Application Report
Lit. Number – Apr. 2013

Design a UCD3138 Controlled Single Phase PFC

Bosheng Sun

High Performance Isolated

ABSTRACT

The UCD3138[1] is a digital power supply controller from Texas Instruments offering superior levels of integration and performance in a single chip solution. The flexible nature of the UCD3138 makes it suitable for a wide variety of power conversion applications. In addition, multiple peripherals inside the device have been specifically optimized to enhance the performance of ac/dc applications such as power factor correction (PFC).

The UCD3138 is a fully programmable solution offering customers complete control of their application. However, the use of digital controllers in PFC design brings new challenges to many analog designers in their effort to change the design from the analog space to its new digital environment. This application note gives a step by step guidance on how to design a UCD3138 controlled average current mode single phase PFC. It covers the hardware interface, voltage loop and current loop implementation, system protection, firmware structure, internal state machines, as well as some advanced features. Finally, a graphical user interface (GUI) and a method for tuning the coefficients of a PFC system are presented. For interleaved or bridgeless PFC design, please refer to application notes [2] and [3].

Contents

Figures

1 Overview

1.1 Block Diagram

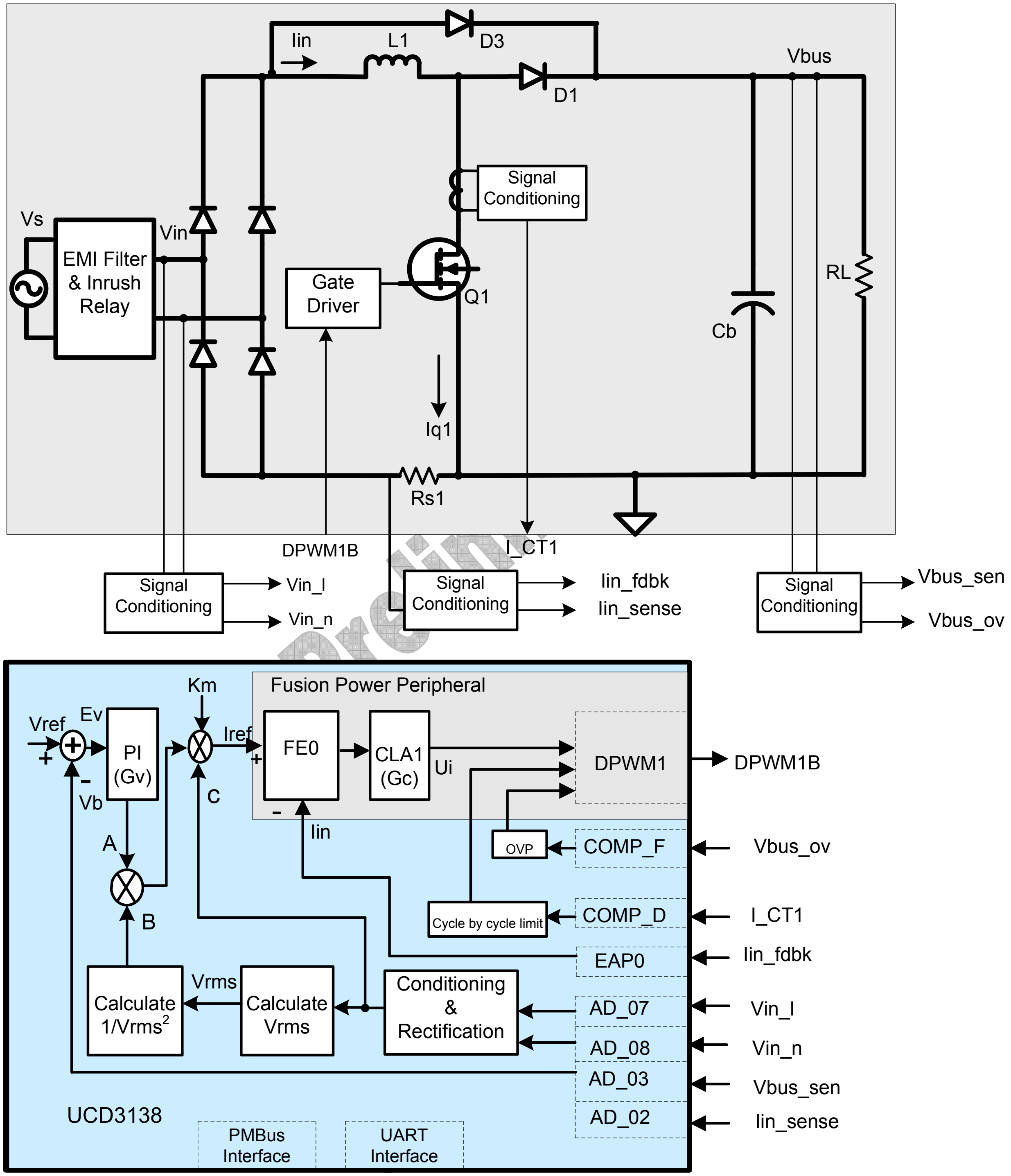


Figure 1. UCD3138 controlled single phase PFC block diagram

Figure 1 is an example of block diagram of a single phase PFC controlled by the UCD3138. The input voltage Vin is sensed before the bridge rectifier, the line and neutral are sensed separately by two ADC channels AD_07 and AD_08. PFC output voltage Vbus_sen is sensed by another ADC channel AD_03. In addition, a separate Vout sensing circuit is connected to an on chip analog comparator COMP_F for over voltage protection (OVP). The current signal is sensed by current shunt and its feedback signal Iin_fdbk is connected to error ADC channel EAP0 for current loop control. This signal is further filtered and connected to AD_02 for input power and RMS current measurement. A current transformer is used to sense the MOSFET instantaneous current and its output is connected to an on chip analog comparator, COMP_D, for cycle-by-cycle current protection. The control loop generates one PWM output, DPWM1B, to drive the MOSFETs through an external gate driver.

Average current mode control is used for input current regulation. The current reference is calculated based on Vin, voltage loop output and input voltage feed forward. The averaged input current signal is sensed and compared with this reference, the error goes through a 2-pole 2-zero digital compensator CLA1; a PWM signal is generated based on the compensator output to control the PFC.

It needs to be mentioned here that the above configuration reflects TI's PFC evaluation board PWR026. It is not necessary to follow this configuration. For example, Iin_fdbk can be connected to a different EAP channel, a different CLA can be used for compensation, and the PFC can be driven by different DPWM outputs as well. However, in order to maximize source code reuse and reduce design time it is recommended to use a configuration similar to PWR026.

1.2 Signal Conditioning and Interface

For each input signal to the UCD3138, its magnitude should accommodate the measurement range of the UCD3138. In the UCD3138, the ADC measurement range is 0 – 2.5V, the error ADC measurement range is 0 - 1.6V, the analog comparator range is 0 – 2.5V. On the other hand, to have the best signal-to-noise ratio, the input signal should be as big as possible. For these reason, the signal conditioning for each input signal should follow the subsequent guidelines.

- For Vin, the voltage divider: $K_{vin} \le \frac{2.5}{\sqrt{2} * V_{in\_\max}}$
- For Vout, the voltage divider: $K_{vout} \le \frac{2.5}{V_{out\_\max}}$
- For I_shunt, the OP_AMP gain: $K_i \le \frac{1.6}{I_{in\_\max} * R_{shunt}}$
- For current transformer: I_CT1 ≤ 2.5V

2 Voltage Loop

2.1 Overview

Since the speed constraints on the voltage loop bandwidth are typically low, it can easily be implemented by pure firmware. As shown in Figure 1. Vout_sen is sensed by a 12-bit ADC. An error signal is calculated based on the target output voltage and then processed by a proportional-integral (PI) controller. The output of this PI controller will take part in the current reference calculation.

To meet the load transient response requirement, a non-linear PI gain is used. When the voltage error exceeds a threshold, a larger PI gain is used.

2.2 Firmware Implementation of PI Controller

Following is the code example for this nonlinear PI controller. Two different gains are used in this example. If the load transient response still is not met, a third or forth gain can be added.

All the codes in this application note are just examples of how to implement a specific function, it does not contain the variable definitions and how the function gets called. Although plenty of comments are provided to explain how it works, there may still exist unclearness. To better understand the code, please refer to PWR026 PFC EVM source code for details.

```
inline int32 proportional_integral(int32 error) //error is difference between ADC value and reference
{
        int32 output, steady_state_error;

        if( abs(error) < iv.pis.nl_threshold) //if error is small, then use small PI gain
        {
                steady_state_error = iv.vbus_target - (iv.vbus_filtered >> 6);
                iv.pis.p = iv.pis.kp * steady_state_error; //proportional
                iv.pis.i = iv.pis.i + (iv.pis.ki * steady_state_error); //integrator
        }
        else
        {       //non-linear gain for Voltage loop
                iv.pis.p = iv.pis.kp_nl * error; //proportional
                iv.pis.i = iv.pis.i + (iv.pis.ki_nl * error); //integrator
        }

        if(iv.pis.i > PI_I_HIGH_LIMIT) //clamp integrator
        {
                iv.pis.i = PI_I_HIGH_LIMIT;
        }
        else if (iv.pis.i < PI_I_LOW_LIMIT)
        {
                iv.pis.i = PI_I_LOW_LIMIT;
        }

        output = (iv.pis.p + iv.pis.i) >> 12; //scale for Q15 from Q15 coefficients and Q12 from ADC

        if(output > PI_OUTPUT_HIGH_LIMIT) //clamp PI output
        {
                output = PI_OUTPUT_HIGH_LIMIT;
        }
        else if (output < PI_OUTPUT_LOW_LIMIT)
        {
                output = PI_OUTPUT_LOW_LIMIT;
        }

        iv.pis.output = output;
        return output;
}
```

3 Current Loop

3.1 Overview

The PFC current loop is used to regulate the inductor current so that the input current will follow the input voltage. To do this, the current reference, which takes the same shape as the input voltage, needs to be calculated first. For an average current mode controlled PFC, the current reference is calculated as:

$$I_{ref} = K_m * A * B * C \quad (1)$$

while:
K_m: multiplier gain
A: Voltage loop output

B: $1/V_{in\_rms}^2$
C: sensed input voltage $K_{vin}*V_{in}$

Once the current reference is calculated, the corresponding function blocks in the chip need to be configured to close the loop. There are 3 major hardware blocks for the current loop: Front End, Filter, and DPWM. These blocks will be introduced one by one.

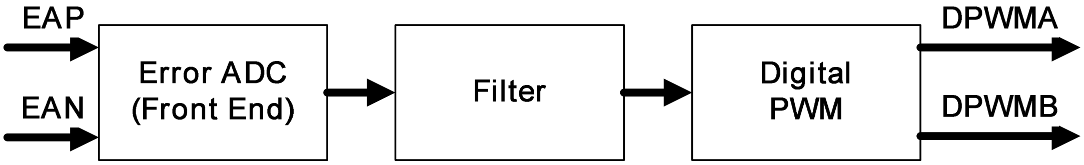


Figure 2. Current loop

3.2 Multiplier Gain K_m

The multiplier gain K_m is defined as follows:
From (1),

$$I_{ref} = K_m * A * C * B = K_m (U_v)(V_{in} * K_{vin}) / V_{rms}^2 \qquad (2)$$

where,

U_v : voltage loop output
V_{rms} : RMS voltage of digitized input voltage
K_{vin} : Input voltage divider

For digital implementation, the voltage signals in (2) are digitized, a suitable fixed-point notation is chosen so that each signal is normalized with the maximum value equals to 1. For maximum power output, at minimum Vin, the voltage controller output and the reference current command will be at their maximum values, U_{vmax} and I_{ref_max} respectively. Since I_{ref} and U_v are calculated in per unit, their maximum values are, $I_{ref\_max} = 1$, $U_{vmax} = 1$. Therefore,

$$Km = I_{ref\_\max}[\frac{V_{rms(\min)}^2}{V_{\min(pk)}K_{vin}U_{v\max}}] = \frac{V_{rms(\min)}^2}{V_{\min(pk)}K_{vin}} \qquad (3)$$

For sine wave input, this can be written as,

$$K_m = \frac{K_{vin}^2 V_{\min(pk)}^2}{2V_{\min(pk)}K_{vin}} = 0.5K_{vin}V_{\min(pk)} \qquad (4)$$

3.3 Vin Sensing and Rectification

Vin line and neutral are measured by 2 ADC channels separately, then it is rectified by firmware.

```
inline void rectify_vac(void)
{
        if(iv.adc_raw[AC_L_CHANNEL] > iv.adc_raw[AC_N_CHANNEL] ) //this is the cycle for line
        {
                iv.vin_raw = iv.adc_raw[AC_L_CHANNEL] - iv.adc_raw[AC_N_CHANNEL];
                iv.positive = 1; //tell other functions that this is positive cycle
        }
        else //cycle for neutral
        {
                iv.vin_raw = iv.adc_raw[AC_N_CHANNEL] - iv.adc_raw[AC_L_CHANNEL];
                iv.positive = 0; //tell other functions that this is negative cycle
        }
        iv.vin_sum = iv.vin_raw + iv.vin_sum - (iv.vin_sum >> 2);
        iv.vin_filtered = iv.vin_sum >> 2; //filtered vin measurement
}
```

3.4 Calculate $V_{in\_rms}^2$

The RMS value is defined as:

$$V_{rms}{}^2 = \frac{1}{T_{ac}} * \int_0^{T_{ac}} V(t)^2 \, dt \qquad (5)$$

In discrete format:

$$V_{rms}{}^2 = \frac{\sum V(n)^2}{N} \qquad (6)$$

Vin is sampled every 20us, then the sampled Vin is squared and accumulated in each AC cycle. The RMS value is calculated by divide the number of samples in one cycle.

First, calculate $V(n)^2$

```
iv.vin_squared = (iv.vin_filtered * iv.vin_filtered) >> 9;
```

Then, calculate sum

```
//sum V(n)^2 for the negative cycle
inline void accumulate_negative_cycle_values()
{
        iv.negative_vin_squared_accumulate = iv.vin_squared + iv.negative_vin_squared_accumulate;
}

// sum V(n)^2 for the positive cycle
inline void accumulate_positive_cycle_values()
{
        iv.positive_vin_squared_accumulate = iv.vin_squared + v.positive_vin_squared_accumulate;
}
```

Finally, calculate Vin_rms^2

```
//calculate Vin_rms^2 for the negative cycle
inline void store_negative_cycle_values(void)
{
        iv.vin_squared_average = iv.negative_vin_squared_accumulate / iv.negative_cycle_counter;
        iv.vin_squared_for_ac_drop = iv.vin_squared_average;
}

//calculate Vin_rms^2 for the positive cycle
inline void store_positive_cycle_values(void)
{
        iv.vin_squared_average = iv.positive_vin_squared_accumulate / iv.positive_cycle_counter;
        iv.vin_squared_for_ac_drop = iv.vin_squared_average;
}
```

3.5 Calculate Vin Feed Forward

The following function is used to calculated voltage feed forward Km * B

Km: multiplier gain
B: 1/Vin_rms^{2}

```
inline void voltage_feed_forward(void) //calculate Km/Vrms^2
{
        if(iv.vin_squared_average < VAC_MIN_OFF_SQ_AVG) //if VAC is below normal operation range
        {
                iv.vff_multiplier = K_FEED_FORWARD / VAC_MIN_OFF_SQ_AVG;
                //Q30/Q15 = Q15 limit to minimum operating voltage to avoid overflow
        }
        else //here if vac is within range
        {
                if(abs(iv.vin_squared_average – (iv.vin_squared_slow_average >>
                VRECT_SQUARED_SLOW_AVERAGE_SHIFT)) >        (iv.vin_squared_slow_average >>
                (VRECT_SQUARED_SLOW_AVERAGE_SHIFT + 4)))
                //compares difference between fast and slow VAC values to a percentage of the slow value.
                //instead of multiplying the slow value times a constant, it uses a shift.  So a shift of +4, for
                //example = 1/16  or .0625% of the slow value.
                //so the code below is executed if the difference between fast and slow values is greater
                //than the percentage.  It uses the fast value.
                {
                        iv.vff_multiplier = K_FEED_FORWARD / iv.vin_squared_average;
                }
                else //here if the fast and slow values are close - use the slow value.
                {
                        if(iv.vin_squared_slow_average < (VAC_MIN_OFF_SQ_AVG <<
                                VRECT_SQUARED_SLOW_AVERAGE_SHIFT))
                        {
                                iv.vff_multiplier = K_FEED_FORWARD / AC_MIN_OFF_SQ_AVG;
                                //Q30/Q15 limit to minimum operating voltage to avoid overflow
                        }
                        else
                        {
                                iv.vff_multiplier = K_FEED_FORWARD / (iv.vin_squared_slow_average >>
                                VRECT_SQUARED_SLOW_AVERAGE_SHIFT);
                        }
                }
        }
}
```

3.6 Calculate Current Reference

Now that we have A, B and C, we can calculate the average current reference. This is done in 2 functions:

First, calculate Km * A * B:

```
inline void handle_voltage_loop(void)
{
        iv.i_target_average = ((iv.vff_multiplier >> 5) * proportional_integral(iv.vbus_target -
                v.adc_avg[VBUS_CHANNEL])) >> 11;
}
```

Then, calculate Km * A * B * C, which is the current reference:

```
inline void calculate_current_target_shunt(void)
{
        int32 pointer;

        //for EMI CAP compensation
        iv.cir_buff[iv.cir_buff_ptr] = iv.vin_filtered;
        pointer = (iv.cir_buff_ptr - iv.cir_buff_delay) & 0x3f; //get pointer to delayed signal
        iv.i_target_sensed = ((iv.cir_buff[pointer] * iv.i_target_average) >> 16) + iv.i_target_offset;
        iv.cir_buff_ptr = (iv.cir_buff_ptr + 1) & 0x3f;

        if(iv.i_target_sensed > 0x3ff) //clamp current target at maximum current
        {
                iv.i_target_sensed = 0x3ff;
        }

        FeCtrl0Regs.EADCDAC.bit.DAC_VALUE = iv.i_target_sensed << 4; //disregard dithering bits.
}
```

Now that the current reference is calculated, it’s time to configure the current loop hardware so that the loop can be closed.

3.7 Current Feed Back Front End Configuration

The Front End measures the difference between the current feed back signal and the current reference calculated in section 3.6. It passes this digital error information to the filter. The blocks which need to be configured are: the front end for current feedback signal sensing, the AFE_GAIN, the CLA filter, the compensation coefficients, and the DPWM module. The UCD3138 is very flexible, the front end, filter and DPWM facilitate connections to and from multiple peripherals. The following code example is based on the PWR026 PFC EVM hardware; it uses Frond End0, CLA1 and DPWM1. Other configurations are also possible. For detail of how to configure UCD3138 digital peripherals, please refer to programmer manual [4].

```
void init_front_end0(void)
{
        FeCtrl0Regs.EADCDAC.bit.DAC_VALUE = 0;
        FeCtrl0Regs.EADCCTRL.bit.AFE_GAIN = 1;
}

void init_loop_mux(void)
{
        LoopMuxRegs.SAMPTRIGCTRL.bit.FE0_TRIG_DPWM1_EN = 1; //Use DPWM1 sample trigger for FE0

        LoopMuxRegs.FILTERMUX.bit.FILTER1_FE_SEL = 0; //use FE0 (shunt) to drive CLA1
        LoopMuxRegs.FILTERMUX.bit.FILTER1_PER_SEL = 1;//CLA1 switching period select from DPWM1

        LoopMuxRegs.DPWMMUX.bit.DPWM1_FILTER_SEL =1; //CLA1 is providing input to DPWM1
        LoopMuxRegs.DPWMMUX.bit.DPWM3_SYNC_SEL = 1;  //DPWM1 is the master for DPWM3
}
```

3.8 Current Loop Filter Configuration

The filter takes the error signal from the Front End and passes it through a 2-pole 2-zero digital filter which compensates the disturbance of the current loop. The filter needs to be initialized such that once powered up, the current loop will use well-tuned control parameters to close the current loop:

```
void init_filter1(void)
{
        MiscAnalogRegs.CLKTRIM.bit.HFO_LN_FILTER_EN = 1;
        Filter1Regs.FILTERCTRL.bit.OUTPUT_MULT_SEL = 1; //PID output multiply with period
        Filter1Regs.FILTERCTRL.bit.OUTPUT_SCALE = 0; //no scale

        Filter1Regs.FILTERKICLPHI.bit.KI_CLAMP_HIGH = 0x7FFFF0;
        Filter1Regs.FILTERKICLPLO.bit.KI_CLAMP_LOW = 0x800010;

        Filter1Regs.FILTERYNCLPHI.all = 0x799999; //95%
        Filter1Regs.FILTERYNCLPLO.all = 0;

        Filter1Regs.FILTERCTRL.bit.FILTER_EN = 1;
        //enable OK here, because nothing will happen until DPWM and front end are globally enabled
}
```

In PWR026 PFC EVM, some of the important parameters, such as set-points, AFE_Gain, Oversampling, voltage loop PI gain and current loop PID gain are stored in data flash. Upon power up, they are loaded from data flash to their corresponding registers.

```
void copy_configuration_to_registers(volatile struct FILTER_REGS *dest)
{
        //copy PFC configuration
        iv.vbus_voltage = pfc_config_in_ram.PFC_SETPOINT.VOUT_COMMAND +
                pfc_config_in_ram.PFC_CAL.VOUT_CAL_OFFSET;
        iv.vbus_setpoint = ((Uint32)((iv.vbus_voltage * 4095) / VBUS_FULL_RANGE));

        if(iv.supply_state >= STATE_PFC_ON)
        {
                iv.vbus_target = ((int32)((iv.vbus_voltage * 4095)/VBUS_FULL_RANGE));
        }

        FaultMuxRegs.ACOMPCTRL2.bit.ACOMP_F_THRESH =
                ((Uint32)(pfc_config_in_ram.PFC_SETPOINT.VOUT_OV_LIMIT * 127) / VBUS_FULL_RANGE);

        switching_frequency = pfc_config_in_ram.PFC_SETPOINT.FREQUENCY;

        //copy voltage loop gains
        iv.pis.kp  = pfc_config_in_ram.PI_GAINS.KP;
        iv.pis.ki   = pfc_config_in_ram.PI_GAINS.KI;
        iv.pis.kp_nl                    = pfc_config_in_ram.PI_GAINS.KP_NL;
        iv.pis.ki_nl                    = pfc_config_in_ram.PI_GAINS.KI_NL;
        iv.pis.nl_threshold  = (pfc_config_in_ram.PI_GAINS.NL_THRESHOLD << 12) / VBUS_FULL_RANGE;

        //copy current loop gains
        dest->COEFCONFIG.all       = pfc_config_in_ram.COEFCONFIG.all;
        dest->FILTERKPCOEF0.all = pfc_config_in_ram.FILTERKPCOEF0.all;
        dest->FILTERKPCOEF1.all = pfc_config_in_ram.FILTERKPCOEF1.all;
        dest->FILTERKICOEF0.all  = pfc_config_in_ram.FILTERKICOEF0.all;
        dest->FILTERKICOEF1.all  = pfc_config_in_ram.FILTERKICOEF1.all;
        dest->FILTERKDCOEF0.all = pfc_config_in_ram.FILTERKDCOEF0.all;
        dest->FILTERKDCOEF1.all = pfc_config_in_ram.FILTERKDCOEF1.all;
        dest->FILTERKDALPHA.all= pfc_config_in_ram.FILTERKDALPHA.all;
        dest->FILTERNL0.all = pfc_config_in_ram.FILTERNL0.all;
        dest->FILTERNL1.all = pfc_config_in_ram.FILTERNL1.all;
        dest->FILTERNL2.all = pfc_config_in_ram.FILTERNL2.all;
        dest->FILTERCTRL.bit.NL_MODE = fc_config_in_ram.FILTERMISC.bit.NL_MODE;

        FeCtrl0Regs.EADCCTRL.bit.AFE_GAIN = pfc_config_in_ram.FILTERMISC.bit.AFE_GAIN;

        Dpwm1Regs.DPWMCTRL2.bit.SAMPLE_TRIG1_OVERSAMPLE=
                pfc_config_in_ram.FILTERMISC.bit.SAMPLE_TRIG1_OVERSAMPLE;
}
```

3.9 DPWM Configuration

The output of the compensator is passed to a Digital PWM (DPWM) generator. The DPWM has two outputs, which can be configured in many different ways to accommodate different power topologies. For PFC, either DPWMA or DPWMB can be used. If DPWMA is used, it needs to be set to normal mode, for DPWMB, it needs to be set to Multi-mode. The following code example accommodates the PWR026 PFC EVM, in which DPWMB is used and configured as Multi-mode. Another PWM module DPWM3 is also configured to do PFC ZVS control[5].

```
void init_dpwm1(void) // DPWM1B is used to drive 1st phase
{
        Dpwm1Regs.DPWMCTRL0.bit.PWM_EN = 0;  //disable everything

        Dpwm1Regs.DPWMCTRL1.bit.GPIO_A_EN = 1; //turn off DPWM1A for now
        Dpwm1Regs.DPWMCTRL1.bit.GPIO_B_EN = 1; //turn off DPWM1B for now

         // Enable CBC and Blanking windows
        Dpwm1Regs.DPWMCTRL0.bit.CBC_PWM_AB_EN = 1; // Enable cycle by cycle current limit.
        Dpwm1Regs.DPWMCTRL0.bit.BLANK_B_EN = 1;      // Enable blanking Dpwm1Regs.DPWMBLKBBEG.all = 0x0000;
        Dpwm1Regs.DPWMBLKBEND.all = 0x0500;

        Dpwm1Regs.DPWMFLTCTRL.bit.B_MAX_COUNT = 2;
        Dpwm1Regs.DPWMFLTCTRL.bit.ALL_FAULT_EN = 1; //enable this for OVP

        Dpwm1Regs.DPWMCTRL2.bit.SAMPLE_TRIG_1_EN = 1; //enable sample trigger1

        Dpwm1Regs.DPWMEV1.all = 290;
        Dpwm1Regs.DPWMEV3.all = 290; //aviod 72ns events update window
        Dpwm1Regs.DPWMCTRL0.bit.PWM_MODE = 2; //multi mode
        Dpwm1Regs.DPWMCTRL2.bit.SAMPLE_TRIG1_OVERSAMPLE = 3; //8X oversampling.
        Dpwm1Regs.DPWMCTRL1.bit.EVENT_UP_SEL = 1; //update at end of period

        Dpwm1Regs.DPWMCTRL0.bit.CLA_EN = 1;
        Dpwm1Regs.DPWMCTRL0.bit.PWM_EN = 1;
        //enable OK here, because nothing will happen until DPWM and front end are globally enabled
}
```

```
void init_dpwm3(void) // DPWM3 is used for ZVS control
{
        Dpwm3Regs.DPWMCTRL0.bit.PWM_EN = 0;  //disable everything

        Dpwm3Regs.DPWMCTRL1.bit.GPIO_A_EN = 0;//turn on DPWM3A
        Dpwm3Regs.DPWMCTRL1.bit.GPIO_B_EN = 1;//turn off DPWM3B for now

        Dpwm3Regs.DPWMCTRL0.bit.CLA_EN = 0;     //open loop

        Dpwm3Regs.DPWMCTRL0.bit.MSYNC_SLAVE_EN = 1; //slave mode

        Dpwm3Regs.DPWMCTRL0.bit.PWM_EN = 1;//enable OK here, because nothing will happen until DPWM and front end are
globally enabled
}

void set_new_switching_frequency(void)
{
        iv.switching_period = (SWITCH_FREQ_NUMERATOR/switching_frequency) << 4;
        iv.period_times_2_14 = iv.switching_period << 14;
        iv.dither_max_period = (SWITCH_FREQ_NUMERATOR/(switching_frequency - 4)) << 4;
        iv.dither_min_period = (SWITCH_FREQ_NUMERATOR/(switching_frequency + 4)) << 4;
        iv.dither_step = ((iv.dither_max_period - iv.dither_min_period) << 14)/DITHER_PERIOD;
        //step for dither value
        Dpwm1Regs.DPWMPRD.all = iv.switching_period; //new period for new frequency
        Dpwm3Regs.DPWMPRD.all = iv.switching_period; //new period for new frequency
        Dpwm1Regs.DPWMSAMPTRIG1.all = iv.switching_period - (iv.sample_trigger_offset * 4); // sample at the end of period
        Dpwm1Regs.DPWMPHASETRIG.all = 0; //0 delay for next phase
}

void init_dpwms(void)
{
        init_dpwm1();
        init_dpwm3();
        set_new_switching_frequency();
}
```

4 System Protection

System protection includes current protection and voltage protection. There are two levels of over voltage protection, one is implemented through software with a lower threshold, and the other is through an on chip analog comparator with a higher threshold. The current is protected on a cycle-by-cycle bases.

4.1 Software OVP Protection

This is a pure software OVP protection. Vout is measured by an ADC, the output of ADC is filtered for measurement noise immunity, and then compared with a programmable threshold. The PWM will shut down if the measurement is greater than a user programmable threshold. The ADC continues monitoring Vout, PWM will turn back on once Vout drops below its setpoint. This allows the PFC to enter a hiccup mode. This will be useful for OVP conditions that are not caused by hardware failure, but by a sudden operating condition change, such as a load transient.

```
inline void pfc_on_state_handler(void)
{
        if(iv.vin_squared_average > VAC_MIN_OFF_SQ_AVG) //if Vac above 80 volts
        {
                if(iv.adc_avg[VBUS_CHANNEL] > VBUS_DPWM_OFF_LEVEL)//if we've hit OVP
                {
                        turn_off_pfc();
                        iv.supply_state = STATE_PFC_HICCUP;
                }
        }
        else
        {
                turn_off_pfc();
                init_miscellaneous();
                iv.supply_state = STATE_IDLE;
        }
}

inline void pfc_hiccup_state_handler(void)
{
        if(iv.adc_avg[VBUS_CHANNEL] < VBUS_DPWM_ON_LEVEL) //if OVP gone
        {
                LoopMuxRegs.GLBEN.all = 0x70F; //global enable all Front_ends and DPWMs
                turn_on_pfc();
                iv.supply_state = STATE_PFC_ON;
        }
}

void turn_on_pfc(void)
{
        Dpwm1Regs.DPWMCTRL1.bit.GPIO_B_EN = 0;
}

void turn_off_pfc(void)
{
        Dpwm1Regs.DPWMCTRL1.bit.GPIO_B_EN = 1;

        LoopMuxRegs.GLBEN.all= 0; //disable all front end and DPWMs
}
```

4.2 Hardware OVP Protection

As shown in Figure 1, Vout is also connected to an on chip analog comparator COMP_F. the comparator is configured to turn off the PWM automatically once triggered. The comparator's threshold is also programmable, and its threshold is usually set a little bit higher than the software OVP. This provides a fast OVP protection. If this OVP gets triggered, this usually means the PFC has had a serious hardware failure. Because of this it is latched there once shut down for safety purpose.

Following is the code to configure this OVP:

```
// Enable ACOMP-F pin and connect to DPWM-1 and DPWM-2 for Vbus OV protection
FaultMuxRegs.DPWM1FAULTDET.bit.PWMB_ACOMP_F_EN = 1;  // Connect ACOMP-F to DPWM-1
FaultMuxRegs.ACOMPCTRL2.bit.ACOMP_F_SEL = 0;         // Use threshold register for trip
FaultMuxRegs.ACOMPCTRL2.bit.ACOMP_F_POL = 1;         // Above thresh to trip
FaultMuxRegs.ACOMPCTRL2.bit.ACOMP_F_THRESH = ((Uint32)(PFC_CONFIG_TEMP.PFC_SETPOINT.VOUT_OV_LIMIT *
127) / VBUS_FULL_RANGE);
```

4.3 Cycle by Cycle Current Protection

The current is protected through on chip analog comparators COMP_D and COMP_E. It is cycle by cycle (CBC) based. Once the analog comparator is triggered, the PWM is turned off for the remainder of the cycle, but it will turn back on the next switching cycle. The code to configure the analog comparator for CBC is:

```
// Enable ACOMP-D pin and connect to current limit on DPWM-1
FaultMuxRegs.DPWM1CLIM.bit.ACOMP_D_EN = 1;      // Connect ACOMP-D to DPWM-1
FaultMuxRegs.ACOMPCTRL1.bit.ACOMP_D_SEL = 0;      // Use threshold register for trip
FaultMuxRegs.ACOMPCTRL1.bit.ACOMP_D_POL = 1;      //Above thresh to trip
FaultMuxRegs.ACOMPCTRL1.bit.ACOMP_D_THRESH = OC_COMPARATOR; // Trip value
```

5 Advanced Features

5.1 Frequency Dithering

Frequency dithering refers to modulating the switching frequency to achieve a reduction in conducted EMI noise beyond the capability of the line filter. This total range from minimum to maximum frequency is defined as the dither magnitude, and is centered on the nominal switching frequency. The rate at which PWM traverses from one extreme to the other and back again is defined as the dither rate. Both these two parameters are programmable.

```
inline void frequency_dithering(void)
{
        if(status_1.bits.dither_enabled == 1)
        {
                if(iv.dither_direction == 1)
                {
                    iv.period_times_2_14 = iv.period_times_2_14 + iv.dither_step;
                    iv.switching_period = iv.period_times_2_14 >> 14;
                    if(iv.switching_period > iv.dither_max_period)
                    {
                              iv.switching_period = iv.dither_max_period;
                              iv.dither_direction = 0;
                    }
          }
          else  //if dither direction equalled 0 to start with
          {
                    iv.period_times_2_14 = iv.period_times_2_14 - iv.dither_step;
                    iv.switching_period = iv.period_times_2_14 >> 14;

                    if(iv.switching_period < iv.dither_min_period)
                    {
                              iv.switching_period = iv.dither_min_period;
                              iv.dither_direction = 1;
                    }
          }

          Dpwm1Regs.DPWMPRD.all = iv.switching_period; //new period for new frequency
          Dpwm1Regs.DPWMSAMPTRIG1.all = iv.switching_period – (iv.sample_trigger_offset*4);
     }
}
```

5.2 AC Drop Detection

The AC drop detection algorithm is shown in Figure 3. Vin is checked every 100µs. Its measurement is compared with a predetermined threshold “AC_DROP_V_RECT_THRESHOLD”. If the consecutive samples below this threshold are greater than a predetermined number, “AC_DROP_COUNT_MAX”, then AC drop is detected, a AC drop signal is send out to the host through a GPIO.

The threshold and number of consecutive samples will affect the sensitivity of the AC drop detection algorithm. They can be tuned base on the specific application requirements.

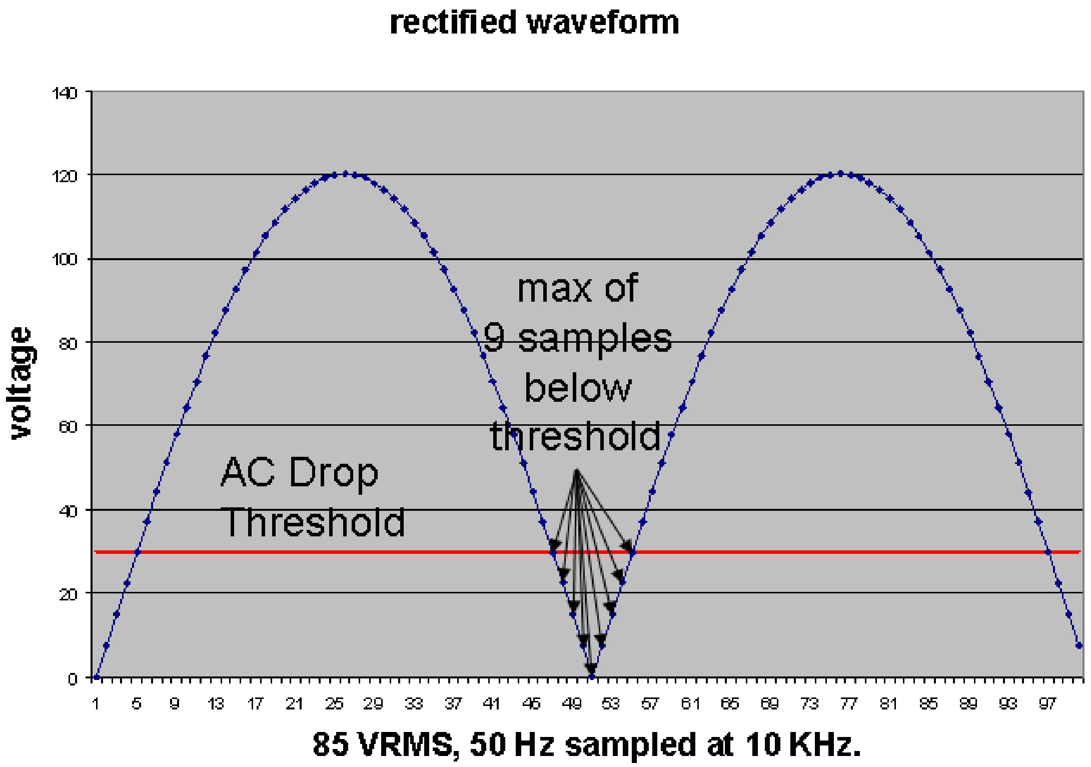


Figure 3. AC Drop Detection

The following is the code to implement this function:

```
inline void check_ac_drop(void)
{
        if(iv.vin_filtered > AC_DROP_V_RECT_THRESHOLD)
        {
                iv.ac_drop_count = 0; //if over threshold, clear counter
        }
        else
        {
                iv.ac_drop_count++;
                if(iv.ac_drop_count > AC_DROP_COUNT_MAX)
                {
                        iv.ac_drop = 1;
                        iv.ac_drop_recovery_not_complete = 1;
                        iv.vin_squared_for_ac_drop = 0; //clear for ac recovery detection
                        MiscAnalogRegs.GLBIOVAL.bit.DPWM3B_IO_VALUE = 0;
                        //pull down opto to signal AC drop to primary side
                }
        }

        if(iv.vin_squared_for_ac_drop > AC_UNDROPPED_THRESHOLD)
        //if above ac not dropped threshold
        {
                iv.ac_drop = 0;  // we've got enough energy, clear AC drop warning
                MiscAnalogRegs.GLBIOVAL.bit.DPWM3B_IO_VALUE = 1;
                //turn off AC drop output signal also - inactive high
        }
}
```

5.3 PFC ZVS or Valley Switching Control

In discontinuous conduction mode (DCM), when the MOSFET is turned off, the boost inductor current starts to decrease. The current will not stop decreasing when it reaches zero. Rather, it will continue going to negative value and an oscillation between the inductor and the total parasitic capacitance at the switching node occurs.

The oscillation period and amplitude are dependent on the inductance and capacitance values as well as the operating point. This oscillation results in a significant current distortion and seriously deteriorates the THD.

A new control method developed by TI [5] can always force the MOSFET to turn on at the point shown by the arrows in figure 4. Since the MOSFET always turns on at the same zero current position, the issue described in previous paragraph is solved, and the current distortion is significantly reduced. Also, because of zero voltage switching (ZVS) and zero current switching (ZCS), the efficiency is also improved.

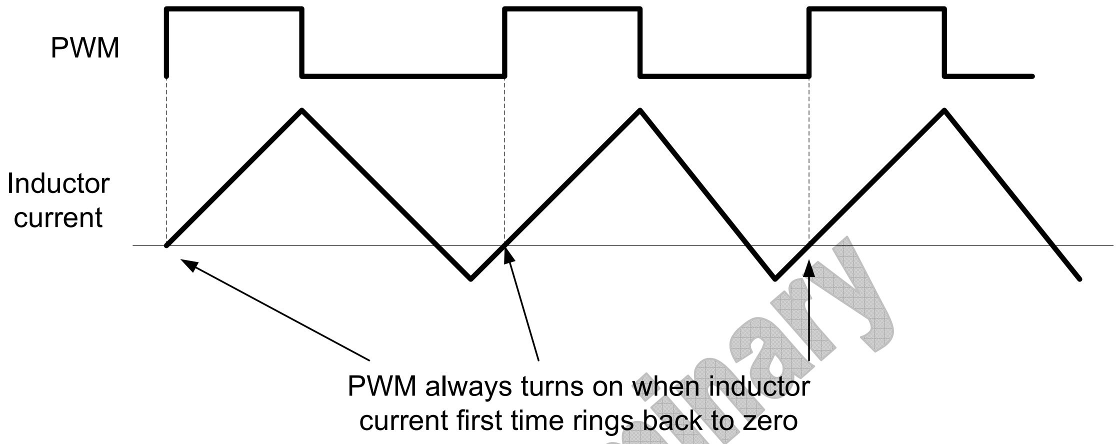

PWM
Inductor current
PWM always turns on when inductor current first time rings back to zero

Figure 4. A New ZVS/Valley Switching Control Algorithm.

For more details of how to implement this control, please refer to application report SLUA644.

5.4 X-CAP Reactive Current Compensation

Every PFC has an electromagnetic interference (EMI) filter on the input end. The X capacitors of the EMI filter will cause the AC input current to lead the AC input voltage, which will degrade power factor (PF). This situation gets worse at light-load and high-line. To increase the PF at light-load, we can force the inductor current to be slightly delayed so that the total AC current will match the input voltage. This can be achieved by delaying the current reference.

```
//stuff for EMI CAP compensation
int16 cir_buff[64]; //64buffer for vin
int32 cir_buff_ptr; //pointer for spot in cir buff;
int32 cir_buff_delay; //delay for waveform from circular buffer.

inline void calculate_current_target_shunt(void)
{
        int32 pointer;

        //for EMI CAP compensation
        iv.cir_buff[iv.cir_buff_ptr] = iv.vin_filtered;
        pointer = (iv.cir_buff_ptr - iv.cir_buff_delay) & 0x3f; //get pointer to delayed signal
        iv.i_target_sensed = ((iv.cir_buff[pointer] * iv.i_target_average) >> 16) + iv.i_target_offset;
        iv.cir_buff_ptr = (iv.cir_buff_ptr + 1) & 0x3f;
        if(iv.i_target_sensed > 0x3ff) //saturate current target at maximum current
        {
                iv.i_target_sensed = 0x3ff;
        }

        FeCtrl0Regs.EADCDAC.bit.DAC_VALUE = iv.i_target_sensed << 4; //disregard dithering bits.
}
```

5.5 Input Power and RMS Current Measurement

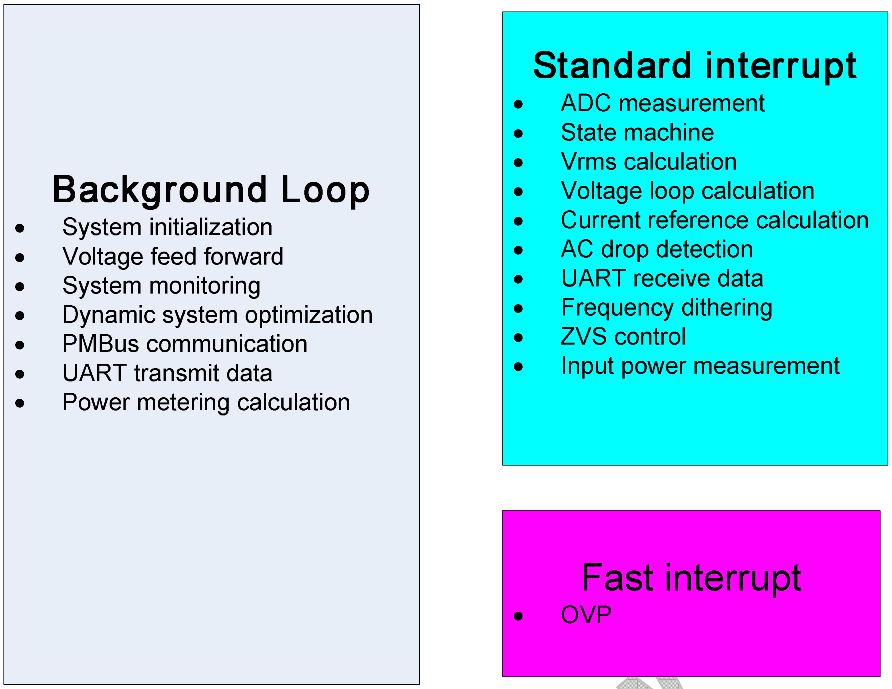
The real-time energy consumption measurement, including real input power and input RMS current measurements, are becoming more and more important in today's "green world" environment. These measurements con be used to adjust power delivery and optimize energy usage. Traditionally the input power and current are measured by a dedicated power metering chip. While the power metering chip proved to be sufficient, it definitely adds extra cost and design effort. Since UCD3138 already has analog to digital converters (ADC) and fast CPU, it can be used to do input power and current measurement as well. A method has been developed by TI to use the existing controller in a PFC application to do input power and Irms measurements at the same time. It has the following features:

- It uses existing PFC controller chip and hardware, resulting extremely low cost
- Simple calibration procedure requires only 2 points
- Firmware EMI current compensation
- Firmware current sense phase shift compensation
- Optimized mathematical calculation with minimum overhead on CPU usage, and no impact on normal PFC control

This method has been tested on different PFC units, with different EMI CAPs, under different operating temperatures, and with different distorted AC inputs. The measurement accuracy requirements are met in all cases. Please refer to the application note [6] for details.

6 Firmware structure

The firmware is divided as 3 major parts: background loop, standard interrupt loop and fast interrupt loop, as shown below:



Background Loop
- System initialization
- Voltage feed forward
- System monitoring
- Dynamic system optimization
- PMBus communication
- UART transmit data
- Power metering calculation

Standard interrupt
- ADC measurement
- State machine
- Vrms calculation
- Voltage loop calculation
- Current reference calculation
- AC drop detection
- UART receive data
- Frequency dithering
- ZVS control
- Input power measurement

Fast interrupt
- OVP

Figure 5. Firmware Structure

6.1 Background Loop

The firmware starts from function main(). In this function, after the system initialization, it goes to an infinite loop. All the non-time critical tasks are put in this loop, it includes:

- Calculate voltage feed forward
- System monitoring
- Dynamic system optimization
- PMBus communication
- UART transmit data
- Power metering calculation

The user can always add any additional non-time critical functions in this loop

```
void main()
{
            MiscAnalogRegs.IOMUX.all = 0; //enable JTAG
            look_for_interrupted_dflash_erase(); //Check to see if the last DFLASH erase was interrupted
            pmbus_write_restore_default_all(); //load PFC configuration from data flash
            init_miscellaneous();
            init_adc_polled();
            init_uart();
            init_front_ends();
            init_dpwms();
            init_filters();
            init_loop_mux();
            init_fault_mux();
            init_timer_interrupt();
            init_pmbus();

            string_out_0("\033[2J"); //clear screen
```

```
        for(;;)
        {
                pmbus_handler();

                emi_current_calculation();

                pmbus_handler();

                input_current_calculation();

                pmbus_handler();

                input_power_calculation();

                pmbus_handler();

                voltage_feed_forward();

                pmbus_handler();

                system_monitoring();

                pmbus_handler();

                if(iv.supply_state == STATE_PFC_ON)
                {
                        dynamic_system_optimization();//change compensation based on Vin
                }

                pmbus_handler();

                if (erase_segment_counter > 0)
                {
                        erase_task();           //  Handle the DFlash segment erases
                }

                pmbus_handler();

                if(uart_tx_timeout >= UART_TX_TIME)
                {
                        output_primary_secondary_message();
                }
                else
                {
                        process_uart_rx_data();
                }
        }
}
```

6.2 Standard Interrupt Loop (IRQ)

Standard interrupt loop is triggered by a timer at every 20µs. It is used to handle all the time critical tasks. It includes:

- ADC measurements
- PFC State machine
- Vin_rms calculation
- Voltage loop calculation
- Current reference calculation

- Vin drop detection
- UART receive data
- Frequency dithering
- ZVS control
- Input power measurement

However, to handle all these tasks in 20µs will cause interrupt overflow. To deal with this issue, the tasks distribution state machine is used to handle different task at different time interval.

6.2.1 Tasks Distribution State Machine

```
void standard_interrupt(void)
{
        poll_adc();
        rectify_vac();
        calculate_current_target_shunt();

#ifdef ZVS_CONTROL
        if(ipm_or_zvs) //switch between ipm and zvs, IRQ will overflow if do them together
        {
                input_power_measurement();
                ipm_or_zvs = 0;
        }
        else
        {
                handle_zvs_control();
                ipm_or_zvs = 1;
        }
#else
        input_power_measurement();
#endif

        switch(iv.interrupt_state)
        {
                case I_STATE_1 :
                                handle_voltage_loop();
                                iv.interrupt_state = I_STATE_2;
                                break;

                case I_STATE_2 :
                                half_cycle_processing();
                                iv.interrupt_state = I_STATE_3;
                                break;

                case I_STATE_3 :
                                check_ac_drop();
                                iv.interrupt_state = I_STATE_4;
                                break;

                case I_STATE_4 :
                                uart_receive_data();
                                iv.interrupt_state = I_STATE_5;
                                break;

                case I_STATE_5 :
                                supply_state_handler(); //run PFC state machine

                                if(zvs_flag == 0) //only do dither when zvs control is disabled
                                {
                                        frequency_dithering();
```

```
                    }

                    iv.interrupt_state = I_STATE_1;
                    break;

            default: //if it's in an illegal state
                    iv.interrupt_state = I_STATE_1; //start it up again
                    break;
        }
        TimerRegs.T16PWM0CMPCTRL.all = 3; //clear interrupt bit by a read/write.
}
```

6.2.2 PFC State Machine

PFC state machine is only one of the tasks in standard interrupt, it is called every 100µs. A typical PFC state machine is shown below:

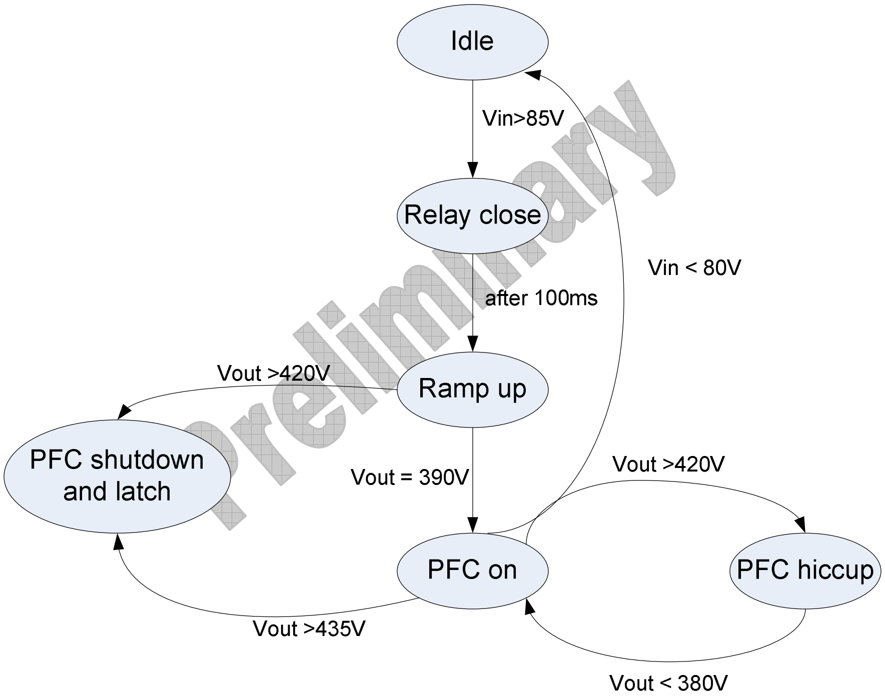


Figure 6. PFC State Machine

As soon as Vin is greater than 85V, the relay closes and the PFC starts up. A 100ms delay is added after relay closes to deal with the relay bouncing issue. After that, the PFC will gradually ramp up its output voltage until Vout reaches its setpoint. At this point, the PFC enters its final regulation state and will stay there until some abnormal conditions occurs, such as Vout over voltage or Vin under voltage.

```
inline void supply_state_handler(void)
{
        switch(iv.supply_state)
        {
            case STATE_IDLE :
                    idle_state_handler();
                    break;

            case STATE_RELAY_BOUNCE:
```

```
                        relay_bounce_state_handler();
                        break;

                case STATE_RAMP_UP :
                        ramp_up_state_handler();
                        break;

                case STATE_PFC_ON:
                        pfc_on_state_handler();
                        break;

                case STATE_PFC_HICCUP:
                        pfc_hiccup_state_handler();
                        break;

                case STATE_PFC_SHUT_DOWN:
                        pfc_shut_down_state_handler();
                        break;

                default:
                        break;
        }
}
```

6.3 Fast Interrupt (FIQ)

The FIQ is triggered by the comparator on AD06 (Comparator F). Since DPWM1B is already turned off to protect the PFC, what the FIQ does is only to report an OVP failure through a GPIO and set the PFC state into a shut down latched state. The customer can always add more time critical tasks in function:

```
#pragma INTERRUPT(fast_interrupt,FIQ)
void fast_interrupt(void)
{
        volatile int32 temp;

        turn_off_pfc();

        iv.supply_state = STATE_PFC_SHUT_DOWN;

        temp = FaultMuxRegs.FAULTMUXINTSTAT.all; //read to clear the interrupt flag
}
```

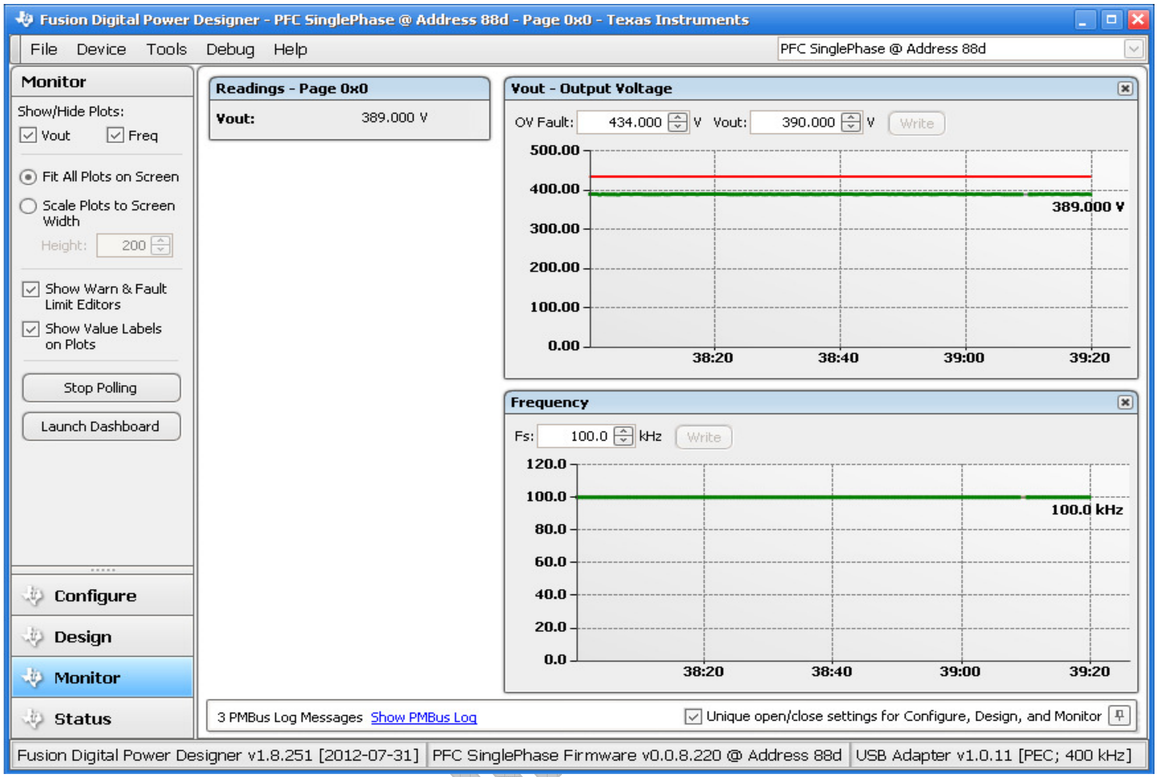
7 Graphical User Interface (GUI)

A graphical user interface (GUI) named “Fusion Digital Power Designer” is provided by Texas Instruments to facilitate UCD3138 controlled power converter designed. By talking to the GUI through the PMBus, the PFC operating status can be monitored, its operation setpoints can be configured, and the control loop can be tuned on the fly.

The GUI is deigned to support the most popular topologies, including PFC. Different topologies will have different interfaces. A setup id is used in the PFC firmware to tell the GUI that this is a PFC, so that when the GUI starts, it will open a interface to accommodate the PFC topology. In addition, the setup id specifies the PFC topology (single phase, interleaved or bridgeless), and it also includes the hardware modules used in the PFC current loop: which front end, which CLA and which DPWM modules are used. The following setup id is used in the single phase PFC EVM PWR026:

```
#define SETUP_ID        "VERSION1|PFC001"
```

In this case, the “PFC001” is defined as a single phase PFC, with FE0, CLA1 and DPWM1 consist the current loop.



Show/Hide Plots: ☑ Vout ☑ Freq

● Fit All Plots on Screen

○ Scale Plots to Screen Width

Height: 200

☑ Show Warn & Fault Limit Editors

☑ Show Value Labels on Plots

Vout: 389.000 V

OV Fault: 434.000 V Vout: 390.000 V

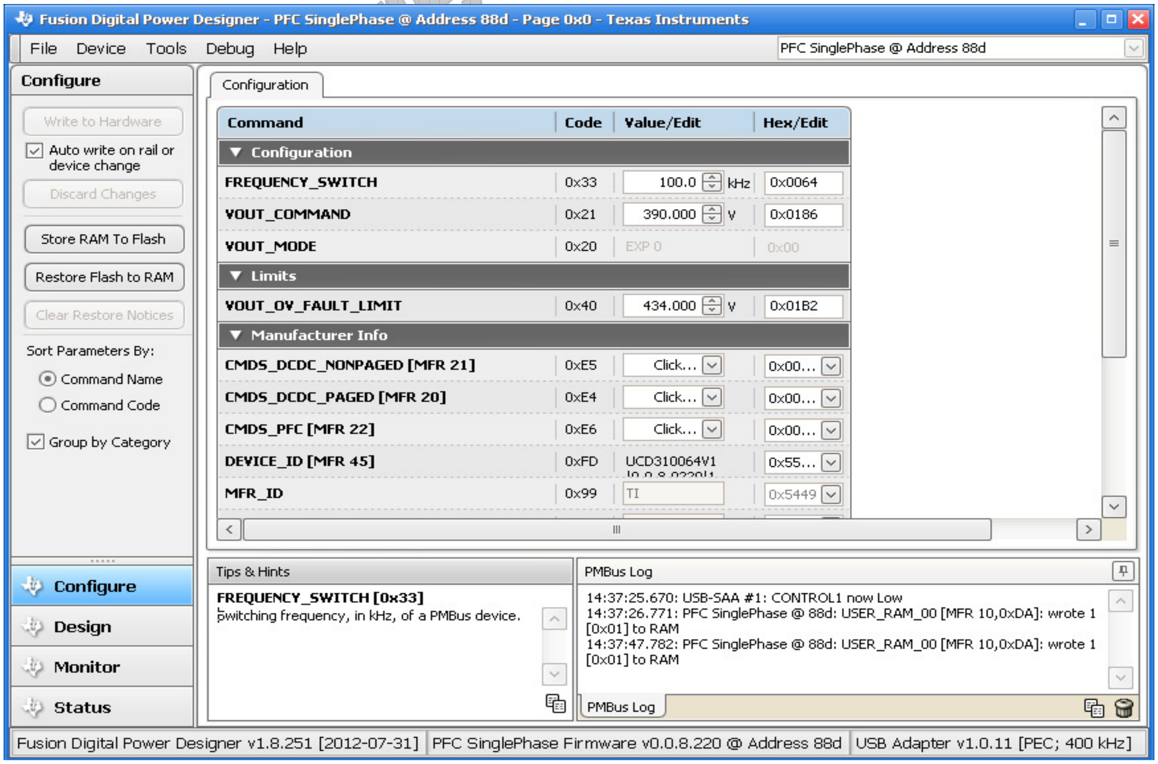
Fs: 100.0 kHz

3 PMBus Log Messages Show PMBus Log

☑ Unique open/close settings for Configure, Design, and Monitor

Fusion Digital Power Designer v1.8.251 [2012-07-31] | PFC SinglePhase Firmware v0.0.8.220 @ Address 88d | USB Adapter v1.0.11 [PEC; 400 kHz]

Figure 7. Monitor PFC Operating Status



☑ Auto write on rail or device change

Sort Parameters By:

● Command Name

○ Command Code

☑ Group by Category

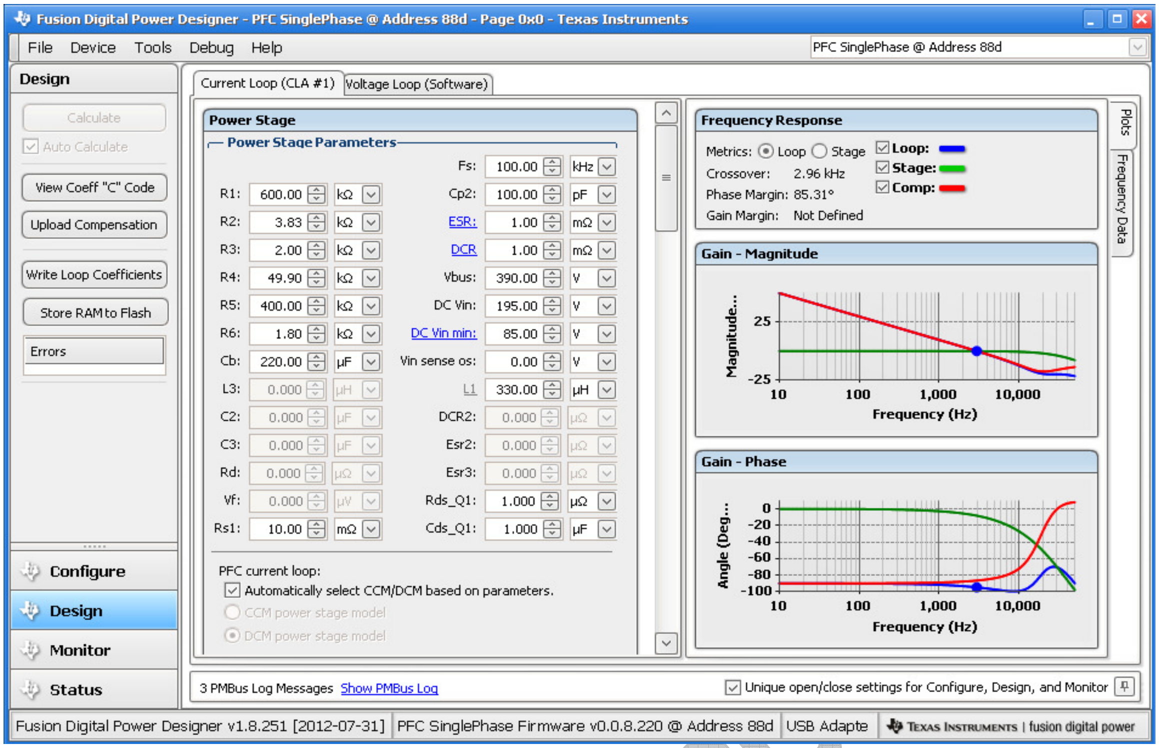
Command	Code	Value/Edit	Hex/Edit
▼ Configuration			
FREQUENCY_SWITCH	0x33	100.0 kHz	0x0064
VOUT_COMMAND	0x21	390.000 V	0x0186
VOUT_MODE	0x20	EXP 0	0x00
▼ Limits			
VOUT_OV_FAULT_LIMIT	0x40	434.000 V	0x01B2
▼ Manufacturer Info			
CMDS_DCDC_NONPAGED [MFR 21]	0xE5	Click...	0x00...
CMDS_DCDC_PAGED [MFR 20]	0xE4	Click...	0x00...
CMDS_PFC [MFR 22]	0xE6	Click...	0x00...
DEVICE_ID [MFR 45]	0xFD	UCD310064V1	0x55...
MFR_ID	0x99	TI	0x5449

FREQUENCY_SWITCH [0x33]
Switching frequency, in kHz, of a PMBus device.

```
14:37:25.670: USB-SAA #1: CONTROL1 now Low
14:37:26.771: PFC SinglePhase @ 88d: USER_RAM_00 [MFR 10,0xDA]: wrote 1
[0x01] to RAM
14:37:47.782: PFC SinglePhase @ 88d: USER_RAM_00 [MFR 10,0xDA]: wrote 1
[0x01] to RAM
```

Fusion Digital Power Designer v1.8.251 [2012-07-31] | PFC SinglePhase Firmware v0.0.8.220 @ Address 88d | USB Adapter v1.0.11 [PEC; 400 kHz]

Figure 8. Configure PFC Operation Setpoints



Fs: 100.00 kHz

R1:	600.00	kΩ	Cp2:	100.00	pF
R2:	3.83	kΩ	ESR:	1.00	mΩ
R3:	2.00	kΩ	DCR	1.00	mΩ
R4:	49.90	kΩ	Vbus:	390.00	V
R5:	400.00	kΩ	DC Vin:	195.00	V
R6:	1.80	kΩ	DC Vin min:	85.00	V
Cb:	220.00	µF	Vin sense os:	0.00	V
L3:	0.000	µH	L1	330.00	µH
C2:	0.000	µF	DCR2:	0.000	µΩ
C3:	0.000	µF	Esr2:	0.000	µΩ
Rd:	0.000	µΩ	Esr3:	0.000	µΩ
Vf:	0.000	µV	Rds_Q1:	1.000	µΩ
Rs1:	10.00	mΩ	Cds_Q1:	1.000	µF

PFC current loop:
☑ Automatically select CCM/DCM based on parameters.
○ CCM power stage model
● DCM power stage model

Metrics: ● Loop ○ Stage
Crossover: 2.96 kHz
Phase Margin: 85.31°
Gain Margin: Not Defined
☑ Loop: 🟦
☑ Stage: 🟩
☑ Comp: 🟥

3 PMBus Log Messages Show PMBus Log
☑ Unique open/close settings for Configure, Design, and Monitor

Fusion Digital Power Designer v1.8.251 [2012-07-31] | PFC SinglePhase Firmware v0.0.8.220 @ Address 88d | USB Adapte | Texas Instruments | fusion digital power

Figure 9. Tune PFC Control Loop

The current and voltage loop can be tuned through GUI. As shown in Figure 9, This GUI provides interface to tune the current and voltage control loop compensator, it also shown the loop bode plot, as well as bandwidth, phase margin, and gain margin. The loop tuning is much simplified. For details of the GUI, please refer to [7].

8 PFC Tuning and THD Reduction

PFC current loop tuning can be a time consuming and challenging task for the PFC design engineer. It requires the current waveform not only to be stable, but also to be smooth with very low THD and high PF. It gets more and more challenging with the ever increasing THD and PF requirements. The digital controller provides more flexibility and additional ways to achieve these increasing performance requirements. To make this task easier TI provides a GUI that greatly simplifies the work involved in these tasks. Additionally, application note [8] also provides a step by step guide of how to tune the current loop of a UCD3138 controlled PFC, it also summarizes some of the most common but effective methods to reduce the current distortion in a digitally controlled PFC. All the methods are analyzed and tested.

Reference:

[1] UCD3138 datasheet
[2] TI application note: Design a UCD3138 Controlled Interleaved PFC
[3] TI application note: Design a UCD3138 Controlled Bridgeless PFC
[4] SLUU995: UCD3138 Digital Power Peripherals Programmer’s Manual
[5] SLUA644, TI application report
[6] TI application note: A Low Cost Input Power and RMS Current Measurement Solution
[7] TI user guide: Isolated Fusion GUI User Guide
[8] TI application note: UCD3138 PFC Tuning.