```c
// Helper function module

#include "ES_Configure.h"
#include "ES_Framework.h"
#include "ES_DeferRecall.h"

#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_gpio.h"
#include "inc/hw_sysctl.h"
#include "inc/hw_timer.h"
#include "inc/hw_nvic.h"
#include "inc/hw_pwm.h"
#include "driverlib/sysctl.h"
#include "driverlib/pin_map.h"      // Define PART_TM4C123GH6PM in project
#include "driverlib/gpio.h"
#include "BITDEFS.H"
#include "string.h"

#include "Helpers.h"
#include "DEFINITIONS.h"

// Initializes at GPIO Port
void GPIO_Init(uint8_t SYSCTL, int port_base, uint8_t bits, uint8_t direction)
{
    //Enable the Port Pins
    HWREG(SYSCTL_RCGCGPIO) |= SYSCTL;

    //Wait for the Pins to be enabled before proceeding
    while ((HWREG(SYSCTL_PRGPIO) & SYSCTL) != SYSCTL);

    //Assign the port to be digital
    HWREG(port_base + GPIO_O_DEN) |= bits;      //Assign digital port to PE0

    //Depending on Direction Assign
    if (direction == INPUT)
    {
        HWREG(port_base + GPIO_O_DIR) &= ~bits; //Set data direction to be an
              input (aka set bit to 0)
    }
    else
    {
        HWREG(port_base + GPIO_O_DIR) |= bits;  //Set data direction to be an
              output (aka set bit to 1)
    }
}

// Sets the specified GPIO pins high
void GPIO_Set(int port, uint8_t bits)
{
    HWREG(port+(GPIO_O_DATA + ALL_BITS)) |= bits;
}

// Sets the specified GPIO pins low
void GPIO_Clear(int port, uint8_t bits)
{
    HWREG(port+(GPIO_O_DATA + ALL_BITS)) &= ~bits; //use not to clear the bits
}


/**********************************************************************
```

```
PWM Initialization Funtion

*********************************************************************/
#define PWM_TICKS_PER_MS 1250 // 40,000/32

typedef struct
{
    t8_t prpwm;
    int pwm_base;
} Module;

typedef struct
{
    t8_t ctl_mode;
    uint8_t ctl_enable;
} Block;

typedef struct
{

    int up_one;
    int up_zero;
    int down_one;
    int down_zero;
    int up_down_mode;
    uint8_t gpio_pin;
    int enable_const;
} Generator;

typedef struct
{
    sk;
    int newval;
    int rcgc;
    int base;
    int pin;
} AFSELUnit;

static Module modules[] =
{
    {.rcgc = SYSCTL_RCGCPWM_R0, .prpwm = SYSCTL_PRPWM_R0, .pwm_base = PWM0_BASE},
    {.rcgc = SYSCTL_RCGCPWM_R1, .prpwm = SYSCTL_PRPWM_R1, .pwm_base = PWM1_BASE},
};

static Block blocks[] =
{
    {.ctl = PWM_O_0_CTL, .load = PWM_O_0_LOAD, .ctl_mode = PWM_0_CTL_MODE, .
     ctl_enable = PWM_0_CTL_ENABLE},
    {.ctl = PWM_O_1_CTL, .load = PWM_O_1_LOAD, .ctl_mode = PWM_1_CTL_MODE, .
     ctl_enable = PWM_1_CTL_ENABLE},
    {.ctl = PWM_O_2_CTL, .load = PWM_O_2_LOAD, .ctl_mode = PWM_2_CTL_MODE, .
     ctl_enable = PWM_2_CTL_ENABLE},
    {.ctl = PWM_O_3_CTL, .load = PWM_O_3_LOAD, .ctl_mode = PWM_3_CTL_MODE, .
     ctl_enable = PWM_3_CTL_ENABLE},
};

static Generator generators[] =
{
    {.gen = PWM_O_0_GENA, .cmp = PWM_O_0_CMPA, .up_one = PWM_0_GENA_ACTCMPAU_ONE,
     .up_zero = PWM_0_GENA_ACTCMPAU_ZERO, .down_one = PWM_0_GENA_ACTCMPAD_ONE, .
     down_zero = PWM_0_GENA_ACTCMPAD_ZERO, .up_down_mode = PWM_0_CTL_GENAUPD_LS,
     .enable_const = PWM_ENABLE_PWM0EN},
```

```c
    {.gen = PWM_O_1_GENA, .cmp = PWM_O_1_CMPA, .up_one = PWM_1_GENA_ACTCMPAU_ONE,
     .up_zero = PWM_1_GENA_ACTCMPAU_ZERO, .down_one = PWM_1_GENA_ACTCMPAD_ONE, .
     down_zero = PWM_1_GENA_ACTCMPAD_ZERO, .up_down_mode = PWM_1_CTL_GENAUPD_LS,
     .enable_const = PWM_ENABLE_PWM2EN},
    {.gen = PWM_O_2_GENA, .cmp = PWM_O_2_CMPA, .up_one = PWM_2_GENA_ACTCMPAU_ONE,
     .up_zero = PWM_2_GENA_ACTCMPAU_ZERO, .down_one = PWM_2_GENA_ACTCMPAD_ONE, .
     down_zero = PWM_2_GENA_ACTCMPAD_ZERO, .up_down_mode = PWM_2_CTL_GENAUPD_LS,
     .enable_const = PWM_ENABLE_PWM4EN},
    {.gen = PWM_O_3_GENA, .cmp = PWM_O_3_CMPA, .up_one = PWM_3_GENA_ACTCMPAU_ONE,
     .up_zero = PWM_3_GENA_ACTCMPAU_ZERO, .down_one = PWM_3_GENA_ACTCMPAD_ONE, .
     down_zero = PWM_3_GENA_ACTCMPAD_ZERO, .up_down_mode = PWM_3_CTL_GENAUPD_LS,
     .enable_const = PWM_ENABLE_PWM6EN},
    {.gen = PWM_O_0_GENB, .cmp = PWM_O_0_CMPB, .up_one = PWM_0_GENB_ACTCMPBU_ONE,
     .up_zero = PWM_0_GENB_ACTCMPBU_ZERO, .down_one = PWM_0_GENB_ACTCMPBD_ONE, .
     down_zero = PWM_0_GENB_ACTCMPBD_ZERO, .up_down_mode = PWM_0_CTL_GENBUPD_LS,
     .enable_const = PWM_ENABLE_PWM1EN},
    {.gen = PWM_O_1_GENB, .cmp = PWM_O_1_CMPB, .up_one = PWM_1_GENB_ACTCMPBU_ONE,
     .up_zero = PWM_1_GENB_ACTCMPBU_ZERO, .down_one = PWM_1_GENB_ACTCMPBD_ONE, .
     down_zero = PWM_1_GENB_ACTCMPBD_ZERO, .up_down_mode = PWM_1_CTL_GENBUPD_LS,
     .enable_const = PWM_ENABLE_PWM3EN},
    {.gen = PWM_O_2_GENB, .cmp = PWM_O_2_CMPB, .up_one = PWM_2_GENB_ACTCMPBU_ONE,
     .up_zero = PWM_2_GENB_ACTCMPBU_ZERO, .down_one = PWM_2_GENB_ACTCMPBD_ONE, .
     down_zero = PWM_2_GENB_ACTCMPBD_ZERO, .up_down_mode = PWM_2_CTL_GENBUPD_LS,
     .enable_const = PWM_ENABLE_PWM5EN},
    {.gen = PWM_O_3_GENB, .cmp = PWM_O_3_CMPB, .up_one = PWM_3_GENB_ACTCMPBU_ONE,
     .up_zero = PWM_3_GENB_ACTCMPBU_ZERO, .down_one = PWM_3_GENB_ACTCMPBD_ONE, .
     down_zero = PWM_3_GENB_ACTCMPBD_ZERO, .up_down_mode = PWM_3_CTL_GENBUPD_LS,
     .enable_const = PWM_ENABLE_PWM7EN},
};

static AFSELUnit PWMInfo[] =
{
    {.clear_mask = 0xf0ffffff, .newval = (4<<(6*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R1, .base = GPIO_PORTB_BASE, .pin = GPIO_PIN_6},
    {.clear_mask = 0x0fffffff, .newval = (4<<(7*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R1, .base = GPIO_PORTB_BASE, .pin = GPIO_PIN_7},
    {.clear_mask = 0xfff0ffff, .newval = (4<<(4*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R1, .base = GPIO_PORTB_BASE, .pin = GPIO_PIN_4},
    {.clear_mask = 0xff0fffff, .newval = (4<<(5*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R1, .base = GPIO_PORTB_BASE, .pin = GPIO_PIN_5},
    {.clear_mask = 0xfff0ffff, .newval = (4<<(4*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R4, .base = GPIO_PORTE_BASE, .pin = GPIO_PIN_4},
    {.clear_mask = 0xff0fffff, .newval = (4<<(5*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R4, .base = GPIO_PORTE_BASE, .pin = GPIO_PIN_5},
    {.clear_mask = 0xfffffff0, .newval = (4<<(0*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R3, .base = GPIO_PORTD_BASE, .pin = GPIO_PIN_0},
    {.clear_mask = 0xffffff0f, .newval = (4<<(1*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R3, .base = GPIO_PORTD_BASE, .pin = GPIO_PIN_1},
    {.clear_mask = 0xfffffff0, .newval = (5<<(0*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R3, .base = GPIO_PORTD_BASE, .pin = GPIO_PIN_0},
    {.clear_mask = 0xffffff0f, .newval = (5<<(1*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R3, .base = GPIO_PORTD_BASE, .pin = GPIO_PIN_1},
    {.clear_mask = 0xfff0ffff, .newval = (5<<(4*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R4, .base = GPIO_PORTE_BASE, .pin = GPIO_PIN_4},
    {.clear_mask = 0xff0fffff, .newval = (5<<(5*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R4, .base = GPIO_PORTE_BASE, .pin = GPIO_PIN_5},
    {.clear_mask = 0xfffffff0, .newval = (5<<(0*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R5, .base = GPIO_PORTF_BASE, .pin = GPIO_PIN_0},
    {.clear_mask = 0xffffff0f, .newval = (5<<(1*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R5, .base = GPIO_PORTF_BASE, .pin = GPIO_PIN_1},
    {.clear_mask = 0xfffff0ff, .newval = (5<<(2*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R5, .base = GPIO_PORTF_BASE, .pin = GPIO_PIN_2},
    {.clear_mask = 0xffff0fff, .newval = (5<<(3*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R5, .base = GPIO_PORTF_BASE, .pin = GPIO_PIN_3}
};

void InitPWM(uint8_t module, uint8_t block, uint8_t generator, uint32_t period)
{
    // Enable clock to PWM
    HWREG(SYSCTL_RCGCPWM) |= modules[module].rcgc;

    // enable clock to port
    HWREG(SYSCTL_RCGCGPIO) |= PWMInfo[(module * 8) + (block * 2) + generator].
          rcgc;

    // Set PWM clock based on system clock
    HWREG(SYSCTL_RCC) = (HWREG(SYSCTL_RCC) & ~SYSCTL_RCC_PWMDIV_M) | (
          SYSCTL_RCC_USEPWMDIV | SYSCTL_RCC_PWMDIV_32);

    // wait for pwm clock
    while ((HWREG(SYSCTL_PRPWM) & modules[module].prpwm) != modules[module].
           prpwm);

    // disable pwm
    HWREG(modules[module].pwm_base+blocks[block].ctl) &= ~blocks[block].
          ctl_enable;

    // program generator
    HWREG(modules[module].pwm_base + generators[4 * generator + block].gen) = (
          generators[4 * generator + block].up_one | generators[4 * generator +
          block].down_zero);

    // Set period
    HWREG(modules[module].pwm_base+blocks[block].load) = (((period *
          PWM_TICKS_PER_MS)/ MICROSECONDS_DIVISOR) -1) >> 1;

    // enable the PWM outputs
    HWREG(modules[module].pwm_base+PWM_O_ENABLE) |= (generators[4 * generator +
          block].enable_const);

    // Enable alternate functionality of GPIO pins
    HWREG(PWMInfo[(module * 8) + (block * 2) + generator].base +GPIO_O_AFSEL) |=
          PWMInfo[(module * 8) + (block * 2) + generator].pin;

    // Select PWM as alternate function
    HWREG(PWMInfo[(module * 8) + (block * 2) + generator].base +GPIO_O_PCTL) = (
          HWREG(PWMInfo[(module * 8) + (block * 2) + generator].base+GPIO_O_PCTL)
          & PWMInfo[(module * 8) + (block * 2) + generator].clear_mask) +
          PWMInfo[(module * 8) + (block * 2) + generator].newval;

    // Enable pins as digital I/O
    HWREG(PWMInfo[(module * 8) + (block * 2) + generator].base+GPIO_O_DEN) |=
          PWMInfo[(module * 8) + (block * 2) + generator].pin;

    // Enable pins as output
    HWREG(PWMInfo[(module * 8) + (block * 2) + generator].base+GPIO_O_DIR) |=
          PWMInfo[(module * 8) + (block * 2) + generator].pin;

    // Select up-down counting, set local synchronization, and enable generator
    HWREG(modules[module].pwm_base + blocks[block].ctl) = (HWREG(modules[module].
          pwm_base + blocks[block].ctl) & !(blocks[block].ctl_mode | blocks[
          block].ctl_enable | generators[4 * generator + block].up_down_mode)) |
          (blocks[block].ctl_mode | blocks[block].ctl_enable | generators[4 *
          generator + block].up_down_mode);
```

```
}

/**********************************************************************

PWM Setting Funtions

**********************************************************************/

void setPWM_value(float duty, uint8_t module, uint8_t block, uint8_t generator,
                  uint32_t period)
{
    if (duty >= 100.0f)
    {
        HWREG(modules[module].pwm_base + generators[4 * generator + block].gen)
              = (generators[4 * generator + block].up_one | generators[4 *
              generator + block].down_one);
        HWREG(modules[module].pwm_base + generators[4 * generator + block].cmp)
              = 1;
    }
    else if (duty <= 0.0f)
    {
        HWREG(modules[module].pwm_base + generators[4 * generator + block].gen)
              = (generators[4 * generator + block].up_zero | generators[4 *
              generator + block].down_zero);
        HWREG(modules[module].pwm_base + generators[4 * generator + block].cmp)
              = 1;
    }
    else
    {
        // Reset the cmp direction, in case our last PWM was 100% or 0% duty
        HWREG(modules[module].pwm_base + generators[4 * generator + block].gen)
              = (generators[4 * generator + block].up_one | generators[4 *
              generator + block].down_zero);
        //Cast the Duty Cycle as a UInt32 and set our new value
        HWREG(modules[module].pwm_base + generators[4 * generator + block].cmp)
              = HWREG(modules[module].pwm_base + blocks[block].load) - ((int)((
              duty * period * PWM_TICKS_PER_MS) / (100 * MICROSECONDS_DIVISOR)) >
              > 1);
    }
}


/**********************************************************************

Input Capture Initialization Funtion

**********************************************************************/

typedef struct
{
    int rcgc;
    int prwtimer;
} Timerpair;

typedef struct
{
    nt mr;
    int mr_m;
    int mr_period;
    int ams;
    int cdir;
    int cmr;
    int cap;
    int event_m;
    int stall;
    int toim;
    int cint;
    int cint_periodic;
    int eim;
    int raw;
} Timer;

typedef struct
{
    c_priority;
    int nvic_enable;
    int nvic_enable_bit;
    int priority_shift;
} NVICInfo;

static Timer timers[] =
{
    {.raw = TIMER_O_TAR, .enable = TIMER_CTL_TAEN, .ilr = TIMER_O_TAILR, .mr =
     TIMER_O_TAMR, .mr_m = TIMER_TAMR_TAMR_M, .mr_period =
     TIMER_TAMR_TAMR_PERIOD, .ams = TIMER_TAMR_TAAMS, .cdir = TIMER_TAMR_TACDIR,
     .cmr = TIMER_TAMR_TACMR, .cap = TIMER_TAMR_TAMR_CAP, .event_m =
     TIMER_CTL_TAEVENT_M, .stall = TIMER_CTL_TASTALL, .toim = TIMER_IMR_TATOIM, .
     cint = TIMER_ICR_CAECINT, .cint_periodic = TIMER_ICR_TATOCINT, .eim =
     TIMER_IMR_CAEIM},
    {.raw = TIMER_O_TBR, .enable = TIMER_CTL_TBEN, .ilr = TIMER_O_TBILR, .mr =
     TIMER_O_TBMR, .mr_m = TIMER_TBMR_TBMR_M, .mr_period =
     TIMER_TBMR_TBMR_PERIOD, .ams = TIMER_TBMR_TBAMS, .cdir = TIMER_TBMR_TBCDIR,
     .cmr = TIMER_TBMR_TBCMR, .cap = TIMER_TBMR_TBMR_CAP, .event_m =
     TIMER_CTL_TBEVENT_M, .stall = TIMER_CTL_TBSTALL, .toim = TIMER_IMR_TBTOIM, .
     cint = TIMER_ICR_CBECINT, .cint_periodic = TIMER_ICR_TBTOCINT, .eim =
     TIMER_IMR_CBEIM}
};

static Timerpair timerpairs[] =
{
    {.base = WTIMER0_BASE, .rcgc = SYSCTL_RCGCWTIMER_R0, .prwtimer =
     SYSCTL_PRWTIMER_R0},
    {.base = WTIMER1_BASE, .rcgc = SYSCTL_RCGCWTIMER_R1, .prwtimer =
     SYSCTL_PRWTIMER_R1},
    {.base = WTIMER2_BASE, .rcgc = SYSCTL_RCGCWTIMER_R2, .prwtimer =
     SYSCTL_PRWTIMER_R2},
    {.base = WTIMER3_BASE, .rcgc = SYSCTL_RCGCWTIMER_R3, .prwtimer =
     SYSCTL_PRWTIMER_R3},
    {.base = WTIMER4_BASE, .rcgc = SYSCTL_RCGCWTIMER_R4, .prwtimer =
     SYSCTL_PRWTIMER_R4},
    {.base = WTIMER5_BASE, .rcgc = SYSCTL_RCGCWTIMER_R5, .prwtimer =
     SYSCTL_PRWTIMER_R5}
};

static AFSELUnit TimerInfo[] =
{
    {.clear_mask = 0xfff0ffff, .newval = (7<<(4*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R2, .base = GPIO_PORTC_BASE, .pin = GPIO_PIN_4},
    {.clear_mask = 0xff0fffff, .newval = (7<<(5*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R2, .base = GPIO_PORTC_BASE, .pin = GPIO_PIN_5},
    {.clear_mask = 0xf0ffffff, .newval = (7<<(6*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R2, .base = GPIO_PORTC_BASE, .pin = GPIO_PIN_6},
    {.clear_mask = 0x0fffffff, .newval = (7<<(7*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R2, .base = GPIO_PORTC_BASE, .pin = GPIO_PIN_7},
    {.clear_mask = 0xfffffff0, .newval = (7<<(0*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R3, .base = GPIO_PORTD_BASE, .pin = GPIO_PIN_0},
    {.clear_mask = 0xffffff0f, .newval = (7<<(1*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R3, .base = GPIO_PORTD_BASE, .pin = GPIO_PIN_1},
    {.clear_mask = 0xfffff0ff, .newval = (7<<(2*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R3, .base = GPIO_PORTD_BASE, .pin = GPIO_PIN_2},
    {.clear_mask = 0xffff0fff, .newval = (7<<(3*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R3, .base = GPIO_PORTD_BASE, .pin = GPIO_PIN_3},
    {.clear_mask = 0xfff0ffff, .newval = (7<<(4*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R3, .base = GPIO_PORTD_BASE, .pin = GPIO_PIN_4},
    {.clear_mask = 0xff0fffff, .newval = (7<<(5*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R3, .base = GPIO_PORTD_BASE, .pin = GPIO_PIN_5},
    {.clear_mask = 0xf0ffffff, .newval = (7<<(6*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R3, .base = GPIO_PORTD_BASE, .pin = GPIO_PIN_6},
    {.clear_mask = 0x0fffffff, .newval = (7<<(7*BITS_PER_NYBBLE)), .rcgc =
     SYSCTL_RCGCGPIO_R3, .base = GPIO_PORTD_BASE, .pin = GPIO_PIN_7}
};

static NVICInfo TimerNVIC[] =
{
    {.nvic_priority = NVIC_PRI23, .nvic_enable = NVIC_EN2, .nvic_enable_bit =
     BIT30HI, .priority_shift = 21},
    {.nvic_priority = NVIC_PRI23, .nvic_enable = NVIC_EN2, .nvic_enable_bit =
     BIT31HI, .priority_shift = 29},
    {.nvic_priority = NVIC_PRI24, .nvic_enable = NVIC_EN3, .nvic_enable_bit =
     BIT0HI, .priority_shift = 5},
    {.nvic_priority = NVIC_PRI24, .nvic_enable = NVIC_EN3, .nvic_enable_bit =
     BIT1HI, .priority_shift = 13},
    {.nvic_priority = NVIC_PRI24, .nvic_enable = NVIC_EN3, .nvic_enable_bit =
     BIT2HI, .priority_shift = 21},
    {.nvic_priority = NVIC_PRI24, .nvic_enable = NVIC_EN3, .nvic_enable_bit =
     BIT3HI, .priority_shift = 29},
    {.nvic_priority = NVIC_PRI25, .nvic_enable = NVIC_EN3, .nvic_enable_bit =
     BIT4HI, .priority_shift = 5},
    {.nvic_priority = NVIC_PRI25, .nvic_enable = NVIC_EN3, .nvic_enable_bit =
     BIT5HI, .priority_shift = 13},
    {.nvic_priority = NVIC_PRI25, .nvic_enable = NVIC_EN3, .nvic_enable_bit =
     BIT6HI, .priority_shift = 21},
    {.nvic_priority = NVIC_PRI25, .nvic_enable = NVIC_EN3, .nvic_enable_bit =
     BIT7HI, .priority_shift = 29},
    {.nvic_priority = NVIC_PRI26, .nvic_enable = NVIC_EN3, .nvic_enable_bit =
     BIT8HI, .priority_shift = 5},
    {.nvic_priority = NVIC_PRI26, .nvic_enable = NVIC_EN3, .nvic_enable_bit =
     BIT9HI, .priority_shift = 13}
};

void InitInputCapture(
    uint8_t timer_num,
    uint8_t timer_letter,
    uint8_t priority,
    uint32_t time_length      //included just for uniformity
)
{
    //printf("Paramaters: %d, %d, %d, %d \n\r", timer_num, timer_letter,
             priority, time_length);
    // start by enabling the clock to the timer (Wide Timer 0)
    HWREG(SYSCTL_RCGCWTIMER) |= timerpairs[timer_num].rcgc;

    // enable the clock to Port C / phototransistor port
    HWREG(SYSCTL_RCGCGPIO) |= TimerInfo[(timer_num * 2) + timer_letter].rcgc;

    // since we added this Port C clock init, we can immediately start
    // into configuring the timer, no need for further delay
    // make sure that timer (Timer A) is disabled before configuring
    HWREG(timerpairs[timer_num].base+TIMER_O_CTL) &= ~timers[timer_letter].
          enable;

    // set it up in 32bit wide (individual, not concatenated) mode
    // the constant name derives from the 16/32 bit timer, but this is a 32/64
    // bit timer so we are setting the 32bit mode
    HWREG(timerpairs[timer_num].base+TIMER_O_CFG) = TIMER_CFG_16_BIT;

    // we want to use the full 32 bit count, so initialize the Interval Load
    // register to 0xffff.ffff (its default value :-)
    HWREG(timerpairs[timer_num].base+timers[timer_letter].ilr) = 0xffffffff;

    // set up timer A in capture mode (TAMR=3, TAAMS = 0),
    // for edge time (TACMR = 1) and up-counting (TACDIR = 1)
    HWREG(timerpairs[timer_num].base+timers[timer_letter].mr) = (HWREG(
          timerpairs[timer_num].base+timers[timer_letter].mr) & ~timers[
          timer_letter].ams) |
            (timers[timer_letter].cdir | timers[timer_letter].cmr | timers[
             timer_letter].cap);

    // To set the event to rising edge, we need to modify the TAEVENT bits
    // in GPTMCTL. Rising edge = 00, so we clear the TAEVENT bits
    HWREG(timerpairs[timer_num].base+TIMER_O_CTL) &= ~timers[timer_letter].
          event_m;

    // Now Set up the port to do the capture (clock was enabled earlier)
    // start by setting the alternate function for Port C bit 4 (WT0CCP0)
    HWREG(TimerInfo[(timer_num * 2) + timer_letter].base+GPIO_O_AFSEL) |=
          TimerInfo[(timer_num * 2) + timer_letter].pin;

    // Then, map bit 4's alternate function to WT0CCP0
    // 7 is the mux value to select WT0CCP0, 16 to shift it over to the
    // right nibble for bit 4 (4 bits/nibble * 4 bits)
    HWREG(TimerInfo[(timer_num * 2) + timer_letter].base+GPIO_O_PCTL) =(HWREG(
          TimerInfo[(timer_num * 2) + timer_letter].base+GPIO_O_PCTL) &
          TimerInfo[(timer_num * 2) + timer_letter].clear_mask) + TimerInfo[(
          timer_num * 2) + timer_letter].newval;

    // Enable pin on Port for digital I/O
    HWREG(TimerInfo[(timer_num * 2) + timer_letter].base+GPIO_O_DEN) |=
          TimerInfo[(timer_num * 2) + timer_letter].pin;

    // make pin on Port into an input
    HWREG(TimerInfo[(timer_num * 2) + timer_letter].base+GPIO_O_DIR) &=
          ~TimerInfo[(timer_num * 2) + timer_letter].pin;

    // back to the timer to enable a local capture interrupt
    HWREG(timerpairs[timer_num].base+TIMER_O_IMR) |= timers[timer_letter].eim;
          //~TIMER_IMR_CAEIM;

    //Set Priority to higher than zero (1) so that it does not conflict with our
                                        other interrupt.
    //Interrupt 94 is INTC, hence BIT21 will set the priority to 1 (page 155 of
                                                                    Tiva
                                                                    Datasheet)
    HWREG(TimerNVIC[(timer_num * 2) + timer_letter].nvic_priority) = priority <<
          TimerNVIC[(timer_num * 2) + timer_letter].priority_shift;

    // enable the Timer A in Wide Timer 0 interrupt in the NVIC
    // it is interrupt number 94 so appears in EN2 at bit 30
```

```
    HWREG(TimerNVIC[(timer_num * 2) + timer_letter].nvic_enable) |= TimerNVIC[(
          timer_num * 2) + timer_letter].nvic_enable_bit;

    // make sure interrupts are enabled globally
    __enable_irq();

    // now kick the timer off by enabling it and enabling the timer to
    // stall while stopped by the debugger
    HWREG(timerpairs[timer_num].base+TIMER_O_CTL) |= (timers[timer_letter].
          enable | timers[timer_letter].stall);
}

/************************************************************************

Periodic Interrupt Initialization Funtion

************************************************************************/
void InitPeriodic(
    uint8_t timer_num,
    uint8_t timer_letter,
    uint8_t priority,
    uint32_t time_length)
{
    volatile uint32_t Dummy; // use volatile to avoid over-optimization
    //printf("Paramaters: %d, %d, %d, %d \n\r", timer_num, timer_letter,
             priority, time_length);

    // Enable clock to the timer
    HWREG(SYSCTL_RCGCWTIMER) |= timerpairs[timer_num].rcgc;

    // wait for ready register
    while ((HWREG(SYSCTL_PRWTIMER) & timerpairs[timer_num].prwtimer) !=
           timerpairs[timer_num].prwtimer);

    // Disable the timer
    HWREG(timerpairs[timer_num].base+TIMER_O_CTL) &= ~timers[timer_letter].
          enable;

    // Configure timer in 32-bit mode
    HWREG(timerpairs[timer_num].base+TIMER_O_CFG) = TIMER_CFG_16_BIT;

    // Set timer to periodic mode
    HWREG(timerpairs[timer_num].base+timers[timer_letter].mr) = (HWREG(
          timerpairs[timer_num].base+timers[timer_letter].mr) & ~timers[
          timer_letter].mr_m) | timers[timer_letter].mr_period;

    // Set timeout to 2ms
    HWREG(timerpairs[timer_num].base + timers[timer_letter].ilr) = (time_length
          * TICKS_PER_MS) / MICROSECONDS_DIVISOR;

    // Lower the priority of the wide timer 1 periodic timeout interrupt
    HWREG(TimerNVIC[(timer_num * 2) + timer_letter].nvic_priority) = (HWREG(
          TimerNVIC[(timer_num * 2) + timer_letter].nvic_priority) /*&
          priority_clear_mask*/) | (priority << TimerNVIC[(timer_num * 2) +
          timer_letter].priority_shift);

    // Enable local interrupt
    HWREG(timerpairs[timer_num].base+TIMER_O_IMR) |= timers[timer_letter].toim;

    // Enable interrupt in NVIC
    HWREG(TimerNVIC[(timer_num * 2) + timer_letter].nvic_enable) = TimerNVIC[(
          timer_num * 2) + timer_letter].nvic_enable_bit;
    // Globally enable interrupts
    __enable_irq();

    // Reenable timer in debug-stall mode
    HWREG(timerpairs[timer_num].base+TIMER_O_CTL) |= (timers[timer_letter].
          enable | timers[timer_letter].stall);
}

/*********************************************************************

Using Interrupt Funtions

*********************************************************************/
//Call this to clear the interrupt
void clearCaptureInterrupt(
    uint8_t timer_num,
    uint8_t timer_letter,
    uint8_t priority,
    uint32_t time_length)
{
    HWREG(timerpairs[timer_num].base +TIMER_O_ICR) = timers[timer_letter].cint;
}

void clearPeriodicInterrupt(
    uint8_t timer_num,
    uint8_t timer_letter,
    uint8_t priority,
    uint32_t time_length)
{
    HWREG(timerpairs[timer_num].base +TIMER_O_ICR) = timers[timer_letter].
          cint_periodic;
}

//Call this to disable the interrupt
void disableCaptureInterrupt(
    uint8_t timer_num,
    uint8_t timer_letter,
    uint8_t priority,
    uint32_t time_length)
{
    HWREG(timerpairs[timer_num].base + TIMER_O_IMR) &= ~timers[timer_letter].eim;
}

//Call this to enable the interrupt
void enableCaptureInterrupt(
    uint8_t timer_num,
    uint8_t timer_letter,
    uint8_t priority,
    uint32_t time_length)
{
    HWREG(timerpairs[timer_num].base + TIMER_O_IMR) |= timers[timer_letter].eim;
}

//Call this to disable the interrupt
void disablePeriodicInterrupt(
    uint8_t timer_num,
    uint8_t timer_letter,
    uint8_t priority,
    uint32_t time_length)
{
    HWREG(timerpairs[timer_num].base + TIMER_O_IMR) &= ~timers[timer_letter].
```

```
          toim;
}

//Call this to enable the interrupt
void enablePeriodicInterrupt(
    uint8_t timer_num,
    uint8_t timer_letter,
    uint8_t priority,
    uint32_t time_length)
{
    HWREG(timerpairs[timer_num].base + TIMER_O_IMR) |= timers[timer_letter].toim;
}


//call this to get the time of the interrupt
uint32_t captureInterrupt(
    uint8_t timer_num,
    uint8_t timer_letter,
    uint8_t priority,
    uint32_t time_length)
{
    return HWREG(timerpairs[timer_num].base + timers[timer_letter].raw);
}


/**********************************************************************
Other Funtions

**********************************************************************/
//Clamp Function
float clamp(float X, uint8_t min, uint8_t max)
{
    return (X > max) ? max : ((X < min) ? min : X);
}
```