EENG461, LAB4 report

Mark Clark and Jeremy Munson

Objective The primary objective of this lab was to learn and understand how general-purpose timers on the TM4C123GH6PM can be used to implement an edge-time event, also called an input capture.

Introduction The lab contains three tasks. The first task is to implement an ultrasonic sensor to measure the distance between the sensor and an object. The second task is to adjust a servo to an angle proportional to the distance. The angle must be between 0 degrees and 180 degrees, proportional with 0 being the shortest distance and 1 meter being the longest distance. The third task is to print the distance every third of a second.

Narrative The ultrasonic sensor operates with three stages. The first stage is the triggering event. The trigger for the sensor requires a high pulse with a width of at least 10 microseconds. We produce the trigger pulse is produced using a PWM with a period of [at least] 60 milliseconds and a high time of the desired 10 microseconds. After the trigger pulse, the sensor produces an 8 cycle sonic burst. The sonic wave travels through the air where it eventually bounces off an object and is reflected back to the sensor. When the sonic wave reaches the sensor, the sensor interrupts the echo and sends a pulse through the echo pin to the connected microcontroller. Input capture using a general-purpose timer was implemented, which essentially means the microcontroller is actively waiting for the rising edge of the echo pulse and records the time when it receives the rising edge. The MCU then waits for the falling edge to record the ending time. The total echo time is then computed by:

```
uint32_t echo_time_count = endTime - startTime;
```

If a timer wrap around occurs, the echo time is alternatively calculated by:

```
echo_time_count = (0xFFFFFFFF - startTime) + endTime + 1;
```

The elapsed time in microseconds is found using:

```
echo_time = (echo_time_count / 16000000) * 1000000;
// Echo_time_count is in cycles
// 16000000 is the clock frequency
// 1000000 is the number of microseconds in a second
// (cycles / (cycles/second)) * (microseconds/seconds) = microseconds
```

The distance in millimeters can then be calculated using the speed of sound.

```
distance_millimeters = (echo_time * v_sound * (1000/1) * (1/1000000)) / 2;
// v_sound is in meter/second, so it must be converted to millimeter/microsecond
// (1000/1) is millimeters/meter for conversion to millimeters
// (1/1000000) is seconds/microsecond for conversion to microseconds
// Divide by 2 because the echo is twice the distance because the echo is round trip
 ↪  travel time
```

After the echo time and distance equations were simplified, the code became the single line:

```
distance_millimeters = (((echo_time_count) / 16.0f) / 58.0f) * 10;
```

After the distance has been computed, it is then used to determine the angle at which to set the servo. The angle is calculated by using:

```
set_angle = ((float)distance_millimeters * DEG_OF_ROTATION) / 1000;
```

A conditional statement is implemented to check if the angle is between a set range, 0 to 180 degrees in this case, and checks if the distance is different from the previous one set. If both conditions are true, a new duty cycle is calculated using the equation:

```
duty_cycle = (((set_angle * (MAX_PULSE - MIN_PULSE)/DEG_OF_ROTATION) + MIN_PULSE) /
 ↪  SERVO_PERIOD);
```

The new duty cycle is set by loading the PWM comparator with the cycles corresponding to that duty cycle. The duty cycle in this case is set by:

```
PWM0_2_CMPB_R = (uint16_t)((*duty_cycle) * PWM_CYCLES_PER_US * SERVO_PERIOD);
```

For purposes of debugging and data analysis, every third of a second, the distance value is printed out. The prints reflected nothing out of the ordinary occurred during the prints. However, it is worth noting that it was observed that any time the distance was greater than 1000 millimeters, the servo stopped adjusting as the angle had maxed out where desired.

Concluding Remarks This lab presented many challenges, mostly with the ultrasonic sensor. The ultrasonic sensor operates at 5V supply and logic, while the TM4C123GH6PM uses 3.3V logic. The TM4C123 is 5V tolerant, but some 5v back-feeding on the trig pin was discovered, so a resistor was added between both the trigger on the MCU and the trigger on the ultrasonic sensor. Additionally, for stability purposes, a logic shifter was used on the echo signal to shift it from 5V to 3.3V. The ultrasonic sensor was also found to be unreliable when detecting soft objects such as skin or clothes, as they may absorb sound-waves, hence solid, smooth objects gave the most stable readings. The sonic sensor is also unreliable if sonic pulses are delivered near it's maximum rate.

```
#ifndef EENG461_LAB_4_MAIN_H
#define EENG461_LAB_4_MAIN_H


#include <driverlib/rom.h>
#define SYSCLKFREQ 16000000

int main (void);
void Disable_Interrupts(void);
void Enable_Interrupts(void);

#endif //EENG461_LAB_4_MAIN_H
```

Appendix - main.h

```
#include <stdbool.h>
#include "common/tm4c123gh6pm.h"
```

```c
#include "main.h"
#include "gpioCode.h"
#include "timerCode.h"
#include "pwmCode.h"
#include "sonicSensorCode.h"
#include "uart_print.h"

int main (void) {

    //printf UART setup
    //UARTConfigure(115200);
    setup_uart_printer();

    //GPIO/Switch Configuration
    GPIOConfigure();

    //Timer Configuration
    initTimers();

    //PWM Configuration
    PWMConfigure();

    //Ultrasonic Sensor Configuration
    ultrasonicConfigure();

    Enable_Interrupts(); //Enable Global Interrupts

    uint16_t last_distance = 0; //Last stored distance
    uint16_t distance_millimeters = 0;
    float set_angle = 0.0f;            // 0 <= set_angle <= 180
    float duty_cycle = 0.0f;
    uint32_t last_print_time = 0;

    while (1) {
        distance_millimeters = (((echo_time_count) / 16.0f) / 58.0f) * 10;  //uS/58
↪   = cm

        set_angle = ((float)distance_millimeters * DEG_OF_ROTATION) / 1000;

        /* If a duty cycle change occured, calculate new value and set pulse width
         ↪  */
        if (last_distance != distance_millimeters && (0 <= set_angle && set_angle <=
         ↪  DEG_OF_ROTATION)) {
            duty_cycle = (((set_angle * (MAX_PULSE - MIN_PULSE)/DEG_OF_ROTATION) +
↪   MIN_PULSE) / SERVO_PERIOD);
            ServoPWMSetDutyCycle(&duty_cycle);
            last_distance = distance_millimeters;
        }

        if(uptime_300ms != last_print_time) {
            printlf("[%d] Distance [mm]: %d | Angle: %f | Duty Cycle: %f\n\r",
```

```
                    uptime_300ms, distance_millimeters, &set_angle, &duty_cycle);
            last_print_time = uptime_300ms;
        }

    }

    return (0);
}

/*
 * Taken from Lab Assignment
 */
void Disable_Interrupts(void) {
    __asm ("  CPSID I\n");
}

void Enable_Interrupts(void) {
    __asm ("  CPSIE I\n");
}
```

Appendix - main.c

```
#ifndef EENG461_LAB_4_GPIOCODE_H
#define EENG461_LAB_4_GPIOCODE_H

#define GREEN_LED (1 << 3)
#define BLUE_LED (1 << 2)
#define RED_LED (1 << 1)
#define RGB_PINS GREEN_LED | BLUE_LED | RED_LED
#define SW1_PIN (1 << 4)

#define GPIO_PIN5 (1 << 5)

void GPIOConfigure(void);
void toggleRedLED(void);
void toggleBlueLED(void);
void toggleGreenLED(void);

#endif //EENG461_LAB_4_GPIOCODE_H
```

Appendix - gpioCode.h

```
#include <stdint.h>
#include <stdbool.h>
#include "common/tm4c123gh6pm.h"
#include "gpioCode.h"

void GPIOConfigure(void) {
    // Enable GPIO clock
```

```
    SYSCTL_RCGCGPIO_R |= SYSCTL_RCGCGPIO_R5;
    while(!(SYSCTL_PRGPIO_R & SYSCTL_PRGPIO_R5)) {};

    // Configure pins
    GPIO_PORTF_DEN_R |= RGB_PINS;
    GPIO_PORTF_DR8R_R |= RGB_PINS;

    // Set initial values
    GPIO_PORTF_DATA_R &= ~RGB_PINS; //All off to start

    // Set pin directions
    GPIO_PORTF_DIR_R |= RGB_PINS;

    //SW1 pullup
    GPIO_PORTF_PUR_R |= SW1_PIN;
    GPIO_PORTF_DEN_R |= SW1_PIN;

    //Enable interrupts on value of buttons
    GPIO_PORTF_IS_R &= ~SW1_PIN; //Edge triggered
    GPIO_PORTF_IBE_R |= SW1_PIN; //Both Edges
    GPIO_PORTF_IM_R |= SW1_PIN;  //Unmask the pin

    NVIC_EN0_R |= (1 << 30);      // Enable Port F interrupts in nvic
}

void toggleRedLED(void) {
    GPIO_PORTF_DATA_BITS_R[RED_LED] ^= RED_LED;
}

void toggleBlueLED(void) {
    GPIO_PORTF_DATA_BITS_R[BLUE_LED] ^= BLUE_LED;
}

void toggleGreenLED(void) {
    GPIO_PORTF_DATA_BITS_R[GREEN_LED] ^= GREEN_LED;
}
```

Appendix - gpioCode.c

```
#ifndef EENG461_LAB_4_TIMERCODE_H
#define EENG461_LAB_4_TIMERCODE_H

#include <stdint.h>
#include <stdbool.h>

extern volatile uint32_t uptime_milliseconds;
extern volatile uint32_t uptime_300ms;

void initTimers(void);
void ultraSonicEchoInCapTimerToggle(bool enable);
```

```
void timeKeeperISR (void);

#endif //EENG461_LAB_4_TIMERCODE_H
```

Appendix - timerCode.h

```
#include <stdint.h>
#include <stdbool.h>
#include "common/tm4c123gh6pm.h"
#include "timerCode.h"
#include "main.h"
#include "gpioCode.h"
#include "sonicSensorCode.h"

volatile uint32_t uptime_milliseconds;
volatile uint32_t uptime_300ms;

void initTimers(void) {

    /*
     * Millisecond Timer
     */
    SYSCTL_RCGCTIMER_R |= SYSCTL_RCGCTIMER_R0;                    //Enable Run
↪   Mode Clock Gating Control for Timer 0
    while (!(SYSCTL_PRTIMER_R & SYSCTL_RCGCTIMER_R0)) {}          //Wait for
     ↪  peripheral to be ready
    TIMER0_CTL_R &= ~TIMER_CTL_TAEN;                              //Disable Timer
    TIMER0_CFG_R = TIMER_CFG_32_BIT_TIMER;                        //32-bit Timer
    TIMER0_TAMR_R |= TIMER_TAMR_TAMR_PERIOD;                      //Set Timer to
↪   count down periodically
    TIMER0_TAILR_R = SYSCLKFREQ / 1000 - 1;                       //Set period for
↪   1ms
    TIMER0_TAPR_R = 0;                                            //Set prescaler
↪   to zero
    TIMER0_ICR_R |= TIMER_ICR_TATOCINT;                           //Clear
↪   Interrupt
    TIMER0_IMR_R |= TIMER_IMR_TATOIM;                             //Enable
↪   Interrupt as Timeout
    NVIC_EN0_R |= 1 << (INT_TIMER0A - 16);                        //Enable NVIC
↪   Interrupt
    TIMER0_CTL_R |= TIMER_CTL_TAEN;                               //Enable Timer

    /*
     * Timer for Ultrasonic Sesnor Echo Input Capture
     */
    SYSCTL_RCGCWTIMER_R |= SYSCTL_RCGCWTIMER_R2;                  //Enable Run
↪   Mode Clock Gating Control for Timer 2
    while (!(SYSCTL_PRWTIMER_R & SYSCTL_PRWTIMER_R2)) {}          //Wait for
     ↪  peripheral to be ready
    SYSCTL_RCGCGPIO_R |= SYSCTL_RCGCGPIO_R3;                      //Enable Run
↪   Mode Clock Gate Control for GPIO Port B
```

```
    while(!(SYSCTL_PRGPIO_R & SYSCTL_PRGPIO_R3)) {}                 //Wait for
     ↪  peripheral to be ready
    GPIO_PORTD_AFSEL_R |= (1 << 1);                                 //Set PB1 to
 ↪  alternate function
    GPIO_PORTD_DEN_R |= (1 << 1);                                   //Set PB1 to
 ↪  digital
    GPIO_PORTD_PCTL_R |= GPIO_PCTL_PD1_WT2CCP1;                     //Set PB1 to
 ↪  route to Timer 2
    WTIMER2_CTL_R &= ~TIMER_CTL_TBEN;                               //Disable Timer
    WTIMER2_PP_R |= TIMER_PP_SIZE_32;                               //Set timer to
 ↪  be a wide timer

    WTIMER2_CFG_R = TIMER_CFG_16_BIT;                               //32-bit wide
 ↪  timer for input capture
    WTIMER2_TBMR_R = (TIMER_TBMR_TBCMR | TIMER_TBMR_TBMR_CAP | TIMER_TBMR_TBCDIR);
                                                                    //Set Timer to
                                                                     ↪  count down
                                                                     ↪  once
    WTIMER2_CTL_R |= TIMER_CTL_TBEVENT_BOTH;                        //Trigger
 ↪  capture event on both edges
    WTIMER2_IMR_R = TIMER_IMR_CBEIM;                                //Enable
 ↪  Interrupt as Capture Event
    NVIC_EN3_R |= 1 << (INT_WTIMER2B - 16 - (32*3));                //Enable NVIC
 ↪  Interrupt
}

void ultraSonicEchoInCapTimerToggle(bool enable) {
    if (enable) {
        WTIMER2_ICR_R |= TIMER_ICR_CBECINT;                                 //Clear
 ↪  Interrupt
        WTIMER2_IMR_R = TIMER_IMR_CBEIM;
 ↪  //Enable Interrupt as Capture Event
        WTIMER2_CTL_R |= TIMER_CTL_TBEN;
 ↪  //Enable Timer
    } else {
        WTIMER2_CTL_R &= ~TIMER_CTL_TBEN;
 ↪  //Disable Timer
    }
}

void timeKeeperISR (void) {

    TIMER0_IMR_R &= ~TIMER_IMR_TATOIM;
 ↪  //Disable Interrupt
    TIMER0_ICR_R |= TIMER_ICR_TATOCINT;                                 //Clear
 ↪  Interrupt

    uptime_milliseconds++;

    if (uptime_milliseconds % 300 == 0) {
        uptime_300ms++;
```

```
    }

    TIMER0_IMR_R |= TIMER_IMR_TATOIM;                                    //Enable
 ↪  Interrupt
}
```

Appendix - timerCode.c

```
#ifndef EENG461_LAB_3_PWMCODE_H
#define EENG461_LAB_3_PWMCODE_H

#include <stdint.h>

#define RED_LED (1 << 1)
#define PWM_CYCLES_PER_MS 1000
#define PWM_CYCLES_PER_US 1

//Servo Globals
#define SERVO_PERIOD 20000     // 20ms period
#define MIN_PULSE 500         // -90deg pulse (~0.5ms pulse)
#define MAX_PULSE 2500         // 90deg pulse (~2.5ms pulse)
#define DEG_OF_ROTATION 180.0f // How many degrees the servo can rotate

void PWMConfigure(void);
void ServoPWMSetPeriod(uint16_t cycles_per_period);
void ServoPWMSetDutyCycle(float *duty_cycle);
void ServoPWMEnable(void);
void ServoPWMDisable(void);
void SonicTriggerPWMEnable(void);
void SonicTriggerPWMDisable(void);

#endif //EENG461_LAB_3_PWMCODE_H
```

Appendix - pwmCode.h

```
#include <stdint.h>
#include "common/tm4c123gh6pm.h"
#include "pwmCode.h"
#include "sonicSensorCode.h"

void PWMConfigure(void) {
    /*
     * Servo is connected to (PE5) is on M0PWM5G2
     */

    /*
     * Set PWM SysCtl Registers
     */
    SYSCTL_RCC_R |= SYSCTL_RCC_USEPWMDIV;
 ↪  //Use PWM Clock Divider
```

```
    SYSCTL_RCC_R = (SYSCTL_RCC_R & ~SYSCTL_RCC_PWMDIV_M) | SYSCTL_RCC_PWMDIV_16;
↪   //PWM Clock Divider at 16
    SYSCTL_RCGCPWM_R |= SYSCTL_RCGCPWM_R0;
↪   //Enable PWM Module 0
    while(!(SYSCTL_PRPWM_R & SYSCTL_PRPWM_R0)) {}
     ↪  //Wait for peripheral to be ready

    /*
     * PWM for Servo
     */
    SYSCTL_RCGCGPIO_R |= SYSCTL_RCGCGPIO_R4;                                    //Enable
↪   GPIO Port E
    while(!(SYSCTL_PRGPIO_R & SYSCTL_PRGPIO_R4)) {}                             //Wait
     ↪  for peripheral to be ready
    GPIO_PORTE_AFSEL_R |= (1 << 5);                                             //Set
↪   Alternate Function for PE5
    GPIO_PORTE_PCTL_R |= GPIO_PCTL_PE5_M0PWM5;                                  //Set
↪   Port Control to PE5 PWM value
    GPIO_PORTE_DEN_R |= (1 << 5);                                               //Set
↪   Digital Enable for PE5
    PWM0_2_CTL_R = 0x0;
↪   //Disable PWM0 Gen2
    PWM0_2_GENB_R = PWM_2_GENB_ACTCMPBD_ONE | PWM_2_GENB_ACTLOAD_ZERO |
↪   PWM_2_GENB_ACTZERO_ONE;
                                                                                //Set
                                                                                 ↪  PWM1
                                                                                 ↪  Gen2-B
                                                                                 ↪  comparator,
                                                                                 ↪  load,
                                                                                 ↪  and
                                                                                 ↪  zero
    PWM0_2_LOAD_R = PWM_CYCLES_PER_US * SERVO_PERIOD;                           //Set
↪   PWM Gen2 Period to 20ms
    PWM0_2_CMPB_R = PWM_CYCLES_PER_US * (MAX_PULSE-MIN_PULSE) / 2;              //Set
↪   initial duty cycle to ~90deg

    /*
     * PWM for Ultrasonic Trigger
     */
    SYSCTL_RCGCGPIO_R |= SYSCTL_RCGCGPIO_R1;                                    //Enable
↪   Run Mode Clock Gate Control for GPIO Port B
    while(!(SYSCTL_PRGPIO_R & SYSCTL_PRGPIO_R1)) {}                             //Wait
     ↪  for peripheral to be ready
    GPIO_PORTB_AFSEL_R |= (1 << 4);                                             //Set
↪   Alternate Function for PB4
    GPIO_PORTB_PCTL_R |= GPIO_PCTL_PB4_M0PWM2;                                  //Set
↪   Port Control to PB4 PWM value
    GPIO_PORTB_DEN_R |= (1 << 4);                                               //Set
↪   Digital Enable for PB4
    PWM0_1_CTL_R = 0x0;
↪   //Disable PWM0 Gen1
```

```
    PWM0_1_GENA_R = PWM_1_GENA_ACTCMPAD_ONE | PWM_1_GENA_ACTLOAD_ZERO |
↪   PWM_1_GENA_ACTZERO_ONE;
                                                                        //Set
                                                                        ↪  PWM1
                                                                        ↪  Gen1-A
                                                                        ↪  comparator,
                                                                        ↪  load,
                                                                        ↪  and
                                                                        ↪  zero
    PWM0_1_LOAD_R = PWM_CYCLES_PER_MS * SONIC_TRIGGER_PERIOD_MS;        //Set
↪   PWM Gen1 Period to 60ms
    PWM0_1_CMPA_R = PWM_CYCLES_PER_US * SONIC_TRIGGER_PULSE_WIDTH_uS;   //Set ON
↪   time to 10uS

    /*
     * Enable PWMs
     */
    ServoPWMEnable();                                                   //Enable
↪   Servo PWM
    SonicTriggerPWMEnable();                                            //Enable
↪   Sonic Trigger PWM
}

void ServoPWMSetPeriod(uint16_t cycles_per_period) {
    ServoPWMDisable();
↪   //Disable PWM
    PWM0_2_LOAD_R = cycles_per_period;
↪   //Set new period
    ServoPWMEnable();
↪   //Enable PWM
}

void ServoPWMSetDutyCycle(float *duty_cycle) {
    ServoPWMDisable();
↪   //Disable PWM
    PWM0_2_CMPB_R = (uint16_t)((*duty_cycle) * PWM_CYCLES_PER_US * SERVO_PERIOD);
↪   //Set new duty cycle
    ServoPWMEnable();
↪   //Enable PWM
}

void ServoPWMEnable(void) {
    PWM0_2_CTL_R |= PWM_2_CTL_ENABLE;                                   //Enable
↪   PWM0 Gen 2
    PWM0_ENABLE_R |= PWM_ENABLE_PWM5EN;                                 //Enable
↪   PWM0 Output 5
}

void ServoPWMDisable(void) {
    PWM0_2_CTL_R &= ~PWM_2_CTL_ENABLE;
↪   //Disable PWM0 Gen 2
```

```
    PWM0_ENABLE_R &= ~PWM_ENABLE_PWM5EN;
 ↪  //Disable PWM0 Output 5
}

void SonicTriggerPWMEnable(void) {
    PWM0_1_CTL_R |= PWM_1_CTL_ENABLE;                                    //Enable
 ↪  PWM0 Gen 2
    PWM0_ENABLE_R |= PWM_ENABLE_PWM2EN;                                  //Enable
 ↪  PWM0 Output 5
}

void SonicTriggerPWMDisable(void) {
    PWM0_1_CTL_R &= ~PWM_1_CTL_ENABLE;
 ↪  //Disable PWM0 Gen 2
    PWM0_ENABLE_R &= ~PWM_ENABLE_PWM2EN;
 ↪  //Disable PWM0 Output 5
}
```

Appendix - pwmCode.c

```
#ifndef EENG461_LAB_4_SONICSENSORCODE_H
#define EENG461_LAB_4_SONICSENSORCODE_H

#include <stdint.h>
#include "common/tm4c123gh6pm.h"

#define SONIC_TRIGGER_PERIOD_MS 80
#define SONIC_TRIGGER_PULSE_WIDTH_uS 10

extern volatile uint32_t echo_time_count;
extern volatile uint32_t startTime;
extern volatile uint32_t endTime;

void ultrasonicConfigure(void);
void sonicEchoISR();

#endif //EENG461_LAB_4_SONICSENSORCODE_H
```

Appendix - sonicSensorCode.h

```
#include "common/tm4c123gh6pm.h"
#include "sonicSensorCode.h"
#include "timerCode.h"
#include "gpioCode.h"

volatile uint32_t echo_time_count;
volatile uint32_t startTime;
volatile uint32_t endTime;
static enum {FALLING, RISING} prev_echo_edge = FALLING;
```

```
void ultrasonicConfigure(void) {

    ultraSonicEchoInCapTimerToggle(true);                    //Enable Input Capture
 ↪  Timer

}

void sonicEchoISR() {
    WTIMER2_IMR_R &= ~TIMER_IMR_CBEIM;                                    //Disable
 ↪  Interrupt
    WTIMER2_ICR_R |= TIMER_ICR_CBECINT;                                   //Clear
 ↪  Interrupt

    /*
     * Check whether this trigger is the rising edgge or falling edge.
     * If rising edge, record the time as start time
     * If falling edge, record the time as end time and calculate total time
 ↪  betwewen rising and falling edge
     * The previous echo edge is set to the current edge type so the next trigger
 ↪  checks the other edge type.
     */
    const enum {FALLING, RISING} echo_state = (GPIO_PORTD_DATA_BITS_R[(1 << 1)] ==
     ↪  (1 << 1)) ? RISING : FALLING;

    /*
     * Early exit for unchanged states
     */
    if ((echo_state == RISING && prev_echo_edge == RISING) || (echo_state == FALLING
     ↪  && prev_echo_edge == FALLING)) {
        WTIMER2_IMR_R |= TIMER_IMR_CBEIM;
 ↪  //Re-enable Interrupt
        return;
    }

    switch (echo_state) {
        case RISING:
            // Record the time of the rising edge
            startTime = WTIMER2_TBR_R;
            prev_echo_edge = RISING; //If edge rising, the previous state will be
 ↪  set to rising (current state)
            break;
        case FALLING:
            // Record the time of the falling edge
            endTime = WTIMER2_TBR_R;
            prev_echo_edge = FALLING; //If edge falling, the previous state will be
 ↪  set to falling (current state)
            // Calculate the elapsed time
            if (endTime > startTime) {
                echo_time_count = endTime - startTime;
            } else {
```

```
                // Timer overflow occurred
                echo_time_count = (0xFFFFFFFF - startTime) + endTime + 1;
            }
            break;
    }

    WTIMER2_IMR_R |= TIMER_IMR_CBEIM;
 ↪  //Re-enable Interrupt

}
```

Appendix - sonicSensorCode.c

```
void setup_uart_printer(void);

/*!
 *  Supports %s (string), %d (signed decimal), %u (unsigned decimal), and %f (float)
 * FLOATS SHOULD BE PASSED AS A POINTER TO FLOAT - THIS IS DUE TO A C LANGUAGE
 ↪  LIMITATION
 * COMBINED WITH NO HARDWARE (double)
 * Floats are limited in magnitude to UINT32_MAX
 * Floats only print a fixed 3 decimal places
 * */
void printlf(char format[], ...);

void print_string(const char * const str);
```

Appendix - uart_print.h

```
#include <stdarg.h>
#include <stdbool.h>
#include <stdint.h>
#include <common/tm4c123gh6pm.h>
#include <driverlib/rom.h>
#include <driverlib/uart.h>
#include <driverlib/gpio.h>
#include <driverlib/sysctl.h>
#include <inc/hw_uart.h>
#include <inc/hw_sysctl.h>
#include <inc/hw_memmap.h>

#define INFINITY 1.0f/0.f

void setup_uart_printer(void){
    ROM_SysCtlClockSet(SYSCTL_SYSDIV_12_5 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
 ↪  SYSCTL_XTAL_16MHZ);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
    while(!ROM_SysCtlPeripheralReady(SYSCTL_PERIPH_UART0)){};
```

```
    ROM_GPIOPadConfigSet(GPIO_PORTA_BASE, 3, GPIO_STRENGTH_8MA, GPIO_PIN_TYPE_STD);
    ROM_GPIODirModeSet(GPIO_PORTA_BASE, 3, GPIO_DIR_MODE_HW);
    ROM_UARTConfigSetExpClk(UART0_BASE, ROM_SysCtlClockGet(), 9600,
                            (UART_CONFIG_WLEN_8 | UART_CONFIG_STOP_TWO |
                             ↪  UART_CONFIG_PAR_NONE));
}

void putchar(char c) {
    static int chars_sent_recently = 0;
    ROM_UARTCharPut(UART0_BASE, c);
    while(ROM_UARTBusy(UART0_BASE)) {};

    //Really dumb, but avoid overwhelming ICDI
    if(++chars_sent_recently > 7) {
        chars_sent_recently = 0;
        for(int i = 0; i < 150; i++) {
            __asm("mov r1,r1\n");
        }
    }
}

void print_string(const char * const str) {
    for(int i = 0; str[i] != '\0'; i++) {
        putchar( str[i]);
    }
}

void print_unsigned_decimal(uint32_t num){

    char buf[11]; //Large enough to fit any value of num

    int places = 0;

    do {
        buf[places++] = (char)('0' + (num % 10));
        num /= 10;
    } while (num > 0);

    for(; places; places--) {
        putchar( buf[places-1]);
    }
}

void print_decimal(int32_t num){

    if (num < 0) {
        putchar( '-');
        num = 0 - num;
    }
```

```
    print_unsigned_decimal((uint32_t)num);
}

void print_float(float number) {

    if(number != number) { //NaN is not equal to anything, including NaN
        print_string("NaN");
        return;
    }

    if(number == INFINITY || number == -INFINITY) {
        print_string("inf");
        return;
    }

    if(number > UINT32_MAX || (0-number) > UINT32_MAX) {
        print_string("[out of range]");
        return;
    }

    char buf[15]; // large enough to fit any value

    if(number < 0) {
        putchar( '-');
        number = 0 - number;
    }

    uint32_t integerPart = (uint32_t)number;
    uint32_t decimalPart = (uint32_t)((number - integerPart) * 1000);

    int places = 0;

    for(int i = 0; i < 3; i++) {
        buf[places++] = (char)('0' + decimalPart % 10);
        decimalPart /= 10;
    };

    buf[places++] = '.';

    print_unsigned_decimal(integerPart);

    for(; places; places--) {
        putchar( buf[places-1]);
    }
}

void printlf(char format[], ...) {
    va_list args;
    va_start(args, format);

    char *str;
```

```
    int32_t num;
    uint32_t numu;
    float *numf;

    for(int i=0; format[i] != '\0'; i++) {
        switch(format[i]) {
            case '%' :
                i++;
                switch(format[i]) {
                    case 'u': //unsigned decimal number
                        numu = va_arg(args, uint32_t);
                        print_unsigned_decimal(numu);
                    break;

                    case 'd': //signed decimal number
                        num = va_arg(args, int32_t);
                        print_decimal(num);
                    break;

                    case '\0': // End of string
                        putchar( '%');
                        i--; //let the for loop catch this
                    break;

                    case 's': //string
                        str = va_arg(args, char*);
                        print_string(str);
                    break;

                    case 'f': //float
                        numf = va_arg(args, float *);
                        print_float(*numf);
                    break;

                    default: //Not recognized
                    putchar( '%');
                    putchar( format[i]);
                    break;
                }
            break;

            default:
                putchar( format[i]);
            break;
        }
    }

    va_end(args);
}
```

Appendix - uart_print.c