Analog Devices, Inc.

System Services
Dual **PWM Service**

Pulse Width Modulation Unit

For Blackfin ADSP-BF50x family.

Introduction

This document is included in a "special release" of the system services libraries, for the ADSP-BF50x family of Blackfin processors, which feature two identical, programmable, pulse width modulation (PWM) modules. The following information supplements the PWM Chapter (13) of the VisualDSP++ 5.0 Device Drivers and System Services Manual for Blackfin processors. This information is to be used until the next revision of the VisualDSP++ Help System, at which time the information provided in this document will be rolled into the Device Drivers and System Services Manual, included in the VisualDSP help system.

No changes have been made to the existing API functions or command sets, for the ADSP-BF50x dual PWM service. There are some additional structures and enumerations passed with the commands, to specify the PWM number.

"Dual PWM" Service Initialization

The application initializes each PWM separately, by passing commands to the function adi_pwm_Init(). The command set is described in the API reference section of chapter 13 of the VisualDSP++ 5.0 Device Drivers and System Services Manual. Alterations have been made only to the parameters which are passed with these commands, to specify which PWM the command is intended

for. For example, enumeration values which previously specified either enable or disable status, have been extended to specify enable or disable, for either PWM 0 or PWM 1.

If only one PWM is used, no commands should be passed for the other, unused PWM. It will be left disabled.

PWM Service Events

The PWM service supports two external asynchronous "events" for each PWM: trip and sync.

There are four events for ADDSP-BF50x, , two trip events and two sync events, identified by an 'Event ID' defined in the include file 'adi_pwm.h'. These four events can each be configured to generate an interrupt to the core, by calling the adi_pwm_InstallCallback() function, the same as with the single PWM service.

```
ADI_PWM0_EVENT_TRIP,
ADI_PWM0_EVENT_SYNC,
ADI_PWM1_EVENT_TRIP,
ADI_PWM1_EVENT_SYNC,
```

The original PWM service for the ADSP-BF51x had just one of each type of event, so there are just two event IDs, each identified by a unique associated constant value.

```
ADI_PWM_EVENT_TRIP,
ADI_PWM_EVENT_SYNC,
```

PWM Interrupt Handlers

Within the PWM Service are two Sync interrupt handlers and two PWM Trip interrupt handlers.

PWM Number

A new enumeration called ADI_PWM_NUMBER is placed within some of the other data structures, to select which of the two PWMs an action is intended for. The value of ADI_NUM_PWM will specify how many PWMs there are. In this case, there are two.

ADI_PWM_NUMBER

Name	Numeric Value	Description
ADI_PWM_0	0	Specifies PWM 0
ADI_PWM_0	1	Specifies PWM 1
ADI_NUM_PWM	2	Placed as last enum, to count number of PWMs

Duty Cycle Structure

When the ADI_PWM_SET_DUTY_CYCLE command is passed to adi_pwm_Init() or adi_pwm_Control(), to set the duty cycles for any of the three channels, the channel and duty cycle are specified in the fields of the accompanying data structure. This data structure now also has a PWM number field, to specify which PWM number the duty cycle is for.

ADI_PWM_CHANNEL_DUTY_CYCLE

Type	Name	Description
ADI_PWM_NUMBER	PwmNumber	Specifies PWM that Value is for
ADI_PWM_CHANNEL	Channel	Specifies Channel or channel pair that Value is for
u32	Value	Specifies Value of duty cycle

Example)

Instead of just specifying the channel and value, as before (shown below)

```
ADI_PWM_CHANNEL_DUTY_CYCLE  pwm_dutyA_struct = { ADI_PWM_CHANNEL_A, pPhaseData->A };
ADI_PWM_CHANNEL_DUTY_CYCLE  pwm_dutyB_struct = { ADI_PWM_CHANNEL_B, pPhaseData->B };
ADI_PWM_CHANNEL_DUTY_CYCLE  pwm_dutyC_struct = { ADI_PWM_CHANNEL_C, pPhaseData->C };
```

Specify also the PWM number, when populating the data structure.

```
ADI_PWM_CHANNEL_DUTY_CYCLE  pwm0_dutyA_struct = {  ADI_PWM_0,   ADI_PWM_CHANNEL_A, pPhaseData->A };
ADI_PWM_CHANNEL_DUTY_CYCLE  pwm0_dutyB_struct = {  ADI_PWM_0,   ADI_PWM_CHANNEL_B, pPhaseData->B };
ADI_PWM_CHANNEL_DUTY_CYCLE  pwm0_dutyC_struct = {  ADI_PWM_0,   ADI_PWM_CHANNEL_C, pPhaseData->C };
```

Then pass the data structure in a command pair, same as before.

```
ADI_PWM_COMMAND_PAIR  PWM_InitTable [] =
{
   {  ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm0_dutyA_struct  },
   {  ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm0_dutyB_struct  },
   {  ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm0_dutyC_struct  },
   {  ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm1_dutyA_struct  },
   {  ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm1_dutyB_struct  },
   {  ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm1_dutyC_struct  }
Etc.,
```

Port MUX Mapping

The PWM pins on the ADSP-BF50x EZ-Kit are multiplexed. Just like the original PWM service, the Ports manager should be configured, prior to initializing the PWM service. A ports manager configuration table is passed to the adi_ports_Configure() function. The difference is that the table for the ADSP-BF50x has 16 entries (for two PWMs) whereas the table for the ADSP-BF51x had only 8 entries.

```
typedef enum ADI_PWM_PORT_MUX
{
   ADI_PWM_MUX_PRI,      /* primary port mapping */
   ADI_PWM_MUX_SEC,      /* secondary port mapping */
} ADI_PWM_PORT_MUX;
```

The port map structure has an entry for each signal.

```
typedef struct ADI_PWM_PORT_MAP
{
   u32 AH_0_MUX:1;
  u32 AL_0_MUX:1;
  u32 BH_0_MUX:1;
  u32 BL_0_MUX:1;
  u32 CH_0_MUX:1;
  u32 CL_0_MUX:1;
  u32 SYNC_0_MUX:1;
  u32 TRIP_0_MUX:1;
  u32 AH_1_MUX:1;
  u32 AL_1_MUX:1;
  u32 BH_1_MUX:1;
  u32 BL_1_MUX:1;
  u32 CH_1_MUX:1;
  u32 CL_1_MUX:1;
  u32 SYNC_1_MUX:1;
  u32 TRIP_1_MUX:1;
} ADI_PWM_PORT_MAP,  * pADI_PWM_PORT_MAP;
```

Below are the primary port mappings.

```
 ADI_PORTS_DIRECTIVE PrimaryPortsTable[ADI_PWM_NUM_MUX_TABLE_ENTRIES] =
  {
      ADI_PORTS_DIRECTIVE_PWM0_AH,
      ADI_PORTS_DIRECTIVE_PWM0_AL,
      ADI_PORTS_DIRECTIVE_PWM0_BH,
      ADI_PORTS_DIRECTIVE_PWM0_BL,
      ADI_PORTS_DIRECTIVE_PWM0_CH,
      ADI_PORTS_DIRECTIVE_PWM0_CL,
      ADI_PORTS_DIRECTIVE_PWM0_TRIP_PF6,
      ADI_PORTS_DIRECTIVE_PWM0_SYNC_PF7,
      ADI_PORTS_DIRECTIVE_PWM1_AH,
      ADI_PORTS_DIRECTIVE_PWM1_AL,
      ADI_PORTS_DIRECTIVE_PWM1_BH,
      ADI_PORTS_DIRECTIVE_PWM1_BL,
      ADI_PORTS_DIRECTIVE_PWM1_CH,
      ADI_PORTS_DIRECTIVE_PWM1_CL,
      ADI_PORTS_DIRECTIVE_PWM1_SYNC,
      ADI_PORTS_DIRECTIVE_PWM1_TRIP
};
```

Below are the secondary mappings, which are nearly identical to the primary, except for TRIP and SYNC for PWM 0.

```
 ADI_PORTS_DIRECTIVE SecondaryPortsTable[ADI_PWM_NUM_MUX_TABLE_ENTRIES] =
{
    ADI_PORTS_DIRECTIVE_PWM0_AH,
    ADI_PORTS_DIRECTIVE_PWM0_AL,
    ADI_PORTS_DIRECTIVE_PWM0_BH,
    ADI_PORTS_DIRECTIVE_PWM0_BL,
    ADI_PORTS_DIRECTIVE_PWM0_CH,
    ADI_PORTS_DIRECTIVE_PWM0_CL,
    ADI_PORTS_DIRECTIVE_PWM0_TRIP_PF11,
    ADI_PORTS_DIRECTIVE_PWM0_SYNC_PF12,
    ADI_PORTS_DIRECTIVE_PWM1_AH,
    ADI_PORTS_DIRECTIVE_PWM1_AL,
    ADI_PORTS_DIRECTIVE_PWM1_BH,
    ADI_PORTS_DIRECTIVE_PWM1_BL,
    ADI_PORTS_DIRECTIVE_PWM1_CH,
    ADI_PORTS_DIRECTIVE_PWM1_CL,
    ADI_PORTS_DIRECTIVE_PWM1_SYNC,
    ADI_PORTS_DIRECTIVE_PWM1_TRIP
};
```

The TRIP and SYNC signals on PWM 0, are the only signals which have secondary port mappings. They are selectable with a secondary mux setting, while the rest of the settings should all be set to PRIMARY, because there is only one choice. Setting them to secondary has no effect.

```
ADI_PWM_PORT_MAP pwm_PortMuxMap =
{
  ADI_PWM_MUX_PRI,  // AH_0_MUX:1;
  ADI_PWM_MUX_PRI,  // AL_0_MUX:1;
  ADI_PWM_MUX_PRI,  // BH_0_MUX:1;
  ADI_PWM_MUX_PRI,  // BL_0_MUX:1;
  ADI_PWM_MUX_PRI,  // CH_0_MUX:1;
  ADI_PWM_MUX_PRI,  // CL_0_MUX:1;
  ADI_PWM_MUX_PRI,  // SYNC_0_MUX:1;
  ADI_PWM_MUX_PRI,  // TRIP_0_MUX:1;
  ADI_PWM_MUX_PRI,  // AH_1_MUX:1;
  ADI_PWM_MUX_PRI,  // AL_1_MUX:1;
  ADI_PWM_MUX_PRI,  // BH_1_MUX:1;
  ADI_PWM_MUX_PRI,  // BL_1_MUX:1;
  ADI_PWM_MUX_PRI,  // CH_1_MUX:1;
  ADI_PWM_MUX_PRI,  // CL_1_MUX:1;
  ADI_PWM_MUX_PRI,  // SYNC_1_MUX:1;
  ADI_PWM_MUX_PRI,  // TRIP_1_MUX:1;
};
```

Below: Using the Command Pair to set the Port Mux according to the mapping defined above.

```
  { ADI_PWM_CMD_SET_PORT_MUX,    (void*) &pwm_PortMuxMap },
```

PWM Enable/Disable status

This enumeration is used in conjunction with a command, and sometimes with a channel ID, to indicate whether a particular feature is to be enabled or disabled. It is used to enable or disable channels, gate chopping, switch reluctance, Sync and Trip interrupts. For the single PWM ADSP-BF51x, this enumeration just has two possibilities: enable (1) and disable (0). For the dual PWM ADSP-BF50x, this enumeration has six possibilities. The first two, ADI_PWM_DISABLE and ADI_PWM_ENABLE, are still used as generic enable and disable values (1 and 0). There are also four values in which the PWM number is embedded, to specify not only the enable status but which PWM to enable or disable.

ADI_PWM_ENABLE_STATUS

Name	Value	Description
ADI_PWM_DISABLE	0	Generic Disable
ADI_PWM_DISABLE	1	Generic Enable
ADI_PWM0_DISABLE	0	PWM0 Disable
ADI_PWM0_DISABLE	1	PWM0 Enable
ADI_PWM1_DISABLE	2	PWM1 Disable
ADI_PWM1_DISABLE	3	PWM1 Enable

Example) enable PWM 1

```
Result = adi_pwm_Control( ADI_PWM_CMD_SET_PWM_ENABLE, (void*)ADI_PWM1_ENABLE );
```

Example) disable PWM 0

```
Result = adi_pwm_Control( ADI_PWM_CMD_SET_PWM_ENABLE, (void*)ADI_PWM0_DISABLE );
```

PWM Channel Enable/Disable status

The channel and value structure is used as a command argument to be passed along with a command to the **adi_pwm_Init()** or **adi_pwm_Control()** functions. A channel is combined with an enable status, for use with the commands ADI_PWM_CMD_SET_CHANNEL_ENABLE, ADI_PWM_CMD_SET_CROSSOVER and ADI_PWM_CMD_SET_LOW_SIDE_INVERT. This structure indicates the channel ID and the enable status of the specified signal for that channel. This exact same structure is also used for the dual PWM ADSP-BF50x, because the enable status enumeration (**ADI_PWM_ENABLE_STATUS)** also specifies the PWM for which the command is intended.

Example) enable ALL channels on PWM 0

```
ADI_PWM_CHANNEL_STATUS pwm0_EnableaAll_struct = { ADI_PWM_CHANNEL_ALL, ADI_PWM0_ENABLE };
```

Example) disable ALL channels on PWM 1

```
ADI_PWM_CHANNEL_STATUS  pwm1_Enable_struct = { ADI_PWM_CHANNEL_ALL, ADI_PWM1_DISABLE };
```

Example) enable channel pair B on PWM 0

```
ADI_PWM_CHANNEL_STATUS  pwm0_EnableB_struct = { ADI_PWM_CHANNEL_B,  ADI_PWM0_ENABLE };
```

Example) enable channel pair CL for PWM1.

```
ADI_PWM_CHANNEL_STATUS  pwm1_EnableCL_struct = { ADI_PWM_CHANNEL_CL,  ADI_PWM1_ENABLE }
```

For example purposes only, the command pair table below combines the above example command pairs.

```
ADI_PWM_COMMAND_PAIR PWMInitTable[] =
{
    { ADI_PWM_CMD_SET_CHANNEL_ENABLE, (void*)  & pwm0_EnableaAll_struct },
    { ADI_PWM_CMD_SET_CHANNEL_ENABLE, (void*)  & pwm1_Enable_struct },
    { ADI_PWM_CMD_SET_CHANNEL_ENABLE, (void*)  & pwm0_EnableB_struct },
    { ADI_PWM_CMD_SET_CHANNEL_ENABLE, (void*)  & pwm1_EnableCL_struct }
    /* indicates the last command of the table */
     { ADI_PWM_CMD_END, (void*) 0    }
};
```

PWM Operating Mode

For the dual PWM on the ADSP-BF50x, the enumeration for setting the operating Update Mode (Double or Single Update) has a single and double update choice for each of the two PWMs. These enumerations are used the same as before.

ADI_PWM_EVENT_UPDATE_MODE

Name	Description
ADI_PWM0_SINGLE_UPDATE	single update operating mode on PWM 0
ADI_PWM0_DOUBLE_UPDATE	double update operating mode on PWM 0
ADI_PWM1_SINGLE_UPDATE	single update operating mode on PWM 1
ADI_PWM1_DOUBLE_UPDATE	double update operating mode on PWM 1

Setting Internal or External Sync Pulse

This enumeration is used in conjunction with the command ADI_PWM_CMD_SET_SYNC_SOURCE to set the synchronization pulse source to external or external.

ADI_PWM_SYNC_SOURCE

Name	Numeric Value	Description
ADI_PWM0_SOURCE_INTERNAL	0	Internal Source for PWM 0
ADI_PWM0_ SOURCE_EXTERNAL	1	External Source for PWM 0
ADI_PWM1_SOURCE_INTERNAL	2	Internal Source for PWM 1
ADI_PWM1_ SOURCE_EXTERNAL	3	External Source for PWM 1

For ADSP-BF50x, there is an internal and external choice for each PWM. The parameter is passed the same way as before. When the synchronization pulse source is external, this enumeration is used in conjunction with the command ADI_PWM_CMD_SET_SYNC_SELECT to set the external synchronization pulse to synchronous or asynchronous.

Setting the External Sync Pulse Source (synch or asynch)

For ADSP-BF50x, there is a synch and asynch choice for each PWM. The parameter is passed the same way as before.

ADI_PWM_SYNC_SEL

Name	Numeric Value	Description
ADI_PWM0_SYNC_ASYNCH	0	External sync source for PWM 0 is asynchronous
ADI_PWM0_SYNC_SYNCH	1	External sync source for PWM 0is synchronous.
ADI_PWM1_SYNC_ASYNCH	2	External sync source for PWM 1 is asynchronous
ADI_PWM1_SYNC_SYNCH	3	External sync source for PWM 1 is synchronous.

Setting PWM Polarity

This struct is used to set the polarity for all signals in PWMCTRL register. For Moy there is a LOW and HIGH choice for each PWM. The parameter is passed the same way as before.

ADI_PWM_POLARITY

Name	Numeric Value	Description
ADI_PWM0_POLARITY_LOW	0	Low polarity for PWM 0
ADI_PWM0_POLARITY_HIGH	1	High polarity for PWM 0
ADI_PWM1_POLARITY_LOW	2	Low polarity for PWM 1
ADI_PWM1_POLARITY_HIGH	3	High polarity for PWM 1

New Struct for combining PWM number with a 32 bit Value

Commands that requires a simple u32 type value (for example, setting Sync period, Pulse Width, Dead Time) also require a PWM NUMBER, to specify which PWM the command is intended for. The PWM number is combined with the u32 value, such as dead time or the period, when passed to the adi_pwm_Init() or adi_pwm_Control functions().

PWM_NUMBER_AND_VALUE

Type	Name	Description
ADI_PWM_NUMBER	PwmNumber	PWM0 or PWM1
Void *	Value	the value to set

Example) Set Sync period value for PWM 0.

```
ADI_PWM_NUMBER_AND_VALUE  pwm0_SyncPeriod = { ADI_PWM_0,  PWM_SyncPeriod };
{ ADI_PWM_CMD_SET_PERIOD,    (void*) &pwm0_SyncPeriod },
```

Example) Set Sync period value for PWM 1.

```
ADI_PWM_NUMBER_AND_VALUE  pwm1_SyncPeriod = { ADI_PWM_1,  PWM_SyncPeriod };
```

Use in command pair table:

```
{ ADI_PWM_CMD_SET_PERIOD,     (void*) &pwm1_SyncPeriod },
```

Example) Set Sync Pulse Width value for PWM 0.

```
ADI_PWM_NUMBER_AND_VALUE  pwm0_SyncWidth = { ADI_PWM_0,  PWM_SyncWidth };
```

Use in command pair table:

```
{ ADI_PWM_CMD_SET_SYNC_PULSE_WIDTH,  (void*) &pwm0_SyncWidth },
```

Example) Set Sync Pulse Width value for PWM 1.

```
ADI_PWM_NUMBER_AND_VALUE  pwm1_SyncWidth = { ADI_PWM_1,  PWM_SyncWidth };
```

Use in command pair table:

```
{ ADI_PWM_CMD_SET_SYNC_PULSE_WIDTH,  (void*)  &pwm1_SyncWidth },
```

Example) Set Dead Time value for PWM 0

```
ADI_PWM_NUMBER_AND_VALUE pwm0_DeadTime = { ADI_PWM_0,  PWM_DeadTime };
```

Use in command pair table:

```
{ ADI_PWM_CMD_SET_DEAD_TIME,   (void*) &pwm0_DeadTime },
```

Example) Set Dead Time value for PWM 1

```
ADI_PWM_NUMBER_AND_VALUE pwm1_DeadTime = { ADI_PWM_1, PWM_DeadTime };
```

Use in command pair table:

```
{ ADI_PWM_CMD_SET_DEAD_TIME,   (void*) &pwm1_DeadTime },
```

Structs for combining PWM number with other data structures, for “Get” commands.

When “getting” a value from the PWM service, using commands such as ADI_PWM_CMD_GET_SWITCH_RELUCTANCE (and others) the new structures, ADI_PWM_NUMBER_AND_ENABLE_STATUS and ADI_PWM_NUMBER_AND_CHANNEL_STATUS, allow the PWM number to be specified in the “PwmNumber” field, so that the proper data type is returned in the “Status” field.

PWM_NUMBER_AND_ENABLE_STATUS

Type	Name	Description
ADI_PWM_NUMBER	PwmNumber	Either PWM0 or PWM1
ADI_PWM_ENABLE_STATUS	Status	Enable Status

Example) Use PWM_NUMBER_AND_ENABLE_STATUS structure to get switch reluctance enable status for PWM 0.

```
ADI_PWM_NUMBER_AND_ENABLE_STATUS  PwmNumStatus;
PwmNumStatus.PwmNumber = ADI_PWM_0;
pADI_PWM_NUMBER_AND_ENABLE_STATUS  pPwmNumStatus = & PwmNumStatus;

adi_pwm_Control(  ADI_PWM_CMD_GET_SWITCH_RELUCTANCE, pPwmNumStatus );
Result  =  PwmNumStatus.Status;
```

When “getting” a value from the PWM service, by passing a command such as ADI_PWM_CMD_GET_CHANNEL_ENABLE, the PWM_NUMBER_AND_CHANNEL_STATUS structure allows the PWM number to be specified in the “PwmNumber” field, and a ADI_PWM_CHANNEL_STATUS data type to be passed, so that the proper enable status for the proper channel can be returned in the “ChannelStatus” field.

PWM_NUMBER_AND_ CHANNEL_STATUS

Type	Name	Description
ADI_PWM_NUMBER	PwmNumber	Either PWM0 or PWM1
ADI_PWM_CHANNEL_STATUS	ChannelStatus	Channel status value

Example) Get the enable status for PWM 0 Channel B.

```
ADI_PWM_CHANNEL_STATUS  ChannelBEnableStruct;
```

```
ChannelBEnableStruct.Channel =  ADI_PWM_CHANNEL_B;
ADI_PWM_NUMBER_AND_CHANNEL_STATUS  PwmNumChannelStatus;
PwmNumChannelStatus.PwmNumber = ADI_PWM_0;
PwmNumChannelStatus.ChannelStatus = ChannelBEnableStruct;
```

Pass to adi_pwm_Control( )

```
adi_pwm_Control(   ADI_PWM_CMD_GET_CHANNEL_ENABLE, &PwmNumChannelStatus );
Result = pPwmNumChannelStatus->Status;
```

Programming Example for two PWMs on the ADSP-BF50x

A sine wave example for the ADSP-BF51x family is currently installed in the VisualDSP folder "…\VisualDSP 5.0\Blackfin\Examples\ADSP-BF518F EZ-Board\Services\PWM\pwm_sine_wave".

The example program shown below is the ADSP-BF50x equivalent of that sine wave example.

Both PWM 0 and PWM 1 are used in this example.

```
void main(void)
{
   ADI_PWM_RESULT Result;
   u32 CompareValue;

   u32 fcclk, fsclk, fvco;
   ADI_PWM_NUMBER_AND_CHANNEL_STATUS GetPwmChannelStatus;

   ADI_PWM_NUMBER pwm;
   ADI_PWM_CHANNEL channel;
   u32 EnableStatus;

   ADI_PWM_PORT_MAP pwm_PortMuxMap =
   {
     ADI_PWM_MUX_PRI,   // AH_0_MUX:1;
     ADI_PWM_MUX_PRI,   // AL_0_MUX:1;
     ADI_PWM_MUX_PRI,   // BH_0_MUX:1;
     ADI_PWM_MUX_PRI,   // BL_0_MUX:1;
     ADI_PWM_MUX_PRI,   // CH_0_MUX:1;
     ADI_PWM_MUX_PRI,   // CL_0_MUX:1;
     ADI_PWM_MUX_PRI,   // SYNC_0_MUX:1;
     ADI_PWM_MUX_PRI,   // TRIP_0_MUX:1;
     ADI_PWM_MUX_PRI,   // AH_1_MUX:1;
     ADI_PWM_MUX_PRI,   // AL_1_MUX:1;
     ADI_PWM_MUX_PRI,   // BH_1_MUX:1;
     ADI_PWM_MUX_PRI,   // BL_1_MUX:1;
     ADI_PWM_MUX_PRI,   // CH_1_MUX:1;
     ADI_PWM_MUX_PRI,   // CL_1_MUX:1;
     ADI_PWM_MUX_PRI,   // SYNC_1_MUX:1;
     ADI_PWM_MUX_PRI,   // TRIP_1_MUX:1;
   } ;

   /* This structure enables all channels at once  for PWM 0 */
   ADI_PWM_CHANNEL_STATUS  pwm_EnablePWM0_struct = { ADI_PWM_CHANNEL_ALL,  ADI_PWM0_ENABLE };

   /* This structure enables all channels at once  for PWM 1 */
   ADI_PWM_CHANNEL_STATUS  pwm_EnablePWM1_struct = { ADI_PWM_CHANNEL_ALL,  ADI_PWM1_ENABLE };
```

```
PWM_PHASE_DATA PhaseData;
 /* 0 = 50% duty cycle */
 PhaseData.A = 0;
 PhaseData.B = 0;
 PhaseData.C = 0;

 pPWM_PHASE_DATA  pPhaseData =  &PhaseData;

 /* Initialize the system services */
 Result = (ADI_PWM_RESULT)adi_ssl_Init();

/* Request the system clock from the power management service */
adi_pwr_GetFreq(&fcclk, &fsclk, &fvco);

/* Calculate the dead time, period and pulse width parameters based on (SCLK) system clock. */
u32 PWM_SyncPeriod = (fsclk / PWM_SWITCHING_FREQUENCY / 2);

fsclk = fsclk / 1000000;

u32 PWM_DeadTime  =   (PWM_DEAD_TIME / 1000 * fsclk / 2 );
u32 PWM_SyncWidth  = (SYNC_WIDTH  / 1000 * fsclk );

PhaseData.Max_Duty_Value = (PWM_SyncPeriod/2) + PWM_DeadTime;

/* use this structure to set Sync period  value for a specific PWM number */
ADI_PWM_NUMBER_AND_VALUE pwm0_SyncWidth = { ADI_PWM_0, PWM_SyncWidth };
ADI_PWM_NUMBER_AND_VALUE pwm1_SyncWidth = { ADI_PWM_1, PWM_SyncWidth };

/* use this structure to set Sync period  value for a specific PWM number */
ADI_PWM_NUMBER_AND_VALUE pwm0_SyncPeriod = { ADI_PWM_0, PWM_SyncPeriod };
ADI_PWM_NUMBER_AND_VALUE pwm1_SyncPeriod = { ADI_PWM_1, PWM_SyncPeriod };

/* use this structure to set Dead Time  value for a specific PWM number */
ADI_PWM_NUMBER_AND_VALUE pwm0_DeadTime = { ADI_PWM_0, PWM_DeadTime };
ADI_PWM_NUMBER_AND_VALUE pwm1_DeadTime = { ADI_PWM_1, PWM_DeadTime };

/* Below are the structures to set duty cycles for individual channel pairs, on each PWM */

/* Populate the three structures to set the duty cycles for the three channels on PWM 0*/
ADI_PWM_CHANNEL_DUTY_CYCLE  pwm0_dutyA_struct = { ADI_PWM_0, ADI_PWM_CHANNEL_A, pPhaseData->A };
ADI_PWM_CHANNEL_DUTY_CYCLE  pwm0_dutyB_struct = { ADI_PWM_0, ADI_PWM_CHANNEL_B, pPhaseData->B };
ADI_PWM_CHANNEL_DUTY_CYCLE  pwm0_dutyC_struct =  {ADI_PWM_0, ADI_PWM_CHANNEL_C, pPhaseData->C };

/* Populate the three structures to set the duty cycles for the three channels on PWM 1 */
ADI_PWM_CHANNEL_DUTY_CYCLE  pwm1_dutyA_struct = { ADI_PWM_1, ADI_PWM_CHANNEL_A, pPhaseData->A, };
ADI_PWM_CHANNEL_DUTY_CYCLE  pwm1_dutyB_struct = { ADI_PWM_1, ADI_PWM_CHANNEL_B, pPhaseData->B, };
ADI_PWM_CHANNEL_DUTY_CYCLE  pwm1_dutyC_struct = { ADI_PWM_1, ADI_PWM_CHANNEL_C, pPhaseData->C, };

 /* structures to enable crossover for each channel pair on PWM 0 */
 ADI_PWM_CHANNEL_STATUS pwm0_CrossA_struct = { ADI_PWM_CHANNEL_A, ADI_PWM0_DISABLE  };
 ADI_PWM_CHANNEL_STATUS pwm0_CrossB_struct = { ADI_PWM_CHANNEL_B, ADI_PWM0_DISABLE  } ;
 ADI_PWM_CHANNEL_STATUS pwm0_CrossC_struct = { ADI_PWM_CHANNEL_C, ADI_PWM0_DISABLE  };

 /* structures to enable crossover for each channel pair on PWM 1 */
 ADI_PWM_CHANNEL_STATUS pwm1_CrossA_struct = { ADI_PWM_CHANNEL_A, ADI_PWM1_DISABLE };
 ADI_PWM_CHANNEL_STATUS pwm1_CrossB_struct = { ADI_PWM_CHANNEL_B, ADI_PWM1_DISABLE } ;
 ADI_PWM_CHANNEL_STATUS pwm1_CrossC_struct = { ADI_PWM_CHANNEL_C, ADI_PWM1_DISABLE };

/* OPTIONAL!! – set IVG num of PWM 0 */
ADI_PWM_NUMBER_AND_VALUE    pwm0_TRIPIVG_struct = { ADI_PWM_0,  7 };
ADI_PWM_NUMBER_AND_VALUE    pwm0_SYNCIVG_struct = { ADI_PWM_0,  11 };

 /* OPTIONAL!! – set IVG num of PWM 1  */
 ADI_PWM_NUMBER_AND_VALUE    pwm1_TRIPIVG_struct = { ADI_PWM_1, 7 };
 ADI_PWM_NUMBER_AND_VALUE    pwm1_SYNCIVG_struct = { ADI_PWM_1, 11 };

/* Create the PWM Initialization Command Pair Table */
```

```
    ADI_PWM_COMMAND_PAIR PWMInitTable[] =
    {
        /* required initialization commands */
        { ADI_PWM_CMD_SET_PORT_MUX,   (void*) &pwm_PortMuxMap },

        { ADI_PWM_CMD_SET_PERIOD,   (void*) &pwm0_SyncPeriod },
        { ADI_PWM_CMD_SET_PERIOD,   (void*) &pwm1_SyncPeriod },

        { ADI_PWM_CMD_SET_SYNC_PULSE_WIDTH,  (void*) &pwm0_SyncWidth },
        { ADI_PWM_CMD_SET_SYNC_PULSE_WIDTH,  (void*) &pwm1_SyncWidth },

        { ADI_PWM_CMD_SET_DEAD_TIME,   (void*) &pwm0_DeadTime },
        { ADI_PWM_CMD_SET_DEAD_TIME,   (void*) &pwm1_DeadTime },

        { ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm0_dutyA_struct },
        { ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm0_dutyB_struct },
        { ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm0_dutyC_struct },

        { ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm1_dutyA_struct },
        { ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm1_dutyB_struct },
        { ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm1_dutyC_struct },

        { ADI_PWM_CMD_SET_CHANNEL_ENABLE, (void*)  &pwm_EnablePWM0_struct },
        { ADI_PWM_CMD_SET_CHANNEL_ENABLE, (void*)  &pwm_EnablePWM1_struct },

        { ADI_PWM_CMD_SET_POLARITY,   (void*) ADI_PWM0_POLARITY_HIGH },
        { ADI_PWM_CMD_SET_POLARITY,   (void*) ADI_PWM1_POLARITY_HIGH },

        { ADI_PWM_CMD_SET_UPDATE_MODE,  (void*) ADI_PWM0_DOUBLE_UPDATE },
        { ADI_PWM_CMD_SET_UPDATE_MODE,  (void*) ADI_PWM1_DOUBLE_UPDATE },

        /* OPTIONAL – change the IVG num of PWM 0 and 1  */
        { ADI_PWM_CMD_SET_SYNC_IVG,      (void*) &pwm0_SYNCIVG_struct },
        { ADI_PWM_CMD_SET_TRIP_IVG,      (void*) &pwm0_TRIPIVG_struct },
        { ADI_PWM_CMD_SET_SYNC_IVG,      (void*) &pwm1_SYNCIVG_struct },
        { ADI_PWM_CMD_SET_TRIP_IVG,      (void*) &pwm1_TRIPIVG_struct },

        /* indicate the last command of the table */
        { ADI_PWM_CMD_END, (void*)0      }
    };

    /* Initialize the PWM  service */
     Result =  adi_pwm_Init( PWMInitTable, (void*)NULL );

    /* Install Sync Callback PWM 0 */
    Result = adi_pwm_InstallCallback( ADI_PWM0_EVENT_SYNC, (void*)pPhaseData,
                                                NULL, (ADI_DCB_CALLBACK_FN) PwmCallback  );

     /* Install Sync Callback PWM 1  */
     Result = adi_pwm_InstallCallback( ADI_PWM1_EVENT_SYNC, (void*)pPhaseData,
                                                NULL, (ADI_DCB_CALLBACK_FN) PwmCallback  );

    /* finally, enable the PWM 0 */
    Result = adi_pwm_Control( ADI_PWM_CMD_SET_PWM_ENABLE, (void*)ADI_PWM0_ENABLE );

     /* finally, enable the PWM 1 */
    Result = adi_pwm_Control( ADI_PWM_CMD_SET_PWM_ENABLE, (void*)ADI_PWM1_ENABLE );

    While ( 1 )
    {
       /* main loop */
    }
}


/*****************************************************************
  PWM Callback
  The callback handles the PWM trip and/or sync event.
```

```
*****************************************************************/
static void PwmCallback(void *ClientHandle, u32  Event, void *pArg)
{
   ADI_PWM_RESULT Result;

   float PhaseA;
   float PhaseB;
   float PhaseC;

   switch ( (u32)Event )
   {
      case ADI_PWM0_EVENT_SYNC:

        /* This event occurs when the synchronization interrupt is enabled, the callback is installed for the synchronization event, and the
           sync pulse occurs. Typically the callback updates the three PWM channel duties according to their control algorithm based on
           expected motor operation and sampled existing motor operation.  It may also trigger an ADC to sample data  */
     {
        /* structures to set duty cycle for individual channel pairs */
        pPWM_PHASE_DATA pPhaseData = (PWM_PHASE_DATA*)ClientHandle;


        /* Perform Control Algorithm to get next duty cycles */
        ControlAlgorithm( pPhaseData );

        /* Populate the three structures to set the duty cycles for the three channels */
        ADI_PWM_CHANNEL_DUTY_CYCLE  pwm0_dutyA_struct = {ADI_PWM_0, ADI_PWM_CHANNEL_A, pPhaseData->A };
        ADI_PWM_CHANNEL_DUTY_CYCLE  pwm0_dutyB_struct = {ADI_PWM_0, ADI_PWM_CHANNEL_B, pPhaseData->B };
        ADI_PWM_CHANNEL_DUTY_CYCLE  pwm0_dutyC_struct = {ADI_PWM_0, ADI_PWM_CHANNEL_C, pPhaseData->C };

        /* define a command pair table to pass, to update the three duty cycles */
        ADI_PWM_COMMAND_PAIR PWM_UpdateTable [] =
        {
           { ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm0_dutyA_struct },
           { ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm0_dutyB_struct },
           { ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm0_dutyC_struct },
           /* indicate the last command of the table */
           { ADI_PWM_CMD_END, (void*)0 }
        };

        /* Pass the command pair table to the control function */
        Result = adi_pwm_Control(  ADI_PWM_CMD_TABLE, (void*)PWM_UpdateTable );
        break;

   case ADI_PWM1_EVENT_SYNC:

       /* This event occurs when the synchronization interrupt is enabled, the callback is installed for the synchronization event, and the
          sync pulse occurs. Typically the callback updates the three PWM channel duties according to their control algorithm based on
          expected motor operation and sampled existing motor operation.   It may also trigger an ADC to sample data  */
     {

        /* structures to set duty cycle for individual channel pairs */
        pPWM_PHASE_DATA pPhaseData = (PWM_PHASE_DATA*)ClientHandle;

        /* Perform Control Algorithm to get next duty cycles */
        ControlAlgorithm( pPhaseData );

        /* Populate the three structures to set the duty cycles for the three channels */
        ADI_PWM_CHANNEL_DUTY_CYCLE  pwm1_dutyA_struct = {ADI_PWM_1, ADI_PWM_CHANNEL_A, pPhaseData->A };
        ADI_PWM_CHANNEL_DUTY_CYCLE  pwm1_dutyB_struct = {ADI_PWM_1, ADI_PWM_CHANNEL_B, pPhaseData->B };
        ADI_PWM_CHANNEL_DUTY_CYCLE  pwm1_dutyC_struct = {ADI_PWM_1, ADI_PWM_CHANNEL_C, pPhaseData->C };

        /* define a command pair table to pass, to update the three duty cycles */
        ADI_PWM_COMMAND_PAIR PWM_UpdateTable [] =
        {
           { ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm1_dutyA_struct },
           { ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm1_dutyB_struct },
           { ADI_PWM_CMD_SET_DUTY_CYCLE,  (void*) &pwm1_dutyC_struct },
           /* indicate the last command of the table */
```

```
            { ADI_PWM_CMD_END, (void*)0 }
        };

        /* Pass the command pair table to the control function */
        Result = adi_pwm_Control(  ADI_PWM_CMD_TABLE, (void*)PWM_UpdateTable );
        break;
      }
      case ADI_PWM0_EVENT_TRIP:
        /* This event occurs if the Trip input is enabled, the trip interrupt is enabled, and this callback is installed for the trip event. */
      {
        Result = adi_pwm_Control( ADI_PWM_CMD_CLEAR_TRIP_INT, (void*) ADI_PWM_0 );
        break;
      }
      case ADI_PWM1_EVENT_TRIP:
        /* This event occurs if the Trip input is enabled, the trip interrupt is enabled, and this callback is installed for the trip event. */
      {
        Result = adi_pwm_Control( ADI_PWM_CMD_CLEAR_TRIP_INT, (void*) ADI_PWM_1 );
        break;
      }
    }
    return;
}
```