Designing a UCD3138 Controlled Bridgeless 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 a 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 of how to design a UCD3138 controlled bridgeless PFC. It covers the hardware interface, voltage loop and current loop implementation, protection, firmware structure, internal state machines, as well as some advanced features. Finally, a graphical user interface (GUI) and how to tune a new designed PFC are presented. For single phase or interleaved PFC design, please refer to application note [2] and [3].

Contents

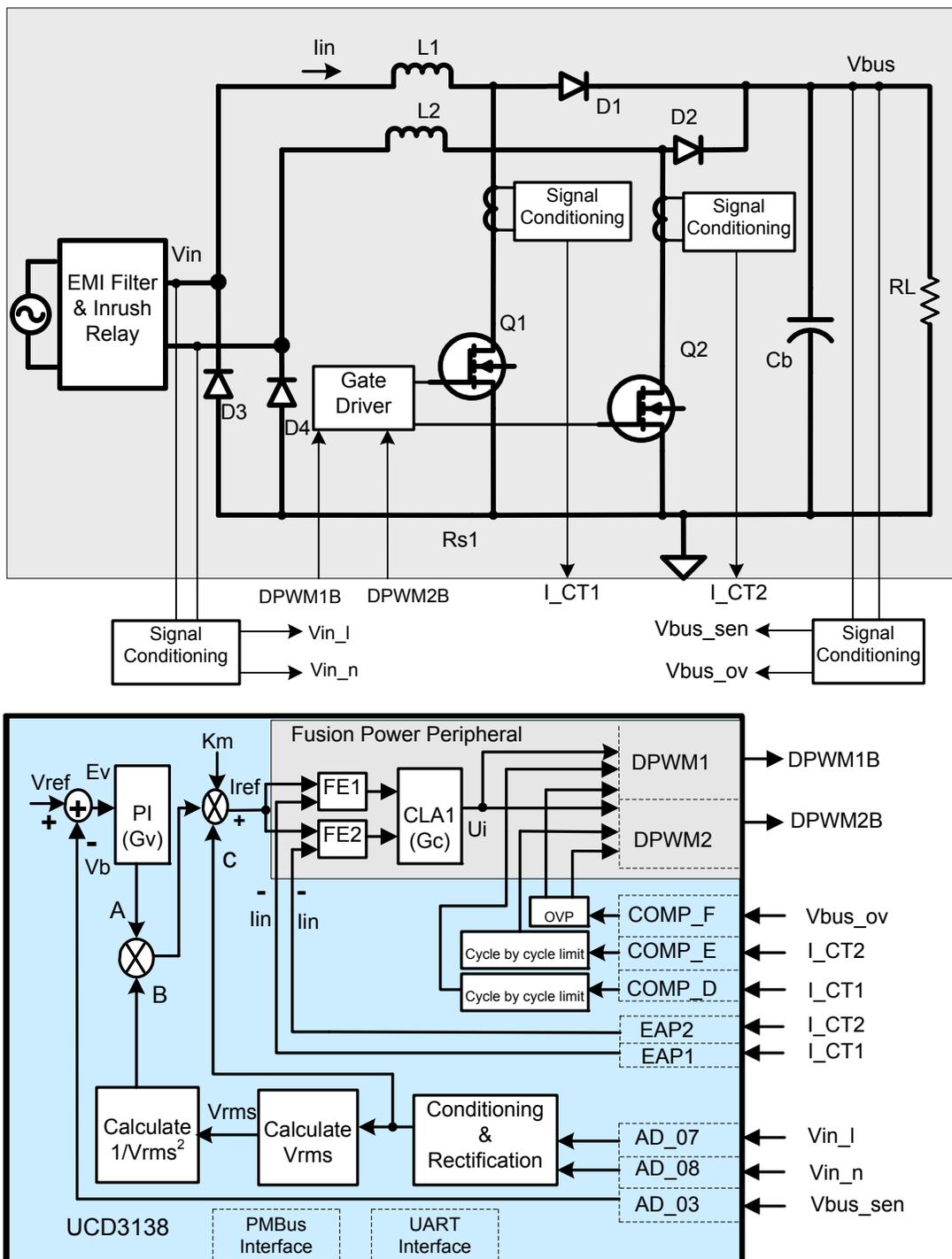# 1 Overview

1.1 Block Diagram

Figure 1. UCD3138 controlled bridgeless PFC block diagram

Figure 1 is an example of block diagram of a bridgeless PFC controlled by UCD3138. The input voltage Vin 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 connect to an on chip analog comparator COMP_F for over voltage protection (OVP). Two current transformers are used to sense the MOSFET instantaneous current and their output I_CT1 and I_CT2 are connected to EAP1 and EAP2 respectively for current loop control. In addition, I_CT1 and I_CT2 are also connected to two on chip analog comparators COMP_E and COMP_D for cycle-by-cycle current protection. The control loop generates two PWM outputs DPWM1B and DPWM2B to drive the MOSFETs through gate driver.

An average current mode control is used for input current regulation: current reference is calculated based on Vin, voltage loop output and voltage feed forward. This averaged current reference is then translated to instantaneous signal as if they were sensed at the middle of CT output. The translated reference is then compared to the middle point value of CT output, 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 accommodates with TI's PFC evaluation board PWR026. It is not necessary to follow this configuration. For example, I_CT1/I_CT2 can be connected to a different EPA channel, a different CLA can be used for compensation, and PFC can be driven by different DPWM output as well. However, it is recommended using the similar configuration as PWR026, so that most of the PWR026 source code can be reused and the design period can be much shorter.

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: $K_{vin} \leq \dfrac{2.5}{\sqrt{2} * V_{in\_max}}$

For Vout, the voltage divider: $K_{vout} \leq \dfrac{2.5}{V_{out\_max}}$

For current transformer: the maximum middle point of I_CT1 and I_CT2 should less than 1.6V, and the maximum peak value should less than 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 is still 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 in steady state range
        {
                steady_state_error = iv.vbus_target - (iv.vbus_filtered >> 6);
                iv.pis.p = iv.pis.kp * steady_state_error;
                iv.pis.i = iv.pis.i + (iv.pis.ki * steady_state_error);
        }
        else
        {       //non-linear gain for Voltage loop
                iv.pis.p = iv.pis.kp_nl * error;//Q15*Q12
                iv.pis.i = iv.pis.i + (iv.pis.ki_nl * error);
        }

        if(iv.ac_drop_recovery_not_complete)
        {
                if(((error < 0) && (iv.pis.i > 0)) || ((error > 0) && (iv.pis.i < 0)))
                {
                        iv.pis.i = 0; //reset the integral just when AC voltage has restored
                        iv.ac_drop_recovery_not_complete = 0; //AC drop recovery completed
                }
        }

        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 of input voltage, needs to be calculated first. For average current mode controlled PFC, the average current reference is calculated as:

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

while:
K_m: multiplier gain
A: Voltage loop output
B: $1/V_{in\_rms}^2$
C: sensed input voltage $K_{vin}*V_{in}$

However, for bridgeless with CT sensing, because the current transformer is placed right above the switch, it only senses the switch current, which is only the rising part of the inductor current. So the calculated average current reference needs to be translated to instantaneous current signal as if it is sensed by CT.

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.
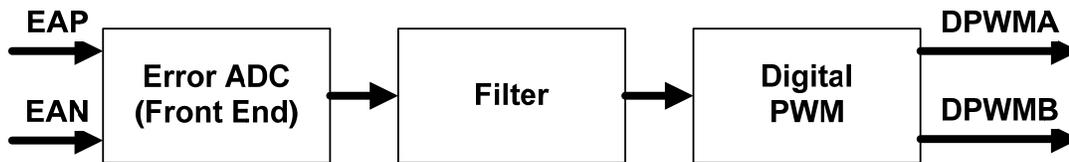

Figure 2. Current loop

3.2 Multiplier Gain K_m

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

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

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

For sine wave input, this can be written as,

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

3.3 Vin Sensing and Rectification

The input AC voltage is measured by separately sensing the line and neutral voltages with referencing to internal power ground. By comparing the sensed line and neutral signals, the firmware will know whether this is a positive half-cycle or a negative half-cycle. During a positive half-cycle, the first DC/DC boost circuit, L1-Q1-D1, is active and the boost current returns to AC neutral through diode D4. During a negative half-cycle, the second DC/DC boost circuit, L2-Q2-D2, is active and the boost current return to AC line through diode D3.

A bridgeless PFC essentially consists of two-phase boost circuits, but only one phase is active at any moment. Compared with conventional single phase PFC using the same power devices, the switching losses of a bridgeless PFC and a single phase PFC should be the same. However, bridgeless PFC current passes only one slow diode (D4 for positive half-cycle and D3 for negative half-cycle) instead of two at any time. Thus, the efficiency improvement relies on the conduction loss difference between one diode and two. Moreover, the bridgeless PFC efficiency can be further improved by turning the inactive switch fully on. For example, during a positive cycle, while Q1 is controlled by the PWM signal, Q2 can be fully turned on. Since the voltage drop on MOSFET Q2 may be lower than diode D4 when the flowing current is below certain a value, the return current partially or totally flows through L1-D1-RL-Q2-L2, and then back to AC source. The conduction loss is decreased and the circuit efficiency can be improved, especially at light load. Similarly, during a negative cycle, Q1 is fully turned on while Q2 is switching. The control waveform for Q1 and Q2 is shown in Figure 3.

Figure 3. PWM waveform for bridgeless PFC
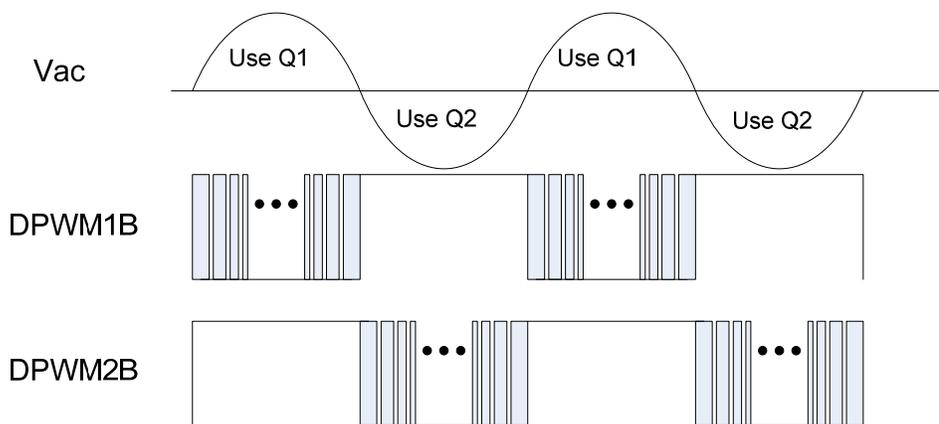
The following is the code of this mechanism for bridgeless PFC:

```
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
```

```
                Dpwm1Regs.DPWMCTRL1.bit.GPIO_B_VAL = 0;  //first turn off line
                Dpwm2Regs.DPWMCTRL1.bit.GPIO_B_EN = 1;  //then turn off neutral
                FeCtrl1Regs.EADCDAC.bit.DAC_VALUE = iv.i_target_sensed << 4; //use EADC1
                LoopMuxRegs.FILTERMUX.bit.FILTER1_FE_SEL = 1; //use EADC1 to drive filter 1

                if(iv.pwm_on == 1)
                {
                        Dpwm1Regs.DPWMCTRL1.bit.GPIO_B_EN = 0;  //now turn on line PWM

                        //if it is really positive cycle, not a glitch, turn on the 2nd phase switch
                        if(iv.vin_filtered > 80)
                        {
                                Dpwm2Regs.DPWMCTRL1.bit.GPIO_B_VAL = 1;
                        }
                        else
                        {
                                Dpwm2Regs.DPWMCTRL1.bit.GPIO_B_VAL = 0;
                        }
                }
                else
                {
                        Dpwm2Regs.DPWMCTRL1.bit.GPIO_B_VAL = 0;  //neutral drive low
                        Dpwm1Regs.DPWMCTRL1.bit.GPIO_B_EN = 1;  //now disable line PWM
                }
        }
        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

                Dpwm2Regs.DPWMCTRL1.bit.GPIO_B_VAL = 0;  //first turn off neutral
                Dpwm1Regs.DPWMCTRL1.bit.GPIO_B_EN = 1;  //then turn off line PWM
                FeCtrl2Regs.EADCDAC.bit.DAC_VALUE = iv.i_target_sensed << 4; //use EADC2
                LoopMuxRegs.FILTERMUX.bit.FILTER1_FE_SEL = 2; //use EADC2 to drive filter 1

                if(iv.pwm_on == 1)
                {
                        Dpwm2Regs.DPWMCTRL1.bit.GPIO_B_EN = 0;  //now turn on neutral PWM

                        //if it is really negative cycle, not a glitch, turn on the 1st phase switch
                        if(iv.vin_filtered > 80)
                        {
                                Dpwm1Regs.DPWMCTRL1.bit.GPIO_B_VAL = 1;
                        }
                        else
                        {
                                Dpwm1Regs.DPWMCTRL1.bit.GPIO_B_VAL = 0;
                        }
                }
                else
                {
                        Dpwm1Regs.DPWMCTRL1.bit.GPIO_B_VAL = 0;  //line drive low
                        Dpwm2Regs.DPWMCTRL1.bit.GPIO_B_EN = 1;  //now disable neutral PWM
                }
        }
        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:

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

In discrete format:

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

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

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_rms2 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_rms2 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 calculate 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 Average Current Reference

Now we got A, B, 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 average current reference. As mentioned at the 1.1, this average current reference needs to be translated to instantaneous current as if it is sensed at the middle point of CT output.

3.7 Translate average reference to instantaneous reference

For PFC with CT sensing, because the current transformer is placed right above the switch, it only senses the switch current, which is only the rising part of the inductor current. For digital control implementation, this switch current signal is measured at the middle of PWM on time T_a. It is an instantaneous value, represented as I_{sense} in Figures 4a and 4b below. The measured switching current I_{sense} is equal to the average PFC inductor current only when the current is continuous. When the current becomes discontinuous, I_{sense} is not equal to the average PFC inductor current any more. In order to compute the inductor average current, the relationship between the middle point sensing current I_{sense} and the average inductor current over a switching period should be derived and be applicable for both continuous conduction mode (CCM) and discontinuous conduction mode (DCM).
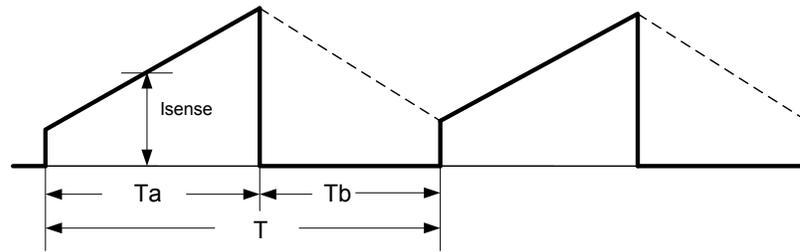
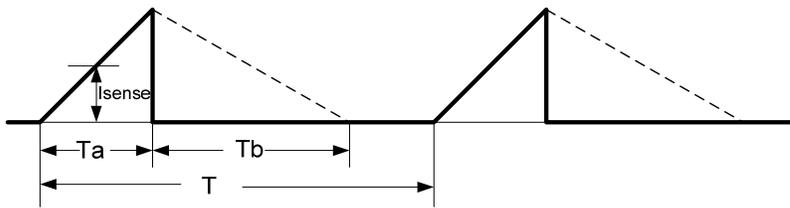**Figure 4a. Sensed current waveform at continuous conduction mode**

Figure 4b. Sensed current waveform at discontinuous conduction mode

For a boost-type converter in steady state operation, the volt-second of the boost inductor should maintain balance in each switching period:

$$T_a * V_{in} = T_b * (V_O - V_{in}) \qquad (7)$$

Here T_a is the current rising time (PWM on time), T_b is the current falling time (PWM off time), V_{in} is input voltage, and V_o is output voltage, assuming all power devices are ideal.

From Figures above, we can calculate the inductor average current I_{ave} in terms of I_{sense}:

$$I_{ave} = I_{sense} * \frac{(T_a + T_b)}{T} \qquad (8)$$

Where T is the switching period.

Combine (7) and (8) together, we get:

$$I_{sense} = \frac{I_{ave} * T * (V_o - V_{in})}{T_a * V_o} \qquad (9)$$

Through Equation 9, the average inductor current I_{ave} is interpreted in an instantaneous switch current I_{sense}. I_{ave} is the desired current and I_{sense} is the current reference for current control loop. The real instantaneous switch current is sensed and compared with this reference, the error is sent to a fast error ADC (EAP), and finally the digitized error signal is sent to a digital compensator to close the current control loop.

```
inline void calculate_current_target_ct(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.cir_buff_ptr = (iv.cir_buff_ptr + 1) & 0x3f;

        iv.vbus_scaled = (iv.adc_avg[VBUS_CHANNEL] * VBUS_TO_VAC_SCALING) >> 15;

        iv.cla_output_filtered = (Uint32)Filter1Regs.FILTERYNREAD.bit.YN + iv.cla_output_filtered - (iv.cla_output_filtered >> 2);

        if(iv.vbus_scaled > iv.vin_filtered)
        {
                iv.numerator_1 = iv.vbus_scaled - iv.vin_filtered;
        }
        else
        {
                iv.numerator_1 = 0;
        }

        iv.numerator_2 = (iv.i_target_average * iv.numerator_1) >> 8;

        iv.numerator_3 = (iv.cir_buff[pointer] * iv.numerator_2);

        iv.denominator = ((iv.cla_output_filtered >> 6) * iv.vbus_scaled) >> 11;

        iv.i_target_sensed = (iv.numerator_3 / iv.denominator) + iv.i_target_offset;

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

3.8 Current Feed Back Front End Configuration

The Front End measures the difference between the current feed back signal and the current reference calculated in 3.7. It passes this digital error information to the filter. Those need to be configured are: which front end is used for current feedback signal sensing, what is the AFE_GAIN, which CLA filter is used and which DPWM module is used. UCD3138 is very flexible, the front end, filter and DPWM are multi-to-multi connection. The following code example is based on the PWR026 PFC EVM hardware: it uses Frond End1, Frond End2, CLA1, DPWM1 and DPWM2. Other configurations are also possible. For detail of how to configure UCD3138 digital peripherals, please refer to programmer manual [4].

```
void init_front_end1(void) //for CT1 sensing
{
        FeCtrl1Regs.EADCDAC.bit.DAC_VALUE = 0;
}

void init_front_end2(void) //for CT2 sensing
{
        FeCtrl2Regs.EADCDAC.bit.DAC_VALUE = 0;
}

void init_loop_mux(void)
{
        LoopMuxRegs.SAMPTRIGCTRL.bit.FE1_TRIG_DPWM1_EN = 1; //Use DPWM1 sample trigger for FE1
        LoopMuxRegs.SAMPTRIGCTRL.bit.FE2_TRIG_DPWM2_EN = 1; //Use DPWM2 sample trigger for FE2

        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.DPWM2_FILTER_SEL =1; //CLA1 is providing input to DPWM2
        LoopMuxRegs.DPWMMUX.bit.DPWM2_SYNC_SEL = 1;  //DPWM1 is the master for DPWM2
```

}

3.9 Current Loop Filter Configuration

The filter takes the error signal from 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 the already well tuned control parameters to close the 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 = 0x785000;//clamp to prevent CT from saturation 94%
        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, the well tuned filter parameters kp, ki, kd and alpha are stored in data flash. Upon power up, they are loaded from data flash to 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;


        FeCtrl1Regs.EADCCTRL.bit.AFE_GAIN = pfc_config_in_ram.FILTERMISC.bit.AFE_GAIN;
        FeCtrl2Regs.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;
        Dpwm2Regs.DPWMCTRL2.bit.SAMPLE_TRIG1_OVERSAMPLE =
                pfc_config_in_ram.FILTERMISC.bit.SAMPLE_TRIG1_OVERSAMPLE;
}
```

3.10 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 bridgeless PFC with CT sensing, since we need to sense the middle point of CT output, only DPWMB is used and configured as Triangular mode. In this mode, the PWM pulse is centered in the middle of the period, rather than starting at one end or the other. In Triangular Mode, only DPWM B is available. It is very easy to put a fixed sample trigger exactly in the center of the On-time, because the center of the on-time does not move in this mode. The following code example accommodates PWR026 PFC EVM, in which DPWM1B and DPWM2B are used and configured as Triangular mode.

```
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.DPWMCTRL0.bit.PWM_MODE = 3; //triangular mode
        Dpwm1Regs.DPWMCTRL1.bit.EVENT_UP_SEL = 0; //update right away

        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_dpwm2(void) // DPWM2B is used to drive 2nd phase
{
        Dpwm2Regs.DPWMCTRL0.bit.PWM_EN = 0;  //disable everything

        Dpwm2Regs.DPWMCTRL1.bit.GPIO_A_EN = 1; //turn off DPWM2A for now
        Dpwm2Regs.DPWMCTRL1.bit.GPIO_B_EN = 1; //turn off DPWM2B for now

        // Enable CBC and Blanking windows
        Dpwm2Regs.DPWMCTRL0.bit.CBC_PWM_AB_EN = 1; // Enable cycle by cycle current limit.
```

```
        Dpwm2Regs.DPWMCTRL0.bit.BLANK_B_EN = 1;      // Enable blanking
        Dpwm2Regs.DPWMBLKBBEG.all = 0x0000;
        Dpwm2Regs.DPWMBLKBEND.all = 0x0500;

        Dpwm2Regs.DPWMFLTCTRL.bit.B_MAX_COUNT = 2;
        Dpwm2Regs.DPWMFLTCTRL.bit.ALL_FAULT_EN = 1; //enable this for OVP

        Dpwm2Regs.DPWMCTRL2.bit.SAMPLE_TRIG_1_EN = 1; //enable sample trigger1

        Dpwm2Regs.DPWMCTRL0.bit.PWM_MODE = 3; //triangular mode
        Dpwm2Regs.DPWMCTRL0.bit.MSYNC_SLAVE_EN = 0; //master mode
        Dpwm2Regs.DPWMCTRL1.bit.EVENT_UP_SEL = 0; //update right away
        Dpwm2Regs.DPWMCTRL2.bit.SAMPLE_TRIG1_MODE = 0;//triggered at sample trigger register value

        Dpwm2Regs.DPWMCTRL0.bit.CLA_EN = 1;
        Dpwm2Regs.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
        Dpwm2Regs.DPWMPRD.all = iv.switching_period; //new period for new frequency
        Dpwm1Regs.DPWMSAMPTRIG1.all = (iv.switching_period >> 1) + (iv.sample_trigger_offset * 4);
         // halfway plus some for driver delays
        Dpwm2Regs.DPWMSAMPTRIG1.all = (iv.switching_period >> 1) + (iv.sample_trigger_offset * 4);
        // halfway plus some for driver delays;
        Dpwm1Regs.DPWMPHASETRIG.all = 0; //0 delay for next phase
}

void init_dpwms(void)
{
        init_dpwm1();
        init_dpwm2();
        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 as cycle-by-cycle based.

4.1 Software OVP Protection

This is pure software OVP protection. Vout is measured by 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 threshold. ADC continues monitoring Vout, PWM will turn back on once Vout drops below its setpoint. This allows the PFC enters a hiccup mode. This will be useful for OVP not caused by hardware failure, but by sudden operation condition change, such as 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;
        }
}
```

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 PWM automatically once get 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 serious hardware issue. For safety purpose, it will be latched there once shut down.

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.DPWM2FAULTDET.bit.PWMB_ACOMP_F_EN = 1;  // Connect ACOMP-F to DPWM-2
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 comparator COMP_D and COMP_E. It is cycle-by-cycle (CBC) based. Once the analog comparator is triggered, the PWM is chopped for the remaining 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
```

```
// Enable ACOMP-E pin and connect to current limit on DPWM-2
FaultMuxRegs.DPWM2CLIM.bit.ACOMP_E_EN = 1;        // Connect ACOMP-E to DPWM-2
FaultMuxRegs.ACOMPCTRL2.bit.ACOMP_E_SEL = 0;      // Use threshold register for trip
FaultMuxRegs.ACOMPCTRL2.bit.ACOMP_E_POL = 1;      // Above thresh to trip
FaultMuxRegs.ACOMPCTRL2.bit.ACOMP_E_THRESH = OC_COMPARATOR; // Trip value
```

5 Advanced Features

The following lists some of the common used features for bridgeless PFC. For more advanced PFC features, please refer to [2].

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 around 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
                Dpwm2Regs.DPWMPRD.all = iv.switching_period; //new period for new frequency

                Dpwm1Regs.DPWMSAMPTRIG1.all = (iv.switching_period >> 1) + (iv.sample_trigger_offset * 4);
                Dpwm2Regs.DPWMSAMPTRIG1.all = (iv.switching_period >> 1) + (iv.sample_trigger_offset * 4);
        }
}
```

5.2 AC Drop Detection

The AC drop detection algorithm is shown in Figure 5. Vin is sampled every 100µs. Its measurement is compared with a predetermined threshold "AC_DROP_V_RECT_THRESHOLD". If the consecutive samples

below this threshold greater than a predetermined number "AC_DROP_COUNT_MAX", then AC drop is detected, a AC drop signal is send out to host through a GPIO.

The threshold and number of consecutive samples will affect the sensitive of AC drop detection, they can be tuned base on requirement.
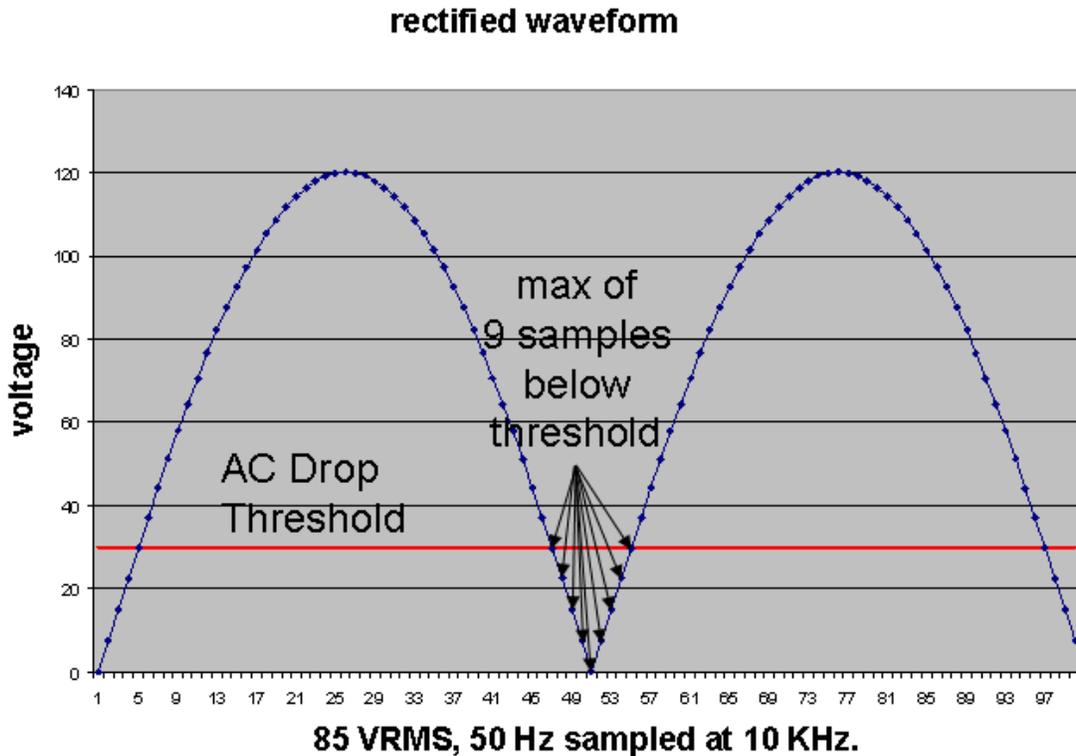

Figure 5. AC Drop Detection

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 X-CAP Reactive Current Compensation

Every PFC has an electromagnetic interference (EMI) filter at the input end. The X capacitors of the EMI filter will cause the AC input current leading AC 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 delayed a little bit so that the total AC current will match the input voltage. This can be achieved by delay 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_ct(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.cir_buff_ptr = (iv.cir_buff_ptr + 1) & 0x3f;

        iv.vbus_scaled = (iv.adc_avg[VBUS_CHANNEL] * VBUS_TO_VAC_SCALING) >> 15;

        iv.cla_output_filtered = (Uint32)Filter1Regs.FILTERYNREAD.bit.YN + iv.cla_output_filtered - (iv.cla_output_filtered >> 2);

        if(iv.vbus_scaled > iv.vin_filtered)
        {
                iv.numerator_1 = iv.vbus_scaled - iv.vin_filtered;
        }
        else
        {
                iv.numerator_1 = 0;
        }

        iv.numerator_2 = (iv.i_target_average * iv.numerator_1) >> 8;

        iv.numerator_3 = (iv.cir_buff[pointer] * iv.numerator_2);

        iv.denominator = ((iv.cla_output_filtered >> 6) * iv.vbus_scaled) >> 11;

        iv.i_target_sensed = (iv.numerator_3 / iv.denominator) + iv.i_target_offset;

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

This is a traditional way to compensate the X-cap reactive current; reference [5] provides another novel method with better performance.

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

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

Fast interrupt
- OVP
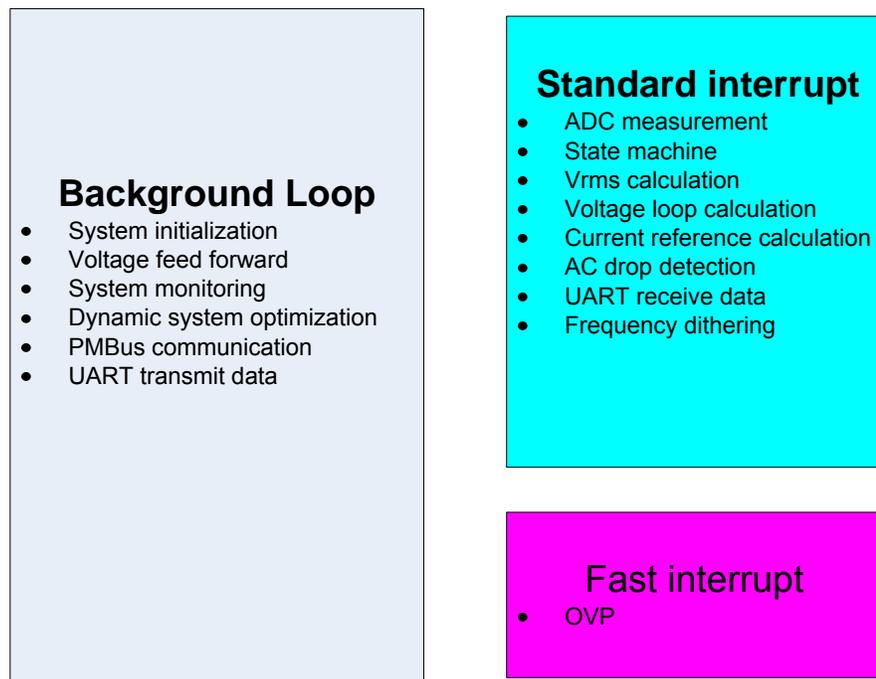
Figure 6. Firmware Structure

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

User can always add any 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();

        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

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_ct();

        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

                                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:
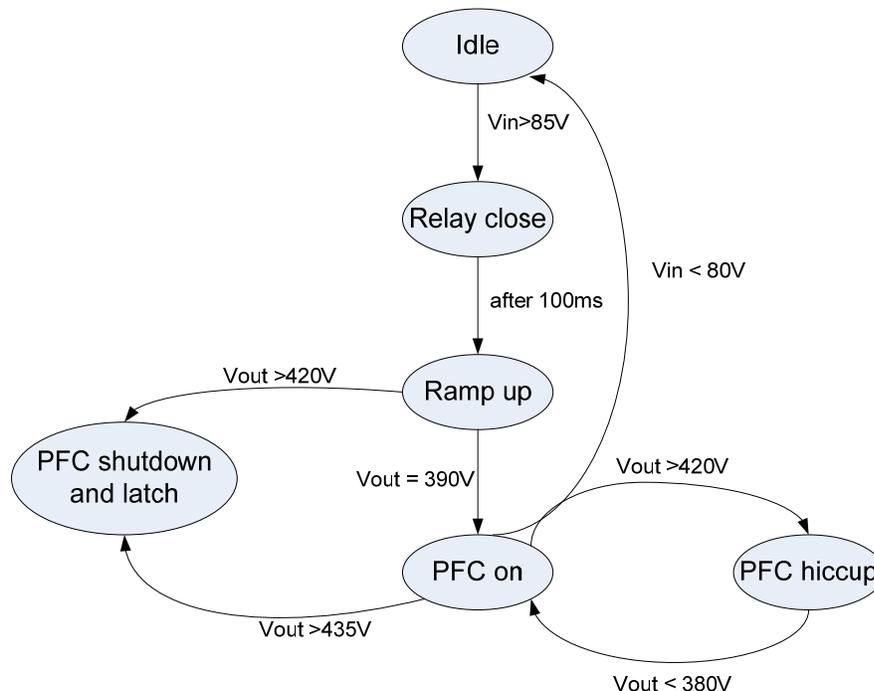
Figure 7. PFC State Machine

As soon as Vin is greater than 85V, the relay close and PFC is starting up. A 100ms delay is added after relay close to deal with the relay bouncing issue. After that, PFC will gradually ramp up its output voltage until Vout reaches its setpoint. At this point, 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 and DPWM2B are 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 latch 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 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 topology will have different interface. 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 accommodate to a PFC. In addition, the setup id specific the PFC topology (single phase, interleaved or bridgeless), and it also includes the hardware modules used in PFC current loop: which front end, which CLA and which DPWM are used. The following setup id is used in the bridgeless PFC EVM PWR026:

#define SETUP_ID   "VERSION1|PFC002"

In this case, the "PFC002" is defined as bridgeless PFC, with FE1, FE2, CLA1, DPWM1, DPWM2 consist the current loop.
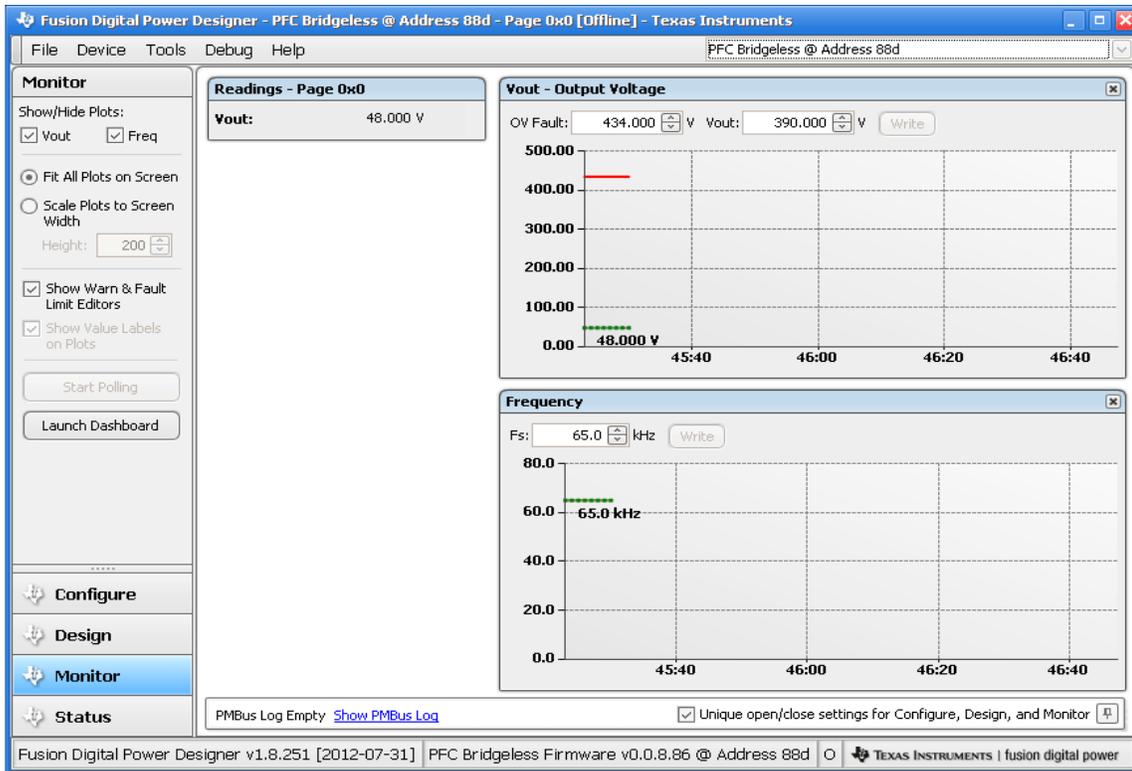
Figure 8. Monitor PFC Operating Status
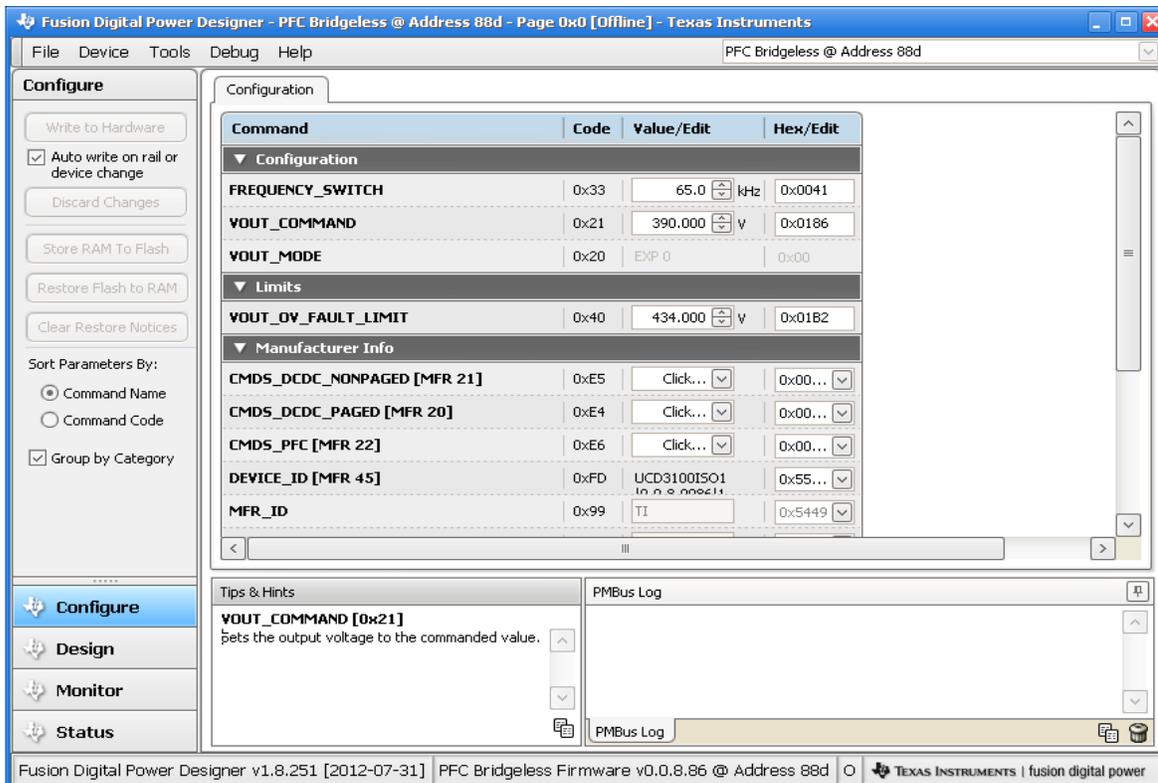
Figure 9. Configure PFC Operation Setpoints
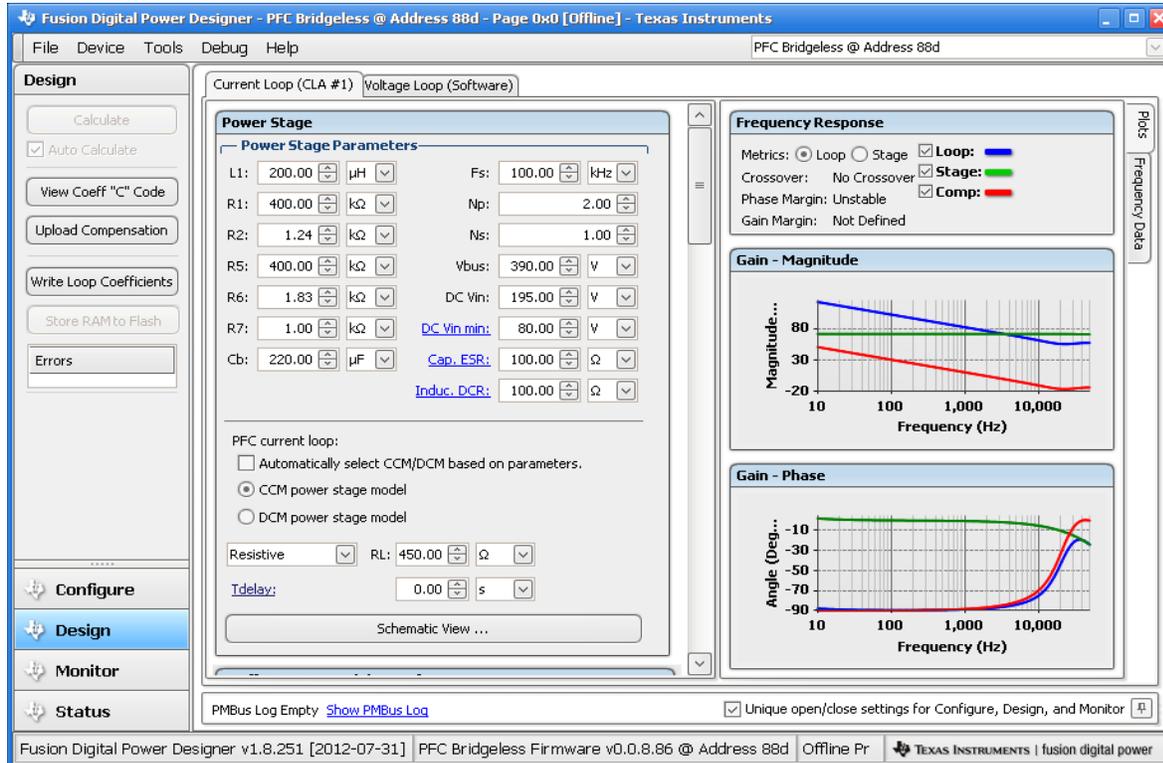
Figure 10. Tune PFC Control Loop

The current and voltage loop can be tuned through the GUI. As shown in Figure 10, 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 [6].

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 [7] 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.

Reference:

[1] UCD3138 datasheet

[2] TI application note: Designing a UCD3138 Controlled Single Phase PFC

[3] TI application note: Designing a UCD3138 Controlled Interleaved PFC

[4] SLUU995: UCD3138 Digital Power Peripherals Programmer's Manual

[5] TI application note: A Novel EMI Filter X-Cap Reactive Current Compensation Method to increase PF

[6] TI user guide: SLUA676

[7] TI application note: UCD3138 PFC Tuning.

IMPORTANT NOTICE

Products		Applications	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		