Lab 2: Hardware Manipulation and Timers with the ESP32

Overview

Work directly with the ESP32's hardware, directly manipulating registers, utilizing hardware timers, and interfacing with peripherals like LEDs, sensors, etc.

Objectives

- Understand and manipulate memory using pointers.
- Control hardware directly through registers.
- Implement timing exercises to manage tasks.
- Interface and control simple peripherals.
- Read and react to analog inputs from sensors.

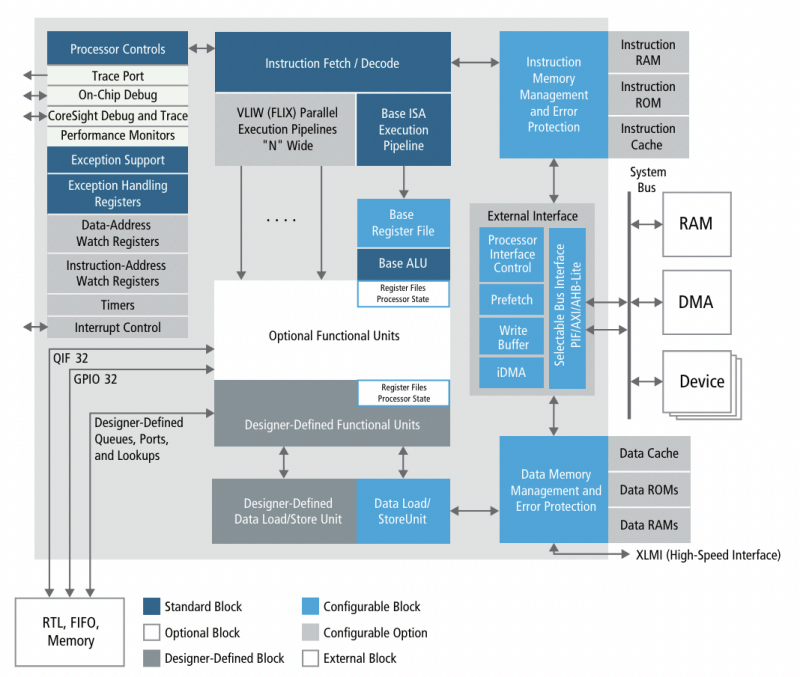


Figure 1: Block diagram of the architecture for the Xtensa LX7 processor used in the Arduino Nano ESP32 (source pdf)

References:

- Lab Quick Reference
- Bitwise Operations
- Pointers, Video Series
- Video on Registers
- Clocks and Timers

Part I: Pointers and Registers (15 points)

The Basics

Pointers and registers are fundamental concepts in embedded systems programming, providing direct control over hardware resources. Pointers act as memory address holders, and enable direct manipulation of hardware data. Registers, on the other hand, are dedicated memory locations within the microcontroller that controls hardware functions and store status information. By using pointers to access and modify register values, you gain precise control over hardware, allowing for optimized performance, real-time control, and low-level peripheral interaction.

Volatile Keyword

The '**`volatile`**' keyword tells the compiler that the value of the variable (or memory address) can change at any time without any action being taken by the code the compiler sees:

```
volatile int statusRegister;
```

Registers can be modified by hardware events, interrupts, or other parts of the system outside the current scope of code. Without `volatile`, the compiler might optimize away repeated accesses to the same memory location, assuming the value won't change, which is not always true. By marking a variable as '`volatile`', you make sure that every read and write operation accesses the actual memory location every time.

Step 1: GPIO Control via Registers

In this step we will replicate controlling an external LED from Part III of Lab 1, but through directly accessing and manipulating hardware registers.

Materials:

- ESP32 Development Board
- Breadboard
- LED (any color)
- 220 Ohm resistors
- Jumper wires

GPIO Registers

In this step, we will use and manipulate two 32-bit registers through predefined macros:

```
// This macro points to the memory address of the GPIO enable
// register, which controls whether a pin is input or output
GPIO_ENABLE_REG

// This macro points to the memory address of the GPIO output
// register, which controls whether a pin outputs HIGH (1) or LOW (0)
// voltage.
GPIO_OUT_REG
```

So, for instance, if the 32-bit GPIO enable register is set to:

```
0000 0000 0000 0000 0000 0000 0110 0000
```

Then GPIO 5 and 6 are set to **`OUTPUT`** and the rest default to **`INPUT`**. (See here for the ESP32-S3 DevKitC-1 pin layout)

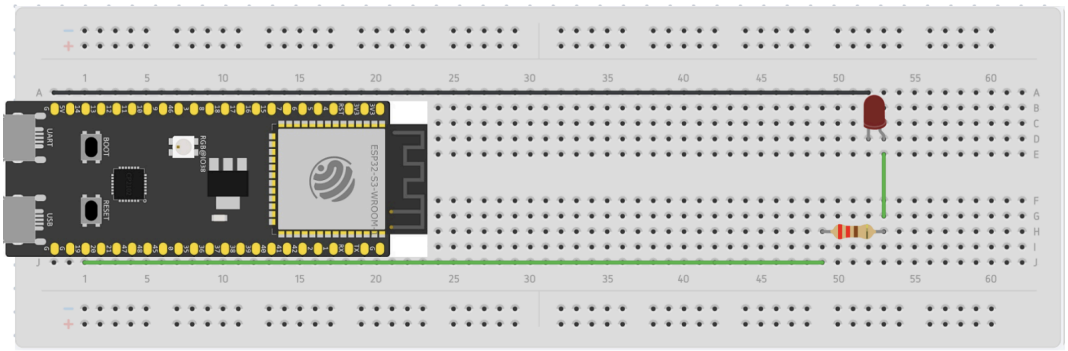


Figure 2: Controlling an external LED

```
#include "driver/gpio.h"
#include "soc/io_mux_reg.h"
#include "soc/gpio_reg.h"
#include "soc/gpio_periph.h"
#define GPIO_PIN 5 // GPIO5
void setup() {
 // This predefined macro sets the functionality of a specified pin (to UART, SPI,
 // GPIO, etc). Here we are setting pin 5 to be a general purpose input/output
 // pin.
 PIN_FUNC_SELECT(GPIO_PIN_MUX_REG[GPIO_PIN], PIN_FUNC_GPIO);

// ===============> TODO:
```

```
 // Write code here (1-3 lines) to mark pin 5 as output
 // using the GPIO_ENABLE_REG macro
}
void loop() {
 // ===============> TODO:
 // Turn LED on (set corresponding bit in GPIO output register)
 // Write code here (1-3 lines) to mark pin 5 as HIGH output
 // using the GPIO_OUT_REG macro
 // Wait for 1 second

 // ===============> TODO:
 // Turn LED off (clear corresponding bit in GPIO output register)
 // Write code here (1-3 lines) to mark pin 5 as LOW output
 // using the GPIO_OUT_REG macro
 // Wait for 1 second
}
```

Figure 3: Pseudocode for Step 2

Task:

- Build a circuit similar to the one shown in *Figure 2* above.
- Using the provided skeleton code (*Figure 3*), write your sketch to control the external LED, with the LED blinking on and off in one second intervals.
    - NOTES:
        - DO NOT use `pinMode()` or `digitalWrite()`
        - Recall that `GPIO_ENABLE_REG` and `GPIO_OUT_REG` are macros that store the *address* of the GPIO enable and GPIO output registers, respectively. You must first cast these macros as (volatile) 32-bit unsigned integer pointers, then access the contents stored at these addresses before setting/clearing any bits.
        - DO NOT simply assign a value to a register - you must use bitwise operations to *only* change your target bits. Some bits within a 32-bit register are reserved for other functionality, so not only is it good practice to utilize bitwise operations, it also reduces the likelihood of bricking your microcontroller.

Step 2: Comparing Times

In this step we will compare the execution speed of arduino library functions vs direct register access.

Arduino Library Functions

The `millis()` and `micros()` methods in Arduino return the amount of time since the program started running in milliseconds and microseconds, respectively. These functions are particularly useful for tasks that require precise time intervals, such as measuring the duration of pulses, creating precise delays, and synchronizing events.

Task:

- Using the below pseudo-code (*Figure 4*) and the `micros()` method, measure the time it takes for `digitalWrite()` to change the output voltage to `HIGH` and then back to `LOW`.
- Write a second sketch that repeats the above, but instead of `digitalWrite()` use direct register access like in Step 2. Record the two times.
- NOTE: To emphasize these differences, we ask you to use control flow to measure the total time to rapidly change output from `LOW->HIGH` for some large number of repetitions (say 1000). Print out the total time to complete all repetitions, and then compare between the two methods (library functions vs direct register access).

```
void setup() {
 // ================> TODO:
 // Set a pin as output
}
void loop() {
 // ================> TODO:
 // For 1000 repetitions:
 //          Measure time to:
 //                 - Turn pin's output to HIGH
 //                 - Turn pin's output to LOW
 // Print out total time to the serial monitor
 // 1 second delay
}
```

Figure 4: Pseudocode for Step 3

Part I Deliverables:

- A live demo of the LED blinking from Step 1.
- Step 1 sketch code (with comments)
- Step 2 sketch code (with comments)
    - Measuring Direct Register Access
    - Measuring Arduino Library Functions
- ***Lab Report Question***
    - In your Lab 2 Report discussion section answer the following question: Using your measured times from Step 3, compare and contrast the advantages and disadvantages of library functions vs direct register access.

Part II: Timing and Synchronization (20 points)

Step 3: Using ESP32 Timers

Timers are crucial components in microcontrollers that allow precise control over the timing of events. They can generate periodic interrupts, create accurate delays, and measure time intervals. In simple terms, a timer is like a stopwatch built into the microcontroller that can trigger actions when certain time periods elapse. The ESP32 microcontroller has multiple hardware timers which can be used to execute functions without blocking the main program.

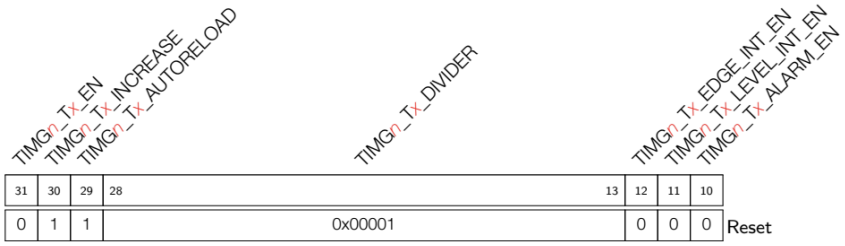
Materials:

- ESP32 Development Board
- Breadboard
- External LED (any color)
- 220 Ohm resistor
- Jumper wires

Timer Registers

In this step, we will directly interact with the ESP32's timer registers to control an LED. For our timer, we'll focus on three key registers:

- **`TIMG_T0CONFIG_REG(0)`**: This macro stores the address of the register that configures Timer 0 in Timer Group 0. It allows us to set the prescaler (which divides the main clock frequency to a lower frequency suitable for our timer), enable auto-reload (which makes the timer reset automatically when it reaches a specified value), and start the timer.



TIMG*n*_T*x*_EN When set, the timer *x* time-base counter is enabled. (R/W)

TIMG*n*_T*x*_INCREASE When set, the timer *x* time-base counter will increment every clock tick. When cleared, the timer *x* time-base counter will decrement. (R/W)

TIMG*n*_T*x*_AUTORELOAD When set, timer *x* auto-reload at alarm is enabled. (R/W)

TIMG*n*_T*x*_DIVIDER Timer *x* clock (T*x*_clk) prescale value. (R/W)

Figure 5: Timer configuration register breakdown (ESP reference manual PDF)

- **`TIMG_T0LO_REG(0)`**: This macro stores the address of the register that holds the current value of the lower 32 bits of Timer 0. By reading this register, we can determine how much time has passed since the timer started.
- **`TIMG_T0UPDATE_REG(0)`**: This macro stores the address of the Update Register. Writing to this register triggers an update, ensuring that the current timer value is copied to the `TIMG_T0LO_REG(0)` register for us to read.

Task:

- Using the registers specified above, specify an incrementing counter and an appropriate prescaler (the default clock source is the 80 MHz APB_CLK).
    - Make sure you add the following includes statement in your sketch: **`#include "soc/timer_group_reg.h"`**
    - Ensure you are setting the correct bits within the configuration register shown above. **DO NOT** simply assign a value to a register - you must use bitwise operations to *only* change your target bits.
    - Remember that these macros store the *address* of the registers we need to interact with.
- Use the oscilloscope to ensure that the timer is working as intended:
    - Connect the Oscilloscope:
        - Connect one of the oscilloscope probes to the GPIO pin you are controlling (e.g., the pin connected to the LED).
        - Connect the oscilloscope's ground clip to the ground of your ESP32 circuit.
    - Configure the Oscilloscope:
        - Set the oscilloscope to the appropriate voltage range to capture the HIGH and LOW states of your signal.

        - Adjust the time base to capture the duration of your signal's on and off periods accurately.
    - Measure the Timing:
        - Run your code that toggles the GPIO pin.
        - Observe the waveform on the oscilloscope screen. You should see a square wave corresponding to the LED blinking pattern.
        - Use the oscilloscope's measurement tools to determine the timing of the signal. For instance, measure the time between rising edges or the width of the pulses.
    - Analyze the Results:
        - Compare the oscilloscope measurements with the expected timing from your code.
        - If there are discrepancies, use the visual data to debug and adjust your code accordingly.

Deliverables for Part II:

- A live demo of the LED blinking from Step 3 with live oscilloscope measurements.
- Step 3 sketch code (with comments)

Part III: Peripheral Interaction (15 points)

Step 4: Controlling Peripherals with a Photoresistor

In this step, you will learn how to use a photoresistor to control the brightness of an LED. This involves reading analog values from the photoresistor and using those values to adjust the LED's brightness.

Materials:

- ESP32 Development Board
- Breadboard
- Photoresistor
- 10K Ohm resistor (for creating a voltage divider)
- Jumper wires
- LED
- 220 Ohm resistor

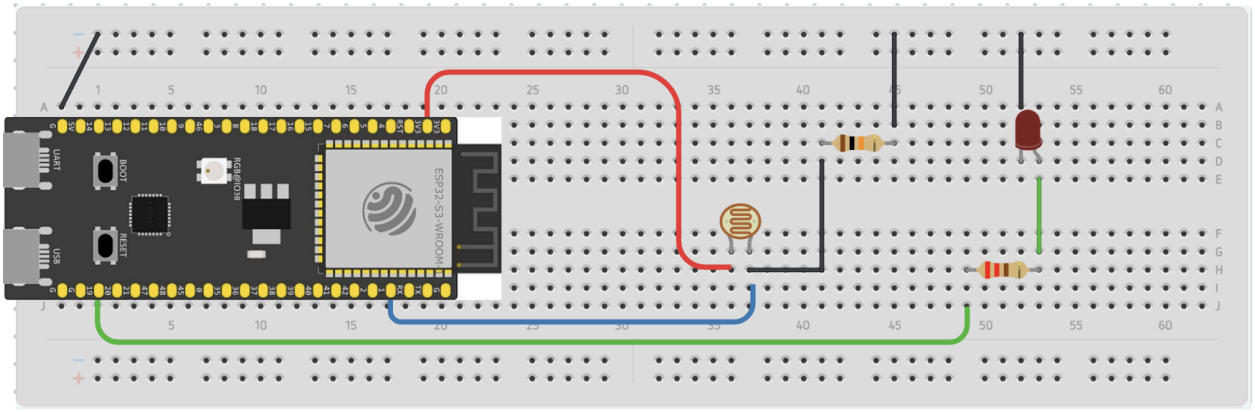


Figure 6: Photoresistor controlling an LED brightness

LEDC Library Functions

The LED PWM controller (LEDC) on the ESP32, which uses dedicated hardware timers, is designed to generate PWM signals for various purposes including the brightness of LEDs. The ESP32 supports up to 8 PWM channels, which can generate independent waveforms for controlling multiple devices. Starting with the updated ESP32 Arduino core (v3.x+), a new simplified set of functions replaces the older `ledcSetup`, `ledcAttachPin`, and `ledcWrite(channel, duty)` methods. The key updated functions are:

Some key methods in this library are:

- **`ledcAttach(pin, freq, res)`**: Configures the PWM signal on the specified GPIO pin.
  - **`pin`**: GPIO pin number to output the PWM signal.
  - **`freq`**: Frequency of the PWM signal in Hertz.
  - **`res`**: Resolution of the PWM signal (1-14 bits).
- **`ledcWrite(pin, duty)`**: Sets the duty cycle for the PWM signal on the specified pin.
  - **`pin`**: GPIO pin number to write the duty cycle to.
  - **`duty`**: Duty cycle to set (from 0 to $2^{res} - 1$).

Arduino Library Functions

- **`analogRead()`**: this function reads the voltage on an analog input pin and returns a digital value corresponding to that voltage. This value, usually ranging from 0 to 4095 depending on the microcontroller, represents the analog signal's intensity.

Task:

- Build the circuit according to *Figure 6* above:
    - Connect one end of the photoresistor to the 3.3V supply and the other end to a node on the breadboard.
    - Connect one end of the 10K Ohm resistor to the same node and the other end to a common ground.
    - Connect a wire from the node between the photoresistor and the 10K Ohm resistor to an analog input pin on the ESP32.
    - Connect the LED to a digital output pin with a 220 Ohm resistor in series.
- Write a sketch which reads the ambient lighting in the room and adjusts the brightness of the LED using the LEDC library functions described above and **`analogRead()`**. The LED brightness is mapped to the reading of the photoresistor: when the photoresistor detects less light (a higher resistance or lower analog value), the LED should gradually get brighter. This creates a visible effect where the LED brightness increases smoothly as the ambient light decreases

    - *NOTE*: You may *NOT* use **`analogWrite()`**.

Deliverables for Part III:

- A live demo of the LED responding to changes in ambient lighting.
- Step 4 sketch code (with comments)

Part IV: Ambient Light-Triggered Buzzer Frequency Sequence (20 points)

Step 5: Using a Photoresistor to Set an Alarm

In this step, you will use a photoresistor, the LEDC library, and timer logic to control a buzzer. When the ambient light level drops below a certain threshold, the buzzer will play a sequence of increasing sound frequencies (low → medium → high). If the ambient light is above the threshold, the buzzer remains silent.

This task combines the analog input logic from Part III and the timer-based control from Part II. Instead of adjusting LED brightness, you will generate sound on a buzzer using PWM at different frequencies.

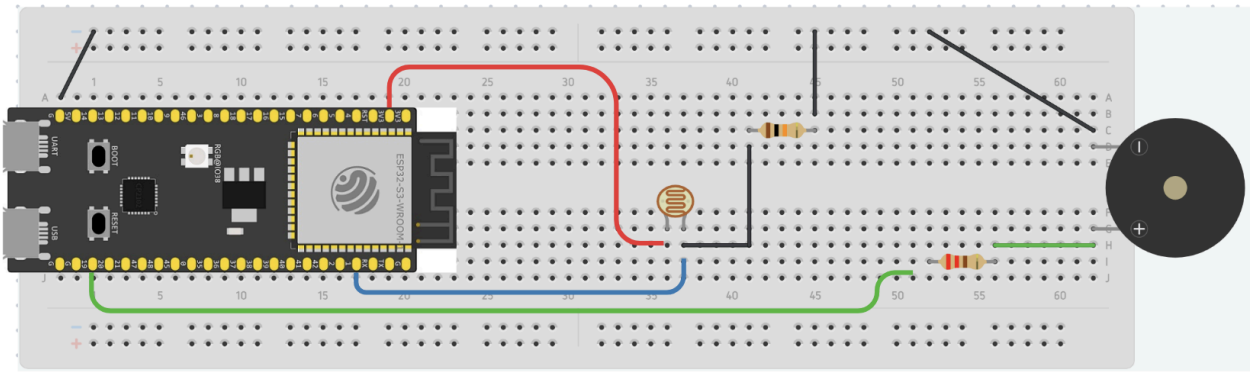


Figure 7: Photoresistor controlling Buzzer frequency sequence

Task:

1. Build the Circuit:
   Set up the photoresistor and 10K resistor as a voltage divider (as in Part III).
   - Connect the midpoint to an analog input pin on the ESP32.
   - Connect the buzzer to a digital output pin capable of PWM.
   - Ensure the buzzer shares a common ground with the ESP32.
2. Write your Sketch:

- You may use `analogRead()` to monitor the photoresistor value.
- Set a light threshold.
- If the light level is below or above the threshold (depends on the way you want to design it):
  - Start a one-time buzzer sequence using a timer.
  - Split the sequence into three time segments:
    - First third: play a **low frequency** (e.g., 500 Hz)
    - Second third: play a **medium frequency** (e.g., 1000 Hz)
    - Final third: play a **high frequency** (e.g., 2000 Hz)
  - Use `ledcAttach()` to configure the buzzer pin for PWM.
  - Use `ledcWrite()` with a non-zero duty cycle to produce sound.
- If the light level doesn't meet your threshold (either above or below, depends on your design):
  - Stop the sequence and turn the buzzer off by setting `ledcWrite()` to 0.

Note:
Do not use `tone()` or `analogWrite()`. You must use the **LEDC library functions**.

Deliverables for Part IV:

- A live demo of the buzzer producing a sequence of sounds**.**
- Step 5 sketch code (with comments)

Conclusion

In this lab, we learned about direct hardware manipulation and timing through registers and pointers. By understanding and controlling memory via pointers, manipulating GPIO registers, and using hardware timers, we can achieve precise hardware control and efficiency. In this lab we progressed from basic pointer and register manipulation to more complex tasks such as measuring execution times for register access versus library functions, and integrating peripheral control with sensors and LEDs, introducing the advantages of low-level programming in embedded systems.

Final Deliverables Recap

Live Demos:

- *Part I:* The external LED blinking from Step 1.
- *Part II:* The external LED blinking from Step 3 with live oscilloscope measurements.
- *Part III:* The LED responding to changes in ambient lighting from Step 4.
- *Part IV*: The passive buzzer producing a sequence of sounds from Step 5.

Code:

NOTE: Please follow the Code Guidelines when documenting your code.

The sketch files for the following sections should be descriptively named (e.g. Lab2Part2.ino), zipped together in a single folder containing your last name and the lab number (e.g. LastNameLab2.zip), and submitted through Canvas:

- *Part I:*
    - Step 1: The external LED blinking from direct register access.
    - Step 2: Comparing times between library functions and direct register access.
- *Part II:* The external LED blinking with direct register access from step 3.
- *Part III:* The LED responding to changes in ambient lighting from step 4.
- *Part IV*: The passive buzzer producing a sequence of sounds from Step 5.

Report:

- Follow the outline and instructions listed in the Lab Report Template here.
- Make sure to go over the following in the discussion section:
    - Using your measured times from **Part I**, compare and contrast the advantages and disadvantages of library functions vs direct register access.

- Submit your Lab Report in pdf form *separately* from the zip file containing your code.

Grading

Lab 2 grading will be based on the Live Demos, Code, and Report:

- The Demonstrations will be graded as Complete, Attempted, or Incomplete and requires a visual confirmation by an instructor. **In total 150 points.**
- The code portion will be graded according to the following rubric:

Criteria	Exceptional (28-30 points)	Good (25-27 points)	Average (22-24 points)	Poor (<22 points)
Code	• Thoroughly documented according to guidelines • Well-organized and efficient • Compiles without errors • Performs expected behavior consistently	• Mostly documented with minor unexplained sections • Mostly well-organized • Compiles, potentially with minor modifications. • Largely performs expected behavior with some minor deviations	• Some documentation is present, but significant sections are unexplained • Somewhat organized • Compiles with modifications • Some expected behavior, but inconsistencies present	• No documentation • Poor organization • Does not compile • Unclear what the code was attempting to do.

- The report portion will be graded according to the following criteria:

Criteria	Exceptional (9-10 points)	Good (8-9 points)	Average (7-8 points)	Poor (<7 points)
Report	• Clear introduction and objectives. • Detailed design rationale and implementation. • Thorough overview of code organization. • In-depth discussion of technical difficulties, team collaboration, and answers to lab questions. • Strong summary of project outcomes. • Consistent citation style with all resources listed. • Detailed contributions and hours spent.	• Mostly clear introduction and objectives. • Comprehensive design and implementation description with minor gaps. • Good overview of code organization, explaining main modules and their roles. • Good discussion of technical difficulties, team collaboration, and answers to lab questions with minor omissions. • Adequate summary of project outcomes. • Mostly consistent citation style with relevant resources listed. • Clear contributions and hours spent.	• Adequate introduction and objectives with missing details. • Basic design and implementation description may have significant gaps. • Overview of code organization with some unclear modules. • Basic discussion of technical difficulties, team collaboration, with noticeable omissions. • Basic summary of project outcomes. • Limited citation style with some resources listed. • Basic contributions and hours spent.	• No, basic, or unclear introduction and objectives. • Incomplete or poorly described design and implementation. • Poor overview of code organization with unclear or missing modules. • Minimal discussion of difficulties and collaboration with major omissions. • Brief and ineffective summary of project outcomes • Inconsistent or missing citations. • Incomplete contributions and hours spent.

Total: 150 points (Code: 50 + Report: 30 + Demo: 70)

Extra Credit: +5% if submitted by Saturday deadline