Ultraviolet LED Biofouling Mitigation

A Senior Project by Andrew Lam

Advised by Dr. Bridget Benson

Computer Engineering Department

California Polytechnic State University, San Luis Obispo

June 10th, 2014

Table of Contents

Table of Contents 2
List of Figures 4
List of Tables 5
Acknowledgements 6
Abstract 7
I. Introduction 8
II. Background 9
 a) Biofouling 9
 b) Ultraviolet Light and LEDs 9
 c) Pulse Width Modulation (PWM) 9
III. Requirements 10
IV. Specifications 11
 a) System Overview 11
 b) Development Board - MSP-EXP430F5438 12
 c) PWM UV LED Breakout Interface 13
 d) Submergible UV LED Array 14
V. Design 15
 a) System Architecture: 15
 b) Hardware 15
  1. Circuit Design 15
  2. Selecting MOSFETs 16
  3. Selecting Current-Limiting Resistors 16
  4. Schematic 16
  5. Layout 18
  6. Safety 19
 c) Software 20
  1. Overview 20
  2. Programming Methodology 20
  3. PWM Signal Generation 21
  4. UART Command Line Interface 22
VI. Testing 23
 a) Software Testing 23
 b) Hardware Testing 23
 c) Waterproof Testing 24
VII. Preliminary Qualitative Experiment 25

a) Overview 25
b) Location 25
c) Deployment 26
d) Results – Week 1 to 3 27
1. 405nm, 30 degree, 3mm Package – $14.36 28
2. 390nm, 10 degree, 5mm Package - $1.59 29
3. 385nm, 30 degree, 3mm Package - $1.00 30
4. 365nm, 10 degree, 5mm Metal Encased Package - $13.75 31
e) Observations and Analysis 32
VIII. Conclusion / Future Work 33
Bibliography 34
Appendix 35
a) Analysis of Senior Project Design 35
1. Summary of Functional Requirements 35
2. Primary Constraints 35
3. Economic 35
4. Manufacturability 35
5. Sustainability 35
6. Ethical 36
7. Health and Safety 36
8. Social and Political 36
9. Development 36
b) Bill of Materials (Breakout Interface) 37
c) Source Code – Preliminary Experiment (1 of 1 file) 38
d) My Source Code with UI, RTC, UART (1 of 3 files) 39
e) Source Code –Open Source UART Driver by Gustavo Litovsky (2 of 3 Files) 44
f) Source Code –Open Source UART Driver by Gustavo Litovsky (3 of 3 Files) 47

List of Figures

Figure 1: Biofouling.......... 9
Figure 2 : 3mm, 5mm, and 5mm Metal-Encased UV LED Packages 9
Figure 3: Pulse Width Modulation of LED Brightness.......... 9
Figure 4: Experimental Setup 11
Figure 5: TI MSPEXP430F5438 Experimenter Board 12
Figure 6: PWM UV LED Breakout Interface 13
Figure 7: UV LED Array Encased in Borosilicate Glass Tube 14
Figure 8: Black Box Diagram for the Experimental Setup 15
Figure 9: Dimming Circuit for 1 LED 15
Figure 10: Eagle Schematic for the PWM UV LED Breakout Interface 17
Figure 11: Eagle Board Layout for PWM UV LED Breakout Interface 18
Figure 12: Assembled Interface with External Connections.......... 19
Figure 13: Top-Level Software Flow Chart.......... 20
Figure 14: Prototype Command Line Interface in PuTTY 22
Figure 15: Testing the UV LED Array for Functionality. 23
Figure 16: PVC Pressure Fitting on Glass Tube 24
Figure 17: Raw Seawater System at Cal Poly CCMS.......... 25
Figure 18: Setup Deployment in the Raw Seawater System.......... 26
Figure 19: 405nm, Week 1.......... 28
Figure 20: 405nm, Week 3.......... 28
Figure 21: 390nm, Week 1.......... 29
Figure 22: 390nm, Week 3.......... 29
Figure 23: 385nm, Week 1.......... 30
Figure 24: 385nm, Week 3.......... 30
Figure 25: 365nm, Week 1.......... 31
Figure 26: 365nm, Week 3.......... 31

List of Tables

Table 1: Component List for PWM UV LED Breakout Interface 19
Table 2: Duty Cycle Accuracy Testing 23

Acknowledgements

This project was made possible through the help of many other individuals. First, I would like to thank Thom Maughan from the Monterey Bay Aquarium Research Institute for sponsoring this project and mentoring me. Next I would like to thank Dr. Jianbiao Pan from the Industrial and Manufacturing Engineering Department for giving me the opportunity to manufacture a portion of my project in his course. Also, I would like to thank Tom Moylan and Jason Felton from the Center for Costal Marine Sciences for enabling my system deployment at the Cal Poly Pier. Last but not least, I would like to thank Dr. Bridget Benson from the Computer Engineering Department for advising me and pointing me to the right resources throughout this project.

Abstract

The goal is to determine if low-cost UV LEDs can mitigate marine biofouling on small glass or acrylic camera lenses. A microprocessor-controlled experimental setup to control the illumination of low-cost UV LEDs of various wavelength and packaging was fashioned. The system consists of a programmed microcontroller, a manufactured LED breakout interface, and a submergible UV LED array enclosed in a borosilicate glass tube. A preliminary qualitative assessment of four different UV LEDs was conducted during a three-week deployment of the experimental setup in the raw seawater system at the Cal Poly Center for Costal and Marine Sciences in Avila Beach, California. Although some evidence from testing suggests that UV LEDS can mitigate biofouling, a longer qualitative test and an additional quantitative test would have to be conducted for conclusive findings.

I. Introduction

Thom Maughan, an Embedded Software Engineer at the Monterey Bay Aquarium Research Institute (MBARI), has a camera that is to be deployed on the backs of great white sharks (SharkCafeCam). Like other man-made marine instrumentation, the camera lens will be subject to biofouling during deployment. Avoiding mechanical solutions to prevent lens biofouling, the proposed solution is to design and manufacture a device that schedules UV illumination on the lens.

Prior to manufacturing a device to be deployed with the camera, studies are needed in order to determine which UV LEDs are effective in mitigating biofouling from camera lenses. Thus, this project consists of the design, manufacturing, and fabrication of a microprocessor-controlled system that enables the illumination of various UV LEDs of different wavelengths at various luminous intensities. The setup was deployed in a marine setting to begin a preliminary qualitative study on a glass surface similar to that of camera lenses.

II. Background

a) Biofouling

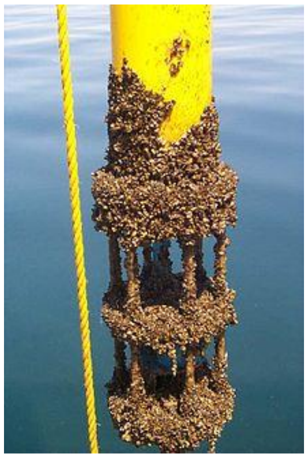


Figure 1: Biofouling

Biofouling, or biological fouling, is the accumulation of microorganisms such as plants, algae, and animals on wetted surfaces. This natural occurrence is a problem that persists today among many applications of marine instrumentation. Engineers and marine biologists at MBARI regularly face biofouling when running experiments at sea.

b) Ultraviolet Light and LEDs

UV light spans the light spectrum in a range between 400nm and 100nm. In the past, experiments have suggested that UV radiation can have a significant effect on larval settlement of barnacles [1]. Preliminary testing in this project utilizes four types of LEDs in the UVA spectrum. LEDs have relatively low power draw - which makes them a good candidate for use with embedded devices.

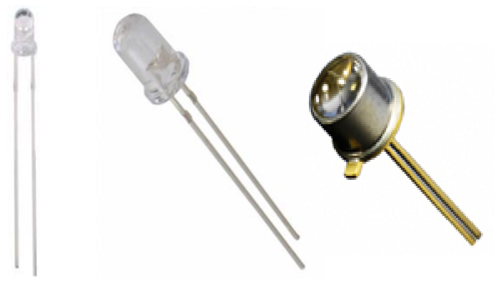


Figure 2 : 3mm, 5mm, and 5mm Metal-Encased UV LED Packages

c) Pulse Width Modulation (PWM)

PWM refers to modulating how long a digital signal is logically high (the duty cycle) based on a reference clock signal. PWM can effectively modulate the average voltage of a signal and can be used vary the brightness of LEDs.

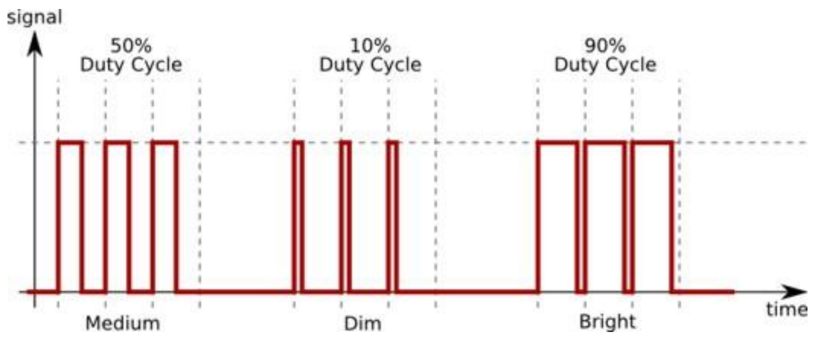


Figure 3: Pulse Width Modulation of LED Brightness

III. Requirements

In order to begin a preliminary qualitative test with UV LEDs, the system was defined to meet the criteria below.

The experimental setup shall:

- Be partially submergible in seawater
- Operate for at least a week without needed human supervision
- Illuminate multiple UV LEDs simultaneously on a glass surface
- Provide low, medium, and high settings for illumination
- Have reconfigurable illumination settings

IV. Specifications

a) System Overview

The system is best described as three parts: a development board, a PWM UV LED breakout interface, and a submergible UV LED array. Together, these three components complete the qualitative experimental setup.

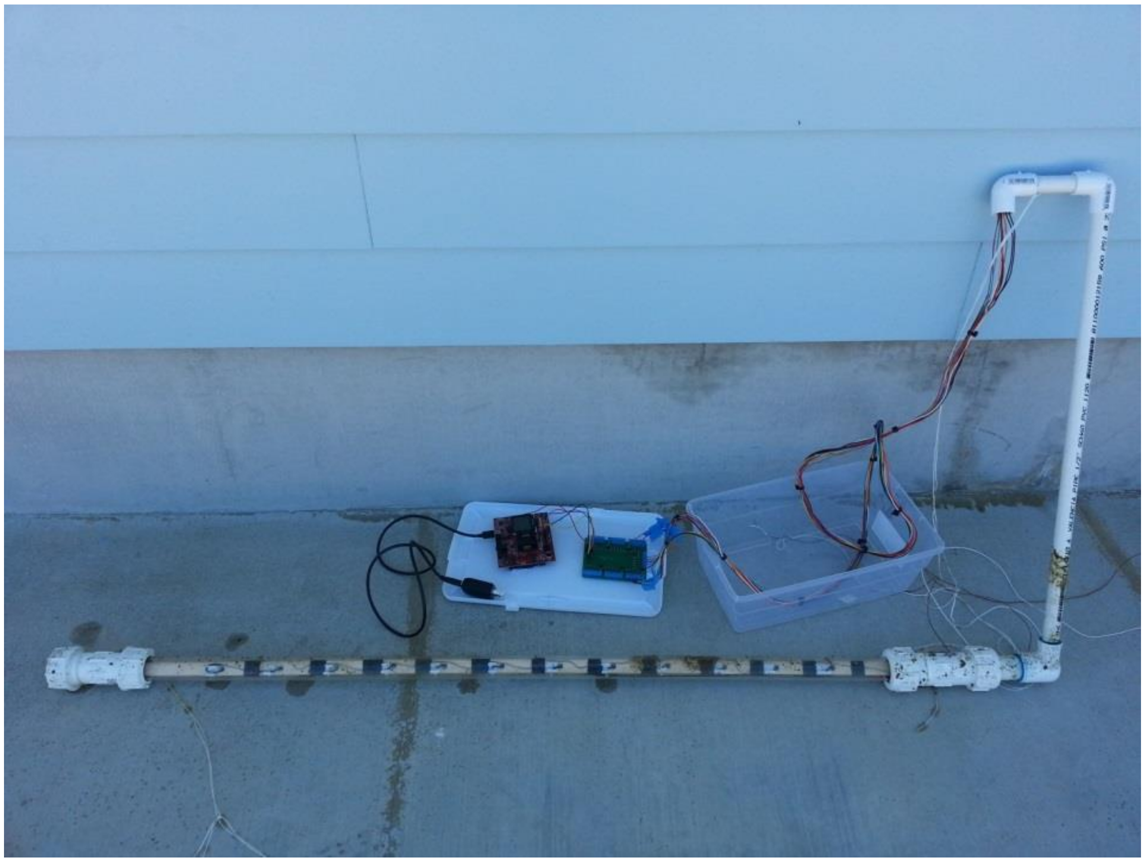


Figure 4: Experimental Setup

b) **Development Board - MSP-EXP430F5438**

To provide illumination settings, the experimental setup utilizes a Texas Instruments MSP-EXP430F5438 Experimenter Board. This board was chosen because it was ready available from MBARI and it utilizes the familiar Code Composer Studio for software development.

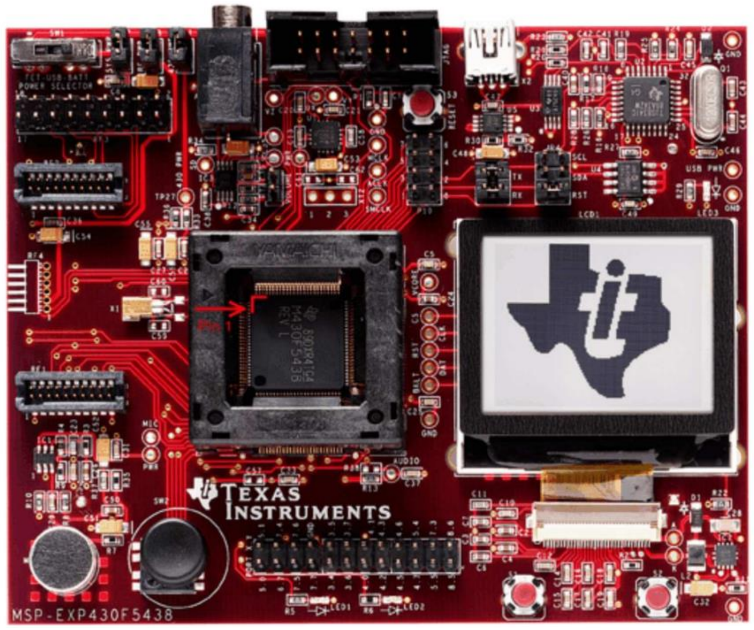


Figure 5: TI MSPEXP430F5438 Experimenter Board

Specifications of interest for the Experimenter Board include:

- MSP430F8438A microcontroller
- Seven PWM output pins
    - $V_{peak} = 3.3V$
- UART communication over mini-USB connection
    - Programmable baud rate
    - Independent interrupt capability for receive and transmit
- Real Time Clock

The full list of specifications for the MSP430F8438A can be found in the manufacturer documentation [2].

c) **PWM UV LED Breakout Interface**

Designed to interface with the Experimenter Board, the UV LED breakout interface provides robust circuitry to power and dim up to 28 external UV LEDs. In the preliminary usage of this interface, 12 UV LEDs are powered.

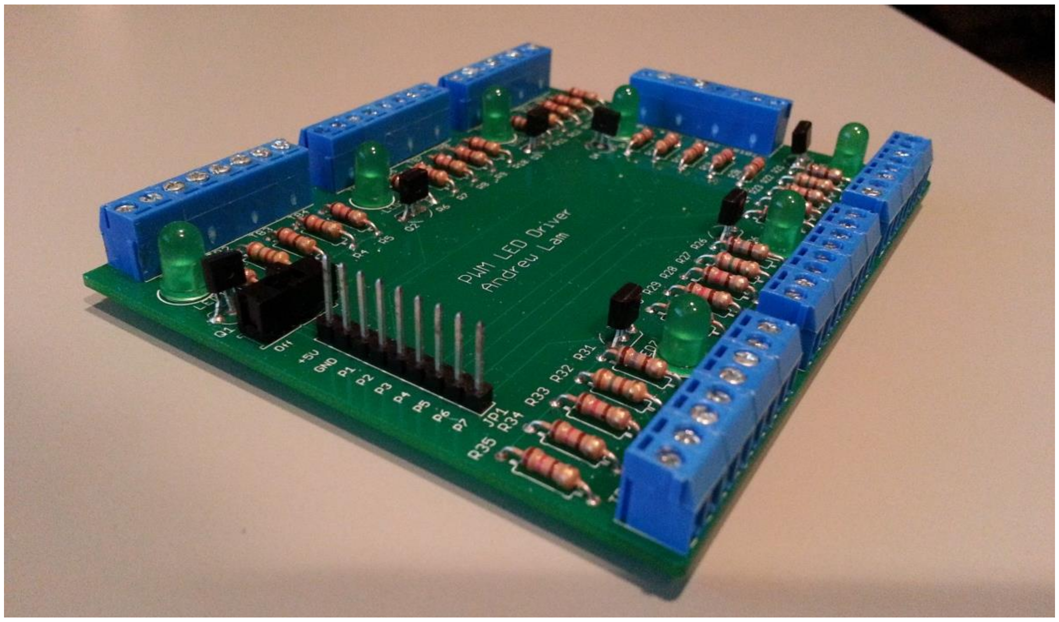


Figure 6: PWM UV LED Breakout Interface

Specifications for the PWM UV LED Breakout Interface include:

- Dimensions: 10mm x 8mm
- +5V power supply input
- Common GND input from a microcontroller
- 7 PWM signal inputs from a microcontroller
    - Requires at least $2V_{peak}$
- Terminal Block outputs for up to 28 UV LEDs
    - 7 of each UV LED: 356nm, 385nm, 390nm, 405nm
- Can accommodate up to7 illumination settings per LED type

d) **Submergible UV LED Array**

The third component of the experimental setup is the UV LED Array, which was submerged underwater to observe how the system could mitigate biofouling on its borosilicate glass tube encasing.

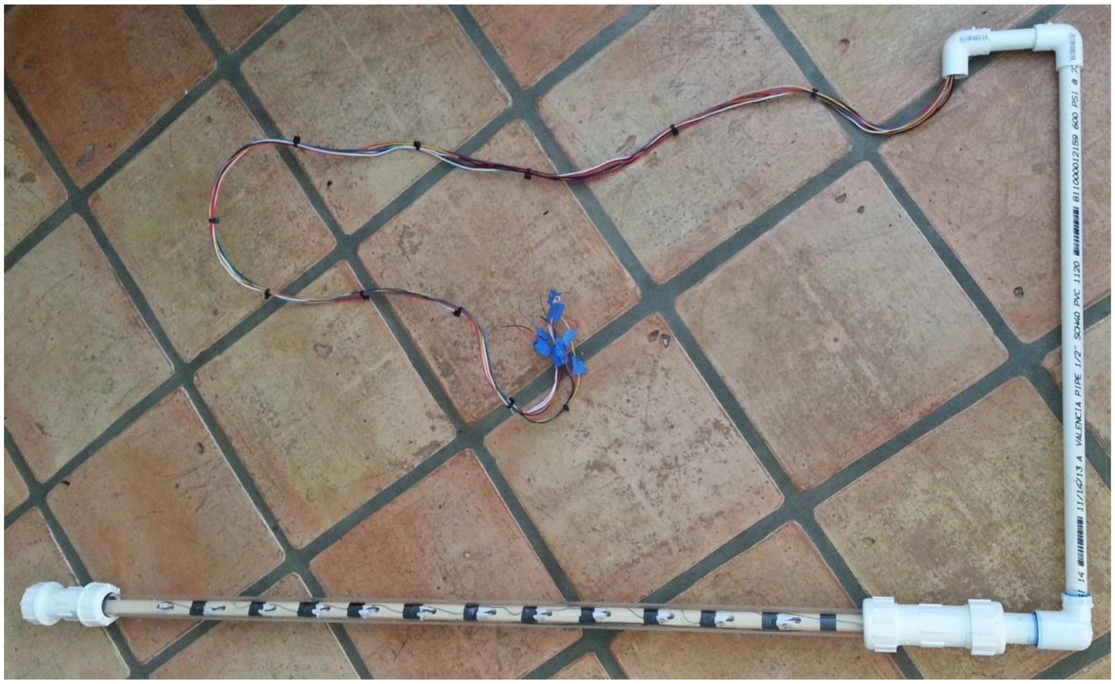


Figure 7: UV LED Array Encased in Borosilicate Glass Tube

Specifications for the UV LED Array include:

- Dimensions: 55in x 28in
- 12 UV LEDs
    - 3 of each: 356nm, 385nm, 390nm, 405nm (various packages)
    - 3” spacing in between each LED
- 1 common +5V connection (LED anodes)
- 12 separate LED cathode connections
- Waterproofed by PVC pressure fittings and blue PVC cement

V. Design

a) System Architecture:

The black box diagram below provides an overview of the IO functions in the system.

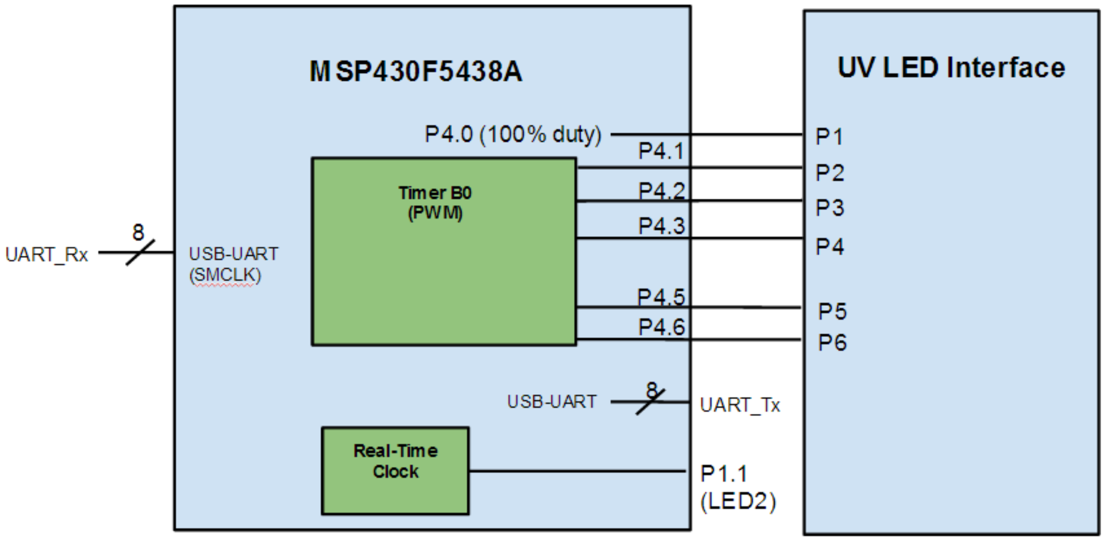


Figure 8: Black Box Diagram for the Experimental Setup

b) Hardware

1. Circuit Design

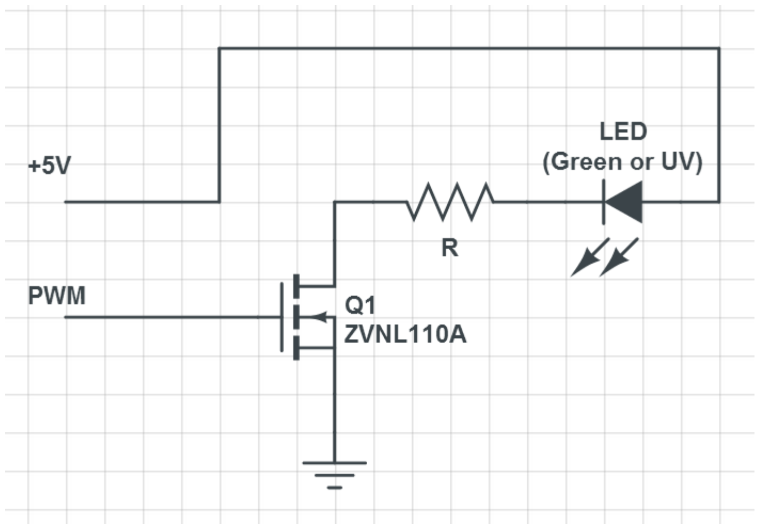


Figure 9: Dimming Circuit for 1 LED

The circuit above illustrates the logic behind varying the luminous intensity of an LED. When a PWM input toggles the gate of the MOSFET low, the series connection between the +5V voltage source, LED, current-limiting resistor, and ground is shorted. When done at high frequency, the LEDs visually exhibit a strobe effect, and their average power can be modulated.

2. Selecting MOSFETs

ZVNL110A N-Channel MOSFETs have a max drain-source current rating of 320mA, which makes them ideal for powering 5 LEDs at max power. Also, the MOSFETS have a gate-source threshold voltage of 1.5V at 1mA, which makes them ideal for switching with 3.3V microcontroller PWM inputs.

3. Selecting Current-Limiting Resistors

The current-limiting resistors for each LED (green, 365nm, 485nm, 490nm, 405nm) were chosen so that at 100% duty cycle, each LED would operate at about 20% below their recommended forward current in ideal conditions (as stated in the datasheets). The formula for determining current-limiting resistance R is as follows:

$$R=\frac{Vs-Vf}{i}$$

Where:
V_s = the supply voltage
V_f = the forward voltage drop of the LED in Volts
i = the LED forward current in Amps

4. Schematic

Expanding on the previous circuit shown in figure 9, each PWM input was paired with a ZVNL110A MOSFET, which powered 5 corresponding green and UV LEDs in parallel. The schematic, shown in figure 10, was created with the aid of CadSoft Eagle 6.5.0 software.

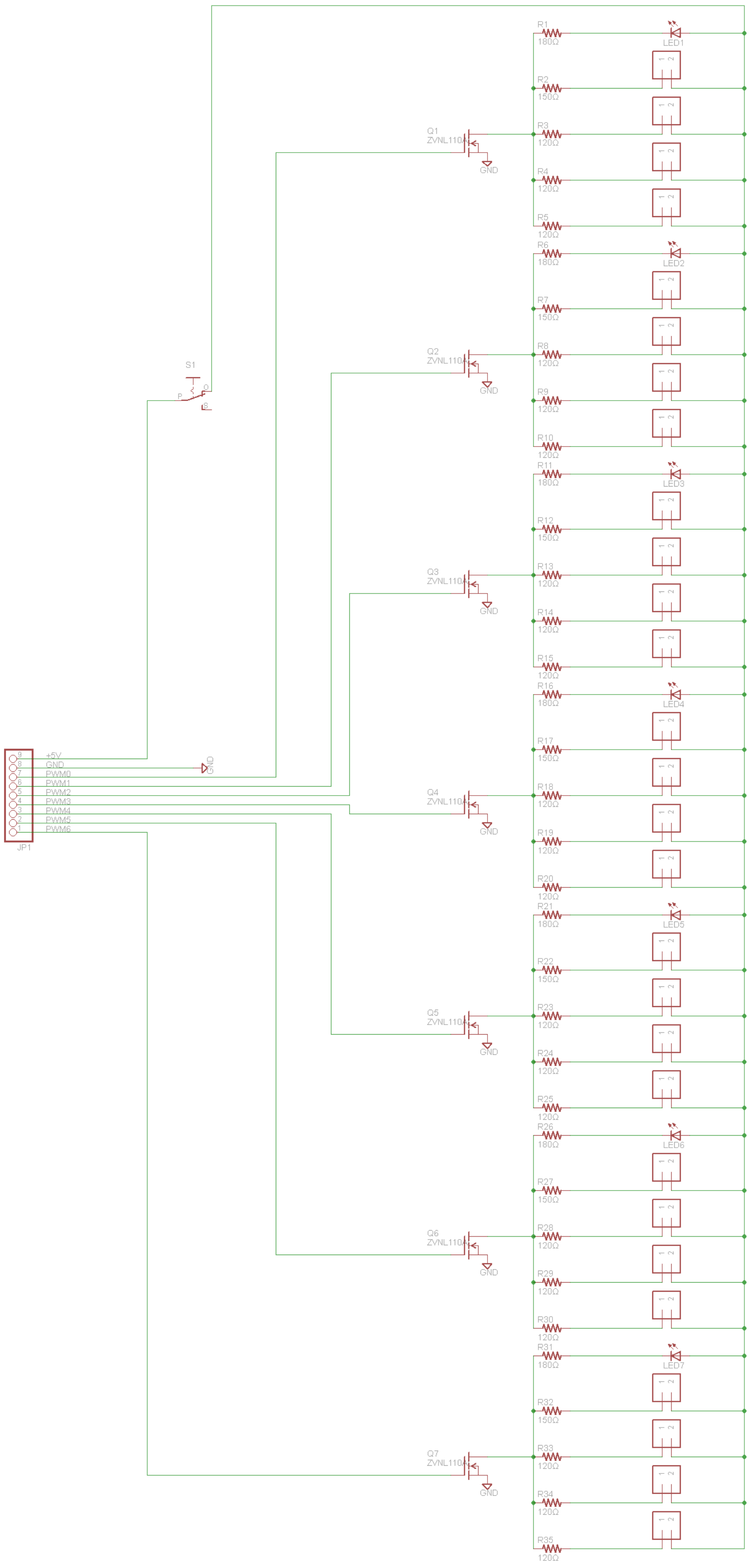


Figure 10: Eagle Schematic for the PWM UV LED Breakout Interface

5. Layout

The 2-layer Eagle PCB layout for the interface is shown in figure 11. Being part of an experimental setup, the size of the board was not a large factor – the interface utilizes the maximum board size from the Eagle 6.5.0 freeware license. Rather, speed and ease of assembly were more important factors. For this reason, through-hole component packages were chosen for the entire board and component footprints were gathered from existing footprint libraries such as the Adafruit Eagle Footprint Library.

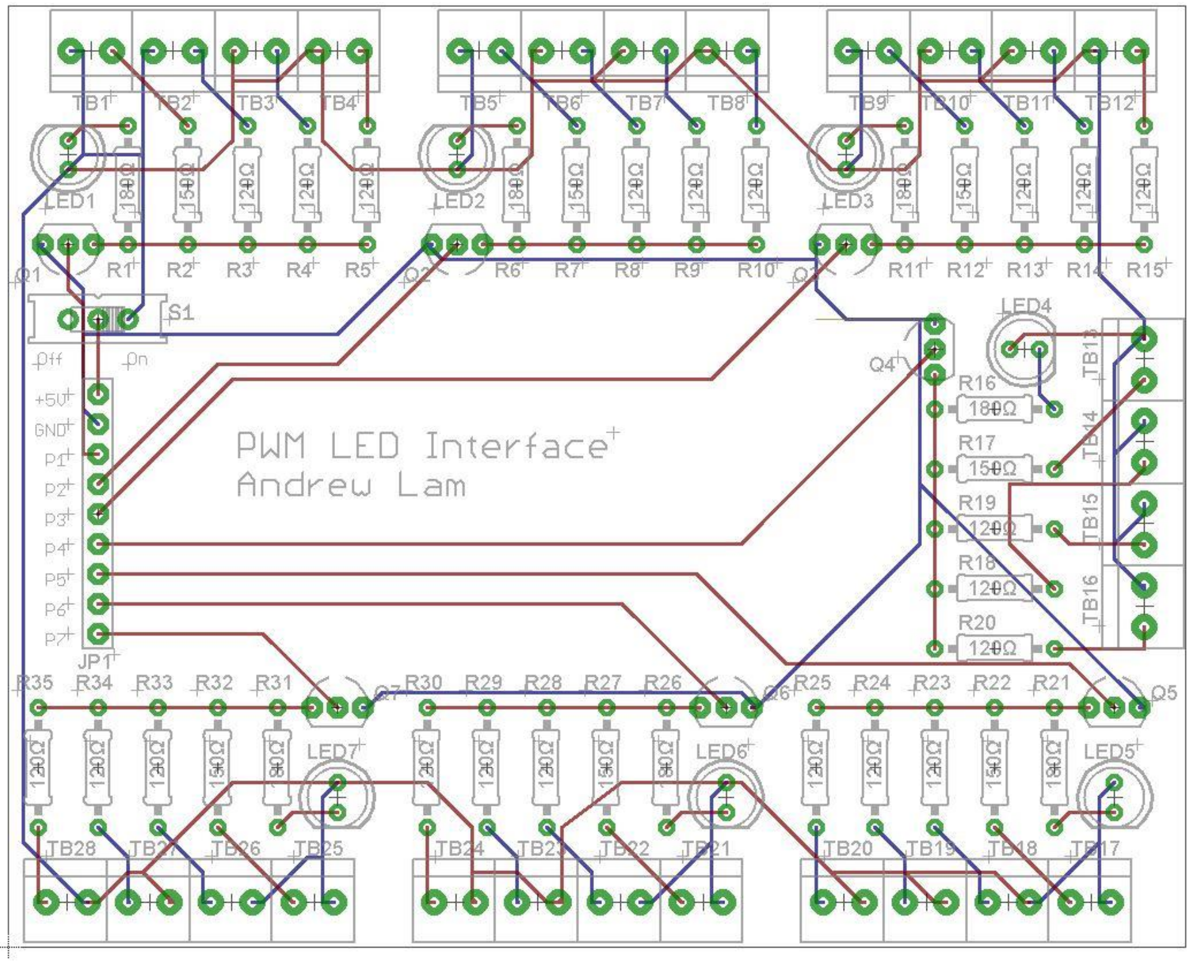


Figure 11: Eagle Board Layout for PWM UV LED Breakout Interface

In the layout, the red lines are the top-side copper traces and the blue lines are the bottom-side copper traces. On-board LEDs were placed to correspond to the UV LEDs that share a PWM signal. The components corresponding to the silkscreen reference designators are listed in table 1.

Quantity	Reference Designator	Description	Value / Rating	Package Info
7	LEDx	On-Board Green LED	565nm, 30deg	Radial Through-Hole
7	Rx	Carbon Film Resistor	180Ω, 1/4W, ±5%	Axial Through-Hole
7	Rx	Carbon Film Resistor	150Ω, 1/4W, ±5%	Axial Through-Hole
21	Rx	Carbon Film Resistor	120Ω, 1/4W, ±5%	Axial Through-Hole
7	Qx	ZVNL110A N-Channel MOSFET	320mA, 1.5V(th)	TO92-3 Through-Hole
1	S1	EG1218 Slide Switch	0.2A @ 30VDC	Through-Hole
28	TBx	Terminal Block	2 Position @ 3.5mm Pitch	Through-Hole
1	JP1	Wire Header	9 Position @.1" spacing	Through-Hole

Table 1: Component List for PWM UV LED Breakout Interface

6. Safety

The green on-board LEDs serve as both a safety measure and debugging tool. Working with certain wavelengths of ultraviolet light is harmful to living organisms – therefore it is advised to reference the corresponding green LED when configuring a UV LED's luminous intensity. For additional safety, an on-off switch that toggles the +5V power supply for all LEDs was included on the interface.

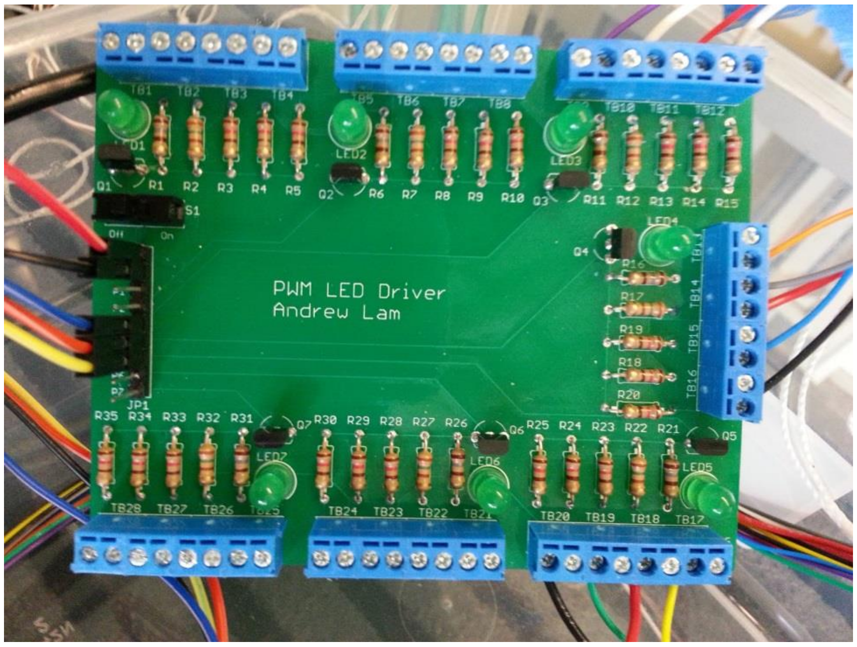


Figure 12: Assembled Interface with External Connections

c) Software

1. Overview

The top level software is expressed in figure 13. After initializations, the program enters the main while loop - where it has the option to read and handle input from a command line interface, update duty cycles, and toggle an on-board LED on a ~1second interval.

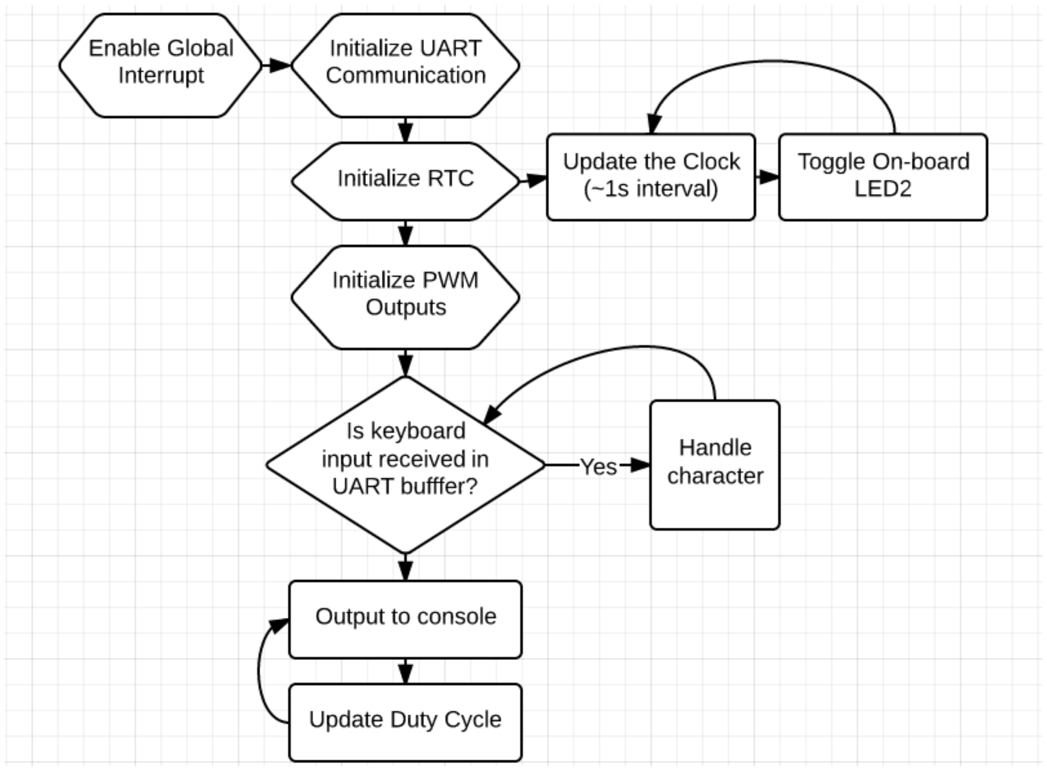


Figure 13: Top-Level Software Flow Chart

2. Programming Methodology

All programming for this project was done through the use of TI Code Composer Studio version 5.5.0. Quick software prototyping was made possible using the MSP-FET430UIF USB debugging interface that was included with the development board.

3. PWM Signal Generation

The main functionality of the software is to generate PWM signals to send to the UV LED Breakout Interface. The goal for the software was to be as modular as possible and utilize all of the PWM outputs from the Experimenter's Board. Generating PWM signals from the MSP430F5438A is a relatively straightforward task. The Experimenter's Board has a total of 7 PWM outputs, two from timer A1 and five from timer B0. To initialize PWM outputs on the experimenter board, the steps taken are as follows:

- Define output pins

```
P4DIR |= 0x7E;                  // P4 outputs (4.1, 4.2, 4.3, 4.5, 4.6)
P4SEL |= 0x7E;                  // P4 option select
```

- Define PWM period for each timer

```
TBCCR0 = 512-1;                 // PWM Period
```

- Define duty cycles (capture control register values)

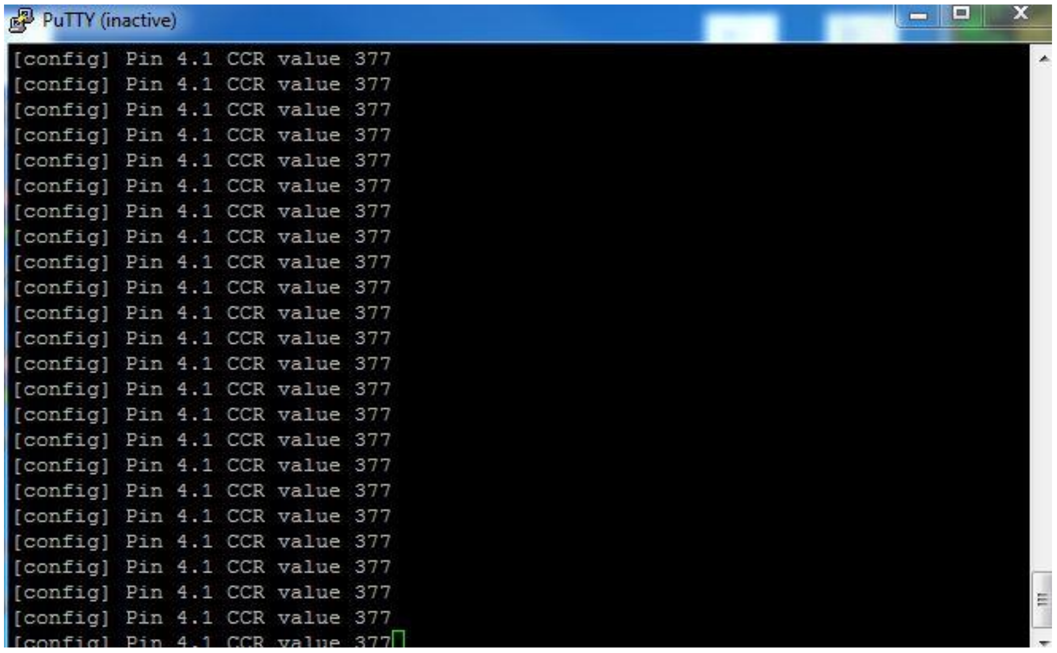
```
TBCCR1 = 383;                   // CCR1 PWM Duty Cycle (75%)
TBCCR2 = 128;                   // CCR2 PWM duty cycle (25%)
TBCCR3 = 64;                    // CCR3 PWM duty cycle (12.5%)
TBCCR5 = 32;                    // CCR5 PWM duty cycle (6.26%)
TBCCR6 = 16;                    // CCR6 PWM duty cycle (1.566%)
```

- Specify PWM operation mode

```
TBCTL = TBSSEL_1 + MC_1 + TBCLR; // ACLK, up mode, clear TBR
```

4. UART Command Line Interface

An additional functionality of the software is to communicate through UART-USB to a laptop, workstation, or any other device that supports serial communication. The idea is to enable an end-user of experimental setup, such as a marine biologist, to quickly and easily reconfigure the illumination settings during an experiment if needed. Base functionality of the command line interface reads and sends characters to and from the command line to update duty cycle using an open source driver found on GitHub.



```
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
[config] Pin 4.1 CCR value 377
```

Figure 14: Prototype Command Line Interface in PuTTY

The UI handles keyboard input as follows:

- On receiving a ‘1’ character, the program cycles through pins to configure.
- On receiving a ‘+’ or ‘-’ character, the program increases or decreases the duty cycle of the PWM signal generated at the selected pin

VI. Testing

a) Software Testing

To ensure proper software functionality, testing for PWM signals was done by measuring the duty cycle of output pins on an oscilloscope as keyboard input was read. The table below was compiled during testing.

Capture Control Register	Expected Duty Cycle (From TI Example Code)	Actual Duty Cycle
383	75%	75%
128	25%	25%
64	12.50%	12.50%
32	6.26%	6.26%
16	3.13%	3.13%
8	1.57%	1.56%

Table 2: Duty Cycle Accuracy Testing

b) Hardware Testing

To ensure proper hardware functionality, electrical connections were tested with a multimeter before and after assembly of the Breakout Interface and UV LED array. Once the experimental setup was fully assembled, qualitative testing of the hardware was done by illuminating the LEDs and making sure that each LED was functioning. Shown below, the UV LED array is illuminating all 12 LEDs at the settings specified by the software (low, medium, or high).

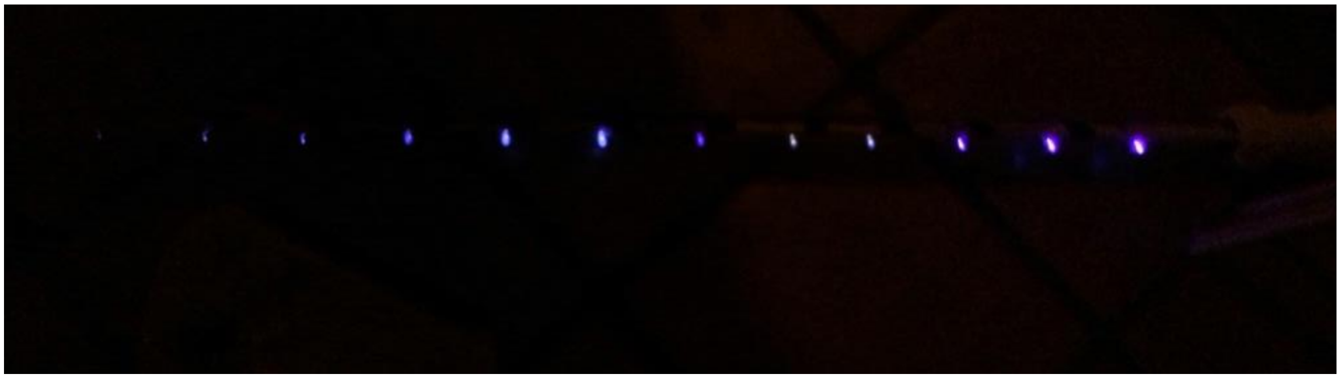


Figure 15: Testing the UV LED Array for Functionality.

(From left to right: 365nm, 390nm, 385nm, 405nm)

c) **Waterproof Testing**

Before placing any electronics into the UV LED array encasing, the PVC ends of what would be the LED array were submerged in water overnight to ensure that water would not seep in. As an added precaution, the threads of the fittings were wrapped in plumber's tape.

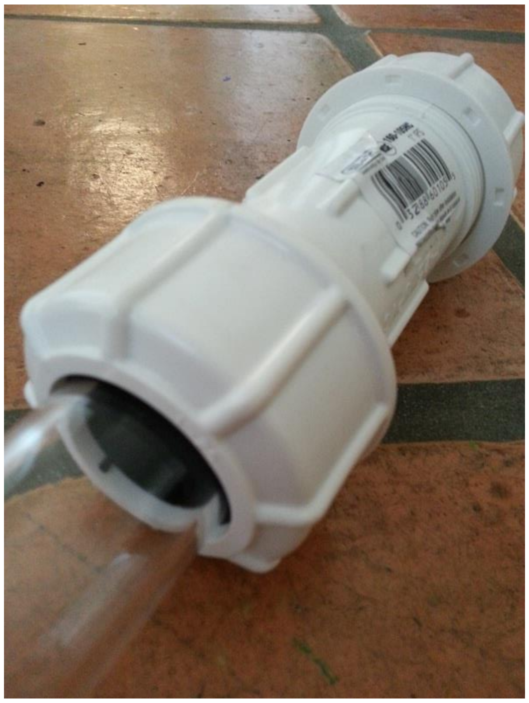


Figure 16: PVC Pressure Fitting on Glass Tube

VII. Preliminary Qualitative Experiment

a) Overview

Rather than making a large monetary investment to begin a controlled scientific experiment in a laboratory, the idea of the preliminary experiment was to get a general idea about the effect of UV LED illumination on glass, and to get an idea about how long it might take to observe biofouling for future experiments.

b) Location

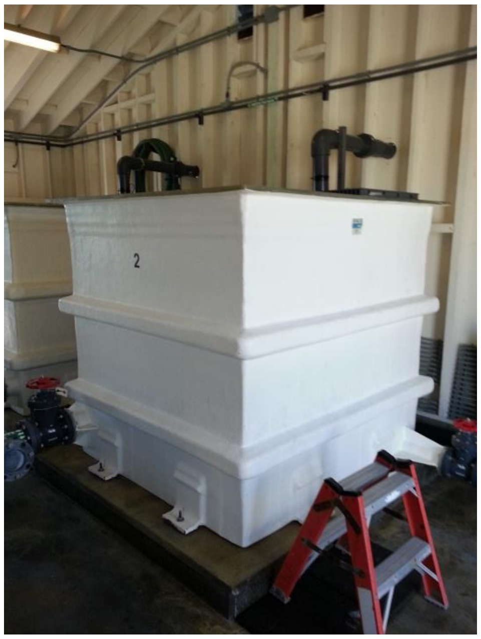


Figure 17: Raw Seawater System at Cal Poly CCMS

The qualitative experiment location was determined to be in the raw seawater system at the Center for Costal and Marine Sciences (Cal Poly Pier) in Avila Beach, California. This location was chosen because there was a large amount of biofouling in the system's main tank. The raw seawater system pumps water into the tank, which then supplies water to the other smaller tanks and student projects throughout the facility. Inside the tank was five months' worth of previous biofouling. The tank is drained and cleaned every six months, so this gave a 1-month time window for deployment.

c) **Deployment**

Shown below is the deployment of the experimental setup in the raw seawater system. The system was housed in a plastic container for additional robustness – on occasion, the workers must spray down the room with fresh water to prevent salt damage to the building. The system ran off the power from a nearby wall outlet, and the power to the LEDs was taken from a power supply the Experimenter's Board.

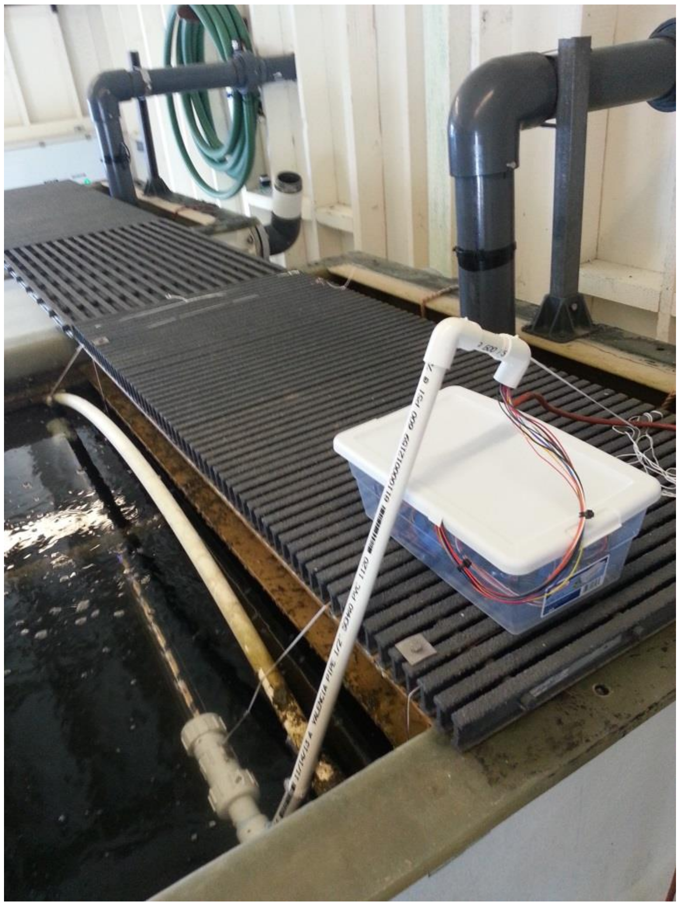


Figure 18: Setup Deployment in the Raw Seawater System

During deployment, the system was checked on 3 times a week for a period of three weeks (the system was taken out right before tank cleaning) and was photographed occasionally to observe any growth.

d) **Results – Week 1 to 3**

In each following photo, the LEDs are ordered from left to right as low (3.13% duty), medium (25% duty), and high (100%duty) illumination settings.

In the week 3 photos, there is the addition of a borosilicate glass tube without electronics in it. This tube was placed in the tank to be viewed as an experimental control. Note that growth occurred double in the tube because water was able to flow through it, as well as outside it. In the photos, the control tube is pictured at week 4 deployment.

1. **405nm, 30 degree, 3mm Package – $14.36**

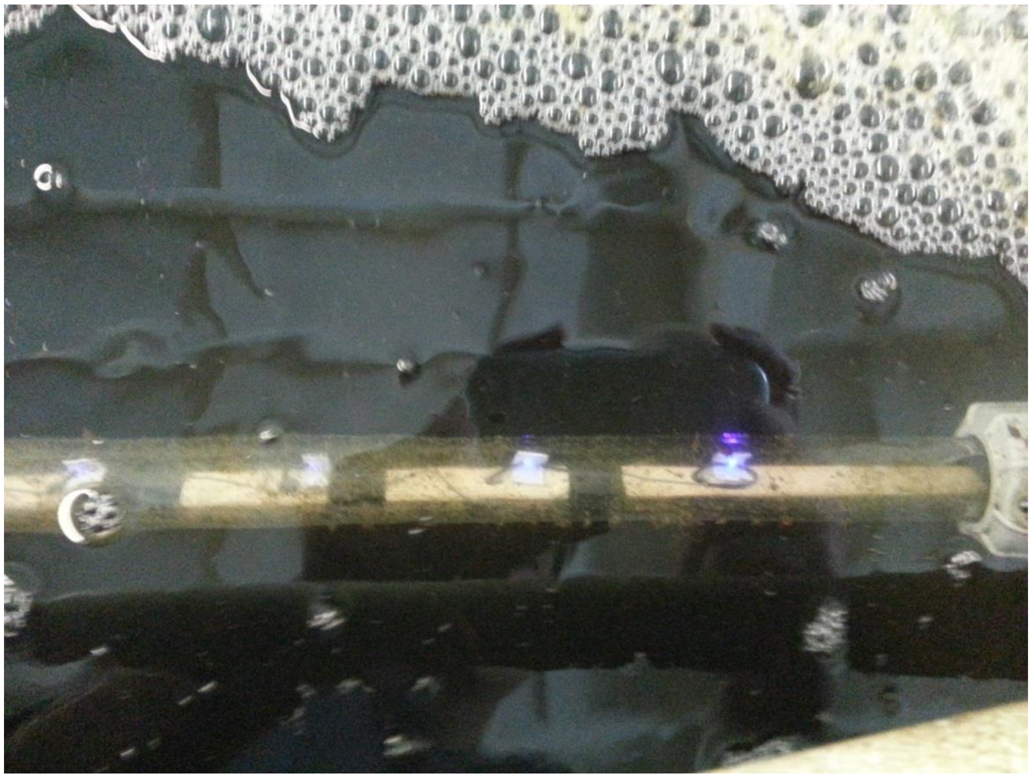


Figure 19: 405nm, Week 1

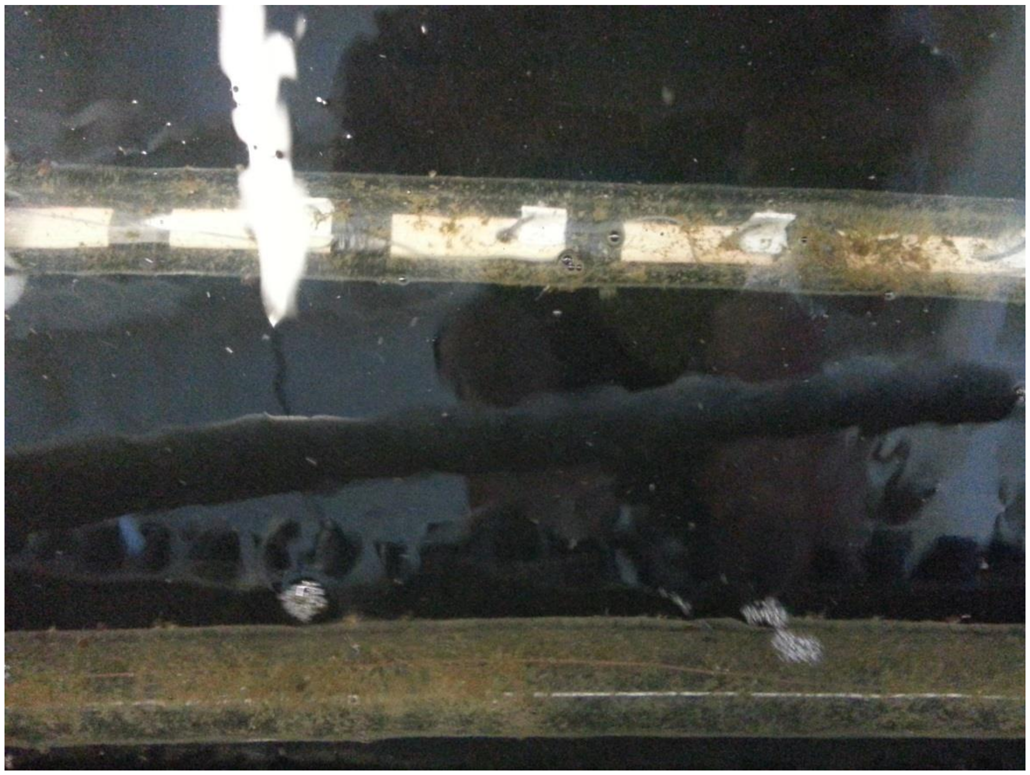


Figure 20: 405nm, Week 3

2. 390nm, 10 degree, 5mm Package - $1.59

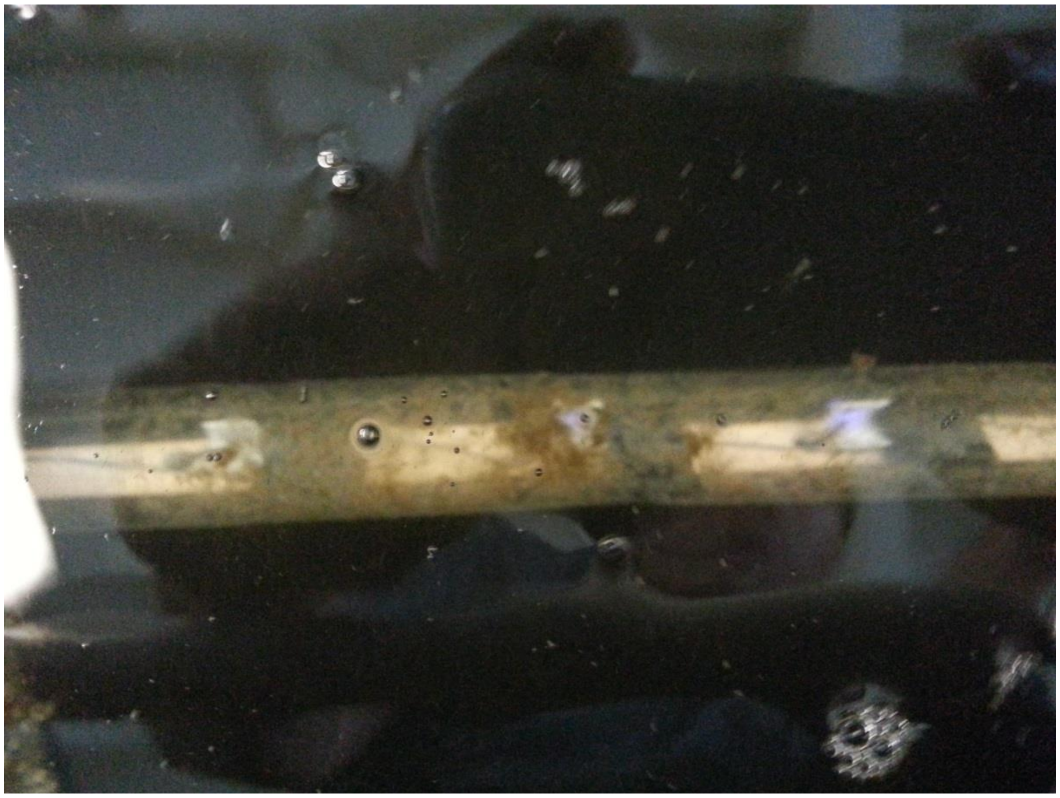


Figure 21: 390nm, Week 1

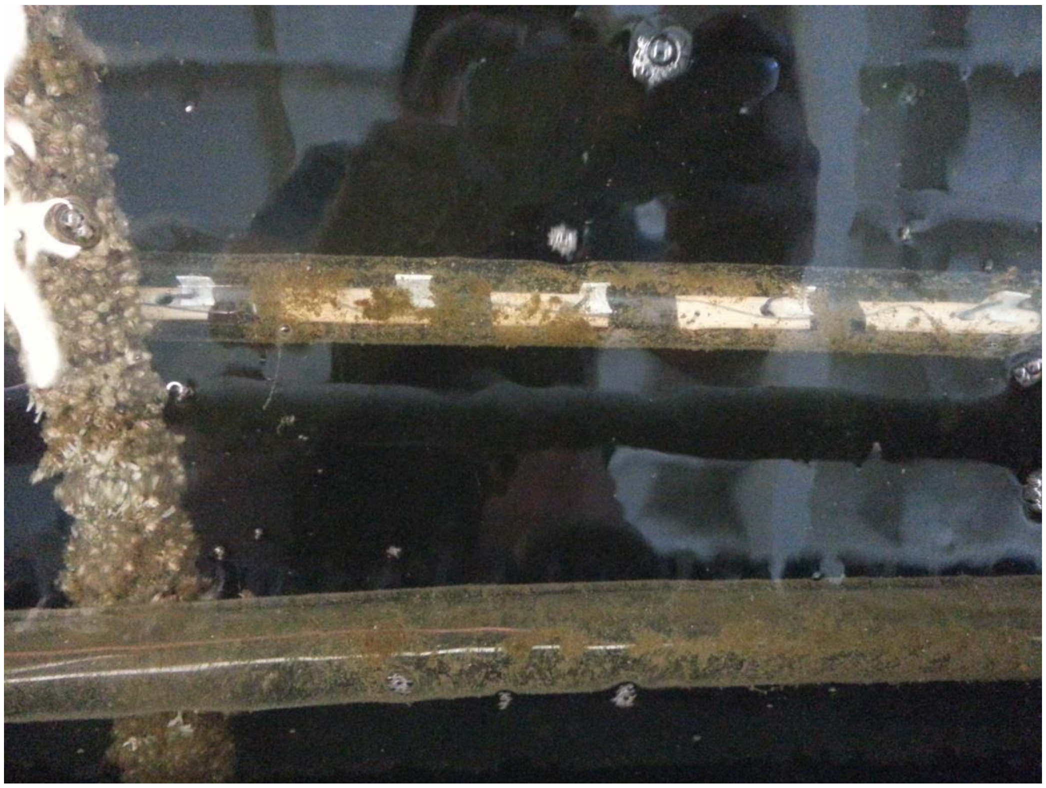


Figure 22: 390nm, Week 3

3. **385nm, 30 degree, 3mm Package - $1.00**

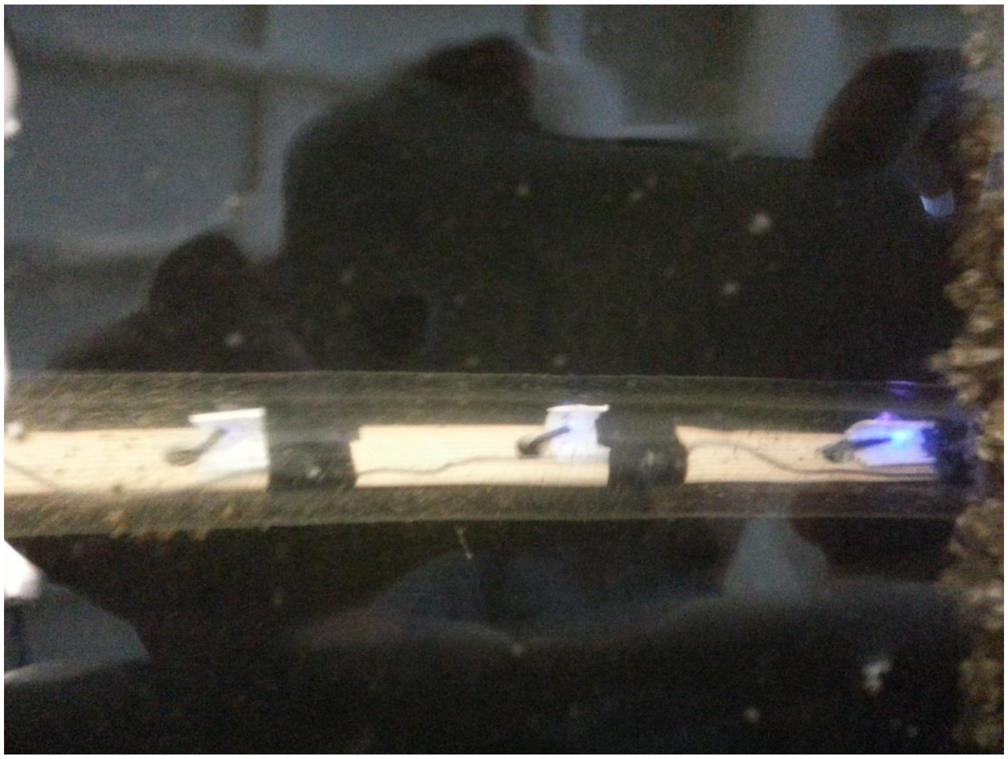


Figure 23: 385nm, Week 1

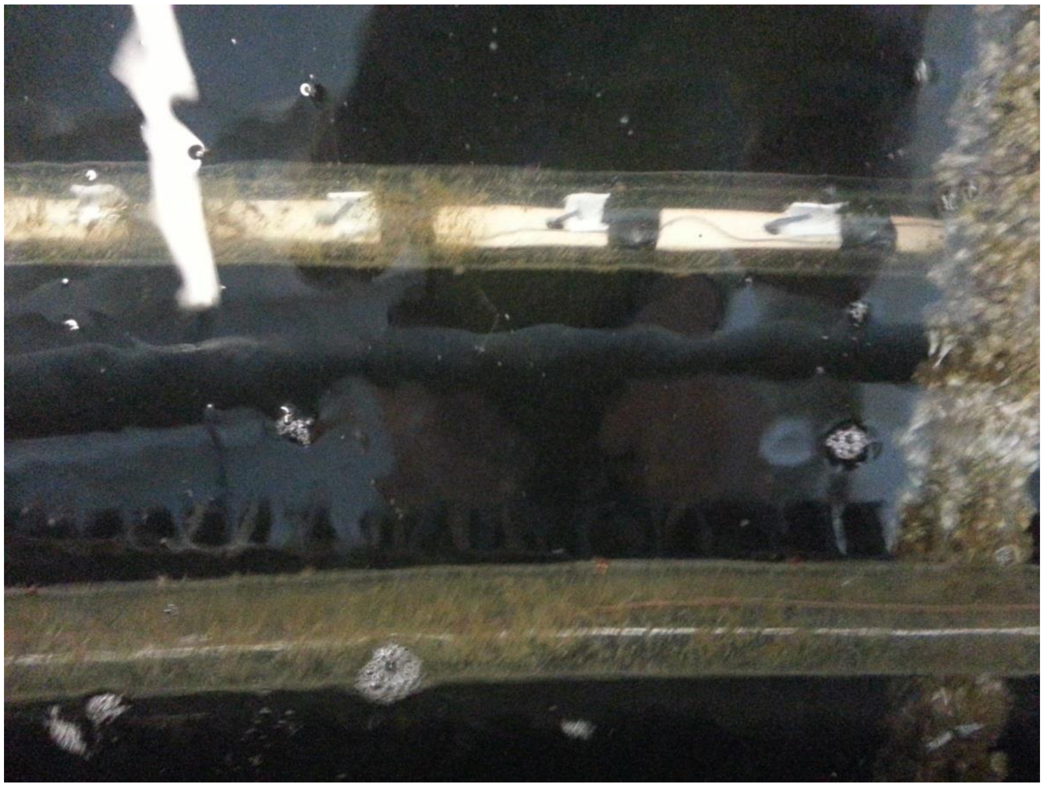


Figure 24: 385nm, Week 3

4. 365nm, 10 degree, 5mm Metal Encased Package - $13.75

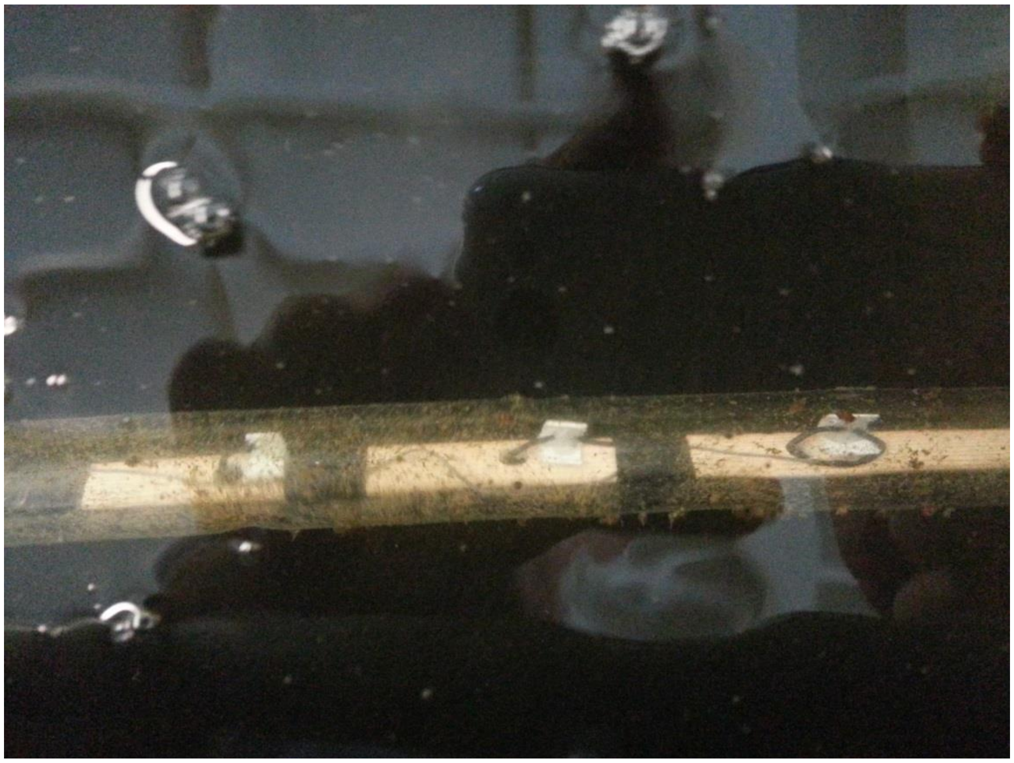


Figure 25: 365nm, Week 1

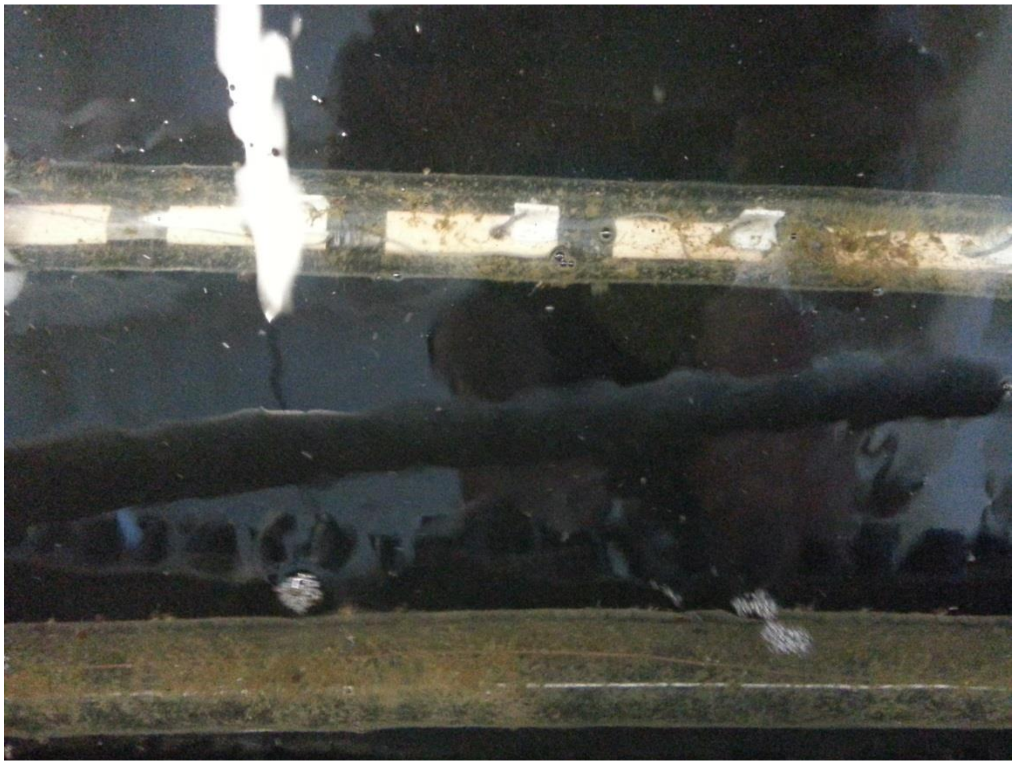


Figure 26: 365nm, Week 3

e) **Observations and Analysis**

Based on visual observations alone, it looked like the packaging of each UV LED might have played significant role in the amount of growth on the borosilicate glass tube. It is unclear if wavelength had any significant effects alone. Seen in figures 23 and 24, the 385nm, 30 degree, 5mm LED had drastically less biofouling than the 365nm, 10 degree, metal encased LED shown in figures 25 and 26. This was surprising because the 365nm UV LED was the candidate expected to mitigate the most biofouling, as its wavelength is farthest into the UV spectrum and it was the most expensive at $13.74 per unit. The most effective LED at 385nm was only $1.00 per unit! The other two LED types did not exhibit any noticeable observable results in the 3-week time period. The results suggest that 3 weeks was not enough time to conduct a biofouling experiment in sea water. Shown in figure 22, objects in the tank such as the PVC pipe (on the left of the picture) have the potential to foul significantly more over 6 months.

VIII. Conclusion / Future Work

This project was useful for learning about biofouling, UV LEDs, waterproofing, device manufacturing, and many more of the struggles that marine biologists and engineers at MBARI face on a daily basis when working with marine instrumentation. The current state of the project is finished. However, the study to determine which UV LEDs should be chosen for deployment with the SharkCafeCam is far from complete. It remains unsure if UV LEDs have a significant effect on mitigating biofouling on camera lenses. The preliminary qualitative experiment did not provide any concrete information as to which UV LEDs mitigates biofouling the best, but it gave an idea about how much time and effort would be needed in order to gain significant results. Ideally, a multi-month experimentation window that would be much more suitable for pinpointing the UV LED that should be deployed with the camera - results of this experiment cannot be rushed. Future revisions and iteration of this experiment would require more experimental control through a better planned experimental setup.

Expanding on the preliminary experimental setup, experimental control can be achieved in multiple ways. Factors such as varying LED packages and distances from the glass surfaces could be isolated. More LEDs could be tested at once for better observational data. A sensor could be used to measure UVA and UVB radiation presence inside and outside the borosilicate glass tube.

Another, more costly experimental setup would consist of a quantitative experiment dealing with barnacle larvae in a laboratory. A similar experiment was once conducted at Hong Kong University but did not involve LEDs [1]. By isolating barnacle larvae and observing and counting settlement after measured UV LED radiation doses, it is possible to quantify which LED wavelengths are most effective in stopping barnacle settlement by counting larvae. It should be noted that barnacles alone do not cause biofouling. This study would need to expand to studying the settlement of other organisms.

Bibliography

1.

Hung, Os, V. Thiyagarajan, Rss Wu, and Py Qian. "Effect of Ultraviolet Radiation on Biofilms and Subsequent Larval Settlement of Hydroides Elegans."*Marine Ecology Progress Series* 304 (2005): 155-66. Web. 22 May 2014.

2.

MSP430x5xx and MSP430x6xx Family User's Guide. Texas Instruments.
<http://www.ti.com/lit/ug/slau208n/slau208n.pdf>

Appendix

a) Analysis of Senior Project Design

1. Summary of Functional Requirements

My project illuminates up to 28 UV LEDs at various luminous intensities in order to aid a scientific study on biofouling mitigation. My preliminary experiment tested 12 LEDs at various wavelengths and intensities in raw seawater to observe biofouling.

2. Primary Constraints

The largest constraint throughout my project was time. My original plans to create a standalone device were hindered when I realized that I did not have the time to do so. This was because I spent a large part of the first quarter writing software, when I should have been more focused on planning the experiment instead. The software I wrote ended up being mostly insignificant once the experiment was finally planned out.

3. Economic

The project's estimated cost for me was originally around $25. The actual cost of my component parts was around $40 because I had to order extra components. The cost for a user to operate the device is relatively small. At most, it would take 3 hours per week to supervise its deployment. The device's bill of materials is attached in the appendix.

4. Manufacturability

The manufacturing course I took did not have a reliable way of assembling very fine-pitched surface mount components on a PCB in my given time frame. For this reason, I chose not to utilize an MSP430F5438A MCU on-board my PCB (I was able to obtain the same results with an external MSP430F5438A).

5. Sustainability

As a requirement, the device is to sustain itself for at least a week without human supervision. There are no challenges of maintaining this project. This project could potentially cause UV radiation harm to organisms in the sea, which can in turn impact ecosystems.

6. Ethical

Killing the organisms that would normally grow on the illuminated surface and altering their habitat with UV radiation may be viewed as unethical. Also, deploying instrumentation on a shark could possibly raise a debate because it would likely alter the shark's quality of life.

7. Health and Safety

Other than the previously described information about UV radiation, there is no health concern related to this project. This project is not to be used as a tanning booth or as a controller for medical sterilization.

8. Social and Political

This project is supportive of marine research and can affect the sociopolitical climate surrounding the field. Hopefully, this project will impact the way people view marine research in a positive way and give light to the difficulty of the problems engineers and scientists are working together to solve.

9. Development

During the course of this project, I learned how to use CadSoft Eagle software for PCB layout, I learned how to waterproof an electronic device, I learned how to purchase components on DigiKey, and I learned how to outsource manufacturing overseas. I also learned how to pick PVC pipe fittings from a home improvement store.

b) Bill of Materials (Breakout Interface)

Bill of Materials
Project: PWM UV LED Breakout Interface
Date: 6 June 2014

Item Number	Quantity	Reference Designator	Description	Value / Rating	Package Info	Digi-Key Part Number	Manufacturer
1	7	LEDx	Green LED	565nm, 30deg	Radial Through-Hole	754-1265-ND	Kingbright Corp.
2	7	Rx	Carbon Film Resistor	180Ω, 1/4W, ±5%	Axial Through-Hole	CF14JT180RCT-ND	Stackpole Electronics Inc.
3	7	Rx	Carbon Film Resistor	150Ω, 1/4W, ±5%	Axial Through-Hole	CF14JT150RCT-ND	Stackpole Electronics Inc.
4	21	Rx	Carbon Film Resistor	120Ω, 1/4W, ±5%	Axial Through-Hole	30 CF14JT120RCT-ND	Stackpole Electronics Inc.
5	7	Qx	ZVNL110A N-Channel MOSFET	320mA, 1.5V(th)	TO92-3 Through-Hole	ZVNL110ASCT-ND	Diodes Incorporated
6	1	S1	EG1218 Slide Switch	0.2A @ 30VDC	Through-Hole	EG1903-ND	E-Switch
7	28	TBx	Terminal Block	2 Position @ 3.5mm Pitch	Through-Hole	A98036-ND	TE Connectivity
8	1	JP1	Wire Header	9 Position @.1" spacing	Through-Hole	S7007-ND	Sullins Connector Solutions

c) Source Code – Preliminary Experiment (1 of 1 file)

```c
/*
 * main.c
 * Author: Andrew Lam
 * Description: Hard-Coded PWM Initialization for Preliminary Deployment
 */

#include <msp430.h>

   // P4.1/TB0.1|--> CCR1 - 75% PWM 383
   // P4.2/TB0.2|--> CCR2 - 25% PWM 128
   // P4.3/TB0.3|--> CCR3 - 12.5% PWM 64
   // P4.4/TB0.4|--> CCR4 - 6.26% PWM 32  (DNE on MSP430EXPF5438A)
   // P4.5/TB0.5|--> CCR5 - 3.13% PWM 16
   // P4.6/TB0.6|--> CCR6 - 1.566% PWM 8
   //Using ~1.045MHz SMCLK as TACLK, the timer period is ~500us

int main(void) {
    WDTCTL = WDTPW | WDTHOLD;// Stop watchdog timer

    //TIMER B0 PWM OUTPUTS 0b00001110 = 0x0E
    P4DIR |= 0x7E;                          // P4 outputs (4.1, 4.2, 4.3, 4.5, 4.6)
    P4SEL |= 0x7E;                          // P4 option select

    P4OUT |= BIT0;
    P4DIR |= BIT0;

    TBCCR0 = 512-1;                               // PWM Period

    TBCCR1 = 383;                                 // CCR1 PWM Duty Cycle (~75% duty)
    TBCCR2 = 128;                                 // CCR2 PWM duty cycle (~25% duty)
    TBCCR3 = 64;                                  // CCR3 PWM duty cycle (~12.5% duty)
    TBCCR5 = 32;                                  // CCR5 PWM duty cycle (~6.26% duty)
    TBCCR6 = 16;                                  // CCR6 PWM duty cycle (~1.566% duty)

    TBCCTL1 = OUTMOD_7;                           // CCR1 reset/set
    TBCCTL2 = OUTMOD_7;                           // CCR2 reset/set
    TBCCTL3 = OUTMOD_7;                           // CCR3 reset/set
    TBCCTL5 = OUTMOD_7;                           // CCR5 reset/set
    TBCCTL6 = OUTMOD_7;                           // CCR6 reset/set

    TBCTL = TBSSEL_1 + MC_1 + TBCLR;              // SMCLK, up mode, clear TBR

        return 0;
}
```

d) **My Source Code with UI, RTC, UART (1 of 3 files)**

```
/*
 * main.c
 * Author: Andrew Lam
 * Description: Initializations and main program loop.
 * Contains UI, RTC, and UART code. UART Driver is separate.
 */

#include <msp430.h>
#include "uart.h"
#include <string.h>
#include <stdio.h>

#define MAX_OUTPUT_PINS 8 //assume 4.4 works. 6 from timerB0 + 2 from timer A0
#define MAX_DUTY_CYCLE 383 //383 is about 75%
#define MIN_DUTY_CYCLE 0

void SetupRTC(void);
void SetupPWM(void);
void UpdateDuty(void);

void setSMCLK8MHz(void);
void SetupUART(void);

//variables for clock
unsigned int seconds = 0;
unsigned int minutes = 0;
unsigned int hours = 0;
unsigned int days = 0;

//For use with UART Driver
UARTConfig cnf;
USCIUARTRegs uartUsciRegs;
USARTUARTRegs uartUsartRegs;
unsigned char uartTxBuf[200];
unsigned char uartRxBuf[200];

typedef struct
{
   int duty;
 //int pulseDuration;
} PWMConfig;

PWMConfig pwmOutputs[MAX_OUTPUT_PINS];
void main(void) {

   WDTCTL = WDTPW + WDTHOLD;            // Stop WDT
   __enable_interrupt();                // Enable Global Interrupts

   //RTC OUTPUT
   P1OUT |= BIT1;                       // P1.1 is output
   P1DIR |= BIT1;                       // LED2 (for 1 Second Toggle)

   P1SEL &= ~BIT0;                      //P1.0 will toggle on UART input
   P1DIR |= BIT0;
   P1OUT |= BIT0;

   //TIMER B0 PWM OUTPUTS 0b01111110 = 0x7E
   P4DIR |= 0x7E;                       // P4 outputs
   P4SEL |= 0x7E;                       // P4 option select
   //TIMER A1 PWM OUTPUTS
   P7DIR |= 0b00001000                  //P7.3 is output (A1CCR2)
   P7SEL |= 0b00001000;
   P8DIR |= 0x01000000                  //P8.6 is output (A1CCR1)
   P8SEL |= 0b01000000;

   setSMCLK8MHz();                      // Configure SMCLK to 8MHz
   SetupUART();                         // UART: 115200 Baud
   SetupRTC();
   SetupPWM();

   unsigned int pwmIdx = 0;
   char buffer[100];
   unsigned int i;

   // Initialize PWM outputs to 75% duty
   for (i = 0; i < MAX_OUTPUT_PINS; i++)
          pwmOutputs[i].duty = MAX_DUTY_CYCLE;

  //MAIN PROGRAM LOOP
   while(1) {

          memset(buffer,0,100);
          sprintf(buffer, "%d", pwmIdx + 1);

          if (pwmIdx < 6) {
          uartSendStringBlocking(&cnf, "[config] Pin 4.");
             uartSendDataBlocking(&cnf, (unsigned char *)buffer, 1);
          }
          else if (pwmIdx == 6)
             uartSendStringBlocking(&cnf, "[config] Pin 7.3");
          else if (pwmIdx == 7)
             uartSendStringBlocking(&cnf, "[config] Pin 8.6");

          memset(buffer,0,100);
          sprintf(buffer, "%d", pwmOutputs[pwmIdx].duty);

          uartSendStringBlocking(&cnf, " CCR value ");
          uartSendDataBlocking(&cnf, (unsigned char *)buffer, 3);
       uartSendStringBlocking(&cnf, "\n\r");

          int bytesAvailable = numUartBytesReceived(&cnf);
          if(bytesAvailable > 0) {
                      unsigned char tempBuf[100];
                      memset(tempBuf,0,100);
                      volatile int bytesRead =
                       readRxBytes(&cnf, tempBuf, bytesAvailable, 0);

                      if(bytesRead == bytesAvailable) {
                             if(tempBuf[0] == '1') {
                                    P1OUT ^= BIT0;
                                    pwmIdx++;
                    if (pwmIdx == MAX_OUTPUT_PINS)
                      pwmIdx = 0;

                    if (pwmIdx == 3)//since 4.4 doesnt exist
                      pwmIdx = 4;
                             }
                             if(tempBuf[0] == '-') {
                                    P1OUT ^= BIT0;
```

```
                                        pwmOutputs[pwmIdx].duty--;
                                        if (pwmOutputs[pwmIdx].duty < MIN_DUTY_CYCLE)
                                           pwmOutputs[pwmIdx].duty = MIN_DUTY_CYCLE;
                                        UpdateDuty();
                        }
                                if(tempBuf[0] == '+') {
                                        P1OUT ^= BIT0;

                                        pwmOutputs[pwmIdx].duty++;
                                        if (pwmOutputs[pwmIdx].duty > MAX_DUTY_CYCLE)
                                           pwmOutputs[pwmIdx].duty = MAX_DUTY_CYCLE;

                                        UpdateDuty();
                                }
                        }
                        else
                        {
                                __no_operation();
                        }
                }
        }
}

//Initializes UART driver.
void SetupUART(void) {

                initUartDriver();
                // Configure UART Module on USCIA1
                cnf.moduleName = USCI_A1;
                // Use UART Pins P5.7 and P5.6
                cnf.portNum = PORT_5;
                cnf.RxPinNum = PIN7;
                cnf.TxPinNum = PIN6;
                // 115200 Baud from 8MHz SMCLK
                cnf.clkRate = 8000000L;
                cnf.baudRate = 115200L;
                cnf.clkSrc = UART_CLK_SRC_SMCLK;
                // 8N1
                cnf.databits = 8;
                cnf.parity = UART_PARITY_NONE;
                cnf.stopbits = 1;
                int res = configUSCIUart(&cnf,&uartUsciRegs);
                if(res != UART_SUCCESS) {
                        // Failed to initialize UART for some reason
                        __no_operation();
                }
                // Configure the buffers that will be used by the UART Driver.
                // These buffers are exclusively for the UART driver's use and
                // should not be touched by the application itself. Note
                //that they may affect performance if they're too small.
                setUartTxBuffer(&cnf, uartTxBuf, 200);
                setUartRxBuffer(&cnf, uartRxBuf, 200);
                enableUartRx(&cnf);
}

//Initializes the RTC to ~1 second
void SetupRTC(void) {

        RTCCTL01 = RTCTEVIE + RTCSSEL_2 + RTCTEV_0; // Counter Mode, RTC1PS, 8-bit ovf
                                                    // overflow interrupt enable
        RTCPS0CTL = RT0PSDIV_2;                     // ACLK, /8, start timer
        RTCPS1CTL = RT1SSEL_2 + RT1PSDIV_3;         // out from RT0PS, /16, start timer
}

//Initializes PWMs
void SetupPWM(void) {

   TBCCR0 = 512-1;                            // PWM Period
   TBCCTL1 = OUTMOD_7;                        // CCR1 reset/set
   TBCCTL2 = OUTMOD_7;                        // CCR2 reset/set
   TBCCTL3 = OUTMOD_7;                        // CCR3 reset/set
 //TBCCTL4 = OUTMOD_7;                        // CCR4 reset/set
   TBCCTL5 = OUTMOD_7;                        // CCR5 reset/set
   TBCCTL6 = OUTMOD_7;                        // CCR6 reset/set
   TBCTL = TBSSEL_1 + MC_1 + TBCLR;           // ACLK, up mode, clear TBR
   UpdateDuty();
}


//Updates duty cycles for all PWM outputs
void UpdateDuty(void) {
          TBCCR0 = 512-1;                            // PWM Period
          TBCCR1 = pwmOutputs[0].duty;               // CCR1 PWM Duty Cycle
          TBCCR2 = pwmOutputs[1].duty;               // CCR2 PWM duty cycle
          TBCCR3 = pwmOutputs[2].duty;               // CCR3 PWM duty cycle
        //TBCCR4 = pwmOutputs[3].duty;               // CCR4 PWM duty cycle
          TBCCR5 = pwmOutputs[4].duty;               // CCR5 PWM duty cycle
          TBCCR6 = pwmOutputs[5].duty;               // CCR6 PWM duty cycle

          //TACCR1 = pwmOutputs[6].duty;               // CCR1 PWM duty cycle
          //TACCR2 = pwmOutputs[7].duty;               // CCR2 PWM duty cycle
}

//Sets SMCLK to 8MHz
void setSMCLK8MHz()
{
        // Set clock to 8MHz
        UCSCTL3 |= SELREF_2;                      // Set DCO FLL reference = REFO
        UCSCTL4 |= SELA_2;                        // Set ACLK = REFO
        __bis_SR_register(SCG0);                  // Disable the FLL control loop
        UCSCTL0 = 0x0000;                         // Set lowest possible DCOx, MODx
        UCSCTL1 = DCORSEL_5;                      // Select DCO range 16MHz operation
        UCSCTL2 = FLLD_1 + 244;                   // Set DCO Multiplier for 8MHz
                                                                  // (N + 1) *
FLLRef = Fdco
                                                                  // (244 + 1)
* 32768 = 8MHz
                                                                  // Set FLL
Div = fDCOCLK/2
        __bic_SR_register(SCG0);                  // Enable the FLL control loop
        // Worst-case settling time for the DCO when the DCO range bits have been
        // changed is n x 32 x 32 x f_MCLK / f_FLL_reference. See UCS chapter in 5xx
        // UG for optimization.
        // 32 x 32 x 8 MHz / 32,768 Hz = 250000 = MCLK cycles for DCO to settle
        __delay_cycles(250000);

        do { // Loop until XT1,XT2 & DCO fault flag is cleared
        UCSCTL7 &= ~(XT2OFFG + XT1LFOFFG + XT1HFOFFG + DCOFFG);
                                                                // Clear
XT2,XT1,DCO fault flags
        SFRIFG1 &= ~OFIFG;                      // Clear fault flags
        } while (SFRIFG1&OFIFG);                 // Test oscillator fault flag
```

```
}


//ISR for RTC Interrupts. Counter not yet implemented.
#pragma vector=RTC_VECTOR
__interrupt void RTC_ISR(void)
{
   switch(__even_in_range(RTCIV,16))
   {
     case 0: break;                              // No interrupts
     case 2: break;                              // RTCRDYIFG
     case 4:                                     // RTCTEVIFG
        P1OUT ^= BIT1;
        seconds++;
        /*
        if seconds = 60
               minutes++;
               seconds = 0;
        if minutes = 60
               hours++;
               minutes = 0;
        if hours = 24
               days++;
               hours = 0;
        */
        break;
     case 6: break;                              // RTCAIFG
     case 8: break;                              // RT0PSIFG
     case 10: break;                             // RT1PSIFG
     case 12: break;                             // Reserved
     case 14: break;                             // Reserved
     case 16: break;                             // Reserved
     default: break;
   }
}
```

e) Source Code –Open Source UART Driver by Gustavo Litovsky (2 of 3 Files)

```
/*
* Copyright (c) 2012 All Right Reserved, Gustavo Litovsky
*
* You may use this file for any purpose, provided this copyright notice and
* the attribution remains.
*
* THIS CODE AND INFORMATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY
* KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
* PARTICULAR PURPOSE.
*
* Author: Gustavo Litovsky
* gustavo@glitovsky.com
*
* File:    uart.h
* Summary: Definitions for MSP430 UART Driver
*
*/

#ifndef UART_H_
#define UART_H_

#ifndef NULL
#define NULL 0
#endif

#define PORT_1 1
#define PORT_2 2
#define PORT_3 3
#define PORT_4 4
#define PORT_5 5
#define PORT_6 6
#define PORT_7 7
#define PORT_8 8
#define PORT_9 9

#define PIN0 0
#define PIN1 1
#define PIN2 2
#define PIN3 3
#define PIN4 4
#define PIN5 5
#define PIN6 6
#define PIN7 7

enum UART_ERR_CODES
{
       UART_SUCCESS = 0,
       UART_BAD_MODULE_NAME,
       UART_BAD_CLK_SOURCE,
       UART_INSUFFICIENT_TX_BUF,
       UART_INSUFFICIENT_RX_BUF,
       UART_BAD_PORT_SELECTED
};

typedef enum
{
       USCI_A0,  /**< USCI_A0 Module  */
       USCI_A1,  /**< USCI_A1 Module  */
```

```
	USCI_A2,  /**< USCI_A2 Module  */
	USART_0,  /**< USART_0 Module  */
	USART_1  /**< USART_1 Module  */

}UART_MODULE_NAMES;

typedef enum
{
	UART_CLK_SRC_ACLK,  /**< ACLK as source for UART baud rate generator  */
	UART_CLK_SRC_SMCLK  /**< SMCLK as source for UART baud rate generator  */

}UART_CLK_SRCS;

typedef enum
{
	UART_PARITY_NONE,  /**< No Parity  */
	UART_PARITY_EVEN,  /**< Even Parity  */
	UART_PARITY_ODD    /**< Odd Parity  */

}UART_PARITY;

/** @struct USCIUARTRegs
 *  @brief This structure contains pointers to the relevant
 *            registers necessary to configure and use a USCI UART module.
 *
 */
typedef struct
{
	unsigned char * CTL0_REG;
	unsigned char * CTL1_REG;
	unsigned char * MCTL_REG;
	unsigned char * BR0_REG;
	unsigned char * BR1_REG;
	unsigned char * IE_REG;
	unsigned char * RX_BUF;
	unsigned char * TX_BUF;
	unsigned char * IFG_REG;
} USCIUARTRegs;

/** @struct USARTUARTRegs
 *  @brief This structure contains pointers to the relevant
 *            registers necessary to configure and use a USART UART module.
 *
 */
typedef struct
{
	unsigned char * ME_REG;
	unsigned char * U0CTL_REG;
	unsigned char * UTCLT0_REG;
	unsigned char * UBR0_REG;
	unsigned char * UBR1_REG;
	unsigned char * UMCTL_REG;
	unsigned char * IE_REG;
	unsigned char * RX_BUF;
	unsigned char * TX_BUF;
	unsigned char * IFG_REG;
	unsigned char TXIFGFlag;
	unsigned char RXIFGFlag;
	unsigned char TXIE;
	unsigned char RXIE;
} USARTUARTRegs;
```

```c
/** @struct UARTConfig
 *  @brief This struct contains all the configuration needed for
 *        UART operation.
 *
 */
typedef struct
{
       UART_MODULE_NAMES moduleName; /**< Module Name that specifies which module to use  */
       char portNum;                 /**< GPIO Port Number  */
       char TxPinNum;                /**< GPIO TX Pin Number  */
       char RxPinNum;                /**< GPIO RX Pin Number  */
       unsigned long clkRate;        /**< Clock rate of the clock used as source for module
*/
       unsigned long baudRate;       /**< UART baud rate desired  */
       UART_CLK_SRCS clkSrc;         /**< Clock source used for UART module  */
       char databits;                /**< Number of data bits used for communications  */
       char stopbits;                /**< Number of stop bits used for communications  */
       UART_PARITY parity;           /**< Parity used for communications  */
       USCIUARTRegs * usciRegs;
       USARTUARTRegs * usartRegs;
       unsigned char * txBuf;
       unsigned char * rxBuf;
       int txBufLen;
       int rxBufLen;
       int rxBytesReceived;
       int txBytesToSend;
       int txBufCtr;
} UARTConfig;


/* Function Declarations */
int configUSCIUart(UARTConfig * prtInf,USCIUARTRegs * confRegs);
int configUSARTUart(UARTConfig * prtInf, USARTUARTRegs * confRegs);
int uartSendDataBlocking(UARTConfig * prtInf,unsigned char * buf, int len);
int uartSendStringBlocking(UARTConfig * prtInf,char * string);
int initUSCIUart(USCIUARTRegs * confRegs, UARTConfig * prtInf);
int initUartPort(UARTConfig * prtInf);
void initBufferDefaults(UARTConfig * prtInf);
void setUartTxBuffer(UARTConfig * prtInf, unsigned char * buf, int bufLen);
void setUartRxBuffer(UARTConfig * prtInf, unsigned char * buf, int bufLen);
void initUartDriver();
int uartSendDataInt(UARTConfig * prtInf,unsigned char * buf, int len);
void enableUartRx(UARTConfig * prtInf);
int numUartBytesReceived(UARTConfig * prtInf);
unsigned char * getUartRxBufferData(UARTConfig * prtInf);
int readRxBytes(UARTConfig * prtInf, unsigned char * data, int numBytesToRead, int offset);

#endif /* UART_H_ */
```

f) Source Code –Open Source UART Driver by Gustavo Litovsky (3 of 3 Files)

```
/*
* Copyright (c) 2012 All Right Reserved, Gustavo Litovsky
*
* You may use this file for any purpose, provided this copyright notice and
* the attribution remains.
*
* THIS CODE AND INFORMATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY
* KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A
* PARTICULAR PURPOSE.
*
* Author: Gustavo Litovsky
* gustavo@glitovsky.com
*
* File:    uart.c
* Summary: Implementation of generic MSP430 UART Driver
*
*/

#include <msp430.h>
#include <math.h>
#include <string.h>
#include "uart.h"

// Port Information List so user isn't forced to pass information all the time
UARTConfig * prtInfList[5];

/*!
 * \brief Initializes the UART Driver
 *
 *
 * @param None
 * \return None
 *
 */
void initUartDriver()
{
       int i = 0;
       for(i = 0; i < 5; i++)
       {
              prtInfList[i] = NULL;
       }
}

/*!
 * \brief Configures the MSP430 pins for UART module
 *
 *
 * @param prtInf is UARTConfig instance with the configuration settings
 * \return Success or errors as defined by UART_ERR_CODES
 *
 */
int initUartPort(UARTConfig * prtInf)
{
       unsigned char * prtSelReg = NULL;
       switch(prtInf->portNum)
       {
#ifdef __MSP430_HAS_PORT1_R__
```

```
		case 1:
			prtSelReg = (unsigned char *)&P1SEL;
			break;
#endif
#ifdef __MSP430_HAS_PORT2_R__
		case 2:
			prtSelReg = (unsigned char *)&P2SEL;
			break;
#endif
#ifdef __MSP430_HAS_PORT3_R__
		case 3:
			prtSelReg = (unsigned char *)&P3SEL;
			break;
#endif
#ifdef __MSP430_HAS_PORT4_R__
		case 4:
			prtSelReg = (unsigned char *)&P4SEL;
			break;
#endif
#ifdef __MSP430_HAS_PORT5_R__
		case 5:
			prtSelReg = (unsigned char *)&P5SEL;
			break;
#endif
#ifdef __MSP430_HAS_PORT6_R__
		case 6:
			prtSelReg = (unsigned char *)&P6SEL;
			break;
#endif
#ifdef __MSP430_HAS_PORT7_R__
		case 7:
			prtSelReg = (unsigned char *)&P7SEL;
			break;
#endif
#ifdef __MSP430_HAS_PORT8_R__
		case 8:
			prtSelReg = (unsigned char *)&P8SEL;
			break;
#endif
#ifdef __MSP430_HAS_PORT9_R__
		case 9:
			prtSelReg = (unsigned char *)&P9SEL;
			break;
#endif
		default:
			prtSelReg = NULL;
			break;
	}

	if(prtSelReg == NULL)
	{
		return UART_BAD_PORT_SELECTED;
	}

	// Configure Port to use UART Module by setting the corresponding bits on the PxSEL
	// register.
	*prtSelReg |=  (BIT0 << prtInf->RxPinNum) | (BIT0 << prtInf->TxPinNum);

	return UART_SUCCESS;
}
```

```
/*!
 * \brief Configures the UART Pins and Module for communications
 *
 * This function accepts a UARTConfig instance and initializes
 * the UART module appropriately. See UARTConfig for more info.
 *
 *
 * @param prtInf is UARTConfig instance with the configuration settings
 * @param confRegs is a pointer to a struct holding the configuration register
 * \return Success or errors as defined by UART_ERR_CODES
 *
 */
#if defined(__MSP430_HAS_USCI__) || defined(__MSP430_HAS_USCI_A0__) ||
defined(__MSP430_HAS_USCI_A1__) || defined(__MSP430_HAS_USCI_A2__)
int configUSCIUart(UARTConfig * prtInf,USCIUARTRegs * confRegs)
{
	initUartPort(prtInf);

	// Configure the pointers to the right registers
	switch(prtInf->moduleName)
	{
#if defined(__MSP430_HAS_USCI_A0__) || defined(__MSP430_HAS_USCI__)
		case USCI_A0:
			confRegs->CTL0_REG = (unsigned char *)&UCA0CTL0;
			confRegs->CTL1_REG = (unsigned char *)&UCA0CTL1;
			confRegs->MCTL_REG = (unsigned char *)&UCA0MCTL;
			confRegs->BR0_REG  = (unsigned char *)&UCA0BR0;
			confRegs->BR1_REG  = (unsigned char *)&UCA0BR1;
			confRegs->IE_REG  = (unsigned char *)&UCA0IE;
			confRegs->RX_BUF = (unsigned char *)&UCA0RXBUF;
			confRegs->TX_BUF = (unsigned char *)&UCA0TXBUF;
			confRegs->IFG_REG = (unsigned char *)&UCA0IFG;
			break;
#endif
#ifdef __MSP430_HAS_USCI_A1__
		case USCI_A1:
			confRegs->CTL0_REG = (unsigned char *)&UCA1CTL0;
			confRegs->CTL1_REG = (unsigned char *)&UCA1CTL1;
			confRegs->MCTL_REG = (unsigned char *)&UCA1MCTL;
			confRegs->BR0_REG  = (unsigned char *)&UCA1BR0;
			confRegs->BR1_REG  = (unsigned char *)&UCA1BR1;
			confRegs->IE_REG  =  (unsigned char *)&UCA1IE;
			confRegs->RX_BUF =   (unsigned char *)&UCA1RXBUF;
			confRegs->TX_BUF =   (unsigned char *)&UCA1TXBUF;
			confRegs->IFG_REG =  (unsigned char *)&UCA1IFG;
			break;
#endif
#ifdef __MSP430_HAS_USCI_A2__
		case USCI_A2:
			confRegs->CTL0_REG = (unsigned char *)&UCA2CTL0;
			confRegs->CTL1_REG = (unsigned char *)&UCA2CTL1;
			confRegs->MCTL_REG = (unsigned char *)&UCA2MCTL;
			confRegs->BR0_REG  = (unsigned char *)&UCA2BR0;
			confRegs->BR1_REG  = (unsigned char *)&UCA2BR1;
			confRegs->IE_REG  = (unsigned char *)&UCA2IE;
			confRegs->RX_BUF = (unsigned char *)&UCA2RXBUF;
			confRegs->TX_BUF = (unsigned char *)&UCA2TXBUF;
			confRegs->IFG_REG = (unsigned char *)&UCA2IFG;
			break;
#endif
```

```
        }

        // Place Module in reset to allow us to modify its bits
        *confRegs->CTL1_REG |= UCSWRST;

        // Configure UART Settings
        if(prtInf->parity == UART_PARITY_EVEN)
        {
                *confRegs->CTL0_REG |= UCPEN | UCPAR;
        }
        else if(prtInf->parity == UART_PARITY_ODD)
        {
                *confRegs->CTL0_REG |= UCPEN;
        }

        if(prtInf->databits == 7)
        {
                *confRegs->CTL0_REG |= UC7BIT;
        }

        if(prtInf->stopbits == 2)
        {
                *confRegs->CTL0_REG |= UCSPB;
        }

        // Configure clock source

        // Clear clock source bits, then set the proper ones;
        *confRegs->CTL1_REG &= ~(UCSSEL1 | UCSSEL0);

        switch(prtInf->clkSrc)
        {
                case UART_CLK_SRC_ACLK:
                        *confRegs->CTL1_REG |= (UCSSEL0);
                        break;
                case UART_CLK_SRC_SMCLK:
                        *confRegs->CTL1_REG |= (UCSSEL1);
                        break;
        }

        // Set the baudrate dividers and modulation
        unsigned int N_div;
        N_div = prtInf->clkRate / prtInf->baudRate;

        float N_div_f;
        N_div_f = (float)prtInf->clkRate / (float)prtInf->baudRate;

        if(N_div >= 16)
        {
                // We can use Oversampling mode
                N_div /= 16;
                *confRegs->BR0_REG = (N_div & 0x00FF);
                *confRegs->BR1_REG = ((N_div & 0xFF00) >> 8);

                N_div_f /= 16.0;
                *confRegs->MCTL_REG = (unsigned char)(((N_div_f) - round(N_div_f))*16.0f) << 4;
// Set BRF
                *confRegs->MCTL_REG |= UCOS16; // Enable Oversampling Mode
        }
        else
        {
                // We must use the Low Frequency mode
                *confRegs->BR0_REG = (N_div & 0x00FF);
                *confRegs->BR1_REG = ((N_div & 0xFF00) >> 8);

                *confRegs->MCTL_REG = (unsigned char)((N_div_f - round(N_div_f))*8.0f) << 1; //
Set BRS
        }

        // Take Module out of reset
        *confRegs->CTL1_REG &= ~UCSWRST;

        // Sets the pointer to the register configuration so we don't
        // have to keep passing it around and can pass only the UART configuration
        prtInf->usciRegs = confRegs;

        initBufferDefaults(prtInf);

        // Assign pointer to port information to the array so it can be accessed later
        prtInfList[prtInf->moduleName] = prtInf;

        return UART_SUCCESS;
}
#endif

/*!
 * \brief Initializes RX and TX buffers pointers
 *
 *
 * @param None.
 * \return None.
 *
 */
void initBufferDefaults(UARTConfig * prtInf)
{
        prtInf->txBuf = NULL;
        prtInf->txBufLen = 0;

        prtInf->rxBuf = NULL;
        prtInf->rxBufLen = 0;

        prtInf->rxBytesReceived = 0;
        prtInf->txBytesToSend = 0;
        prtInf->txBufCtr = 0;
}

/*!
 * \brief Configures the UART Pins and Module for communications
 *
 * This function accepts a UARTConfig instance and initializes
 * the UART module appropriately. See UARTConfig for more info.
 *
 *
 * @param prtInf is UARTConfig instance with the configuration settings
 *
 * \return Success or errors as defined by UART_ERR_CODES
 *
 */
#if defined(__MSP430_HAS_UART0__) || defined(__MSP430_HAS_UART1__)
int configUSARTUart(UARTConfig * prtInf, USARTUARTRegs * confRegs)
{
        initUartPort(prtInf);
        // Configure the pointers to the right registers
        switch(prtInf->moduleName)
        {
#ifdef __MSP430_HAS_UART0__
                case USART_0:
                        *confRegs->ME_REG =     ME1;
                        *confRegs->U0CTL_REG =  UCTL0;
                        *confRegs->UTCLT0_REG = UTCTL0;
                        *confRegs->UBR0_REG =   UBR00;
                        *confRegs->UBR1_REG =   UBR10;
                        *confRegs->UMCTL_REG = UMCTL0;
                        *confRegs->IE_REG =     IE1;
                        *confRegs->RX_BUF =     RXBUF0;
                        *confRegs->TX_BUF =     TXBUF0;
                        *confRegs->IFG_REG =    IFG1 ;
                        confRegs->TXIFGFlag =  UTXIFG0;
                        confRegs->RXIFGFlag =  URXIFG0;
                        confRegs->TXIE = UTXIE0;
                        confRegs->RXIE = URXIE0;
                        break;
#endif

#ifdef __MSP430_HAS_UART1__
                case USART_1:
                        *confRegs->ME_REG =     ME2;
                        *confRegs->U0CTL_REG =  UCTL1;
                        *confRegs->UTCLT0_REG = UTCTL1;
                        *confRegs->UBR0_REG =   UBR01;
                        *confRegs->UBR1_REG =   UBR11;
                        *confRegs->UMCTL_REG = UMCTL1;
                        *confRegs->IE_REG =     IE2;
                        *confRegs->RX_BUF =     RXBUF1;
                        *confRegs->TX_BUF =     TXBUF1;
                        confRegs->TXIFGFlag =  UTXIFG1;
                        confRegs->RXIFGFlag =  URXIFG1;
                        confRegs->TXIE = UTXIE1;
                        confRegs->RXIE = URXIE1;

                        break;
#endif
        }

        // Place Module in reset to allow us to modify its bits
        *confRegs->U0CTL_REG |= SWRST;

        // Configure UART Settings
        if(prtInf->parity == UART_PARITY_EVEN)
        {
                *confRegs->U0CTL_REG |= PENA;
                *confRegs->U0CTL_REG |= PEV;
        }
        else if(prtInf->parity == UART_PARITY_ODD)
        {
                *confRegs->U0CTL_REG |= PENA;
                *confRegs->U0CTL_REG &= ~PEV;
        }
        else if(prtInf->parity == UART_PARITY_NONE)
        {
                *confRegs->U0CTL_REG &= ~PENA;
                *confRegs->U0CTL_REG &= ~PEV;
        }

        if(prtInf->databits == 7)
        {
                *confRegs->U0CTL_REG &= ~CHAR;
        }
        else if(prtInf->databits == 8)
        {
                *confRegs->U0CTL_REG |= CHAR;
        }

        if(prtInf->stopbits == 1)
        {
                *confRegs->U0CTL_REG &= ~SPB;
        }
        else if(prtInf->stopbits == 2)
        {
                *confRegs->U0CTL_REG |= SPB;
        }

        // Configure clock source

        // Clear clock source bits, then set the proper ones;
        *confRegs->UTCLT0_REG &= ~(SSEL1 | SSEL0);

        switch(prtInf->clkSrc)
        {
                case UART_CLK_SRC_ACLK:
                        *confRegs->UTCLT0_REG |= (SSEL0);
                        break;
                case UART_CLK_SRC_SMCLK:
                        *confRegs->UTCLT0_REG |= (SSEL1);
                        break;
        }

        // Set the baudrate dividers and modulation
        unsigned int N_div;
        N_div = prtInf->clkRate / prtInf->baudRate;


        *confRegs->UBR0_REG = (N_div & 0x00FF);
        *confRegs->UBR1_REG = ((N_div & 0xFF00) >> 8);

        // Modulation currently set to 0. Needs proper handling
        *confRegs->UMCTL_REG = 0;

        // Take Module out of reset
        *confRegs->U0CTL_REG &= ~SWRST;


        prtInf->usartRegs = confRegs;

        initBufferDefaults(prtInf);

        // Assign pointer to port information to the array so it can be accessed later
        prtInfList[prtInf->moduleName] = prtInf;

        return UART_SUCCESS;
}

#endif
```

```
/*!
 * \brief Returns the number of bytes in the RX Buffer
 *
 *
 * @param prtInf is a pointer to the UART configuration
 *
 * \return Number of bytes in RX Buffer
 *
 */
int numUartBytesReceived(UARTConfig * prtInf)
{
       return prtInf->rxBytesReceived;
}

/*!
 * \brief Returns a pointer to the RX Buffer
 *
 *
 * @param prtInf is a pointer to the UART configuration
 *
 * \return pointer to the RX buffer of the UART
 *
 */
unsigned char * getUartRxBufferData(UARTConfig * prtInf)
{
       return prtInf->rxBuf;
}


/*!
 * \brief Sends len number of bytes from the buffer using the specified
 * UART, but does so by blocking.
 *
 * This function is blocking, although interrupts may trigger during its
 * execution.
 *
 * @param prtInf is a pointer to the UART configuration
 * @param buf is a pointer to the buffer containing the bytes to be sent.
 * @param len is an integer containing the number of bytes to send.
 *
 * \return Success or errors as defined by UART_ERR_CODES
 *
 */
int uartSendDataBlocking(UARTConfig * prtInf,unsigned char * buf, int len)
{
       int i = 0;
       for(i = 0; i < len; i++)
       {
              if(prtInf->moduleName == USCI_A0|| prtInf->moduleName == USCI_A1 || prtInf-
>moduleName == USCI_A2)
              {
#if defined(__MSP430_HAS_USCI__) || defined(__MSP430_HAS_USCI_A0__) ||
defined(__MSP430_HAS_USCI_A1__) || defined(__MSP430_HAS_USCI_A2__)
                     while(!( *prtInf->usciRegs->IFG_REG & UCTXIFG));
                     *prtInf->usciRegs->TX_BUF = buf[i];
#endif
              }

              if(prtInf->moduleName == USART_0|| prtInf->moduleName == USART_1)
              {
```

```
#if defined(__MSP430_HAS_UART0__) || defined(__MSP430_HAS_UART1__)
                        while(!(*prtInf->usartRegs->IFG_REG & prtInf->usartRegs->TXIFGFlag));
                        *prtInf->usciRegs->TX_BUF = buf[i];
#endif
                }
        }

        return UART_SUCCESS;
}


/*!
 * \brief Sends a string over the specified UART
 *
 * This function is blocking, although interrupts may trigger during its
 * execution.
 *
 * @param prtInf is a pointer to the UART configuration
 * @param uartId is the unique UART identifier, usually provided by the call to configUart.
 * @param string is a pointer to the string to be sent
 *
 * \par The string provided to the function must be null terminated.
 *
 * \return 0 if successful , -1 if failed
 *
 */
int uartSendStringBlocking(UARTConfig * prtInf,char * string)
{
        int res = 0;
        res = uartSendDataBlocking(prtInf,(unsigned char*)string, strlen(string));

        return res;
}

/*!
 * \brief Sets the UART TX Buffer
 *
 *
 * @param prtInf is UARTConfig instance with the configuration settings
 * @param buf is a pointer to a user provided buffer to be used by the UART driver
 * @param bufLen the length of the buffer provided to the UART driver
 * \return Success or errors as defined by UART_ERR_CODES
 *
 */
void setUartTxBuffer(UARTConfig * prtInf, unsigned char * buf, int bufLen)
{
        prtInf->txBuf = buf;
        prtInf->txBufLen = bufLen;

        int i = 0;
        for(i = 0; i < bufLen; i++)
        {
                buf[i] = 0;
        }
}

/*!
 * \brief Sets the UART RX Buffer
 *
 *
 * @param prtInf is UARTConfig instance with the configuration settings
 * @param buf is a pointer to a user provided buffer to be used by the UART driver
 * @param bufLen the length of the buffer provided to the UART driver
 * \return Success or errors as defined by UART_ERR_CODES
 *
 */
void setUartRxBuffer(UARTConfig * prtInf, unsigned char * buf, int bufLen)
{
	prtInf->rxBuf = buf;
	prtInf->rxBufLen = bufLen;

	int i = 0;
	for(i = 0; i < bufLen; i++)
	{
		buf[i] = 0;
	}
}

/*!
 * \brief Sends len number of bytes from the buffer using the specified
 * UART using interrupt driven.
 *
 * TX Interrupts are enabled and each time that the UART TX Buffer is empty
 * and there is more data to send, data is sent. Once the byte is sent, another
 * interrupt is triggered, until all bytes in the buffer sent.
 *
 * @param prtInf is a pointer to the UART configuration
 * @param buf is a pointer to the buffer containing the bytes to be sent.
 * @param len is an integer containing the number of bytes to send.
 *
 * \return Success or errors as defined by UART_ERR_CODES
 *
 */
int uartSendDataInt(UARTConfig * prtInf,unsigned char * buf, int len)
{
	if(len > prtInf->txBufLen )
	{
		return UART_INSUFFICIENT_TX_BUF;
	}

	int i = 0;
	for(i = 0; i < len; i++)
	{
		prtInf->txBuf[i] = buf[i];
	}

#if defined(__MSP430_HAS_USCI__) || defined(__MSP430_HAS_USCI_A0__) ||
defined(__MSP430_HAS_USCI_A1__) || defined(__MSP430_HAS_USCI_A2__)
	// Send the first byte. Since UART interrupt is enabled, it will be called once the
byte is sent and will
	// send the rest of the bytes
	if(prtInf->moduleName == USCI_A0 || prtInf->moduleName == USCI_A1 || prtInf->moduleName
== USCI_A2)
	{
		prtInf->txBytesToSend = len;
		prtInf->txBufCtr = 0;

		// Enable TX IE
		*prtInf->usciRegs->IFG_REG &= ~UCTXIFG;
		*prtInf->usciRegs->IE_REG |= UCTXIE;

		// Trigger the TX IFG. This will cause the Interrupt Vector to be called
		// which will send the data one byte at a time at each interrupt trigger.
		*prtInf->usciRegs->IFG_REG |= UCTXIFG;
	}
#endif

#if defined(__MSP430_HAS_UART0__) || defined(__MSP430_HAS_UART1__)
	if(prtInf->moduleName == USART_0|| prtInf->moduleName == USART_1)
	{
		prtInf->txBytesToSend = len;
		prtInf->txBufCtr = 0;

		// Clear TX IFG and Enable TX IE
		*prtInf->usartRegs->IFG_REG &= ~ prtInf->usartRegs->TXIFGFlag;
		*prtInf->usartRegs->IE_REG |= prtInf->usartRegs->TXIE;

		// Trigger the TX IFG. This will cause the Interrupt Vector to be called
		// which will send the data one byte at a time at each interrupt trigger.
		*prtInf->usartRegs->IFG_REG |= prtInf->usartRegs->TXIFGFlag;

	}
#endif

	return UART_SUCCESS;
}

void enableUartRx(UARTConfig * prtInf)
{
#if defined(__MSP430_HAS_USCI__) || defined(__MSP430_HAS_USCI_A0__) ||
defined(__MSP430_HAS_USCI_A1__) || defined(__MSP430_HAS_USCI_A2__)
	if(prtInf->moduleName == USCI_A0|| prtInf->moduleName == USCI_A1 || prtInf->moduleName
== USCI_A2)
	{
		// Enable RX IE
		*prtInf->usciRegs->IFG_REG &= ~UCRXIFG;
		*prtInf->usciRegs->IE_REG |= UCRXIE;
	}
#endif

#if defined(__MSP430_HAS_UART0__) || defined(__MSP430_HAS_UART1__)
	if(prtInf->moduleName == USART_0|| prtInf->moduleName == USART_1)
	{
		// Enable RX IE
		*prtInf->usartRegs->IFG_REG &= ~	prtInf->usartRegs->RXIFGFlag;
		*prtInf->usartRegs->IE_REG |= prtInf->usartRegs->RXIE;
	}
#endif
}

#if defined(__MSP430_HAS_UART0__)
// UART0 TX ISR
#pragma vector=USART0TX_VECTOR
__interrupt void usart0_tx (void)
{
	// Send data if the buffer has bytes to send
	if(prtInfList[USART_0]->txBytesToSend > 0)
	{
		*prtInfList[USART_0]->usciRegs->TX_BUF = prtInfList[USART_0]-
>txBuf[prtInfList[USART_0]->txBufCtr];
		prtInfList[USART_0]->txBufCtr++;

		// If we've sent all the bytes, set counter to 0 to stop the sending
```

```
               if(prtInfList[USART_0]->txBufCtr == prtInfList[USART_0]->txBytesToSend)
               {
                 prtInfList[USART_0]->txBufCtr = 0;
               }
       }
}


// UART0 RX ISR
#pragma vector=USART0RX_VECTOR
__interrupt void usart0_rx (void)
{
        // Store received byte in RX Buffer
       prtInfList[USART_0]->rxBuf[prtInfList[USART_0]->rxBytesReceived] =
*prtInfList[USART_0]->usciRegs->RX_BUF;
       prtInfList[USART_0]->rxBytesReceived++;

       // If the received bytes filled up the buffer, go back to beginning
       if(prtInfList[USART_0]->rxBytesReceived > prtInfList[USART_0]->rxBufLen)
       {
         prtInfList[USART_0]->rxBytesReceived = 0;
       }
}

#endif

#if defined(__MSP430_HAS_UART1__)
// UART1 TX ISR
#pragma vector=USART1TX_VECTOR
__interrupt void usart1_tx (void)
{
       // Send data if the buffer has bytes to send
       if(prtInfList[USART_1]->txBytesToSend > 0)
       {
               *prtInfList[USART_1]->usciRegs->TX_BUF = prtInfList[USART_1]-
>txBuf[prtInfList[USART_1]->txBufCtr];
               prtInfList[USART_1]->txBufCtr++;

               // If we've sent all the bytes, set counter to 0 to stop the sending
               if(prtInfList[USART_1]->txBufCtr == prtInfList[USART_1]->txBytesToSend)
               {
                 prtInfList[USART_1]->txBufCtr = 0;
               }
       }
}


// UART1 RX ISR
#pragma vector=USART1RX_VECTOR
__interrupt void usart1_rx (void)
{
        // Store received byte in RX Buffer
       prtInfList[USART_1]->rxBuf[prtInfList[USART_1]->rxBytesReceived] =
*prtInfList[USART_1]->usciRegs->RX_BUF;
       prtInfList[USART_1]->rxBytesReceived++;

       // If the received bytes filled up the buffer, go back to beginning
       if(prtInfList[USART_1]->rxBytesReceived > prtInfList[USART_1]->rxBufLen)
       {
         prtInfList[USART_1]->rxBytesReceived = 0;
       }
```

```
}

#endif

#if defined(__MSP430_HAS_USCI__) || defined(__MSP430_HAS_USCI_A0__)
#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR(void)
{
       switch(__even_in_range(UCA0IV,4))
       {
         case 0:break;                              // Vector 0 - no interrupt
         case 2:                                    // Vector 2 - RXIFG
                // Store received byte in RX Buffer
                prtInfList[USCI_A0]->rxBuf[prtInfList[USCI_A0]->rxBytesReceived] =
*prtInfList[USCI_A0]->usciRegs->RX_BUF;
                prtInfList[USCI_A0]->rxBytesReceived++;

                // If the received bytes filled up the buffer, go back to beginning
                if(prtInfList[USCI_A0]->rxBytesReceived > prtInfList[USCI_A0]->rxBufLen)
                {
                       prtInfList[USCI_A0]->rxBytesReceived = 0;
                }
              break;
         case 4:                                    // Vector 4 - TXIFG
                // Send data if the buffer has bytes to send
                if(prtInfList[USCI_A0]->txBytesToSend > 0)
                {
                       *prtInfList[USCI_A0]->usciRegs->TX_BUF = prtInfList[USCI_A0]-
>txBuf[prtInfList[USCI_A0]->txBufCtr];
                       prtInfList[USCI_A0]->txBufCtr++;

                       // If we've sent all the bytes, set counter to 0 to stop the sending
                       if(prtInfList[USCI_A0]->txBufCtr == prtInfList[USCI_A0]-
>txBytesToSend)
                       {
                              prtInfList[USCI_A0]->txBufCtr = 0;

                              // Disable TX IE
                              *prtInfList[USCI_A0]->usciRegs->IE_REG &= ~UCTXIE;

                              // Clear TX IFG
                              *prtInfList[USCI_A0]->usciRegs->IFG_REG &= ~UCTXIFG;
                       }
                }
                break;
         default: break;
       }
}
#endif

#if defined(__MSP430_HAS_USCI__) || defined(__MSP430_HAS_USCI_A1__)
#pragma vector=USCI_A1_VECTOR
__interrupt void USCI_A1_ISR(void)
{
       switch(__even_in_range(UCA1IV,4))
       {
         case 0:break;                              // Vector 0 - no interrupt
         case 2:                                    // Vector 2 - RXIFG
                // Store received byte in RX Buffer
                prtInfList[USCI_A1]->rxBuf[prtInfList[USCI_A1]->rxBytesReceived] =
*prtInfList[USCI_A1]->usciRegs->RX_BUF;
```

```
                        prtInfList[USCI_A1]->rxBytesReceived++;

                        // If the received bytes filled up the buffer, go back to beginning
                        if(prtInfList[USCI_A1]->rxBytesReceived > prtInfList[USCI_A1]->rxBufLen)
                        {
                                prtInfList[USCI_A1]->rxBytesReceived = 0;
                        }
                      break;
                case 4:
                        // Send data if the buffer has bytes to send
                        if(prtInfList[USCI_A1]->txBytesToSend > 0)
                        {
                                *prtInfList[USCI_A1]->usciRegs->TX_BUF = prtInfList[USCI_A1]-
>txBuf[prtInfList[USCI_A1]->txBufCtr];
                                prtInfList[USCI_A1]->txBufCtr++;

                                // If we've sent all the bytes, set counter to 0 to stop the sending
                                if(prtInfList[USCI_A1]->txBufCtr == prtInfList[USCI_A1]-
>txBytesToSend)
                                {
                                        prtInfList[USCI_A1]->txBufCtr = 0;

                                        // Disable TX IE
                                        *prtInfList[USCI_A1]->usciRegs->IE_REG &= ~UCTXIE;

                                        // Clear TX IFG
                                        *prtInfList[USCI_A1]->usciRegs->IFG_REG &= ~UCTXIFG;
                                }
                        }
                        break;                                  // Vector 4 - TXIFG
                default: break;
        }
}
#endif

#if defined(__MSP430_HAS_USCI__) || defined(__MSP430_HAS_USCI_A2__)
#pragma vector=USCI_A2_VECTOR
__interrupt void USCI_A2_ISR(void)
{
        switch(__even_in_range(UCA2IV,4))
        {
          case 0:break;                                         // Vector 0 - no interrupt
          case 2:                                               // Vector 2 - RXIFG
                        // Store received byte in RX Buffer
                        prtInfList[USCI_A2]->rxBuf[prtInfList[USCI_A2]->rxBytesReceived] =
*prtInfList[USCI_A2]->usciRegs->RX_BUF;
                        prtInfList[USCI_A2]->rxBytesReceived++;

                        // If the received bytes filled up the buffer, go back to beginning
                        if(prtInfList[USCI_A2]->rxBytesReceived > prtInfList[USCI_A2]->rxBufLen)
                        {
                                prtInfList[USCI_A2]->rxBytesReceived = 0;
                        }
                      break;
                case 4:
                        if(prtInfList[USCI_A2]->txBytesToSend > 0)
                        {
                                *prtInfList[USCI_A2]->usciRegs->TX_BUF = prtInfList[USCI_A2]-
>txBuf[prtInfList[USCI_A2]->txBufCtr];
                                prtInfList[USCI_A2]->txBufCtr++;
                                // If we've sent all the bytes, set counter to 0 to stop the sending
                                if(prtInfList[USCI_A2]->txBufCtr == prtInfList[USCI_A2]-
>txBytesToSend)
                                {
                                        prtInfList[USCI_A2]->txBufCtr = 0;

                                        // Disable TX IE
                                        *prtInfList[USCI_A2]->usciRegs->IE_REG &= ~UCTXIE;

                                        // Clear TX IFG
                                        *prtInfList[USCI_A2]->usciRegs->IFG_REG &= ~UCTXIFG;
                                }
                        }
                        break;                          // Vector 4 - TXIFG
                default: break;
        }
}
#endif

/*!
 * \brief Reads bytes from the Rx buffer
 *
 *      Note that when bytes are read using this function, their count will be reset.
 *      Therefore, it is recommended to read all bytes of interest at once and buffer on
 *      the user side as necessary.
 *
 * @param prtInf is UARTConfig instance with the configuration settings
 * @param data is a pointer to a user provided buffer to which the data will be written
 * @param numBytesToRead is the number of bytes to read from the buffer
 * @param offset is the offset in the buffer to read. Default is 0 to start at beginning of RX
buffer
 * \return number of bytes placed in the data buffer
 *
 */
int readRxBytes(UARTConfig * prtInf, unsigned char * data, int numBytesToRead, int offset)
{
        int bytes = 0;

        // Ensure we don't read past what we have in the buffer
        if(numBytesToRead+offset <= prtInf->rxBytesReceived )
        {
                      bytes = numBytesToRead;
        }
        else if(offset < prtInf->rxBufLen)
        {
                      // Since offset is valid, we provide all possible bytes until the end of the
buffer
                      bytes = prtInf->rxBytesReceived - offset;
        }
        else
        {
                      return 0;
        }

        int i = 0;
        for(i = 0; i < bytes; i++)
        {
                      data[i] = prtInf->rxBuf[offset+i];
        }

        // reset number of bytes available, regardless of how many bytes are left
        prtInf->rxBytesReceived = 0;

        return i;

}
```