#include "adc\_logic.h"

#include "BSP/adc.h"

#include "BSP/gpio.h"

#include "BSP/system.h"

#include "BSP/timer.h"

#include "command\_processor.h"

#include "events\_process.h"

#include "math.h"

#include "pfc\_logic.h"

#include "string.h"

/\*--------------------------------------------------------------

DEFINES

--------------------------------------------------------------\*/

#define UMAX\_INITIAL\_VALUE (-1000000L) /\*\*< Initial value to calculate the maximum \*/

#define UMIN\_INITIAL\_VALUE (1000000L) /\*\*< Initial value to calculate the minimum \*/

#define PID\_LEAKAGE\_COEFFICIENT (1.0f) /\*\*< The leakage coefficient for the PIC controller. Can be 0.999f \*/

#define PID\_USE\_DIFFERENTIAL (0) /\*\*< Use differential coefficient in the PID controller \*/

#define GLOBAL\_LEAKAGE\_COEFFICIENT (0.995f) /\*\*< The leakage coefficient accumulator variables \*/

#define PERIOD\_REQUIRED (20000UL) /\*\*< The required period value in [us] \*/

#define PERIOD\_DRIFT (1000UL) /\*\*< The acceptable period drift in [us] \*/

#define MAXIMUM\_PERIOD\_DIFF (10) /\*\*< The acceptable period difference \*/

#undef PROTECTION\_ADC\_OVERLOAD\_CHECK /\*\*< Check ADC data for overload \*/

#undef PROTECTION\_OVERCURRENT\_CHECK /\*\*< Check currents \*/

#undef PROTECTION\_OVERVOLTAGE\_CHECK/\*\*< Check voltages \*/

/\*--------------------------------------------------------------

PRIVATE DATA

--------------------------------------------------------------\*/

static float K\_phase\_shift = -1.0f; /\*\*< The K coefficient for the frequency and phase shift correction \*/

static float K\_filter\_x = 0.5f; /\*\*< The K coefficient for the ADC signal shape smoothing \*/

static float K\_filter\_F = 0.8f; /\*\*< The K coefficient for the frequency measurement filter \*/

static float K\_filter\_P = 0.3f; /\*\*< The K coefficient for the period measurement filter \*/

static float diff = 0; /\*\*< The period difference (between required and measured) \*/

/\*\* A channel needs the square calculation (e.g. effective value) \*/

static const uint8\_t needSquare[] = {

0, /\*CH10 - ADC\_UCAP\*/

1, /\*CH11 - ADC\_U\_A\*/

1, /\*CH12 - ADC\_U\_B\*/

1, /\*CH13 - ADC\_U\_C\*/

0, /\*CH0 - ADC\_I\_A\*/

0, /\*CH1 - ADC\_I\_B\*/

0, /\*CH2 - ADC\_I\_C\*/

0, /\*CH3 - ADC\_I\_ET\*/

0, /\*CH5 - ADC\_I\_TEMP1\*/

0, /\*CH6 - ADC\_I\_TEMP2\*/

1, /\*CH14 - ADC\_EDC\_A\*/

1, /\*CH15 - ADC\_EDC\_B\*/

1, /\* - \*/

1, /\* - \*/

1, /\* - \*/

1, /\* - \*/

1 /\* - \*/

};

static uint16\_t adc\_dma\_buffer[ADC\_CHANNEL\_NUMBER]; /\*\*< The buffer for the DMA ADC data \*/

static float adc\_values[ADC\_CHANNEL\_NUMBER + ADC\_MATH\_NUMBER]; /\*\*< Temporary storage of the ADC data \*/

static uint16\_t adc\_values\_raw[ADC\_CHANNEL\_NUMBER]; /\*\*< Temporary storage of the ADC data (raw values) \*/

static uint8\_t current\_buffer = 0; /\*\*< Current buffer ID (for double buffering) \*/

static uint8\_t last\_buffer = 0; /\*\*< Last buffer ID (for double buffering) \*/

static uint16\_t symbol = 0; /\*\*< The current position in the buffer \*/

static uint8\_t new\_period = 0; /\*\*< A new period measurement has been started \*/

static float VLet\_1 = 0; /\*\*< Last value for the VL error \*/

static float VLIt\_1 = 0; /\*\*< Last value for the VL integral part \*/

static float Ia\_e\_1 = 0; /\*\*< Last value for the Ia error \*/

static float Ia\_It\_1 = 0; /\*\*< Last value for the Ia integral part \*/

static float Ib\_e\_1 = 0; /\*\*< Last value for the Ib error \*/

static float Ib\_It\_1 = 0; /\*\*< Last value for the Ib integral part \*/

static float Ic\_e\_1 = 0; /\*\*< Last value for the Ic error \*/

static float Ic\_It\_1 = 0; /\*\*< Last value for the Ic integral part \*/

static float Kp = 0.5f; // Пропорциональный коэффициент

static float Ki = 0.01f; /\*\*< The integral coefficient, TODO: test with Ki=0.0003; \*/

static float Kd = 0.1f; // Дифференциальный коэффициент

/\*\* ADC data structure \*/

typedef struct

{

/\*\* Double buffers for the measured data \*/

float ch[BUF\_NUM][ADC\_CHANNEL\_FULL\_COUNT][ADC\_VAL\_NUM];

float active[ADC\_CHANNEL\_FULL\_COUNT]; /\*\*< RMS or mean value with correction, instantenous values \*/

uint16\_t active\_raw[ADC\_CHANNEL\_NUMBER]; /\*\*< RMS or mean value without correction, instantenous values \*/

float sum\_raw\_sqr[BUF\_NUM][ADC\_CHANNEL\_FULL\_COUNT]; /\*\*< Temporary sum for calculate adc active value (squared) \*/

float sum\_raw[BUF\_NUM][ADC\_CHANNEL\_FULL\_COUNT]; /\*\*< Temporary sum for calculate adc active value \*/

} adc\_t;

/\*\* pfc operation data \*/

typedef struct

{

adc\_t adc; /\*\*< ADC data \*/

complex\_amp\_t U\_50Hz[PFC\_NCHAN]; /\*\*< ADC data \*/

float period\_delta; /\*\*< The instantenous period error \*/

float period\_fact; /\*\*< The instantenous period value \*/

float U\_0Hz[PFC\_NCHAN]; /\*\*< The DC part of the U signal waveform \*/

float I\_0Hz[PFC\_NCHAN]; /\*\*< The DC part of the I signal waveform \*/

float U\_phase[PFC\_NCHAN]; /\*\*< The phase the U signal waveform \*/

float thdu[PFC\_NCHAN]; /\*\*< The total harmonic distortion (on U) \*/

float temperature; /\*\*< The temperature of the unit \*/

} pfc\_t;

static pfc\_t pfc; /\*\*< PFC operation data instance \*/

/\*--------------------------------------------------------------

PRIVATE FUNCTIONS

--------------------------------------------------------------\*/

/\*\*

\* @brief ADC DMA complete callback

\*/

void adc\_cplt\_callback(void)

{

memcpy(adc\_values\_raw, adc\_dma\_buffer, sizeof(adc\_dma\_buffer) / 2);

}

/\*\*

\* @brief PID controller

\*

\* @param et The current value of the error

\* @param et\_1 The last value of the error

\* @param Kp The proportional coefficient

\* @param Ki The integral coefficient

\* @param Kd The differential coefficient

\* @param It\_1 Integral value accumulator

\*

\* @return Regulated value, return 0 in case of an error

\*/

static float PID(float et, float\* et\_1, float Kp, float Ki, float Kd, float\* It\_1)

{

if (!et\_1 || !It\_1) return 0;

float Pt = Kp \* et;

float It = \*It\_1 + Ki \* et;

\*It\_1 = It \* PID\_LEAKAGE\_COEFFICIENT;

#if PID\_USE\_DIFFERENTIAL == 1

float Dt = Kd \* (et - \*et\_1);

float Ut = Pt + It + Dt;

#else

float Ut = Pt + It;

#endif

\*et\_1 = et;

return Ut;

};

/\*\*

\* @brief Lock the ADC module (mutex)

\*/

static void adc\_lock(void)

{

/\* TODO: Add lock functionality \*/

}

/\*\*

\* @brief Unlock the ADC module (mutex)

\*/

static void adc\_unlock(void)

{

/\* TODO: Add unlock functionality \*/

}

/\*\*

\* @brief ADC DMA half complete callback

\*/

static void adc\_half\_cplt\_callback(void)

{

gpio\_pwm\_test\_on();

int i\_isr;

adc\_stop();

adc\_lock();

memcpy(&adc\_values\_raw[ADC\_CHANNEL\_NUMBER / 2], &adc\_dma\_buffer[ADC\_CHANNEL\_NUMBER / 2], sizeof(adc\_dma\_buffer) / 2);

adc\_start((uint32\_t\*)adc\_dma\_buffer, sizeof(adc\_dma\_buffer));

for (i\_isr = 0; i\_isr < ADC\_CHANNEL\_NUMBER; i\_isr++)

{

adc\_values[i\_isr] = adc\_values\_raw[i\_isr];

}

settings\_calibrations\_t calibrations = settings\_get\_calibrations();

/\* Apply calibrations \*/

for (i\_isr = 0; i\_isr < ADC\_CHANNEL\_NUMBER; i\_isr++)

{

adc\_values[i\_isr] -= calibrations.offset[i\_isr];

adc\_values[i\_isr] \*= calibrations.calibration[i\_isr];

pfc.adc.ch[current\_buffer][i\_isr][symbol] = adc\_values[i\_isr];

}

if (!pfc\_is\_pwm\_on())

{

Ia\_e\_1 = 0 - adc\_values[ADC\_I\_A];

Ib\_e\_1 = 0 - adc\_values[ADC\_I\_B];

Ic\_e\_1 = 0 - adc\_values[ADC\_I\_C];

}

else

{

settings\_capacitors\_t capacitors = settings\_get\_capacitors();

float VL = PID(

capacitors.Ucap\_nominal - adc\_get\_cap\_voltage(),

&VLet\_1,

capacitors.ctrl\_Ucap\_Kp,

capacitors.ctrl\_Ucap\_Ki,

0,

&VLIt\_1);

// Идеальные синусоиды для каждой фазы

float idealA = sin(symbol / 128.0f \* 2.0f \* M\_PI);

float idealB = sin(symbol / 128.0f \* 2.0f \* M\_PI + 2.0f \* M\_PI / 3.0f);

float idealC = sin(symbol / 128.0f \* 2.0f \* M\_PI + 4.0f \* M\_PI / 3.0f);

// Реальные измерения

float realA = pfc.adc.ch[last\_buffer][ADC\_MATH\_A][symbol] / pfc.adc.active[ADC\_MATH\_A];

float realB = pfc.adc.ch[last\_buffer][ADC\_MATH\_B][symbol] / pfc.adc.active[ADC\_MATH\_B];

float realC = pfc.adc.ch[last\_buffer][ADC\_MATH\_C][symbol] / pfc.adc.active[ADC\_MATH\_C];

// Ошибки

float errorA = realA - idealA;

float errorB = realB - idealB;

float errorC = realC - idealC;

// Применяем ПИД-регуляторы

float va = PID(errorA, &Ia\_e\_1, 0, Ki, 0, &Ia\_It\_1);

float vb = PID(errorB, &Ib\_e\_1, 0, Ki, 0, &Ib\_It\_1);

float vc = PID(errorC, &Ic\_e\_1, 0, Ki, 0, &Ic\_It\_1);

// Ограничиваем выходные значения

if (va > (1.0f - EPS)) va = 1.0f - EPS;

if (vb > (1.0f - EPS)) vb = 1.0f - EPS;

if (vc > (1.0f - EPS)) vc = 1.0f - EPS;

if (va < (-1.0f + EPS)) va = -1.0f + EPS;

if (vb < (-1.0f + EPS)) vb = -1.0f + EPS;

if (vc < (-1.0f + EPS)) vc = -1.0f + EPS;

// Генерация ШИМ

uint32\_t ccr1 = (float)PWM\_PERIOD \* (-va \* 0.5f + 0.5f);

uint32\_t ccr2 = (float)PWM\_PERIOD \* (-vb \* 0.5f + 0.5f);

uint32\_t ccr3 = (float)PWM\_PERIOD \* (-vc \* 0.5f + 0.5f);

timer\_write\_pwm(ccr1, ccr2, ccr3);

pfc.adc.ch[current\_buffer][ADC\_MATH\_C\_A][symbol] = va;

pfc.adc.ch[current\_buffer][ADC\_MATH\_C\_B][symbol] = vb;

pfc.adc.ch[current\_buffer][ADC\_MATH\_C\_C][symbol] = vc;

}

symbol++;

if (symbol >= ADC\_VAL\_NUM)

{

symbol = 0;

current\_buffer ^= 1;

last\_buffer = !current\_buffer;

new\_period = 1;

}

adc\_unlock();

gpio\_pwm\_test\_off();

}

/\*--------------------------------------------------------------

PUBLIC FUNCTIONS

--------------------------------------------------------------\*/

/\*

\* @brief Run the algorithm

\*/

void algorithm\_process(void)

{

if (new\_period)

{

adc\_lock();

settings\_filters\_t filters = settings\_get\_filters();

float K\_I = (filters.K\_I);

float K\_U = (filters.K\_U);

float K\_Ucap = (filters.K\_Ucap);

float K\_Iinv = (1 - filters.K\_I);

float K\_Uinv = (1 - filters.K\_U);

float K\_Ucapinv = (1 - filters.K\_Ucap);

float umax[3] = {UMAX\_INITIAL\_VALUE};

float umin[3] = {UMIN\_INITIAL\_VALUE};

for (int i = 0; i < ADC\_VAL\_NUM; i++)

{

/\* Calculate mathematical channels \*/

float Uab = pfc.adc.ch[last\_buffer][ADC\_EDC\_B][i] - pfc.adc.ch[last\_buffer][ADC\_EDC\_A][i];

float Ubc = pfc.adc.ch[last\_buffer][ADC\_EDC\_C][i] - pfc.adc.ch[last\_buffer][ADC\_EDC\_B][i];

float Uca = pfc.adc.ch[last\_buffer][ADC\_EDC\_A][i] - pfc.adc.ch[last\_buffer][ADC\_EDC\_C][i];

float Uan = (2 \* Uab + Ubc) / 3;

float Ubn = (-Uab + Ubc) / 3;

float Ucn = -(Uan + Ubn);

pfc.adc.ch[last\_buffer][ADC\_MATH\_A][i] = Uan;

pfc.adc.ch[last\_buffer][ADC\_MATH\_B][i] = Ubn;

pfc.adc.ch[last\_buffer][ADC\_MATH\_C][i] = Ucn;

for (int i\_isr = 0; i\_isr < ADC\_CHANNEL\_NUMBER + ADC\_MATH\_NUMBER; i\_isr++)

{

if (needSquare[i\_isr])

{

/\* Calculate effective value for voltages and currents \*/

pfc.adc.sum\_raw\_sqr[last\_buffer][i\_isr] += SQUARE\_F(pfc.adc.ch[last\_buffer][i\_isr][i]);

}

else

{

/\* Calculate mean values for steady parameters \*/

pfc.adc.sum\_raw\_sqr[last\_buffer][i\_isr] += pfc.adc.ch[last\_buffer][i\_isr][i];

}

}

/\* Apply the filtration fo U and I parameters \*/

for (int i\_isr = 0; i\_isr < PFC\_NCHAN; i\_isr++)

{

IIR\_1ORDER(

pfc.adc.ch[last\_buffer][ADC\_EDC\_A + i\_isr][i],

pfc.adc.ch[current\_buffer][ADC\_EDC\_A + i\_isr][i],

pfc.adc.ch[last\_buffer][ADC\_EDC\_A + i\_isr][i],

K\_U,

K\_Uinv);

IIR\_1ORDER(

pfc.adc.ch[last\_buffer][ADC\_I\_A + i\_isr][i],

pfc.adc.ch[current\_buffer][ADC\_I\_A + i\_isr][i],

pfc.adc.ch[last\_buffer][ADC\_I\_A + i\_isr][i],

K\_I,

K\_Iinv);

if (umax[i\_isr] < pfc.adc.ch[last\_buffer][ADC\_EDC\_A + i\_isr][i]) umax[i\_isr] = pfc.adc.ch[last\_buffer][ADC\_EDC\_A + i\_isr][i];

if (umin[i\_isr] > pfc.adc.ch[last\_buffer][ADC\_EDC\_A + i\_isr][i]) umin[i\_isr] = pfc.adc.ch[last\_buffer][ADC\_EDC\_A + i\_isr][i];

}

IIR\_1ORDER(

pfc.adc.ch[last\_buffer][ADC\_UCAP][i],

pfc.adc.ch[current\_buffer][ADC\_UCAP][i],

pfc.adc.ch[last\_buffer][ADC\_UCAP][i],

K\_Ucap,

K\_Ucapinv);

}

/\* Calculate grid parameters \*/

for (int i\_isr = 0; i\_isr < ADC\_CHANNEL\_NUMBER + ADC\_MATH\_NUMBER; i\_isr++)

{

if (needSquare[i\_isr])

{

pfc.adc.active[i\_isr] = sqrt(pfc.adc.sum\_raw\_sqr[last\_buffer][i\_isr] / ((float)ADC\_VAL\_NUM));

}

else

{

pfc.adc.active[i\_isr] = pfc.adc.sum\_raw\_sqr[last\_buffer][i\_isr] / ((float)ADC\_VAL\_NUM);

}

pfc.adc.sum\_raw\_sqr[last\_buffer][i\_isr] = 0;

}

/\* Process oscillog data \*/

//HAL\_GPIO\_TogglePin(GPIOD, LED\_1\_Pin);

protocol\_write\_osc\_data(pfc.adc.ch[last\_buffer]);

/\* Process events \*/

events\_check\_rms\_overcurrent();

events\_check\_rms\_voltage();

events\_check\_period(pfc.period\_fact);

/\* Adjust the frequency \*/

/\*float f=auto\_correlation\_frequency(pfc.adc.ch[last\_buffer][ADC\_EDC\_A]);

IIR\_1ORDER(

pfc.period\_fact,

f,

pfc.period\_fact,

(1.0f-K\_filter\_F),

K\_filter\_F

);

\*/

float x = pfc.adc.ch[last\_buffer][ADC\_MATH\_A][PFC\_ACHAN];

float x\_last = x;

float cntr = 0; //(umax[PFC\_ACHAN]-umin[PFC\_ACHAN])/2;

static float P\_last = 0, P;

for (int j = 0; j < ADC\_VAL\_NUM; j++)

{

IIR\_1ORDER(

x,

pfc.adc.ch[last\_buffer][ADC\_MATH\_A][j],

x,

(1.0f - K\_filter\_x),

K\_filter\_x);

if (x\_last > cntr && x <= cntr)

{

P = ((float)j - 1.0f) + ((float)x\_last - cntr) / ((float)x\_last - (float)x);

if ((P - P\_last) > ((float)ADC\_VAL\_NUM / 4 \* 3))

{

float last\_period = pfc.period\_fact;

IIR\_1ORDER(

pfc.period\_fact,

(P - P\_last) \* (1.0f / (float)ADC\_VAL\_NUM) \* pfc.period\_fact,

pfc.period\_fact,

(1.0f - K\_filter\_F),

K\_filter\_F);

P\_last = P;

IIR\_1ORDER(

diff,

diff,

(P - (float)ADC\_VAL\_NUM / 2),

(1.0f - K\_filter\_P),

K\_filter\_P);

if (diff > MAXIMUM\_PERIOD\_DIFF) diff = MAXIMUM\_PERIOD\_DIFF;

if (diff < -MAXIMUM\_PERIOD\_DIFF) diff = -MAXIMUM\_PERIOD\_DIFF;

pfc.period\_fact -= diff \* K\_phase\_shift;

pfc.period\_delta = pfc.period\_fact - last\_period;

}

}

x\_last = x;

}

P\_last -= ADC\_VAL\_NUM;

if (pfc.period\_fact > (PERIOD\_REQUIRED + PERIOD\_DRIFT)) pfc.period\_fact = (PERIOD\_REQUIRED + PERIOD\_DRIFT);

if (pfc.period\_fact < (PERIOD\_REQUIRED - PERIOD\_DRIFT)) pfc.period\_fact = (PERIOD\_REQUIRED - PERIOD\_DRIFT);

/\* TODO: Move coefficients to the defines \*/

uint32\_t arr = (200000000.0f / 2.0f / (float)ADC\_VAL\_NUM \* (pfc.period\_fact / 1000000.0f));

timer\_correct\_period(arr);

pfc\_process();

new\_period = 0;

adc\_unlock();

}

}

/\*

\* @brief Get capacitors voltage

\*

\* @return Capacitors voltage [V]

\*/

float adc\_get\_cap\_voltage(void)

{

adc\_lock();

float ud\_ret = pfc.adc.active[ADC\_UCAP];

adc\_unlock();

return ud\_ret;

}

/\*

\* @brief Start ADC processing

\*

\* @return The status of the operation

\*/

status\_t adc\_logic\_start(void)

{

adc\_register\_callbacks(adc\_cplt\_callback, adc\_half\_cplt\_callback);

adc\_start((uint32\_t\*)adc\_dma\_buffer, sizeof(adc\_dma\_buffer));

timer\_start\_adc\_timer();

return PFC\_SUCCESS;

}

/\*

\* @brief Get the complex amplitude and phase

\* @note NULL pointers can be passed to omit a variable

\*

\* @param[out] U\_50Hz The complex amplitude and phase

\* @param[out] period\_delta The period error

\*/

void adc\_get\_complex\_phase(complex\_amp\_t\* U\_50Hz, float\* period\_delta)

{

adc\_lock();

if (U\_50Hz) memcpy(U\_50Hz, pfc.U\_50Hz, sizeof(pfc.U\_50Hz));

if (period\_delta) \*period\_delta = pfc.period\_delta;

adc\_unlock();

}

/\*

\* @brief Get grid parameters

\* @note NULL pointers can be passed to omit a variable

\*

\* @param[out] U\_0Hz The DC part of the U signal waveform

\* @param[out] I\_0Hz The DC part of the I signal waveform

\* @param[out] U\_phase The phase the U signal waveform

\* @param[out] thdu The total harmonic distortion (on U)

\* @param[out] period\_fact The instantenous period value

\*/

void adc\_get\_params(float\* U\_0Hz, float\* I\_0Hz, float\* U\_phase, float\* thdu, float\* period\_fact)

{

adc\_lock();

if (U\_0Hz) memcpy(U\_0Hz, pfc.U\_0Hz, sizeof(pfc.U\_0Hz));

if (I\_0Hz) memcpy(I\_0Hz, pfc.I\_0Hz, sizeof(pfc.I\_0Hz));

if (U\_phase) memcpy(U\_phase, pfc.U\_phase, sizeof(pfc.U\_phase));

if (thdu) memcpy(thdu, pfc.thdu, sizeof(pfc.thdu));

if (period\_fact) \*period\_fact = pfc.period\_fact;

adc\_unlock();

}

/\*

\* @brief Get temperature

\*

\* @return The temperature

\*/

float adc\_get\_temperature(void)

{

float ret\_temp = 0;

adc\_lock();

ret\_temp = pfc.temperature;

adc\_unlock();

return ret\_temp;

}

/\*

\* @brief Set temperature

\*

\* @param temperature The temperature

\*/

void adc\_set\_temperature(float temperature)

{

adc\_lock();

pfc.temperature = temperature;

adc\_unlock();

}

/\*

\* @brief Get instantenous (active) ADC values

\* @note NULL pointers can be passed to omit a variable

\*

\* @active[out] active Instantenous ADC values

\*/

void adc\_get\_active(float\* active)

{

adc\_lock();

if (active) memcpy(active, pfc.adc.active, sizeof(pfc.adc.active));

adc\_unlock();

}

/\*

\* @brief Get instantenous (active) ADC values in a raw form

\* @note NULL pointers can be passed to omit a variable

\*

\* @active[out] active\_raw Instantenous ADC values in a raw form

\*/

void adc\_get\_active\_raw(float\* active\_raw)

{

adc\_lock();

if (active\_raw) memcpy(active\_raw, pfc.adc.active\_raw, sizeof(pfc.adc.active\_raw));

adc\_unlock();

}

/\*

\* @brief Clear accumulator values

\*/

void adc\_clear\_accumulators(void)

{

VLet\_1 = 0;

VLIt\_1 = 0;

Ia\_e\_1 = 0;

Ib\_e\_1 = 0;

Ic\_e\_1 = 0;

Ia\_It\_1 = 0;

Ib\_It\_1 = 0;

Ic\_It\_1 = 0;

}

/\*\* @} \*/