Automated Febreze Project Report

Name: Trevin LaVoie
Instructor: Professor Plett
Date: 12/17/2025

Project Description:

The system uses an AVR ATtiny45 microcontroller to control a servo motor via PWM. The servo triggers the Febreze spray at intervals set by the user. Powered through a 9V battery, using a voltage regulator to step down the 9V battery to 5V, powers the microcontroller to control the servo motor.

The device operates in two modes: **automatic** and **manual**. In automatic mode, the microcontroller uses internal timers to trigger the servo at predefined intervals. In manual mode, a push-button allows the user to activate the spray on demand. Manual mode offers flexibility for immediate use, while automatic mode is designed for consistency. This dual-mode functionality sets the design apart from typical single-mode dispensers, which are limited to either manual or automatic.

Additional design considerations include compact packaging on a custom PCB and component selection to balance cost, performance, and reliability. Future enhancements could incorporate adjustable spray duration, indicator lights for low fragrance levels, and wireless connectivity for remote controls.

Parts Lists:

Index	Part Number	Description	Quantity	Unit Price	Cost
1	LM340AT-5.0/NO*PB*	IC REG LINEAR 5V 1.5A TO220-3	1	$ 1.80	$ 1.80
2	PJ-202A	CONN PWR JACK 2X5.5MM KINKED PIN	1	$ 0.70	$ 0.70
3	80	BATT CONNECT SNAP 9V BARREL PLUG	1	$ 2.95	$ 2.95
4	1N4004G-T	DIODE STANDARD 400V 1A DO41	1	$ 0.20	$ 0.20
5	C320C224M5U5TA7301	CAP CER 0.22UF 50V Z5U RADIAL	2	$ 0.44	$ 0.88
6	EG1218	SWITCH SLIDE SPDT 200MA 30V	1	$ 0.84	$ 0.84
7	ATTINY45-20PU	IC MCU 8BIT 4KB FLASH 8DIP	1	$ 1.54	$ 1.54
8	CF14JT33K0	RES 33K OHM 5% 1/4W AXIAL	1	$ 0.10	$ 0.10
9	61300311121	CONN HEADER VERT 3POS 2.54MM	3	$ 0.12	$ 0.36
10	TS02-66-90-BK-160-SCR-D	SWITCH TACTILE SPST-NO 0.05A 12V	1	$ 0.12	$ 0.12
11	N/A	Circuit Board (Front & Back)	1	$ 10.0000	$ 10.00
12	N/A	1/16–3/16 in Roll Pin	2	$ 0.1000	$ 0.20
13	N/A	4 mm Snap Ring	2	$ 0.2000	$ 0.40
14	N/A	3/16 Ball Bearing	2	$ 8.0000	$ 16.00
15	N/A	75 mm Rod	1	$ 1.0000	$ 1.00
16	N/A	3 mm RC Connectors	2	$ 0.5000	$ 1.00
17	N/A	Brass Rod (75 mm or similar)	1	$ 1.0000	$ 1.00
18	N/A	Eco Boost Servo	1	$ 12.0000	$ 12.00
19	N/A	Strap Buckle	1	$ 1.0000	$ 1.00
20	N/A	9V Raymond’s Battery	1	$ 2.0000	$ 2.00
				Total:	$ 54.09

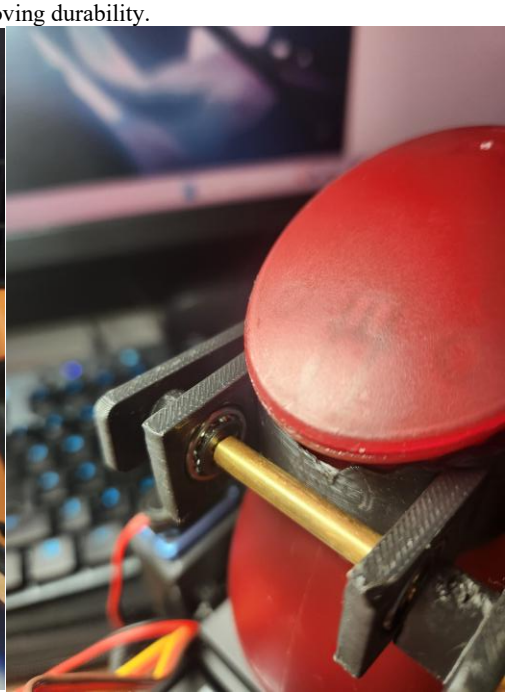
Automated Febreze Project Report

The BOM for this project includes electronic, mechanical, and 3-D printed parts. Key electronic components include the LM340AT-5.0 voltage regulator, which ensures a stable 5V supply for the ATtiny45 microcontroller, and the ATTINY45-20PU MCU, which controls the servo motor. Supporting components such as resistors, capacitors, and connectors were used to provide noise control and a stable power connection.



Mechanical parts like the roll pins, snap rings, and ball bearings were used for securing the servo mechanism and supporting a smooth motion for triggering the Febreze. The Eco Boost Servo provides the mechanical torque needed to press the Febreze trigger. Additionally, the custom PCB circuit board integrates all electronic components into a compact layout, reducing wiring complexity and improving durability.





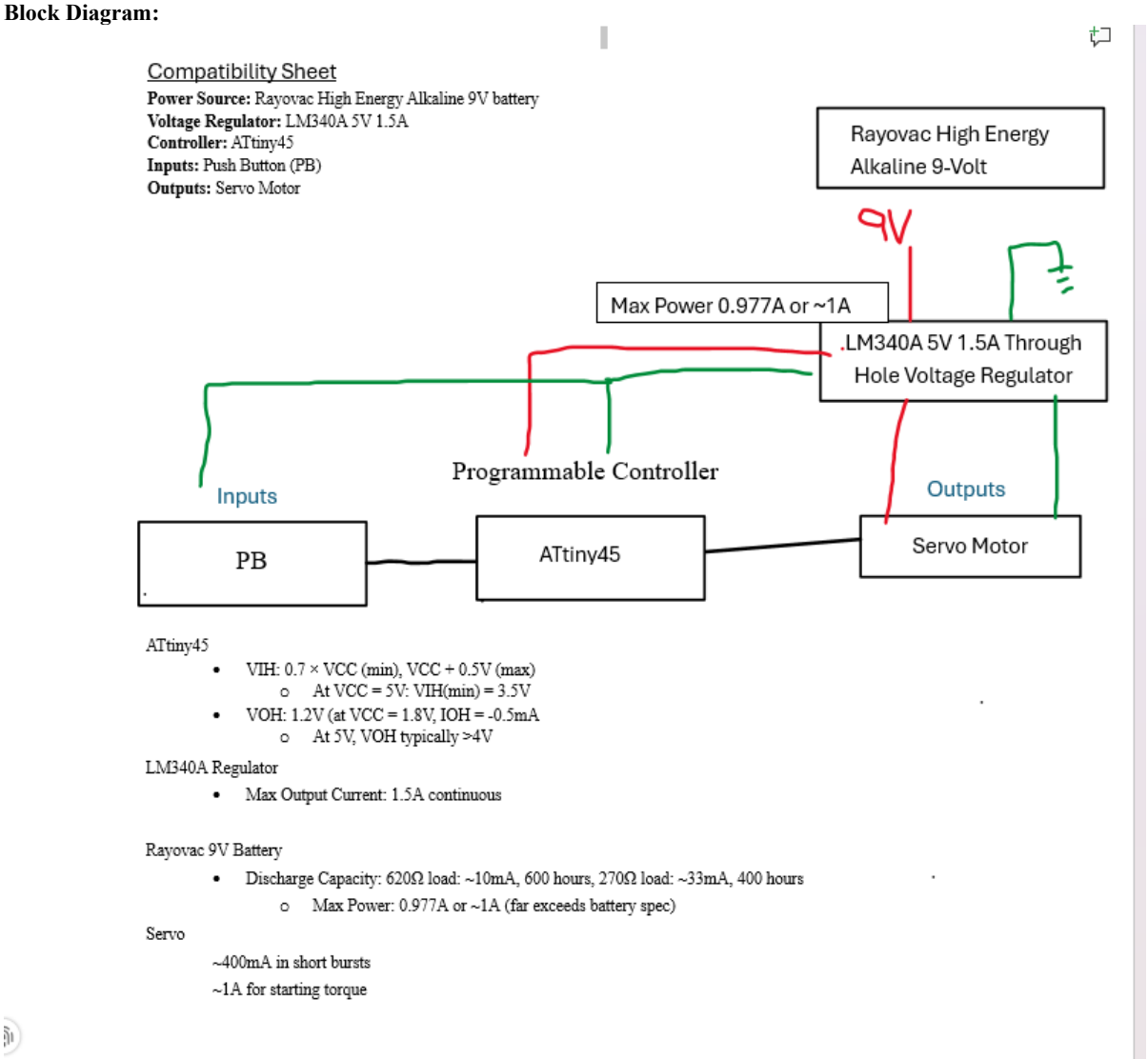
From the BOM analysis, the highlighted are the three highest-cost items are:

- **3/16 Ball Bearings** – $16
- **Eco Boost Servo** – $12
- **Circuit Board (Front & Back)** – $10

Together, these account for approximately 70% of the total cost ($54.09).

Further Discussion: How to reduce costs without compromising functionality? Options might include sourcing alternative servo models with similar torque ratings, using lower-cost bearings or mechanical substitutes, or exploring PCB fabrication services that offer competitive pricing. Could 3D-printed parts replace some mechanical components to further optimize cost?

Block Diagram:



The block diagram shows the ATtiny45 microcontroller as the central control unit, powered by a 9V battery regulated to 5V. It drives the servo motor through PWM signals to activate the Febreze spray. A push button provides manual input, while timers inside the microcontroller manage automatic spray intervals.

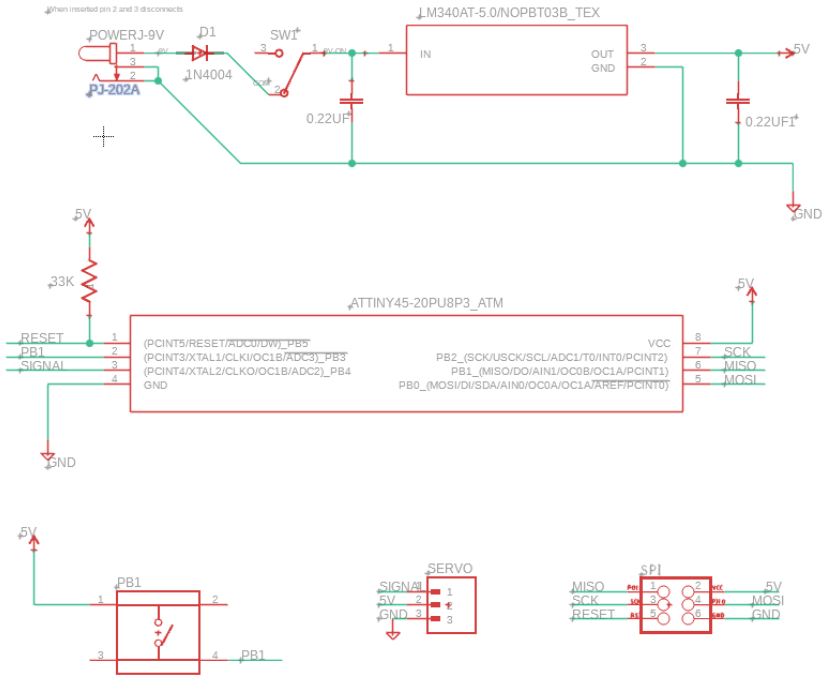
A better battery design is needed because the current 9V alkaline battery cannot supply the high current required by the servo motor, leading to reduced performance and shorter operating life.



The high surge current required by the servo motor, which can draw up to 1A during startup, dramatically reduces the operating life. These effects lead to the battery dropping below voltage spec, reducing torque, and shortening the overall effectiveness of the design. Additionally, alkaline batteries have limited capacity under high loads, causing frequent replacements and inefficiency.

To improve performance and power efficiency, a better consideration would be a battery rated for higher current output, such as a Li-ion pack or NiMH rechargeable cells. These options provide better discharge characteristics, longer runtime, and can support the servo's peak current demands without compromising system stability.

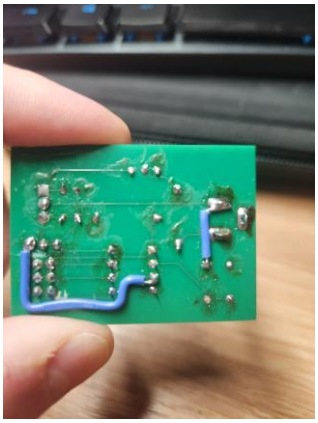
Wiring Diagram



Automated Febreze Project Report

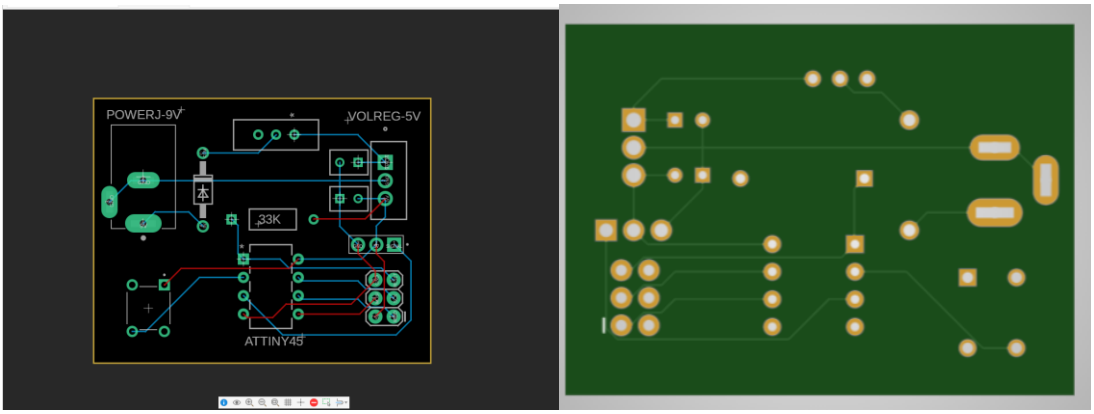
The circuit diagram shows the following main functions:

- **Voltage Regulation:** The LM340AT-5.0 regulator converts the 9V battery input to a stable 5V supply for the microcontroller and servo.
- **Microcontroller Control:** The ATtiny45 handles PWM signals to control the servo motor and reads input from the push button.
- **Protection & Filtering:** A diode (1N4004) prevents reverse polarity, and capacitors smooth voltage fluctuations.
- **Servo Interface:** The servo is connected to PB1 for signal control, powered by the regulated 5V line.
- **Push Button Input:** Connected to PB3, allowing manual spray activation.



Some design flaws include:

- Traces routed too close to pins, increasing the risk of soldering errors.
- Using pull-down instead of pull-up for the push button, which is less reliable and more prone to noise.

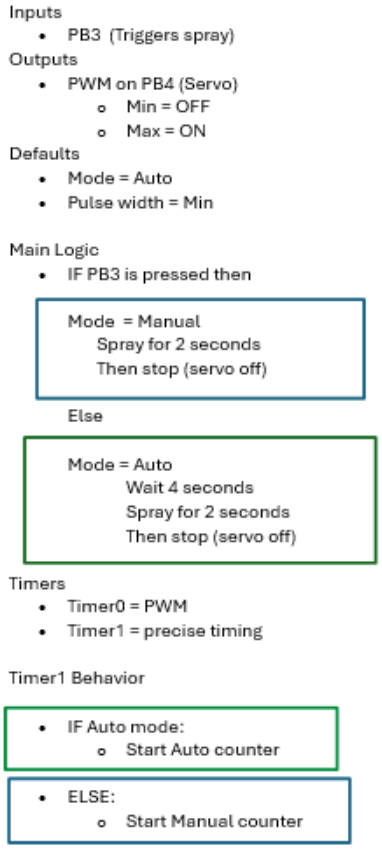


Good vs Bad Routing

- **Good Routing (Left):** Leaves enough clearance between traces and component pins, reducing the risk of solder bridges.
- **Bad Routing (Right):** Places traces right against the pins, which can cause shorts during soldering and make repairs difficult.

Automated Febreze Project Report

Flowchart: (Pseudo Code)



Main Idea

The pseudo-code outlines the control logic for the automated Febreze bottle, focusing on two modes: manual and automatic. The overall idea is to ensure that the system responds to user input while maintaining timed spray intervals for convenience and efficiency.

Manual and Automatic Mode

In manual mode, the push button acts as the trigger. When pressed, the servo activates for two seconds to spray and then stops, preventing excessive use. In automatic mode, the system waits for a preset interval, sprays for two seconds, and then returns to idle, repeating this cycle without user intervention.

The reason there are two separate variables for counting in manual and automatic modes is to keep the timing logic independent for each mode. This prevents interference between the manual spray trigger and the automatic interval cycle, ensuring accurate timing and smooth operation in both modes.

Additional Features

Future improvements could include adjustable spray duration, customizable intervals, and integration of sensors to detect when empty or full. This would make the system more energy-efficient and user-friendly.

Additionally, a good future improvement for saving power consumption would be adding an idle or sleep mode for the microcontroller when the system is not actively spraying. This reduces unnecessary energy use and extends battery life, whereas in the current model, the servo is ON at all times.

Main Code

```
int main (void) {
	configure_IO(); //PB3 = Output & PB4 = Input
	start_Timer0(); //Counts from 0 to 256
	start_Timer1();

	while(1) {
		if(bit_low(PINB, bit(PB3))) {
			mode = Manual;
			if(frequency_ManualCount < 200) {
				PW_ramp();
			} else {
				frequency_ManualCount = 200;
				pulsewidth = Min;
				OCR0A = pulsewidth;
			}
		} else {
			frequency_ManualCount = 0;
			mode = Auto;
			if(second_AutoCount > 9) {
				frequency_AutoCount = 0;
				second_AutoCount = 0;
				minute_AutoCount = 0;
				while (second_AutoCount < 2) {
					PW_ramp();
				}
				pulsewidth = Min;
				OCR0A = pulsewidth;
			} else {
				pulsewidth = Min;
				OCR0A = pulsewidth;
			}
		}
	}
}
```

main(): The main function contains logic for switching between manual and automatic modes. It monitors the push button input, updates counters, and calls the appropriate functions to spray at the correct time while maintaining PWM.

```
void start_Timer1(void) {
	set_bit(TCCR1, bit(CS12) | bit(CS11) | bit(CS10)); // PCK/64 (pg. 89)
	OCR1A = 156; // Compare value for ~1 second at 1MHz (pg. 91)
	set_bit(TIMSK, bit(OCIE1A)); // Enable Compare Match Interrupt (pg. 92)
	sei(); // Enable global interrupts
}

ISR(TIMER1_COMPA_vect) {
	TCNT1 = 0;

	if(mode == Auto) {
		frequency_AutoCount++; //100hz
		if(frequency_AutoCount > 99) {
			frequency_AutoCount = 0;
			second_AutoCount++;
			if(second_AutoCount > 59) {
				second_AutoCount = 0;
				minute_AutoCount++;
			}
		}
	} else {
		frequency_ManualCount++;
	}
}
```

start_Timer1(): This function configures Timer1 for precise timing operations. It uses compare match interrupts to count seconds and minutes set to ~100Hz for precise automatic spray intervals without affecting PWM control.

```
void start_Timer0() {
        TCCR0A = 0; //Normal Mode (pg. 79)
        set_bit(TCCR0B, bit(CS01) | bit(CS00));// PCK/64 (pg. 80)
        OCR0A = pulsewidth;
        set_bit (TIMSK, bit (TOIE0) | bit (OCIE0A)); //Enable Overflow and Compare Match Interrupt (pg 81)
        sei(); //Enable global interrupts
}

ISR(TIMER0_COMPA_vect) { //Pulse Width
        clear_bit(PORTB, bit(PB4)); //Clear bit
}

ISR(TIMER0_OVF_vect) {  //16.384ms or 61.035Hz
        set_bit(PORTB, bit(PB4)); //Set bit
}
```

start_Timer0(): This function initializes Timer0 to generate PWM signals for the servo motor. It sets the timer mode, prescaler, and enables interrupts.

```
#define Min 40
#define Max 38

void PW_ramp(void) {
        pulsewidth = Max;
        OCR0A = pulsewidth;
        _delay_ms(500); //Sprays a short burst when triggered (Full Power)
}
```

PW_ramp(): This function adjusts the servo's pulse width to trigger the spray. It ramps the PWM signal to the maximum value for a short burst, then returns it to the minimum to stop the spray.

PWM signal is ON when it's at its Max value. After a short delay, the function resets the pulse width to the Min value, which returns the servo to its idle position. This approach provides precise control over the spray duration and prevents unnecessary strain on the servo by exceeding the maximum mechanical limits.

Summary: (Reflection)

I learned engineering practices while designing this project. For example, I learned to recognize future problems and adjust my design when needed. Importantly, this project taught me how to apply engineering principles to complete a complex design.

I also gained hands-on experience in designing a control system using an ATtiny45 microcontroller, implementing PWM for servo control, and managing timing with interrupts. Additionally, I created a custom PCB and learned how to troubleshoot wiring issues.

What I learned

I developed skills in writing C code with timers and interrupts and learned how to optimize logic for efficiency. I improved my ability to select components by understanding how specifications affect performance. I also learn how to create a working with a bill of materials to help me visualize coss and identify better ways to choose resources for better performances.

What I Can Do Better

One area I could have improved was budgeting. While my focus was on completing the project and ensuring it worked independently, I did not set a clear financial plan. This led to decisions driven by availability rather than cost-efficiency. If I had established a budget early, I could have prioritized materials, reduced unnecessary expenses, and aligned spending with my design goals.

Automated Febreze Project Report

Attached Code:

```
/*
 * Author : Trevin
 */
#include <avr/io.h>
#include <avr/interrupt.h>
#define F_CPU 1E6
#include <util/delay.h>
#define set_bit(reg, mask) (reg = reg | mask)
#define clear_bit(reg, mask) (reg = reg & ~mask)
#define bit_high(reg, mask) (reg & mask)
#define bit_low(reg, mask) (!(reg & mask))
#define bit(num) (1<<num)
/*Notes: Settings
*Manual: Controlled by user
*Auto: Automatic Intervals
*/
#define Manual 0
#define Auto 1
/*Notes: PWM
*Min 2.6ms (Fully extended to the right)
*Max 0.4ms (Fully extended to the left
*Hardware limitations ~22.5 degrees
*37 = 2.368ms (Does Not Pull)
*36 = 2.304ms (Pulls)                    Minimum Power
*35 = 2.240ms (Pulls)
*34 = 2.176ms (Maximum)                  Assume Max Power (0.997 Amps)
*/
#define Min 40      //OFF
#define Max 38      //ON
/*Notes: Variables
*
*/
unsigned char pulsewidth = Min;                  //Defaults Min
volatile unsigned char mode = Auto;              //Defaults Auto mode
//Automode Variables for counting
volatile unsigned char frequency_AutoCount = 0;
volatile unsigned char second_AutoCount = 0;
volatile unsigned char minute_AutoCount = 0;
//Manual Variables for counting
volatile unsigned char frequency_ManualCount = 0;
/*Notes: Timers
*Clock Speed 1Mhz Prescaler 64 (64us per bit)
*Timer0 is for PWM
*Timer1 is for a precise 1-second interrupt
*/
void start_Timer0() {
        TCCR0A = 0; //Normal Mode (pg. 79)
        set_bit(TCCR0B, bit(CS01) | bit(CS00));// PCK/64 (pg. 80)
        OCR0A = pulsewidth;
        set_bit (TIMSK, bit (TOIE0) | bit (OCIE0A)); //Enable Overflow and Compare Match Interrupt (pg 81)
        sei(); //Enable global interrupts
}
ISR(TIMER0_COMPA_vect) { //Pulse Width
        clear_bit(PORTB, bit(PB4)); //Clear bit
}
ISR(TIMER0_OVF_vect) { //16.384ms or 61.035Hz
        set_bit(PORTB, bit(PB4)); //Set bit
}
void start_Timer1(void) {
        set_bit(TCCR1, bit(CS12) | bit(CS11) | bit(CS10)); // PCK/64 (pg. 89)
```

```
	OCR1A = 156; // Compare value for ~1 second at 1MHz (pg. 91)
	set_bit(TIMSK, bit(OCIE1A)); // Enable Compare Match Interrupt (pg. 92)
	sei(); // Enable global interrupts
}
ISR(TIMER1_COMPA_vect) {
	TCNT1 = 0; //Timer1 does not have a CTC Mode, so manually set TCNT1 = 0

	if(mode == Auto) {
		frequency_AutoCount++; //100hz
		if(frequency_AutoCount > 99) {
			frequency_AutoCount = 0;
			second_AutoCount++;
			if(second_AutoCount > 59) {
				second_AutoCount = 0;
				minute_AutoCount++;
			}
		}
	} else {
		frequency_ManualCount++;
	}
}
void configure_IO(void) {
	set_bit(DDRB, bit (PB4)); //PB4 = output
	set_bit(PORTB, bit(PB3)); //PB3 = input PULLUP
}
void PW_ramp(void) {
	pulsewidth = Max;
	OCR0A = pulsewidth;
	_delay_ms(500); //Sprays a short burst when triggered
}
int main (void) {
	configure_IO(); //PB3 = Output & PB4 = Input
	start_Timer0(); //Counts from 0 to 256
	start_Timer1();	//Counts from 0 to 156

	while(1) {
		if(bit_low(PINB, bit(PB3))) {
			mode = Manual;
			if(frequency_ManualCount < 200) {
				PW_ramp();
			} else {
				frequency_ManualCount = 200;
				pulsewidth = Min;
				OCR0A = pulsewidth;
			}
		} else {
			frequency_ManualCount = 0;
			mode = Auto;
			if(second_AutoCount > 9) {
				frequency_AutoCount = 0;
				second_AutoCount = 0;
				minute_AutoCount = 0;
				while (second_AutoCount < 2) {
					PW_ramp();
				}
				pulsewidth = Min;
				OCR0A = pulsewidth;
			} else {
				pulsewidth = Min;
				OCR0A = pulsewidth;
			}
		}
```

```
        }
}
```