQ^T evenly distributed across each cell. Because the charge losses are small, the SuperCap Demo system will still end relatively balanced with 55mV between the maximum and minimum cell voltages. This can be seen in Figure 33.

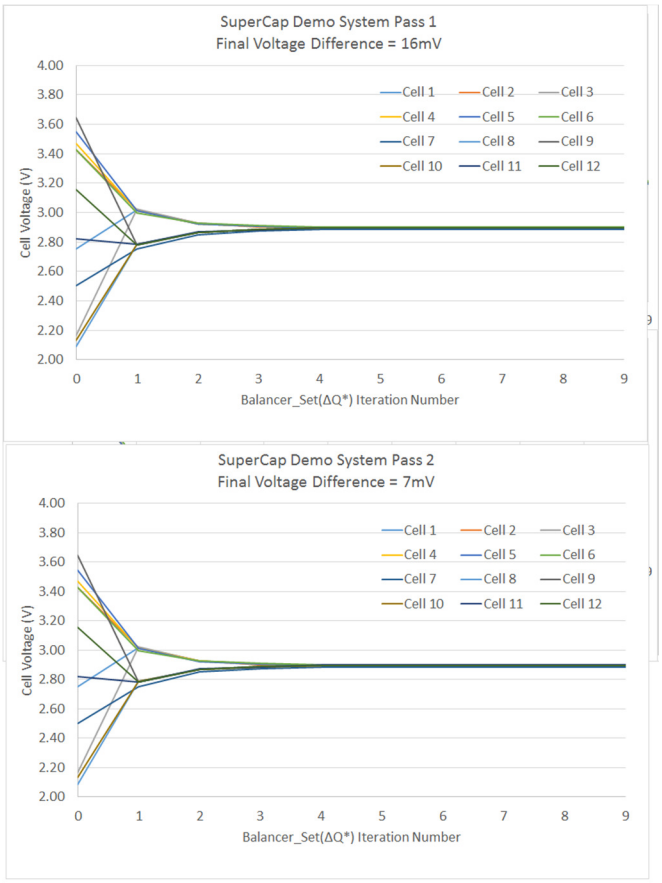


Figure 33 - SuperCap Demo Balance Simulation Pass 1

The SuperCap Demo SOC function performs a second pass, however, where it calls the Balancer_Set(ΔQ*) function again with its ΔQ* estimate adjusted with the , Q_{Loss} returned from the first Balancer_Set(ΔQ*) call redistributed according to the cell capacities. Figure 32 shows that after this second pass the cell voltages are now matched within 6mV due to the Q_{Loss} values being applied less to the cell with least capacity (cell 7) and applied more to the cell with the most capacity (cell 6).

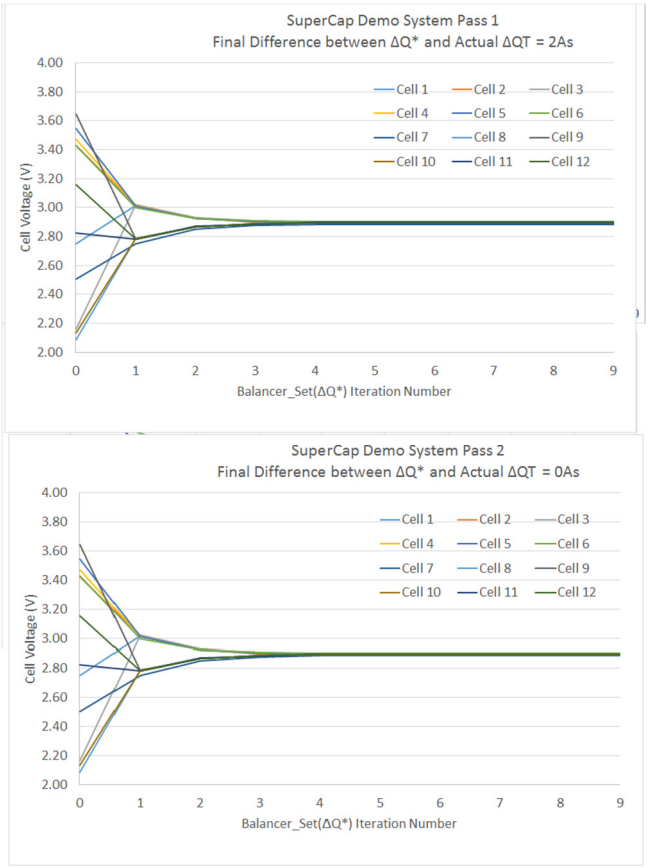


Figure 32 - SuperCap Demo Balance Simulation Pass 2

The Balancer_Set(ΔQ*) function is currently only implemented for one DC2100A board using 12 cells.