Freescale Semiconductor
Application Note

Document Number:AN4583
Rev. 0, 09/2012

MC56F84789 Peripherals Synchronization for Interleaved PFC Control

by: Jaroslav Musil
Automotive and Industrial Solutions Group

Contents

1 Introduction

In recent decades, there is a huge increase in applications of power electronics. These applications generally use an inverter that is connected to the rectified DC bus voltage. In high-power applications, the rectifier's charging current pulses distort the sine wave current taken from the outlet that comprehends undesirable higher number harmonics out on the power lines.

To avoid the distorted current consumption from the outlet, the power factor correction (PFC) modules have become one of the basic parts for power electronics. There are many kinds of PFC implementation. This application note's goal is to demonstrate which peripherals of Freescale's Digital Signal Controller (DSC) are used and how they are configured to control an interleaved (2-MOSFET) PFC.

Current trends are to put the entire system into one processor. The processor that controls the PFC is generally expected to drive an application like 3-phase motor control. That is why certain peripherals like PWM and ADC channels (typically used for motor control) are reserved and not used for the PFC control in this application note.

This application note explains

- how to set up a PWM module to control two MOSFETs (or IGBTs)
- how and where to generate trigger signals for the ADC module from the PWM module
- where to call the PFC control algorithm

2 Digital signal controller

One of the suitable DSCs is MC56F84789 of the MC56F84xx DSC family. This controller has the following features that the application can benefit from:

- DSC 56800Ex core
- Core and peripheral clock 100 MHz
- High resolution PWM module with a possibility of multiple triggers
- 16-bit SAR ADC with a possibility of DMA
- Timer with a large spectrum of configuration
- Two cross-bar units to interconnect signals between the peripherals
- And-Or-Invert to logically mix the signals among the peripherals
- 4-channel DMA
- Interrupt controller with priorities

The processor has many more modules but for this application note just the above-mentioned modules will be used.

Figure 1 shows the 2-MOSFET interleaved PFC schematics and Figure 2 how the processor signals are connected to power electronics on the board.

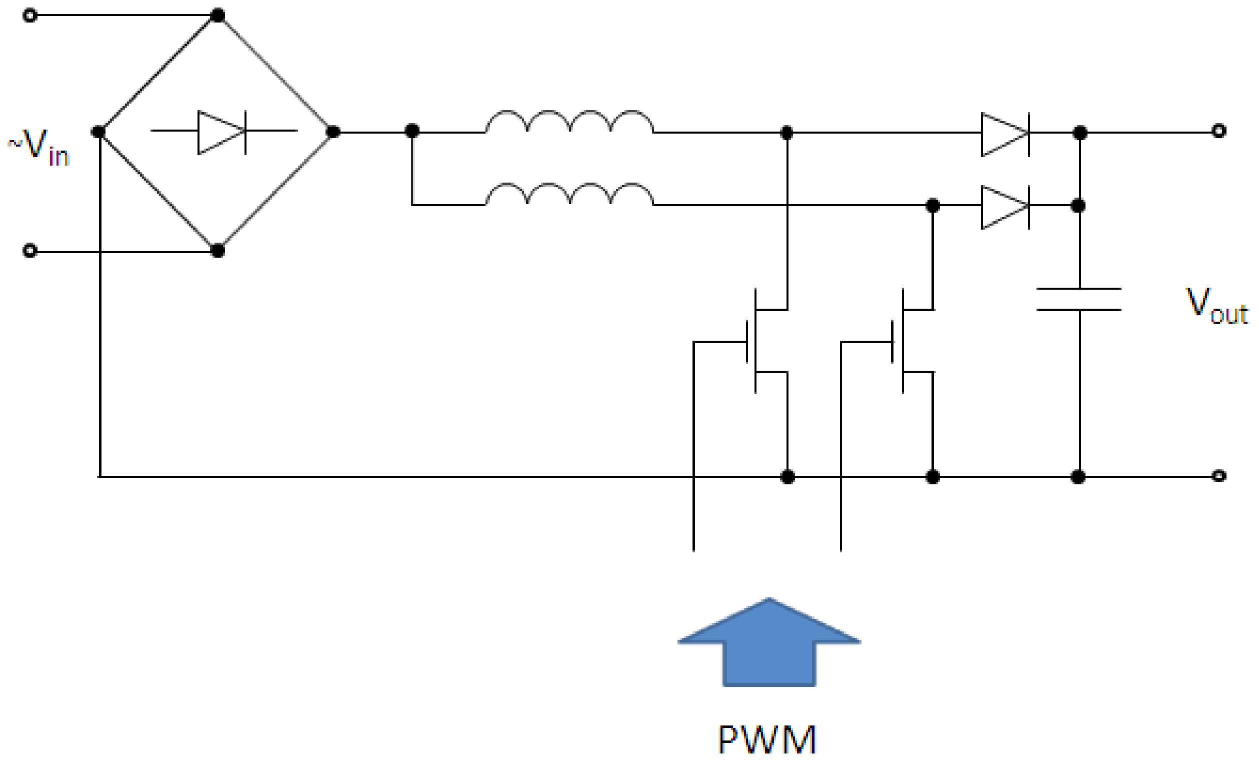

Figure 1. Interleaved PFC schematics

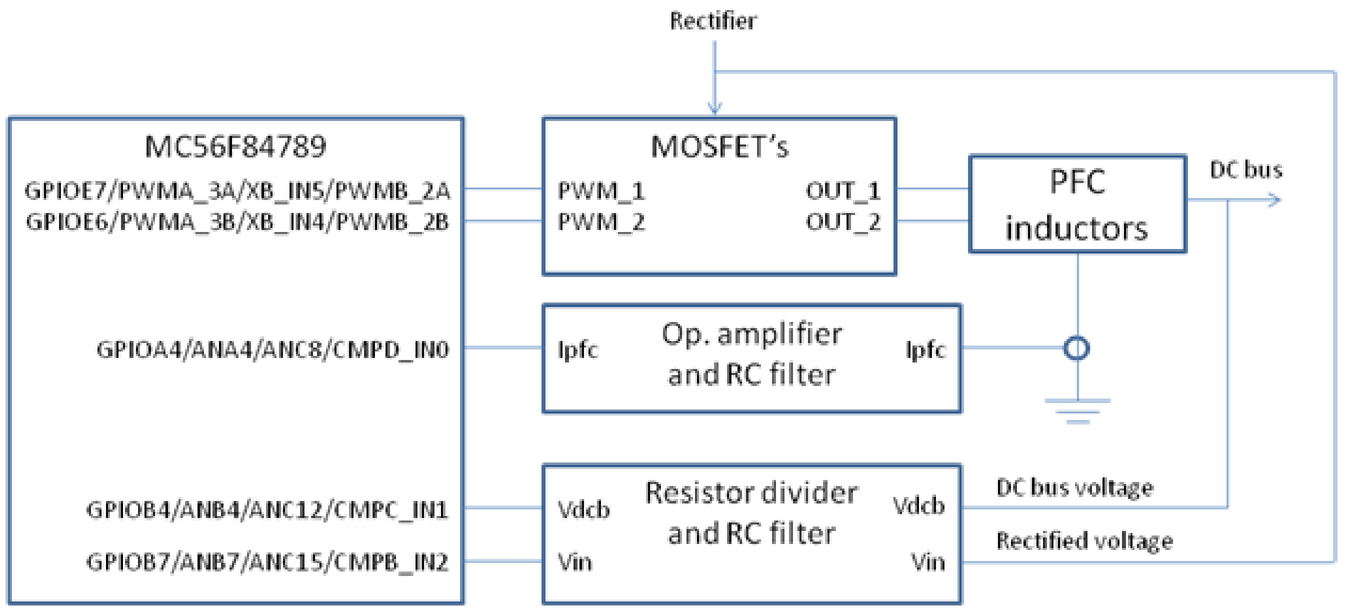
Rectifier
MC56F84789
GPIOE7/PWMA_3A/XB_IN5/PWMB_2A
GPIOE6/PWMA_3B/XB_IN4/PWMB_2B
GPIOA4/ANA4/ANC8/CMPD_IN0
GPIOB4/ANB4/ANC12/CMPC_IN1
GPIOB7/ANB7/ANC15/CMPB_IN2
MOSFET's
PWM_1
PWM_2
OUT_1
OUT_2
PFC inductors
DC bus
Ipfc
Op. amplifier and RC filter
Ipfc
Vdcb
Resistor divider and RC filter
Vdcb
Vin
Vin
DC bus voltage
Rectified voltage

Figure 2. Processor connection to the boards' circuitry

3 Steps in configuation flow

To configure DSC properly, the necessary steps that must be followed are:

- PWM Configuration—configures the PWM A sub-module 3 to generate interleaved signals for two MOSEFTs.
- PWM Triggers for ADC—sets up the points where the ADC will be triggered to sample the signals.
- ADC and DMA Configuration—configures the ADC and DMA modules to sample the desired signals.
- Signals Interconnection—configures the cross-bar switches and the AOI module to lead the trigger signals between the peripherals.
- PWM Start and Reading ADC Samples—sets up the interrupt to read the sampled values and call the algorithms.

4 PWM configuration

An interleaved PFC is driven with two MOSFETs in this application. The PWM signals for both MOSFETs will be center-aligned and shifted by 180 degrees to each other. The frequency of the PWM will be 80 kHz; the fast loop calculation will be in the ratio 1 to 4 with respect to the PWM frequency, therefore 20 kHz. The PFC will use the PWM A module, sub-modules 3; independent mode; non-inverted output logic. The sub-modules 0–2 are reserved for possible motor control.

The first step is to configure the PWM A sub-module 3 as shown in Figure 3.

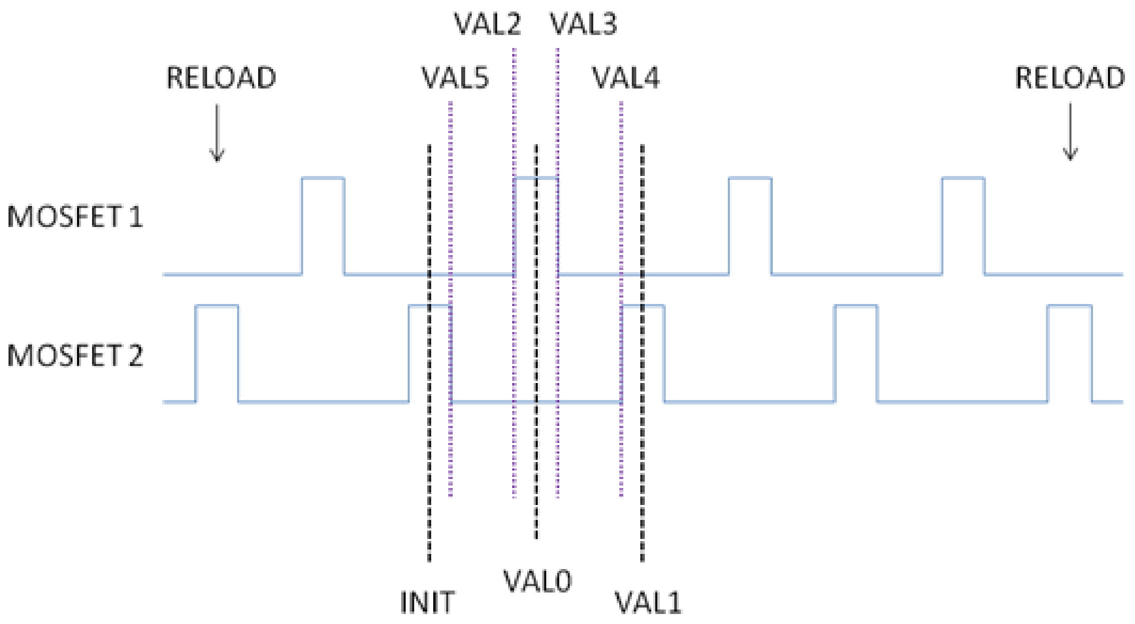

Figure 3. PWM configuration—interleaved control

4.1 Clock for PWM modules

To habilitate the clock for both PWM modules it is necessary to configure Peripheral Clock Register 3 (SIM_PCE3) from System Integration Module (SIM).

Syntax to enable the clock for PWM A sub-module 3:

```
SIM_PCE3  |= SIM_PCE3_PWMACH3;
```

4.2 PWM Control Register

PFC motors will use the full cycle on every fourth opportunity reload. The PWM clock frequency is maximum, so the prescaler is 1. The control registers will be set up in the following way:

```
PWMA_SM3CTRL = (PWMA_SM3CTRL_LDFQ_0 | PWMA_SM3CTRL_LDFQ_1 | PWMA_SM3CTRL_FULL);
```

4.3 PWM Control 2 Register

To set up the independent mode and the local reload from sub-module 3, the PWM Control 2 Register must be properly configured. The IPBus clock is used as the clock source. The register is configured as:

```
PWMA_SM3CTRL2 = PWMA_SM3CTRL2_INDEP;
```

4.4 PWM modulo setup

The PWM modulo to generate 80 kHz is derived from the 100 MHz clock. Thus, the modulo is 100 MHz / 80 kHz = 1250 ticks. The PWM modules have the INIT value where the counter starts and VAL1 value where the counter is reinitialized. So the INIT value will be set up to the negative value of the half modulo, and the VAL1 value will be set up to the positive value of the half modulo – 1. The reload is in the full cycle, so at the INIT value. To sum it up: INIT = –625 (0xFD8F), VAL1 = 624 (0x0270), VAL0 = 0.

Below is the way how to configure it:

```
PWMA_SM3INIT = 0xFD8F;
PWMA_SM3VAL1 = 0x0270;
PWMA_SM3VAL0 = 0x0000;
```

4.5 25% duty cycle initialization

In the final application, usually the duty cycle is initialized to 0. This example is to show how to initialize a non-zero duty cycle, so care must be taken if the signals are connected to the MOSFET under voltage.

To initialize the module with the 25% duty cycle, it is necessary to set up the VAL2 and VAL3 registers for MOSFET 1 and the VAL4 and VAL5 registers for MOSFET 2. The signals are phase inverted (180-degree shift) so the principle of applying the duty cycles will be: VAL2 is loaded with the negative value of the half modulo multiplied by 0.25; VAL3 is loaded with the positive value of the half modulo multiplied by 0.25. Therefore, VAL2 = –156 (0xFF64), VAL3 = 156 (0x009C). As the other signal is phase inverted, the duty cycle must be subtracted from 1 and the values will be loaded in the inverted order. Thus, the half modulo will be multiplied by 0.75 (1.0 – 0.25), that is, 468. Therefore, VAL5 = –468 (0xFE2C) and VAL4 = 468 (0x01D4).

The way how to set it up is shown below:

```
PWMA_SM3VAL2 = 0xFF64;
PWMA_SM3VAL3 = 0x009C;

PWMA_SM3VAL4 = 0x01D4;
PWMA_SM3VAL5 = 0xFE2C;
```

4.6 Zero deadtime

PFC does not require any deadtime so the DTCNT0 and DTCNT1 registers will be set to zero. The following code configures the deadtime to zero:

```
PWMA_SM3DTCNT0 = 0;
PWMA_SM3DTCNT1 = 0;
```

4.7 Disable faults

This example will not use the fault logic so it is necessary to disable the fault mapping registers in the following way:

```
PWMA_SM3DISMAP0 = 0;
PWMA_SM3DISMAP1 = 0;
```

4.8 LDOK bit

The final step before running PWM is to clear and set the LDOK bits in the following way:

```
PWMA_MCTRL |= PWMA_MCTRL_CLDOK_3;
PWMA_MCTRL |= PWMA_MCTRL_LDOK_3;
```

PWM A sub-module 3 is configured to generate signals as shown in Figure 3. But to get the signals out on the pins, the GPIO pins must be properly configured. So, certain GPIO E pins must be set up as peripheral. In case of multiple peripheral options, the PWM option must be chosen. The GPIO E clock must be enabled too. The code is shown below:

```
/* Enable GPIOE clock */
SIM_PCE0 |= SIM_PCE0_GPIOE;

/* PWM A sub-module 3*/
GPIOE_PER |= (GPIOE_PER_PE_6 | GPIOE_PER_PE_7);
SIM_GPSEL &= ~(SIM_GPSEL_E6 | SIM_GPSEL_E7);
```

Finally, the missing items are to send the run command to the module to start generation of the signal and to enable the PWM signals out. As the PWM signals are supposed to be synchronized with other peripherals, the PWM run command will not be applied at the moment.

5 PWM triggers for ADC

The PFC control requires analog signals that are used in the control algorithm to be measured. The required quantities are the PFC current and input (rectified) and output (DC bus) voltages.

Typically, the current is measured on a shunt resistor where the current flows back to the rectifier. So at this point the shunt resistor indicates the sum of currents flowing through the two branches of PFC inductors and MOSFETs and from the circuitry after the PFC. The points where the signals are measured must be synchronized with the PFC PWM module to avoid aliasing. This example measures the signals in the middle of the on-pulse of both MOSFETs.

Now putting it all into the context, we want to measure the following quantities:

- Input voltage after the rectifier—important to get the power lines phase and frequency
- PFC current—important to control the shape of the current
- Output voltage—DC bus voltage is important to control the output voltage

To have space for the algorithm, only four samples per the four PWM cycles will be taken—two samples per one PWM cycle. So the application will take two samples of currents and make its average value to filter it a little.

In the complete application with the motor control, the PFC output (alias DC bus voltage) is measured by the motor control application itself. So it is enough to take the value from the motor control portion of the code and then instead of the output voltage two samples of the input voltage can be taken with average filter application. In this example both the input voltage and the output voltage are taken into account. See Figure 4.

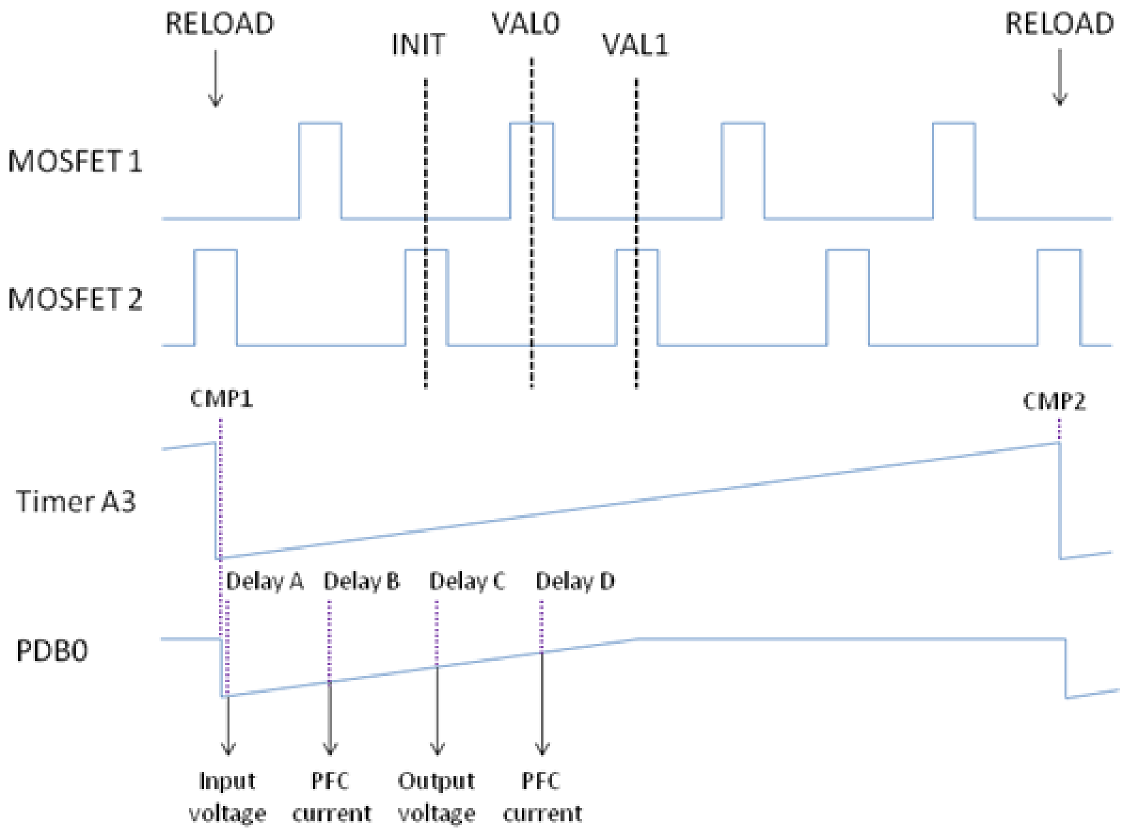

Figure 4. Points of sampling

PWM sub-module 3 has only the VAL0 register free to be used as programmable trigger. If the VAL0 register is used, it will generate the trigger once per the PWM duty cycle. In this case, we need to have two triggers per the PWM cycle.

A solution for this is to use the PDB module that will be synchronized from the PFC PWM reload. The PDB is capable to generate up to four programmable delays and the triggers for the ADC. But it must be somehow triggered by the PWM reload. The reload does not have its own trigger; only the VAL1 register could be used, which will generate the trigger each PWM cycle and force the PDB to reinitialize.

Therefore to generate only one trigger for the PDB, aligned with the PWM reload, a timer will be programmed to have a period of four PWM cycles and will be synchronized with the PWM module reload. The timer will generate a compare pulse of a very short time after the reload which will trigger the PDB module. And the PDB will generate four triggers for the ADC.

5.1 Timer initialization

The timer will run in two modes. The first mode is the mode where the timer start is triggered by the secondary source input pin. Once triggered, the timer will be switched to the simple count mode. To generate an edge to trigger the PDB module, the timer will use the alternating compare mode. So the output will be similar to the PWM where the timer output flag will be toggled using the two compare values. The on-pulse rising edge will generate the trigger for the PDB and then at the PWM reload event the timer must be restarted and the output flag must go low.

The prescaler will be set to the IP bus clock, the secondary input source will be the Counter 3 pin. The count length mode is "count until compare and then reinitialize".

So the compare values will be the following: the Compare 1 value is just a short delay between the PWM reload and the PDB trigger. This will be set to 500 ns, so if the input clock is 100 MHz the Compare 1 value will be 50 (0x32). The Compare 2 value is then the time from the Compare 1 value until the next PWM reload. So the value must be programmed as 4 PFC PWM periods which is 1250 x 4 = 5000. From this number the off-pulse defined by Compare 1 must be subtracted, that is, 4950. When the timer gets to the Compare 1 and Compare 2 values it always takes one tick to restart, so we have to subtract 2 from the value. Therefore, the result is 4948 (0x1354).

The timer clock must be enabled in the SIM module prior to the timer configuration. The timer's configuration code will look like:

```
/* Enable TMRA3 clock */
SIM_PCE0 |= SIM_PCE0_TA3;

/* Initialize the delay after PWM A3 reload */
TMRA_3_COMP1 = 0x0032;

/* Period of the timer is the PFC modulo * 4 - TMRA_3_COMP1 - 2 */
TMRA_3_COMP2 = 0x1354 - TMRA_3_COMP1 - 2;

/* Reset the counter */
TMRA_3_CNTR = 0;

/* Timer config:
* Toggle OFLAG output using alternating compare registers
* Count until compare, then reinitialize
* Secondary source: Counter 3 input pin
* Primary source: IP bus clock divide by 1 prescaler
* Edge of secondary source triggers primary count until compare
*/
TMRA_3_CTRL = TMRA_3_CTRL_OUTMODE_2 | TMRA_3_CTRL_LENGTH | TMRA_3_CTRL_SCS |
TMRA_3_CTRL_PCS_3 | TMRA_3_CTRL_CM_1 | TMRA_3_CTRL_CM_2;

/* Output flag enable */
TMRA_3_SCTRL = TMRA_3_SCTRL_OEN;
```

5.2 PWM triggers

To generate a trigger at the PWM reload the VAL1 trigger will be used. This trigger must be used only once; then the timer runs in synchronization with the PWM independently. So along with the VAL1 trigger, the compare 1 interrupt will be generated where the trigger and compare 1 will be disabled. This compare interrupt must be enabled in the interrupt controller. The initialization code will look like:

```
/* Trigger 1 enabled */
PWMA_SM3TCTRL |= PWMA_SM3TCTRL_OUT_TRIG_EN_1;

/* Compare 1 enabled */
PWMA_SM3INTEN |= PWMA_SM3INTEN_CMPIE_1;

/* Interrupt for the PWMA SM3 CMP Level 2*/
INTC_IPR9 |= INTC_IPR9_PWMA_CMP3;
```

5.3 VAL1 compare interrupt service routine

The PWM A sub-module 3 VAL1 compare interrupt has been configured and the interrupt service routine (ISR) must be created to enable the triggers.

The name of the ISR will be IsrPWMA3. A prototype has to be created for this routine in the prototype section of the code:

```
void IsrPWMA3(void);
```

The name of this function must be copied into the vector table; in case the default CodeWarrior 10.2 project template is used, it is in the file MC56F847xx_vector.asm (located in Project_Settings\Startup_Code). Thus at the address 0xA2, the interrupt no. 81 for the PWM A sub-module 3 will contain the following statement:

```
JSR  >FIsrPWMA3
```

The body of the function itself contains the following actions:

- switching the Timer A3 mode from the triggered mode to the simple count mode
- disabling the PWM A sub-module 3 VAL1 compare interrupt
- disabling the PWM A sub-module 3 compare interrupt in the interrupt controller
- clearing the PWM A sub-module 3 VAL1 compare flag

Putting all these into code will look like:

```
#pragma interrupt alignsp
void IsrPWMA3(void)
{
/* Switch TMRA3 to the simple count mode */
TMRA_3_CTRL = (TMRA_3_CTRL & ~TMRA_3_CTRL_CM) | TMRA_3_CTRL_CM_0;

/* Disables the PWM SM3 CMP interrupt from 1 */
PWMA_SM3INTEN &= ~PWMA_SM3INTEN_CMPIE_1;

/* Disables the PWM SM3 CMP interrupt */
INTC_IPR9 &= ~INTC_IPR9_PWMA_CMP3;

/* Clears PWM SM3 CMP flag from 1 */
PWMA_SM3STS |= PWMA_SM3STS_CMPF_1;
}
```

5.4 PDB initialization

PWM A sub-module 3 and Timer A3 have been coupled. The PDB must be properly set up. PDB 0 will be used to generate those four triggers. The clock for PDB 0 must be enabled in the SIM module prior to its configuration.

The configuration of the PDB is quite simple. An input clock has to be set up, which is the IP bus clock. The trigger used to start PDB will be its Trigger 0. The One Shot mode will be selected. The PDB must be enabled. Then the modulo of the counter must be set up. In our case it counts for two PWM cycles, so we can set it up as PWM modulo multiplied by 2, that is, 1250 x 2 = 2500 (0x09C4).

The trigger B must be enabled and its output is set up as a function of delay A and B. Similarly the trigger D must be enabled and its output is set up as a function of delay C and D.

The delays are set up very easily:

- Delay A—500 ns after the PDB start, that is, 1 μs after the PWM reload. The value is 50 (0x32)
- Delay B—is the time of Delay A plus the PWM half modulo, that is, 50 + 625 = 675 (0x2A3)
- Delay C—is the time of Delay B plus the PWM half modulo, that is, 675 + 625 = 1300 (0x514)
- Delay D—is the time of Delay C plus the PWM half modulo, that is, 1300 + 625 = 1925 (0x785)

Then the LDOK bit must be set to load the new values into the registers. The code to configure the PDB is:

```
/* Enable PDB clock */
SIM_PCE2 |= SIM_PCE2_PDB0;

/* Set peripheral clock, one-shot, hardware trigger 0, enable PDB */
PDB0_MCTRL = PDB0_MCTRL_PDBEN;

/* Set PDB0 Modulo to PWMA3 modulo * 2 */
PDB0_MOD = 0x09C4;

/* Function of A and B delay, trigger B enabled */
```

```
PDB0_CTRLA = PDB0_CTRLA_ABSEL | PDB0_CTRLA_ENB;

/* Function of C and D delay, trigger D enabled */
PDB0_CTRLC = PDB0_CTRLC_CDSEL | PDB0_CTRLC_END;

/* Initialize Delay A */
PDB0_DELAYA = 0x0032;

/* Initialize Delay B = Delay A + PFC PWM half modulo */
PDB0_DELAYB = 0x02A3;

/* Initialize Delay C = Delay B + PFC PWM half modulo */
PDB0_DELAYC = 0x0514;

/* Initialize Delay D = Delay C + PFC PWM half modulo */
PDB0_DELAYD = 0x0785;

/* Load new values to registers */
PDB0_MCTRL |= PDB0_MCTRL_LDOK;
```

These three peripherals have been properly configured yet they need to be inter-connected and the PWM module must be started when the ADC and DMA modules are ready.

6 ADC and DMA configuration

To sample the quantities 16-bit SAR ADC C will be used. The processor has also high-speed 12-bit ADC A and B; but these ADC modules are reserved for the motor control part. So to make the PFC independent the SAR ADC is used.

The ADC has only one result register so it is only capable to take one sample and then the result must be fetched and the ADC, reconfigured for another sample. Each sample reading and ADC reconfiguration is done in the ADC interrupt. Frequent interrupts would be time consuming. Therefore, two DMA channels are used. The first one will save the result from the ADC to the buffer at the end of the conversion. The second DMA channel will copy the new channel id into the ADC register right after the first channel’s transfer. See Figure 5.

The DMA channels will be programmed to transfer four words and then an interrupt will be generated. In this interrupt, the DMA channels must be reinitialized for the next four transfers and the control algorithm is called here. The benefit of this is that only one single interrupt is called within the fast control loop period.

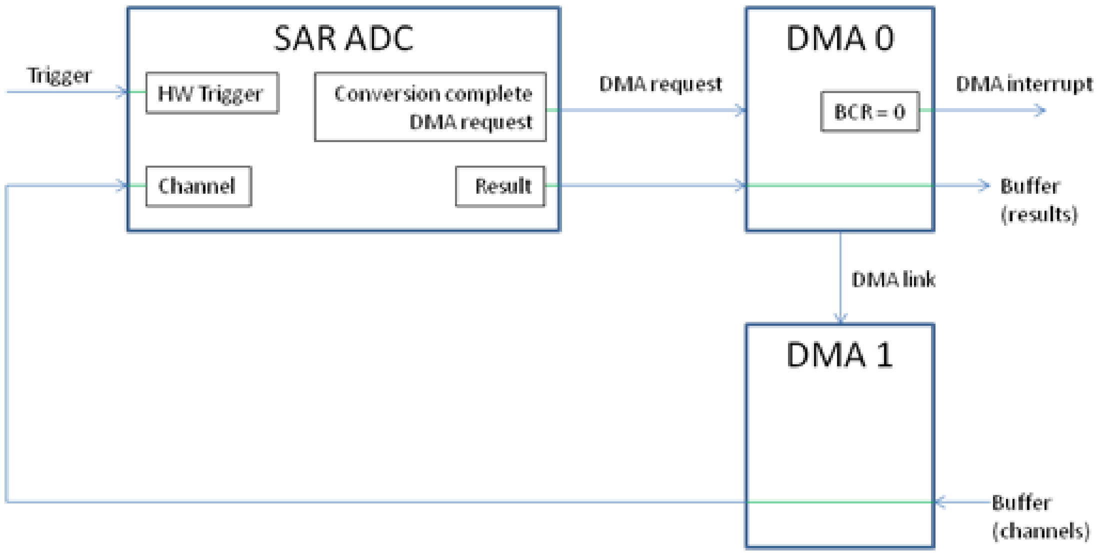

Figure 5. ADC and DMA usage

6.1 User buffer initialization

As mentioned above, two DMA channels will be used. One will transfer from Data Result Register into the buffer, and the other from the buffer to Status and Configuration Register 1. So the results' buffer will be a 4-word array. The channels' buffer will be a 4-word array that will be initialized with the channels' id's. One thing must be taken into consideration from the point of view of channels' order. The first channel must be initialized manually, therefore after the first ADC result transfer, the DMA transfers the second channel id in line from the first position in the buffer. So the channels' buffer will begin with the second channel and end with the first channel. The buffers can be made as static variables by the way shown below:

```
static UWord16 muw16ADCResult[4];
static UWord16 muw16ADCChannel[4] = {8, 12, 8, 15};
```

6.2 ADC configuration

To be able to use the SAR ADC, it is necessary to habilitate its clock in Peripheral Clock Register 2 (SIM_PCE2) from System Integration Module (SIM).

Then in Configuration Register 1 the normal power consumption will be used, short sample time, 16-bit conversion, and the input clock will be the IP bus divided by 2 and the clock divider will be set to 4, so the ADC clock will result in 12.5 MHz.

In Status and Control Register, the hardware trigger will be allowed and the DMA will be enabled.

The last step in the ADC configuration is to initialize the channel that will be converted on the first trigger. It will be the input voltage.

The ADC configuration will look like:

```
/* Enable SAR ADC clock */
SIM_PCE2 |= SIM_PCE2_SARADC;

/* normal power, short time, 12.5 MHz clock, 16-bit */
ADC16_CFG1 = ADC16_CFG1_ADIV_1 | ADC16_CFG1_MODE | ADC16_CFG1_ADICLK_0;

/* HW trigger, DMA */
ADC16_SC2 = ADC16_SC2_ADTRG | ADC16_SC2_DMAEN;

/* First channel initialization, that is, the last in the buffer */
ADC16_SC1A = muw16ADCChannel[3];
```

6.3 AN pins configuration

To read analog values from the pins, the particular GPIO A and B pins must be configured as peripherals with the ADC option. The clock for the GPIO modules must be enabled. The code is shown below:

```
/* Enable GPIOA clock */
SIM_PCE0 |= SIM_PCE0_GPIOA;

/* ADC channel 8 */
GPIOA_PER |= GPIOA_PER_PE_4;

/* Enable GPIOB clock */
SIM_PCE0 |= SIM_PCE0_GPIOB;

/* ADC channels 12 and 15 */
GPIOB_PER |= (GPIOB_PER_PE_4 | GPIOB_PER_PE_7);
```

6.4 DMA configuration

The task of the DMA is to transfer the ADC results to the buffer and set up the channels. Therefore two DMA channels will be used. Let us start with DMA channel 0 that will transfer the results.

First of all, the ADC C peripheral request must be assigned at the DMA channel. Before assigning it, the state machine control for the channel has to be cleared and then assign the peripheral request 14 (ADC C conversion complete). This register is 32-bit and is shared for all the 4 channels. The channel 0 configuration occupies the upper 8 bits, so the number 14 is shifted 24 times to the left.

The source address for the DMA will be the ADC result register. The ADC result register is 32-bit but the upper 16 bits are not used; thus it is enough to transfer the lower 16-bits. The DMA peripheral addressing mode is in bytes while the DSC uses words. The address of the register in bytes is two times the address in words.

The destination address for the DMA will be the results' buffer. Here the address must be converted to bytes.

The byte count register must be properly set up. As we transfer 4 words, the byte counter will be set up to 8 bytes. But before that the possible DMA flags must be cleared to be able to update the register.

The last register that must be configured is DMA Control Register. This consists of the interrupt and peripheral request enablement, cycle steal option (which means only one word is transferred per the peripheral request), the destination address incrementation option, and source and destination size word option. The link option will be chosen to generate a link to another channel after each cycle steal and the linked channel will be Channel 1.

As the interrupt is enabled, the interrupt controller must be configured too.

Finally the configuration for DMA Channel 1 will look like:

```
 /* Request 14 is SAR ADC conversion end */
DMA_REQC |= DMA_REQC_CFSM0;
DMA_REQC = (DMA_REQC & ~DMA_REQC_DMAC0) | ((UWord32)14 << 24);

/* Source address is the SAR ADC data result register. */
DMA_SAR0 = ((uint32_t)FADC16_RA << 1);

/* Destination address is the result's buffer */
DMA_DAR0 = ((uint32_t)guw16ADCResult << 1);

/* Clears the DMA Ch. 0 flags */
DMA_DSR_BCR0 |= DMA_DSR_BCR0_DONE;

/* 4 words data will be transferred */
DMA_DSR_BCR0 = (DMA_DSR_BCR0 & ~DMA_DSR_BCR0_BCR) | 8;

/* Enabled interrupt, enable periph. request, cycle steal, destination increment,  source
size word, destination size word, link after cycle steal, link to ch. 1 */
DMA_DCR0 = DMA_DCR0_EINT | DMA_DCR0_ERQ | DMA_DCR0_CS | DMA_DCR0_DINC | DMA_DCR0_SSIZE_1 |
DMA_DCR0_DSIZE_1 | DMA_DCR0_LINKCC_1 | DMA_DCR0_LCH1_0;

/* Set DMA CH0 interrupt priority as level 2 */
INTC_IPR3 |= INTC_IPR3_DMACH0;
```

The second channel (Channel 1) will be triggered by the Channel 0 link option. It is not necessary to assign any peripheral request to it.

The source address for the DMA will be the channels' buffer. The ADC result register is 32-bit but the upper 16 bits are not used; thus, it is enough to transfer the lower 16 bits. The DMA peripheral addressing mode is in bytes while the DSC uses words. The address of the register in bytes is two times the address in words.

The destination address for the DMA will be ADC Status and Configuration Register 1. Here the address must be converted to bytes.

The byte count register must be properly set up. As we transfer 4 words, the byte counter will be set up to 8 bytes. But before that the possible DMA flags must be cleared to be able to update the register.

The last register that must be configured is DMA Control Register. Here the cycle steal option is set, source address incremented, and source and destination size as word.

The configuration code looks like as shown below:

```
/* Source address is the channels buffer */
DMA_SAR1 = ((uint32_t)guw16ADCChannels << 1);

/* Destination address is the SAR ADC SC1A register. */
DMA_DAR1 = ((uint32_t)FADC16_SC1A << 1);

/* Clears the DMA Ch. 1 flags */
DMA_DSR_BCR1 |= DMA_DSR_BCR1_DONE;

/* 4 words will be transferred */
DMA_DSR_BCR1 = (DMA_DSR_BCR1 & ~DMA_DSR_BCR1_BCR) | 8;

/* Cycle steal, source increment, source size word, dest. size word */
DMA_DCR1 = DMA_DCR1_CS | DMA_DCR1_SINC | DMA_DCR0_SSIZE_1 | DMA_DCR0_DSIZE_1;
```

6.5 DMA interupt service routine

The DMA interrupt has been configured and the interrupt service routine (ISR) must be created to handle the interrupt.

The name of the ISR will be IsrDMA0. A prototype has to be created for this routine in the prototype section of the code:

```
void IsrDMA0(void);
```

The name of this function must be copied into the vector table; in case the default CodeWarrior 10.2 project template is used, it is in the file MC56F847xx_vector.asm (located in Project_Settings\Startup_Code). Thus, at the address of 0x48 the interrupt no. 36 for DMA Channel 0 will contain the following statement:

```
JSR  >FIsrDMA0
```

The body of the function itself contains the following actions:

- clearing the DMA Channel 0 done flag
- reinitializing the DMA channels

So putting all this into the code will look like:

```
#pragma interrupt saveall
void IsrDMA0(void)
{
/* Destination address is the results buffer */
DMA_DAR0 = ((uint32_t)guw16ADCResult << 1);

/* Source address is the channels buffer */
DMA_SAR1 = ((uint32_t)guw16ADCChannels << 1);

/* Clears the DMA Ch. 0 flags */
DMA_DSR_BCR0 |= DMA_DSR_BCR0_DONE;

/* 4 words data will be transferred */
DMA_DSR_BCR0 = (DMA_DSR_BCR0 & ~DMA_DSR_BCR0_BCR) | 8;

/* Clears the DMA Ch. 1 flags */
DMA_DSR_BCR1 |= DMA_DSR_BCR1_DONE;

/* 4 words will be transferred */
DMA_DSR_BCR1 = (DMA_DSR_BCR1 & ~DMA_DSR_BCR1_BCR) | 8;
}
```

7 PWM, Timer, PDB, and ADC signals interconnection

To sum up what has been done so far:

- The PWM module has been configured
- The timer has been configured
- The PDB has been configured
- The ADC has been configured
- The DMA channels have been configured

The next step to make it work is to properly connect the triggers from the PWM A to Timer A3 to synchronize the timer along PWM A sub-module 3. Then the Timer A3 output flag must be connected to the PDB0 input trigger and the PDB0 B and D triggers must be connected to the SAR ADC hardware trigger input. To do this, Inter-Peripheral Cross-Bar Switch A (XBAR A) is used.

XBAR A is capable to connect one output of a peripheral to another input; but in this system we have also two trigger signals from PDB 0 to be connected to one ADC input. Therefore in this case, the logical OR of the triggers has to be made and the OR result signal will be led to the ADC input. We could use the And/Or/Invert (AOI) module to OR the signals but it is not necessary because these two signals are already ORed on the Inter-Peripheral Cross-Bar Switch B (XBAR B) input. XBAR B only connects these signals to the AOI module that will be programmed to pass the input to the output. See Figure 6.

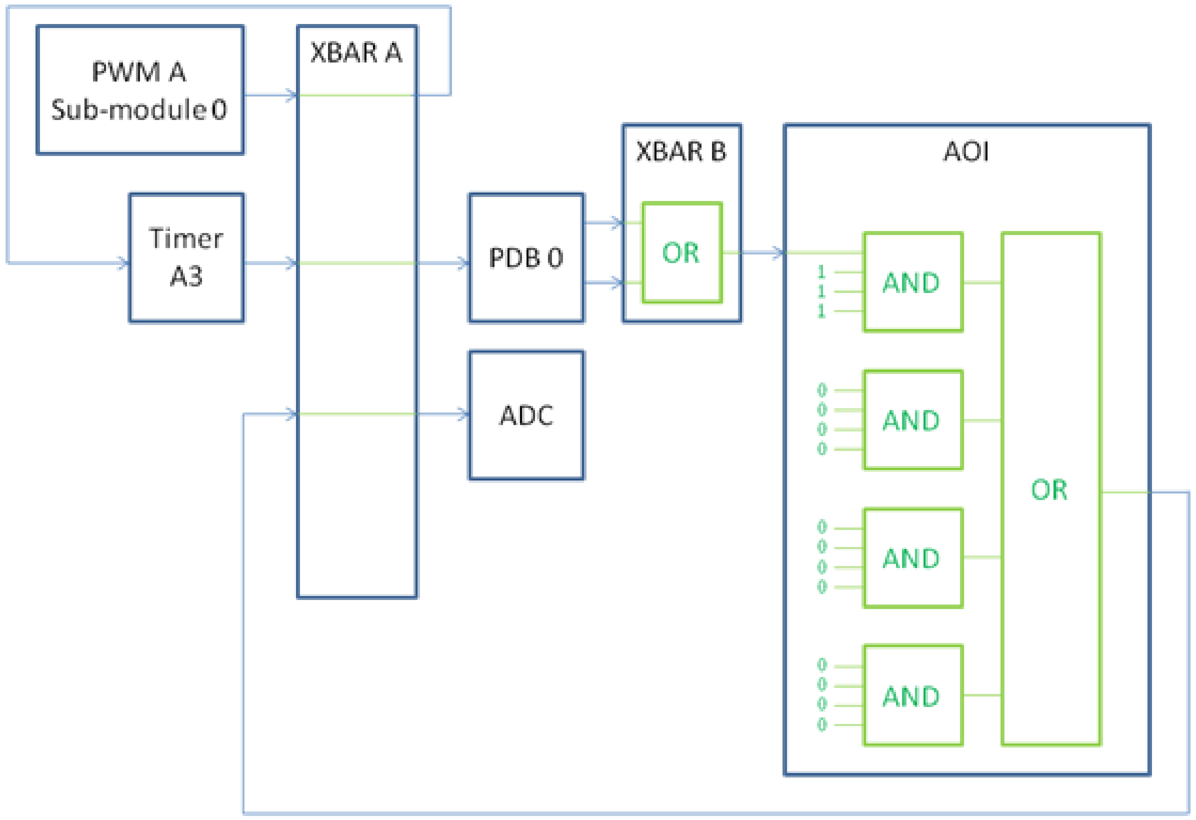

Figure 6. PWM and ADC inter-connection

7.1 XBAR B configuration

XBAR B provides the connections of the peripheral outputs to the AOI module. XBAR B is used to connect the PDB 0 triggers to AOI. So the XBAR_IN12 input will be assigned to the XBAR B input 4. The code will have this syntax:

```
XBARB_SEL2 = 12;
```

7.2 AOI configuration

As the name of this module says, this module is capable of logical AND/OR/INVERT operation. But as XBAR B already makes the logical OR of the signals we will only program it to pass the signal to XBAR A.

The configuration codeline is the following:

```
AOI_BFCRT011 = AOI_BFCRT011_PT0_AC_0 | AOI_BFCRT011_PT0_BC | AOI_BFCRT011_PT0_CC |
AOI_BFCRT011_PT0_DC;
AOI_BFCRT231 = 0;
```

7.3 XBAR A configuration

XBAR A interconnects many signals in this application which are:

- PWM A sub-module 0 trigger 1 (XBAR_IN27) to the Timer A3 input (XBAR_OUT52)
- The Timer A3 output flag (XBAR_IN29) to PDB 0 input trigger 0 (XBAR_OUT38)
- AOI 1 (XBAR_IN47) to the ADC C hardware trigger (XBAR_OUT14)

So the code will look like:

```
/* TMRA3 input from PWMA3 TRG1 */
XBARA_SEL26 = 27;

/* TMRA3 output to PDB0 input trigger 0 */
XBARA_SEL19 = 39;

/* ADC C trigger from AOI */
XBARA_SEL7 = 47;
```

One point that cannot be forgotten is to configure the Timer A3 input option from XBAR A. This input is configured to the pin option by default. This option is cofigured via Internal Peripheral Select Register 0 (SIM_IPS0). So the code has this syntax:

```
/* Input to TMRA3 from XBAR */
SIM_IPS0 |= SIM_IPS0_TA3;
```

8 PWM start and reading ADC samples

At this point all the peripherals are properly configured to work in synchronization. The last two points that must be accomplished is to send the RUN command to the PWM module to start the complete machine and define where to read the sampled signals and calculate the control algorithm of the PFC. This can be done manually or if the motor control application requires the PFC and motor PWMs to be synchronized then it must be started somewhere in the motor PWM interrupt. In this case we will only call the command manually which is:

```
/* Starts PWM A 3 */
PWMA_MCTRL |= PWMA_MCTRL_RUN_3;
```

The samples will be read in the DMA interrupt. The results are stored in the muw16ADCResult array in the sequence: input voltage, PFC current, output voltage, PFC current. They are stored as unsigned 16-bit values; so they must be converted into the signed fractional values. To read them out from the buffer and convert them into particular variables the following syntax can be used:

```
Frac16 f16Vin, f16Vout, f16Ipfc;

f16Vin = (Frac16)(muw16ADCResult[0] >> 1);
f16Vout = (Frac16)(muw16ADCResult[2] >> 1);
f16Ipfc = extract_h((Frac32)((UWord32)muw16ADCResult[1] + (UWord32)muw16ADCResult[3]) << 14);
```

These sampled signals are passed to the control algorithm for the current control and output voltage control. The synchronization can be seen in Figure 7.

The last action that must be done is to get the PWM signals out on the pins. Take care that the PFC MOSFETs are not connected or the application, under voltage. The duty cycle is initialized to 25% just to be able to observe theme on the scope. If the signals are connected to the MOSFETs that are under voltage, it will quickly charge the DC bus capacitors to their limit and beyond where the capacitors can explode! Nevertheless the command to get the PWM signals out to the pins is the following:

```
PWMA_OUTEN |= (PWMA_OUTEN_PWMA_EN_3 | PWMA_OUTEN_PWMB_EN_3);
```

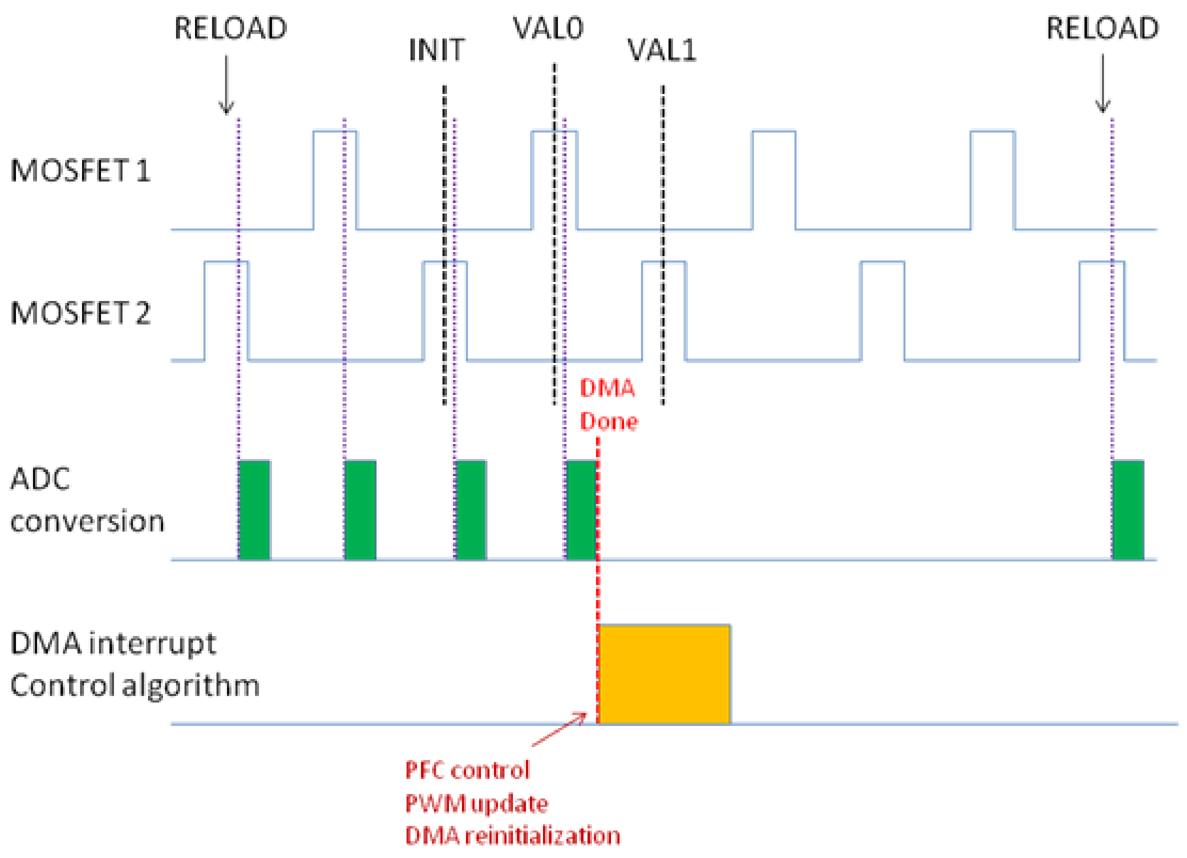

Figure 7. Reading ADC and control algorithms calculation

Now all necessary points have been accomplished to synchronize the PWM and ADC for the PFC control.

9 Complete code

The PWM module, timer, PDB, ADC, and DMA have been configured and synchronized, the triggers and interrupts programmed. The modules have been interconnected by the cross-bar switches. Now let us put all the code into the context from the top to the bottom to see what we have done so far.

9.1 Interrupt vector table

File MC56F847xx_vector.asm (located in Project_Settings\Startup_Code)

```
JSR  >FIsrPWMA3; /* 0xa2 Interrupt no. 81 */
JSR  >FIsrDMA0;  /* 0x48 Interrupt no. 36 */
```

9.2 Prototypes

```
static void GPIOA_Init(void);
static void GPIOB_Init(void);
static void GPIOE_Init(void);
static void XBAR_Init(void);
static void PWMA_SM3_Init(void);
static void PWMA_SM3_Run(void);
static void PWMA_SM3_Enable(void);
static void DMA_Init(void);
static void ADC16_Init(void);
static void PDB_Init(void);
static void TMRA3_Init(void);
void IsrPWMA3(void);
void IsrDMA0 (void);
```

9.3 Static variables

```
static UWord16 muw16ADCResult[4];
static UWord16 muw16ADCChannel[4] = {8, 12, 8, 15};
```

9.4 Functions

```
static void GPIOA_Init(void)
{
    /* Enable GPIOA clock */
    SIM_PCE0 |= SIM_PCE0_GPIOA;

    /* ADC channel 8 */
    GPIOA_PER |= GPIOA_PER_PE_4;
}

static void GPIOB_Init(void)
{
    /* Enable GPIOB clock */
    SIM_PCE0 |= SIM_PCE0_GPIOB;

    /* ADC channels 12 and 15 */
    GPIOB_PER |= (GPIOB_PER_PE_4 | GPIOB_PER_PE_7);
}

static void GPIOE_Init(void)
{
    /* Enable GPIOE clock */
    SIM_PCE0 |= SIM_PCE0_GPIOE;

    /* PWM A sub-module 3*/
    GPIOE_PER |= (GPIOE_PER_PE_6 | GPIOE_PER_PE_7);
    SIM_GPSEL &= ~(SIM_GPSEL_E6 | SIM_GPSEL_E7);
}

static void XBAR_Init(void)
{
     /* TMRA3 input from PWMA3 TRG1 */
     XBARA_SEL26 = 27;

     /* TMRA3 output to PDB0 input trigger 0 */
     XBARA_SEL19 = 39;

     /* PDB TRG B | D to XBAR B */
     XBARB_SEL2 = 12;

     /* AOI out = A, that is, TRG B | D, passed via XBAR B */
     AOI_BFCRT011 = AOI_BFCRT011_PT0_AC_0 | AOI_BFCRT011_PT0_BC
     | AOI_BFCRT011_PT0_CC | AOI_BFCRT011_PT0_DC;

     AOI_BFCRT231 = 0;

     /* ADC C trigger from AOI */
     XBARA_SEL7 = 47;
}

static void PWMA_SM3_Init(void)
{
     /* Enable PWM A SM3 Module Clock */
     SIM_PCE3 |= SIM_PCE3_PWMACH3;

     /* Reload every 4 opportunity */
     PWMA_SM3CTRL = (PWMA_SM3CTRL_LDFQ_0 | PWMA_SM3CTRL_LDFQ_1 | PWMA_SM3CTRL_FULL);

     /* Independent mode */
     PWMA_SM3CTRL2 = PWMA_SM3CTRL2_INDEP;

     /* setup for pwm frequency of 80 KHz */
     PWMA_SM3INIT = 0xFD8F;
     PWMA_SM3VAL1 = 0x0270;

     PWMA_SM3VAL0 = 0x0000;

     /* 25% duty cycle */
     PWMA_SM3VAL2 = 0xFF64;
     PWMA_SM3VAL3 = 0x009C;

     PWMA_SM3VAL4 = 0x01D4;
     PWMA_SM3VAL5 = 0xFE2C;

     /* deadtime count register to zero */
     PWMA_SM3DTCNT0 = 0;
     PWMA_SM3DTCNT1 = 0;

     /* Fault A 0-3 inactive */
     PWMA_SM3DISMAP0 = 0x0000;

     /* Fault A 4-7 inactive */
     PWMA_SM3DISMAP1 = 0x0000;

     /* Trigger 1 enabled */
     PWMA_SM3TCTRL |= PWMA_SM3TCTRL_OUT_TRIG_EN_1;

     /* Compare 1 enabled */
     PWMA_SM3INTEN |= PWMA_SM3INTEN_CMPIE_1;

     /* Interrupt for the SM3 CMP Level 2*/
     INTC_IPR9 |= INTC_IPR9_PWMA_CMP3;

     /* Clear LDOK bit */
     PWMA_MCTRL |= PWMA_MCTRL_CLDOK_3;
     /* LDOK */
     PWMA_MCTRL |= PWMA_MCTRL_LDOK_3;
}

static void PWMA_SM3_Run(void)
{
     /* Starts PWM A 3 */
     PWMA_MCTRL |= PWMA_MCTRL_RUN_3;
}

static void PWMA_SM3_Enable(void)
{
      PWMA_OUTEN |= (PWMA_OUTEN_PWMA_EN_3 | PWMA_OUTEN_PWMB_EN_3);
}

static void DMA_Init(void)
{
      /* Request 14 is SAR ADC conversion end */
      DMA_REQC |= DMA_REQC_CFSM0;
      DMA_REQC = (DMA_REQC & ~DMA_REQC_DMAC0) | ((UWord32)14 << 24);

      /* Source address is the SAR ADC data result register. */
      DMA_SAR0 = ((uint32_t)FADC16_RA << 1);

      /* Destination address is the results buffer */
      DMA_DAR0 = ((uint32_t)guw16ADCResult << 1);

      /* Clears the DMA Ch. 0 flags */
      DMA_DSR_BCR0 |= DMA_DSR_BCR0_DONE;

      /* 4 words data will be transfered */
      DMA_DSR_BCR0 = (DMA_DSR_BCR0 & ~DMA_DSR_BCR0_BCR) | 8;

      /* Enabled interupt, enable periph. request, cycle steal, destination
increment,  sourse size word, dest. size word, link after cycle steal,
      link to ch. 1 */
      DMA_DCR0 = DMA_DCR0_EINT | DMA_DCR0_ERQ | DMA_DCR0_CS | DMA_DCR0_DINC |
DMA_DCR0_SSIZE_1 | DMA_DCR0_DSIZE_1 | DMA_DCR0_LINKCC_1 | DMA_DCR0_LCH1_0;


      /* Source address is the channels buffer */
      DMA_SAR1 = ((uint32_t)guw16ADCChannels << 1);

      /* Destination address is the SAR ADC SC1A register. */
      DMA_DAR1 = ((uint32_t)FADC16_SC1A << 1);

      /* Clears the DMA Ch. 1 flags */
      DMA_DSR_BCR1 |= DMA_DSR_BCR1_DONE;

      /* 4 words will be transfered */
      DMA_DSR_BCR1 = (DMA_DSR_BCR1 & ~DMA_DSR_BCR1_BCR) | 8;

      /* Cycle steal, source increment, source size word, dest. size word */
      DMA_DCR1 = DMA_DCR1_CS | DMA_DCR1_SINC | DMA_DCR0_SSIZE_1 | DMA_DCR0_DSIZE_1;

      /* Set DMA CH0 interrupt priority as level 2 */
      INTC_IPR3 |= INTC_IPR3_DMACH0;
}

static void ADC16_Init(void)
{
      /* Enable SAR ADC clock */
      SIM_PCE2 |= SIM_PCE2_SARADC;

      /* normal power, short time, 12.5MHz clock, 16-bit */
      ADC16_CFG1 = ADC16_CFG1_ADIV_1 | ADC16_CFG1_MODE | ADC16_CFG1_ADICLK_0;

      /* HW trigger, DMA */
      ADC16_SC2 = ADC16_SC2_ADTRG | ADC16_SC2_DMAEN;
      /* First channel initialization, that is, the last in the buffer */
      ADC16_SC1A = muw16ADCChannel[3];
}



static void PDB_Init(void)
{
     /* Enable PDB clock */
     SIM_PCE2 |= SIM_PCE2_PDB0;

     /* Set peripheral clock, one-shot, hardware trigger 0, enable PDB */
     PDB0_MCTRL = PDB0_MCTRL_PDBEN;

     /* Set PDB0 Modulo to PWMA3 modulo * 2 */
     PDB0_MOD = 0x09C4;

     /* Function of A and B delay, trigger B enabled */
     PDB0_CTRLA = PDB0_CTRLA_ABSEL | PDB0_CTRLA_ENB;

     /* Function of C and D delay, trigger D enabled */
     PDB0_CTRLC = PDB0_CTRLC_CDSEL | PDB0_CTRLC_END;

     /* Initialize Delay A */
     PDB0_DELAYA = 0x0032;

     /* Initialize Delay B = Delay A + PFC PWM half modulo */
     PDB0_DELAYB = 0x02A3;

     /* Initialize Delay C = Delay B + PFC PWM half modulo */
     PDB0_DELAYC = 0x0514;

     /* Initialize Delay D = Delay C + PFC PWM half modulo */
     PDB0_DELAYD = 0x0785;

     /* Load new values to registers */
     PDB0_MCTRL |= PDB0_MCTRL_LDOK;
}

static void TMRA3_Init(void)
{
     /* Enable TMRA3 clock */
     SIM_PCE0 |= SIM_PCE0_TA3;

     /* Initialize the delay after PWM A3 reload */
     TMRA_3_COMP1 = 0x0032;

     /* Period of the timer is the PFC modulo * 4 - TMRA_3_COMP1 - 2 */
     TMRA_3_COMP2 = 0x1354 - TMRA_3_COMP1 - 2;

     /* Reset the counter */
     `TMRA_3_CNTR = 0;

     /* Timer config:
     * Toggle OFLAG output using alternating compare registers
     * Count until compare, then re-initialize
     * Socondary source: Counter 3 input pin
     * Primary source: IP bus clock divide by 1 prescaler
     * Edge of secondary source triggers primary count until compare
     */
     TMRA_3_CTRL = TMRA_3_CTRL_OUTMODE_2 | TMRA_3_CTRL_LENGTH | TMRA_3_CTRL_SCS |
TMRA_3_CTRL_PCS_3 | TMRA_3_CTRL_CM_1 | TMRA_3_CTRL_CM_2;


     /* Output flag enable */
     TMRA_3_SCTRL = TMRA_3_SCTRL_OEN;
}

#pragma interrupt alignsp
```

```
void IsrPWMA3(void)
{
     /* Switch TMRA3 to the simple count mode */
     TMRA_3_CTRL = (TMRA_3_CTRL & ~TMRA_3_CTRL_CM) | TMRA_3_CTRL_CM_0;

     /* Disables the PWM SM3 CMP interrupt from 1 */
     PWMA_SM3INTEN &= ~PWMA_SM3INTEN_CMPIE_1;

     /* Disables the PWM SM3 CMP interrupt */
     INTC_IPR9 &= ~INTC_IPR9_PWMA_CMP3;

     /* Clears PWM SM3 CMP flag from 1 */
     PWMA_SM3STS |= PWMA_SM3STS_CMPF_1;
}

#pragma interrupt saveall
void IsrDMA0(void)
{
    Frac16 f16Vin, f16Vout, f16Ipfc;

    /* Input voltage */
    f16Vin = (Frac16)(muw16ADCResult[0] >> 1);

    /* Output voltage */
    f16Vout = (Frac16)(muw16ADCResult[2] >> 1);

    /* PFC current */
    f16Ipfc = extract_h((Frac32)((UWord32)muw16ADCResult[1] +
(UWord32)muw16ADCResult[3]) << 14);


    ... control algoritm ...


    /* Destination address is the results buffer */
    DMA_DAR0 = ((uint32_t)guw16ADCResult << 1);

    /* Source address is the channels buffer */
    DMA_SAR1 = ((uint32_t)guw16ADCChannels << 1);

    /* Clears the DMA Ch. 0 flags */
    DMA_DSR_BCR0 |= DMA_DSR_BCR0_DONE;

    /* 4 words data will be transfered */
    DMA_DSR_BCR0 = (DMA_DSR_BCR0 & ~DMA_DSR_BCR0_BCR) | 8;

    /* Clears the DMA Ch. 1 flags */
    DMA_DSR_BCR1 |= DMA_DSR_BCR1_DONE;

    /* 4 words will be transfered */
    DMA_DSR_BCR1 = (DMA_DSR_BCR1 & ~DMA_DSR_BCR1_BCR) | 8;
}
```

9.5 Functions call order

The initialization of the application requires the functions to be called in the following order:

```
GPIOA_Init();
GPIOB_Init();
GPIOE_Init();
XBAR_Init();
PWMA_SM3_Init();
DMA_Init();
ADC16_Init();
```

```
PDB_Init();
TMRA3_Init();
PWMA_SM3_Run();
```

To enable PWM to the pins use the following command but double check that the MOSFETs cannot charge the capacitors:

```
PWMA_SM3_Enable();
```

10 Definitions and acronyms

Table 1. Definitions and acronyms used in this application note

GPIO	General Port Input Output
ADC	Analog-to-Digital Converter
XBAR	Cross-Bar Switch
PWM	Pulse-Width Modulation
TMR	Timer
PDB	Programmable Delay Block
DMA	Direct Memory Access
ISR	Interrupt Service Routine
AOI	And/Or/Invert Module
SIM	System Integration Module
DSC	Digital Signal Controller
PFC	Power Factor Correction
Motor control	In this application note, a process that controls an electrical motor such as BLDC PMSM, AC-induction, or other

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

Document Number: AN4583
Rev. 0, 09/2012