**Appendix E**

MBB Arduino Code

/\*\*

@file MBB\_Program

@par Project

@verbatim

AUV Power System

@endverbatim

@par Distribution

@verbatim

<distribution>

@endverbatim

@author Rachel Joseph and Chester Schlosser

@par Contributor(s)

@verbatim

-

@endverbatim

@version 1.0

@date March 17, 2013

@brief This is the Master Battery Brain (MBB) program for the AUV Power System

@par Modifications and Development Time-line:

@verbatim

Date Modifier's Name Modification

------------ ---------------- ----------------------------------

<Month, day, year> <name> <description of change>

<Month, day, year> <name> <description of change>

@endverbatim

@section version\_changes Changes made in this Version

@verbatim

Line Change Description

------ -------------------------------------

<line> <change description>

@endverbatim

the following are optional(you decide if they are necessary):

@details <a more in-depth description of the file/program>

@par Keywords

@verbatim

 <list, of, keywords>

@endverbatim

@par Usage:

@verbatim

<insert usage statement(s) here>

@endverbatim

@section todo\_bugs\_modification\_section Todo, Bugs, and Modifications

@bug <insert bug here>

@todo <insert future work to be done here>

\*/

////////////////////////////////////////////////////////////////////////////////////

#include <Wire.h> //Includes wire library for I2C communication

float percentBattLife[9]; /\*\* Percent battery life remaining of SBB calculated from the coulomb count \*\*/

byte data[20]; /\*\* Data is used to compile the foats once received from the slave \*\*/

long previousMillis = 0; /\*\* Stores last value of the timer that executes the serial print of all the battery's data \*\*/

long previousMillisB = 0; /\*\* Stores last value of the timer that executes the batteryBalance, batteryChoice, and switchControl functions \*\*/

float dBusITot = 0; /\*\* Dirty Bus Total Current \*\*/

float cBusITot = 0; /\*\* Clean Bus Total Current \*\*/

int dBusNumBattsFinal = 0; /\*\* Number of batteries on the clean bus \*\*/

int cBusNumBattsFinal = 0; /\*\* Number of batteries on the dirty bus \*\*/

int cBusNumBattsEcho = 0; /\*\* Number of batteries/bus actually echoed from SBB \*\*/

int dBusNumBattsEcho = 0; /\*\* Number of batteries/bus actually echoed from SBB \*\*/

boolean batteryAvailable[9] = {1,1,0,0,0,0,0,0,0}; /\*\* This array stores battery status, if drained or not ( 0 = drained, 1 = available) \*\*/

int batteryBusState[9][3] = {0}; /\*\* Stores bus location.

 [i][0] stores echo from SBB of battery actual bus status ( 3=Off, 1=clean bus, 2 = dirty bus ),

 [i][1] stores the desired bus state of the battery ( 3=Off, 1=clean bus, 2 = dirty bus )

 [i][2] stores the bus state to be transmitted, will be checked for minimum battery requirements \*\*/

float SBBBattValues[9][4] = {0}; /\*\* Stores values from SBB about battery power.

 [i][0] Stores PSB Voltage

 [i][1] Stores PSB Current

 [i][2] Stores Batt Coulomb

 [i][3] Stores battery voltage before PSB \*\*/

boolean batteryDepleted[9]; /\*\* [i] Stores Battery Depletion State, 1 = Depleted, 0 = Good \*\*/

boolean SBBCommState[9]; /\*\* [i] Stores Communication State, 0 = Good, everything else = bad \*\*/

const int MyAddress = 10; /\*\* I2C Address for MBB \*\*/

int MBB\_1\_Status; /\*\* Status bit for MBB\_1, 1 when functioning properly and 0 when comm or power failure \*\*/

int MBB\_2\_Status; /\*\* Status bit for MBB\_2, 1 when functioning properly and 0 when comm or power failure \*\*/

int comFailCount1 = 0; /\*\* Counter in that increments when a redundant MBB communication attempt fails \*\*/

int comFailCount2 = 0; /\*\* Counter in that increments when a redundant MBB 2 communication attempt fails \*\*/

/\*\*

 \* @author Rachel Joseph

 \* @brief Main setup function for the MBB program

 \* @param -

 \* @param -

 \* @returns -

 \* @returns -

 \*\*/

void setup()

{

 Wire.begin(MyAddress); //! Joins the I2C bus

 Serial.begin(9600); //! Starts serial for output

 //! Initializes all batteries to input state 3, CPSB and DPSB both off

 for (int i = 0; i < 9; i++)

 {

 batteryBusState[i][0] = 3;

 batteryBusState[i][1] = 3;

 batteryBusState[i][2] = 3;

 }

}

/\*\*

 \* @author Rachel Joseph and Chester Schlosser

 \* @brief Main loop function of program. Depending on the address of the MBB, it executes either the MBB\_1\_Master, MBB\_2\_Master, or MBB\_3\_Master program

 \* @param -

 \* @param -

 \* @returns -

 \* @returns -

 \*\*/

void loop()

{

 //! If MyAddress is 10, then MBB1 is main master and the MBB\_1\_Master function will be executed.

 if (MyAddress == 10)

 {

 MBB\_1\_Master();

 }

 //! If MyAddress is 11, them MBB2 is main master and the MBB\_2\_Master function will be executed.

 if (MyAddress == 11)

 {

 MBB\_2\_Master();

 }

 //! If MyAddress is 12,then MBB3 is main master and the MBB\_3\_Master function will be executed.

 if (MyAddress == 12)

 {

 MBB\_3\_Master();

 }

}

/\*\*

 \* @author Rachel Joseph and Chester Schlosser

 \* @brief RequestEvent executes every time the MBB requests data from the SBBs.

 \* @param Address - indicated with SBB the MBB would like to receive data from

 \* @param dataSize - indicated how many bytes of data the MBB is expecting to receive from the SBB

 \* @returns -

 \* @returns -

 \*\*/

void requestEvent(int address, int dataSize)

{

 // Declare variables

 float tempFloatA = 0.0;

 float tempFloatB = 0.0;

 float tempFloatC = 0.0;

 float tempFloatD = 0.0;

 int tempIntA = 0;

 int tempIntB = 0;

 /\*\* Sends a request to the SBBs for 20 bits of data consisting of the bus voltage,

 current, coulomb count, battery voltage, echoed battery inputState, and battery status and then stores the data \*\*/

 Wire.requestFrom(address, dataSize);

 if (Wire.available())

 {

 int i = 0;

 while( Wire.available() )

 {

 data[i] = Wire.read();

 i = i + 1;

 }

 //Stores the received SBB PSB Voltage using a union

 union tempFloatA\_tag {byte tempFloatA\_b[4]; float tempFloatA\_fval;} tempFloatA\_Union;

 tempFloatA\_Union.tempFloatA\_b[0] = data[0];

 tempFloatA\_Union.tempFloatA\_b[1] = data[1];

 tempFloatA\_Union.tempFloatA\_b[2] = data[2];

 tempFloatA\_Union.tempFloatA\_b[3] = data[3];

 tempFloatA = tempFloatA\_Union.tempFloatA\_fval;

 SBBBattValues[address-1][0] = tempFloatA;

 //Stores the received SBB Current using a union

 union tempFloatB\_tag {byte tempFloatB\_b[4]; float tempFloatB\_fval;} tempFloatB\_Union;

 tempFloatB\_Union.tempFloatB\_b[0] = data[4];

 tempFloatB\_Union.tempFloatB\_b[1] = data[5];

 tempFloatB\_Union.tempFloatB\_b[2] = data[6];

 tempFloatB\_Union.tempFloatB\_b[3] = data[7];

 tempFloatB = tempFloatB\_Union.tempFloatB\_fval;

 SBBBattValues[address-1][1] = tempFloatB;

 //Stores the received SBB Coulomb Count using a union

 union tempFloatC\_tag {byte tempFloatC\_b[4]; float tempFloatC\_fval;} tempFloatC\_Union;

 tempFloatC\_Union.tempFloatC\_b[0] = data[8];

 tempFloatC\_Union.tempFloatC\_b[1] = data[9];

 tempFloatC\_Union.tempFloatC\_b[2] = data[10];

 tempFloatC\_Union.tempFloatC\_b[3] = data[11];

 tempFloatC = tempFloatC\_Union.tempFloatC\_fval;

 SBBBattValues[address-1][2] = tempFloatC;

 //Stores the received SBB Voltage using a union

 union tempFloatD\_tag {byte tempFloatD\_b[4]; float tempFloatD\_fval;} tempFloatD\_Union;

 tempFloatD\_Union.tempFloatD\_b[0] = data[12];

 tempFloatD\_Union.tempFloatD\_b[1] = data[13];

 tempFloatD\_Union.tempFloatD\_b[2] = data[14];

 tempFloatD\_Union.tempFloatD\_b[3] = data[15];

 tempFloatD = tempFloatD\_Union.tempFloatD\_fval;

 SBBBattValues[address-1][3] = tempFloatD;

 //Stores the received SBB PSB echoed Input State using a union

 union tempIntA\_tag {byte tempIntA\_b[2]; int tempIntA\_fval;} tempIntA\_Union;

 tempIntA\_Union.tempIntA\_b[0] = data[16];

 tempIntA\_Union.tempIntA\_b[1] = data[17];

 tempIntA = tempIntA\_Union.tempIntA\_fval;

 batteryBusState[address-1][0] = tempIntA;

 //Stores the received SBB depletion state

 union tempIntB\_tag {byte tempIntB\_b[2]; int tempIntB\_fval;} tempIntB\_Union;

 tempIntB\_Union.tempIntB\_b[0] = data[18];

 tempIntB\_Union.tempIntB\_b[1] = data[19];

 tempIntB = tempIntB\_Union.tempIntB\_fval;

 if( tempIntB == 0 )

 batteryDepleted[address-1] = 0;

 else if( tempIntB == 1 )

 batteryDepleted[address-1] = 1;

 }

}

/\*\*

 \* @author Chester Schlosser

 \* @brief BatteryBalance balances the current on each battery to keep ideal current on every battery for maximum life. It calculates the total number of batteries idealy ran on each bus

 \* @param -

 \* @param -

 \* @returns -

 \* @returns -

 \*\*/

void batteryBalance()

{

 //! Current on every battery has an ideal value for maximum efficency

 //! Current will drive equation to decide how many batteries should be switched to each bus

 // Variables

 // You can change IdealBattI to whatever current you want the MBB to start adding more batteries to the bus

 int idealBattI = 5; /\*\* CMS - Ideal draw in amps, purely a guess \*\*/

 int ActiveBattsNum = 0; /\*\* Total number of batteries we can use \*\*/

 int cBusNumBattsRaw = 0; /\*\* The number of batteries on the clean bus \*\*/

 int dBusNumBattsRaw = 0; /\*\* The number of batteries on the dirty buss \*\*/

 float cdBusRatio = 0; /\*\* The ration of the number of batteries on the clean and dirty buss \*\*/

 //! Resets ActiveBattSNum each time before calculating

 ActiveBattsNum = 0;

 for( int i = 0; i < 9; i++ )

 {

 if( batteryAvailable[i] == 1 )

 ActiveBattsNum++;

 }

 /\*\* Calculates the number of batteries that will occur for ideal current.

 Add 0.5 in order to round up if truncation occurs \*\*/

 cBusNumBattsRaw = cBusITot/idealBattI + 0.5;

 dBusNumBattsRaw = dBusITot/idealBattI + 0.5;

 //! Creates ratio to decide batteries/bus if NumBattsRaw > ActiveBattNum

 if( ( cBusNumBattsRaw+dBusNumBattsRaw ) > ActiveBattsNum )

 {

 cdBusRatio = cBusNumBattsRaw/( cBusNumBattsRaw+dBusNumBattsRaw );

 cBusNumBattsFinal = cdBusRatio\*ActiveBattsNum; // Truncated down to prefer the dirty bus

 //! Checks to be sure the calculations did not assign 0 batteries to clean bus on accident

 if( cBusNumBattsFinal < 1 )

 cBusNumBattsFinal = 1;

 dBusNumBattsFinal = ActiveBattsNum - cBusNumBattsFinal;

 if( dBusNumBattsFinal < 1 )

 {

 dBusNumBattsFinal = 1;

 cBusNumBattsFinal--;

 }

 }

 else

 {

 if( cBusNumBattsRaw < 1 )

 {

 cBusNumBattsFinal = 1;

 }

 else

 {

 cBusNumBattsFinal = cBusNumBattsRaw;

 }

 if( dBusNumBattsRaw < 1 )

 {

 dBusNumBattsFinal = 1;

 }

 else

 {

 dBusNumBattsFinal = dBusNumBattsRaw;

 }

 if( ( cBusNumBattsFinal+dBusNumBattsFinal) > ActiveBattsNum )

 {

 if( cBusNumBattsFinal > 1 )

 cBusNumBattsFinal--;

 else if( dBusNumBattsFinal > 1 )

 dBusNumBattsFinal--;

 }

 }

}

/\*\*

 \* @author Chester Schlosser

 \* @brief BatteryChoice() chooses on which bus each battery is placed

 \* @param -

 \* @param -

 \* @returns -

 \* @returns -

 \*\*/

void batteryChoice()

{

 // Variables

 int cBusNumBattsTemp = 0; //! Temporary number of batteries on the clean bus

 int dBusNumBattsTemp = 0; //! Temporary number of batteries on the dirty bus

 // Reset variables

 cBusNumBattsTemp = 0;

 dBusNumBattsTemp = 0;

 //! Clears out old battery bus requests

 for( int j = 0; j < 9; j++ )

 {

 batteryBusState[j][1] = 3;

 }

 //! Searches through available batteries and decide best options

 for( int i = 0; i < 9; i++ )

 {

 if( batteryAvailable[i] == 1 )

 {

 //! Clean bus has preferenance to be filled first

 if ( ( cBusNumBattsFinal-cBusNumBattsTemp ) > 0 )

 {

 batteryBusState[i][1] = 1;

 cBusNumBattsTemp++;

 }

 else if( ( dBusNumBattsFinal-dBusNumBattsTemp ) > 0 )

 {

 batteryBusState[i][1] = 2;

 dBusNumBattsTemp++;

 }

 }

 }

}

/\*\*

 \* @author Rachel Joseph

 \* @brief Tests functionality of MBB\_1 and MBB\_2

 \* @param -

 \* @param -

 \* @returns -

 \* @returns -

 \*\*/

 void testMainMaster(int address)

 {

 /\*\* If MBB\_1 or MBB\_2 experiences a comm/power loss, their status bit becomes 0 and one of the redundant masters takes over

 as main master until the origional main master comes back on line \*\*/

 Wire.beginTransmission(address);

 if (address == 10)

 {

 if(Wire.endTransmission() == 0)

 {

 comFailCount1 = 0; //! Reset fail count if a successful com occurs

 MBB\_1\_Status = 1;

 }

 else

 {

 comFailCount1 = comFailCount1 + 1; //! Increase fail count if a com error occurs

 MBB\_1\_Status = 0;

 }

 }

 if (address == 11)

 {

 if(Wire.endTransmission() == 0)

 {

 comFailCount2 = 0; //! Reset fail count if a successful com occurs

 MBB\_2\_Status = 1;

 }

 else

 {

 comFailCount2 = comFailCount2 + 1; //! Increase fail count if a com error occurs

 MBB\_2\_Status = 0;

 }

 }

}

/\*\*

 \* @author Rachel Joseph and Chester Schlosser

 \* @brief Main\_Master Program

 \* @param -

 \* @param -

 \* @returns -

 \* @returns -

 \*\*/

void Main\_Master()

{

 //! Sends the requested input state to each SBB

 for (int i = 0; i < 9; i++)

 {

 Wire.beginTransmission(i+1); // Shift +1 because storage is 0 Based

 Wire.write(batteryBusState[i][2]);

 SBBCommState[i] = Wire.endTransmission();

 }

 //! Tests every battery for depletion or communication loss. If either has occured make it unavailable for use; dont put the battery into cycling.

 for( int i = 0; i < 9; i++ )

 {

 if( batteryDepleted[i] == 0 && SBBCommState[i] == 0 )

 batteryAvailable[i] = 1;

 else

 batteryAvailable[i] = 0;

 }

 //! If state communication transmission is successful, request data from all nine batteries.

 for( int i = 0; i < 9; i++ )

 if( SBBCommState[i] == 0 )

 requestEvent( i+1, 20 ); // Shift for 0 base

 //! Calculate % battery life remaining for every battery

 for( int i = 0; i < 9; i++ )

 {

 if( batteryAvailable[i] == 1 )

 percentBattLife[i] = ( (288000 - SBBBattValues[i][2]) / 288000) \* 100;

 else

 percentBattLife[i] = 0; // Assign to 0 for when calculating total life

 }

 //! Calculates total current loads. Reset both current sums to prevent accumulation from last scan

 dBusITot = 0;

 cBusITot = 0;

 //! Runs a loop that checks if a battery is charged. If it is charged, the program checks which bus the battery is on and adds its current to the appropriate current sum

 for( int i = 0; i < 9; i++ )

 {

 if( batteryAvailable[i] != 0 )

 {

 if( batteryBusState[i][1] == 1 )

 {

 cBusITot += SBBBattValues[i][1];

 }

 else if( batteryBusState[i][1] == 2 )

 {

 dBusITot += SBBBattValues[i][1];

 }

 }

 }

 // Calcultate number of batteries on every bus

 // Reset Count

 cBusNumBattsEcho = 0;

 dBusNumBattsEcho = 0;

 // Run loop, add each battery to its current bus count

 for( int j = 0; j < 9; j++ )

 {

 if( batteryBusState[j][0] == 1 )

 cBusNumBattsEcho++;

 else if( batteryBusState[j][0] == 2 )

 dBusNumBattsEcho++;

 }

 unsigned long currentMillisB = millis();

 if( currentMillisB - previousMillisB < 0 )

 {

 previousMillisB = 0;

 }

 if ( currentMillisB-previousMillisB > 500 )

 {

 // save the last print time

 previousMillisB = currentMillisB;

 batteryBalance();

 batteryChoice();

 switchControl();

 }

 // Print Serial Data

 unsigned long currentMillis = millis();

 if( currentMillis - previousMillis < 0 )

 {

 previousMillis = 0;

 }

 if ( currentMillis - previousMillis > 2000 )

 {

 // save the last print time

 previousMillis = currentMillis;

 Serial.println("Batt# Bus BusV I Coulombs %Life BattV Available Desired Transmitted");

 for( int i = 0; i < 9; i++ )

 {

 Serial.print(i+1);

 Serial.print(" ");

 Serial.print(batteryBusState[i][0]);

 Serial.print(" ");

 Serial.print(SBBBattValues[i][0], 2);

 Serial.print("V ");

 Serial.print(SBBBattValues[i][1], 2);

 Serial.print("A ");

 Serial.print(SBBBattValues[i][2], 1);

 Serial.print(" ");

 Serial.print(percentBattLife[i], 1);

 Serial.print("% ");

 Serial.print(SBBBattValues[i][3], 2);

 Serial.print("V ");

 Serial.print(batteryAvailable[i]);

 Serial.print(" ");

 Serial.print(batteryBusState[i][0]);

 Serial.print(" ");

 Serial.print(batteryBusState[i][1]);

 Serial.print(" ");

 Serial.print(batteryBusState[i][2]);

 Serial.println(" ");

 }

 /\*\*Prints total clean bus current, total dirty bus current,

 total number of batteries on the clean bus, and total number of batteries on the dirty bus for testing purposes \*\*/

 Serial.print("Total Clean Bus Current: ");

 Serial.println(cBusITot);

 Serial.print("Total Dirty Bus Current: ");

 Serial.println(dBusITot);

 Serial.print("Number of Batteries on the Clean Bus: ");

 Serial.println(cBusNumBattsFinal);

 Serial.print("Number of Batteries on the Dirty Bus: ");

 Serial.println(dBusNumBattsFinal);

 Serial.println();

 Serial.println();

 }

}

/\*\*

 \* @author Rachel Joseph

 \* @brief MBB\_1 Main Program - This code is executed if MBB\_1 is the main master

 \* @param -

 \* @param -

 \* @returns -

 \* @returns -

 \*\*/

 void MBB\_1\_Master()

 {

 //! When MBB1 is main master, run the Main\_Master() function.

 Main\_Master();

 }

 /\*\*

 \* @author Rachel Joseph and Chester Schlosser

 \* @brief MBB\_2 Main Program - This code is executed if MBB\_2 is the main master

 \* @param -

 \* @param -

 \* @returns -

 \* @returns -

 \*\*/

 void MBB\_2\_Master()

 {

 testMainMaster(10); //! Tests the functionality of MBB\_1

 //! If MBB\_1\_Status isn't 1, MBB2 takes over as Main Master and the Main\_Master() function is executed.

 //! The MBB2 can only take over if a comm attempt has failed 5 times

 if( (MBB\_1\_Status != 1) && (comFailCount1 > 5) )

 {

 Main\_Master();

 }

 else

 delay(3000);

 }

/\*\*

 \* @author Rachel Joseph and Chester Schlosser

 \* @brief MBB\_3 Main Program - This code is executed if MBB\_3 is the main master

 \* @param -

 \* @param -

 \* @returns -

 \* @returns -

 \*\*/

 void MBB\_3\_Master()

 {

 testMainMaster(10); //! Tests the functionality of MBB\_1

 testMainMaster(11); //! Tests the functionality of MBB\_2

 //! If MBB\_1\_Status and MBB\_2\_Status isn't 1, then MBB\_3 takes over as main master and the Main\_Master() function is executed.

 //! The MBB2 can only take over if a comm attempt has failed 5 times for both of the other MBBs

 if( (MBB\_1\_Status != 1) && (MBB\_2\_Status != 1) && (comFailCount1 > 5) && (comFailCount2 > 5) )

 {

 Main\_Master();

 }

 else

 delay(3000);

 }

 /\*\*

 \* @author Chester Schlosser

 \* @brief SwitchControl() requests switching in the correct order and prevents 0 batteries on a bus while switching

 \* @param -

 \* @param -

 \* @returns -

 \* @returns -

 \*\*/

void switchControl()

{

 // Variables

 int cBattsTemp = 0;

 int dBattsTemp = 0;

 //! Scans through batteries and check that none are in temporary switch state

 for( int i = 0; i < 9; i++ )

 {

 //! If battery echo is not the same as the last switch request, exit the switching loop

 if( batteryBusState[i][0] != batteryBusState[i][2] && batteryAvailable[i] == 1 )

 return;

 }

 cBattsTemp = cBusNumBattsEcho;

 dBattsTemp = dBusNumBattsEcho;

 for( int i = 0; i < 9; i++ )

 {

 if( batteryBusState[i][0] != batteryBusState[i][1] )

 {

 // If the echo state is clean bus

 if( batteryBusState[i][0] == 1 )

 {

 // Checks if it is safe before switching and verifies one batt will remain on the bus, and current will not be maxed

 if( cBattsTemp > 1 && ( ( cBattsTemp-1 ) \*20 > cBusITot ) )

 {

 // If one batt will remain on bus, switch

 batteryBusState[i][2] = batteryBusState[i][1];

 cBattsTemp--;

 }

 else

 {

 // If reassign is not allowed, wait till next scan

 batteryBusState[i][2] = batteryBusState[i][0];

 }

 }

 // Else if echo state is dirty bus

 else if( batteryBusState[i][0] == 2 )

 {

 // Check one batt will remain, and current will not be maxed

 if( dBattsTemp > 1 && ( ( dBattsTemp-1 ) \*20 > dBusITot ) )

 {

 // If one batt will remain on bus, switch

 batteryBusState[i][2] = batteryBusState[i][1];

 dBattsTemp--;

 }

 else

 {

 // If reassign is not allowed, wait till next scan

 batteryBusState[i][2] = batteryBusState[i][0];

 }

 }

 // If the current battery is not assigned, just switch it

 else if( batteryBusState[i][0] == 3 )

 {

 batteryBusState[i][2] = batteryBusState[i][1];

 }

 else

 {

 batteryBusState[i][2] = batteryBusState[i][0];

 }

 }

 else

 {

 batteryBusState[i][2] = batteryBusState[i][0];

 }

 //! Check to be sure echoed state is usable, otherwise wont fall under clean bus/ dirty bus

 //switching control, will just get stuck in a loop

 if( batteryBusState[i][0] != 1 && batteryBusState[i][0] != 2 && batteryBusState[i][0] != 3 )

 batteryBusState[i][2] = 3;

 }

 return;

}

/\*\*

 \* @author Chester Schlosser

 \* @brief CheckSwitch() decides if a switch can occur. Checks min batts and current

 \* @param -

 \* @param -

 \* @returns -

 \* @returns -

 \*\*/

void checkSwitch()

{

 int i = 0;

 return;

}