Here is a list of all modules:

- License Terms and Copyright Information
- Abbreviations and Definitions
- Overview
- Architecture Description
- APP Configuration Parameters
- Enumerations
- Data structures
- Methods
- Usage
- Release History
License Terms and Copyright Information

Copyright (c) 2015, Infineon Technologies AG All rights reserved.

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).
# Abbreviations and Definitions

## Abbreviations:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAVE™</td>
<td>Digital Application Virtual Engineer</td>
</tr>
<tr>
<td>APP</td>
<td>DAVE™ Application</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>MCU</td>
<td>Microcontroller Unit</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>LLD</td>
<td>Low Level Driver</td>
</tr>
<tr>
<td>IO</td>
<td>Input Output</td>
</tr>
<tr>
<td>CCU4</td>
<td>Capture Compare Unit 4</td>
</tr>
<tr>
<td>CCU8</td>
<td>Capture Compare Unit 8</td>
</tr>
<tr>
<td>CCUx</td>
<td>Capture Compare Unit 4/Capture Compare Unit 8</td>
</tr>
</tbody>
</table>

## Definitions:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singleton</td>
<td>Only single instance of the APP is permitted</td>
</tr>
<tr>
<td>Sharable</td>
<td>Resource sharing with other APPs is permitted</td>
</tr>
<tr>
<td>initProvider</td>
<td>Provides the initialization routine</td>
</tr>
<tr>
<td>Physical connectivity</td>
<td>Hardware inter/intra peripheral (constant) signal connection</td>
</tr>
<tr>
<td>Conditional connectivity</td>
<td>Constrained hardware inter/intra peripheral signal connection</td>
</tr>
<tr>
<td>Aggregation</td>
<td>Indicates consumption of low level (dependent)</td>
</tr>
</tbody>
</table>
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, shows the functional overview of the PWM APP. The CCU peripheral starts an 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** 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
5. XMC1400 Reference Manual
6. XMC1300 Reference Manual
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

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Input/Output</th>
<th>Availability</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>event_period_match</td>
<td>Output</td>
<td>Always</td>
<td>Period match interrupt signal: This can be connected to an INTERRUPT APP to generate the interrupt for each period match event.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Compare match interrupt signal: This can be connected with INTERRUPT</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>event_compare_match</th>
<th>Output</th>
<th>Always</th>
<th>APP to generate the interrupt for each compare match event.</th>
</tr>
</thead>
<tbody>
<tr>
<td>timer_status</td>
<td>Output</td>
<td>Always</td>
<td>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).</td>
</tr>
<tr>
<td>pwm_output</td>
<td>Output</td>
<td>Always</td>
<td>Output (OUT) signal: The output PWM signal can be connected with any pad pin. The list of available pins are shown in &quot;Manual Pin Allocator&quot; tab in DAVE™.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Global signal connection Connected between the kernel and the respective slice. Used to</td>
</tr>
</tbody>
</table>
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.

\[
Cmp_{\text{reg}} = \frac{\text{Period} \times (100 - \text{Duty})}{100}
\]

<table>
<thead>
<tr>
<th>Duty</th>
<th>Duty Min %</th>
<th>Duty Max %</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

Note: The value loaded to period register is \((\text{period} - 1)\)

**Figure 2:** Edge Aligned Symmetric Mode

Example:
Let

\[ \text{Clock} = 120\text{MHz}. \]

\[ \text{Prescaler} = 0; \]

\[ \text{Required frequency of operation (F)} = 100\text{KHz}. \]

\[ \text{Duty required (D)} = 30\%. \]

\[ \text{Period} = \frac{\text{Clock}}{(1 \ll \text{Prescaler})F} \]

\[ \text{Cmp}_{\text{reg}} = \frac{\text{Period} (100 - D)}{100} \]

\[ \text{Period} = \frac{120,000,000}{(1 \ll 0)100,000} \]

\[ \text{Period} = 1200 \]

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

\[ \text{Cmp}_{\text{reg}} = \frac{1200(100 - 30)}{100} \]

\[ \text{Cmp}_{\text{reg}} = 840 \]
PWM

APP Configuration Parameters

App Configuration Parameters

![General Settings](image)

**Figure 1: General Settings**
Figure 2: Event Settings
Figure 3: Pin Settings
# Enumerations

<table>
<thead>
<tr>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PWM_TIMER_SLICE</code></td>
<td>The type identifies the CCU4 or CCU8 timer selected.</td>
</tr>
<tr>
<td><code>PWM_TIMER_STATUS</code></td>
<td>The type identifies the timer status.</td>
</tr>
<tr>
<td><code>PWM_INTERRUPT</code></td>
<td>The type identifies the timer interrupts.</td>
</tr>
<tr>
<td><code>PWM_OUTPUT_PASSIVE_LEVEL</code></td>
<td>The type identifies the timer interrupts.</td>
</tr>
<tr>
<td><code>PWM_STATUS</code></td>
<td>The type identifies App state.</td>
</tr>
<tr>
<td><code>PWM_ERROR_CODES</code></td>
<td></td>
</tr>
</tbody>
</table>

```cpp
enum PWM_TIMER_SLICE
{
    PWM_TIMER_SLICE
}
```

```cpp
enum PWM_TIMER_STATUS
{
    PWM_TIMER_STATUS
}
```

```cpp
enum PWM_INTERRUPT
{
    PWM_INTERRUPT_PERIODMATCH = 0U,
    PWM_INTERRUPT_COMPAREMATCH = 2U
}
```

```cpp
enum PWM_OUTPUT_PASSIVE_LEVEL
{
    PWM_OUTPUT_PASSIVE_LEVEL_LOW = 0,
    PWM_OUTPUT_PASSIVE_LEVEL_HIGH
}
```

```cpp
enum PWM_STATUS
{
    PWM_STATUS_SUCCESS = 0,
    PWM_STATUS_FAILURE,
    PWM_STATUS_UNINITIALIZED,
    PWM_STATUS_RUNNING,
    PWM_STATUS_STOPPED
}
```

```cpp
enum PWM_ERROR_CODES
{
    PWM_OPER_NOT_ALLOWED_ERROR
}
```
The type identifies the App Error Codes.

<table>
<thead>
<tr>
<th>Typedef</th>
<th>Enum Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef</td>
<td>PWM_TIMER_SLICE</td>
<td>PWM_TIMER_SLICE_t</td>
</tr>
<tr>
<td>enum</td>
<td></td>
<td>The type identifies the CCU4 or CCU8 timer selected.</td>
</tr>
<tr>
<td>typedef</td>
<td>PWM_TIMER_STATUS</td>
<td>PWM_TIMER_STATUS_t</td>
</tr>
<tr>
<td>enum</td>
<td></td>
<td>The type identifies the timer status.</td>
</tr>
<tr>
<td>typedef</td>
<td>PWM_INTERRUPT</td>
<td>PWM_INTERRUPT_t</td>
</tr>
<tr>
<td>enum</td>
<td></td>
<td>The type identifies the timer interrupts.</td>
</tr>
<tr>
<td>typedef</td>
<td>PWM_OUTPUT_PASSIVE_LEVEL</td>
<td>PWM_OUTPUT_PASSIVE_LEVEL_t</td>
</tr>
<tr>
<td>enum</td>
<td></td>
<td>The type identifies the timer interrupts.</td>
</tr>
<tr>
<td>typedef</td>
<td>PWM_STATUS</td>
<td>PWM_STATUS_t</td>
</tr>
<tr>
<td>enum</td>
<td></td>
<td>The type identifies App state.</td>
</tr>
<tr>
<td>typedef</td>
<td>PWM_ERROR_CODES</td>
<td>PWM_ERROR_CODES_t</td>
</tr>
<tr>
<td>enum</td>
<td></td>
<td>The type identifies the App Error Codes.</td>
</tr>
</tbody>
</table>
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.

```c
enum PWM_OUTPUT_PASSIVE_LEVEL
{
    PWM_OUTPUT_PASSIVE_LEVEL_LOW, // Passive level = Low
    PWM_OUTPUT_PASSIVE_LEVEL_HIGH  // Passive level = High
};
```

Definition at line 135 of file PWM.h.

```c
enum PWM_STATUS
{
    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
};
```
<table>
<thead>
<tr>
<th><strong>PWM_STATUS_RUNNING</strong></th>
<th>mode</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PWM_STATUS_STOPPED</strong></td>
<td>CCU slice timer is stopped</td>
</tr>
</tbody>
</table>

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

Data structures
## Data Structures

<table>
<thead>
<tr>
<th>struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>Initialization parameters of the PWM App. More...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>typedef struct</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>PWM_t</td>
</tr>
<tr>
<td>Initialization parameters of the PWM App.</td>
</tr>
</tbody>
</table>
# PWM

## Methods

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Function Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAVE_APP_VERSION_t</td>
<td>PWM_GetAppVersion (void)</td>
<td>Get PWM APP version.</td>
</tr>
<tr>
<td>PWM_STATUS_t</td>
<td>PWM_Init (PWM_t *const handle_ptr)</td>
<td>Initializes the PWM APP.</td>
</tr>
<tr>
<td>void</td>
<td>PWM_Start (PWM_t *const handle_ptr)</td>
<td>Starts the PWM generation.</td>
</tr>
<tr>
<td>void</td>
<td>PWM_Stop (PWM_t *const handle_ptr)</td>
<td>Stops the PWM generation.</td>
</tr>
<tr>
<td>PWM_STATUS_t</td>
<td>PWM_SetFreq (PWM_t *const handle_ptr, uint32_t pwm_freq_hz)</td>
<td>Configures the PWM Frequency.</td>
</tr>
<tr>
<td>PWM_STATUS_t</td>
<td>PWM_SetFreqAndDutyCycle (PWM_t *const handle_ptr, uint32_t pwm_freq_hz, uint32_t duty_cycle)</td>
<td>Configures the PWM Frequency and duty cycle.</td>
</tr>
<tr>
<td>void</td>
<td>PWM_ClearEvent (PWM_t *const handle_ptr, PWM_INTERRUPT_t pwm_interrupt)</td>
<td>Clears the PWM related interrupt.</td>
</tr>
<tr>
<td>bool</td>
<td>PWM_GetInterruptStatus (PWM_t *const handle_ptr, PWM_INTERRUPT_t pwm_interrupt)</td>
<td>Gets the corresponding interrupt status.</td>
</tr>
<tr>
<td>bool</td>
<td>PWM_GetTimerStatus (PWM_t *const handle_ptr)</td>
<td>Gets the corresponding timer status.</td>
</tr>
<tr>
<td></td>
<td>PWM_SetDutyCycle (PWM_t *const handle_ptr)</td>
<td></td>
</tr>
</tbody>
</table>
**Methods**

<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PWM_SetActiveLevel</td>
<td>PWM_STATUS_t handle_ptr, uint32_t duty_cycle</td>
<td>Configure the PWM duty cycle.</td>
</tr>
<tr>
<td>void PWM_SetPassiveLevel</td>
<td>(PWM_t *const handle_ptr, PWM_OUTPUT_PASSIVE_LEVEL_t pwm_output_passive_level)</td>
<td>Configure the passive level of the PWM output waveform.</td>
</tr>
<tr>
<td>PWM_SetPeriodMatchValue</td>
<td>PWM_STATUS_t *const handle_ptr, uint32_t period_match_value</td>
<td>Loads the required period match value.</td>
</tr>
</tbody>
</table>
Function Documentation

```c
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 window and connect the event_compare_match and event_period_match to
  // the 2 different INTERRUPT APPs. Give the handler for the compare match interrupt as PWM_compare_match_interrupt
  // and the handler for the period match interrupt as PWM_period_match_interrupt

#include <DAVE.h>

uint32_t period_count;
void PWM_period_match_interrupt(void)
{
    static uint32_t frequency = 1000; // start frequency 1Khz
    PWM_ClearEvent(&PWM_0, PWM_INTERRUPT_PERIODMATCH);
    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_COMPAREMATCH);
}

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:  

```c
#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) {
```
Definition at line 734 of file PWM.c.

```c
bool PWM_GetInterruptStatus ( PWM_t *const handle_ptr, PWM_INTERRUPT_t pwm_interrupt )
```

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 occurred else returns false.

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

```c
#include <DAVE.h>
uint32_t period_count;
```
void PWM_compare_period_match_interrupt(void) {
    if (PWM_GetInterruptStatus(&PWM_0, PWM_INTERRUPT_PERIODMATCH)) {
        PWM_Stop(&PWM_0); // A single shot PWM generated
        PWM_ClearEvent(&PWM_0, PWM_INTERRUPT_COMPAREMATCH);
    }
}

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().

```c
#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_Init().
    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);
    }
```
```c
// 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.

```c
PWM_STATUS_t PWM_SetFreq(PWM_t *const handle_ptr,
                           uint32_t      PwmFreqHz)
```

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_0,100000);

    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 handle_ptr,
                                      uint32_t PwmFreqHz,
                                      uint32_t duty_cycle )

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.

```c
#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.

```c
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.

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_LEVEL 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_LEVEL 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_PASSIVE_LEVEL_HIGH);
    for (Counter = 0; Counter <= 0xFFFF; Counter++);
    PWM_SetPassiveLevel(&PWM_0, PWM_OUTPUT_PASSIVE_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 handle_ptr, uint32_t period_match_value)

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.

```c
#include <DAVE.h>

int main(void)
{
    uint32_t Counter;
    DAVE_Init();

    PWM_SetFreq(&PWM_1,1U); // Set the required frequency as 1Hz
    for(Counter = 0;Counter <=0xFFFF;Counter++);

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

```c
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.

```c
#include <DAVE.h>

int main(void)
{
}
```
DAVE_Init();
   //This needs to be called if "Start after initialization" 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.
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)

```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 100%
    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_INTERRUPTCOMPAREMATCH);
}

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 while loop below can be replaced with an user error handler */
        XMC_DEBUG(("DAVE Apps initialization failed with 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

### Release History

<table>
<thead>
<tr>
<th>Release History</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
PWM

Data Structures

Here are the data structures with brief descriptions:

| PWM_HandleType | Initialization parameters of the PWM App |
## PWM

### PWM_HandleType

**Struct Reference**

**Data structures**

---

<table>
<thead>
<tr>
<th>Home</th>
<th>Data Structures</th>
<th>Data Structure Index</th>
<th>Data Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PWM</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PWM_HandleType</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Struct Reference</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Data structures</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Detailed Description

Initialization parameters of the PWM App.

Definition at line 184 of file PWM.h.

#include <PWM.h>
Data Fields

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMC_GPIO_PORT_t *const</td>
<td>gpio_out_port</td>
</tr>
<tr>
<td>XMC_GPIO_CONFIG_t *const</td>
<td>gpio_out_config</td>
</tr>
<tr>
<td>uint32_t</td>
<td>compare_value</td>
</tr>
<tr>
<td>uint32_t</td>
<td>period_value</td>
</tr>
<tr>
<td>uint32_t</td>
<td>duty_cycle</td>
</tr>
<tr>
<td>uint32_t</td>
<td>shadow_mask</td>
</tr>
<tr>
<td>const PWM_TIMER_SLICE_t</td>
<td>timer_type</td>
</tr>
<tr>
<td>PWM_STATUS_t</td>
<td>state</td>
</tr>
<tr>
<td>const uint8_t</td>
<td>gpio_out_pin</td>
</tr>
<tr>
<td>const uint8_t</td>
<td>kernel_number</td>
</tr>
<tr>
<td>const uint8_t</td>
<td>slice_number</td>
</tr>
<tr>
<td>const bool</td>
<td>start_control</td>
</tr>
<tr>
<td>const bool</td>
<td>period_match_enable</td>
</tr>
<tr>
<td>const bool</td>
<td>compare_match_enable</td>
</tr>
<tr>
<td>const GLOBAL_CCU4_t *const</td>
<td>global_ccu4_handle</td>
</tr>
<tr>
<td>const GLOBAL_CCU8_t *const</td>
<td>global_ccu8_handle</td>
</tr>
<tr>
<td>const XMC_CCU4_MODULE_t *const</td>
<td>ccu4_kernel_ptr</td>
</tr>
<tr>
<td>const XMC_CCU8_MODULE_t *const</td>
<td>ccu8_kernel_ptr</td>
</tr>
<tr>
<td>const XMC_CCU4_SLICE_t *const</td>
<td>ccu4_slice_ptr</td>
</tr>
<tr>
<td>const XMC_CCU8_SLICE_t *const</td>
<td>ccu8_slice_ptr</td>
</tr>
<tr>
<td>const XMC_CCU4_SLICE_COMPARE_CONFIG_t *const</td>
<td>ccu4_slice_config_ptr</td>
</tr>
<tr>
<td>const XMC_CCU8_SLICE_COMPARE_CONFIG_t *const</td>
<td>ccu8_slice_config_ptr</td>
</tr>
<tr>
<td>const XMC_CCU4_SLICE_SR_ID_t</td>
<td>ccu4_slice_period_match_node</td>
</tr>
<tr>
<td>const XMC_CCU8_SLICE_SR_ID_t</td>
<td>ccu8_slice_period_match_node</td>
</tr>
<tr>
<td>const XMC_CCU4_SLICE_SR_ID_t</td>
<td>ccu4_slice_compare_match_node</td>
</tr>
<tr>
<td>const XMC_CCU8_SLICE_SR_ID_t</td>
<td>ccu8_slice_compare_match_node</td>
</tr>
</tbody>
</table>
## 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_compare_match_node**

SR node line for period match

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

**const XMC_CCU4_SLICE_COMPARE_CONFIG_t**\* const **PWM_HandleType::ccu4_slice_config_ptr**

Pointer to CCU4 configuration handle

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

**const XMC_CCU4_SLICE_SR_ID_t** **PWM_HandleType::ccu4_slice_period_match_node**

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_ptr

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_compare_match_node

SR node line for period match

Definition at line 258 of file PWM.h.

const XMC_CCU8_SLICE_COMPARE_CONFIG_t* const PWM_HandleType::ccu8_slice_config_ptr

Pointer to CCU8 configuration handle

Definition at line 222 of file PWM.h.

const XMC_CCU8_SLICE_SR_ID_t PWM_HandleType::ccu8_slice_period_match_node

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**
<table>
<thead>
<tr>
<th>Declaration</th>
<th>Description</th>
<th>Line</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GLOBAL_CCU4 App handle</code></td>
<td>Definition at line 189 of file PWM.h.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>GLOBAL_CCU8_t* const PWM_HandleType::global_ccu8_handle</code></td>
<td>GGLOBAL_CCU8 App handle</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>const XMC_GPIO_CONFIG_t* const PWM_HandleType::gpio_out_config</code></td>
<td>Holds the pin configuration for the PWM output</td>
<td>228</td>
<td>PWM.h.</td>
</tr>
<tr>
<td><code>const uint8_t PWM_HandleType::gpio_out_pin</code></td>
<td>Holds the pin number for the PWM output</td>
<td>262</td>
<td>PWM.h.</td>
</tr>
<tr>
<td><code>XMC_GPIO_PORT_t* const PWM_HandleType::gpio_out_port</code></td>
<td>Holds the port number for the PWM output</td>
<td>226</td>
<td>PWM.h.</td>
</tr>
<tr>
<td><code>const uint8_t PWM_HandleType::kernel_number</code></td>
<td>Indicates the CCUx kernel number</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PWM_STATUS_t PWM_HandleType::state</strong></td>
<td>The current state of the PWM App instance. Definition at line 268 of file PWM.h. Referred by PWM_SetPeriodMatchValue().</td>
</tr>
<tr>
<td><strong>const PWM_TIMER_SLICE_t PWM_HandleType::timer_type</strong></td>
<td>Type of CCU selected for the PWM generation. Definition at line 240 of file PWM.h. Referred by PWM_ClearEvent(), PWM_GetInterruptStatus(), PWM_GetTimerStatus(), PWM_Init(), PWM_SetDutyCycle(), PWM_SetFreq(), PWM_SetFreqAndDutyCycle(), PWM_SetPassiveLevel(), PWM_SetPeriodMatchValue(), PWM_Start(), and PWM_Stop().</td>
</tr>
</tbody>
</table>

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

- PWM.h
# PWM

## Data Structure Index

<table>
<thead>
<tr>
<th>P</th>
<th>PWM_HandleType</th>
</tr>
</thead>
</table>

*Note: This table lists the PWM_HandleType as P.*
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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccu4_kernel_ptr</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>ccu4_slice_compare_match_node</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>ccu4_slice_config_ptr</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>ccu4_slice_period_match_node</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>ccu4_slice_ptr</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>ccu8_kernel_ptr</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>ccu8_slice_compare_match_node</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>ccu8_slice_config_ptr</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>ccu8_slice_period_match_node</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>ccu8_slice_ptr</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>compare_match_enable</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>compare_value</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>duty_cycle</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>global_ccu4_handle</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>global_ccu8_handle</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>gpio_out_config</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>gpio_out_pin</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>gpio_out_port</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>kernel_number</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>period_match_enable</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>period_value</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>shadow_mask</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>slice_number</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>start_control</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>state</td>
<td>PWM_HandleType</td>
</tr>
<tr>
<td>timer_type</td>
<td>PWM_HandleType</td>
</tr>
</tbody>
</table>
PWM

File List

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

- **PWM.c** [code]
- **PWM.h** [code]
### PWM

<table>
<thead>
<tr>
<th>Home</th>
<th>File List</th>
<th>Globals</th>
<th>Functions</th>
</tr>
</thead>
</table>

### 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAVE_APP_VERSION_t</td>
<td><code>PWM_GetAppVersion (void)</code></td>
<td>Get PWM APP version.</td>
</tr>
<tr>
<td>PWM_STATUS_t</td>
<td><code>PWM_Init (PWM_t *const handle_ptr)</code></td>
<td>Initializes the PWM APP.</td>
</tr>
<tr>
<td>void</td>
<td><code>PWM_Start (PWM_t *const handle_ptr)</code></td>
<td>Starts the PWM generation.</td>
</tr>
<tr>
<td>void</td>
<td><code>PWM_Stop (PWM_t *const handle_ptr)</code></td>
<td>Stops the PWM generation.</td>
</tr>
<tr>
<td>PWM_STATUS_t</td>
<td><code>PWM_SetDutyCycle (PWM_t *const handle_ptr, uint32_t duty_cycle)</code></td>
<td>Configure the PWM duty cycle.</td>
</tr>
<tr>
<td>PWM_STATUS_t</td>
<td><code>PWM_SetFreq (PWM_t *const handle_ptr, uint32_t pwm_freq_hz)</code></td>
<td>Configures the PWM Frequency.</td>
</tr>
<tr>
<td>PWM_STATUS_t</td>
<td><code>PWM_SetFreqAndDutyCycle (PWM_t *const handle_ptr, uint32_t pwm_freq_hz, uint32_t duty_cycle)</code></td>
<td>Configures the PWM Frequency and duty cycle.</td>
</tr>
<tr>
<td>bool</td>
<td><code>PWM_GetTimerStatus (PWM_t *const handle_ptr)</code></td>
<td>Gets the corresponding timer status.</td>
</tr>
<tr>
<td>bool</td>
<td><code>PWM_GetInterruptStatus (PWM_t *const handle_ptr, PWM_INTERRUPT_t pwm_interrupt)</code></td>
<td>Gets the corresponding interrupt status.</td>
</tr>
<tr>
<td>void</td>
<td><code>PWM_ClearEvent (PWM_t *const handle_ptr, PWM_INTERRUPT_t pwm_interrupt)</code></td>
<td>Clears the PWM related interrupt.</td>
</tr>
<tr>
<td></td>
<td><code>PWM_SetPassiveLevel (PWM_t *const</code></td>
<td></td>
</tr>
<tr>
<td>Function Name</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>void handle_ptr, \nPWM_OUTPUT_PASSIVE_LEVEL_t \npwm_output_passive_level)</td>
<td>Configure the passive level of the PWM output waveform.</td>
<td></td>
</tr>
<tr>
<td>PWM_STATUS_t \nPWM_SetPeriodMatchValue (PWM_t \n*const handle_ptr, uint32_t \nperiod_match_value)</td>
<td>Loads the required period match value.</td>
<td></td>
</tr>
</tbody>
</table>
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 window and connect the event_compare_match and event_period_match to
  // the 2 different INTERRUPT APPs. Give the handler for the compare match interrupt as PWM_compare_match_interrupt
  // and the handler for the period match interrupt as PWM_period_match_interrupt
  #include <DAVE.h>
  uint32_t period_count;

void PWM_period_match_interrupt(void)
{
    static uint32_t frequency = 1000; // start frequency 1Khz
    PWM_ClearEvent(&PWM_0, PWM_INTERRUPT_PERIODMATCH);
    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_COMPAREMATCH);
}

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`.

```c
bool PWM_GetInterruptStatus ( PWM_t *const handle_ptr, PWM_INTERRUPT_t pwm_interrupt )
```

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 occurred else returns false.

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

```c
#include <DAVE.h>
uint32_t period_count;

void PWM_compare_period_match_interrupt(void) {
```

// Drag 1 INTERRUPT APP into the project. In the PWM APPs UI editor enable compare match event and period
// match event. Goto the Signal connectivity window and connect the event_compare_match and event_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_INTERRUPT_PERIODMATCH))
 {
    PWM_Stop(&PWM_0); // A single shot PWM generated
 }
 PWM_ClearEvent(&PWM_0, PWM_INTERRUPTCOMPAREMATCH);

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().

```c
#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.

```c
#include <DAVE.h>

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

Definition at line 747 of file PWM.c.

References PWM_STATUS_FAILURE, and PWM_HandleType::timer_type.

```c
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.

```c
#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, 1000000);
  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);
}  
```
```c
while(1);
return 0;
}
```

Definition at line 841 of file PWM.c.

References PWM_STATUS_FAILURE, and PWM_HandleType::timer_type.

```c
PWM_STATUS_t PWM_SetFreqAndDutyCycle ( PWM_t *const handle_ptr, uint32_t pwmFreqHz, uint32_t duty_cycle )
```

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.

```c
#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);
```
Definition at line 866 of file PWM.c.

References PWM_STATUS_FAILURE, and PWM_HandleType::timer_type.

```c
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.

**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_LEVEL_HIGH` 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_LEVEL_LOW` is selected the output of the PWM will be low before the compare match occurs after that it would be set to high.

```c
#include <DAVE.h>

int main(void)
{
    return 0;
}
```
uint32_t Counter;

DAVE_Init();

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

PWM_SetPassiveLevel(&PWM_0, PWM_OUTPUT_PASSIVE_LEVEL_HIGH);

for(Counter = 0;Counter <=0xFFFF;Counter++);
PWM_SetPassiveLevel(&PWM_0, PWM_OUTPUT_PASSIVE_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 handle_ptr,
                                       uint32_t period_match_value)

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 \texttt{period\_match\_value}) to maintain the duty cycle.

```c
#include <DAVE.h>

int main(void)
{
    uint32_t Counter;
    DAVE_Init();
    PWM_SetFreq(&PWM_1,1U); // Set the required frequency as 1Hz
    for(Counter = 0;Counter <=0xFFFF;Counter++);
    PWM_SetPeriodMatchValue(&PWM_0,PWM_0.period\_value/2); // 2Hz
    for(Counter = 0;Counter <=0xFFFF;Counter++);
    PWM_SetPeriodMatchValue(&PWM_0,PWM_0.period\_value*2); //1Hz
    for(Counter = 0;Counter <=0xFFFF;Counter++);
    while(1);
    return 0;
}
```

Definition at line 995 of file \texttt{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`.

```c
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.

```c
#include <DAVE.h>

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

```c
#include <DAVE.h>

int main(void)
{
    DAVE_Init();

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

    return 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

<table>
<thead>
<tr>
<th>Home</th>
<th>File List</th>
<th>Globals</th>
<th>Data Structures</th>
</tr>
</thead>
</table>

**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

<table>
<thead>
<tr>
<th>struct</th>
<th>PWM_HandleType</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization parameters of the PWM App.</td>
<td></td>
</tr>
</tbody>
</table>

[More...]
### Typedefs

typedef struct `PWM_HandleType` `PWM_t`  
Initialization parameters of the PWM App.
## Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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 handle_ptr)  
Gets the corresponding timer status.  |
| PWM_STATUS_t | **PWM_SetDutyCycle** (PWM_t *const handle_ptr, uint32_t duty_cycle)  
Configure the PWM duty cycle.  |
| PWM_STATUS_t | **PWM_SetPassiveLevel** (PWM_t handle_ptr)  
Sets the PWM passive level.  |
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void handle_ptr,PWM_OUTPUT_PASSIVE_LEVEL_t pwm_output_passive_level)</td>
<td>Configure the passive level of the PWM output waveform.</td>
</tr>
<tr>
<td>PWM_STATUS_t PWM_SetPeriodMatchValue (PWM_t *const handle_ptr, uint32_t period_match_value)</td>
<td>Loads the required period match value.</td>
</tr>
<tr>
<td>enum PWM_TIMER_SLICE</td>
<td>The type identifies the CCU4 or CCU8 timer selected.</td>
</tr>
<tr>
<td>enum PWM_TIMER_STATUS</td>
<td>The type identifies the timer status.</td>
</tr>
<tr>
<td>enum PWM_INTERRUPT { PWM_INTERRUPT_PERIODMATCH, PWM_INTERRUPT_COMPAREMATCH, 0U, PWM_INTERRUPT_COMPAREMATCH, 2U }</td>
<td>The type identifies the timer interrupts.</td>
</tr>
<tr>
<td>enum PWM_OUTPUT_PASSIVE_LEVEL { PWM_OUTPUT_PASSIVE_LEVEL_LOW = 0, PWM_OUTPUT_PASSIVE_LEVEL_HIGH }</td>
<td>The type identifies the timer interrupt type.</td>
</tr>
<tr>
<td>enum PWM_STATUS { PWM_STATUS_SUCCESS = 0, PWM_STATUS_FAILURE, PWM_STATUS_UNINITIALIZED, PWM_STATUS_RUNNING, PWM_STATUS_STOPPED }</td>
<td>The type identifies the App state.</td>
</tr>
<tr>
<td>Type</td>
<td>Definition</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td><code>enum PWM_ERROR_CODES</code></td>
<td><code>{ PWM_OPER_NOT_ALLOWED_ERROR = 1, PWM_INVALID_PARAM_ERROR = 2 }</code></td>
</tr>
<tr>
<td><code>typedef enum PWM_TIMER_SLICE PWM_TIMER_SLICE_t</code></td>
<td></td>
</tr>
<tr>
<td><code>typedef enum PWM_TIMER_STATUS PWM_TIMER_STATUS_t</code></td>
<td></td>
</tr>
<tr>
<td><code>typedef enum PWM_INTERRUPT PWM_INTERRUPT_t</code></td>
<td></td>
</tr>
<tr>
<td><code>typedef enum PWM_OUTPUT_PASSIVE_LEVEL PWM_OUTPUT_PASSIVE_LEVEL_t</code></td>
<td></td>
</tr>
<tr>
<td><code>typedef enum PWM_STATUS PWM_STATUS_t</code></td>
<td></td>
</tr>
<tr>
<td><code>typedef enum PWM_ERROR_CODES PWM_ERROR_CODES_t</code></td>
<td></td>
</tr>
</tbody>
</table>

Go to the source code of this file.
Here is a list of all documented functions, variables, defines, enums, and typedefs with links to the documentation:

- **p** -

  - `PWM_ClearEvent()`: `PWM.c`, `PWM.h`
  - `PWM_ERROR_CODES`: `PWM.h`
  - `PWM_ERROR_CODES_t`: `PWM.h`
  - `PWM_GetAppVersion()`: `PWM.c`, `PWM.h`
  - `PWM_GetInterruptStatus()`: `PWM.h`, `PWM.c`
  - `PWM_GetTimerStatus()`: `PWM.c`, `PWM.h`
  - `PWM_Init()`: `PWM.c`, `PWM.h`
  - `PWM_INTERRUPT`: `PWM.h`
  - `PWM_INTERRUPT_COMPAREMATCH`: `PWM.h`
  - `PWM_INTERRUPT_PERIODMATCH`: `PWM.h`
  - `PWM_INTERRUPT_t`: `PWM.h`
  - `PWM_INVALID_PARAM_ERROR`: `PWM.h`
  - `PWM_OPER_NOT_ALLOWED_ERROR`: `PWM.h`
  - `PWM_OUTPUT_PASSIVE_LEVEL`: `PWM.h`
  - `PWM_OUTPUT_PASSIVE_LEVEL_HIGH`: `PWM.h`
  - `PWM_OUTPUT_PASSIVE_LEVEL_LOW`: `PWM.h`
  - `PWM_OUTPUT_PASSIVE_LEVEL_t`: `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.h`, `PWM.c`
  - `PWM_Start()`: `PWM.c`, `PWM.h`
- 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

Go to the documentation of this file.

```c
#ifndef PWM_H
#define PWM_H

/*************************
 * HEADER FILES
 *******************************************
**************************************************
#include <xmc_gpio.h>
#include "pwm_conf.h"
#include <DAVE_common.h>

/****************************
 * MACROS
 *******************************************
**************************************************
#if (!((XMC_LIB_MAJOR_VERSION == 2U) &&
(XMC_LIB_MINOR_VERSION >= 0U) &&
(XMC_LIB_PATCH_VERSION >= 0U)))
#error "PWM requires XMC Peripheral Library"
```
v2.0.0 or higher"
00094  #endif
00095
00096  #define PWM_MAX_TIMER_COUNT  (65535U)
00098  /***************************************************************************************
00099  *********************************************************
00100  *****************************************************************************/
00099  * ENUMS
00100  *****************************************************************************************/
00099  
00101  typedef enum PWM_TIMER_SLICE
00102  
00103  
00104  
00105  
00106  } PWM_TIMER_SLICE_t;
00107
00108  typedef enum PWM_TIMER_STATUS
00109  
00110  
00111  
00112  } PWM_TIMER_STATUS_t;
00113
00114  typedef enum PWM_INTERRUPT
00115  
00116  
00117  
00118  } PWM_INTERRUPT_t;
00119
00120  typedef enum PWM_OUTPUT_PASSIVE_LEVEL
00121  
00122  
00123  } PWM_OUTPUT_PASSIVE_LEVEL_t;
typedef enum PWM_STATUS {
    PWM_STATUS_SUCCESS = 0,
    PWM_STATUS_FAILURE,
    PWM_STATUS_UNINITIALIZED,
    PWM_STATUS_RUNNING,
    PWM_STATUS_STOPPED,
} PWM_STATUS_t;

typedef enum PWM_ERROR_CODES {
    PWM_OPER_NOT_ALLOWED_ERROR = 1,
    PWM_INVALID_PARAM_ERROR
} PWM_ERROR_CODES_t;

#if defined(__CC_ARM)
#pragma push
#pragma anon_unions
#elif defined(__TASKING__)
#pragma warning 586
#endif

typedef struct PWM_HandleType {
    union {
        /*Anonymous structure/union guard start*/
        #ifdef PWM_SLICE_USED_CCU4
            /*Anonymous structure/union guard end*/
        #endif
    }
}
GLOBAL_CCU4_t *const global_ccu4_handle;
#ifdef PWM_SLICE_USED_CCU8
GLOBAL_CCU8_t *const global_ccu8_handle;
#endif
}

union {
#ifdef PWM_SLICE_USED_CCU4
XMC_CCU4_MODULE_t *const ccu4_kernel_ptr;
#endif
#ifdef PWM_SLICE_USED_CCU8
XMC_CCU8_MODULE_t *const ccu8_kernel_ptr;
#endif
};

union {
#ifdef PWM_SLICE_USED_CCU4
XMC_CCU4_SLICE_t *const ccu4_slice_ptr;
#endif
#ifdef PWM_SLICE_USED_CCU8
XMC_CCU8_SLICE_t *const ccu8_slice_ptr;
#endif
};

union {
#ifdef PWM_SLICE_USED_CCU4
const XMC_CCU4_SLICE_COMPARE_CONFIG_t *const ccu4_slice_config_ptr;
#endif
};
#ifdef PWM_SLICE_USED_CCU8

const XMC_CCU8_SLICE_COMPARE_CONFIG_t *const ccu8_slice_config_ptr;
#endif

};

XMC_GPIO_PORT_t *const io_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;

union {

#ifdef PWM_SLICE_USED_CCU4

const XMC_CCU4_SLICE_SR_ID_t ccu4_slice_period_match_node;
#endif

#ifdef PWM_SLICE_USED_CCU8

const XMC_CCU8_SLICE_SR_ID_t ccu8_slice_period_match_node;
#endif

};

union {

#ifdef PWM_SLICE_USED_CCU4

const XMC_CCU4_SLICE_SR_ID_t ccu4_slice_compare_match_node;
#endif

#ifdef PWM_SLICE_USED_CCU8

const XMC_CCU8_SLICE_SR_ID_t ccu8_slice_compare_match_node;
#endif

};

}
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;

} PWM_t;

/*Anonymous structure/union guard end*/
#endif
#if defined(__CC_ARM)
#pragma pop
#elif defined(__TASKING__)
#pragma warning restore
#endif
#endif

/*API Prototypes*/

DAVE_APP_VERSION_t PWM_GetAppVersion(void);

PWM_STATUS_t PWM_Init(PWM_t *const handle_ptr);
void PWM_Start(PWM_t *const handle_ptr);
void PWM_Stop(PWM_t *const handle_ptr);
PWM_STATUS_t PWM_SetFreq(PWM_t *const handle_ptr, uint32_t pwm_freq_hz);

PWM_STATUS_t PWM_SetFreqAndDutyCycle(PWM_t *const handle_ptr, uint32_t pwm_freq_hz, uint32_t duty_cycle);

void PWM_ClearEvent(PWM_t *const handle_ptr, PWM_INTERRUPT_t pwm_interrupt);

bool PWM_GetInterruptStatus(PWM_t *const handle_ptr, PWM_INTERRUPT_t pwm_interrupt);

bool PWM_GetTimerStatus(PWM_t *const handle_ptr);

PWM_STATUS_t PWM_SetDutyCycle(PWM_t *const handle_ptr, uint32_t duty_cycle);

void PWM_SetPassiveLevel(PWM_t *const handle_ptr, PWM_OUTPUT_PASSIVE_LEVEL_t pwm_output_passive_level);

PWM_STATUS_t PWM_SetPeriodMatchValue(PWM_t *const handle_ptr, uint32_t period_match_value);

#include "PWM_Extern.h"

#ifdef __cplusplus
}
#endif
/* PWM_H */
Go to the documentation of this file.

```c
#include "pwm.h"
```
#ifdef PWM_SLICE_USED_CCU4
/*Initialize the App and XMC_CCU4 slice. */
PWM_STATUS_t PWM_lCCU4_Init(PWM_t *const handle_ptr);

/*Initialize interrupts*/
void PWM_lCCU4_ConfigInterrupts(PWM_t *const handle_ptr);

/*Starts the CCU4 slice. */
void PWM_lCCU4_Start(PWM_t *const handle_ptr);

/*Stops the CCU4 slice. */
void PWM_lCCU4_Stop(PWM_t *const handle_ptr);

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

/*Sets the frequency for CCU4 slice. */
PWM_STATUS_t PWM_lCCU4_SetFreq(PWM_t *const handle_ptr, uint32_t pwm_freq_hz);

/*Sets the frequency and duty cycle for CCU4 slice. */
PWM_STATUS_t PWM_lCCU4_SetFreqAndDutyCycle(PWM_t *const handle_ptr, uint32_t pwm_freq_hz, uint32_t duty_cycle);
#endif
PWM_STATUS_t PWM_lCCU8_Init(PWM_t *const handle_ptr);

/*Initialize interrupts*/
void PWM_lCCU8_ConfigInterrupts(PWM_t *const handle_ptr);

/*Starts the CCU8 slice. */
void PWM_lCCU8_Start(PWM_t *const handle_ptr);

/*Stops the CCU8 slice. */
void PWM_lCCU8_Stop(PWM_t *const handle_ptr);

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

/*Sets the frequency for CCU8 slice. */
PWM_STATUS_t PWM_lCCU8_SetFreq(PWM_t *const handle_ptr, uint32_t pwm_freq_hz);

/*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);

#endif

#ifdef PWM_SLICE_USED_CCU4
/*Initialize the APP and CCU4 slice. */
PWM_STATUS_t PWM_lCCU4_Init(PWM_t *const handle_ptr)
{
    PWM_STATUS_t status = PWM_STATUS_FAILURE;

```
XMC_ASSERT("PWM_lCCU4_Init:Invalid handle_ptr", (handle_ptr != NULL))

if (PWM_STATUS_UNINITIALIZED == handle_ptr->state) {
    /* Initialize consumed Apps */
    status = (PWM_STATUS_t)GLOBAL_CCU4_Init(handle_ptr->global_ccu4_handle);

    /*Initialize CCU4 slice */
    if (PWM_STATUS_SUCCESS == status) /*check GLOBAL_CCU4_Init status*/ {
        XMC_DEBUG("PWM_lCCU4_Init:Initialzing Slice")
        XMC_CCU4_SLICE_CompareInit(handle_ptr->ccu4_slice_ptr, handle_ptr->ccu4_slice_config_ptr);

        /* Set the period and compare register values */
        XMC_CCU4_SLICE_SetTimerPeriodMatch(handle_ptr->ccu4_slice_ptr, (uint16_t)handle_ptr->period_value);

        XMC_CCU4_SLICE_SetTimerCompareMatch(handle_ptr->ccu4_slice_ptr, (uint16_t)handle_ptr->compare_value);

        XMC_CCU4_EnableShadowTransfer(handle_ptr->ccu4_kernel_ptr, handle_ptr->shadow_mask);

        /* Initialize interrupts */
PWM_lCCU4_ConfigInterrupts(handle_ptr);

XMC_GPIO_Init(handle_ptr->gpio_out_port, handle_ptr->gpio_out_pin, handle_ptr->gpio_out_config);

handle_ptr->state = PWM_STATUS_SUCCESS;

/* Start the PWM generation if start at initialization is enabled */
if ((bool) true == handle_ptr->start_control)
{
    PWM_Start(handle_ptr);
}

status = PWM_STATUS_SUCCESS;
}
else
{
    handle_ptr->state = PWM_STATUS_UNINITIALIZED;
}

return (status);

} /* end of PWM_lCCU4_Init() api */

/* Initialize interrupts */
void PWM_lCCU4_ConfigInterrupts(PWM_t *const handle_ptr)
{

if ((bool) true == handle_ptr->period_match_enable) {
    XMC_DEBUG("PWM_IICC4_ConfigInterrupts:period match enable")
    XMC_CCU4_SLICE_EnableEvent(handle_ptr->ccu4_slice_ptr, XMC_CCU4_SLICE_IRQ_ID_PERIOD_MATCH);
    /* Bind event to Service Request Node to period match event*/
    XMC_CCU4_SLICE_SetInterruptNode(handle_ptr->ccu4_slice_ptr, XMC_CCU4_SLICE_IRQ_ID_PERIOD_MATCH, handle_ptr->ccu4_slice_period_match_node);
}

if ((bool) true == handle_ptr->compare_match_enable) {
    XMC_DEBUG("PWM_IICC4_ConfigInterrupts:compare match enable")
    XMC_CCU4_SLICE_EnableEvent(handle_ptr->ccu4_slice_ptr, XMC_CCU4_SLICE_IRQ_ID_COMPARE_MATCH_UP);
    /* Bind event to Service Request Node to compare match event */
    XMC_CCU4_SLICE_SetInterruptNode(handle_ptr->ccu4_slice_ptr, XMC_CCU4_SLICE_IRQ_ID_COMPARE_MATCH_UP, handle_ptr->ccu4_slice_compare_match_node);
}
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")
    }
}

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));
    }
  }
}
handle_ptr->compare_value = compare;
handle_ptr->duty_cycle = duty_cycle;

XMC_CCU4_SLICE_SetTimerCompareMatch(handle_ptr->ccu4_slice_ptr, (uint16_t)compare);
XMC_CCU4_EnableShadowTransfer(handle_ptr->ccu4_kernel_ptr, handle_ptr->shadow_mask);
status = PWM_STATUS_SUCCESS;
}
}

XMC_DEBUG("PWM_lCCU4_SetDutyCycle:dutycycle set")
return (status);
} /* end of PWM_lCCU4_SetDutyCycle() api */

/*Sets the frequency for CCU4 slice. */
PWM_STATUS_t PWM_lCCU4_SetFreq(PWM_t *const handle_ptr, uint32_t pwm_freq_hz)
{
PWM_STATUS_t status;
uint32_t module_freq;
uint8_t prescaler;
uint32_t period_value;
uint32_t compare;

XMC_ASSERT("PWM_lCCU4_SetFreq:Invalid pwm_freq_hz ", (pwm_freq_hz != 0U))
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))
{
    status = PWM_STATUS_SUCCESS;
    /*Get the Module frequency*/
    module_freq = handle_ptr->global_ccu4_handle->module_frequency;
    /*Calculate the prescaler and the period register values.*/
    while (prescaler <= PWM_MAX_PRESCALER)
    {
        period_value = (uint32_t)((uint32_t)module_freq / (uint32_t)pwm_freq_hz) >> (uint32_t)prescaler;
        /*If the prescaler selected is not big enough goto the next prescaler value else come out.*/
        if (period_value <= PWM_MAX_TIMER_COUNT )
        {
            break;
        }
        prescaler++;
    }
    /*Can't set the frequency if the required value is too small or when the required frequency is too large.*/
    if ((prescaler > PWM_MAX_PRESCALER) || ((uint32_t)0 == period_value))
    {
        XMC_DEBUG("PWM_lCCU4_SetFreq:Frequency could not be set")
    }
status = PWM_STATUS_FAILURE;
}
else {
    /*Calculate the new compare values using
    new period values */
    compare = (period_value * (PWM_MAX_DUTY_CYCLE - handle_ptr->duty_cycle))
              / ((uint32_t)100 * PWM_DUTY_CYCLE_SCALE);
    XMC_CCU4_SLICE_SetPrescaler(handle_ptr->ccu4_slice_ptr, prescaler);
    /* The period register is always one count less than calculated.*/
    period_value = period_value - (uint32_t)1;
    XMC_CCU4_SLICE_SetTimerPeriodMatch(handle_ptr->ccu4_slice_ptr,
                                        (uint16_t)(period_value));
    XMC_CCU4_SLICE_SetTimerCompareMatch(handle_ptr->ccu4_slice_ptr,
                                          (uint16_t)compare);
    XMC_CCU4_EnableShadowTransfer(handle_ptr->ccu4_kernel_ptr,
                                   handle_ptr->shadow_mask);
    handle_ptr->compare_value = compare;
    handle_ptr->period_value = period_value;
    XMC_DEBUG("PWM_lCCU4_SetFreq:frequency set")
}
00345    status = PWM_STATUS_FAILURE;
00346    XMC_DEBUG("PWM_lCCU4_SetFreq:Frequency could not be set")
00347    }  
00348
00349    return status;
00350  }  /* end of PWM_lCCU4_SetFreq() api */
00351 */~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
00352 /*Sets the frequency and duty cycle for CCU4 slice. */
00353 PWM_STATUS_t PWM_lCCU4_SetFreqAndDutyCycle(P
00354 WM_t *const handle_ptr, uint32_t pwm_freq_hz, uint
00355 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
00366 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*/

if ((status != PWM_STATUS_UNINITIALIZED) & & ((uint32_t)0 != pwm_freq_hz))
{
    status = PWM_STATUS_SUCCESS;
    /*Get the Module frequency*/
    module_freq = handle_ptr->global_ccu4_handle->module_frequency;

    /*Calculate the prescaler and the period register values.*/
    while (prescaler <= PWM_MAX_PRESCALER)
    {
        period_value = (uint32_t)((uint32_t)module_freq / (uint32_t)pwm_freq_hz) >> (uint32_t)prescaler;
        /*If the prescaler selected is not big enough goto the next prescaler value else come out.*/
        if (period_value <= PWM_MAX_TIMER_COUNT)
        {
            break;
        }
        prescaler++;
    }

    /*Can't set the frequency if the required value is too small or when the required frequency is too large.*/
    if ((prescaler > PWM_MAX_PRESCALER) || (duty_cycle > PWM_MAX_DUTY_CYCLE) || ((uint32_t)0 = = period_value))
    {
        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 using new period values */
00400      compare = (period_value * ((uint32_t)PWM_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 count less than calculated.*/
00405      period_value = period_value - (uint32_t)1;
00406      XMC_CCU4_SLICE_SetTimerPeriodMatch(handle_ptr->ccu4_slice_ptr, (uint16_t)(period_value)) ;
00407
00408      XMC_CCU4_SLICE_SetTimerCompareMatch(handle_ptr->ccu4_slice_ptr, (uint16_t)compare);
00409
00410      XMC_CCU4_EnableShadowTransfer(handle_ptr->ccu4_kernel_ptr, handle_ptr->shadow_mask);
00411
00412      handle_ptr->compare_value = compare;
00413      handle_ptr->period_value = period_value;
00414      handle_ptr->duty_cycle = duty_cycle;
00415      XMC_DEBUG("PWM_1CCU4_SetFreqAndDutyCycle:frequency and duty cycle set")
00416  }
00417  }
00418  else
{  status = PWM_STATUS_FAILURE;
  XMC_DEBUG("PWM_lCCU4_SetFreqAndDutyCycle: Frequency or duty cycle could not be set")
}
return status;
} /* end of PWM_lCCU4_SetFreqAndDutyCycle() api */

#endif /* end of CCU4 function definitions */

#ifdef PWM_SLICE_USED_CCU8

/*Initialize the APP and CCU8 slice. */
PWM_STATUS_t PWM_lCCU8_Init(PWM_t const handle_ptr)
{
  PWM_STATUS_t status = PWM_STATUS_FAILURE;

  if (PWM_STATUS_UNINITIALIZED == handle_ptr->state)
  {
    /* Initialize consumed Apps */
    status = (PWM_STATUS_t)GLOBAL_CCU8_Init(handle_ptr->global_ccu8_handle);

    /*Initialize CCU8 slice */
    if (PWM_STATUS_SUCCESS == status)
    {
      XMC_DEBUG("PWM_lCCU8_Init:Initializing Slice")
      XMC_CCU8_SLICE_CompareInit(handle_ptr->ccu8_slice_ptr, handle_ptr->ccu8_slice_config_ptr);
    }
  }

#endif /* PWM_SLICE_USED_CCU8 */
/* Set the period and compare register values */
XMC_CCU8_SLICE_SetTimerPeriodMatch(handle_ptr->ccu8_slice_ptr, (uint16_t)handle_ptr->period_value);
XMC_CCU8_SLICE_SetTimerCompareMatch(handle_ptr->ccu8_slice_ptr, XMC_CCU8_SLICE_COMPARE_CHANNEL_1, (uint16_t)handle_ptr->compare_value);
XMC_CCU8_EnableShadowTransfer(handle_ptr->ccu8_kernel_ptr, handle_ptr->shadow_mask);
/* Initialize interrupts */
PWM_lCCU8_ConfigInterrupts(handle_ptr);
XMC_GPIO_Init(handle_ptr->gpio_out_port, handle_ptr->gpio_out_pin, handle_ptr->gpio_out_config);
handle_ptr->state = PWM_STATUS_SUCCESS;
/* Start the PWM generation if start at initialization is enabled */
if ((bool) true == handle_ptr->start_control) {
    PWM_Start(handle_ptr);
}
status = PWM_STATUS_SUCCESS;
} else {
    handle_ptr->state = PWM_STATUS_UNINITIALIZED;
}

return(status);
} /* end of PWM_lCCU8_Init() api */

/* Initialize interrupts */
void PWM_lCCU8_ConfigInterrupts(PWM_t *const handle_ptr) {
    if ((bool) true == handle_ptr->period_match_enable) {
        XMC_DEBUG("PWM_lCCU8_ConfigInterrupts:period match event enable")
        XMC_CCU8_SLICE_EnableEvent(handle_ptr->ccu8_slice_ptr, XMC_CCU8_SLICE_IRQ_ID_PERIOD_MATCH);
    }
    /* Bind event to Service Request Node for period match event */
    XMC_CCU8_SLICE_SetInterruptNode(handle_ptr->ccu8_slice_ptr, XMC_CCU8_SLICE_IRQ_ID_PERIOD_MATCH, handle_ptr->ccu8_slice_period_match_node);
}
if ((bool) true == handle_ptr->compare_match_enable) {
    XMC_DEBUG("PWM_lCCU8_ConfigInterrupts:compare match event enable ");
    XMC_CCU8_SLICE_EnableEvent(handle_ptr->ccu8_slice_ptr, XMC_CCU8_SLICE_IRQ_IDCOMPARE_MATCH_UP_CH_1);
    /* Bind event to Service Request Node for compare match event */
    XMC_CCU8_SLICE_SetInterruptNode(handle_ptr->ccu8_slice_ptr, XMC_CCU8_SLICE_IRQ_IDCOMPARE_MATCH_UP_CH_1,
    handle_ptr->ccu8_slice_compare_match_node);
} }

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~*/

/*Starts the CCU8 slice. */
void PWM_lCCU8_Start(PWM_t *const handle_ptr) {
    XMC_ASSERT("PWM_lCCU8_Start:Invalid handle_ptr", (handle_ptr != NULL))
    if ((PWM_STATUS_SUCCESS == handle_ptr->state) || (PWM_STATUS_STOPPED == handle_ptr->state)) {
        /* Clears IDLE mode for the slice */
        XMC_CCU8_EnableClock(handle_ptr->ccu8_kernel_ptr,handle_ptr->slice_number);
        XMC_CCU8_SLICE_StartTimer(handle_ptr->ccu8_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) {

uint32_t period;
uint32_t compare;
PWM_STATUS_t status;

XMC_ASSERT("PWM_lCCU8_SetDutyCycle:Invalid
handle_ptr", (handle_ptr != NULL))

XMC_ASSERT("PWM_lCCU8_SetDutyCycle:Invalid
duty_cycle", (duty_cycle >= 0) &&
(duty_cycle <= PWM_MAX_DUTY_CYCLE)))

status = PWM_STATUS_FAILURE;
if (handle_ptr->state != PWM_STATUS_UNINITIALIZED)
{
    /* Duty cycle needs between 0 and 10000 */
    if (duty_cycle <= PWM_MAX_DUTY_CYCLE)
    {
        period = (uint32_t)handle_ptr->period_value + 1U;
        /* Duty Cycle(symmetric) = (PR-CR1)+1 / period */
        compare = ((period * ((uint32_t)PWM_MAX_DUTY_CYCLE - duty_cycle)) / ((uint32_t)100 * PWM_DUTY_CYCLE_SCALE));

        handle_ptr->compare_value = compare;
        handle_ptr->duty_cycle = duty_cycle;

        XMC_CCU8_SLICE_SetTimerCompareMatch(handle_ptr->ccu8_slice_ptr, XMC_CCU8_SLICE_COMPARE_CHANNEL_1,
(uint16_t)compare);
XMC_CCU8_EnableShadowTransfer(handle_ptr->ccu8_kernel_ptr, handle_ptr->shadow_mask);

status = PWM_STATUS_SUCCESS;
}

XMC_DEBUG("PWM_lCCU8_SetDutyCycle:dutycycle set")
return (status);
} /* end of PWM_lCCU8_SetDutyCycle() api */

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
/*Sets the frequency for CCU8 slice. */
PWM_STATUS_t PWM_lCCU8_SetFreq(PWM_t *const handle_ptr, uint32_t pwm_freq_hz)
{
PWM_STATUS_t status;
uint32_t module_freq;
uint8_t prescaler;
uint32_t period_value;
uint32_t compare;

XMC_ASSERT("PWM_lCCU8_SetFreq:Invalid pwm_freq_hz ", (pwm_freq_hz != 0U))

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))
{
status = PWM_STATUS_SUCCESS;

/*Get the Module frequency*/
module_freq = handle_ptr->global_ccu8_handle->module_frequency;

/*Calculate the prescaler and the period register values.*/
while (prescaler <= PWM_MAX_PRESCALER)
{
    period_value = (uint32_t)((uint32_t)module_freq / (uint32_t)pwm_freq_hz) >> (uint32_t)prescaler;
    /*If the prescaler selected is not big enough goto the next prescaler value else come out.*/
    if (period_value <= PWM_MAX_TIMER_COUNTER)
    {
        break;
    }
    prescaler++;
}

/*Can't set the frequency if the required value is too small or when the required frequency is too large.*/
if ((prescaler > PWM_MAX_PRESCALER) || ((uint32_t)0 == period_value))
{
    XMC_DEBUG("PWM_lCCU8_SetFreq:frequency could not be set")
    status = PWM_STATUS_FAILURE;
}
else
{
    /*Calculate the new compare values usi
Compare = (period_value * (PWM_MAX_DUTY_CYCLE - handle_ptr->duty_cycle)) / ((uint32_t)100 * PWM_DUTY_CYCLE_SCALE);

/* The period register is always one count less than calculated. */
period_value = period_value - (uint32_t)1;
XMC_CCU8_SLICE_SetTimerPeriodMatch(handle_ptr->ccu8_slice_ptr, (uint16_t)(period_value));
XMC_CCU8_SLICE_SetTimerCompareMatch(handle_ptr->ccu8_slice_ptr, XMC_CCU8_SLICE_COMPARE_CHANNEL_1, (uint16_t)compare);
XMC_CCU8_EnableShadowTransfer(handle_ptr->ccu8_kernel_ptr, handle_ptr->shadow_mask);
handle_ptr->compare_value = compare;
handle_ptr->period_value = period_value;
XMC_DEBUG("PWM_lCCU8_SetFreq:frequency set")

else
{
  status = PWM_STATUS_FAILURE;
  XMC_DEBUG("PWM_lCCU8_SetFreq:frequency c
ould not be set")
00647  }
00648
00649  return status;
00650 } /* end of PWM_lCCU8_SetFreq() api */
00651
00652 /*----------------------------------------------
00653---------------------------------------------
00654---------------------------------------------*/
00655 /*Sets the frequency and duty cycle for CCU8
00656 slice. */
00657 PWM_STATUS_t PWM_lCCU8_SetFreqAndDutyCycle(P
00658 WM_t *const handle_ptr, uint32_t pwm_freq_hz, uint
00659 32_t duty_cycle)
00660 {
00661  PWM_STATUS_t status;
00662  uint32_t module_freq;
00663  uint8_t prescaler;
00664  uint32_t period_value;
00665  uint32_t compare;
00666  XMC_ASSERT("PWM_lCCU8_SetFreqAndDutyCycle:
00667  Invalid pwm_freq_hz ", (pwm_freq_hz != 0U))
00668  XMC_ASSERT("PWM_lCCU8_SetFreqAndDutyCycle:
00669  Invalid duty_cycle",((duty_cycle >= 0) &&
00670  (duty_cycle <= PWM_MAX_ DUTY_CYCLE)))
00671  status = handle_ptr->state;
00672  prescaler = 0U;
00673  period_value = 0U;
00674  if ((status != PWM_STATUS_UNINITIALIZED) &
00675  /* Can't set the frequency when the PWM is
00676 not yet initialized or when required frequency is
00677 0*/
00678  module_freq = (pwm_freq_hz * (1L << prescaler)) / 
00679  1000000L;
00680  if (module_freq < PWM_MIN_FREQ) module_freq = 
00681  PWM_MIN_FREQ;
00682  period_value = ((module_freq * 1000000L) / 
00683  pwm_freq_hz);
& ((uint32_t)0 != pwm_freq_hz))
00672  {
00673       status = PWM_STATUS_SUCCESS;
00674   /* Get the Module frequency */
00675   module_freq = handle_ptr->global_ccu8_handle->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)module_freq / (uint32_t)pwm_freq_hz) >> (uint32_t)prescaler;
00681       /* If the prescaler selected is not big enough goto the next prescaler value else come out. */
00682       if (period_value <= PWM_MAX_TIMER_COUNT)
00683           {
00684               break;
00685           }
00686           prescaler++;
00687   }
00688
00689   /* Can't set the frequency if the required value is too small or when the required frequency 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_1CCU8_SetFreqAndDutyCycle: Frequency or duty cycle could not be set")
00693       status = PWM_STATUS_FAILURE;
00694   }
00695   else
/*Calculate the new compare values using new period values */
compare = (period_value * ((uint32_t)PWM_MAX_DUTY_CYCLE - duty_cycle)) / ((uint32_t)100 * PWM_DUTY_CYCLE_SCALE);

XMC_CCU8_SLICE_SetPrescaler(handle_ptr->ccu8_slice_ptr, prescaler);

/* The period register is always one count less than calculated. */
period_value = period_value - (uint32_t)1;
XMC_CCU8_SLICE_SetTimerPeriodMatch(handle_ptr->ccu8_slice_ptr, (uint16_t)(period_value));

XMC_CCU8_SLICE_SetTimerCompareMatch(handle_ptr->ccu8_slice_ptr, XMC_CCU8_SLICE_COMPARE_CHANNEL_1, (uint16_t)compare);

XMC_CCU8_EnableShadowTransfer(handle_ptr->ccu8_kernel_ptr, handle_ptr->shadow_mask);

handle_ptr->compare_value = compare;
handle_ptr->period_value = period_value;
handle_ptr->duty_cycle = duty_cycle;
XMC_DEBUG("PWM_CCU8_SetFreqAndDutyCycle: Frequency and Duty cycle set")

else
{

}
status = PWM_STATUS_FAILURE;
XMC_DEBUG("PWM_lCCU8_SetFreqAndDutyCycle: Frequency and Duty cycle could not be set")
}

return status;
} /* end of PWM_lCCU8_SetFreqAndDutyCycle() api */

/* This function returns the version of the PWM App*/
DAVE_APP_VERSION_t PWM_GetAppVersion(void)
{
  DAVE_APP_VERSION_t version;
  version.major = (uint8_t)PWM_MAJOR_VERSION;
  version.minor = (uint8_t)PWM_MINOR_VERSION;
  version.patch = (uint8_t)PWM_PATCH_VERSION;
  return version;
}
/* This function initializes the app */
PWM_STATUS_t PWM_Init(PWM_t *const handle_ptr)
{
    PWM_STATUS_t status;
    status = PWM_STATUS_FAILURE;
    XMC_ASSERT("PWM_Init:Invalid handle_ptr", (handle_ptr != NULL))
#ifdef PWM_SLICE_USED_CCU4
    if (PWM_TIMER_SLICE_CCU4 == handle_ptr->timer_type)
    {
        status = PWM_lCCU4_Init(handle_ptr);
    }
#endif
#ifdef PWM_SLICE_USED_CCU8
    if (PWM_TIMER_SLICE_CCU8 == handle_ptr->timer_type)
    {
        status = PWM_lCCU8_Init(handle_ptr);
    }
#endif
    return (status);
}

/* This function starts the PWM generation. This needs to be called even if external start is
void PWM_Start(PWM_t *const handle_ptr) {
	XMC_ASSERT("PWM_Start:Invalid handle_ptr", (handle_ptr != NULL))

#ifdef PWM_SLICE_USED_CCU4
	if (PWM_TIMER_SLICE_CCU4 == handle_ptr->timer_type) {
		PWM_ICCU4_Start(handle_ptr);
	}
#endif

#ifdef PWM_SLICE_USED_CCU8
	if (PWM_TIMER_SLICE_CCU8 == handle_ptr->timer_type) {
		PWM_ICCU8_Start(handle_ptr);
	}
#endif

/* This function stops the PWM generation. */

void PWM_Stop(PWM_t *const handle_ptr) {
	XMC_ASSERT("PWM_Stop:Invalid handle_ptr", (handle_ptr != NULL))

#ifdef PWM_SLICE_USED_CCU4
	if (PWM_TIMER_SLICE_CCU4 == handle_ptr->timer_type)
00801  
00802  
00803  
00804  #endif
00805
00806  
00807  
00808  
00809  
00810  }  
00811  #endif
00812  }
00813
00814  /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
00815  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
00816  ~~~~~~~~~~~~~~~~~~~~~~~*/
00817  /*This function is used to set the duty cycle (uint32_t) of the PWM waveform */
00818  
00819  PWM_STATUS_t PWM_setDutyCycle(PWM_t *const handle_ptr, uint32_t duty_cycle)
00820  
00821  {  
00822  
00823  
00824  
00825  
00826  }  
00827  }  
00828  #endif
00829
ifdef PWM_SLICE_USED_CCU8
if (PWM_TIMER_SLICE_CCU8 == handle_ptr->timer_type)
{
    status = PWM_lCCU8_SetDutyCycle(handle_ptr, duty_cycle);
}
#endif
return (status);

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~*/
/*This function changes the PWM frequency. Input parameter is the frequency value in Hz*/
PWM_STATUS_t PWM_SetFreq(PWM_t *const handle_ptr, uint32_t pwm_freq_hz)
{
    PWM_STATUS_t status;
    status = PWM_STATUS_FAILURE;
    XMC_ASSERT("PWM_SetFreq:Invalid handle_ptr", (handle_ptr != NULL))
#endif
ifdef PWM_SLICE_USED_CCU4
if (PWM_TIMER_SLICE_CCU4 == handle_ptr->timer_type)
{
    status = PWM_lCCU4_SetFreq(handle_ptr, pwm_freq_hz);
}
#endif
ifdef PWM_SLICE_USED_CCU8
if (PWM_TIMER_SLICE_CCU8 == handle_ptr->timer_type)
{  
  status = PWM_lCCU8_SetFreq(handle_ptr, pwm_freq_hz);
}
#endif
return status;

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~*/

/*This function sets frequency and the duty cycle */
PWM_STATUS_t PWM_SetFreqAndDutyCycle(PWM_t *const handle_ptr, uint32_t pwm_freq_hz, uint32_t duty_cycle)
{
  PWM_STATUS_t status;
  status = PWM_STATUS_FAILURE;
  XMC_ASSERT("PWM_SetFreqAndDutyCycle:Invalid handle_ptr", (handle_ptr != NULL))
  #ifdef PWM_SLICE_USED_CCU4
  if (PWM_TIMER_SLICE_CCU4 == handle_ptr->timer_type)
  {
    status = PWM_lCCU4_SetFreqAndDutyCycle(handle_ptr, pwm_freq_hz, duty_cycle);
  }
  #endif
  #ifdef PWM_SLICE_USED_CCU8
  if (PWM_TIMER_SLICE_CCU8 == handle_ptr->timer_type)
  {
    status = PWM_lCCU8_SetFreqAndDutyCycle(h
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;
}
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
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 Acknowledges the corresponding interrupt */

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~*/
/*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_ptr->ccu8_slice_ptr,
                                         (XMC_CCU8_SLICE_OUTPUT_PASSIVE_LEVEL_t)pwm_output_passive_level);
        XMC_CCU8_EnableShadowTransfer(handle_ptr->ccu8_kernel_ptr, handle_ptr->shadow_mask);
        XMC_DEBUG("PWM_SetPassiveLevel:CCU8 slice, passive level changed")
    }
#endif
r->ccu8_slice_ptr, XMC_CCU8_SLICE_OUTPUT_0,
00985 (XMC _CCU8_SLICE_OUTPUT_PASSIVE_LEVEL_t)pwm_output_pass
00986 ive_level);
00986 00987 XMC_CCU8_EnableShadowTransfer(handle_ptr-
00987 ->ccu8_kernel_ptr, handle_ptr->shadow_mask);
00988 XMC_DEBUG("PWM_SetPassiveLevel:CCU8 slic
e, passive level changed")
00989 }
00990 #endif
00991 }
00992 00993 /*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
00993 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
00993 ~~~~~~~~~~~~~~~~~~~~~~~*/
00994 /*Configures the period register */
00995 PWM_STATUS_t PWM_SetPeriodMatchValue(PWM_t *
00995 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
01000 d handle_ptr", (handle_ptr != NULL))
01001 XMC_ASSERT("PWM_SetPeriodMatchValue:Invali
01001 d period_match_value", (period_match_value <= PWM _MAX_PERIOD_VALUE))
01002 01003 status = handle_ptr->state;
01004 01005 if ( (status != PWM_STATUS_UNINITIALIZED)
01005 && (PWM_MAX_PERIOD_VALUE >= period_match_value))
01006 {
01007 compare = (period_match_value * ((uint32
01007 _t)PWM_MAX_DUTY_CYCLE - handle_ptr->duty_cycle))
((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(handle_ptr->ccu4_slice_ptr, (uint16_t)period_match_value);
01014
01015          XMC_CCU4_SLICE_SetTimerCompareMatch(handle_ptr->ccu4_slice_ptr, (uint16_t)compare);
01016
01017          XMC_CCU4_EnableShadowTransfer(handle_ptr->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(handle_ptr->ccu8_slice_ptr, (uint16_t)period_match_value);
01025
01026          XMC_CCU8_SLICE_SetTimerCompareMatch(handle_ptr->ccu8_slice_ptr, XMC_CCU8_SLICECOMPARE_CHANNEL_1,
01027              (uint16_t)compare);
01028
01029          XMC_CCU8_EnableShadowTransfer(handle_ptr->ccu8_kernel_ptr, handle_ptr->shadow_mask);
01030      }
01031  #endif
01032  handle_ptr->period_value = period_match
handle_ptr->compare_value = compare;
status = PWM_STATUS_SUCCESS;
}
else {
    status = PWM_STATUS_FAILURE;
}
return (status);

/*CODE_BLOCK_END*/