EE 3171 Lecture 10

Pulse Width Modulation

Lecture Overview

A Few Definitions to get us started

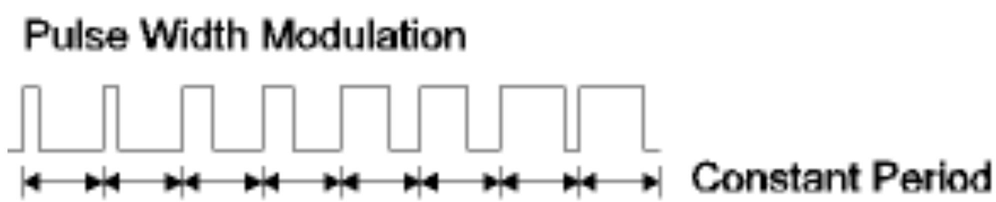
Implementation on the Tiva C

- Registers, programming, etc.

What is PWM?

Pulse Width Modulation

Beginning with a signal with a fixed period, we alter the information sent by modulating (or changing) the width of the pulse.



Duty Cycle

The heart of PWM is the idea of a *duty cycle*.

In other words, of the n time units of the period, during how many is the output = 1?

Usually expressed as a percentage.

n cycles, k cycles where output = 1

Duty cycle = 100 * (k/n)

Duty Cycle Examples

Let the period be 30 ms and the duty cycle be 60%. For how long is the signal high?

= 30 ms * (0.6) = 18 ms

Let the period be 45 ms and the signal is high for 15 ms. What is the duty cycle?

= (15/45) * 100 = 33.33333%

Let the signal be high for 29 ms and the known duty cycle is 30%. What is the period?

= (29/x) * 100 = 30%, x = 96.666 ms

Duty Cycle Implementation

Start with a counter and two registers

- One to set the pulse width and
- One to set the period.

At the start of the period, the counter is reset to zero and the output goes high.

The counter is incremented by a clock.

When the the counter equals the value in the pulse width register, the output goes low.

When the counter reaches that of the period register, the cycle repeats.

PWM on the Tiva C

The Tiva C contains two PWM modules, each with four PWM generator blocks and a control block, for a total of 16 PWM outputs.

The control block determines the polarity of the PWM signals, and which signals are passed through to the pins.

More Tiva C Overview

The timer in each PWM generator runs in one of two modes: Count-Down mode or Count-Up/Down mode.

In Count-Down mode, the timer:

- Counts from the load value to zero
- Goes back to the load value
- Continues counting down

In Count-Up/Down mode, the timer:

- Counts from zero up to the load value
- Back down to zero
- Back up to the load value
- And so on.

Generally, Count-Down mode is used for generating left- or right-aligned PWM signals, while the Count-Up/Down mode is used for generating center-aligned PWM signals.

Configuration/Initialization

1. Enable the PWM clock by writing a value of `0x0010.0000` to the `RCGC0` register in the System Control module.

2. Enable the clock to the appropriate GPIO module via the `RCGC2` register in the System Control module.

3. In the GPIO module, enable the appropriate pins for their alternate function using the `GPIOAFSEL` register.

4. Configure the `PMC`*n* fields in the `GPIOPCTL` register to assign the PWM signals to the appropriate pins.

5. Configure the Run-Mode Clock Configuration (`RCC`) register in the System Control module to use the PWM divide (`USEPWMDIV`) and set the divider (`PWMDIV`).

6. Configure the PWM generator for countdown mode with immediate updates to the parameters. Two registers must be configured: `PWM0CTL` and `PWM0GENA`.

Configuration/Initialization

7. Calculate the period and use this value to set the `PWM0LOAD` register. In Count-Down mode, set the `LOAD` field in the `PWM0LOAD` register to the requested period minus one.

8. Set the pulse width of the `M`*`n`*`PWM0` pin for appropriate duty cycle with a value in the `PWM0CMPA` register.

9. Start the timers in `PWM0CTL`.

10. Enable the output in `PWMENABLE`.

Sample Configuration

Let's configure a 60% duty cycle, 20 KHz PWM output.

The Tiva C uses an 80 MHz clock. If we use a PWM divider of 2, that means our PWM base clock is 40 MHz.

20 KHz means a period of 1/20,000 = 0.00005 s.

How many ticks of a 40 MHz clock make up 0.0005 seconds?

2000!

Period of 40 MHz clock =
0.000000025s.
0.0005/0.000000025 = 2000.

Sample Configuration

So what's the value in the `LOAD` register?

Period - 1 = 1999 = 0x7CF

Value in the compare register?

0.4 * 2000 = 800 = 0x320

Two things:
Many code samples use hex values for these fields for no apparent reason.
We calculate the compare value based on 1-DC.

Initialization Code

Let’s look at the registers and values to finish this initialization.

RCGC0

Here's the bit.

Run Mode Clock Gating Control Register 0 (RCGC0)

Base 0x400F.E000
Offset 0x100
Type RO, reset 0x0000.0040

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved			WDT1	reserved		CAN1	CAN0	reserved			PWM0	reserved		ADC1	ADC0
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved				MAXADC1SPD		MAXADC0SPD		reserved	HIB	reserved		WDT0	reserved		
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

Initialization Code:

```
SYSCTL_RCGC0_R |= 0x00100000;
```

RCGC2

Run Mode Clock Gating Control Register 2 (RCGC2)

Base 0x400F.E000
Offset 0x108
Type RO, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															USB0
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved		UDMA	reserved							GPIOF	GPIOE	GPIOD	GPIOC	GPIOB	GPIOA
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Here, we're using the PWM output on Port D.

Initialization Code:

```
SYSCTL_RCGC2_R |= 0x00000008;
```

You have to pick the GPIO port associated with the PWM output you're using.

Which GPIO Port?

How did we know which GPIO port to use?

See Table 20-1 on page 1234 of the TM4C123 reference manual.

I arbitrarily chose `M1PWM0`, which is output 0 of PWM Module 1, Generator 0.

And is connected to `PD0`.

GPIOAFSEL

GPIO Alternate Function Select (GPIOAFSEL)

GPIO Port A (APB) base: 0x4000.4000
GPIO Port A (AHB) base: 0x4005.8000
GPIO Port B (APB) base: 0x4000.5000
GPIO Port B (AHB) base: 0x4005.9000
GPIO Port C (APB) base: 0x4000.6000
GPIO Port C (AHB) base: 0x4005.A000
GPIO Port D (APB) base: 0x4000.7000
GPIO Port D (AHB) base: 0x4005.B000
GPIO Port E (APB) base: 0x4002.4000
GPIO Port E (AHB) base: 0x4005.C000
GPIO Port F (APB) base: 0x4002.5000
GPIO Port F (AHB) base: 0x4005.D000
Offset 0x420
Type RW, reset -

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved								AFSEL							
Type	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	-	-	-	-	-	-	-	-

This is
PD0.

Initialization Code:

```
GPIO_PORTD_AFSEL_R |= 0x01;
```

GPIOPCTL

GPIO Port Control (GPIOPCTL)

GPIO Port A (APB) base: 0x4000.4000
GPIO Port A (AHB) base: 0x4005.8000
GPIO Port B (APB) base: 0x4000.5000
GPIO Port B (AHB) base: 0x4005.9000
GPIO Port C (APB) base: 0x4000.6000
GPIO Port C (AHB) base: 0x4005.A000
GPIO Port D (APB) base: 0x4000.7000
GPIO Port D (AHB) base: 0x4005.B000
GPIO Port E (APB) base: 0x4002.4000
GPIO Port E (AHB) base: 0x4005.C000
GPIO Port F (APB) base: 0x4002.5000
GPIO Port F (AHB) base: 0x4005.D000
Offset 0x52C
Type RW, reset -

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	PMC7				PMC6				PMC5				PMC4			
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	PMC3				PMC2				PMC1				PMC0			
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Well, what value do we put in there?

PCTL Values for Port D

Table 23-5. GPIO Pins and Alternate Functions

IO	Pin	Analog Function	Digital Function (GPIOPCTL PMCx Bit Field Encoding)[a]										
			1	2	3	4	5	6	7	8	9	14	15
PD0	61	AIN7	SSI3Clk	SSI1Clk	I2C3SCL	M0PWM6	M1PWM0	-	WT2CCP0	-	-	-	-

Initialization Code:

```
GPIO_PORTD_PCTL_R |= 0x05;
```

RCC

Run-Mode Clock Configuration (RCC)

Base 0x400F.E000
Offset 0x060
Type RW, reset 0x078E.3AD1

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved				ACG	SYSDIV				USESYSDIV	reserved	USEPWMDIV	PWMDIV			reserved
Type	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RO	RW	RW	RW	RW	RO
Reset	0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved		PWRDN	reserved	BYPASS	XTAL					OSCSRC		reserved			MOSCDIS
Type	RO	RO	RW	RO	RW	RW	RW	RW	RW	RW	RW	RW	RO	RO	RO	RW
Reset	0	0	1	1	1	0	1	0	1	1	0	1	0	0	0	1

Initialization Code:

```
SYSCTL_RCC_R |= 0x00100000;
SYSCTL_RCC_R &= ~0x000E0000;
SYSCTL_RCC_R += ???;
```

What goes here?

RCC

19:17	PWMDIV	RW	0x7	PWM Unit Clock Divisor This field specifies the binary divisor used to predivide the system clock down for use as the timing reference for the PWM module. The rising edge of this clock is synchronous with the system clock.

Value	Divisor
0x0	/2
0x1	/4
0x2	/8
0x3	/16
0x4	/32
0x5	/64
0x6	/64
0x7	/64 (default)

Initialization Code:

```
SYSCTL_RCC_R |= 0x00100000;
SYSCTL_RCC_R &= ~0x000E0000;
SYSCTL_RCC_R |= 0x00000000;
```

PWMCTL

PWM Master Control (PWMCTL)

PWM0 base: 0x4002.8000
PWM1 base: 0x4002.9000
Offset 0x000
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved												GLOBALSYNC3	GLOBALSYNC2	GLOBALSYNC1	GLOBALSYNC0
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

We're going to first disable then reenable PWM 1.

Disable: `PWM_1_CTL_R = 0;`

Enable: `PWM_1_CTL_R |= 0x01;`

The reenabling occurs after initialization is complete.

PWMGENA

PWMn Generator A Control (PWMnGENA)

PWM0 base: 0x4002.8000
PWM1 base: 0x4002.9000
Offset 0x060
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved				ACTCMPBD		ACTCMPBU		ACTCMPAD		ACTCMPAU		ACTLOAD		ACTZERO	
Type	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Initialization Code for Toggling:

```
PWM_1_GENA_R |= 0x8C;
```

Let's tear this field apart quickly.

PWM0GENA Fields

	ACTCMPAD	ACTCMPAU	ACTLOAD	ACTZERO
Meaning	`cmpA == Counter` (Counting Down)	`cmpA == Counter` (counting up)	`cmpa == LOAD`	`cmpA == 0`
0x0	Do nothing.	Do nothing.	Do nothing.	Do nothing.
0x1	Toggle pwmA.	Toggle pwmA.	Toggle pwmA.	Toggle pwmA.
0x2	Drive pwmA LOW.	Drive pwmA LOW.	Drive pwmA LOW.	Drive pwmA LOW.
0x3	Drive pwmA HIGH.	Drive pwmA HIGH.	Drive pwmA HIGH.	Drive pwmA HIGH.

Count and Duty Cycle

These are pretty straightforward:

```
PWM_1_LOAD_R = 0x7CF;
PWM_1_CMPA_R = 0x320;
```

It's tempting to complicate these, but they're just counters.

PWMENABLE

PWM Output Enable (PWMENABLE)

PWM0 base: 0x4002.8000
PWM1 base: 0x4002.9000
Offset 0x008
Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved								PWM7EN	PWM6EN	PWM5EN	PWM4EN	PWM3EN	PWM2EN	PWM1EN	PWM0EN
Type	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Initialization Code for Output:

```
PWM_1_ENABLE_R |= 0x01;
```

TivaWare for PWM

As always, TI provided an easy way to configure the PWM modules through TivaWare functions instead of Direct Register Access. Let’s look at the key functions.

Knowing the Clock Rate

If the number of ticks we put into the Period and Pulse Width registers are based on the clock rate, we need to know what the clock rate actually is.

There are a pair of functions that are useful here:

```
SysCtlClockSet()

SysCtlClockGet()
```

`SysCtlClockSet()`

Sets the clocking of the device.

```
void SysCtlClockSet(uint32_t ui32Config)
```

`ui32Config` is the required configuration of the device clocking.

This function configures the clocking of the device. The input crystal frequency, oscillator to be used, use of the PLL, and the system clock divider are all configured with this function.

The `ui32Config` parameter is the logical OR of several different values, many of which are grouped into sets where only one can be chosen.

SysCtlClockSet()

The system clock divider is chosen with one of the following values:

> `SYSCTL_SYSDIV_1, SYSCTL_SYSDIV_2, SYSCTL_SYSDIV_3, ... SYSCTL_SYSDIV_64`.

The use of the PLL (phase locked loop) is chosen with either `SYSCTL_USE_PLL` or `SYSCTL_USE_OSC`. The external crystal frequency is chosen with one of the following values:

> `SYSCTL_XTAL_4MHZ`, `SYSCTL_XTAL_5MHZ`, `SYSCTL_XTAL_6_14MHZ`, `SYSCTL_XTAL_8_19MHZ`, `SYSCTL_XTAL_12_2MHZ`, `SYSCTL_XTAL_16MHZ`, `SYSCTL_XTAL_20MHZ`, `SYSCTL_XTAL_24MHZ`, or `SYSCTL_XTAL_25MHz`.
>
> Values below `SYSCTL_XTAL_5MHZ` are not valid when the PLL is in operation.

The oscillator source is chosen with one of the following values:

> `SYSCTL_OSC_MAIN`, `SYSCTL_OSC_INT`, `SYSCTL_OSC_INT4`, `SYSCTL_OSC_INT30`, or `SYSCTL_OSC_EXT32`.
>
> `SYSCTL_OSC_EXT32` is only available on devices with the hibernate module, and then only when the hibernate module has been enabled.

`SysCtlClockSet()`

The internal and main oscillators are disabled with the `SYSCTL_INT_OSC_DIS` and `SYSCTL_MAIN_OSC_DIS` flags, respectively. The external oscillator must be enabled in order to use an external clock source. Note that attempts to disable the oscillator used to clock the device is prevented by the hardware.

To clock the system from an external source (such as an external crystal oscillator), use `SYSCTL_USE_OSC | SYSCTL_OSC_MAIN`. To clock the system from the main oscillator, use `SYSCTL_USE_OSC | SYSCTL_OSC_MAIN`.

To clock the system from the PLL, use `SYSCTL_USE_PLL | SYSCTL_OSC_MAIN`, and select the appropriate crystal with one of the `SYSCTL_XTAL_xxx` values.

Returns: None.

A typical value:

```
SysCtlClockSet(SYSCTL_SYSDIV1 | SYSCTL_USE_OSC |
SYSCTL_XTAL_16MHZ | SYSCTL_OSC_MAIN); // Sets a 16 MHz clock.
```

`SysCtlClockGet()`

Gets the processor clock rate.

```
uint32_t SysCtlClockGet(void)
```

This function determines the clock rate of the processor clock, which is also the clock rate of the peripheral modules (with the exception of PWM, which has its own clock divider; other peripherals may have different clocking, see the device data sheet for details).

This cannot return accurate results if `SysCtlClockSet()` has not been called to configure the clocking of the device, or if the device is directly clocked from a crystal (or a clock source) that is not one of the supported crystal frequencies. In the latter case, this function should be modified to directly return the correct system clock rate.

Returns: The processor clock rate.

`SysCtlPeripheralEnable()`

Enables a peripheral.

Prototype: `void SysCtlPeripheralEnable(uint32_t ui32Peripheral)`

Parameters:

- `ui32Peripheral` is the peripheral to enable.

This function enables a peripheral. At power-up, all peripherals are disabled; they must be enabled in order to operate or respond to register reads/writes.

Two Important GPIO Functions

In order to completely configure a PWM channel, you have to tell the GPIO that it is operating in PWM mode.

There are two functions that *both* need to be used:

```
GPIOPinConfigure()

GPIOPinTypePWM()
```

`GPIOPinConfigure()`

Configures the alternate function of a GPIO pin.

```
void GPIOPinConfigure(uint32_t
ui32PinConfig)
```

`ui32PinConfig` is the pin configuration value, specified as only one of the `GPIO_P??_???` values.

Helpfully, TI decided not to provide any kind of list of these `GPIO_P??_???` values anywhere. At all.

`GPIOPinConfigure()`

This function configures the pin mux that selects the peripheral function associated with a particular GPIO pin. Only one peripheral function at a time can be associated with a GPIO pin, and each peripheral function should only be associated with a single GPIO pin at a time (despite the fact that many of them can be associated with more than one GPIO pin). To fully configure a pin, a `GPIOPinType*()` function should also be called.

The available mappings are supplied on a per-device basis in `pin_map.h`. The `PART_IS_<partno>` define enables the appropriate set of defines for the device that is being used.

> This means you *must* `#include “driverlib/pin_map.h”` in your PWM programs.

Returns: None.

GPIOPinTypePWM()

Configures pin(s) for use by the PWM peripheral.

```
void GPIOPinTypePWM(uint32_t ui32Port, uint8_t ui8Pins)
```

`ui32Port` is the base address of the GPIO port.
`ui8Pins` is the bit-packed representation of the pin(s).

The PWM pins must be properly configured for the PWM peripheral to function correctly. This function provides a typical configuration for those pin(s); other configurations may work as well depending upon the board setup (for example, using the on-chip pull-ups).

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note: This function cannot be used to turn any pin into a PWM pin; it only configures a PWM pin for proper operation. Devices with flexible pin muxing also require a `GPIOPinConfigure()` function call.

Returns: None.

Liar, Liar…

Throughout the documentation, TI says to use `PWM_BASE` as the base address for PWM TivaWare functions.

This is fundamentally wrong.

Consistent with other modules, the proper syntax is `PWM?_BASE`.

Exempli gratia: `PWM0_BASE`.

`PWMGenConfigure()`

Configures a PWM generator.

```
void PWMGenConfigure(uint32_t ui32Base,
uint32_t ui32Gen, uint32_t ui32Config)
```

`ui32Base` is the base address of the PWM module.
`ui32Gen` is the PWM generator to configure.
`ui32Config` is the configuration for the PWM generator.

This function is used to set the mode of operation for a PWM generator. The counting mode, synchronization mode, and debug behavior are all configured. After configuration, the generator is left in the disabled state.

Configuration Parameters

Appropriate values for `ui32Gen`:

`PWM_GEN_0`, `PWM_GEN_1`, `PWM_GEN_2`, or `PWM_GEN_3`

Appropriate values for :

`PWM_GEN_MODE_DOWN` or `PWM_GEN_MODE_UP_DOWN` to specify the counting mode

`PWM_GEN_MODE_SYNC` or `PWM_GEN_MODE_NO_SYNC` to specify the counter load and comparator update synchronization mode

There are lots more, but these are the most important.

PWMGenPeriodSet()

Sets the period of a PWM generator.

```
void PWMGenPeriodSet(uint32_t ui32Base, uint32_t
ui32Gen, uint32_t ui32Period)
```

`ui32Base` is the base address of the PWM module.
`ui32Gen` is the PWM generator to be modified.
`ui32Period` specifies the period of PWM generator output, measured in clock ticks.

This function sets the period of the specified PWM generator block, where the period of the generator block is defined as the number of PWM clock ticks between pulses on the generator block zero signal.

Don't do the LOAD-1 thing here.

`PWMPulseWidthSet()`

Sets the pulse width for the specified PWM output.

```
void PWMPulseWidthSet(uint32_t ui32Base,
uint32_t ui32PWMOut, uint32_t ui32Width)
```

`ui32Base` is the base address of the PWM module.
`ui32PWMOut` is the PWM output to modify.
`ui32Width` specifies the width of the *positive portion* of the pulse.

In other words, the number of HIGH cycles.

This function sets the pulse width for the specified PWM output (e.g., `PWM_OUT_4`), where the pulse width is defined as the number of PWM clock ticks.

`PWMGenEnable()`

Enables the timer/counter for a PWM generator block.

```
void PWMGenEnable(uint32_t ui32Base,
uint32_t ui32Gen)
```

- `ui32Base` is the base address of the PWM module.
- `ui32Gen` is the PWM generator to be enabled.

This function allows the PWM clock to drive the timer/counter for the specified generator block.

PWMOutputState()

Enables or disables PWM outputs.

```
void PWMOutputState(uint32_t ui32Base, uint32_t
ui32PWMOutBits, bool bEnable)
```

- `ui32Base` is the base address of the PWM module.
- `ui32PWMOutBits` are the PWM outputs to be modified.
- `bEnable` determines if the signal is enabled or disabled.

This function enables or disables the selected PWM outputs. The outputs are selected using the parameter `ui32PWMOutBits` (e.g., `PWM_OUT_1_BIT`). If `bEnable` is true, then the selected PWM outputs are enabled, or placed in the active state.

TivaWare Config Example

```
uint32_t ulPeriod;
uint32_t PWMFreq = 20000; // 20 KHz output

SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC |
               SYSCTL_XTAL_16MHZ | SYSCTL_OSC_MAIN);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1);

SysCtlDelay(1);

ulPeriod = SysCtlClockGet() / PWMFreq;

GPIOPinConfigure(GPIO_PD0_M1PWM0);
GPIOPinTypePWM(GPIO_PORTD_BASE, GPIO_PIN_0);

SysCtlPWMClockSet(SYSCTL_PWMDIV_1);

PWMGenConfigure(PWM1_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
PWMGenPeriodSet(PWM1_BASE, PWM_GEN_0, ulPeriod);
PWMPulseWidthSet(PWM1_BASE, PWM_OUT_0, ulPeriod * 0.6);
PWMGenEnable(PWM1_BASE, PWM_GEN_0);
PWMOutputState(PWM1_BASE, PWM_OUT_0_BIT, true);

while (1) ;
```

16 MHz clock.

TivaWare Config Example

```
uint32_t ulPeriod;
uint32_t PWMFreq = 20000; // 20 KHz output

SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC |
               SYSCTL_XTAL_16MHZ | SYSCTL_OSC_MAIN);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1);

SysCtlDelay(1);

ulPeriod = SysCtlClockGet() / PWMFreq;

GPIOPinConfigure(GPIO_PD0_M1PWM0);
GPIOPinTypePWM(GPIO_PORTD_BASE, GPIO_PIN_0);

SysCtlPWMClockSet(SYSCTL_PWMDIV_1);

PWMGenConfigure(PWM1_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
PWMGenPeriodSet(PWM1_BASE, PWM_GEN_0, ulPeriod);
PWMPulseWidthSet(PWM1_BASE, PWM_OUT_0, ulPeriod * 0.6);
PWMGenEnable(PWM1_BASE, PWM_GEN_0);
PWMOutputState(PWM1_BASE, PWM_OUT_0_BIT, true);

while (1) ;
```

M1PWM0 shares PD0. Turn on clocks to both.

TivaWare Config Example

```c
uint32_t ulPeriod;
uint32_t PWMFreq = 20000; // 20 KHz output

SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC |
               SYSCTL_XTAL_16MHZ | SYSCTL_OSC_MAIN);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1);

SysCtlDelay(1);

ulPeriod = SysCtlClockGet() / PWMFreq;

GPIOPinConfigure(GPIO_PD0_M1PWM0);
GPIOPinTypePWM(GPIO_PORTD_BASE, GPIO_PIN_0);

SysCtlPWMClockSet(SYSCTL_PWMDIV_1);

PWMGenConfigure(PWM1_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
PWMGenPeriodSet(PWM1_BASE, PWM_GEN_0, ulPeriod);
PWMPulseWidthSet(PWM1_BASE, PWM_OUT_0, ulPeriod * 0.6);
PWMGenEnable(PWM1_BASE, PWM_GEN_0);
PWMOutputState(PWM1_BASE, PWM_OUT_0_BIT, true);

while (1) ;
```

Flexibly calculate the period based on system clock.

TivaWare Config Example

```
uint32_t ulPeriod;
uint32_t PWMFreq = 20000; // 20 KHz output

SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC |
               SYSCTL_XTAL_16MHZ | SYSCTL_OSC_MAIN);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1);

SysCtlDelay(1);

ulPeriod = SysCtlClockGet() / PWMFreq;

GPIOPinConfigure(GPIO_PD0_M1PWM0);
GPIOPinTypePWM(GPIO_PORTD_BASE, GPIO_PIN_0);

SysCtlPWMClockSet(SYSCTL_PWMDIV_1);

PWMGenConfigure(PWM1_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
PWMGenPeriodSet(PWM1_BASE, PWM_GEN_0, ulPeriod);
PWMPulseWidthSet(PWM1_BASE, PWM_OUT_0, ulPeriod * 0.6);
PWMGenEnable(PWM1_BASE, PWM_GEN_0);
PWMOutputState(PWM1_BASE, PWM_OUT_0_BIT, true);

while (1) ;
```

Two functions to configure the GPIO pins.

TivaWare Config Example

```
uint32_t ulPeriod;
uint32_t PWMFreq = 20000; // 20 KHz output

SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC |
               SYSCTL_XTAL_16MHZ | SYSCTL_OSC_MAIN);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1);

SysCtlDelay(1);

ulPeriod = SysCtlClockGet() / PWMFreq;

GPIOPinConfigure(GPIO_PD0_M1PWM0);
GPIOPinTypePWM(GPIO_PORTD_BASE, GPIO_PIN_0);

SysCtlPWMClockSet(SYSCTL_PWMDIV_1);

PWMGenConfigure(PWM1_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
PWMGenPeriodSet(PWM1_BASE, PWM_GEN_0, ulPeriod);
PWMPulseWidthSet(PWM1_BASE, PWM_OUT_0, ulPeriod * 0.5);
PWMGenEnable(PWM1_BASE, PWM_GEN_0);
PWMOutputState(PWM1_BASE, PWM_OUT_0_BIT, true);

while (1) ;
```

Our calculations were based on PWM clock being the same as system clock.

TivaWare Config Example

```
uint32_t ulPeriod;
uint32_t PWMFreq = 20000; // 20 KHz output

SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC |
               SYSCTL_XTAL_16MHZ | SYSCTL_OSC_MAIN);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1);

SysCtlDelay(1);

ulPeriod = SysCtlClockGet() / PWMFreq;

GPIOPinConfigure(GPIO_PD0_M1PWM0);
GPIOPinTypePWM(GPIO_PORTD_BASE, GPIO_PIN_0);

SysCtlPWMClockSet(SYSCTL_PWMDIV_1);

PWMGenConfigure(PWM1_BASE, PWM_GEN_0, PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
PWMGenPeriodSet(PWM1_BASE, PWM_GEN_0, ulPeriod);
PWMPulseWidthSet(PWM1_BASE, PWM_OUT_0, ulPeriod * 0.6);
PWMGenEnable(PWM1_BASE, PWM_GEN_0);
PWMOutputState(PWM1_BASE, PWM_OUT_0_BIT, true);

while (1) ;
```

Standard 5 functions to configure a PWM signal.

In this case we have GEN_0 and OUT_0, because this particular PWM is controlled by PW1 Module 0.

Different PWM Outputs

What if we wanted to use `M0PWM7`? What would change?

On `PC5`, so enable Port C.

Use `GPIO_PC5_M0PWM7` in `GPIOPinConfigure()`.

> And obviously change the ports/pins in `GPIOPinTypePWM()`.

Uses Module 0 PWM Generator 3, so we would use `PWM0_BASE`, `PWM_GEN_3`, `PWM_OUT_7`.

Summary

We frequently want to generate fixed-period events with different duty cycles.

We use the PWM module to make that happen.

Process:

Start with scaling the clock.

Figure out the period count.

Figure out the duty cycle count.

Configure starting high or low.

Enable the channel.