PWM

Home

Apps

Here is a list of all modules:

PWM

License Terms and Copyright Information

License Terms and Copyright Information

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT

(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

To improve the quality of the software, users are encouraged to share modifications, enhancements or bug fixes with Infineon Technologies AG (dave@infineon.com).

PWM

Abbreviations and Definitions

Abbreviations and Definitions

Abbreviations:	
DAVE™	Digital Application Virtual Engineer
APP	DAVE™ Application
API	Application Programming Interface
GUI	Graphical User Interface
MCU	Microcontroller Unit
SW	Software
HW	Hardware
LLD	Low Level Driver
IO	Input Output
CCU4	Capture Compare Unit 4
CCU8	Capture Compare Unit 8
CCUx	Capture Compare Unit 4/Capture Compare Unit 8

Definitions:	
Singleton	Only single instance of the APP is permitted
Sharable	Resource sharing with other APPs is permitted
initProvider	Provides the initialization routine
Physical connectivity	Hardware inter/intra peripheral (constant) signal connection
Conditional connectivity	Constrained hardware inter/intra peripheral signal connection
Aggregation	Indicates consumption of low level (dependent)

	DAVE APPs

PWM

Home

Overview

Overview

The PWM APP provides the following functionalities using CCU4 or CCU8 peripheral:

1. It allows to enter the desired PWM frequency and the duty cycle.
2. The APP tries to get the best PWM resolution (timer tick). It calculates Timer Tick, Period Value (PV) and Compare Value (CV). Note: The desired values may not be exactly possible. The timer is always used in 16bit mode with prescalar.
3. PWM can be started after initialization or at a later time as required by calling the API **PWM_Start()**
4. PWM can operate in single shot mode or continuous mode.
5. Allows generation of period match or compare match events for interrupt generation (to be connected to an INTERRUPT APP).
6. Connects the PWM output to a GPIO (open drain, push/pull) or to other peripherals via interconnect (timer status flag/ period match event / compare match event)

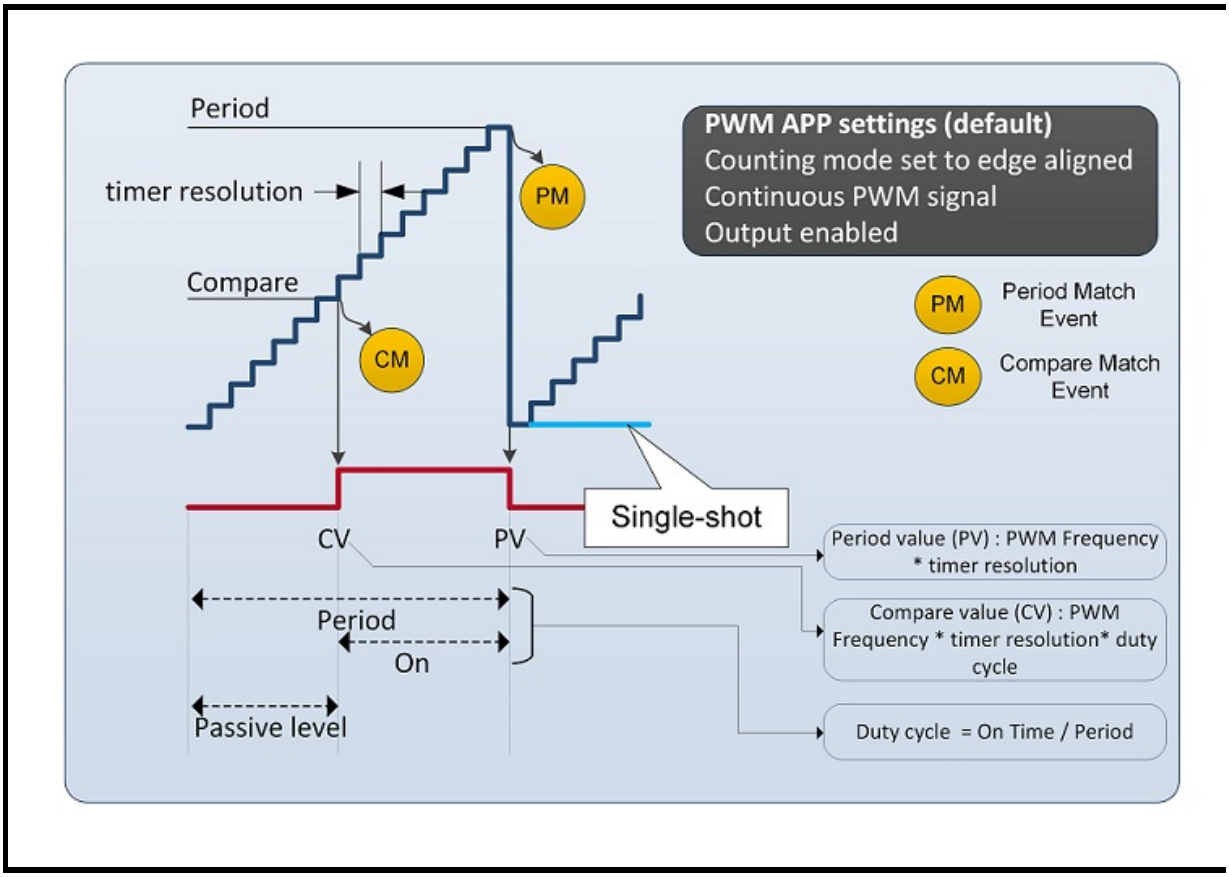


Figure 1 : Overview of PWM APP

Figure 1, shows the functional overview of the PWM APP. The CCU peripheral starts a internal counter which counts the clock pulses provided by the prescalar. When the count reaches the compare match value, the PWM output state will change from passive state (either a high state or low state) to active state(either a low state or high state). The timer will still continue to count even after the compare match event has occurred. When the count value reaches the Period match value, the PWM output state is returned to the passive state. Thus completing one cycle of the PWM. If Single-Shot mode is not selected, then this cycle would repeat continuously. If Single-Shot mode is selected, then the PWM will remain in the passive state and the timer will stop running.

Period match(PM) value is calculated based on the frequency and Compare match is calculated based on duty cycle. The PWM output state changes at period and compare match.

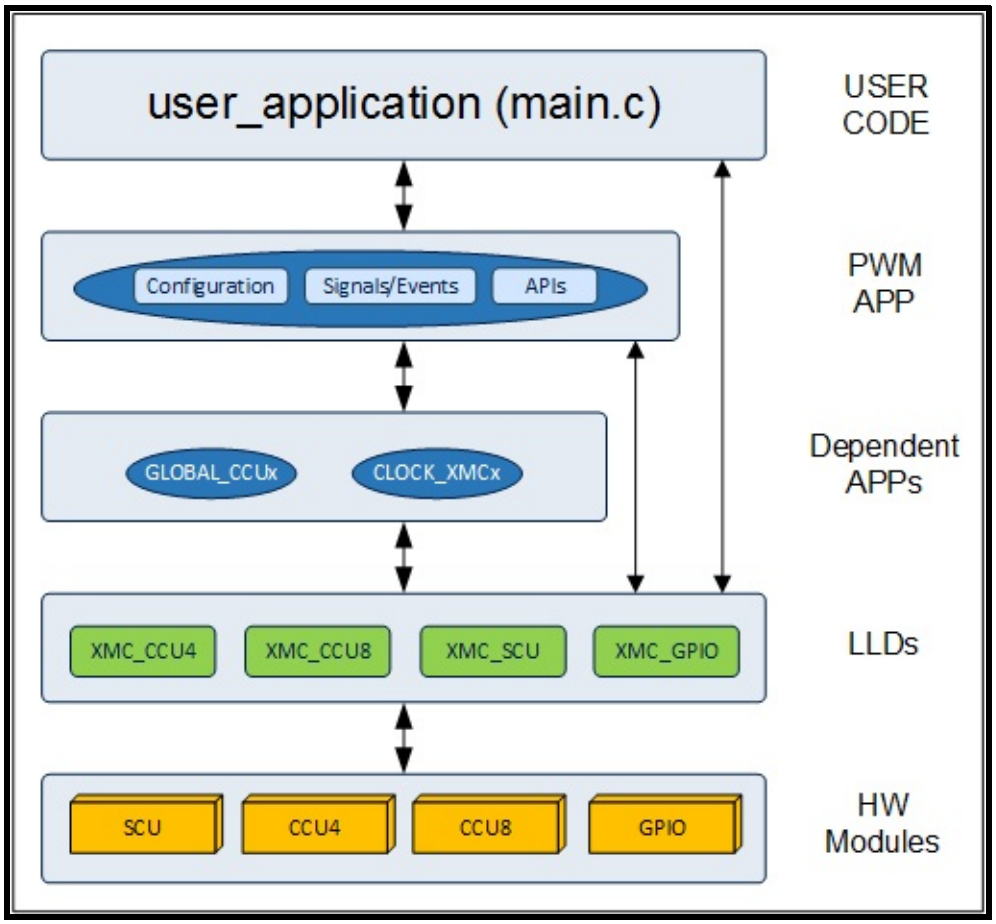


Figure 2 : Hardware and Software connectivity of PWM APP

Figure 2 shows how the APP is structured in DAVE™. XMC controllers provide the CCU4 or CCU8 module to generate the PWM waveforms. The LLD layer provides abstraction for these hardware modules. The PWM APP uses CCU4 or CCU8, SCU and GPIO LLDs and other dependent APPS like GLOBAL_CCUx (x =4, 8) and CLOCK_XMCx (x =4, 1) for the functionality.

Limitations

- This APP does not support coherent update of multiple channels (i.e. multiple instances of the APP). For this purpose, please use the functions available in CCU4/8 LLD.
- Currently, in the PWM APP, API: PWM_SetFreq allows the user to set frequency for the appropriate prescaler value to be used. However, PSIV value is updated immediately and not during period

match. Due to this, if the frequency requires the prescaler to be changed, there is a cycle (1 pulse) where the on/off time is different.

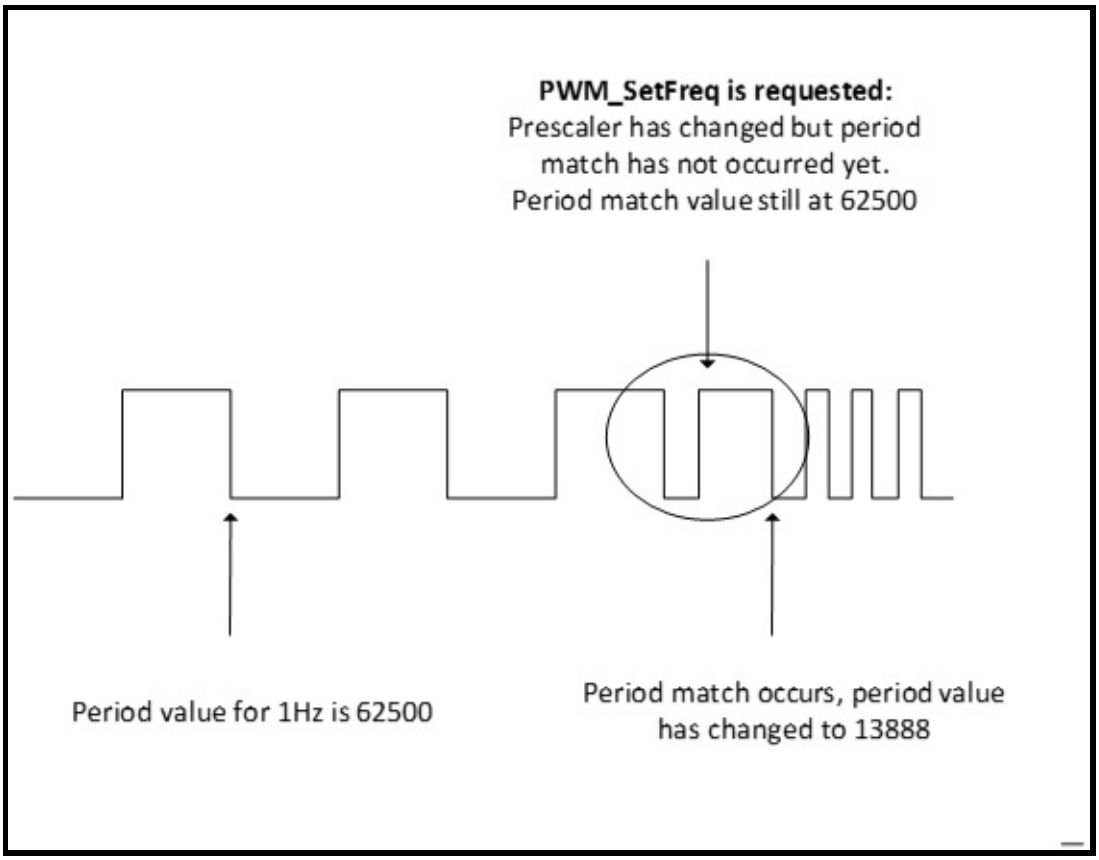


Figure 3 : Limitation of the PWM APP

Supported Devices

The APP supports below devices:

1. XMC4800 / XMC4700 Series
2. XMC4500 Series
3. XMC4400 Series
4. XMC4200 / XMC4100 Series
5. XMC1400 Series
6. XMC1300 Series
7. XMC1200 Series
8. XMC1100 Series

Reference

1. XMC4800 / XMC4700 Reference Manual
2. XMC4500 Reference Manual
3. XMC4400 Reference Manual
4. XMC4200 / XMC4100 Reference Manual
5. XMC1400 Reference Manual
6. XMC1300 Reference Manual
7. XMC1200 Reference Manual
8. XMC1100 Reference Manual

PWM

Home

Architecture Description

Architecture Description

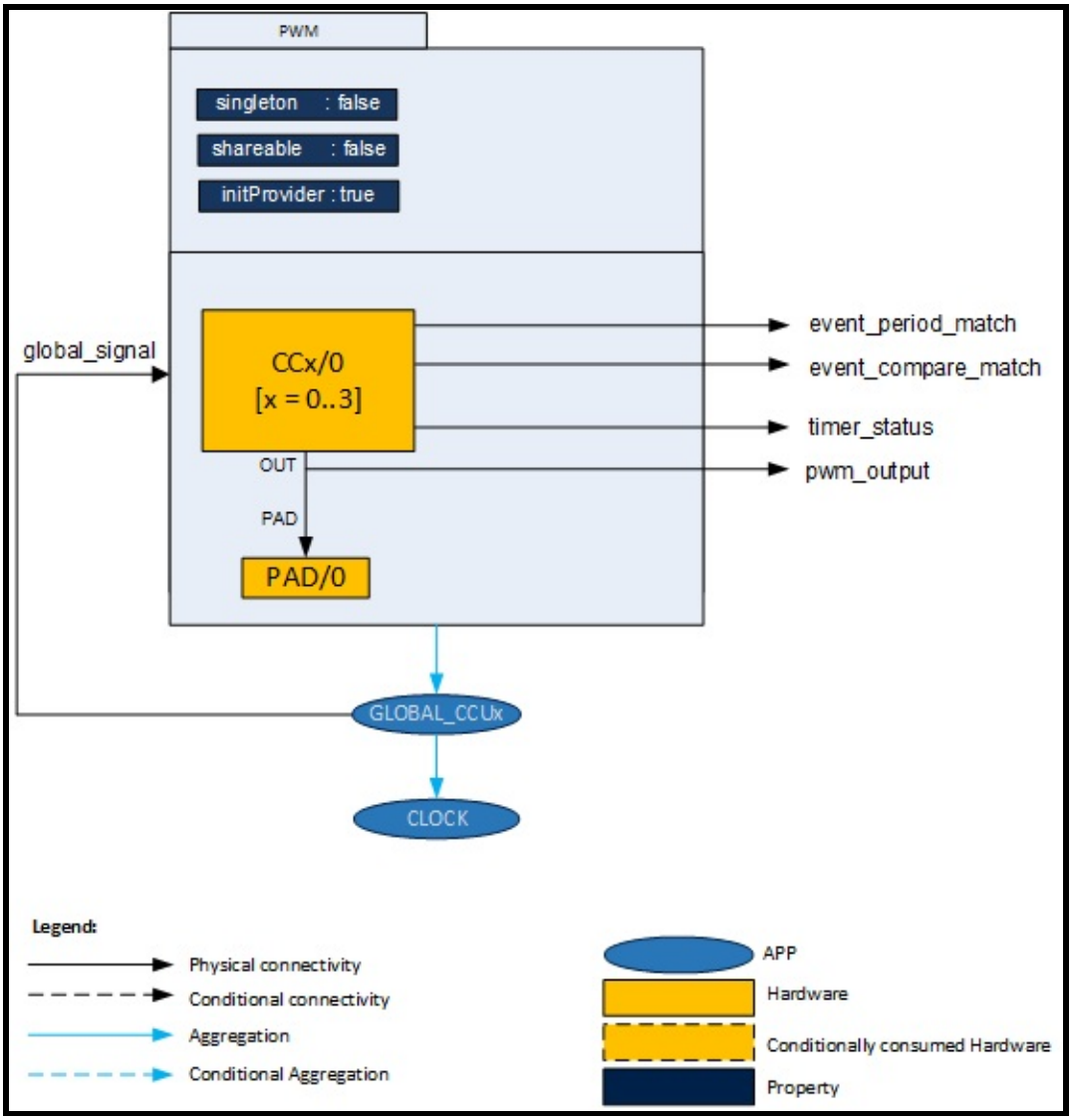


Figure 1 : Architecture of PWM APP

Figure 1 shows the internal software architecture of the PWM APP. The

figure shows the consumed hardware resources, dependent APPs and various signals which are exported out. A PWM APP instance exists in a DAVE™ project with fixed attributes as shown in **Figure 1** and uses the CCU peripheral for generating a PWM signal. This in addition requires the consumption of the GLOBAL_CCUx (x =4, 8) and CLOCK APPS for its configuration and functioning. The PWM APP also provides output signals for inter-peripheral connections.

An instantiated APP (after code generation) generates a specific data structure with the GUI configuration. The name of this data structure can be modified by changing the APP instance label (e.g. change label from default PWM_0 to MY_PWM).

Signals:

The following table presents the signals provided by the APP for connection. It also gives the flexibility to configure and extend the connectivity to other APPs.

Table 1: APP I0 signals

Signal Name	Input/Output	Availability	Description
event_period_match	Output	Always	Period match interrupt signal: This can be connected to an INTERRUPT APP to generate the interrupt for each period match event.
			Compare match interrupt signal: This can be connected with INTERRUPT

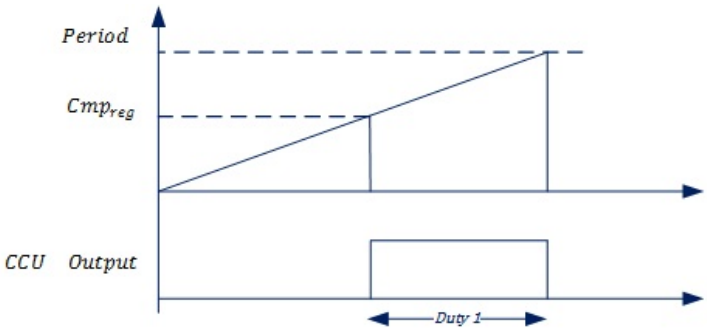
event_compare_match	Output	Always	APP to generate the interrupt for each compare match event.
timer_status	Output	Always	Timer status (ST) signal: This is the slice comparison status value. It can be used as a trigger input to other peripheral modules (e.g. ADC, CCU4, CCU8).
pwm_output	Output	Always	Output (OUT) signal: The output PWM signal can be connected with any pad pin. The list of available pins are shown in "Manual Pin Allocator" tab in DAVE™
			Global signal connection Connected between the kernel and the respective slice. Used to

global_signal	Input	Always	constrain the slice to the kernel provided by GLOBAL_CCUx APP. Connected by default at instantiation.

Frequency and duty cycle

CCU in Edge aligned Symmetric Mode of operation.

In this mode of operation we can use the compare registers to generate 1 output. The minimum duty that can be generated is 0% and maximum is 100%. Here the output is initially LOW until compare match happens. The output remains HIGH until the next one match happens.



$$Cmp_{reg} = \frac{Period * (100 - Duty)}{100}$$

Duty	Duty Min %	Duty Max %
Duty	0	100

Note: The value loaded to period register is (period – 1)

Figure 2: Edge Aligned Symmetric Mode

Example:

Let

$Clock = 120MHz.$

$Prescaler = 0;$

$Required\ frequency\ of\ operation\ (F) = 100KHz.$

$Duty\ required\ (D) = 30\%.$

$$Period = \frac{Clock}{(1 \ll Prescaler)F}$$

$$Cmp_{reg} = \frac{Period\ (100 - D)}{100}$$

$$Period = \frac{120{,}000{,}000}{(1 \ll 0)100{,}000}$$

$Period = 1200$

Note: The value loaded to period register is (period - 1) i.e. 1199.

$$Cmp_{reg} = \frac{1200(100 - 30)}{100}$$

$Cmp_{reg} = 840$

PWM

Home

APP Configuration Parameters

App Configuration Parameters

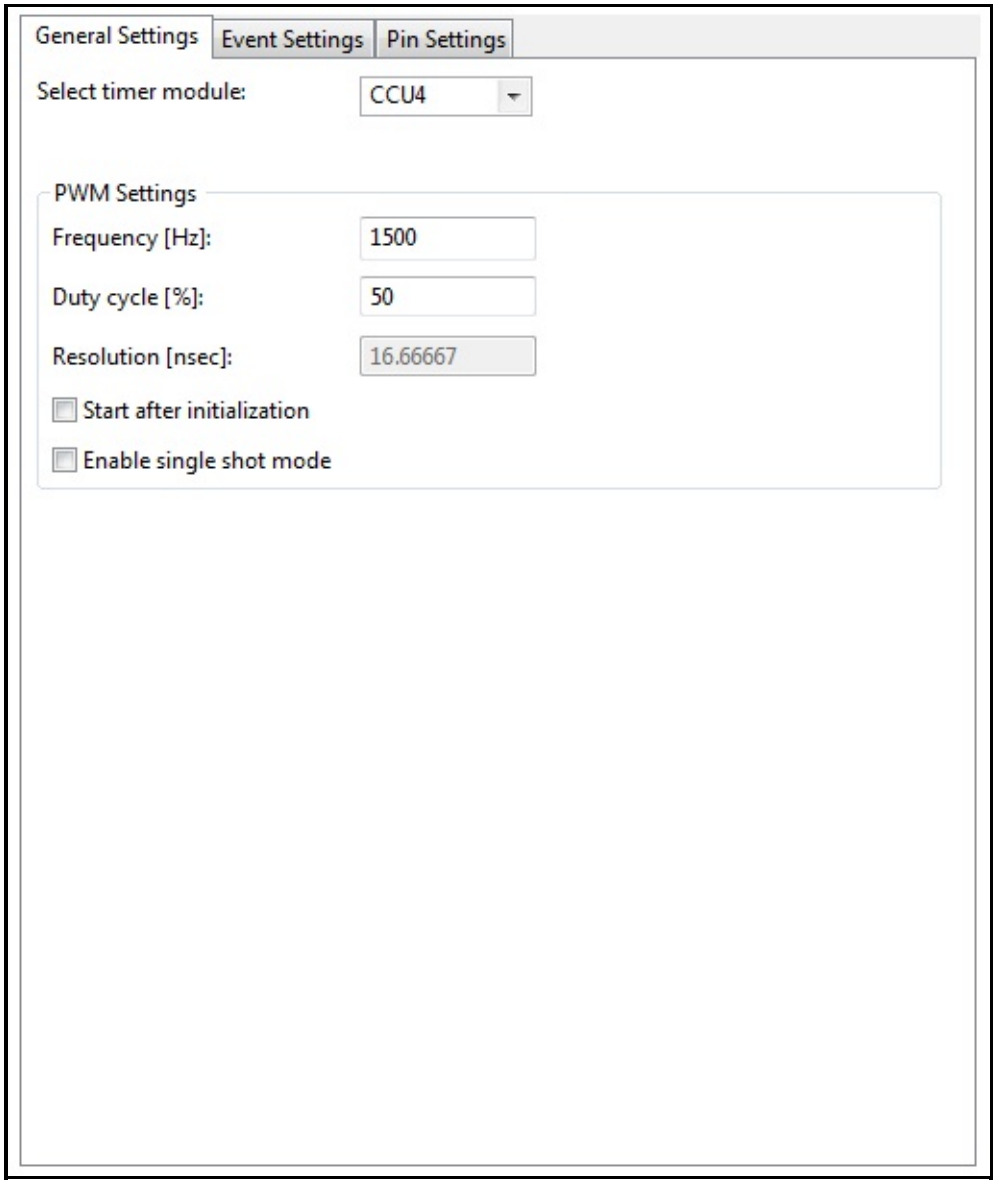


Figure 1: General Settings

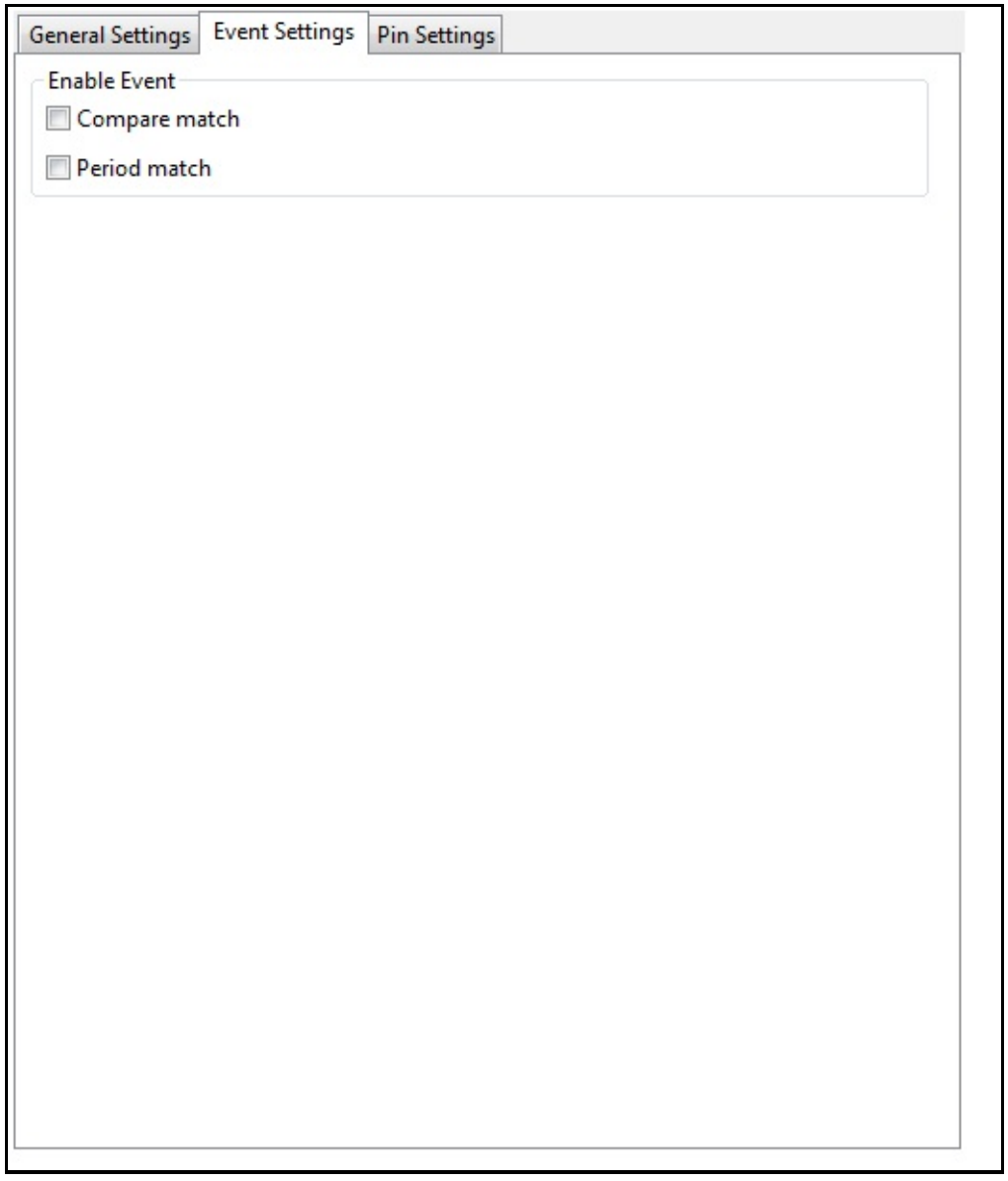


Figure 2: Event Settings

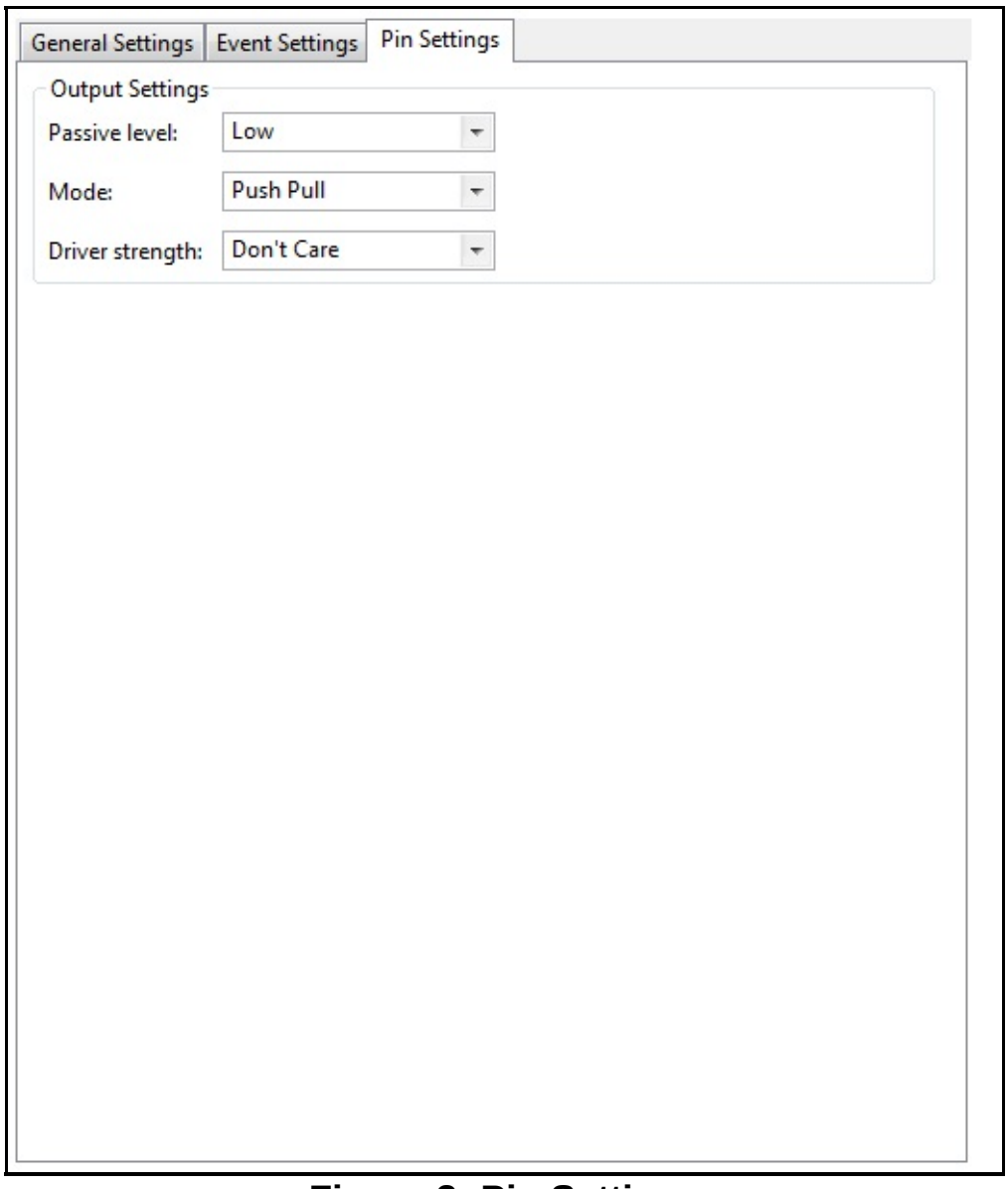


Figure 3: Pin Settings

PWM

Home

Enumerations

enum	**PWM_TIMER_SLICE** The type identifies the CCU4 or timer selected.
enum	**PWM_TIMER_STATUS** The type identifies the timer sta
enum	**PWM_INTERRUPT** { **PWM_INTERRUPT_PERIODM** 0U, **PWM_INTERRUPT_COMPARE** 2U } The type identifies the timer inte More...
enum	**PWM_OUTPUT_PASSIVE_LEV** **PWM_OUTPUT_PASSIVE_LEV** = 0, **PWM_OUTPUT_PASSIVE_LEV** } The type identifies the timer inte More...

enum	**PWM_STATUS** {
	PWM_STATUS_SUCCESS =
	PWM_STATUS_FAILURE,
	PWM_STATUS_UNINITIALIZE
	PWM_STATUS_RUNNING,
	PWM_STATUS_STOPPED
	}
	The type identifies App state. M

enum	**PWM_ERROR_CODES** { **PWM_OPER_NOT_ALLOWED**

	= 1, **PWM_INVALID_PARAM_E** The type identifies the App Erro More...
typedef enum **PWM_TIMER_SLICE**	**PWM_TIMER_SLICE_t** The type identifies the CCU4 or timer selected.
typedef enum **PWM_TIMER_STATUS**	**PWM_TIMER_STATUS_t** The type identifies the timer sta
typedef enum **PWM_INTERRUPT**	**PWM_INTERRUPT_t** The type identifies the timer inte
typedef enum **PWM_OUTPUT_PASSIVE_LEVEL**	**PWM_OUTPUT_PASSIVE_LEV** The type identifies the timer inte
typedef enum **PWM_STATUS**	**PWM_STATUS_t** The type identifies App state.
typedef enum **PWM_ERROR_CODES**	**PWM_ERROR_CODES_t** The type identifies the App Erro

Enumeration Type Documentation

enum PWM_ERROR_CODES

The type identifies the App Error Codes.

Enumerator:

PWM_OPER_NOT_ALLOWED_ERROR	if the current API operation is not possible in the given state
PWM_INVALID_PARAM_ERROR	the parameters passed to an API are invalid

Definition at line **157** of file **PWM.h**.

enum PWM_INTERRUPT

The type identifies the timer interrupts.

Enumerator:

PWM_INTERRUPT_PERIODMATCH	Period match interrupt while counting up
PWM_INTERRUPT_COMPAREMATCH	Compare match interrupt while counting up

Definition at line **126** of file **PWM.h**.

enum PWM_OUTPUT_PASSIVE_LEVEL

The type identifies the timer interrupts.

Enumerator:

PWM_OUTPUT_PASSIVE_LEVEL_LOW	Passive level = Low
PWM_OUTPUT_PASSIVE_LEVEL_HIGH	Passive level = High

Definition at line **135** of file **PWM.h**.

enum PWM_STATUS

The type identifies App state.

Enumerator:

PWM_STATUS_SUCCESS	APP is initialized as per selected parameters
PWM_STATUS_FAILURE	APP initialization function failure
PWM_STATUS_UNINITIALIZED	default state after power on reset
	CCU slice is running in PWM

PWM_STATUS_RUNNING	mode
PWM_STATUS_STOPPED	CCU slice timer is stopped

Definition at line **145** of file **PWM.h**.

PWM

Data structures

Data Structures

struct	**PWM_HandleType** Initialization parameters of the PWM App. More...
typedef struct **PWM_HandleType**	**PWM_t** Initialization parameters of the PWM App.

PWM

Home

Methods

DAVE_APP_VERSION_t	**PWM_GetAppVersion** (void) Get PWM APP version.
PWM_STATUS_t	**PWM_Init** (**PWM_t** *const handle_ptr) Initializes the PWM APP.
void	**PWM_Start** (**PWM_t** *const handle_ptr) Starts the PWM generation.
void	**PWM_Stop** (**PWM_t** *const handle_ptr) Stops the PWM generation.
PWM_STATUS_t	**PWM_SetFreq** (**PWM_t** *const handle_ptr, uint32_t pwm_freq_hz) Configures the PWM Frequency.
PWM_STATUS_t	**PWM_SetFreqAndDutyCycle** (**PWM_t** *const handle_ptr, uint32_t pwm_freq_hz, uint32_t duty_cycle) Configures the PWM Frequency and duty cycle.
void	**PWM_ClearEvent** (**PWM_t** *const handle_ptr, **PWM_INTERRUPT_t** pwm_interrupt) Clears the PWM related interrupt.
bool	**PWM_GetInterruptStatus** (**PWM_t** *const handle_ptr, **PWM_INTERRUPT_t** pwm_interrupt) Gets the corresponding interrupt status.
bool	**PWM_GetTimerStatus** (**PWM_t** *const handle_ptr) Gets the corresponding timer status.
	PWM_SetDutyCycle (**PWM_t** *const

PWM_STATUS_t	handle_ptr, uint32_t duty_cycle) Configure the PWM duty cycle.
void	**PWM_SetPassiveLevel** (**PWM_t** *const handle_ptr, **PWM_OUTPUT_PASSIVE_LEVEL_t** pwm_output_passive_level) Configure the passive level of the PWM output waveform.
PWM_STATUS_t	**PWM_SetPeriodMatchValue** (**PWM_t** *const handle_ptr, uint32_t period_match_value) Loads the required period match value.

Methods

Function Documentation

void PWM_ClearEvent (PWM_t *const handle_ptr,
PWM_INTERRUPT_t pwm_interrupt
)

Clears the PWM related interrupt.

Parameters:

handle_ptr	Constant pointer to the handle structure **PWM_t**
pwm_interrupt	Interrupt to acknowledge.

Returns:

void

Description:

Clears the CCUx related interrupt. When an interrupt occurs it must be acknowledged by clearing the respective flag in the hardware. Clears the interrupt type **PWM_INTERRUPT_t**.

```
   // Drag 2 INTERRUPT APPs into the project. In
the PWM APPs UI editor enable compare match event
 and period
   // match event. Goto the Signal connectivity w
indow and connect the event_compare_match and eve
nt_period_match to
   // the 2 different INTERRUPT APPs. Give the ha
ndler for the compare match interrupt as PWM_comp
are_match_interrupt
   // and the handler for the period match interr
upt as PWM_period_match_interrupt
   #include <DAVE.h>
   uint32_t period_count;
```

```
   void PWM_period_match_interrupt(void)
   {
     static uint32_t frequency = 1000; //start fr
equency 1Khz
     PWM_ClearEvent(&PWM_0, PWM_INTERRUPT_PERIODM
ATCH);
     period_count++;

     if(period_count == 35000) // wait to get 40K
 pwm cycles then increment the frequency by 2kHz
     {
       frequency += 2000;
       PWM_SetFreq(&PWM_0,frequency);
       period_count = 0;
       if(frequency > 30000000)
         frequency = 1000;
     }
   }

   void PWM_compare_match_interrupt(void)
   {
     PWM_ClearEvent(&PWM_0, PWM_INTERRUPT_COMPARE
MATCH);
   }

   int main(void)
   {

     DAVE_Init();

     PWM_Start(&PWM_0);
     while(1);
     return 0;
   }
```

Definition at line **943** of file **PWM.c**.

References **PWM_HandleType::ccu4_slice_ptr**, **PWM_HandleType::ccu8_slice_ptr**, and **PWM_HandleType::timer_type**.

DAVE_APP_VERSION_t PWM_GetAppVersion (void)

Get PWM APP version.

Returns:

DAVE_APP_VERSION_t APP version information (major, minor and patch number)

Description:

The function can be used to check application software compatibility with a specific version of the APP.

Example Usage:

```
 #include <DAVE.h>

 int main(void) {
   DAVE_STATUS_t init_status;
   DAVE_APP_VERSION_t version;

   // Initialize PWM APP:
   // PWM_Init() is called from within DAVE_Init(
).
   init_status = DAVE_Init();

   version = PWM_GetAppVersion();
   if (version.major != 1U) {
     // Probably, not the right version.
   }

   // More code here
   while(1) {
```

```
  }
  return (0);
}
```

Definition at line 734 of file PWM.c.

bool PWM_GetInterruptStatus (PWM_t *const handle_ptr, PWM_INTERRUPT_t pwm_interrup)

Gets the corresponding interrupt status.

Parameters:

handle_ptr	Constant pointer to the handle structure PWM_t
pwm_interrupt	Interrupt to get status.

Returns:

bool returns true if the **pwm_interrupt** has occured else returns false.

Description:

Returns the status of the corresponding interrupt. Reads the appropriate flag and would return true if the event was asserted.

```
  // Drag 1 INTERRUPT APP into the project. In t
he PWM APPs UI editor enable compare match event
and period
  // match event. Goto the Signal connectivity w
indow and connect the event_compare_match and eve
nt_period_match to
  // the same INTERRUPT APP and give the handler
 as PWM_compare_period_match_interrupt.
  #include <DAVE.h>
  uint32_t period_count;
```

```
   void PWM_compare_period_match_interrupt(void)
   {
     if(PWM_GetInterruptStatus(&PWM_0, PWM_INTERR
UPT_PERIODMATCH))
     {
         PWM_Stop(&PWM_0); // A single shot PWM g
enerated
     }
     PWM_ClearEvent(&PWM_0, PWM_INTERRUPT_COMPARE
MATCH);
   }

   int main(void)
   {

     DAVE_Init();

     PWM_Start(&PWM_0);
     while(1);
     return 0;
   }
```

Definition at line **917** of file **PWM.c**.

References **PWM_HandleType::ccu4_slice_ptr**, **PWM_HandleType::ccu8_slice_ptr**, and **PWM_HandleType::timer_type**.

bool PWM_GetTimerStatus (PWM_t *const handle_ptr)

Gets the corresponding timer status.

Parameters:

handle_ptr Constant pointer to the handle structure **PWM_t**

Returns:

bool
returns true if timer is running , false if the timer is idle.

Description:

Returns the state of the timer. Would return a false if the timer is not running. A call to this API results in invalid outputs if invoked before **PWM_Init()**.

```
#include <DAVE.h>

int main(void)
{
 DAVE_Init();

 if(PWM_GetTimerStatus(&PWM_0))
 {
   PWM_Stop(&PWM_0);
 }
 while(1);
 return 0;
}
```

Definition at line **891** of file **PWM.c**.

References **PWM_HandleType::ccu4_slice_ptr**, **PWM_HandleType::ccu8_slice_ptr**, and **PWM_HandleType::timer_type**.

PWM_STATUS_t PWM_Init (PWM_t *const handle_ptr)

Initializes the PWM APP.

Parameters:

handle_ptr Constant pointer to the handle structure **PWM_t**

Returns:

PWM_STATUS_t status of the initialization.

Description:

Initializes the PWM APP. This initializes the CCUx slice to compare mode of operation. Configures required events, GPIO pin as output. It will configure CCU4 or CCU8 slice registers with the selected PWM parameters. If PWM generation is set to start after initialization then after the CCUx related initialization is completed the PWM output will start.

```
  #include <DAVE.h>

  int main(void)
  {
   DAVE_Init(); //PWM_Init() is called by DAVE_In
it().
    while(1);
    return 0;
  }
```

Definition at line **747** of file **PWM.c**.

References **PWM_STATUS_FAILURE**, and **PWM_HandleType::timer_type**.

PWM_STATUS_t PWM_SetDutyCycle (PWM_t *const handle_ptr, uint32_t duty_cycle)

Configure the PWM duty cycle.

Parameters:

handle_ptr	Constant pointer to the handle structure **PWM_t**
duty_cycle	Duty cycle in percentage. The values are scaled by a factor of 100.

Range: [0(d) to 10000(d)] Where 1(d) represents 0.01% duty cycle
10000(d) represents 100% duty cycle

Returns:

Returns PWM_STATUS_FAILURE if the duty_cycle exceeds the valid range.

Description:

Configure the PWM duty cycle by changing the compare match values. The API would configure the duty cycle for the given frequency. The **duty_cycle** is a scaled parameter where a single value change would result in a 0.01% change.

Note:

This is a fixed point implementation. It is expected that there are resolution losses due to scaling in the API. At higher frequencies it may not be possible to achieve the required duty cycle due to the hardware limitation.

```
#include <DAVE.h>

int main(void)
{

   uint32_t Counter;

   DAVE_Init();

   for(Counter = 0;Counter <=0xFFFF;Counter++);

   // Change duty cycle value to 60.00%
   if(PWM_SetDutyCycle(&PWM_0,6000))
   {
     while(1);
   }
```

```
    // Wait to observe the waveform
    for(Counter = 0;Counter <=0xFFFF;Counter++);

    // Change duty cycle value to 54.26%
    if(PWM_SetDutyCycle(&PWM_0,5426))
    {
      while(1);
    }

    // Wait to observe the waveform
    for(Counter = 0;Counter <=0xFFFF;Counter++);

    // Change duty cycle value to an invalid duty
 100.26%
    if(PWM_SetDutyCycle(&PWM_0,10026))
    {
      while(1);
    }
    while(1);
    return 0;
}
```

Definition at line **816** of file **PWM.c**.

References **PWM_STATUS_FAILURE**, and **PWM_HandleType::timer_type**.

PWM_STATUS_t PWM_SetFreq (PWM_t *const handle_ptr, uint32_t pwm_freq_hz)

Configures the PWM Frequency.

Parameters:

handle_ptr	Constant pointer to the handle structure **PWM_t**
PwmFreqHz	Frequency value in Hz.

Range: [1(d) to 60000000(d)]

Returns:

PWM_STATUS_t Would return PWM_STATUS_FAILURE if the **PwmFreqHz** exceeded the maximum frequency achievable.

Description:

Configures the frequency of the PWM signal. The PWM duty cycle remains unchanged. Calculates the least possible resolution(Prescaler) of the CCUx Timer. Using this Prescaler it would calculate the value for the period register and the compare register. Adjusts the compare match value according to the frequency and the existing duty cycle.

```
  #include <DAVE.h>

  int main(void)
  {
     PWM_STATUS_t pwm_setfreq_status;
     uint32_t Counter;

     DAVE_Init();

     // Change duty cycle value to 20%.
         PWM_SetDutyCycle(&PWM_0, 2000);

    // Change PWM frequency to 100kHz
    pwm_setfreq_status = PWM_SetFreq(&PWM_0,10000
0);
    if(PWM_STATUS_FAILURE == pwm_setfreq_status)
    {
        // frequency couldn't be set
        while(1);
    }

    // Wait to observe the waveform
    for(Counter = 0;Counter <=0xFFFF;Counter++);
```

```
        // Change duty cycle value to 60%.
    PWM_SetDutyCycle(&PWM_0, 6000);


    while(1);
    return 0;
}
```

Definition at line **841** of file **PWM.c**.

References **PWM_STATUS_FAILURE**, and **PWM_HandleType::timer_type**.

PWM_STATUS_t PWM_SetFreqAndDutyCycle (PWM_t *const hand
uint32_t pwm
uint32_t duty
)

Configures the PWM Frequency and duty cycle.

Parameters:

handle_ptr	Constant pointer to the handle structure **PWM_t**
PwmFreqHz	Frequency value in Hz. Range: [1(d) to 60000000(d)]
duty_cycle	Duty cycle in percentage. The values are scaled by a factor of 100. Range: [0(d) to 10000(d)] Where 1(d) represents 0.01% duty cycle 10000(d) represents 100% duty cycle

Returns:

Return PWM_STATUS_FAILURE if the PwmFreqHz exceeded the maximum frequency achievable.

Description:

Configures the frequency and duty cycle together. Calculates the least possible resolution(Prescaler) of the CCUx Timer. Using this Prescaler it would calculate the value for the period register. Using the **duty_cycle** the compare register values would be calculated.

Note:

This is a fixed point implementation for duty cycle. It is expected that there are resolution losses due to scaling of the duty cycle in the API. At higher frequencies it may not be possible to achieve the required duty cycle due to the hardware limitation. If the input frequency to the API is very high (> 30MHz) due to integer divisions, the desired frequency might not be set.

```
  #include <DAVE.h>

  int main(void)
  {
     PWM_STATUS_t pwm_setfreq_status;
     uint32_t Counter;

     DAVE_Init();

         // Change duty cycle value to 90.90%.
    PWM_SetDutyCycle(&PWM_0, 9090);

         // Wait to observe the waveform
    for(Counter = 0;Counter <=0xFFFF;Counter++);

    // Change PWM frequency to 100kHz and duty to
 20%
    pwm_setfreq_status = PWM_SetFreqAndDutyCycle(
&PWM_0,100000,2000);
    if(PWM_STATUS_FAILURE == pwm_setfreq_status)
    {
         // frequency couldn't be set
```

```
        while(1);
    }

    while(1);
    return 0;
}
```

Definition at line **866** of file **PWM.c**.

References **PWM_STATUS_FAILURE**, and **PWM_HandleType::timer_type**.

void PWM_SetPassiveLevel (PWM_t *const h
PWM_OUTPUT_PASSIVE_LEVEL_t p
)

Configure the passive level of the PWM output waveform.

Parameters:

handle_ptr	Constant pointer to the handle structure **PWM_t**
pwm_output_passive_level	Passive level LOW or HIGH.

Returns:

void

Description:

Configure the passive level for the PWM signal. If ::PWM_OutputPassiveLevelType::PWM_OUTPUT_PASSIVE_LEV is selected the output of the PWM will be high before the compare match occurs after that it would be set to low. If ::PWM_OutputPassiveLevelType::PWM_OUTPUT_PASSIVE_LEV is selected the output of the PWM will be low before the compare match occurs after that it would be set to high.

```
   #include <DAVE.h>
```

```
   int main(void)
   {

    uint32_t Counter;

    DAVE_Init();

    for(Counter = 0;Counter <=0xFFFF;Counter++);

    PWM_SetPassiveLevel(&PWM_0, PWM_OUTPUT_PASSIV
E_LEVEL_HIGH);
    for(Counter = 0;Counter <=0xFFFF;Counter++);
    PWM_SetPassiveLevel(&PWM_0, PWM_OUTPUT_PASSIV
E_LEVEL_LOW);
    for(Counter = 0;Counter <=0xFFFF;Counter++);

    while(1);
    return 0;
   }
```

Definition at line **964** of file **PWM.c**.

References **PWM_HandleType::ccu4_kernel_ptr**, **PWM_HandleType::ccu4_slice_ptr**, **PWM_HandleType::ccu8_kernel_ptr**, **PWM_HandleType::ccu8_slice_ptr**, **PWM_HandleType::shadow_mask**, and **PWM_HandleType::timer_type**.

PWM_STATUS_t PWM_SetPeriodMatchValue (PWM_t *const handl
uint32_t perio
)

Loads the required period match value.

Parameters:

handle_ptr	Constant pointer to the handle structure **PWM_t**
period_match_value	value which needs to be loaded into the period register. Range: [0x0 to 0xFFFF]

Returns:

void

Description:

Configures or loads the required period value into the period register. This API would also calculate the compare register value (w.r.t **period_match_value**) to maintain the duty cycle.

```
   #include <DAVE.h>

   int main(void)
   {

    uint32_t Counter;

    DAVE_Init();

    PWM_SetFreq(&PWM_1,1U);// Set the required fr
equency as 1Hz

    for(Counter = 0;Counter <=0xFFFF;Counter++);

    PWM_SetPeriodMatchValue(&PWM_0,PWM_0.period_v
alue/2); // 2Hz
    for(Counter = 0;Counter <=0xFFFF;Counter++);
    PWM_SetPeriodMatchValue(&PWM_0,PWM_0.period_v
alue*2); //1Hz
    for(Counter = 0;Counter <=0xFFFF;Counter++);

    while(1);
    return 0;
```

```
  }
```

Definition at line **995** of file **PWM.c**.

References **PWM_HandleType::ccu4_kernel_ptr**, **PWM_HandleType::ccu4_slice_ptr**, **PWM_HandleType::ccu8_kernel_ptr**, **PWM_HandleType::ccu8_slice_ptr**, **PWM_HandleType::compare_value**, **PWM_HandleType::duty_cycle**, **PWM_HandleType::period_value**, **PWM_STATUS_FAILURE**, **PWM_STATUS_SUCCESS**, **PWM_HandleType::shadow_mask**, **PWM_HandleType::state**, and **PWM_HandleType::timer_type**.

void PWM_Start (PWM_t *const handle_ptr)

Starts the PWM generation.

Parameters:

handle_ptr Constant pointer to the handle structure **PWM_t**

Returns:

void

Description:

Start the selected PWM generation. It is needed to be called if "Start after initialization" is unchecked in the UI. Enables the clock for the CCUx slice and also starts the timer operation. Invoke this API only after initialization and when the timer is not running. If invoked outside these conditions, the API will have no effect.

```
#include <DAVE.h>

int main(void)
{
```

```
   DAVE_Init();
   //This needs to be called if "Start after init
ialization" is unchecked
   if(!PWM_GetTimerStatus(&PWM_0))
   {
     PWM_Start(&PWM_0);
   }
   while(1);
   return 0;
  }
```

Definition at line **773** of file **PWM.c**.

References **PWM_HandleType::timer_type**.

void PWM_Stop (PWM_t *const handle_ptr)

Stops the PWM generation.

Parameters:

handle_ptr Constant pointer to the handle structure **PWM_t**

Returns:

void

Description:

Stop the selected PWM generation. Disables the clock for the CCUx slice and also stops the timer operation. Invoke this API only after initialization and when the timer is running. If invoked outside these conditions, the API will have no effect.

```
   #include <DAVE.h>

  int main(void)
  {
   DAVE_Init();
```

```
    if(PWM_GetTimerStatus(&PWM_0))
    {
      PWM_Stop(&PWM_0);
    }
    while(1);
    return 0;
  }
```

Definition at line **794** of file **PWM.c**.

References **PWM_HandleType::timer_type**.

PWM

Home

Usage

Usage

The PWM APP is typically used just for a simple Pulse Width Modulation output generation.

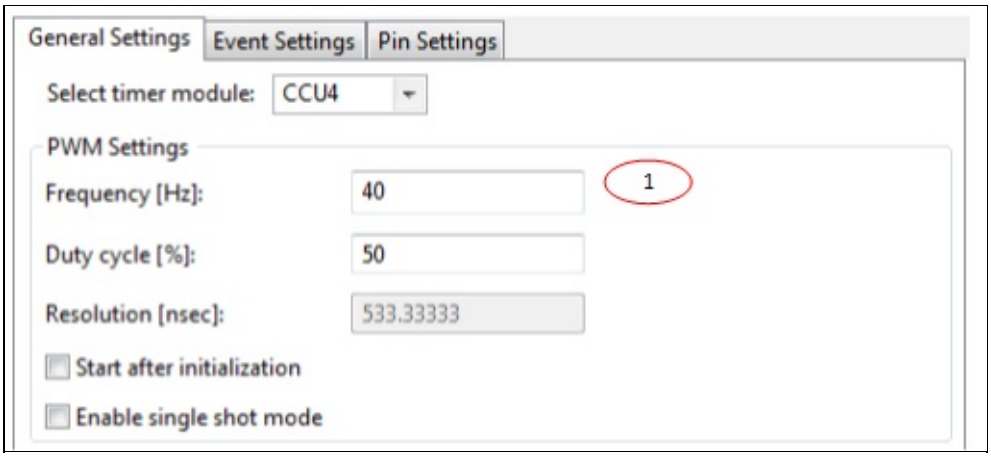
This example demonstrates the brightness control of a LED (Light Emitting Diode) using PWM duty cycle.

Instantiate the required APPs

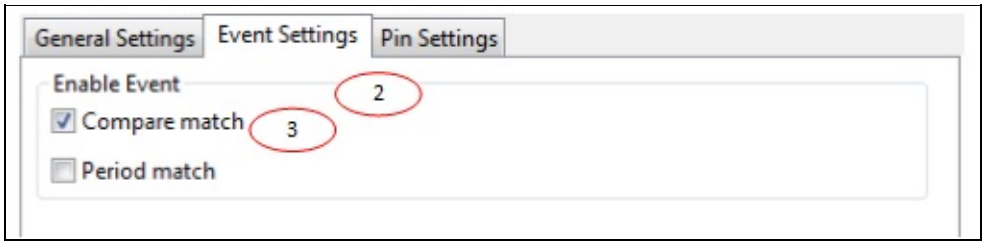
Drag an instance of PWM APP and INTERRUPT APP. Update the fields in the GUI of these APPs with the following configuration.

Configure the APPs

PWM APP:

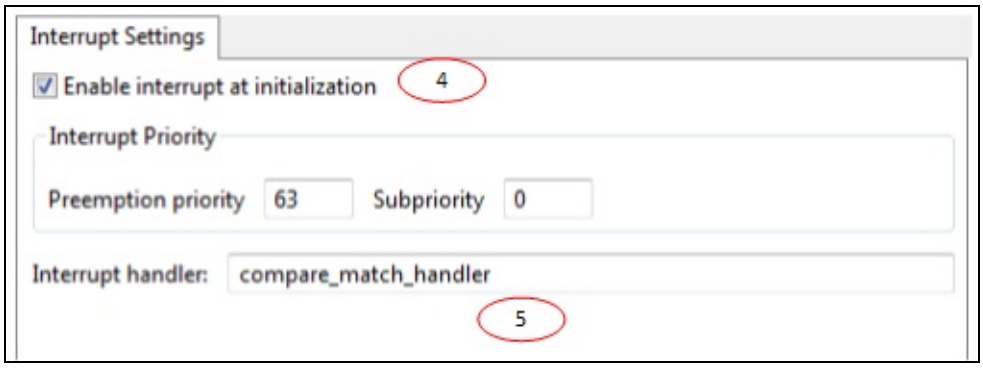


1. Set frequency as 40Hz.



2. Goto the Events Tab.
3. Enable compare match event.

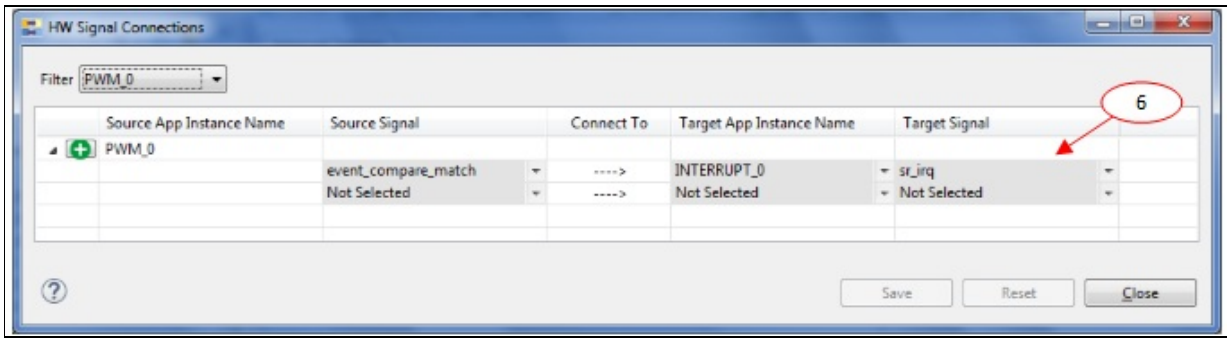
INTERRUPT APP:



4. Check the Enable interrupt at initialization.
5. Provide the interrupt handler as "compare_match_handler".

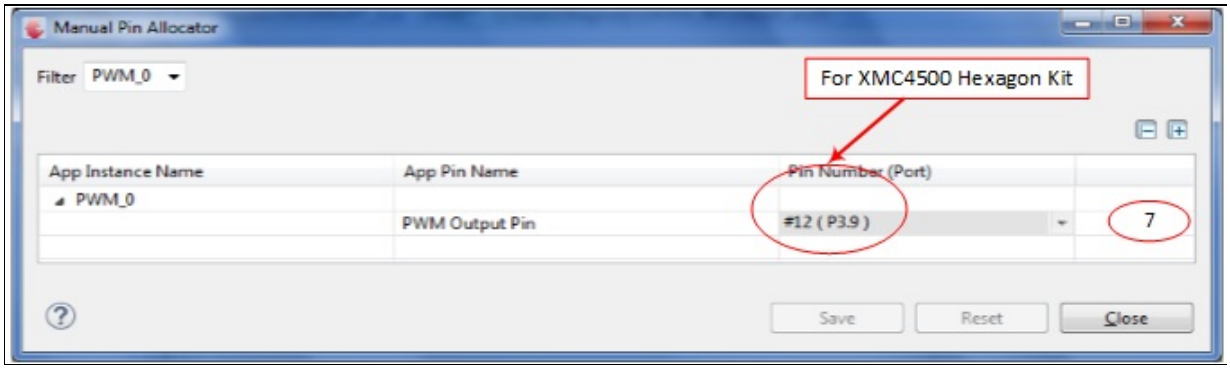
Signal Connection

Establish a HW signal connection between the PWM and the INTERRUPT APP to ensure PWM events generates interrupts.



6. Connect PWM_0/event_compare_match -> INTERRUPT_0/sr_irq to ensure assigning ISR node to the compare match event.

Manual pin allocation



7. Select the LED Pin present in the boot kit
   Note: The pin number is specific to the development board chosen to run this example. The pin shown in the image above may not be available on every XMC kit. Ensure that a proper pin is selected according to the board.

Generate code

Files are generated here: '<project_name>/Dave/Generated/' ('project_name' is the name chosen by the user during project creation). APP instance definitions and APIs are generated only after code generation.

- **Note:** Code must be explicitly generated for every change in the GUI configuration.
  Important: Any manual modification to APP specific files will be overwritten by a subsequent code generation operation.

Sample Application (main.c)

```
#include <DAVE.h>

// Add the following function in main.c
void compare_match_handler(void)
{
 static uint32_t duty = (uint32_t)100;
```

```
  static bool decrement_duty = (bool)false;

  if(decrement_duty != false)
  {
    //Decrement the duty cycle until it reaches 1%
    duty -= (uint32_t)100;
    // Once the duty has reached 1% flag status is
 changed to Increment
    if (duty <= (uint32_t)100)
     {
        decrement_duty = false;
     }
  }
  else
  {
    // Increment the duty cycle until it reaches 1
00%
    duty += (uint32_t)100;
    // Once the duty has reached 100% flag status
is changed to decrement
    if (duty >= (uint32_t)10000)
    {
     decrement_duty = true;
    }
  }
  // Sets the duty cycle of the PWM
  PWM_SetDutyCycle(&PWM_0,duty);

  // Clear the compare match interrupt.
  PWM_ClearEvent(&PWM_0,PWM_INTERRUPT_COMPAREMATCH
 );
 }


 int main(void)
 {
   DAVE_STATUS_t status;
```

```
  status = DAVE_Init();      /* Initialization of
 DAVE Apps  */

  if(status == DAVE_STATUS_FAILURE)
  {
    /* Placeholder for error handler code. The wh
ile loop below can be replaced with an user error
handler */
    XMC_DEBUG(("DAVE Apps initialization failed w
ith status %d\n", status));
    while(1U)
    {
    }
  }

  PWM_Start(&PWM_0);

  while(1U);

  return 1;
 }
```

Build and Run the Project

Observation

The LED brightness will gradually decrease and then increase. The brightness variation cycle will occur repeatedly.

PWM

Home

Release History

Release History

PWM

Data Structures

Here are the data structures with brief descriptions:

PWM_HandleType	Initialization parameters of the PWM App

PWM

Data Fields

PWM_HandleType Struct Reference

Data structures

Detailed Description

Initialization parameters of the PWM App.

Definition at line **184** of file **PWM.h**.

```
#include <PWM.h>
```

Data Fields

XMC_GPIO_PORT_t *const	**gpio_out_port**
const XMC_GPIO_CONFIG_t *const	**gpio_out_config**
uint32_t	**compare_value**
uint32_t	**period_value**
uint32_t	**duty_cycle**
uint32_t	**shadow_mask**
const **PWM_TIMER_SLICE_t**	**timer_type**
PWM_STATUS_t	**state**
const uint8_t	**gpio_out_pin**
const uint8_t	**kernel_number**
const uint8_t	**slice_number**
const bool	**start_control**
const bool	**period_match_enable**
const bool	**compare_match_enable**
GLOBAL_CCU4_t *const	**global_ccu4_handle**
GLOBAL_CCU8_t *const	**global_ccu8_handle**
XMC_CCU4_MODULE_t *const	**ccu4_kernel_ptr**
XMC_CCU8_MODULE_t *const	**ccu8_kernel_ptr**
XMC_CCU4_SLICE_t *const	**ccu4_slice_ptr**
XMC_CCU8_SLICE_t *const	**ccu8_slice_ptr**
const XMC_CCU4_SLICE_COMPARE_CONFIG_t *const	**ccu4_slice_config_ptr**
const XMC_CCU8_SLICE_COMPARE_CONFIG_t *const	**ccu8_slice_config_ptr**
const XMC_CCU4_SLICE_SR_ID_t	**ccu4_slice_period_mat**
const XMC_CCU8_SLICE_SR_ID_t	**ccu8_slice_period_mat**
const XMC_CCU4_SLICE_SR_ID_t	**ccu4_slice_compare_m**
const XMC_CCU8_SLICE_SR_ID_t	**ccu8_slice_compare_m**

Field Documentation

XMC_CCU4_MODULE_t* const PWM_HandleType::ccu4_kernel_ptr

Pointer to CCU4 kernel

Definition at line **199** of file **PWM.h**.

Referenced by **PWM_SetPassiveLevel()**, and **PWM_SetPeriodMatchValue()**.

const XMC_CCU4_SLICE_SR_ID_t PWM_HandleType::ccu4_slice_

SR node line for period match

Definition at line **255** of file **PWM.h**.

const XMC_CCU4_SLICE_COMPARE_CONFIG_t* const PWM_Hand

Pointer to CCU4 configuration handle

Definition at line **219** of file **PWM.h**.

const XMC_CCU4_SLICE_SR_ID_t PWM_HandleType::ccu4_slice_

SR node line for period match

Definition at line **245** of file **PWM.h**.

XMC_CCU4_SLICE_t* const PWM_HandleType::ccu4_slice_ptr

Pointer to CCU4 slice

Definition at line **209** of file **PWM.h**.

Referenced by **PWM_ClearEvent()**, **PWM_GetInterruptStatus()**, **PWM_GetTimerStatus()**, **PWM_SetPassiveLevel()**, and **PWM_SetPeriodMatchValue()**.

XMC_CCU8_MODULE_t* const PWM_HandleType::ccu8_kernel_pt

Pointer to CCU8 kernel

Definition at line **202** of file **PWM.h**.

Referenced by **PWM_SetPassiveLevel()**, and **PWM_SetPeriodMatchValue()**.

const XMC_CCU8_SLICE_SR_ID_t PWM_HandleType::ccu8_slice_

SR node line for period match

Definition at line **258** of file **PWM.h**.

const XMC_CCU8_SLICE_COMPARE_CONFIG_t* const PWM_Han

Pointer to CCU8 configuration handle

Definition at line **222** of file **PWM.h**.

const XMC_CCU8_SLICE_SR_ID_t PWM_HandleType::ccu8_slice_

SR node line for period match

Definition at line **248** of file **PWM.h**.

XMC_CCU8_SLICE_t* const PWM_HandleType::ccu8_slice_ptr

Pointer to CCU8 slice

Definition at line 212 of file PWM.h.

Referenced by PWM_ClearEvent(), PWM_GetInterruptStatus(), PWM_GetTimerStatus(), PWM_SetPassiveLevel(), and PWM_SetPeriodMatchValue().

const bool PWM_HandleType::compare_match_enable

Enable/Disable Compare match interrupt

Definition at line 272 of file PWM.h.

uint32_t PWM_HandleType::compare_value

Value that is pushed into the compare register

Definition at line 230 of file PWM.h.

Referenced by PWM_SetPeriodMatchValue().

uint32_t PWM_HandleType::duty_cycle

Value of duty cycle that is scaled with a factor 100

Definition at line 234 of file PWM.h.

Referenced by PWM_SetPeriodMatchValue().

GLOBAL_CCU4_t* const PWM_HandleType::global_ccu4_handle

GLOBAL_CCU4 App handle

Definition at line **189** of file **PWM.h**.

GLOBAL_CCU8_t* const PWM_HandleType::global_ccu8_handle

GLOBAL_CCU8 App handle

Definition at line **192** of file **PWM.h**.

const XMC_GPIO_CONFIG_t* const PWM_HandleType::gpio_out_c

Holds the pin configuration for the PWM output

Definition at line **228** of file **PWM.h**.

const uint8_t PWM_HandleType::gpio_out_pin

Holds the pin number for the PWM output

Definition at line **262** of file **PWM.h**.

XMC_GPIO_PORT_t* const PWM_HandleType::gpio_out_port

Holds the port number for the PWM output

Definition at line **226** of file **PWM.h**.

const uint8_t PWM_HandleType::kernel_number

Indicates the CCUx kernel number

Definition at line **264** of file **PWM.h**.

const bool PWM_HandleType::period_match_enable

Enable/Disable Period match interrupt

Definition at line **270** of file **PWM.h**.

uint32_t PWM_HandleType::period_value

Value that is pushed into the period register

Definition at line **232** of file **PWM.h**.

Referenced by **PWM_SetPeriodMatchValue()**.

uint32_t PWM_HandleType::shadow_mask

Holds the required shadow mask.

Definition at line **236** of file **PWM.h**.

Referenced by **PWM_SetPassiveLevel()**, and **PWM_SetPeriodMatchValue()**.

const uint8_t PWM_HandleType::slice_number

Indicates the CCUx slice number

Definition at line **266** of file **PWM.h**.

const bool PWM_HandleType::start_control

Enable/disable start of PWM after initialization

Definition at line **268** of file **PWM.h**.

PWM_STATUS_t PWM_HandleType::state

The current state of the PWM App instance

Definition at line **240** of file **PWM.h**.

Referenced by **PWM_SetPeriodMatchValue()**.

const PWM_TIMER_SLICE_t PWM_HandleType::timer_type

Type of CCU selected for the PWM generation

Definition at line **238** of file **PWM.h**.

Referenced by **PWM_ClearEvent()**, **PWM_GetInterruptStatus()**, **PWM_GetTimerStatus()**, **PWM_Init()**, **PWM_SetDutyCycle()**, **PWM_SetFreq()**, **PWM_SetFreqAndDutyCycle()**, **PWM_SetPassiveLevel()**, **PWM_SetPeriodMatchValue()**, **PWM_Start()**, and **PWM_Stop()**.

The documentation for this struct was generated from the following file:

- **PWM.h**

PWM

Data Structure Index

P

P

PWM_HandleType

P

PWM

Here is a list of all documented struct and union fields with links to the struct/union documentation for each field:

- ccu4_kernel_ptr : **PWM_HandleType**
- ccu4_slice_compare_match_node : **PWM_HandleType**
- ccu4_slice_config_ptr : **PWM_HandleType**
- ccu4_slice_period_match_node : **PWM_HandleType**
- ccu4_slice_ptr : **PWM_HandleType**
- ccu8_kernel_ptr : **PWM_HandleType**
- ccu8_slice_compare_match_node : **PWM_HandleType**
- ccu8_slice_config_ptr : **PWM_HandleType**
- ccu8_slice_period_match_node : **PWM_HandleType**
- ccu8_slice_ptr : **PWM_HandleType**
- compare_match_enable : **PWM_HandleType**
- compare_value : **PWM_HandleType**
- duty_cycle : **PWM_HandleType**
- global_ccu4_handle : **PWM_HandleType**
- global_ccu8_handle : **PWM_HandleType**
- gpio_out_config : **PWM_HandleType**
- gpio_out_pin : **PWM_HandleType**
- gpio_out_port : **PWM_HandleType**
- kernel_number : **PWM_HandleType**
- period_match_enable : **PWM_HandleType**
- period_value : **PWM_HandleType**
- shadow_mask : **PWM_HandleType**
- slice_number : **PWM_HandleType**
- start_control : **PWM_HandleType**
- state : **PWM_HandleType**
- timer_type : **PWM_HandleType**

PWM

- ccu4_kernel_ptr : **PWM_HandleType**
- ccu4_slice_compare_match_node : **PWM_HandleType**
- ccu4_slice_config_ptr : **PWM_HandleType**
- ccu4_slice_period_match_node : **PWM_HandleType**
- ccu4_slice_ptr : **PWM_HandleType**
- ccu8_kernel_ptr : **PWM_HandleType**
- ccu8_slice_compare_match_node : **PWM_HandleType**
- ccu8_slice_config_ptr : **PWM_HandleType**
- ccu8_slice_period_match_node : **PWM_HandleType**
- ccu8_slice_ptr : **PWM_HandleType**
- compare_match_enable : **PWM_HandleType**
- compare_value : **PWM_HandleType**
- duty_cycle : **PWM_HandleType**
- global_ccu4_handle : **PWM_HandleType**
- global_ccu8_handle : **PWM_HandleType**
- gpio_out_config : **PWM_HandleType**
- gpio_out_pin : **PWM_HandleType**
- gpio_out_port : **PWM_HandleType**
- kernel_number : **PWM_HandleType**
- period_match_enable : **PWM_HandleType**
- period_value : **PWM_HandleType**
- shadow_mask : **PWM_HandleType**
- slice_number : **PWM_HandleType**
- start_control : **PWM_HandleType**
- state : **PWM_HandleType**
- timer_type : **PWM_HandleType**

PWM

Home

File List | Globals

File List

Here is a list of all documented files with brief descriptions:

PWM.c [code]	
PWM.h [code]	

PWM

PWM.c File Reference

Detailed Description

Date:
2016-07-28

NOTE: This file is generated by DAVE. Any manual modification done to this file will be lost when the code is regenerated.

Definition in file **PWM.c**.

Functions

DAVE_APP_VERSION_t	**PWM_GetAppVersion** (void) Get PWM APP version.
PWM_STATUS_t	**PWM_Init** (**PWM_t** *const handle_ptr) Initializes the PWM APP.
void	**PWM_Start** (**PWM_t** *const handle_ptr) Starts the PWM generation.
void	**PWM_Stop** (**PWM_t** *const handle_ptr) Stops the PWM generation.
PWM_STATUS_t	**PWM_SetDutyCycle** (**PWM_t** *const handle_ptr, uint32_t duty_cycle) Configure the PWM duty cycle.
PWM_STATUS_t	**PWM_SetFreq** (**PWM_t** *const handle_ptr, uint32_t pwm_freq_hz) Configures the PWM Frequency.
PWM_STATUS_t	**PWM_SetFreqAndDutyCycle** (**PWM_t** *const handle_ptr, uint32_t pwm_freq_hz, uint32_t duty_cycle) Configures the PWM Frequency and duty cycle.
bool	**PWM_GetTimerStatus** (**PWM_t** *const handle_ptr) Gets the corresponding timer status.
bool	**PWM_GetInterruptStatus** (**PWM_t** *const handle_ptr, **PWM_INTERRUPT_t** pwm_interrupt) Gets the corresponding interrupt status.
void	**PWM_ClearEvent** (**PWM_t** *const handle_ptr, **PWM_INTERRUPT_t** pwm_interrupt) Clears the PWM related interrupt.
	PWM_SetPassiveLevel (**PWM_t** *const

void	handle_ptr, **PWM_OUTPUT_PASSIVE_LEVEL_t** pwm_output_passive_level) Configure the passive level of the PWM output waveform.
PWM_STATUS_t	**PWM_SetPeriodMatchValue** (**PWM_t** *const handle_ptr, uint32_t period_match_value) Loads the required period match value.

Function Documentation

void PWM_ClearEvent (PWM_t *const handle_ptr,
PWM_INTERRUPT_t pwm_interrupt
)

Clears the PWM related interrupt.

Parameters:

handle_ptr	Constant pointer to the handle structure **PWM_t**
pwm_interrupt	Interrupt to acknowledge.

Returns:

void

Description:

Clears the CCUx related interrupt. When an interrupt occurs it must be acknowledged by clearing the respective flag in the hardware. Clears the interrupt type **PWM_INTERRUPT_t**.

```
   // Drag 2 INTERRUPT APPs into the project. In
the PWM APPs UI editor enable compare match event
 and period
   // match event. Goto the Signal connectivity w
indow and connect the event_compare_match and eve
nt_period_match to
   // the 2 different INTERRUPT APPs. Give the ha
ndler for the compare match interrupt as PWM_comp
are_match_interrupt
   // and the handler for the period match interr
upt as PWM_period_match_interrupt
   #include <DAVE.h>
   uint32_t period_count;
```

```
   void PWM_period_match_interrupt(void)
   {
     static uint32_t frequency = 1000; //start fr
equency 1Khz
     PWM_ClearEvent(&PWM_0, PWM_INTERRUPT_PERIODM
ATCH);
     period_count++;

     if(period_count == 35000) // wait to get 40K
 pwm cycles then increment the frequency by 2kHz
     {
       frequency += 2000;
       PWM_SetFreq(&PWM_0,frequency);
       period_count = 0;
       if(frequency > 30000000)
         frequency = 1000;
     }
   }

   void PWM_compare_match_interrupt(void)
   {
     PWM_ClearEvent(&PWM_0, PWM_INTERRUPT_COMPARE
MATCH);
   }

   int main(void)
   {

     DAVE_Init();

     PWM_Start(&PWM_0);
     while(1);
     return 0;
   }
```

Definition at line **943** of file **PWM.c**.

References **PWM_HandleType::ccu4_slice_ptr**, **PWM_HandleType::ccu8_slice_ptr**, and **PWM_HandleType::timer_type**.

bool PWM_GetInterruptStatus (PWM_t *const handle_ptr, PWM_INTERRUPT_t pwm_interrup)

Gets the corresponding interrupt status.

Parameters:

handle_ptr	Constant pointer to the handle structure **PWM_t**
pwm_interrupt	Interrupt to get status.

Returns:

bool returns true if the **pwm_interrupt** has occured else returns false.

Description:

Returns the status of the corresponding interrupt. Reads the appropriate flag and would return true if the event was asserted.

```
   // Drag 1 INTERRUPT APP into the project. In t
he PWM APPs UI editor enable compare match event
and period
   // match event. Goto the Signal connectivity w
indow and connect the event_compare_match and eve
nt_period_match to
   // the same INTERRUPT APP and give the handler
 as PWM_compare_period_match_interrupt.
   #include <DAVE.h>
   uint32_t period_count;

   void PWM_compare_period_match_interrupt(void)
   {
```

```
     if(PWM_GetInterruptStatus(&PWM_0, PWM_INTERR
UPT_PERIODMATCH))
     {
         PWM_Stop(&PWM_0); // A single shot PWM g
enerated
     }
     PWM_ClearEvent(&PWM_0, PWM_INTERRUPT_COMPARE
MATCH);
   }

   int main(void)
   {

     DAVE_Init();

     PWM_Start(&PWM_0);
     while(1);
     return 0;
   }
```

Definition at line **917** of file **PWM.c**.

References **PWM_HandleType::ccu4_slice_ptr**, **PWM_HandleType::ccu8_slice_ptr**, and **PWM_HandleType::timer_type**.

bool PWM_GetTimerStatus (PWM_t *const handle_ptr)

Gets the corresponding timer status.

Parameters:

handle_ptr Constant pointer to the handle structure **PWM_t**

Returns:

bool
returns true if timer is running , false if the timer is idle.

Description:

Returns the state of the timer. Would return a false if the timer is not running. A call to this API results in invalid outputs if invoked before **PWM_Init()**.

```
#include <DAVE.h>

int main(void)
{
 DAVE_Init();

 if(PWM_GetTimerStatus(&PWM_0))
 {
   PWM_Stop(&PWM_0);
 }
 while(1);
 return 0;
}
```

Definition at line **891** of file **PWM.c**.

References **PWM_HandleType::ccu4_slice_ptr**, **PWM_HandleType::ccu8_slice_ptr**, and **PWM_HandleType::timer_type**.

PWM_STATUS_t PWM_Init (PWM_t *const handle_ptr)

Initializes the PWM APP.

Parameters:

handle_ptr Constant pointer to the handle structure **PWM_t**

Returns:

PWM_STATUS_t status of the initialization.

Description:

Initializes the PWM APP. This initializes the CCUx slice to compare mode of operation. Configures required events, GPIO pin as output. It will configure CCU4 or CCU8 slice registers with the selected PWM parameters. If PWM generation is set to start after initialization then after the CCUx related initialization is completed the PWM output will start.

```
  #include <DAVE.h>

  int main(void)
  {
   DAVE_Init(); //PWM_Init() is called by DAVE_In
it().
    while(1);
    return 0;
  }
```

Definition at line **747** of file **PWM.c**.

References **PWM_STATUS_FAILURE**, and **PWM_HandleType::timer_type**.

PWM_STATUS_t PWM_SetDutyCycle (PWM_t *const handle_ptr, uint32_t duty_cycle)

Configure the PWM duty cycle.

Parameters:

handle_ptr	Constant pointer to the handle structure **PWM_t**
duty_cycle	Duty cycle in percentage. The values are scaled by a factor of 100. Range: [0(d) to 10000(d)] Where 1(d) represents 0.01% duty cycle 10000(d) represents 100% duty cycle

Returns:

Returns PWM_STATUS_FAILURE if the duty_cycle exceeds the valid range.

Description:

Configure the PWM duty cycle by changing the compare match values. The API would configure the duty cycle for the given frequency. The **duty_cycle** is a scaled parameter where a single value change would result in a 0.01% change.

Note:

This is a fixed point implementation. It is expected that there are resolution losses due to scaling in the API. At higher frequencies it may not be possible to achieve the required duty cycle due to the hardware limitation.

```
#include <DAVE.h>

int main(void)
{

   uint32_t Counter;

   DAVE_Init();

   for(Counter = 0;Counter <=0xFFFF;Counter++);

   // Change duty cycle value to 60.00%
   if(PWM_SetDutyCycle(&PWM_0,6000))
   {
     while(1);
   }

   // Wait to observe the waveform
   for(Counter = 0;Counter <=0xFFFF;Counter++);

   // Change duty cycle value to 54.26%
```

```
    if(PWM_SetDutyCycle(&PWM_0,5426))
    {
      while(1);
    }

    // Wait to observe the waveform
    for(Counter = 0;Counter <=0xFFFF;Counter++);

    // Change duty cycle value to an invalid duty
 100.26%
    if(PWM_SetDutyCycle(&PWM_0,10026))
    {
      while(1);
    }
    while(1);
    return 0;
}
```

Definition at line **816** of file **PWM.c**.

References **PWM_STATUS_FAILURE**, and **PWM_HandleType::timer_type**.

PWM_STATUS_t PWM_SetFreq (PWM_t *const handle_ptr, uint32_t pwm_freq_hz)

Configures the PWM Frequency.

Parameters:

handle_ptr	Constant pointer to the handle structure **PWM_t**
PwmFreqHz	Frequency value in Hz. Range: [1(d) to 60000000(d)]

Returns:

PWM_STATUS_t Would return PWM_STATUS_FAILURE if the

PwmFreqHz exceeded the maximum frequency achievable.

Description:

Configures the frequency of the PWM signal. The PWM duty cycle remains unchanged. Calculates the least possible resolution(Prescaler) of the CCUx Timer. Using this Prescaler it would calculate the value for the period register and the compare register. Adjusts the compare match value according to the frequency and the existing duty cycle.

```
  #include <DAVE.h>

  int main(void)
  {
     PWM_STATUS_t pwm_setfreq_status;
     uint32_t Counter;

     DAVE_Init();

     // Change duty cycle value to 20%.
         PWM_SetDutyCycle(&PWM_0, 2000);

    // Change PWM frequency to 100kHz
    pwm_setfreq_status = PWM_SetFreq(&PWM_0,10000
0);
    if(PWM_STATUS_FAILURE == pwm_setfreq_status)
    {
        // frequency couldn't be set
        while(1);
    }

    // Wait to observe the waveform
    for(Counter = 0;Counter <=0xFFFF;Counter++);

        // Change duty cycle value to 60%.
    PWM_SetDutyCycle(&PWM_0, 6000);
```

```
    while(1);
    return 0;
  }
```

Definition at line **841** of file **PWM.c**.

References **PWM_STATUS_FAILURE**, and **PWM_HandleType::timer_type**.

PWM_STATUS_t PWM_SetFreqAndDutyCycle (PWM_t *const hand
uint32_t pwm
uint32_t duty
)

Configures the PWM Frequency and duty cycle.

Parameters:

handle_ptr	Constant pointer to the handle structure **PWM_t**
PwmFreqHz	Frequency value in Hz. Range: [1(d) to 60000000(d)]
duty_cycle	Duty cycle in percentage. The values are scaled by a factor of 100. Range: [0(d) to 10000(d)] Where 1(d) represents 0.01% duty cycle 10000(d) represents 100% duty cycle

Returns:

Return PWM_STATUS_FAILURE if the PwmFreqHz exceeded the maximum frequency achievable.

Description:

Configures the frequency and duty cycle together. Calculates the least possible resolution(Prescaler) of the CCUx Timer. Using this Prescaler it would calculate the value for the period

register. Using the **duty_cycle** the compare register values would be calculated.

Note:

This is a fixed point implementation for duty cycle. It is expected that there are resolution losses due to scaling of the duty cycle in the API. At higher frequencies it may not be possible to achieve the required duty cycle due to the hardware limitation. If the input frequency to the API is very high (> 30MHz) due to integer divisions, the desired frequency might not be set.

```
  #include <DAVE.h>

  int main(void)
  {
     PWM_STATUS_t pwm_setfreq_status;
     uint32_t Counter;

     DAVE_Init();

         // Change duty cycle value to 90.90%.
    PWM_SetDutyCycle(&PWM_0, 9090);

         // Wait to observe the waveform
    for(Counter = 0;Counter <=0xFFFF;Counter++);

    // Change PWM frequency to 100kHz and duty to
 20%
    pwm_setfreq_status = PWM_SetFreqAndDutyCycle(
&PWM_0,100000,2000);
    if(PWM_STATUS_FAILURE == pwm_setfreq_status)
    {
         // frequency couldn't be set
         while(1);
    }

    while(1);
```

```
    return 0;
}
```

Definition at line **866** of file **PWM.c**.

References **PWM_STATUS_FAILURE**, and **PWM_HandleType::timer_type**.

void PWM_SetPassiveLevel (PWM_t *const h
PWM_OUTPUT_PASSIVE_LEVEL_t p
)

Configure the passive level of the PWM output waveform.

Parameters:

handle_ptr	Constant pointer to the handle structure **PWM_t**
pwm_output_passive_level	Passive level LOW or HIGH.

Returns:

void

Description:

Configure the passive level for the PWM signal. If ::PWM_OutputPassiveLevelType::PWM_OUTPUT_PASSIVE_LEV is selected the output of the PWM will be high before the compare match occurs after that it would be set to low. If ::PWM_OutputPassiveLevelType::PWM_OUTPUT_PASSIVE_LEV is selected the output of the PWM will be low before the compare match occurs after that it would be set to high.

```
#include <DAVE.h>

int main(void)
{
```

```
    uint32_t Counter;

    DAVE_Init();

    for(Counter = 0;Counter <=0xFFFF;Counter++);

    PWM_SetPassiveLevel(&PWM_0, PWM_OUTPUT_PASSIV
E_LEVEL_HIGH);
    for(Counter = 0;Counter <=0xFFFF;Counter++);
    PWM_SetPassiveLevel(&PWM_0, PWM_OUTPUT_PASSIV
E_LEVEL_LOW);
    for(Counter = 0;Counter <=0xFFFF;Counter++);

    while(1);
    return 0;
   }
```

Definition at line **964** of file **PWM.c**.

References **PWM_HandleType::ccu4_kernel_ptr**, **PWM_HandleType::ccu4_slice_ptr**, **PWM_HandleType::ccu8_kernel_ptr**, **PWM_HandleType::ccu8_slice_ptr**, **PWM_HandleType::shadow_mask**, and **PWM_HandleType::timer_type**.

PWM_STATUS_t PWM_SetPeriodMatchValue (PWM_t *const handl
uint32_t perio
)

Loads the required period match value.

Parameters:

handle_ptr	Constant pointer to the handle structure **PWM_t**
period_match_value	value which needs to be loaded into the

period register. Range: [0x0 to 0xFFFF]

Returns:

void

Description:

Configures or loads the required period value into the period register. This API would also calculate the compare register value (w.r.t **period_match_value**) to maintain the duty cycle.

```
   #include <DAVE.h>

   int main(void)
   {

    uint32_t Counter;

    DAVE_Init();

    PWM_SetFreq(&PWM_1,1U);// Set the required fr
equency as 1Hz

    for(Counter = 0;Counter <=0xFFFF;Counter++);

    PWM_SetPeriodMatchValue(&PWM_0,PWM_0.period_v
alue/2); // 2Hz
    for(Counter = 0;Counter <=0xFFFF;Counter++);
    PWM_SetPeriodMatchValue(&PWM_0,PWM_0.period_v
alue*2); //1Hz
    for(Counter = 0;Counter <=0xFFFF;Counter++);

    while(1);
    return 0;
   }
```

Definition at line **995** of file **PWM.c**.

References **PWM_HandleType::ccu4_kernel_ptr**, **PWM_HandleType::ccu4_slice_ptr**, **PWM_HandleType::ccu8_kernel_ptr**, **PWM_HandleType::ccu8_slice_ptr**, **PWM_HandleType::compare_value**, **PWM_HandleType::duty_cycle**, **PWM_HandleType::period_value**, **PWM_STATUS_FAILURE**, **PWM_STATUS_SUCCESS**, **PWM_HandleType::shadow_mask**, **PWM_HandleType::state**, and **PWM_HandleType::timer_type**.

void PWM_Start (PWM_t *const handle_ptr)

Starts the PWM generation.

Parameters:

handle_ptr Constant pointer to the handle structure **PWM_t**

Returns:

void

Description:

Start the selected PWM generation. It is needed to be called if "Start after initialization" is unchecked in the UI. Enables the clock for the CCUx slice and also starts the timer operation. Invoke this API only after initialization and when the timer is not running. If invoked outside these conditions, the API will have no effect.

```
  #include <DAVE.h>

  int main(void)
  {
   DAVE_Init();
   //This needs to be called if "Start after init
ialization" is unchecked
   if(!PWM_GetTimerStatus(&PWM_0))
```

```
  {
    PWM_Start(&PWM_0);
  }
  while(1);
  return 0;
}
```

Definition at line **773** of file **PWM.c**.

References **PWM_HandleType::timer_type**.

void PWM_Stop (PWM_t *const handle_ptr)

Stops the PWM generation.

Parameters:

handle_ptr Constant pointer to the handle structure **PWM_t**

Returns:

void

Description:

Stop the selected PWM generation. Disables the clock for the CCUx slice and also stops the timer operation. Invoke this API only after initialization and when the timer is running. If invoked outside these conditions, the API will have no effect.

```
#include <DAVE.h>

int main(void)
{
 DAVE_Init();

 if(PWM_GetTimerStatus(&PWM_0))
 {
   PWM_Stop(&PWM_0);
```

```
    }
    while(1);
    return 0;
  }
```

Definition at line **794** of file **PWM.c**.

References **PWM_HandleType::timer_type**.

Go to the source code of this file.

PWM

PWM.h File Reference

Detailed Description

Date:
2016-07-28

NOTE: This file is generated by DAVE. Any manual modification done to this file will be lost when the code is regenerated.

Definition in file **PWM.h**.

Data Structures

struct	**PWM_HandleType** Initialization parameters of the PWM App. More...

Typedefs

typedef struct **PWM_HandleType**	**PWM_t** Initialization parameters of the PWM App.

Functions

DAVE_APP_VERSION_t	**PWM_GetAppVersion** (void) Get PWM APP version.
PWM_STATUS_t	**PWM_Init** (**PWM_t** *const hand Initializes the PWM APP.
void	**PWM_Start** (**PWM_t** *const har Starts the PWM generation.
void	**PWM_Stop** (**PWM_t** *const har Stops the PWM generation.
PWM_STATUS_t	**PWM_SetFreq** (**PWM_t** *const handle_ptr, uint32_t pwm_freq_ Configures the PWM Frequency
PWM_STATUS_t	**PWM_SetFreqAndDutyCycle** (*const handle_ptr, uint32_t pwm uint32_t duty_cycle) Configures the PWM Frequency cycle.
void	**PWM_ClearEvent** (**PWM_t** *co handle_ptr, **PWM_INTERRUPT** pwm_interrupt) Clears the PWM related interrup
bool	**PWM_GetInterruptStatus** (**PW** *const handle_ptr, **PWM_INTER** pwm_interrupt) Gets the corresponding interrup
bool	**PWM_GetTimerStatus** (**PWM_** handle_ptr) Gets the corresponding timer st
PWM_STATUS_t	**PWM_SetDutyCycle** (**PWM_t** * handle_ptr, uint32_t duty_cycle) Configure the PWM duty cycle.
	PWM_SetPassiveLevel (**PWM**

void	handle_ptr, **PWM_OUTPUT_PASSIVE_LE** pwm_output_passive_level) Configure the passive level of t output waveform.
PWM_STATUS_t	**PWM_SetPeriodMatchValue** (* *const handle_ptr, uint32_t period_match_value) Loads the required period matc
enum	**PWM_TIMER_SLICE** The type identifies the CCU4 or timer selected.
enum	**PWM_TIMER_STATUS** The type identifies the timer sta
enum	**PWM_INTERRUPT** { **PWM_INTERRUPT_PERIODM** 0U, **PWM_INTERRUPT_COMPARE** 2U } The type identifies the timer inte More...
enum	**PWM_OUTPUT_PASSIVE_LE** **PWM_OUTPUT_PASSIVE_LE** = 0, **PWM_OUTPUT_PASSIVE_LE** } The type identifies the timer inte More...
enum	**PWM_STATUS** { **PWM_STATUS_SUCCESS** = **PWM_STATUS_FAILURE**, **PWM_STATUS_UNINITIALIZE** **PWM_STATUS_RUNNING**, **PWM_STATUS_STOPPED** } The type identifies App state. M

enum	**PWM_ERROR_CODES** { **PWM_OPER_NOT_ALLOWED** = 1, **PWM_INVALID_PARAM_E** The type identifies the App Erro More...
typedef enum **PWM_TIMER_SLICE**	**PWM_TIMER_SLICE_t** The type identifies the CCU4 or timer selected.
typedef enum **PWM_TIMER_STATUS**	**PWM_TIMER_STATUS_t** The type identifies the timer sta
typedef enum **PWM_INTERRUPT**	**PWM_INTERRUPT_t** The type identifies the timer inte
typedef enum **PWM_OUTPUT_PASSIVE_LEVEL**	**PWM_OUTPUT_PASSIVE_LEV** The type identifies the timer inte
typedef enum **PWM_STATUS**	**PWM_STATUS_t** The type identifies App state.
typedef enum **PWM_ERROR_CODES**	**PWM_ERROR_CODES_t** The type identifies the App Erro

Go to the source code of this file.

PWM

Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

- p -

- PWM_STATUS : **PWM.h**
- PWM_STATUS_FAILURE : **PWM.h**
- PWM_STATUS_RUNNING : **PWM.h**
- PWM_STATUS_STOPPED : **PWM.h**
- PWM_STATUS_SUCCESS : **PWM.h**
- PWM_STATUS_t : **PWM.h**
- PWM_STATUS_UNINITIALIZED : **PWM.h**
- PWM_Stop() : **PWM.h** , **PWM.c**
- PWM_t : **PWM.h**
- PWM_TIMER_SLICE : **PWM.h**
- PWM_TIMER_SLICE_t : **PWM.h**
- PWM_TIMER_STATUS : **PWM.h**
- PWM_TIMER_STATUS_t : **PWM.h**

PWM

- PWM_ClearEvent() : **PWM.c** , **PWM.h**
- PWM_GetAppVersion() : **PWM.h** , **PWM.c**
- PWM_GetInterruptStatus() : **PWM.c** , **PWM.h**
- PWM_GetTimerStatus() : **PWM.h** , **PWM.c**
- PWM_Init() : **PWM.c** , **PWM.h**
- PWM_SetDutyCycle() : **PWM.c** , **PWM.h**
- PWM_SetFreq() : **PWM.c** , **PWM.h**
- PWM_SetFreqAndDutyCycle() : **PWM.c** , **PWM.h**
- PWM_SetPassiveLevel() : **PWM.c** , **PWM.h**
- PWM_SetPeriodMatchValue() : **PWM.c** , **PWM.h**
- PWM_Start() : **PWM.c** , **PWM.h**
- PWM_Stop() : **PWM.h** , **PWM.c**

PWM

- PWM_ERROR_CODES_t : **PWM.h**
- PWM_INTERRUPT_t : **PWM.h**
- PWM_OUTPUT_PASSIVE_LEVEL_t : **PWM.h**
- PWM_STATUS_t : **PWM.h**
- PWM_t : **PWM.h**
- PWM_TIMER_SLICE_t : **PWM.h**
- PWM_TIMER_STATUS_t : **PWM.h**

PWM

- PWM_ERROR_CODES : **PWM.h**
- PWM_INTERRUPT : **PWM.h**
- PWM_OUTPUT_PASSIVE_LEVEL : **PWM.h**
- PWM_STATUS : **PWM.h**
- PWM_TIMER_SLICE : **PWM.h**
- PWM_TIMER_STATUS : **PWM.h**

PWM

- PWM_INTERRUPT_COMPAREMATCH : **PWM.h**
- PWM_INTERRUPT_PERIODMATCH : **PWM.h**
- PWM_INVALID_PARAM_ERROR : **PWM.h**
- PWM_OPER_NOT_ALLOWED_ERROR : **PWM.h**
- PWM_OUTPUT_PASSIVE_LEVEL_HIGH : **PWM.h**
- PWM_OUTPUT_PASSIVE_LEVEL_LOW : **PWM.h**
- PWM_STATUS_FAILURE : **PWM.h**
- PWM_STATUS_RUNNING : **PWM.h**
- PWM_STATUS_STOPPED : **PWM.h**
- PWM_STATUS_SUCCESS : **PWM.h**
- PWM_STATUS_UNINITIALIZED : **PWM.h**

PWM

PWM.h

Go to the documentation of this file.

```
00001
00076 #ifndef PWM_H
00077 #define PWM_H
00078
00079
00080 /*****************************************************
*****************************************************
*************************
00081  * HEADER FILES
00082  *****************************************************
*****************************************************
*************************/
00083 #include <xmc_gpio.h>
00084 #include "pwm_conf.h"
00085 #include <DAVE_common.h>
00086
00087 /*****************************************************
*****************************************************
*************************
00088  * MACROS
00089  *****************************************************
*****************************************************
*************************/
00090 #if (!((XMC_LIB_MAJOR_VERSION == 2U) && \
00091        (XMC_LIB_MINOR_VERSION >= 0U) && \
00092        (XMC_LIB_PATCH_VERSION >= 0U)))
00093 #error "PWM requires XMC Peripheral Library
```

```
v2.0.0 or higher"
00094 #endif
00095
00096 #define PWM_MAX_TIMER_COUNT  (65535U)
00098  /*****************************************
*****************************************
*************************
00099  * ENUMS
00100  *****************************************
*****************************************
*************************/
00101
00108 typedef enum PWM_TIMER_SLICE
00109 {
00110   PWM_TIMER_SLICE_CCU4 = 0U,
00111   PWM_TIMER_SLICE_CCU8
00112 } PWM_TIMER_SLICE_t;
00113
00117 typedef enum PWM_TIMER_STATUS
00118 {
00119   PWM_TIMER_IDLE = 0U,
00120   PWM_TIMER_RUNNING
00121 } PWM_TIMER_STATUS_t;
00122
00126 typedef enum PWM_INTERRUPT
00127 {
00128   PWM_INTERRUPT_PERIODMATCH      = 0U,

00129   PWM_INTERRUPT_COMPAREMATCH     = 2U

00130 } PWM_INTERRUPT_t;
00131
00135 typedef enum PWM_OUTPUT_PASSIVE_LEVEL
00136 {
00137   PWM_OUTPUT_PASSIVE_LEVEL_LOW  = 0,
00138   PWM_OUTPUT_PASSIVE_LEVEL_HIGH    ,
00139   PWM_OUTPUT_PASSIVE_LEVEL_MAX
```

```
00140 } PWM_OUTPUT_PASSIVE_LEVEL_t;
00141
00145 typedef enum PWM_STATUS
00146 {
00147   PWM_STATUS_SUCCESS = 0,
00148   PWM_STATUS_FAILURE,
00149   PWM_STATUS_UNINITIALIZED,
00150   PWM_STATUS_RUNNING,
00151   PWM_STATUS_STOPPED,
00152 } PWM_STATUS_t;
00153
00157 typedef enum PWM_ERROR_CODES
00158 {
00159   PWM_OPER_NOT_ALLOWED_ERROR = 1,
00160   PWM_INVALID_PARAM_ERROR
00161 } PWM_ERROR_CODES_t;
00162
00167 /**********************************************
**********************************************
*************************
00168 * DATA STRUCTURES
00169 **********************************************
**********************************************
*************************/
00174 /*Anonymous structure/union guard start*/
00175 #if defined(__CC_ARM)
00176   #pragma push
00177   #pragma anon_unions
00178 #elif defined(__TASKING__)
00179   #pragma warning 586
00180 #endif
00181
00184 typedef struct PWM_HandleType
00185 {
00186   union
00187   {
00188 #ifdef  PWM_SLICE_USED_CCU4
```

```
00189         GLOBAL_CCU4_t *const global_ccu4_han
dle;
00190 #endif
00191 #ifdef  PWM_SLICE_USED_CCU8
00192         GLOBAL_CCU8_t *const global_ccu8_han
dle;
00193 #endif
00194   };
00195
00196   union
00197   {
00198 #ifdef  PWM_SLICE_USED_CCU4
00199   XMC_CCU4_MODULE_t *const ccu4_kernel_ptr;

00200 #endif
00201 #ifdef  PWM_SLICE_USED_CCU8
00202   XMC_CCU8_MODULE_t *const ccu8_kernel_ptr;

00203 #endif
00204   };
00205
00206   union
00207   {
00208 #ifdef  PWM_SLICE_USED_CCU4
00209   XMC_CCU4_SLICE_t *const ccu4_slice_ptr;
00210 #endif
00211 #ifdef  PWM_SLICE_USED_CCU8
00212   XMC_CCU8_SLICE_t *const ccu8_slice_ptr;
00213 #endif
00214   };
00215
00216   union
00217   {
00218 #ifdef  PWM_SLICE_USED_CCU4
00219   const XMC_CCU4_SLICE_COMPARE_CONFIG_t *con
st ccu4_slice_config_ptr;
00220 #endif
```

```
00221 #ifdef  PWM_SLICE_USED_CCU8
00222   const XMC_CCU8_SLICE_COMPARE_CONFIG_t *con
st ccu8_slice_config_ptr;
00223 #endif
00224   };
00225
00226   XMC_GPIO_PORT_t                *const  gp
io_out_port;
00228   const XMC_GPIO_CONFIG_t *const  gpio_out_c
onfig;
00230   uint32_t                compare_value;
00232   uint32_t                period_value;
00234   uint32_t                duty_cycle;
00236   uint32_t                shadow_mask;
00238   const PWM_TIMER_SLICE_t  timer_type;
00240   PWM_STATUS_t            state;
00242   union
00243     {
00244   #ifdef  PWM_SLICE_USED_CCU4
00245     const XMC_CCU4_SLICE_SR_ID_t          c
cu4_slice_period_match_node;
00246   #endif
00247   #ifdef  PWM_SLICE_USED_CCU8
00248     const XMC_CCU8_SLICE_SR_ID_t          c
cu8_slice_period_match_node;
00249   #endif
00250     };
00251
00252     union
00253     {
00254   #ifdef  PWM_SLICE_USED_CCU4
00255     const XMC_CCU4_SLICE_SR_ID_t          c
cu4_slice_compare_match_node;
00256   #endif
00257   #ifdef  PWM_SLICE_USED_CCU8
00258     const XMC_CCU8_SLICE_SR_ID_t          c
cu8_slice_compare_match_node;
```

```
00259   #endif
00260     };
00261
00262   const uint8_t gpio_out_pin;
00264   const uint8_t kernel_number;
00266   const uint8_t slice_number;
00268   const bool    start_control;
00270   const bool    period_match_enable;
00272   const bool    compare_match_enable;
00274 } PWM_t;
00275
00279 /*Anonymous structure/union guard end*/
00280 #if defined(__CC_ARM)
00281   #pragma pop
00282 #elif defined(__TASKING__)
00283   #pragma warning restore
00284 #endif
00285
00289 #ifdef __cplusplus
00290 extern "C" {
00291 #endif
00292 /**********************************************
**********************************************
*************************
00293  * API Prototypes
00294  **********************************************
**********************************************
*************************/
00329 DAVE_APP_VERSION_t PWM_GetAppVersion(void);
00330
00354 PWM_STATUS_t PWM_Init(PWM_t *const handle_pt
r);
00355
00383 void PWM_Start(PWM_t *const handle_ptr);
00384
00411 void PWM_Stop(PWM_t *const handle_ptr);
00412
```

```
00459 PWM_STATUS_t PWM_SetFreq(PWM_t *const handle
_ptr, uint32_t pwm_freq_hz);
00460
00515 PWM_STATUS_t PWM_SetFreqAndDutyCycle(PWM_t *
const handle_ptr, uint32_t pwm_freq_hz, uint32_t d
uty_cycle);
00516
00569 void PWM_ClearEvent(PWM_t *const handle_ptr,
PWM_INTERRUPT_t pwm_interrupt);
00570
00609 bool PWM_GetInterruptStatus(PWM_t *const han
dle_ptr, PWM_INTERRUPT_t pwm_interrupt);
00610
00636 bool PWM_GetTimerStatus(PWM_t *const handle_
ptr);
00637
00697 PWM_STATUS_t PWM_SetDutyCycle(PWM_t *const h
andle_ptr, uint32_t duty_cycle);
00698
00699
00734 void PWM_SetPassiveLevel(PWM_t *const handle
_ptr, PWM_OUTPUT_PASSIVE_LEVEL_t pwm_output_passiv
e_level);
00735
00771 PWM_STATUS_t PWM_SetPeriodMatchValue(PWM_t *
const handle_ptr, uint32_t period_match_value);
00772
00773 #include "PWM_Extern.h"
00774
00778 #ifdef __cplusplus
00779 }
00780 #endif
00781
00782 #endif /* PWM_H_ */
```

PWM

PWM.c

Go to the documentation of this file.

```
00001
00076 /*****************************************
*****************************************
*************************
00077  * HEADER FILES
00078  *****************************************
*****************************************
*************************/
00079 #include "pwm.h"
00080
00081 /*****************************************
*****************************************
*************************
00082  * MACROS
00083  *****************************************
*****************************************
*************************/
00084 #define PWM_MAX_DUTY_CYCLE ((uint32_t)10000)
00085 #define PWM_DUTY_CYCLE_SCALE ((uint32_t)100)
00086 #define PWM_MAX_PRESCALER ((uint32_t)15)
00087 #define PWM_MAX_PERIOD_VALUE ((uint32_t) 655
35)
00088
00089 /*****************************************
*****************************************
*************************
00090  * LOCAL ROUTINES
```

```
00091  ***********************************************
***********************************************
************************/
00092 #ifdef PWM_SLICE_USED_CCU4
00093 /*Initialize the App and XMC_CCU4 slice. */
00094 PWM_STATUS_t PWM_lCCU4_Init(PWM_t *const han
dle_ptr);
00095
00096 /*Initialize interrupts*/
00097 void PWM_lCCU4_ConfigInterrupts(PWM_t *const
 handle_ptr);
00098
00099 /*Starts the CCU4 slice. */
00100 void PWM_lCCU4_Start(PWM_t *const handle_ptr
);
00101
00102 /*Stops the CCU4 slice. */
00103 void PWM_lCCU4_Stop(PWM_t *const handle_ptr)
;
00104
00105 /*Sets the duty cycle for CCU4 slice. */
00106 PWM_STATUS_t PWM_lCCU4_SetDutyCycle(PWM_t *c
onst handle_ptr, uint32_t duty_cycle);
00107
00108 /*Sets the frequency for CCU4 slice. */
00109 PWM_STATUS_t PWM_lCCU4_SetFreq(PWM_t *const
handle_ptr, uint32_t pwm_freq_hz);
00110
00111 /*Sets the frequency and duty cycle for CCU4
 slice. */
00112 PWM_STATUS_t PWM_lCCU4_SetFreqAndDutyCycle(P
WM_t *const handle_ptr, uint32_t pwm_freq_hz, uint
32_t duty_cycle);
00113 #endif
00114
00115 #ifdef PWM_SLICE_USED_CCU8
00116 /*Initialize the App and XMC_CCU8 slice. */
```

```
00117 PWM_STATUS_t PWM_lCCU8_Init(PWM_t *const han
dle_ptr);
00118
00119 /*Initialize interrupts*/
00120 void PWM_lCCU8_ConfigInterrupts(PWM_t *const
 handle_ptr);
00121
00122 /*Starts the CCU8 slice. */
00123 void PWM_lCCU8_Start(PWM_t *const handle_ptr
);
00124
00125 /*Stops the CCU8 slice. */
00126 void PWM_lCCU8_Stop(PWM_t *const handle_ptr)
;
00127
00128 /*Sets the duty cycle for CCU8 slice. */
00129 PWM_STATUS_t PWM_lCCU8_SetDutyCycle(PWM_t *c
onst handle_ptr, uint32_t duty_cycle);
00130
00131 /*Sets the frequency for CCU8 slice. */
00132 PWM_STATUS_t PWM_lCCU8_SetFreq(PWM_t *const
handle_ptr, uint32_t pwm_freq_hz);
00133
00134 /*Sets the frequency and duty cycle for CCU8
 slice. */
00135 PWM_STATUS_t PWM_lCCU8_SetFreqAndDutyCycle(P
WM_t *const handle_ptr, uint32_t pwm_freq_hz, uint
32_t duty_cycle);
00136 #endif
00137
00138 #ifdef PWM_SLICE_USED_CCU4
00139
00140 /*Initialize the APP and CCU4 slice. */
00141 PWM_STATUS_t PWM_lCCU4_Init(PWM_t *const han
dle_ptr)
00142 {
00143   PWM_STATUS_t status = PWM_STATUS_FAILURE;
```

```
00144
00145   XMC_ASSERT("PWM_lCCU4_Init:Invalid handle_
ptr" , (handle_ptr != NULL))
00146
00147   if (PWM_STATUS_UNINITIALIZED == handle_ptr
->state)
00148   {
00149     /* Initialize consumed Apps */
00150     status = (PWM_STATUS_t)GLOBAL_CCU4_Init(
handle_ptr->global_ccu4_handle);
00151
00152     /*Initialize CCU4 slice */
00153     if (PWM_STATUS_SUCCESS == status)/*check
 GLOBAL_CCU4_Init status*/
00154     {
00155       XMC_DEBUG("PWM_lCCU4_Init:Initilizing
Slice")
00156       XMC_CCU4_SLICE_CompareInit(handle_ptr-
>ccu4_slice_ptr, handle_ptr->ccu4_slice_config_ptr
);
00157
00158       /* Set the period and compare register
 values */
00159       XMC_CCU4_SLICE_SetTimerPeriodMatch(han
dle_ptr->ccu4_slice_ptr,
00160
 (uint16_t)handle_ptr->period_value);
00161
00162       XMC_CCU4_SLICE_SetTimerCompareMatch(ha
ndle_ptr->ccu4_slice_ptr,
00163
  (uint16_t)handle_ptr->compare_value);
00164
00165       XMC_CCU4_EnableShadowTransfer(handle_p
tr->ccu4_kernel_ptr, handle_ptr->shadow_mask);
00166
00167       /* Initialize interrupts */
```

```
00168       PWM_lCCU4_ConfigInterrupts(handle_ptr)
;
00169
00170       XMC_GPIO_Init(handle_ptr->gpio_out_por
t,handle_ptr->gpio_out_pin,
00171                                 handle_ptr->gpio_out
_config);
00172
00173       handle_ptr->state = PWM_STATUS_SUCCESS
;
00174
00175       /* Start the PWM generation if start a
t initialization is enabled */
00176       if ((bool) true == handle_ptr->start_c
ontrol)
00177       {
00178         PWM_Start(handle_ptr);
00179       }
00180       status = PWM_STATUS_SUCCESS;
00181     }
00182     else
00183     {
00184       handle_ptr->state = PWM_STATUS_UNINITI
ALIZED;
00185     }
00186
00187   }
00188   return (status);
00189 } /* end of PWM_lCCU4_Init() api */
00190
00191 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/
00192 /* Initialize interrupts */
00193 void PWM_lCCU4_ConfigInterrupts(PWM_t *const
 handle_ptr)
00194 {
```

```
00195   if ((bool) true == handle_ptr->period_matc
h_enable)
00196   {
00197     XMC_DEBUG("PWM_lCCU4_ConfigInterrupts:pe
riod match enable")
00198     XMC_CCU4_SLICE_EnableEvent(handle_ptr->c
cu4_slice_ptr, XMC_CCU4_SLICE_IRQ_ID_PERIOD_MATCH)
;
00199
00200     /* Bind event to Service Request Node to
 period match event*/
00201     XMC_CCU4_SLICE_SetInterruptNode(handle_p
tr->ccu4_slice_ptr, XMC_CCU4_SLICE_IRQ_ID_PERIOD_M
ATCH,
00202                                     handle_p
tr->ccu4_slice_period_match_node);
00203   }
00204
00205   if ((bool) true == handle_ptr->compare_mat
ch_enable)
00206   {
00207     XMC_DEBUG("PWM_lCCU4_ConfigInterrupts:co
mpare match enable")
00208     XMC_CCU4_SLICE_EnableEvent(handle_ptr->c
cu4_slice_ptr, XMC_CCU4_SLICE_IRQ_ID_COMPARE_MATCH
_UP);
00209
00210     /* Bind event to Service Request Node to
 compare match event */
00211     XMC_CCU4_SLICE_SetInterruptNode(handle_p
tr->ccu4_slice_ptr, XMC_CCU4_SLICE_IRQ_ID_COMPARE_
MATCH_UP,
00212                                     handle_p
tr->ccu4_slice_compare_match_node);
00213   }
00214 }
00215
```

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/*Starts the CCU4 slice. */
void PWM_lCCU4_Start(PWM_t *const handle_ptr)
{
  if ((PWM_STATUS_SUCCESS == handle_ptr->state) || (PWM_STATUS_STOPPED == handle_ptr->state))
  {
    /* Clears the IDLE mode for the slice */
    XMC_CCU4_EnableClock(handle_ptr->ccu4_kernel_ptr,handle_ptr->slice_number);
    XMC_CCU4_SLICE_StartTimer(handle_ptr->ccu4_slice_ptr);

    handle_ptr->state = PWM_STATUS_RUNNING;
    XMC_DEBUG("PWM_lCCU4_Start:start PWM")
  }
} /* end of PWM_lCCU4_Start() api */

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/*Stops the CCU4 slice. */
void PWM_lCCU4_Stop(PWM_t *const handle_ptr)
{
  if (PWM_STATUS_UNINITIALIZED != handle_ptr->state)
  {
    XMC_CCU4_SLICE_StopTimer(handle_ptr->ccu4_slice_ptr);
    XMC_CCU4_SLICE_ClearTimer(handle_ptr->ccu4_slice_ptr);
    XMC_CCU4_DisableClock(handle_ptr->ccu4_kernel_ptr,handle_ptr->slice_number);

```

```
    handle_ptr->state = PWM_STATUS_STOPPED;
    XMC_DEBUG("PWM_lCCU4_Stop:stop PWM")
  }
} /* end of PWM_lCCU4_Stop() api */

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/*Sets the duty cycle for CCU4 slice. */
PWM_STATUS_t PWM_lCCU4_SetDutyCycle(PWM_t *const handle_ptr, uint32_t duty_cycle)
{
  uint32_t period;
  uint32_t compare;
  PWM_STATUS_t status;

  XMC_ASSERT("PWM_lCCU4_SetDutyCycle:Invalid duty_cycle " , ((duty_cycle >= 0) && (duty_cycle <= PWM_MAX_DUTY_CYCLE)))

  status = PWM_STATUS_FAILURE;
  if (PWM_STATUS_UNINITIALIZED != handle_ptr->state)
  {
    /* Duty cycle needs between 0 and 10000 */
    if (duty_cycle <= PWM_MAX_DUTY_CYCLE)
    {
      /* period = (PR + 1) */
      period = (uint32_t)handle_ptr->period_value + 1U;

      /* Duty Cycle(symmetric) = (PR-CR1)+1 / period */
      compare = ((period * (PWM_MAX_DUTY_CYCLE - duty_cycle)) / ((uint32_t) 100 * PWM_DUTY_CYCLE_SCALE));
```

```
00267
00268       handle_ptr->compare_value = compare;
00269       handle_ptr->duty_cycle = duty_cycle;
00270
00271       XMC_CCU4_SLICE_SetTimerCompareMatch(ha
ndle_ptr->ccu4_slice_ptr, (uint16_t)compare);
00272       XMC_CCU4_EnableShadowTransfer(handle_p
tr->ccu4_kernel_ptr, handle_ptr->shadow_mask);
00273       status = PWM_STATUS_SUCCESS;
00274     }
00275   }
00276
00277   XMC_DEBUG("PWM_lCCU4_SetDutyCycle:dutycycl
e set")
00278   return (status);
00279 } /* end of PWM_lCCU4_SetDutyCycle() api */
00280
00281 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~*/
00282 /*Sets the frequency for CCU4 slice. */
00283 PWM_STATUS_t PWM_lCCU4_SetFreq(PWM_t *const
handle_ptr, uint32_t pwm_freq_hz)
00284 {
00285   PWM_STATUS_t status;
00286   uint32_t module_freq;
00287   uint8_t prescaler;
00288   uint32_t period_value;
00289   uint32_t compare;
00290
00291   XMC_ASSERT("PWM_lCCU4_SetFreq:Invalid pwm_
freq_hz " , (pwm_freq_hz != 0U))
00292
00293   status = handle_ptr->state;
00294   prescaler = 0U;
00295   period_value = 0U;
00296
```

```
00297   /* Can't set the frequency when the PWM is
 not yet initialized or when required frequency is
 0*/
00298   if ((status != PWM_STATUS_UNINITIALIZED) &
& ((uint32_t)0 != pwm_freq_hz))
00299   {
00300     status = PWM_STATUS_SUCCESS;
00301     /*Get the Module frequency*/
00302     module_freq = handle_ptr->global_ccu4_ha
ndle->module_frequency;
00303
00304     /*Calculate the prescaler and the period
 register values.*/
00305     while (prescaler <= PWM_MAX_PRESCALER)
00306     {
00307       period_value = (uint32_t)((uint32_t)mo
dule_freq / (uint32_t)pwm_freq_hz) >> (uint32_t)pr
escaler;
00308       /*If the prescaler selected is not big
 enough goto the next prescaler value else come ou
t.*/
00309       if (period_value <= PWM_MAX_TIMER_COUN
T )
00310       {
00311         break;
00312       }
00313       prescaler++;
00314     }
00315
00316     /*Can't set the frequency if the require
d value is too small or when the required frequenc
y is too large.*/
00317     if ((prescaler > PWM_MAX_PRESCALER) || (
(uint32_t)0 == period_value))
00318     {
00319       XMC_DEBUG("PWM_lCCU4_SetFreq:Frequency
 could not be set")
```

```
00320       status = PWM_STATUS_FAILURE;
00321     }
00322     else
00323     {
00324       /*Calculate the new compare values usi
ng new period values */
00325       compare = (period_value * (PWM_MAX_DUT
Y_CYCLE - handle_ptr->duty_cycle))
00326                           / ((uint32_t) 100 * PWM
_DUTY_CYCLE_SCALE);
00327
00328       XMC_CCU4_SLICE_SetPrescaler(handle_ptr
->ccu4_slice_ptr, prescaler);
00329
00330       /* The period register is always one c
ount less than calculated.*/
00331       period_value = period_value - (uint32_
t)1;
00332       XMC_CCU4_SLICE_SetTimerPeriodMatch(han
dle_ptr->ccu4_slice_ptr, (uint16_t)(period_value))
;
00333
00334       XMC_CCU4_SLICE_SetTimerCompareMatch(ha
ndle_ptr->ccu4_slice_ptr, (uint16_t)compare);
00335
00336       XMC_CCU4_EnableShadowTransfer(handle_p
tr->ccu4_kernel_ptr, handle_ptr->shadow_mask);
00337
00338       handle_ptr->compare_value = compare;
00339       handle_ptr->period_value = period_valu
e;
00340       XMC_DEBUG("PWM_lCCU4_SetFreq:frequency
 set")
00341     }
00342   }
00343   else
00344   {
```

```
00345     status = PWM_STATUS_FAILURE;
00346     XMC_DEBUG("PWM_lCCU4_SetFreq:Frequency c
ould not be set")
00347   }
00348
00349
00350   return status;
00351 } /* end of PWM_lCCU4_SetFreq() api */
00352
00353 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/
00354 /*Sets the frequency and duty cycle for CCU4
 slice. */
00355 PWM_STATUS_t PWM_lCCU4_SetFreqAndDutyCycle(P
WM_t *const handle_ptr, uint32_t pwm_freq_hz, uint
32_t duty_cycle)
00356 {
00357   PWM_STATUS_t status;
00358   uint32_t module_freq;
00359   uint8_t prescaler;
00360   uint32_t period_value;
00361   uint32_t compare;
00362
00363   XMC_ASSERT("PWM_lCCU4_SetFreqAndDutyCycle:
Invalid pwm_freq_hz " , (pwm_freq_hz != 0U))
00364   XMC_ASSERT("PWM_lCCU4_SetFreqAndDutyCycle:
Invalid duty_cycle",((duty_cycle >= 0) &&
00365
                  (duty_cycle <= PWM_MAX_DUTY_CYC
LE)))
00366
00367   status = handle_ptr->state;
00368   prescaler = 0U;
00369   period_value = 0U;
00370
00371   /* Can't set the frequency when the PWM is
```

```
 not yet initialized or when required frequency is
 0*/
00372   if ((status != PWM_STATUS_UNINITIALIZED) &
& ((uint32_t)0 != pwm_freq_hz))
00373   {
00374     status = PWM_STATUS_SUCCESS;
00375     /*Get the Module frequency*/
00376     module_freq = handle_ptr->global_ccu4_ha
ndle->module_frequency;
00377
00378     /*Calculate the prescaler and the period
 register values.*/
00379     while (prescaler <= PWM_MAX_PRESCALER)
00380     {
00381       period_value = (uint32_t)((uint32_t)mo
dule_freq / (uint32_t)pwm_freq_hz) >> (uint32_t)pr
escaler;
00382       /*If the prescaler selected is not big
 enough goto the next prescaler value else come ou
t.*/
00383       if (period_value <= PWM_MAX_TIMER_COUN
T )
00384       {
00385         break;
00386       }
00387
00388       prescaler++;
00389     }
00390
00391     /*Can't set the frequency if the require
d value is too small or when the required frequenc
y is too large.*/
00392     if ((prescaler > PWM_MAX_PRESCALER) || (
duty_cycle > PWM_MAX_DUTY_CYCLE) || ((uint32_t)0 =
= period_value))
00393     {
00394       XMC_DEBUG("PWM_lCCU4_SetFreqAndDutyCyc
```

```
le:Frequency or duty cycle could not be set")
00395       status = PWM_STATUS_FAILURE;
00396     }
00397     else
00398     {
00399       /*Calculate the new compare values usi
ng new period values */
00400       compare = (period_value * ((uint32_t)P
WM_MAX_DUTY_CYCLE - duty_cycle)) / ((uint32_t) 100
 * PWM_DUTY_CYCLE_SCALE);
00401
00402       XMC_CCU4_SLICE_SetPrescaler(handle_ptr
->ccu4_slice_ptr, prescaler);
00403
00404       /* The period register is always one c
ount less than calculated.*/
00405       period_value = period_value - (uint32_
t)1;
00406       XMC_CCU4_SLICE_SetTimerPeriodMatch(han
dle_ptr->ccu4_slice_ptr, (uint16_t)(period_value))
;
00407
00408       XMC_CCU4_SLICE_SetTimerCompareMatch(ha
ndle_ptr->ccu4_slice_ptr, (uint16_t)compare);
00409
00410       XMC_CCU4_EnableShadowTransfer(handle_p
tr->ccu4_kernel_ptr, handle_ptr->shadow_mask);
00411
00412       handle_ptr->compare_value = compare;
00413       handle_ptr->period_value = period_valu
e;
00414       handle_ptr->duty_cycle = duty_cycle;
00415       XMC_DEBUG("PWM_lCCU4_SetFreqAndDutyCyc
le:frequency and duty cycle set")
00416     }
00417   }
00418   else
```

```
00419   {
00420     status = PWM_STATUS_FAILURE;
00421     XMC_DEBUG("PWM_lCCU4_SetFreqAndDutyCycle
:Frequency or duty cycle could not be set")
00422   }
00423
00424
00425   return status;
00426 } /* end of PWM_lCCU4_SetFreqAndDutyCycle()
api */
00427
00428 #endif /* end of CCU4 function definitions */

00429
00430 #ifdef PWM_SLICE_USED_CCU8
00431
00432 /*Initialize the APP and CCU8 slice. */
00433 PWM_STATUS_t PWM_lCCU8_Init(PWM_t *const han
dle_ptr)
00434 {
00435   PWM_STATUS_t status = PWM_STATUS_FAILURE;
00436
00437   if (PWM_STATUS_UNINITIALIZED == handle_ptr
->state)
00438   {
00439     /* Initialize consumed Apps */
00440     status = (PWM_STATUS_t)GLOBAL_CCU8_Init(
handle_ptr->global_ccu8_handle);
00441
00442     /*Initialize CCU8 slice */
00443     if (PWM_STATUS_SUCCESS == status)
00444     {
00445       XMC_DEBUG("PWM_lCCU8_Init:Initilizing
Slice")
00446       XMC_CCU8_SLICE_CompareInit(handle_ptr-
>ccu8_slice_ptr, handle_ptr->ccu8_slice_config_ptr
);
```

```
00447
00448       /* Set the period and compare register
 values */
00449       XMC_CCU8_SLICE_SetTimerPeriodMatch(han
dle_ptr->ccu8_slice_ptr,
00450
(uint16_t)handle_ptr->period_value);
00451
00452       XMC_CCU8_SLICE_SetTimerCompareMatch(ha
ndle_ptr->ccu8_slice_ptr, XMC_CCU8_SLICE_COMPARE_C
HANNEL_1,
00453
  (uint16_t)handle_ptr->compare_value);
00454
00455       XMC_CCU8_EnableShadowTransfer(handle_p
tr->ccu8_kernel_ptr, handle_ptr->shadow_mask);
00456
00457       /* Initialize interrupts */
00458       PWM_lCCU8_ConfigInterrupts(handle_ptr)
;
00459
00460       XMC_GPIO_Init(handle_ptr->gpio_out_por
t,handle_ptr->gpio_out_pin,
00461                             handle_ptr->gpio_out
_config);
00462
00463       handle_ptr->state = PWM_STATUS_SUCCESS
;
00464
00465       /* Start the PWM generation if start a
t initialization is enabled */
00466       if ((bool) true == handle_ptr->start_c
ontrol)
00467       {
00468         PWM_Start(handle_ptr);
00469       }
00470       status = PWM_STATUS_SUCCESS;
```

```
00471     }
00472     else
00473     {
00474       handle_ptr->state = PWM_STATUS_UNINITI
ALIZED;
00475     }
00476
00477   }
00478   return(status);
00479 } /* end of PWM_lCCU8_Init() api */
00480
00481 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~*/
00482 /* Initialize interrupts */
00483 void PWM_lCCU8_ConfigInterrupts(PWM_t *const
 handle_ptr)
00484 {
00485   if ((bool) true == handle_ptr->period_matc
h_enable)
00486   {
00487     XMC_DEBUG("PWM_lCCU8_ConfigInterrupts:pe
riod match event enable")
00488
00489     XMC_CCU8_SLICE_EnableEvent(handle_ptr->c
cu8_slice_ptr, XMC_CCU8_SLICE_IRQ_ID_PERIOD_MATCH)
;
00490
00491     /* Bind event to Service Request Node fo
r period match event */
00492     XMC_CCU8_SLICE_SetInterruptNode(handle_p
tr->ccu8_slice_ptr, XMC_CCU8_SLICE_IRQ_ID_PERIOD_M
ATCH,
00493                                     handle_p
tr->ccu8_slice_period_match_node);
00494   }
00495
```

```
00496   if ((bool) true == handle_ptr->compare_mat
ch_enable)
00497   {
00498     XMC_DEBUG("PWM_lCCU8_ConfigInterrupts:co
mpare match event enable ")
00499
00500     XMC_CCU8_SLICE_EnableEvent(handle_ptr->c
cu8_slice_ptr, XMC_CCU8_SLICE_IRQ_ID_COMPARE_MATCH
_UP_CH_1);
00501     /* Bind event to Service Request Node fo
r compare match event */
00502     XMC_CCU8_SLICE_SetInterruptNode(handle_p
tr->ccu8_slice_ptr, XMC_CCU8_SLICE_IRQ_ID_COMPARE_
MATCH_UP_CH_1,
00503                                     handle_p
tr->ccu8_slice_compare_match_node);
00504   }
00505 }
00506
00507 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~*/
00508 /*Starts the CCU8 slice. */
00509 void PWM_lCCU8_Start(PWM_t *const handle_ptr
)
00510 {
00511   XMC_ASSERT("PWM_lCCU8_Start:Invalid handle
_ptr" , (handle_ptr != NULL))
00512
00513   if ((PWM_STATUS_SUCCESS == handle_ptr->sta
te) || (PWM_STATUS_STOPPED == handle_ptr->state))
00514   {
00515     /* Clears IDLE mode for the slice */
00516     XMC_CCU8_EnableClock(handle_ptr->ccu8_ke
rnel_ptr,handle_ptr->slice_number);
00517     XMC_CCU8_SLICE_StartTimer(handle_ptr->cc
u8_slice_ptr);
```

```
    handle_ptr->state = PWM_STATUS_RUNNING;
    XMC_DEBUG("PWM_lCCU8_Start:start PWM")
  }
} /* end of PWM_lCCU8_Start() api */

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/*Stops the CCU8 slice. */
void PWM_lCCU8_Stop(PWM_t *const handle_ptr)
{
  XMC_ASSERT("PWM_lCCU8_Stop:Invalid handle_ptr" , (handle_ptr != NULL))

  if (PWM_STATUS_UNINITIALIZED != handle_ptr->state)
  {
    XMC_CCU8_SLICE_StopTimer(handle_ptr->ccu8_slice_ptr);
    XMC_CCU8_SLICE_ClearTimer(handle_ptr->ccu8_slice_ptr);
    XMC_CCU8_DisableClock(handle_ptr->ccu8_kernel_ptr, handle_ptr->slice_number);

    handle_ptr->state = PWM_STATUS_STOPPED;
    XMC_DEBUG("PWM_lCCU8_Stop:stop PWM")
  }
} /* end of PWM_lCCU8_Stop() api */

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/*Sets the duty cycle for CCU8 slice. */
PWM_STATUS_t PWM_lCCU8_SetDutyCycle(PWM_t *const handle_ptr, uint32_t duty_cycle)
{
```

```
00545   uint32_t period;
00546   uint32_t compare;
00547   PWM_STATUS_t status;
00548
00549   XMC_ASSERT("PWM_lCCU8_SetDutyCycle:Invalid
 handle_ptr" , (handle_ptr != NULL))
00550   XMC_ASSERT("PWM_lCCU8_SetDutyCycle:Invalid
 duty_cycle",((duty_cycle >= 0) &&
00551
              (duty_cycle <= PWM_MAX_DUTY_CYCLE)))
00552
00553   status = PWM_STATUS_FAILURE;
00554   if (handle_ptr->state != PWM_STATUS_UNINIT
IALIZED)
00555   {
00556     /* Duty cycle needs between 0 and 10000
*/
00557     if (duty_cycle <= PWM_MAX_DUTY_CYCLE)
00558     {
00559       period = (uint32_t)handle_ptr->period_
value + 1U;
00560
00561       /* Duty Cycle(symmetric) = (PR-CR1)+1
/ period */
00562       compare = ((period * ((uint32_t) PWM_M
AX_DUTY_CYCLE - duty_cycle)) / ((uint32_t) 100 * P
WM_DUTY_CYCLE_SCALE));
00563
00564       handle_ptr->compare_value = compare;
00565       handle_ptr->duty_cycle = duty_cycle;
00566
00567       XMC_CCU8_SLICE_SetTimerCompareMatch(ha
ndle_ptr->ccu8_slice_ptr, XMC_CCU8_SLICE_COMPARE_C
HANNEL_1,
00568
 (uint16_t)compare);
00569
```

```
00570       XMC_CCU8_EnableShadowTransfer(handle_p
tr->ccu8_kernel_ptr, handle_ptr->shadow_mask);
00571       status = PWM_STATUS_SUCCESS;
00572     }
00573   }
00574
00575   XMC_DEBUG("PWM_lCCU8_SetDutyCycle:dutycycl
e set")
00576   return (status);
00577 } /* end of PWM_lCCU8_SetDutyCycle() api */
00578
00579 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~*/
00580 /*Sets the frequency for CCU8 slice. */
00581 PWM_STATUS_t PWM_lCCU8_SetFreq(PWM_t *const
handle_ptr, uint32_t pwm_freq_hz)
00582 {
00583   PWM_STATUS_t status;
00584   uint32_t module_freq;
00585   uint8_t prescaler;
00586   uint32_t period_value;
00587   uint32_t compare;
00588
00589   XMC_ASSERT("PWM_lCCU8_SetFreq:Invalid pwm_
freq_hz " , (pwm_freq_hz != 0U))
00590
00591   status = handle_ptr->state;
00592   prescaler = 0U;
00593   period_value = 0U;
00594
00595   /* Can't set the frequency when the PWM is
 not yet initialized or when required frequency is
 0*/
00596   if ((status != PWM_STATUS_UNINITIALIZED) &
& ((uint32_t)0 != pwm_freq_hz))
00597   {
```

```
00598     status = PWM_STATUS_SUCCESS;
00599     /*Get the Module frequency*/
00600     module_freq = handle_ptr->global_ccu8_ha
ndle->module_frequency;
00601
00602     /*Calculate the prescaler and the period
 register values.*/
00603     while (prescaler <= PWM_MAX_PRESCALER)
00604     {
00605       period_value = (uint32_t)((uint32_t)mo
dule_freq / (uint32_t)pwm_freq_hz) >> (uint32_t)pr
escaler;
00606       /*If the prescaler selected is not big
 enough goto the next prescaler value else come ou
t.*/
00607       if (period_value <= PWM_MAX_TIMER_COUN
T)
00608       {
00609         break;
00610       }
00611
00612       prescaler++;
00613     }
00614
00615     /*Can't set the frequency if the require
d value is too small or when the required frequenc
y is too large.*/
00616     if ((prescaler > PWM_MAX_PRESCALER) || (
(uint32_t)0 == period_value))
00617     {
00618       XMC_DEBUG("PWM_lCCU8_SetFreq:frequency
 could not be set")
00619       status = PWM_STATUS_FAILURE;
00620     }
00621     else
00622     {
00623       /*Calculate the new compare values usi
```

```
ng new period values*/
00624       compare = (period_value * (PWM_MAX_DUT
Y_CYCLE - handle_ptr->duty_cycle))
00625                                 / ((uint32_t) 1
00 * PWM_DUTY_CYCLE_SCALE);
00626
00627       XMC_CCU8_SLICE_SetPrescaler(handle_ptr
->ccu8_slice_ptr, prescaler);
00628
00629       /* The period register is always one c
ount less than calculated.*/
00630       period_value = period_value - (uint32_
t)1;
00631       XMC_CCU8_SLICE_SetTimerPeriodMatch(han
dle_ptr->ccu8_slice_ptr, (uint16_t)(period_value))
;
00632
00633       XMC_CCU8_SLICE_SetTimerCompareMatch(ha
ndle_ptr->ccu8_slice_ptr, XMC_CCU8_SLICE_COMPARE_C
HANNEL_1,
00634
 (uint16_t)compare);
00635
00636       XMC_CCU8_EnableShadowTransfer(handle_p
tr->ccu8_kernel_ptr, handle_ptr->shadow_mask);
00637
00638       handle_ptr->compare_value = compare;
00639       handle_ptr->period_value = period_valu
e;
00640       XMC_DEBUG("PWM_lCCU8_SetFreq:frequency
 set")
00641         }
00642   }
00643   else
00644   {
00645     status = PWM_STATUS_FAILURE;
00646     XMC_DEBUG("PWM_lCCU8_SetFreq:frequency c
```

```
ould not be set")
  }

  return status;
} /* end of PWM_lCCU8_SetFreq() api */

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/*Sets the frequency and duty cycle for CCU8 slice. */
PWM_STATUS_t PWM_lCCU8_SetFreqAndDutyCycle(PWM_t *const handle_ptr, uint32_t pwm_freq_hz, uint32_t duty_cycle)
{
  PWM_STATUS_t status;
  uint32_t module_freq;
  uint8_t prescaler;
  uint32_t period_value;
  uint32_t compare;

  XMC_ASSERT("PWM_lCCU8_SetFreqAndDutyCycle:Invalid pwm_freq_hz " , (pwm_freq_hz != 0U))
  XMC_ASSERT("PWM_lCCU8_SetFreqAndDutyCycle:Invalid duty_cycle",((duty_cycle >= 0) &&
                                                           (duty_cycle <= PWM_MAX_DUTY_CYCLE)))

  status = handle_ptr->state;
  prescaler = 0U;
  period_value = 0U;

  /* Can't set the frequency when the PWM is not yet initialized or when required frequency is 0*/
  if ((status != PWM_STATUS_UNINITIALIZED) &
```

```
& ((uint32_t)0 != pwm_freq_hz))
00672   {
00673     status = PWM_STATUS_SUCCESS;
00674     /*Get the Module frequency*/
00675     module_freq = handle_ptr->global_ccu8_ha
ndle->module_frequency;
00676
00677     /*Calculate the prescaler and the period
 register values.*/
00678     while (prescaler <= PWM_MAX_PRESCALER)
00679     {
00680       period_value = (uint32_t)((uint32_t)mo
dule_freq / (uint32_t)pwm_freq_hz) >> (uint32_t)pr
escaler;
00681           /*If the prescaler selected is not
 big enough goto the next prescaler value else com
e out.*/
00682       if (period_value <= PWM_MAX_TIMER_COUN
T)
00683       {
00684         break;
00685       }
00686       prescaler++;
00687     }
00688
00689     /*Can't set the frequency if the require
d value is too small or when the required frequenc
y is too large.*/
00690     if ((prescaler > PWM_MAX_PRESCALER) || (
duty_cycle > PWM_MAX_DUTY_CYCLE) || ((uint32_t)0 =
= period_value))
00691     {
00692       XMC_DEBUG("PWM_lCCU8_SetFreqAndDutyCyc
le:Frequency or duty cycle could not be set")
00693       status = PWM_STATUS_FAILURE;
00694     }
00695     else
```

```
00696     {
00697       /*Calculate the new compare values usi
ng new period values */
00698       compare = (period_value * ((uint32_t)P
WM_MAX_DUTY_CYCLE - duty_cycle)) / ((uint32_t) 100
 * PWM_DUTY_CYCLE_SCALE);
00699
00700       XMC_CCU8_SLICE_SetPrescaler(handle_ptr
->ccu8_slice_ptr, prescaler);
00701
00702       /* The period register is always one c
ount less than calculated.*/
00703       period_value = period_value - (uint32_
t)1;
00704       XMC_CCU8_SLICE_SetTimerPeriodMatch(han
dle_ptr->ccu8_slice_ptr, (uint16_t)(period_value))
;
00705
00706       XMC_CCU8_SLICE_SetTimerCompareMatch(ha
ndle_ptr->ccu8_slice_ptr, XMC_CCU8_SLICE_COMPARE_C
HANNEL_1,
00707
 (uint16_t)compare);
00708
00709       XMC_CCU8_EnableShadowTransfer(handle_p
tr->ccu8_kernel_ptr, handle_ptr->shadow_mask);
00710
00711       handle_ptr->compare_value = compare;
00712       handle_ptr->period_value = period_valu
e;
00713       handle_ptr->duty_cycle = duty_cycle;
00714       XMC_DEBUG("PWM_lCCU8_SetFreqAndDutyCyc
le:Frequency and Duty cycle set")
00715     }
00716   }
00717   else
00718   {
```

```
00719     status = PWM_STATUS_FAILURE;
00720     XMC_DEBUG("PWM_lCCU8_SetFreqAndDutyCycle
:Frequency and Duty cycle could not be set")
00721   }
00722
00723
00724   return status;
00725 } /* end of PWM_lCCU8_SetFreqAndDutyCycle()
api */
00726
00727 #endif /* end of CCU8 function definitions */

00728
00729 /*************************************************
*************************************************
************************
00730  * API IMPLEMENTATION
00731  *************************************************
*************************************************
************************/
00732
00733 /*This function returns the version of the P
WM App*/
00734 DAVE_APP_VERSION_t PWM_GetAppVersion(void)
00735 {
00736   DAVE_APP_VERSION_t version;
00737
00738   version.major = (uint8_t) PWM_MAJOR_VERSIO
N;
00739   version.minor = (uint8_t) PWM_MINOR_VERSIO
N;
00740   version.patch = (uint8_t) PWM_PATCH_VERSIO
N;
00741
00742   return version;
00743 }
00744
```

```
00745 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~*/
00746 /* This function initializes the app */
00747 PWM_STATUS_t PWM_Init(PWM_t *const handle_pt
r)
00748 {
00749   PWM_STATUS_t status;
00750   status = PWM_STATUS_FAILURE;
00751
00752   XMC_ASSERT("PWM_Init:Invalid handle_ptr" ,
 (handle_ptr != NULL))
00753
00754 #ifdef PWM_SLICE_USED_CCU4
00755   if (PWM_TIMER_SLICE_CCU4 == handle_ptr->ti
mer_type)
00756   {
00757     status = PWM_lCCU4_Init(handle_ptr);
00758   }
00759 #endif
00760
00761 #ifdef PWM_SLICE_USED_CCU8
00762   if (PWM_TIMER_SLICE_CCU8 == handle_ptr->ti
mer_type)
00763   {
00764     status = PWM_lCCU8_Init(handle_ptr);
00765   }
00766 #endif
00767
00768   return (status);
00769 }
00770
00771 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~*/
00772 /* This function starts the PWM generation.
This needs to be called even if external start is
```

```
configured.*/
00773 void PWM_Start(PWM_t *const handle_ptr)
00774 {
00775   XMC_ASSERT("PWM_Start:Invalid handle_ptr"
, (handle_ptr != NULL))
00776
00777 #ifdef PWM_SLICE_USED_CCU4
00778   if (PWM_TIMER_SLICE_CCU4 == handle_ptr->ti
mer_type)
00779   {
00780     PWM_lCCU4_Start(handle_ptr);
00781   }
00782 #endif
00783
00784 #ifdef PWM_SLICE_USED_CCU8
00785   if (PWM_TIMER_SLICE_CCU8 == handle_ptr->ti
mer_type)
00786   {
00787     PWM_lCCU8_Start(handle_ptr);
00788   }
00789 #endif
00790 }
00791
00792 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~*/
00793 /* This function stops the PWM generation. */

00794 void PWM_Stop(PWM_t *const handle_ptr)
00795 {
00796
00797   XMC_ASSERT("PWM_Stop:Invalid handle_ptr" ,
 (handle_ptr != NULL))
00798
00799 #ifdef PWM_SLICE_USED_CCU4
00800   if (PWM_TIMER_SLICE_CCU4 == handle_ptr->ti
mer_type)
```

```
00801   {
00802     PWM_lCCU4_Stop(handle_ptr);
00803   }
00804 #endif
00805
00806 #ifdef PWM_SLICE_USED_CCU8
00807   if (PWM_TIMER_SLICE_CCU8 == handle_ptr->ti
mer_type)
00808   {
00809     PWM_lCCU8_Stop(handle_ptr);
00810   }
00811 #endif
00812 }
00813
00814 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~*/
00815 /*This function is used to set the duty cycl
e (uint32_t) of the PWM waveform  */
00816 PWM_STATUS_t PWM_SetDutyCycle(PWM_t *const h
andle_ptr, uint32_t duty_cycle)
00817 {
00818   PWM_STATUS_t status;
00819   status = PWM_STATUS_FAILURE;
00820
00821   XMC_ASSERT("PWM_SetDutyCycle:Invalid handl
e_ptr" , (handle_ptr != NULL))
00822
00823 #ifdef PWM_SLICE_USED_CCU4
00824   if (PWM_TIMER_SLICE_CCU4 == handle_ptr->ti
mer_type)
00825   {
00826     status = PWM_lCCU4_SetDutyCycle(handle_p
tr, duty_cycle);
00827   }
00828 #endif
00829
```

```
00830 #ifdef PWM_SLICE_USED_CCU8
00831   if (PWM_TIMER_SLICE_CCU8 == handle_ptr->ti
mer_type)
00832   {
00833     status = PWM_lCCU8_SetDutyCycle(handle_p
tr, duty_cycle);
00834   }
00835 #endif
00836   return (status);
00837 }
00838
00839 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~*/
00840 /*This function changes the PWM frequency. I
nput parameter is the frequency value in Hz */
00841 PWM_STATUS_t PWM_SetFreq(PWM_t *const handle
_ptr, uint32_t pwm_freq_hz)
00842 {
00843   PWM_STATUS_t status;
00844   status = PWM_STATUS_FAILURE;
00845
00846   XMC_ASSERT("PWM_SetFreq:Invalid handle_ptr"
 , (handle_ptr != NULL))
00847
00848 #ifdef PWM_SLICE_USED_CCU4
00849   if (PWM_TIMER_SLICE_CCU4 == handle_ptr->ti
mer_type)
00850   {
00851     status = PWM_lCCU4_SetFreq(handle_ptr, p
wm_freq_hz);
00852   }
00853 #endif
00854
00855 #ifdef PWM_SLICE_USED_CCU8
00856   if (PWM_TIMER_SLICE_CCU8 == handle_ptr->ti
mer_type)
```

```
00857   {
00858     status = PWM_lCCU8_SetFreq(handle_ptr, p
wm_freq_hz);
00859   }
00860 #endif
00861   return status;
00862 }
00863
00864 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~*/
00865 /*This function sets frequency and the duty
cycle */
00866 PWM_STATUS_t PWM_SetFreqAndDutyCycle(PWM_t *
const handle_ptr, uint32_t pwm_freq_hz, uint32_t d
uty_cycle)
00867 {
00868   PWM_STATUS_t status;
00869   status = PWM_STATUS_FAILURE;
00870
00871   XMC_ASSERT("PWM_SetFreqAndDutyCycle:Invali
d handle_ptr" , (handle_ptr != NULL))
00872
00873 #ifdef PWM_SLICE_USED_CCU4
00874   if (PWM_TIMER_SLICE_CCU4 == handle_ptr->ti
mer_type)
00875   {
00876     status = PWM_lCCU4_SetFreqAndDutyCycle(h
andle_ptr, pwm_freq_hz, duty_cycle);
00877   }
00878 #endif
00879
00880 #ifdef PWM_SLICE_USED_CCU8
00881   if (PWM_TIMER_SLICE_CCU8 == handle_ptr->ti
mer_type)
00882   {
00883     status = PWM_lCCU8_SetFreqAndDutyCycle(h
```

```
andle_ptr, pwm_freq_hz, duty_cycle);
  }
#endif
  return status;
}

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/*This function changes the PWM timer status_timer */
bool PWM_GetTimerStatus(PWM_t *const handle_ptr)
{
  bool status_timer;
  status_timer = (bool)false;

  XMC_ASSERT("PWM_GetTimerStatus:Invalid handle_ptr" , (handle_ptr != NULL))

#ifdef PWM_SLICE_USED_CCU4
  if (PWM_TIMER_SLICE_CCU4 == handle_ptr->timer_type)
  {
    status_timer = XMC_CCU4_SLICE_IsTimerRunning(handle_ptr->ccu4_slice_ptr);
  }
#endif

#ifdef PWM_SLICE_USED_CCU8
  if (PWM_TIMER_SLICE_CCU8 == handle_ptr->timer_type)
  {
    status_timer = XMC_CCU8_SLICE_IsTimerRunning(handle_ptr->ccu8_slice_ptr);
  }
#endif
```

```

  return (status_timer);
}

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/*This function returns the interrupt status_timer */
bool PWM_GetInterruptStatus(PWM_t *const handle_ptr, PWM_INTERRUPT_t pwm_interrupt)
{
  bool status;
  status = (bool) false;

  XMC_ASSERT("PWM_GetInterruptStatus:Invalid handle_ptr" , (handle_ptr != NULL))

#ifdef PWM_SLICE_USED_CCU4
  if (PWM_TIMER_SLICE_CCU4 == handle_ptr->timer_type)
  {
    status = XMC_CCU4_SLICE_GetEvent(handle_ptr->ccu4_slice_ptr, (XMC_CCU4_SLICE_IRQ_ID_t)pwm_interrupt);
  }
#endif

#ifdef PWM_SLICE_USED_CCU8
  if (PWM_TIMER_SLICE_CCU8 == handle_ptr->timer_type)
  {
    status = XMC_CCU8_SLICE_GetEvent(handle_ptr->ccu8_slice_ptr, (XMC_CCU8_SLICE_IRQ_ID_t)pwm_interrupt);
  }
#endif
```

```c

  return status;
}

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/*This function Acknowledges the corresponding interrupt */
void PWM_ClearEvent(PWM_t *const handle_ptr, PWM_INTERRUPT_t pwm_interrupt)
{
  XMC_ASSERT("PWM_ClearEvent:Invalid handle_ptr" , (handle_ptr != NULL))

#ifdef PWM_SLICE_USED_CCU4
  if (PWM_TIMER_SLICE_CCU4 == handle_ptr->timer_type)
  {
    XMC_CCU4_SLICE_ClearEvent(handle_ptr->ccu4_slice_ptr, (XMC_CCU4_SLICE_IRQ_ID_t) pwm_interrupt);
  }
#endif

#ifdef PWM_SLICE_USED_CCU8
  if (PWM_TIMER_SLICE_CCU8 == handle_ptr->timer_type)
  {
    XMC_CCU8_SLICE_ClearEvent(handle_ptr->ccu8_slice_ptr, (XMC_CCU8_SLICE_IRQ_ID_t) pwm_interrupt);
  }
#endif
}

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/*This function sets the passive level of the PWM*/
void PWM_SetPassiveLevel(PWM_t *const handle_ptr, PWM_OUTPUT_PASSIVE_LEVEL_t pwm_output_passive_level)
{
  XMC_ASSERT("PWM_SetPassiveLevel:Invalid handle_ptr" , (handle_ptr != NULL))
  XMC_ASSERT("PWM_SetPassiveLevel:Invalid pwm_output_passive_level " ,
              (pwm_output_passive_level < PWM_OUTPUT_PASSIVE_LEVEL_MAX));

#ifdef PWM_SLICE_USED_CCU4
  if (PWM_TIMER_SLICE_CCU4 == handle_ptr->timer_type)
  {
    XMC_CCU4_SLICE_SetPassiveLevel(handle_ptr->ccu4_slice_ptr,
                                         (XMC_CCU4_SLICE_OUTPUT_PASSIVE_LEVEL_t)pwm_output_passive_level);

    XMC_CCU4_EnableShadowTransfer(handle_ptr->ccu4_kernel_ptr, handle_ptr->shadow_mask);
    XMC_DEBUG("PWM_SetPassiveLevel:CCU4 slice, passive level changed")
  }
#endif

#ifdef PWM_SLICE_USED_CCU8
  if (PWM_TIMER_SLICE_CCU8 == handle_ptr->timer_type)
  {
    XMC_CCU8_SLICE_SetPassiveLevel(handle_pt
```

```
r->ccu8_slice_ptr, XMC_CCU8_SLICE_OUTPUT_0,
00985                                                 (XMC
_CCU8_SLICE_OUTPUT_PASSIVE_LEVEL_t)pwm_output_pass
ive_level);
00986
00987     XMC_CCU8_EnableShadowTransfer(handle_ptr
->ccu8_kernel_ptr, handle_ptr->shadow_mask);
00988     XMC_DEBUG("PWM_SetPassiveLevel:CCU8 slic
e, passive level changed")
00989   }
00990 #endif
00991 }
00992
00993 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~*/
00994 /*Configures the period register */
00995 PWM_STATUS_t PWM_SetPeriodMatchValue(PWM_t *
const handle_ptr, uint32_t period_match_value)
00996 {
00997   uint32_t compare;
00998   PWM_STATUS_t status;
00999
01000   XMC_ASSERT("PWM_SetPeriodMatchValue:Invali
d handle_ptr" , (handle_ptr != NULL))
01001   XMC_ASSERT("PWM_SetPeriodMatchValue:Invali
d period_match_value" , (period_match_value <= PWM
_MAX_PERIOD_VALUE))
01002
01003   status = handle_ptr->state;
01004
01005   if ( (status != PWM_STATUS_UNINITIALIZED)
&& (PWM_MAX_PERIOD_VALUE >= period_match_value))
01006   {
01007     compare = (period_match_value * ((uint32
_t)PWM_MAX_DUTY_CYCLE - handle_ptr->duty_cycle))
01008                                                 /
```

```
 ((uint32_t) 100 * PWM_DUTY_CYCLE_SCALE);
01009
01010 #ifdef PWM_SLICE_USED_CCU4
01011     if (PWM_TIMER_SLICE_CCU4 == handle_ptr->
timer_type)
01012     {
01013       XMC_CCU4_SLICE_SetTimerPeriodMatch(han
dle_ptr->ccu4_slice_ptr, (uint16_t)period_match_va
lue);
01014
01015       XMC_CCU4_SLICE_SetTimerCompareMatch(ha
ndle_ptr->ccu4_slice_ptr, (uint16_t)compare );
01016
01017       XMC_CCU4_EnableShadowTransfer(handle_p
tr->ccu4_kernel_ptr, handle_ptr->shadow_mask);
01018     }
01019 #endif
01020
01021 #ifdef PWM_SLICE_USED_CCU8
01022     if (PWM_TIMER_SLICE_CCU8 == handle_ptr->
timer_type)
01023     {
01024       XMC_CCU8_SLICE_SetTimerPeriodMatch(han
dle_ptr->ccu8_slice_ptr, (uint16_t)period_match_va
lue);
01025
01026       XMC_CCU8_SLICE_SetTimerCompareMatch(ha
ndle_ptr->ccu8_slice_ptr, XMC_CCU8_SLICE_COMPARE_C
HANNEL_1,
01027
          (uint16_t)compare);
01028
01029       XMC_CCU8_EnableShadowTransfer(handle_p
tr->ccu8_kernel_ptr, handle_ptr->shadow_mask);
01030     }
01031 #endif
01032     handle_ptr->period_value =  period_match
```

```
_value;
01033     handle_ptr->compare_value =  compare;
01034     status = PWM_STATUS_SUCCESS;
01035   }
01036   else
01037   {
01038     status = PWM_STATUS_FAILURE;
01039   }
01040
01041   return (status);
01042 }
01043 /*CODE_BLOCK_END*/
01044
```