#include <stdlib.h>

#include "definitions.h"

#include "function\_prototypes.h"

#include "peripheral\_registers.h"

#include "data.h"

#include "math.h"

//#include "sineLUT.h"

unsigned char \* nextSCITXAddress(unsigned char \*);

enum STATES handleNoState(void);

enum STATES handleOpenState(void);

enum STATES handleCloseState(void);

enum STATES handleFloatState(void);

enum STATES handleTrippingState(void);

void saveCalibrationConstants(void);

void passCalibrationPacketOn(unsigned char \*);

int generateNoiseLevel(const struct SINE\_WAVE\_DEF PhSignal);

//some delay function

delay(unsigned length)

{

while (length != 0)

{

timerC1\_counter = 0x1D4;

timerC1\_control.bit.count\_mode = 1;

while (timerC1\_status.bit.TCF == FALSE)

{

};

timerC1\_status.bit.TCF = FALSE;

--length;

}

}

send\_msg(char \*ptr)

{

while (\*ptr != 0)

{

\*txQ\_in = \*ptr; ++ptr;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

}

\*txQ\_in = 0x0A;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = 0x0D;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

SCI0\_control\_reg.bit.TEIE = 1; /\* Enable transmitter ready intrpt \*/

}

//Function for sending test data back

test\_msg(char \*ptr, uns input)

{

char index, hex\_number;

while (\*ptr != 0)

{

\*txQ\_in = \*ptr; ++ptr;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

}

\*txQ\_in = ' ';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = '=';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = ' ';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

for (index = 0; index <= 3; ++index)

{

hex\_number = (char)(input >> (12 - (4 \* index))) ANDb 0xF;

if ((hex\_number >= 0) AND hex\_number <= 9)

hex\_number += 0x30;

else hex\_number += 0x37;

\*txQ\_in = hex\_number;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

}

\*txQ\_in = 0x0A;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = 0x0D;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

SCI0\_control\_reg.bit.TEIE = 1; /\* Enable transmitter ready intrpt \*/

}

test\_msgl(char \*ptr, long input)

{

char index, hex\_number;

while (\*ptr != 0)

{

\*txQ\_in = \*ptr; ++ptr;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

}

\*txQ\_in = ' ';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = '=';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = ' ';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

for (index = 0; index <= 7; ++index)

{

hex\_number = (char)(input >> (28 - (4 \* index))) ANDb 0xF;

if ((hex\_number >= 0) AND hex\_number <= 9)

hex\_number += 0x30;

else hex\_number += 0x37;

\*txQ\_in = hex\_number;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

}

\*txQ\_in = 0x0A;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = 0x0D;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

SCI0\_control\_reg.bit.TEIE = 1; /\* Enable transmitter ready intrpt \*/

}

test\_msgd(char \*ptr, int input)

{

char index, hex\_number;

int accum;

while (\*ptr != 0)

{

\*txQ\_in = \*ptr; ++ptr;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

}

\*txQ\_in = ' ';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = '=';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = ' ';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

if (input < 0)

{

\*txQ\_in = '-';

input = -input;

}

else \*txQ\_in = '+';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

accum = input/10000; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

accum = input % 10000; accum = accum/1000; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

accum = input % 1000; accum = accum/100; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

accum = input % 100; accum = accum/10; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

accum = input % 10; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = ' ';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = 0x0A;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = 0x0D;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

SCI0\_control\_reg.bit.TEIE = 1; /\* Enable transmitter ready intrpt \*/

}

test\_msgdl(char \*ptr, long input)

{

char index, hex\_number;

long accum;

while (\*ptr != 0)

{

\*txQ\_in = \*ptr; ++ptr;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

}

\*txQ\_in = ' ';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = '=';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = ' ';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

if (input < 0)

{

\*txQ\_in = '-';

input = -input;

}

else \*txQ\_in = '+';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

accum = input/100000000; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

input = input % 100000000;

accum = input/10000000; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

input = input % 10000000;

accum = input/1000000; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

input = input % 1000000;

accum = input/100000; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

input = input % 100000;

accum = input/10000; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

accum = input % 10000; accum = accum/1000; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

accum = input % 1000; accum = accum/100; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

accum = input % 100; accum = accum/10; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

accum = input % 10; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = ' ';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = 0x0A;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = 0x0D;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

SCI0\_control\_reg.bit.TEIE = 1; /\* Enable transmitter ready intrpt \*/

}

test\_msgud(char \*ptr, uns input)

{

char index, hex\_number;

uns accum;

while (\*ptr != 0)

{

\*txQ\_in = \*ptr; ++ptr;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

}

\*txQ\_in = ' ';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = '=';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = ' ';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

accum = input/10000; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

accum = input % 10000; accum = accum/1000; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

accum = input % 1000; accum = accum/100; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

accum = input % 100; accum = accum/10; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

accum = input % 10; \*txQ\_in = (uns)accum + 0x30;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = ' ';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = 0x0A;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = 0x0D;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

SCI0\_control\_reg.bit.TEIE = 1; /\* Enable transmitter ready intrpt \*/

}

test\_msgfloat(char \*ptr, float input)

{

uns index, hex\_number;

while (\*ptr != 0)

{

\*txQ\_in = \*ptr; ++ptr;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

}

\*txQ\_in = ' ';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = '=';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = ' ';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

if (input < 0)

{

\*txQ\_in = '-';

input = -input;

}

else \*txQ\_in = '+';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

hex\_number = (uns)(input)/100000;

if (hex\_number != 0)

{

\*txQ\_in = (uns)(hex\_number + 0x30);

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

}

input = input - ((float)hex\_number \* 100000);

hex\_number = (uns)(input)/10000;

if (hex\_number != 0)

{

\*txQ\_in = (uns)(hex\_number + 0x30);

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

}

input = input - ((float)hex\_number \* 10000);

hex\_number = (uns)(input)/1000;

if (hex\_number != 0)

{

\*txQ\_in = (uns)(hex\_number + 0x30);

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

}

input = input - ((float)hex\_number \* 1000);

hex\_number = (uns)(input)/100;

if (hex\_number != 0)

{

\*txQ\_in = (uns)(hex\_number + 0x30);

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

}

input = input - ((float)hex\_number \* 100);

hex\_number = (uns)(input)/10;

if (hex\_number != 0)

{

\*txQ\_in = (uns)(hex\_number + 0x30);

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

}

input = input - ((float)hex\_number \* 10);

hex\_number = (uns)(input);

\*txQ\_in = (uns)(hex\_number + 0x30);

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = '.';

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

input = input - (float)hex\_number;

input = input \* 10;

hex\_number = (uns)(input);

\*txQ\_in = (uns)(hex\_number + 0x30);

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

input = input - (float)hex\_number;

input = input \* 10;

hex\_number = (uns)(input);

\*txQ\_in = (uns)(hex\_number + 0x30);

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = 0x0D;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

\*txQ\_in = 0x0A;

if (txQ\_in >= &txQ[txQ\_length - 1]) txQ\_in = &txQ[0];

else ++txQ\_in;

SCI0\_control\_reg.bit.TEIE = 1; /\* Enable transmitter ready intrpt \*/

}

ftest\_msg(char \*header\_string, float float\_data)

{

int len, l1, l2;

char \*tempQ\_in = &tempQ[0];

len = sprintf(&tempQ[0], "%s = %+f", header\_string, float\_data);

/\*

if ((txQ\_in + len) < (&txQ + txQ\_length))

\*/

/\*

l1 = (int)txQ\_in +len;

l2 = (int)(&txQ) +txQ\_length;

if (l1 < l2)

\*/

if (((int)txQ\_in + len) < ((int)(&txQ) + txQ\_length))

{

while (len != 0)

{

\*txQ\_in = \*tempQ\_in;

++txQ\_in; ++tempQ\_in; --len;

}

}

SCI0\_control\_reg.bit.TEIE = 1; /\* Enable transmitter ready intrpt \*/

}

/\*Description:

This function handles the multiplication for my fixed point scheme. Because

the numbers do not represent integers, but fractions, the shift is required in

order to return the right result.

0x40 = 1

LSB value = 0.015625

Max Pos Value = 511

Max Neg Value = -512

Inputs:

Two ints of the fixed point algorithm.

Outputs:

An int of the fixed point algorithm.

\*/

int fp\_mult(int multiplicand, int multiplier)

{

long result;

int iresult;

result = (long)multiplicand \* (long)multiplier;

result >>= 6;

iresult = (int)result;

if((multiplicand > 0 && multiplier > 0) || (multiplicand < 0 && multiplier < 0))

{

if (iresult < 0)

iresult = 0x7FFF;

}

else

{

if (iresult > 0)

iresult = 0x8000;

}

return iresult;

}

int MULT2\_14fi(union FIXED2\_14 f, int i)

{

long result;

result = (long)f.Full \* (long)i;

result >>= 14;

return (int)result;

}

union FIXED2\_14 MULT2\_14ff(union FIXED2\_14 x, union FIXED2\_14 y)

{

long temp;

temp = (long)x.Full \* (long)y.Full;

temp >>= 14;

x.Full = (int)temp;

return x;

}

/\*Description:

This function handles the division for my fixed point scheme. Because

the numbers do not represent integers, but fractions, a bunch of extra steps

must be taken to return the right result.

Inputs:

Two ints of the fixed point algorithm.

Outputs:

An int of the fixed point algorithm.

\*/

int fp\_div(int dividend, int divisor)

{

long l\_dividend;

long l\_divisor;

long l\_result;

int result;

l\_dividend = (long)dividend << 16;

l\_result = l\_dividend / (long)divisor;

if (l\_result >= 4194304)

{

result = 0x7FFF;

}

else if (l\_result <= -4194304)

{

result = 0x8000;

}

else

{

l\_result >>= 10;

result = (int)l\_result;

}

return result;

}

/\*Description:

This functions controls putting the data from the serial queue into the proper

phase and current or voltage. Also handles the noise because I was too lazy to

create a completely different structure which would change from my original

idea. For testing purposes, this is fine.

Inputs:

The pointer to the the pointer to the begining of the data in the Rx queue. The

data following this opcode is fixed and we will read the next 9 (including what

is pointed to) data addresses.

Outputs:

None, but information is stored in the proper Phase and Current or Voltage

\*/

void store\_phase\_data(uns \*input\_message\_pointer, uns \*queueAddress)

{

struct PHASE\_SIGNALS \*phase\_pointer;

struct SINE\_WAVE\_DEF \*sineWaveDefinitionPointer;

//two variables for test

uns valuesSetRight = 0;

uns \*originalPointer;

unsigned char i;

unsigned char \*testPtr;

unsigned char testArray[10];

unsigned char \*\*testPtr2;

unsigned char ptrtest1, ptrtest2, \*testaddress1, \*testaddress2, \*testaddress3, \*testaddress4, \*testaddress5;

uns errorTest;

//unsigned long offsetf, offsettest1, offsettest2;

input\_message\_pointer = nextRXQueueAddress(input\_message\_pointer, queueAddress);

if(\*input\_message\_pointer > 0)

{

passPacketOn(input\_message\_pointer, queueAddress);

return;

}

phase\_pointer = &TempPhase;

input\_message\_pointer = nextRXQueueAddress(input\_message\_pointer, queueAddress);

switch (\*input\_message\_pointer)

{

case 84: //'T') //Transformer Voltage

originalPointer = input\_message\_pointer;

sineWaveDefinitionPointer = &(phase\_pointer->Vt);

break;

case 73: //'I') //same as above, but for the current part of the phase

originalPointer = input\_message\_pointer;

sineWaveDefinitionPointer = &(phase\_pointer->I);

break;

case 82: //'R') //this is when the (R)andom noise information gets set.

originalPointer = input\_message\_pointer;

while (valuesSetRight == 0)

{

input\_message\_pointer = originalPointer;

testaddress1 = input\_message\_pointer;

input\_message\_pointer = nextRXQueueAddress(input\_message\_pointer, queueAddress);

NoiseObject.noiseOnOff = \*input\_message\_pointer;

testaddress2 = input\_message\_pointer;

input\_message\_pointer = nextRXQueueAddress(input\_message\_pointer, queueAddress);

NoiseObject.noise\_amp = \*input\_message\_pointer;

testaddress3 = input\_message\_pointer;

input\_message\_pointer = nextRXQueueAddress(input\_message\_pointer, queueAddress);

NoiseObject.noise\_freq = \*input\_message\_pointer;

testaddress4 = input\_message\_pointer;

if (testaddress1 != testaddress2 && testaddress2 != testaddress3 &&

testaddress3 != testaddress4)

valuesSetRight = 1;

else

valuesSetRight = 0;

}

break;

case 78: //Network Voltage

originalPointer = input\_message\_pointer;

sineWaveDefinitionPointer = &(phase\_pointer->Vn);

break;

}

while (valuesSetRight == 0)

{

uns angleLowByte, angleHighByte;

double angleTemp, angleTemp1, angleTemp2, angleTemp3;

valuesSetRight = 1;

input\_message\_pointer = originalPointer;

testaddress1 = input\_message\_pointer;

input\_message\_pointer = nextRXQueueAddress(input\_message\_pointer, queueAddress);

sineWaveDefinitionPointer->inputAmplitude.Full = \*input\_message\_pointer << 8;

input\_message\_pointer = nextRXQueueAddress(input\_message\_pointer, queueAddress);

sineWaveDefinitionPointer->inputAmplitude.Full += \*input\_message\_pointer;

sineWaveDefinitionPointer->amplitude = MULT2\_14ff(sineWaveDefinitionPointer->inputAmplitude, sineWaveDefinitionPointer->ConversionFactor);

input\_message\_pointer = nextRXQueueAddress(input\_message\_pointer, queueAddress);

angleHighByte = (uns)\*input\_message\_pointer;

input\_message\_pointer = nextRXQueueAddress(input\_message\_pointer, queueAddress);

angleLowByte = (uns)\*input\_message\_pointer;

angleTemp = angleLowByte + angleHighByte \* 256;

sineWaveDefinitionPointer->inputOffset = (unsigned int)((double)samp\_per\_cycle \* angleTemp/(double)360);

input\_message\_pointer = nextRXQueueAddress(input\_message\_pointer, queueAddress);

testaddress2 = input\_message\_pointer;

//sineWaveDefinitionPointer->rotate\_sig = \*input\_message\_pointer;

if(\*input\_message\_pointer != 0)

{

sineWaveDefinitionPointer->noiseEnabled = 1;

sineWaveDefinitionPointer->noiseShifter = \*input\_message\_pointer - 128;

}

else

{

sineWaveDefinitionPointer->noiseEnabled = 0;

sineWaveDefinitionPointer->noiseShifter = 0;

}

//sineWaveDefinitionPointer

//TEST

setHarmonicArrayData(\*sineWaveDefinitionPointer, &(sineWaveDefinitionPointer->frequencyComponents), input\_message\_pointer);

//generateHarmonicArray(PhA.I.frequencyComponents, PhA.I.harmonic\_array);

}

}

/\*Description:

This function sets the corresponding DAC output for the signal being generated.

It also handles updating the rotation

Inputs:

\*Ph\_signal - Pointer to the signal (Ph and V or I) that we are generating

DAC\_output\_select - the address that the DAC needs to select the output for the incoming signal

noise\_value - the noise to be added to the current output sample

offset\_plus - a "timer pulse" that is set to tell the function to rotate the signal one step

if its corresponding rotate\_sig is set.

Outputs:

Controls the DAC to update the

\*/

uns set\_DAC\_value(struct SINE\_WAVE\_DEF Ph\_signal, int DACCommand, int noise\_value, int offset\_plus, uns sample\_number, const int \*harmonicArray)

{

//extern sine\_LUT[samp\_per\_cycle];

int sineIndex;

int harmonicIndex;

int outputtemp;

int loutput = 0;

int testi = 0;

//Controls the rotating of the signal

if (Ph\_signal.rotate\_sig == 255 && offset\_plus == 1)

Ph\_signal.inputOffset++;

if (Ph\_signal.inputOffset == samp\_per\_cycle)

Ph\_signal.inputOffset = 0;

//adjusts the Index used for the LUT using the offset

sineIndex = sample\_number + Ph\_signal.inputOffset;

if (sineIndex >= samp\_per\_cycle)

sineIndex = sineIndex - samp\_per\_cycle;

//////add in the LUT stuff here.

//Add in Harmonic Array

//outputtemp = fp\_mult(sine\_LUT[sineIndex], Ph\_signal.amplitude);

outputtemp = MULT2\_14fi(Ph\_signal.amplitude, sine\_LUT[sineIndex]);

outputtemp = outputtemp + 0x7FFF;

//actually putting the information

sendDACData(DACCommand, outputtemp);

return Ph\_signal.inputOffset;

}

uns set\_DAC\_valueCurrent(struct SINE\_WAVE\_DEF Ph\_signal, int DACCommand, int noise\_value, int offset\_plus, uns sample\_number, const int \*harmonicArray)

{

//extern sine\_LUT[samp\_per\_cycle];

int sineIndex;

int harmonicIndex;

int outputtemp;

int loutput = 0;

int testi = 0;

//Controls the rotating of the signal

if (Ph\_signal.rotate\_sig == 255 && offset\_plus == 1)

Ph\_signal.inputOffset++;

if (Ph\_signal.inputOffset == samp\_per\_cycle)

Ph\_signal.inputOffset = 0;

//adjusts the Index used for the LUT using the offset

sineIndex = sample\_number + Ph\_signal.inputOffset;

if (sineIndex >= samp\_per\_cycle)

sineIndex = sineIndex - samp\_per\_cycle;

//////add in the LUT stuff here.

//Add in Harmonic Array

//outputtemp = fp\_mult(sine\_LUT[sineIndex], Ph\_signal.amplitude);

outputtemp = MULT2\_14fi(Ph\_signal.amplitude, sine\_LUTCurrent[sineIndex]);

outputtemp = outputtemp + 0x7FFF;

//actually putting the information

sendDACData(DACCommand, outputtemp);

return Ph\_signal.inputOffset;

}

/\*Description:

Generates a pseudo-random gaussian number in a quick way. The default algorithm, while probably had a longer

periodicity might be longer, for our purposes, it was unnecessary. Considering it was slow, I had to create

a faster one.

It is made gaussian by passing it through a low-pass filter (just an average of the last X number of samples)

Inputs:

PhA.V.noise\_amp - determines how much the noise is shifted by at the end

PhA.V.noise\_freq - how many samples to average.

Outputs:

Pseudo-random gaussian int.

\*/

int random\_number()

{

static int seed = 0x74b3;

static int random\_array[samp\_per\_cycle];

static long gaussian;

static int li = 0;

static unsigned int return\_value;

int gaussian\_output = 0;

seed = (seed >> 1) ^ (-(signed int)(seed & 1) & 0xDA71);

random\_array[li] = seed;

gaussian += random\_array[li];

li++;

if (li == NoiseObject.noise\_freq || li == samp\_per\_cycle - 1) //change two numbers dope

{

li = 0;

}

gaussian -= random\_array[li];

gaussian\_output = gaussian >> NoiseObject.noise\_amp;

//return\_value = timerC3\_counter ^ seed;

return gaussian\_output;

}

/\*Description:

Generates the Harmonic Array based on the amplitudes given in the FrequencyComponents pass to it

Inputs:

Frequency component containing the 3rd, 5th, 7th, and 9th Harmonic amplitudes.

Outputs:

Array containing the harmonic data to add to the fundamental.

\*/

void generateHarmonicArray(struct FREQUENCY\_COMPONENTS localFrequencyComponents, int \*sineArray)

{

//extern sine\_LUT[samp\_per\_cycle];

int i, ih;

int sineAdjust, sineOriginal;

for (i = 0, ih = 0; i < samp\_per\_cycle; ++i, ih += 3) //(samp\_per\_cycle / 3)

{

if(ih >= samp\_per\_cycle) ih = ih - samp\_per\_cycle;

if(i == 90)

i = 90;

sineAdjust = fp\_mult(localFrequencyComponents.thirdHarmonic, sine\_LUT[ih]);

sineOriginal = sineArray[i];

sineArray[i] = sineAdjust;

}

for (i = 0, ih = 0; i < samp\_per\_cycle; ++i, ih += 5)

{

if(ih >= samp\_per\_cycle) ih = ih - samp\_per\_cycle;

sineAdjust = fp\_mult(localFrequencyComponents.fifthHarmonic, sine\_LUT[ih]);

sineOriginal = sineArray[i];

sineArray[i] += sineAdjust;

if(sineOriginal > 0 && sineAdjust > 0 && sineArray[i] < 0)

{

sineArray[i] = 0x7FFF;

}

else if (sineOriginal < 0 && sineAdjust < 0 && sineArray[i] > 0)

{

sineArray[i] = 0x8000;

}

//sineArray[i] += fp\_mult(localFrequencyComponents.fifthHarmonic, sine\_LUT[ih]);

}

for (i = 0, ih = 0; i < samp\_per\_cycle; i++, ih += 7)

{

if(ih >= samp\_per\_cycle) ih = ih - samp\_per\_cycle;

//ih = (7 \* i)%samp\_per\_cycle;

sineAdjust = fp\_mult(localFrequencyComponents.seventhHarmonic, sine\_LUT[ih]);

sineOriginal = sineArray[i];

sineArray[i] += sineAdjust;

if(sineOriginal > 0 && sineAdjust > 0 && sineArray[i] < 0)

{

sineArray[i] = 0x7FFF;

}

else if (sineOriginal < 0 && sineAdjust < 0 && sineArray[i] > 0)

{

sineArray[i] = 0x8000;

}

//sineArray[i] += fp\_mult(localFrequencyComponents.seventhHarmonic, sine\_LUT[ih]);

}

for (i = 0, ih = 0; i < samp\_per\_cycle; i++, ih += 2)

{

if(ih >= samp\_per\_cycle) ih = ih - samp\_per\_cycle;

//ih = (9 \* i)%samp\_per\_cycle;

sineAdjust = fp\_mult(localFrequencyComponents.ninthHarmonic, sine\_LUT[ih]);

sineOriginal = sineArray[i];

sineArray[i] += sineAdjust;

if(sineOriginal > 0 && sineAdjust > 0 && sineArray[i] < 0)

{

sineArray[i] = 0x7FFF;

}

else if (sineOriginal < 0 && sineAdjust < 0 && sineArray[i] > 0)

{

sineArray[i] = 0x8000;

}

//sineArray[i] += fp\_mult(localFrequencyComponents.ninthHarmonic, sine\_LUT[ih]);

}

/\*

////TEST temp change 9th harmonic to 2nd

for (i = 0, ih = 0; i < samp\_per\_cycle; i++, ih += 9)

{

if(ih >= samp\_per\_cycle) ih = ih - samp\_per\_cycle;

//ih = (9 \* i)%samp\_per\_cycle;

sineAdjust = fp\_mult(localFrequencyComponents.ninthHarmonic, sine\_LUT[ih]);

sineOriginal = sineArray[i];

sineArray[i] += sineAdjust;

if(sineOriginal > 0 && sineAdjust > 0 && sineArray[i] < 0)

{

sineArray[i] = 0x7FFF;

}

else if (sineOriginal < 0 && sineAdjust < 0 && sineArray[i] > 0)

{

sineArray[i] = 0x8000;

}

//sineArray[i] += fp\_mult(localFrequencyComponents.ninthHarmonic, sine\_LUT[ih]);

}

\*/

}

/\*Description:

Puts Data in the queue into the proper harmonic components

Inputs:

Pointer to the pointer to the data in the queue.

Outputs:

Just updates the frequencyComponent

Notes:

Will be changed slightly to accomodate the fact that each output wave

can have a different freq component

\*/

void setHarmonicArrayData(struct SINE\_WAVE\_DEF signal, struct FREQUENCY\_COMPONENTS \*localFrequencyComponents, uns \*inputPointer)

{

if (inputPointer >= &rxQ[rxQ\_length-1])

inputPointer = &rxQ[0];

else (inputPointer)++;

localFrequencyComponents->thirdHarmonic = \*inputPointer;

if (inputPointer >= &rxQ[rxQ\_length-1])

inputPointer = &rxQ[0];

else (inputPointer)++;

localFrequencyComponents->fifthHarmonic = \*inputPointer;

if (inputPointer >= &rxQ[rxQ\_length-1])

inputPointer = &rxQ[0];

else (inputPointer)++;

localFrequencyComponents->seventhHarmonic = \*inputPointer;

if (inputPointer >= &rxQ[rxQ\_length-1])

inputPointer = &rxQ[0];

else (inputPointer)++;

localFrequencyComponents->ninthHarmonic = \*inputPointer;

generateHarmonicArray(\*localFrequencyComponents, signal.harmonic\_array);

}

/\*Description:

Adjust the timer that controls the sample speed for the sine wave generation.

It can be changed to adjust the frequency of the output waves either for testing

or to match the line frequency due to imperfections in the clocking crystal

Inputs:

Ph\_signal - the phase to adjust (each can be individually adjusted)

timerAdjustValue - how much off of the standard value to adjust the frequency.

Outputs:

None. Changes value of timing\_register in the Phase signal definition.

\*/

void adjustFrequencyTimer(struct SINE\_WAVE\_DEF Ph\_signal, uns timerAdjustValue)

{

\*Ph\_signal.timing\_register = (30000000 - timerAdjustValue)/(60\*samp\_per\_cycle);

}

/\*Description:

Translates the value in the queue for the frequency adjust function

Inputs:

pointer to the pointer to the value in the RX queue that should be adjusted.

Outputs:

None. Calls adjustFrequencyTimer.

\*/

void setFrequencyData(uns \*inputPointer)

{

long localTest;

localTest = (long)\*inputPointer \* 100000;

adjustFrequencyTimer(PhA.Vn, localTest);

}

/\*Description:

Changes the counter for the signal generation timer either plus or minus

1 percent

Inputs:

pointer to the pointer to the value in the RX queue

which should point 2, 1, or 0

Outputs:

Adjusts TimerC3\_load

\*/

void retuneChip(uns \*inputPointer)

{

signed char localValue;

localValue = \*inputPointer;

switch (localValue)

{

case 2: //reset

timerC3\_load = 30000000/(samp\_per\_cycle \* 60);

timerC2\_load = 30000000/(samp\_per\_cycle \* 60);

timerD0\_load = 30000000/(samp\_per\_cycle \* 60);

break;

case 1: //Increase Timer value increasing the period

timerC3\_load += 1;

timerC2\_load += 1;

timerD0\_load += 1;

break;

case 0: //Decrease Timer value

timerC3\_load -= 1;

timerC2\_load -= 1;

timerD0\_load -= 1;

break;

}

}

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

///Function that controls passing the packet on if the packet is not for this chip

///Simply recreates the first 2 characters of the packet, decrements the chip count variable

///and then pulls the rest of the data from the packet out of the receive queue and puts

///it in the transmit queue and then triggers a transmission

void passPacketOn(uns \*inputPointer, uns \*queueAddress)

{

char phase;

if(\*inputPointer == 1)

phase = 1;

else

phase = 0;

//Fill in the packet Data that has already been passed by the pointer

sendIPCommByte(13, phase);

sendIPCommByte('P', phase);

sendIPCommByte(0, phase);

do

{

inputPointer = nextRXQueueAddress(inputPointer, queueAddress);

sendIPCommByte(\*inputPointer, phase);

}

while(\*inputPointer != 0x14);

// DATA.C CHANGE GPIOD\_Data\_reg.bit.DB3 ^= 1;

GPIOD\_Data\_reg.bit.DB4 ^= 1;

}

void passCalibrationPacketOn(unsigned char \* packetPtr)

{

char phase;

unsigned int testCount = 0;

if(\*packetPtr == 1)

phase = 1;

else

phase = 0;

sendIPCommByte(7, phase);

sendIPCommByte('C', phase);

sendIPCommByte(0, phase);

do

{

testCount++;

packetPtr = nextRXQueueAddress(packetPtr, rxQ);

sendIPCommByte(\*packetPtr, phase);

}

while(\*packetPtr != 0x14);

GPIOD\_Data\_reg.bit.DB4 ^= 1;

}

//#pragma interrupt called

void generateDACOutputValue(const struct SINE\_WAVE\_DEF PhSignal, int DACCommand, unsigned int sampleNumber)

{

int value = 0;

int fundamentalValue = 0;

int harmonicValue = 0;

int inPhaseSampleNumber = 0;

int randomNumber = 0;

inPhaseSampleNumber = sampleNumber + PhSignal.inputOffset;

if(PhSignal.inputOffset > 0)

{

value = 5;

}

if(inPhaseSampleNumber >= samp\_per\_cycle)

inPhaseSampleNumber -= samp\_per\_cycle;

fundamentalValue = sine\_LUT[inPhaseSampleNumber];

harmonicValue = (PhSignal.harmonic\_array)[inPhaseSampleNumber];

value = MULT2\_14fi(PhSignal.amplitude, fundamentalValue);

fundamentalValue = value;

//value = fundamentalValue; //TEST

//value = value + harmonicValue;

//check for rail

if(fundamentalValue > 0 && harmonicValue > 0 && value < 0)

{

value = 0xFFFF;

}

else if (fundamentalValue < 0 && harmonicValue < 0 && value > 0)

{

value = 0x0000;

}

else

{

value += 0x8000;

}

//Add in noise if enabled

if(PhSignal.noiseEnabled)

{

randomNumber = generateNoiseLevel(PhSignal);

value = value + randomNumber;

}

sendDACData(DACCommand, value);

}

void generateDACOutputValueCurrent(const struct SINE\_WAVE\_DEF PhSignal, int DACCommand, unsigned int sampleNumber)

{

//extern sine\_LUT[samp\_per\_cycle];

int value = 0;

int fundamentalValue = 0;

int harmonicValue = 0;

int inPhaseSampleNumber = 0;

int randomNumber = 0;

inPhaseSampleNumber = sampleNumber + PhSignal.inputOffset;

if(PhSignal.inputOffset > 0)

{

value = 5;

}

if(inPhaseSampleNumber >= samp\_per\_cycle)

inPhaseSampleNumber -= samp\_per\_cycle;

fundamentalValue = sine\_LUTCurrent[inPhaseSampleNumber];

harmonicValue = (PhSignal.harmonic\_array)[inPhaseSampleNumber];

//value = fp\_mult(fundamentalValue, PhSignal.amplitude);

value = MULT2\_14fi(PhSignal.amplitude, fundamentalValue);

fundamentalValue = value;

//value = fundamentalValue; //TEST

value = value + harmonicValue;

//check for rail

if(fundamentalValue > 0 && harmonicValue > 0 && value < 0)

{

value = 0xFFFF;

}

else if (fundamentalValue < 0 && harmonicValue < 0 && value > 0)

{

value = 0x0000;

}

else

{

value += 0x8000;

}

//Add in noise if enabled

if(PhSignal.noiseEnabled)

{

randomNumber = generateNoiseLevel(PhSignal);

value = value + randomNumber;

}

sendDACData(DACCommand, value);

}

int generateNoiseLevel(const struct SINE\_WAVE\_DEF PhSignal)

{

long returnInt;

returnInt = RandomNumber;

returnInt \*= (long)PhSignal.noiseShifter;

returnInt >>= 7;

return (int)returnInt;

}

//#pragma interrupt called

void sendDACData(int command, int data)

{

while(DAC\_semaphore == 1){}

DAC\_semaphore = 1;

while(SPI0\_status\_control\_reg.bit.SPTE == 0){}

DACSync = 0;

//All data write LSB at transmit\_data reg 0

SPI0\_transmit\_data = command; //Write Data Buffer D Command

while(SPI0\_status\_control\_reg.bit.SPTE == 0){}

SPI0\_transmit\_data = data >> 8; //Data for Buffer High 16 bits

while(SPI0\_status\_control\_reg.bit.SPTE == 0){}

SPI0\_transmit\_data = data & 0xFF; //Data for Buffer low 16 bits

while(SPI0\_status\_control\_reg.bit.SPTE == 0){}

DACSync = 1;

DAC\_semaphore = 0;

}

//#pragma interrupt called

void sendIPCommByte(int data, char phase)

{

static SPI1Semaphore = 0;

static unsigned int count = 0;

///TEST////

txQ[count] = data;

count++;

if(count >= 100)

count = 0;

while(SPI1Semaphore == 1)

{}

SPI1Semaphore = 1;

while(GPIOD\_Data\_reg.bit.DB12 == 0){}

while(SPI1\_status\_control\_reg.bit.SPTE == 0){}

if(phase == 1) //PhaseB

GPIOD\_Data\_reg.bit.DB10 = 0;

else //PhaseC

GPIOD\_Data\_reg.bit.DB11 = 0;

SPI1\_transmit\_data = data;

while(SPI1\_status\_control\_reg.bit.SPTE == 0){}

//dummy = SPI1\_received\_data;

if(phase == 1) //PhaseB

GPIOD\_Data\_reg.bit.DB10 = 1;

else //PhaseC

GPIOD\_Data\_reg.bit.DB11 = 1;

SPI1Semaphore = 0;

}

unsigned int adjustedSampleNumber(unsigned int sampleNumber, unsigned int rotateAmount)

{

unsigned int result;

result = sampleNumber + rotateAmount;

if (result > samp\_per\_cycle)

result = result - samp\_per\_cycle;

return result;

}

void SendIRQA()

{

GPIOE\_Data\_reg.bit.DB8 = 1;//DATA.C CHANGE GPIOE\_Data\_reg.bit.TC0 = 1;

asm(nop);

asm(nop);

asm(nop);

asm(nop);

asm(nop);

asm(nop);

asm(nop);

asm(nop);

asm(nop);

asm(nop);

asm(nop);

asm(nop);

GPIOE\_Data\_reg.bit.DB8 = 0;//DATA.C CHANGE GPIOE\_Data\_reg.bit.TC0 = 0;

}

void SendIRQB()

{

GPIOE\_Data\_reg.bit.DB9 = 1;//D.C Change GPIOE\_Data\_reg.bit.TC1 = 1;

asm(nop);

asm(nop);

asm(nop);

GPIOE\_Data\_reg.bit.DB9 = 0; //GPIOE\_Data\_reg.bit.TC1 = 0;

}

void WriteExternalMemory(int data, int address)

{

}

int ReadExternalMemory(int address)

{

}

void CloseEvent(void)

{

\*txQ\_in = 'E';

txQ\_in = nextSCITXAddress(txQ\_in);

\*txQ\_in = 1;

txQ\_in = nextSCITXAddress(txQ\_in);

\*txQ\_in = 'C';

txQ\_in = nextSCITXAddress(txQ\_in);

\*txQ\_in = 0x0D;

txQ\_in = nextSCITXAddress(txQ\_in);

SCI0\_control\_reg.bit.TEIE = 1;

}

void TripEvent(void)

{

\*txQ\_in = 'E';

txQ\_in = nextSCITXAddress(txQ\_in);

\*txQ\_in = 1;

txQ\_in = nextSCITXAddress(txQ\_in);

\*txQ\_in = 'T';

txQ\_in = nextSCITXAddress(txQ\_in);

\*txQ\_in = 0x0D;

txQ\_in = nextSCITXAddress(txQ\_in);

SCI0\_control\_reg.bit.TEIE = 1;

}

void FloatEvent(void)

{

\*txQ\_in = 'E';

txQ\_in = nextSCITXAddress(txQ\_in);

\*txQ\_in = 1;

txQ\_in = nextSCITXAddress(txQ\_in);

\*txQ\_in = 'F';

txQ\_in = nextSCITXAddress(txQ\_in);

\*txQ\_in = 0x0D;

txQ\_in = nextSCITXAddress(txQ\_in);

SCI0\_control\_reg.bit.TEIE = 1;

}

void send\_char(char c)

{

\*txQ\_in = c;

txQ\_in = nextSCITXAddress(txQ\_in);

SCI0\_control\_reg.bit.TEIE = 1;

}

void SendHex(unsigned int i)

{

\*txQ\_in = ConvertToASCII(i >> 12);

txQ\_in = nextSCITXAddress(txQ\_in);

\*txQ\_in = ConvertToASCII(i >> 8);

txQ\_in = nextSCITXAddress(txQ\_in);

\*txQ\_in = ConvertToASCII(i >> 4);

txQ\_in = nextSCITXAddress(txQ\_in);

\*txQ\_in = ConvertToASCII(i);

txQ\_in = nextSCITXAddress(txQ\_in);

SCI0\_control\_reg.bit.TEIE = 1;

}

unsigned char ConvertToASCII(unsigned int i)

{

i = i & 0x0F;

switch(i)

{

case 0:

return 0x30;

case 1:

return 0x31;

case 2:

return 0x32;

case 3:

return 0x33;

case 4:

return 0x34;

case 5:

return 0x35;

case 6:

return 0x36;

case 7:

return 0x37;

case 8:

return 0x38;

case 9:

return 0x39;

case 10:

return 0x41;

case 11:

return 0x42;

case 12:

return 0x43;

case 13:

return 0x44;

case 14:

return 0x45;

case 15:

return 0x46;

default:

return 0x47;

}

}

unsigned char \* nextSCITXAddress(unsigned char \* ptr)

{

if (ptr >= &txQ[txQ\_length - 1]) ptr = &txQ[0];

else ++ptr;

return ptr;

}

void CalibrationRoutine(unsigned char \* packetPtr)

{

char phaseChar, commandChar;

int calAmount;

struct SINE\_WAVE\_DEF \*sineWave;

packetPtr = nextRXQueueAddress(packetPtr, rxQ);

if(\*packetPtr != 0)

{

//(\*packetPtr)--;

passCalibrationPacketOn(packetPtr);

return;

}

packetPtr = nextRXQueueAddress(packetPtr, rxQ);

commandChar = \*packetPtr;

packetPtr = nextRXQueueAddress(packetPtr, rxQ);

phaseChar = \*packetPtr;

packetPtr = nextRXQueueAddress(packetPtr, rxQ);

if(commandChar == 'S') //For Save

{

saveCalibrationConstants();

return;

}

else

{

calAmount = (int)\*packetPtr;

calAmount <<= 8;

packetPtr = nextRXQueueAddress(packetPtr, rxQ);

calAmount += (int)\*packetPtr;

switch(phaseChar)

{

case 'N': //Network

RAMParameters.NetworkCalibration.Full += calAmount;

TempPhase.Vn.ConversionFactor.Full += calAmount;

TempPhase.Vn.amplitude = MULT2\_14ff(TempPhase.Vn.ConversionFactor, TempPhase.Vn.inputAmplitude);

break;

case 'T': //Transformer

RAMParameters.TransformerCalibration.Full += calAmount;

TempPhase.Vt.ConversionFactor.Full += calAmount;

TempPhase.Vt.amplitude = MULT2\_14ff(TempPhase.Vt.ConversionFactor, TempPhase.Vt.inputAmplitude);

break;

case 'I': //Current

RAMParameters.CurrentCalibration.Full += calAmount;

TempPhase.I.ConversionFactor.Full += calAmount;

TempPhase.I.amplitude = MULT2\_14ff(TempPhase.I.ConversionFactor, TempPhase.I.inputAmplitude);

break;

case 'P':

RAMParameters.PhaseCorrection += calAmount;

TempPhase.I.offsetCorrection += calAmount;

break;

}

}

}

void saveCalibrationConstants(void)

{

//WriteRAMToFlash();

if(Master == 1)

{

sendIPCommByte(7, 0);

sendIPCommByte('C', 0);

sendIPCommByte(0, 0);

sendIPCommByte('S', 0);

sendIPCommByte(0, 0);

sendIPCommByte(0, 0);

sendIPCommByte(0, 0);

sendIPCommByte(0x14, 0);

sendIPCommByte(7, 1);

sendIPCommByte('C', 1);

sendIPCommByte(0, 1);

sendIPCommByte('S', 1);

sendIPCommByte(0, 1);

sendIPCommByte(0, 1);

sendIPCommByte(0, 1);

sendIPCommByte(0x14, 1);

}

}

void EnableLatch(void)

{

ReadyToLatch = 1;

}

void LatchData(void)

{

PhA = TempPhase;

ReadyToLatch = 0;

}