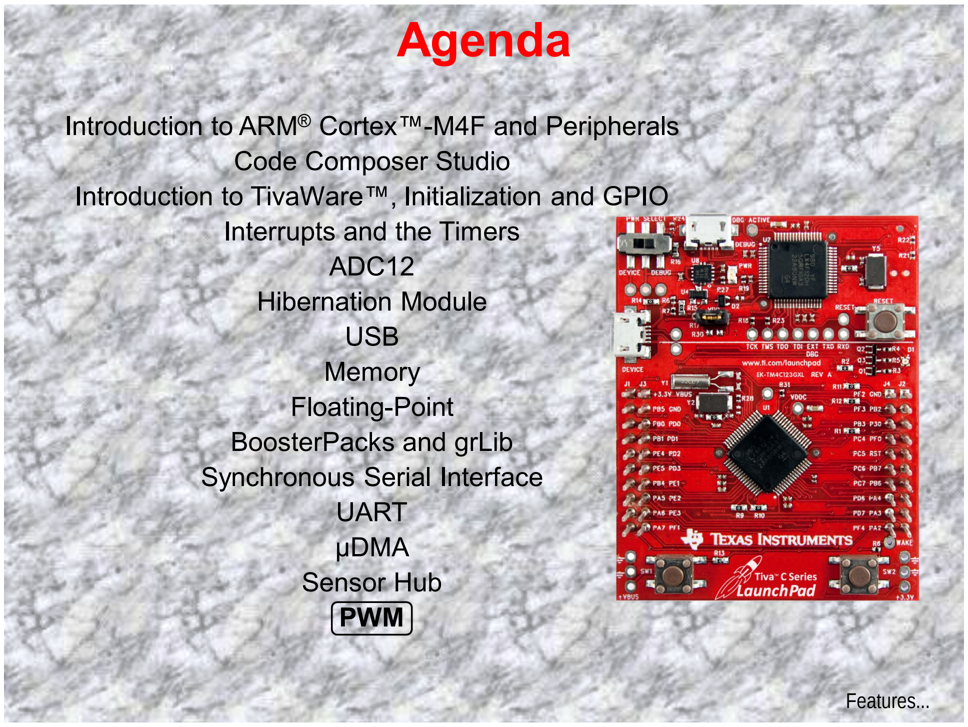
PWM

Introduction

Pulse width modulation or PWM is a method of digitally encoding analog signal levels. It is used extensively in servo positioning, motor control, power supplies and lighting control.



Agenda

Introduction to ARM® Cortex™-M4F and Peripherals
Code Composer Studio
Introduction to TivaWare™, Initialization and GPIO
Interrupts and the Timers
ADC12
Hibernation Module
USB
Memory
Floating-Point
BoosterPacks and grLib
Synchronous Serial Interface
UART
μDMA
Sensor Hub
PWM

Features...

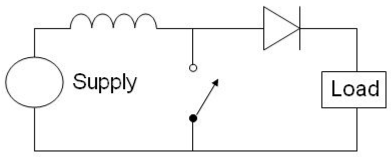
Chapter Topics

Pulse Width Modulation

Pulse Width Modulation

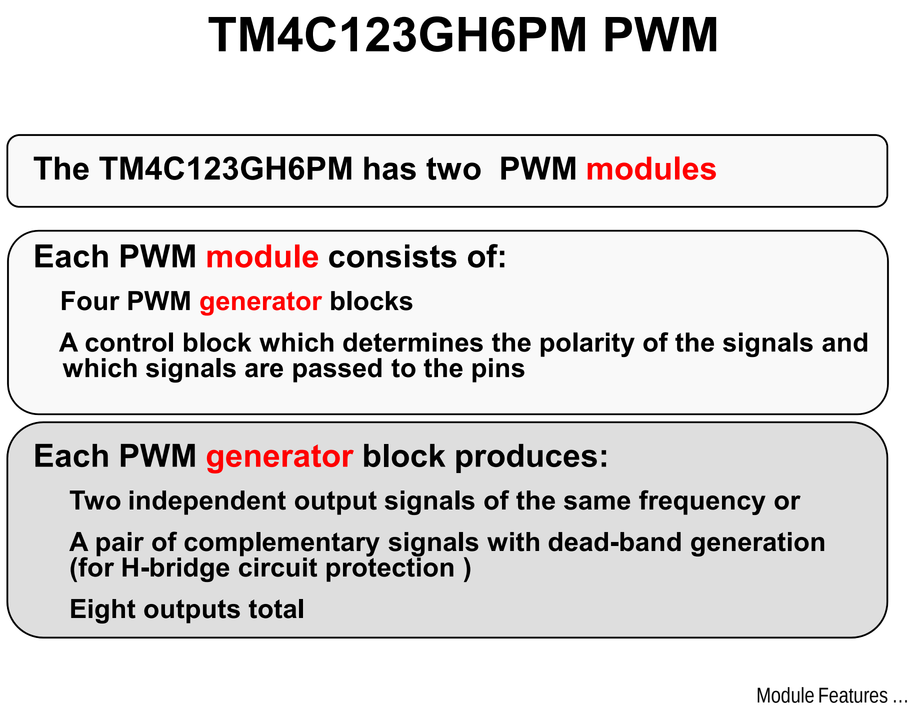
Pulse Width Modulation (PWM) is a method of digitally encoding analog signal levels. High-resolution digital counters are used to generate a square wave of a given frequency, and the duty cycle of that square wave is modulated to encode the analog signal.

Typical applications for PWM are switching power supplies, motor control, servo positioning and lighting control.



TM4C123GH6PM PWM ... for

TM4C123GH6PM PWM



TM4C123GH6PM PWM

The TM4C123GH6PM has two PWM modules

Each PWM module consists of:

- Four PWM generator blocks
- A control block which determines the polarity of the signals and which signals are passed to the pins

Each PWM generator block produces:

- Two independent output signals of the same frequency or
- A pair of complementary signals with dead-band generation (for H-bridge circuit protection)
- Eight outputs total

Module Features ...

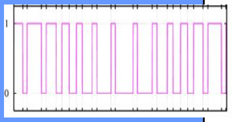
PWM Generator Features

PWM Generator Features

One hardware fault input for low-latency shutdown

One 16-bit counter

- Down or Up/Down count modes
- Output frequency controlled by a 16-bit load value
- Load value updates can be synchronized
- Produces output signals at zero and load value



Two PWM comparators

- Comparator value updates can be synchronized
- Produces output signals on match

PWM signal generator

- Output PWM signal is constructed based on actions taken as a result of the counter and PWM comparator output signals
- Produces two independent PWM signals

PWM Generator Features (cont)

Dead-band generator

- Produces two PWM signals with programmable dead-band delays suitable for driving a half-H bridge
- Can be bypassed, leaving input PWM signals unmodified

Flexible output control block with:

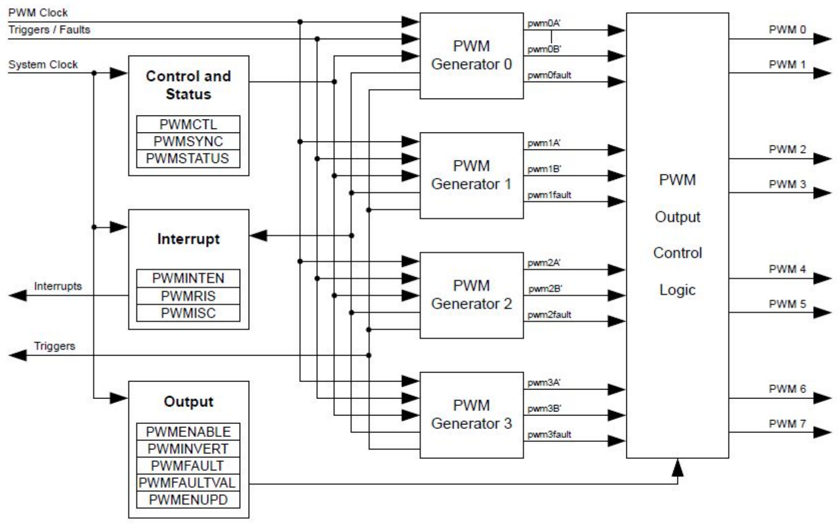
- PWM output enable of each PWM signal
- Optional output inversion of each PWM signal (polarity control)
- Optional fault handling for each PWM signal
- Synchronization of timers in the PWM generator blocks
- Synchronization of timer/comparator updates across the PWM generator blocks
- Interrupt status summary of the PWM generator blocks

Can initiate an ADC sample sequence

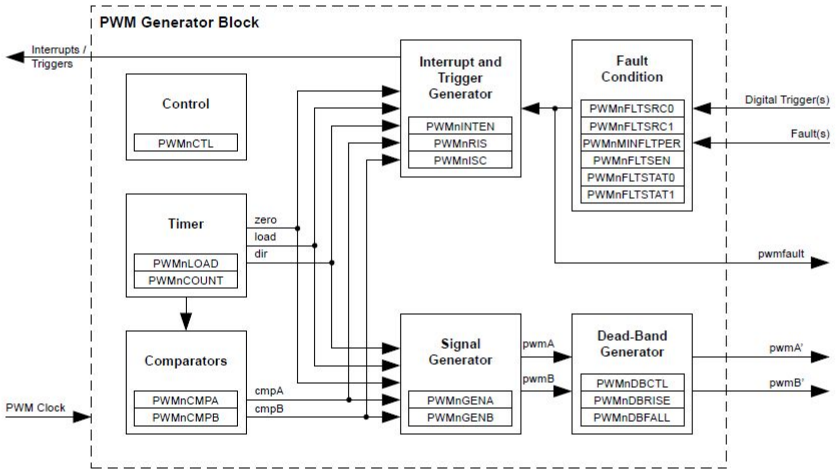
Block diagram ...

Block Diagrams

PWM Module Block Diagram



PWM Generator Block Diagram



Lab ...

Lab 03: PWM

Objective

In this lab you'll use the PWM on the Tiva C Series device to control the the brightness of LED present in the development board.

Software

1. Create the CCS project with empty main.c file by following the instructions from previous labs.

2. ► Open main.c and add (or copy/paste) the following lines to the top of the file:

```
#include <stdint.h>
#include <stdbool.h>
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/debug.h"
#include "driverlib/pwm.h"
#include "driverlib/pin_map.h"
#include "inc/hw_gpio.h"
#include "driverlib/rom.h"
```

3. We'll use a 55Hz base frequency to control the servo. ► Skip a line and add the following definition right below the includes:

```
#define PWM_FREQUENCY  55
```

main()

4. ► Skip a line and enter the following lines after the error checking routine as a template for main().

```
int main(void)
{

}
```

5. The following variables will be used to program the PWM. They are defined as "volatile" to guarantee that the compiler will not eliminate them, regardless of the optimization setting. The ui8Adjust variable will allow us to adjust the brightness of the LED. 83 is the center position to create a 1.5mS pulse from the PWM
Here's how we came up with 83 … In the servo control code (covered shortly) we're going to divide the PWM period by 1000. Since the programmed frequency is 55HZ and the period is 18.2mS, dividing that by 1000 gives us a pulse resolution of 1.82µS.

Multiplying that by 83 gives us a pulse-width of 1.51mS. Other selections for the resolution, etc. would be just as valid as long as they produced a 1.5mS pulse-width. Take care though to be sure that your numbers will fit within the 16-bit registers.

► Insert these four lines as the first in `main()` :

```
volatile uint32_t ui32Load;
volatile uint32_t ui32PWMClock;
volatile uint8_t ui8Adjust;
ui8Adjust = 83;
```

6. Let's run the CPU at 40MHz. The PWM module is clocked by the system clock through a divider, and that divider has a range of 2 to 64. By setting the divider to 64, it will run the PWM clock at 625 kHz. Note that we're using the ROM versions to reduce our code size.
► Leave a line for spacing and add these lines after the previous ones in `main()`.

```
SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_OSC_MAIN|SYSCTL_XTAL_16MHZ);
SysCtlPWMClockSet(SYSCTL_PWMDIV_64);
```

7. We need to enable the PWM1 and the GPIOF module (for the LaunchPad buttons on PF0 and PF4 and PWM output on PF1).
► Skip a line and add the following lines of code after the last:

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM1);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
```

8. Port F pin 1 (PF1) must be configured as a PWM output pin for module 1, PWM generator 2 (check out the schematic).
► Skip a line and add the following lines of code after the last:

```
ROM_GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_1);
ROM_GPIOPinConfigure(GPIO_PF1_M1PWM5);
```

9. Port F pin 0 and pin 4 are connected to the S2 and S1 switches on the LaunchPad. In order for the state of the pins to be read in our code, the pins must be pulled up. (The BUTTONSPOLL API could do this for us, but that API checks for individual button presses rather than a button being held down). Pulling up a GPIO pin is normally pretty straight-forward, but PF0 is considered a critical peripheral since it can be configured to be a NMI input. Since this is the case, we will have to unlock the GPIO commit control register to make this change. This feature was mentioned in chapter 3 of the workshop.

The first three lines below unlock the GPIO commit control register, the fourth configures PF0 & 4 as inputs and the fifth configures the internal pull-up resistors on both pins. The drive strength setting is merely a place keeper and has no function for an input.

► Skip a line and add these 5 lines after the last:

```
HWREG(GPIO_PORTF_BASE + GPIO_O_LOCK) = GPIO_LOCK_KEY;
HWREG(GPIO_PORTF_BASE + GPIO_O_CR) |= 0x01;
HWREG(GPIO_PORTF_BASE + GPIO_O_LOCK) = 0;
GPIODirModeSet(GPIO_PORTF_BASE, GPIO_PIN_4|GPIO_PIN_0, GPIO_DIR_MODE_IN);
GPIOPadConfigSet(GPIO_PORTF_BASE, GPIO_PIN_4|GPIO_PIN_0, GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);
```

10. The PWM clock is SYSCLK/64 (set in step 12 above). Divide the PWM clock by the desired frequency (55Hz) to determine the count to be loaded into the Load register. Then subtract 1 since the counter down-counts to zero. Configure module 1 PWM generator 2 as a down-counter and load the count value.
    ► Skip a line and add these four lines after the last:

```
ui32PWMClock = SysCtlClockGet() / 64;
ui32Load = (ui32PWMClock / PWM_FREQUENCY) - 1;
PWMGenConfigure(PWM1_BASE, PWM_GEN_2, PWM_GEN_MODE_DOWN);
PWMGenPeriodSet(PWM1_BASE, PWM_GEN_2, ui32Load);
```

11. Now we can make the final PWM settings and enable it. The first line sets the pulse-width. The PWM Load value is divided by 1000 (which determines the minimum resolution for the servo) and the multiplied by the adjusting value. These numbers could be changed to provide more or less resolution. In lines two and three, PWM module 1, generator 2 needs to be enabled as an output and enabled to run.
    ► Skip a line and add these three lines after the last:

```
PWMPulseWidthSet(PWM1_BASE, PWM_OUT_5, ui8Adjust * ui32Load / 1000);
PWMOutputState(PWM1_BASE, PWM_OUT_5_BIT, true);
PWMGenEnable(PWM1_BASE, PWM_GEN_2);
```

12. ► Skip a line and add a `while(1)` loop just before the final closing brace. At this point you can test-build your code. If you run it, the Red LED will glow with medium brightness.

```
while(1)
{
}
```

Controlling the Brightness of LED

13. This code will read the PF4 pin to see if SW1 is pressed. No debouncing is needed since we're not looking for individual key pressed. Each time this code is run it will decrement the adjust variable by one unless it reaches the lower 1mS limit. This number, like the center and upper positions was determined by measuring the output of the PWM. The last line loads the PWM pulse width register with the new value. This load is done asynchronously to the output. In a more critical design you might want to consult the databook concerning making this load differently.
► Add the following code inside the `while(1)` loop.

```
if(GPIOPinRead(GPIO_PORTF_BASE,GPIO_PIN_4)==0x00)
{
        ui8Adjust--;
        if (ui8Adjust < 56)
        {
                ui8Adjust = 56;
        }
        PWMPulseWidthSet(PWM1_BASE, PWM_OUT_5, ui8Adjust * ui32Load / 1000);
}
```

14. The next code will read the PF0 pin to see if SW2 is pressed to increment the pulse width. The maximum limit is set to reach 2.0mS.
► Skip a line and add the following code after the last inside the `while(1)` loop.

```
if(GPIOPinRead(GPIO_PORTF_BASE,GPIO_PIN_0)==0x00)
{
        ui8Adjust++;
        if (ui8Adjust > 111)
        {
                ui8Adjust = 111;
        }
        PWMPulseWidthSet(PWM1_BASE, PWM_OUT_5, ui8Adjust * ui32Load / 1000);
}
```

15. This final line determines the speed of the loop. If the servo moves too quickly or too slowly for you, feel free to change the count to your liking.
► Skip a line and add the this line after the last inside the `while(1)` loop.

```
SysCtlDelay(100000);
```

If your code looks strange, don't forget that you can automatically correct the indentation.

► Save your changes.

If you're having issues, you can find this code in your lab03 project folder as `main.txt`.

Build and Run the Code

16. Compile and download your application by clicking the Debug button.

17. Click the Resume button to run the program. Use the SW1 and SW2 buttons on the LaunchPad to change the brightness of LED. Feel free to set breakpoints and monitor the load and pulse width variables if you like.

18. When you're finished, click the Terminate button to return to the Editing perspective, close the lab03 project and close Code Composer Studio.

You're done with Lab03