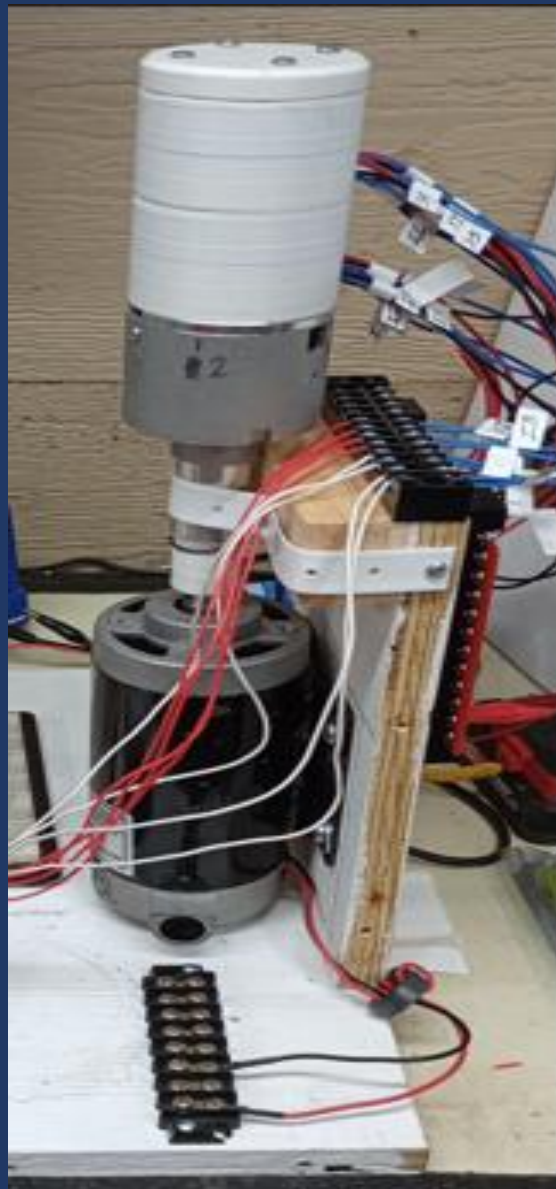


# Stanley Meyer's Laser Distributors

**An electronic analysis with additional mechanical dimensions**



**Ethan Crowder**

## Introduction & Acknowledgements:

Stan Meyer's technology encompasses a multitude of avenues, often overshadowed by the iconic water-powered car. While this vehicle remains a remarkable achievement, it represents only a culmination of numerous separate discoveries made throughout Meyer's extensive research and development career. The focus of this book is to bridge the gap in understanding by providing technical analysis of the electronic circuitry employed by him. This analysis is drawn from his literature and publicly released photographs, offering a detailed examination of the methodologies he described. By delving into these technical aspects, this book aims to shed light on the broader scope of Meyer's work.

The lack of comprehensive technical analysis has led to numerous misconceptions, and a general dismissal of Stan's accomplishments. This deficiency has fostered an environment where the true significance of Stan's work is often misunderstood. Inspired by this critical gap in understanding, this book was written with the intention of preserving the knowledge surrounding Stan's contributions and ensure that Stan's achievements receive the recognition they rightfully deserve.

### Special Thanks:

- 1.) Stanley Allen Meyer, in who's memory this work is dedicated to.
- 2.) Don Gabel who publicly released Stan's estate photographs, providing invaluable evidentiary proof of his technology and generously permitting their inclusion in this work.

### Notes From the Author:

**Disclaimer:** Due to the potentially dangerous nature of this technology, everything contained herein is meant for educational purposes only. The author does not accept any liabilities if replications are pursued.

**Mechanical Dimensions:** Appendix A through H contain dimensions for devices covered herein. The dimensions are based on photographic analysis and limited literature from Stan. While the best effort was made for accuracy, there may be some discrepancies. Unfortunately, original dimensions were not known at the time of this writing. Dimensions were based off replications via additive manufacturing techniques (3D printing). Traditional machining operations/techniques may warrant revision.

**Copyright Registration: FORTHCOMING**

### **Supplemental Additions:**

While this book provides technical analysis of the electronic circuit functionality, it is not all encompassing. Further lab work may be added as a follow up at a later date.

## Table of Contents

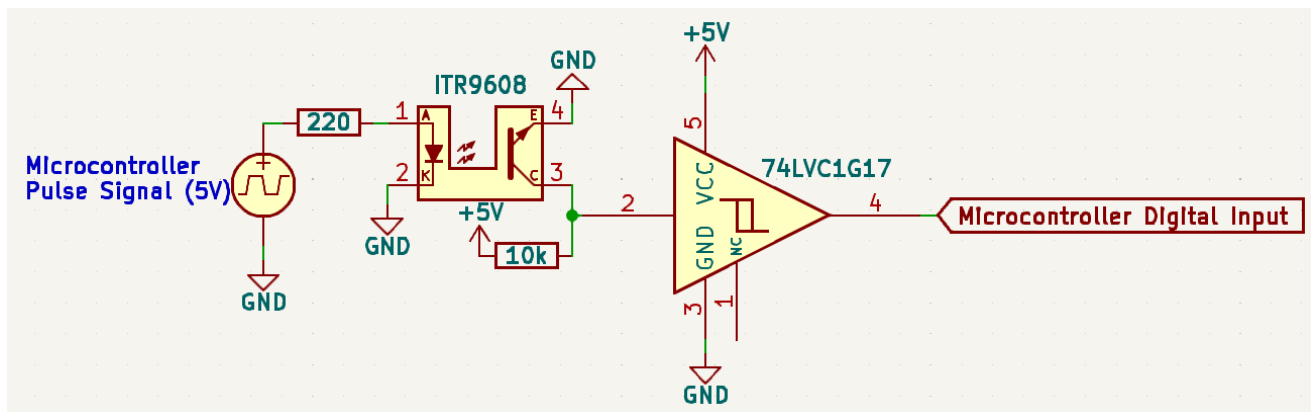
ITR9608 optical-slot switch analysis.....	page 3
Single-stage laser distributor analysis.....	pages 4-5
1600cc engine & distributor cycles.....	page 6
Stan's original PCBs & assumption of operation.....	page 7
Dual-stage laser distributor overview.....	page 8
Location of PCBs on the GMS (Gas Management System).....	page 9
Arduino code review & schematic.....	pages 10-11
PIC18F57Q43 code review / schematic & modernization.....	pages 12-14
Appendix A: Dimensions of replication.....	pages 15-22

## ITR9608 Optical-Slot Switch Sensors

In this replication, ITR96008 optical-slot switch sensors were utilized. As shown in the schematic below, the sensor is composed of an infrared LED (940nm – pin #1) emitter (or transmitter – “Tx”) paired with a receiving photo-transistor (or receiver – “Rx”). When the light beam is uninterrupted, the transistor enters a conducting state (ON). Conversely, if the beam is interrupted, the transistor enters a non-conductive state (OFF). This provides isolation between two circuits. The maximum  $I_F$  of the LED is 60mA. The resistor used in the arrangement is a 220 $\Omega$ -1/4watt, limiting the current to 20mA, which is the maximum current that can be safely delivered by most microcontroller pins. The cathode of the LED (pin #2) is tied to the microcontroller’s ground rail.

To produce a resting logic state, being HIGH, a 10k $\Omega$ -1/4watt resistor is used as a pull-up resistor, being placed in series between +5VDC and the transistor’s cathode (pin #3). An input into a non-inverting Schmitt trigger buffer (74LVC1G17) is connected at the lower side of the 10k $\Omega$ . The transistor’s emitter (pin #4) is tied to ground (0VDC). The Schmitt trigger is used to output a clean pulse signal, eliminating debounce during voltage level transitions. When the light beam is interrupted, the transistor is in a non-conducting state, which causes the pull-up resistor to bring the Schmitt trigger input to a HIGH logic value. When the light beam is not interrupted, the transistor’s photosensitive base reacts, bringing the transistor into a state of conduction (saturation). This drops the voltage of the pull-up resistor to produce a LOW logic value output.

Note: It was found that the infrared beam can pass through PLA plastic. To remedy this, aluminum tape was attached to the outer peripheral surface of each light-gate disk. This is indicative of Stan needing to use a machined metallic disc, which was most likely aluminum. Due to budget constraints, it cannot be said with certainty that denser plastics, such as Delrin – which Stan preferred – wouldn’t provide the same level of beam interruption as a metallic counterpart.



The truth table below illustrates the sequence, along with logical values during beam interruption and non-interruption, that will occur throughout the application within the laser distributors.

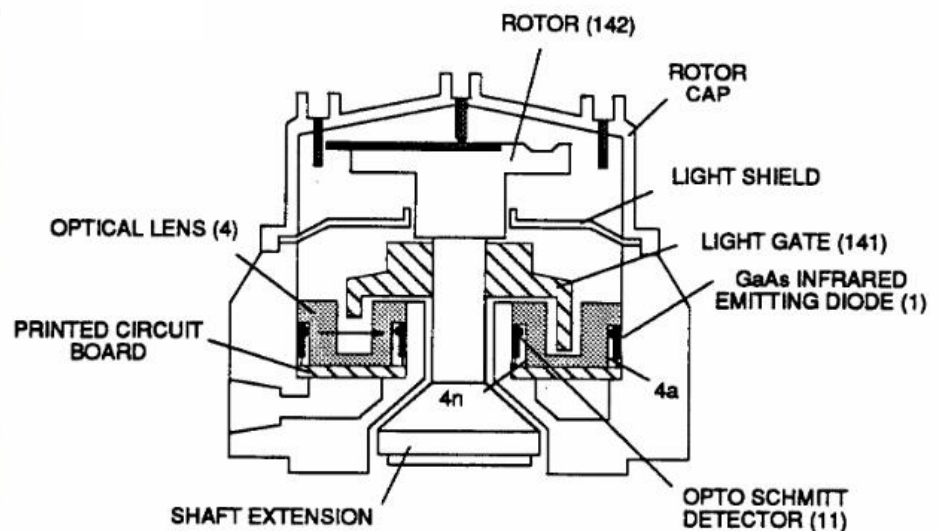
### Logic Table:

LOW	HIGH	HIGH	HIGH
HIGH	LOW	HIGH	HIGH
HIGH	HIGH	LOW	HIGH
HIGH	HIGH	HIGH	LOW

## Single-Stage Laser Distributor

The single-stage laser distributor was utilized with the gaseous fuel injection system. This early injection system utilized Hydrogen and Oxygen in gaseous form, released from the fuel cell, in conjunction with non-combustible gases, such as the recycled exhaust gases, to provide a fuel that performed similarly to gasoline.

Stan's original description from Memo WFC 422 DA, page 3-24: *Laser Distributor assembly functions in similar manner as Laser Accelerator except that light-gate rotates in the same direction of spark-rotor and being displaced opposite to rotor blade, allowing intermixed process ambient air gases and fuel-gases to enter engine cylinder as illustrated in injector control circuit. Rotating light-gate triggering circuit assembly sequentially activates pulse shaping generator to produce a constant 50% duty-cycle pulse train to analog voltage generator of hydrogen gas management system. Interlocking Laser Accelerator output with Laser Distributor output causes fuel-injectors to be "tuned" with both air management system and hydrogen gas control circuit to maintain constant fuel-mixing ratio during engine performance. As Laser Accelerator advanced toward "peak" engine performance, fuel-injectors open gate-time (on-time) increases proportionately, opposite or reverse movement of Laser Accelerator decreases injectors on-time which, in turn, reduces engine speed.*



**FIGURE 3-44: LASER DISTRIBUTOR**  
(Memo WFC 422 DA)

As seen in the pictures above, the distributor has an additional section between the body and rotor cap. Within this section, optical light sensors comprising 940nm LED emitter transmitters that are paired with phototransistors, acting as the receivers, are positioned directly across from each other. Four pairs in total, which correlates with the number of cylinders. While Stan doesn't provide the model of these components, we can assume he used the same as in the laser accelerator, which contained an internal Schmitt trigger for signal transition shaping. The light gate should be machined from a piece of metal with a slot to allow light communication between a single receiver and transmitter, while at the same time, blocking communication between the rest. This keeps a constant logic state of the Tx/Rx pairs, except for the pair being triggered.

The conventional rotor button and rotor cap, along with the standard high-voltage ignition switching arrangement, are positioned above the optical sensor section. While not expressly stated by Stan, the author speculates that the PCB was adjustable within the distributor to provide a mechanical adjustment in conjunction with electronic circuit timing adjustment.

As shown below, this gaseous-fuel injection process utilized machine blocks that delivered the pre-mixed fuel into the cylinder during the intake cycle. The standard spark plug wire can be seen next to the block. While difficult to determine with certainty, the injection process appears to use on-market solenoid valves. Stan may have used Bosch's Jetronic injectors which were originally implemented by VW in 1967. Standard HDPE tubing can be seen, which was utilized as a fuel rail/delivery method. This would indicate the pressure isn't very high.



### **1600cc VW engine cycles:**

The 1600cc VW air-cooled engine had an ignition sequence of cylinder 1, 4, 3, 2. The distributor shaft rotates one revolution for every 2 crankshaft revolutions. Conventional gasoline injection timing is  $\sim 10^0 - 20^\circ$  before the intake valve opens and  $\sim 10^0 - 20^\circ$  after the intake valve closes, depending on the valve overlap of the engine. These degrees are assuming liquid gasoline fuel. Since the fuel is already in a gaseous state, not requiring atomization via orifice diameter and pressurization, it will mix more rapidly. Stan's process modulated the gaseous fuel burn rate to gasoline. However, he would've been able to retard the ignition cycle a few degrees after TDC (top dead center) if desired.

### **Engine cycles with respect to crank angle (approx.)**

STROKE	CRANK ANGLE	VALVE EVENT
Intake	0 - 180 degrees	intake opens
Compression	180 - 360 degrees	both closed
Combustion	360 - 540 degrees	both closed
Exhaust	540 - 720 degrees	exhaust opens



### Engine and cylinder cycles:

Red = Power/Combustion cycle, Green = Intake cycle, Orange = Compression cycle, Blue = Exhaust cycle

Crank Angle (°)	Dist Angle (°)	Cylinder 1	Cylinder 4	Cylinder 3	Cylinder 2
0°	0°	Power	Exhaust	Intake	Compress
180°	90°	Exhaust	Power	Compress	Intake
360°	180°	Intake	Compress	Power	Exhaust
540°	270°	Compress	Intake	Exhaust	Power
720°	360° (0°)	Power	Exhaust	Intake	Compress

Distributor Terminal Orientation	3	2
	4	1

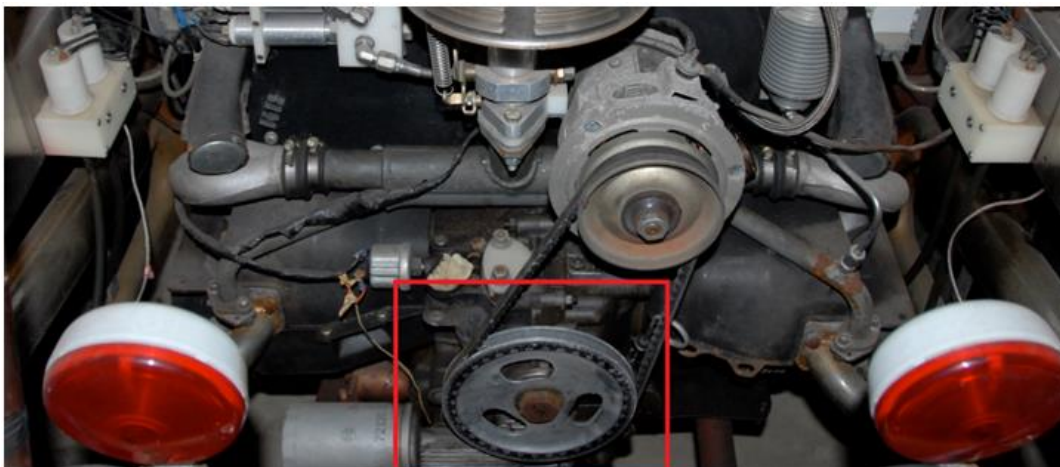
Crank 0	Dist 0	cyl 3	ENGINE	cyl 1
		cyl 4		cyl 2
Crank 180	Dist 90	cyl 3	ENGINE	cyl 1
		cyl 4		cyl 2
Crank 360	Dist 180	cyl 3	ENGINE	cyl 1
		cyl 4		cyl 2
Crank 540	Dist 270	cyl 3	ENGINE	cyl 1
		cyl 4		cyl 2

### Optical sensor triggering cycles:

Purple = Optical sensor pairs triggered via light-gate, Gray = Optical sensor pairs in a logic state that reflects non-triggering.

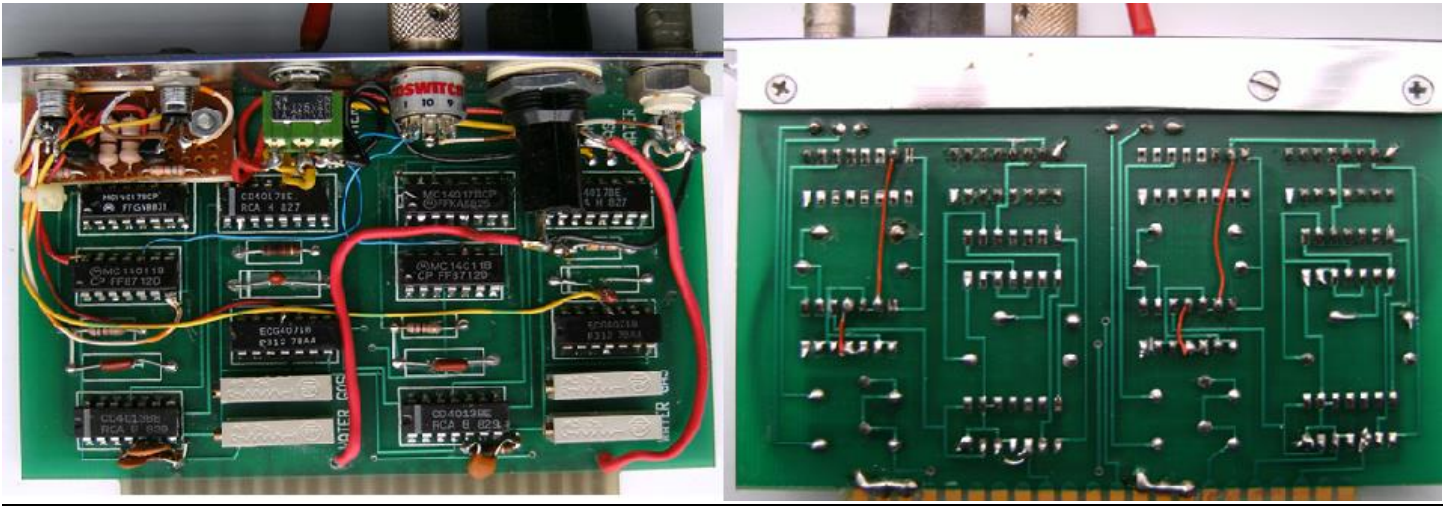
CYCLE 1 Crank at 0/720 degrees			CYCLE 2 Crank at 180 degrees			CYCLE 3 Crank at 360 degrees			CYCLE 4 Crank at 540 degrees		
3	OS2	2	3	OS2	2	3	OS2	2	3	OS2	2
OS3	Shaft CW	OS1	OS3	Shaft CW	OS1	OS3	Shaft CW	OS1	OS3	Shaft CW	OS1
4	OS4	1	4	OS4	1	4	OS4	1	4	OS4	1

The previous diagrams are conceptual only. In reality, the physical placement of the optical sensor disk would need to be optimized. If a fixed 90° offset to the HV terminals is desired, then calculating a delay to achieve a particular degree for injection cycle on the intake stroke would need to be ascertained. Stan's use of an aftermarket crankshaft pulley, as shown in the photo below, with degree increments would've provided angular displacement while observing the displacement of the distributor and associated optical sensors.



## Stan's Original PCBs

PCB top & bottom side (photo courtesy Don Gabel):



### Functional overview:

Each PCB, a total of two, has two individual circuits that, when combined, provide a quad-channel optical ignition pulse processor retrofitted into a VW 1600cc distributor. Each ITR9608 sensor detects the passage of the optical disk's rotation, being mounted on the distributor shaft, outputting an active-low logic-level signal to its own independent logic circuit. Based on electronic IC models and principles, we can make the following assumptions:

The CD4013 flip-flop could be used for edge detection and noise filtering. It should be understood that Stan's original optical sensor receiver had an internal Schmitt trigger.

CD4017 decade counters are typically used for sequential timing or wider pulse generation that could be divided to provide a visually recognized indication via the LED.

CD4011 (NAND) and CD4071 (OR) gates are typically used for logic control and pulse shaping. Various configurations of resistors and capacitors would produce RC time constant-based pulse delays.

Each sensor's output is fed into the D flip-flops (4013). This cleans up the signal, synchronizes it to a local clock (or itself), and prevents false triggering due to jitter or mechanical bounce. The 4017s are used for step sequencing, likely creating a delay or managing output timing. OR gates (4071) aggregate timing states or enable lines. NAND gates (4011) gate the timing signal and perform final pulse logic. It could also be used to produce a pulse extending monostable multivibrator stage. One stage likely functions as a pulse-width limiter (ignition dwell control), shaping the final output. Each output would trigger an individual ignition coil.

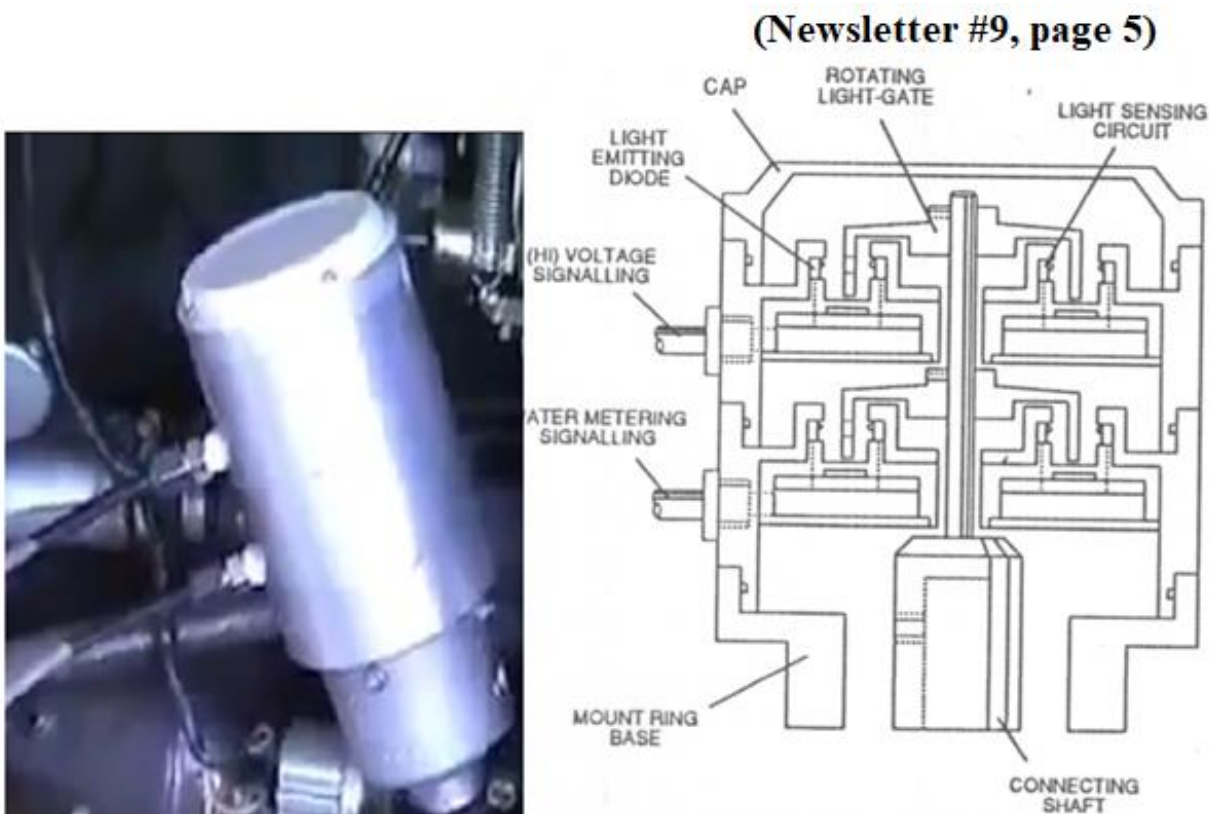
While the circuitry of these PCBs is overall simple in concept, it suggests Stan was investigating the capability of retarding or advancing the ignition timing. Most likely, this was accomplished by pulse extending via NAND gates. Thankfully, modern microcontrollers offer such control with much easier methods. The stock mechanical advance distributor, on the 1600cc VW engine, provides an advancement effect via the weights during higher engine rpm. However, Stan's departure to a completely digital (dual-stage) distributor would've had to mimic this behavior electronically.



## Dual-Stage Laser Distributor

Development of an injection system that utilized liquid water that was split into constituent components – Hydrogen and Oxygen – required greater control. Stan's embodiment of this device is shown below, including a cut-away diagram of the internals. This diagram is from Stan's Newsletter #9, page 5. In a modern comparison, Stan's device operated similar to an optical encoder, having a rotating "light-gate" that triggers a pair of optical sensors.

As seen, the dual-stage distributor has two levels. Each level is composed of four optical sensors, having an infrared LED transmitter (940nm) pair with a phototransistor receiver. One level triggers the water-fuel injectors, and the other level controls the triggering of the high-voltage DC pulse. In accordance with Meyer's literature, this high-voltage pulse performs the fracturing process and ignition simultaneously. These two stages would be offset from one another to perform the cycles shown in the timing chart at the bottom of this page.



### Dual-stage timing chart:

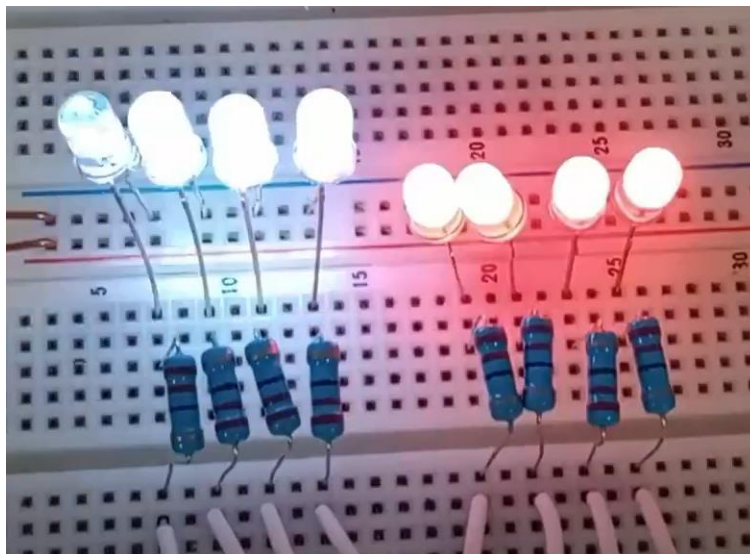
Injection Level - Cycle 1			Injection Level - Cycle 2			Injection Level - Cycle 3			Injection Level - Cycle 4		
3	OS2	2	3	OS2	2	3	OS2	2	3	OS2	2
OS3	Shaft	OS1	OS3	Shaft	OS1	OS3	Shaft	OS1	OS3	Shaft	OS1
4	OS4	1	4	OS4	1	4	OS4	1	4	OS4	1
Ignition Level - Cycle 1			Ignition Level - Cycle 2			Ignition Level - Cycle 3			Ignition Level - Cycle 4		
3	OS2	2	3	OS2	2	3	OS2	2	3	OS2	2
OS3	Shaft	OS1	OS3	Shaft	OS1	OS3	Shaft	OS1	OS3	Shaft	OS1
4	OS4	1	4	OS4	1	4	OS4	1	4	OS4	1

## Gas Management System (GMS) Integration

In a video where Stephen Meyer is demonstrating the functionality of the GMS unit, he points out the injector and distributor states. Below, the injector stages are outlined in red and the distributor stages are outlined in yellow. During the operation, the injector LEDs are sequential (1-2-3-4). Additionally, the distributor board's LEDs move in their respective sequence.



In accordance with the original LED GMS interface panels, different color LEDs were used to provide a visual representation when the outputs were triggered. White indicates the injection cycle, and red indicates the ignition cycle. Current limiting resistors are  $220\Omega$ -1/2watt.



## Microcontroller Programming Review – Arduino

Within the scope of this replication, two different microcontroller platforms were utilized. The first was an Arduino, which was chosen for its popularity with hobbyists and its extensive online community. The second was a PIC18F57Q43 curiosity development board from Microchip. The full code for each are in the appendices. Beginning with the Arduino code, constant integers and regular integer variables are declared:

“*Injector\_X\_Sensor*” references each optical sensor related to injector control. “*Injector\_X\_Solenoid*” references the injector solenoid coils. “*Ignition\_Coil\_X\_Sensor*” references each optical sensor related to ignition control. “*Ignition\_Coil\_X*” references the ignition coils. “*Injector\_Coil\_X\_Sensor\_State*” creates a variable the digital state will be contained. “*Ignition\_Coil\_X\_Sensor\_State*” creates a variable the digital state will be contained.

```
1  const int Injector_One_Sensor = 0;           //Set sensor for injector one LED as DI on pin #0
2  const int Injector_Two_Sensor = 1;           //Set sensor for injector two LED as DI on pin #1
3  const int Injector_Three_Sensor = 2;         //Set sensor for injector three LED as DI on pin #2
4  const int Injector_Four_Sensor = 3;          //Set sensor for injector four LED as DI on pin #3
5  const int Ignition_Coil_One_Sensor = 4;      //Set sensor for ignition coil one LED as DI on pin #4
6  const int Ignition_Coil_Two_Sensor = 5;      //Set sensor for ignition coil two LED as DI on pin #5
7  const int Ignition_Coil_Three_Sensor = 6;    //Set sensor for ignition coil three LED as DI on pin #6
8  const int Ignition_Coil_Four_Sensor = 7;     //Set sensor for ignition coil four LED as DI on pin #7
9  const int Injector_One_Solenoid = 8;         //Set injector solenoid one LED as DO on pin #8
10 const int Injector_Two_Solenoid = 9;         //Set injector solenoid two LED as DO on pin #9
11 const int Injector_Three_Solenoid = 10;       //Set injector solenoid three LED as DO on pin #10
12 const int Injector_Four_Solenoid = 11;       //Set injector solenoid four LED as DO on pin #11
13 const int Ignition_Coil_One = 12;            //Set ignition coil one LED as DO on pin #12
14 const int Ignition_Coil_Two = 13;           //Set ignition coil two LED as DO on pin #13
15 const int Ignition_Coil_Three = 14;         //Set ignition coil three LED as DO on pin #14
16 const int Ignition_Coil_Four = 15;          //Set ignition coil four LED as DO on pin #15
17 int Injector_One_Sensor_State;               //Create an integer variable for logical state checking
18 int Injector_Two_Sensor_State;               //Create an integer variable for logical state checking
19 int Injector_Three_Sensor_State;             //Create an integer variable for logical state checking
20 int Injector_Four_Sensor_State;              //Create an integer variable for logical state checking
21 int Ignition_Coil_One_Sensor_State;          //Create an integer variable for logical state checking
22 int Ignition_Coil_Two_Sensor_State;          //Create an integer variable for logical state checking
23 int Ignition_Coil_Three_Sensor_State;        //Create an integer variable for logical state checking
24 int Ignition_Coil_Four_Sensor_State;         //Create an integer variable for logical state checking
```

Next, the variables are assigned as INPUTS and/or OUTPUTS. The coils are assigned as OUTPUTS, the sensors are assigned as INPUTS.

```
void setup() {
  pinMode(Injector_One_Sensor, INPUT);          //Assign injector one sensor as an input
  pinMode(Injector_Two_Sensor, INPUT);          //Assign injector two sensor as an input
  pinMode(Injector_Three_Sensor, INPUT);        //Assign injector three sensor as an input
  pinMode(Injector_Four_Sensor, INPUT);         //Assign injector four sensor as an input
  pinMode(Ignition_Coil_One_Sensor, INPUT);     //Assign ignition coil one sensor as an input
  pinMode(Ignition_Coil_Two_Sensor, INPUT);     //Assign ignition coil two sensor as an input
  pinMode(Ignition_Coil_Three_Sensor, INPUT);   //Assign ignition coil three sensor as an input
  pinMode(Ignition_Coil_Four_Sensor, INPUT);    //Assign ignition coil four sensor as an input
  pinMode(Injector_One_Solenoid, OUTPUT);       //Assign injector one solenoid as an output
  pinMode(Injector_Two_Solenoid, OUTPUT);       //Assign injector two solenoid as an output
  pinMode(Injector_Three_Solenoid, OUTPUT);     //Assign injector three solenoid as an output
  pinMode(Injector_Four_Solenoid, OUTPUT);      //Assign injector four solenoid as an output
  pinMode(Ignition_Coil_One, OUTPUT);           //Assign ignition coil one as an output
  pinMode(Ignition_Coil_Two, OUTPUT);           //Assign ignition coil two as an output
  pinMode(Ignition_Coil_Three, OUTPUT);         //Assign ignition coil three as an output
  pinMode(Ignition_Coil_Four, OUTPUT);          //Assign ignition coil four as an output
}
```

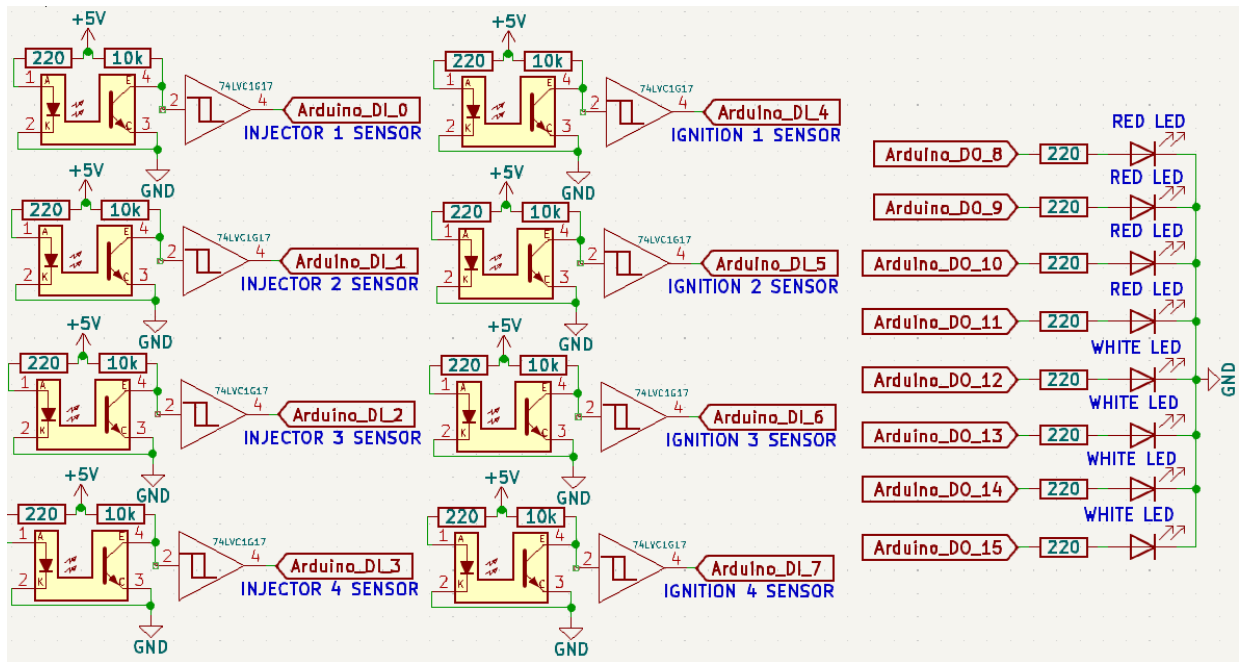
Within the main loop, IF/ELSE statements were utilized. This was done for simplicity, where more efficient methods could've been used. For example, employing a switch-case structure. Each injector sensor state is read by the “*digitalRead*” function and stored in the *Injector\_One\_Sensor\_State*. Depending on the value, the IF/ELSE will determine whether the output is triggered or not. In the code excerpt below, the sensor state is checked to see if it is LOW. If the sensor is LOW, this turns on the output assigns as *Injector\_One\_Solenoid*. In this scenario, the solenoid was represented by an LED. Next, the ELSE statement evaluates and executes the cases in which the criteria of the IF statement are false, meaning if the sensor state is not LOW, but HIGH, the output is turned off.

```
void loop() {
  Injector_One_Sensor_State = digitalRead(Injector_One_Sensor);
  if (Injector_One_Sensor_State == LOW) {
    digitalWrite(Injector_One_Solenoid, HIGH);
  }
  else {
    digitalWrite(Injector_One_Solenoid, LOW);
  }
}
```

Just like the injector sensor states, the ignition states are evaluated and executed in the same manner, as shown below.

```
Ignition_Coil_One_Sensor_State = digitalRead(Ignition_Coil_One_Sensor);
if (Ignition_Coil_One_Sensor_State == LOW) {
  digitalWrite(Ignition_Coil_One, HIGH);
}
else {
  digitalWrite(Ignition_Coil_One, LOW);
}
```

**Schematic:**





# Microcontroller Programming Review – PIC18F57Q43

Unlike Arduino’s abstracted functions such as “*digitalWrite()*” and “*delay()*”, programming PIC microcontrollers in MPLAB requires direct interaction with hardware registers. This allows for greater control and efficiency, but also demands a deeper understanding of the microcontroller's architecture. Microchip’s extensive, in-depth datasheets cover the memory mapping to configure I/O pins, timers, and peripherals manually. While this adds complexity, it also opens the door to highly optimized and deterministic code—ideal for time-critical applications, such as engine control management systems. From simplicity, the code that will be discussed herein is based on polling. In reality, the use of hardware interrupts would need to be incorporated, among other methods. Furthermore, the MPLAB development software provides powerful tools for development and debugging, including the MPLAB Code Configurator (MCC), which can generate boilerplate code for peripherals, along with a data visualizer, which allows for real-time monitoring of internal variables and performance.

At the beginning of every program, headers must be included to defines all the registers, bits, and configuration settings for a specific microcontroller.

```
#include <xc.h> //(Includes the core device-specific header file for the PIC18 device)
#include <stdint.h> //(Standard headers for fixed-width integer types.)
#include <stdbool.h> //(Standard headers for boolean support.)
```

Configuration of the core behavior is accomplished by using compiler directives “pragma config” for hardware such as internal and external oscillators (clocks), watchdog timer, code protection, brown out settings, etc.

```
#pragma config FEXTOSC = OFF, RSTOSC = HFINT32, CLKOUTEN = OFF
//FEXTOSC = OFF means no external oscillator is being used
//RSTOSC = HFINT32 means using the 32MHz internal oscillator at startup
//CLKOUTEN = OFF disables clock output on CLKOUT pin
#pragma config MCLRE = INTMCLR, WDTE = OFF
//MCLRE = INTMCLR means master clear (MCLR) pin is used as a digital input, not for resetting
//WDTE = oFF means the watchdog timer is disabled.
```

Configuring the oscillator frequency. Max internal oscillator setting for the PIC18F57Q43 can be 64MHz.

```
#define _XTAL_FREQ 32000000
//This macro defines the oscillator frequency (32MHz) for use by __delay_ms and __delay_us macros)
```

Configuring macros for injector and ignition ITR9608 optical sensors that will be digital inputs to the bits (0-7) on Port A. Macros are text replacements in the preprocessor stage. They are not data structures or instances and they do not take up any memory.

```
#define INJ_SENSOR_0 PORTAbits.RA0 //Assigning injector 1 ITR9608 sensor to PortA, bit0
#define INJ_SENSOR_1 PORTAbits.RA1 //Assigning injector 2 ITR9608 sensor to PortA, bit1
#define INJ_SENSOR_2 PORTAbits.RA2 //Assigning injector 3 ITR9608 sensor to PortA, bit2
#define INJ_SENSOR_3 PORTAbits.RA3 //Assigning injector 4 ITR9608 sensor to PortA, bit3
#define IGN_SENSOR_0 PORTAbits.RA4 //Assigning ignition 1 ITR9608 sensor to PortA, bit4
#define IGN_SENSOR_1 PORTAbits.RA5 //Assigning ignition 2 ITR9608 sensor to PortA, bit5
#define IGN_SENSOR_2 PORTAbits.RA6 //Assigning ignition 3 ITR9608 sensor to PortA, bit6
#define IGN_SENSOR_3 PORTAbits.RA7 //Assigning ignition 4 ITR9608 sensor to PortA, bit7
```



Configure macros for injector and ignition coils that will be digital outputs to bits (0-7) on Port B. In MPLAB, the LAT, which is short for LATCH, allows output bits to be latched ON or OFF.

```
#define INJ_OUT_0    LATBbits.LATB0 //Assigning injector 1 solenoid coil to PortB, bit0
#define INJ_OUT_1    LATBbits.LATB1 //Assigning injector 2 solenoid coil to PortB, bit1
#define INJ_OUT_2    LATBbits.LATB2 //Assigning injector 3 solenoid coil to PortB, bit2
#define INJ_OUT_3    LATBbits.LATB3 //Assigning injector 4 solenoid coil to PortB, bit3
#define IGN_OUT_0    LATBbits.LATB4 //Assigning ignition 1 coil to PortB, bit4
#define IGN_OUT_1    LATBbits.LATB5 //Assigning ignition 2 coil to PortB, bit5
#define IGN_OUT_2    LATBbits.LATB6 //Assigning ignition 3 coil to PortB, bit6
#define IGN_OUT_3    LATBbits.LATB7 //Assigning ignition 4 coil to PortB, bit7
```

Next, initialization function for the IO (Inputs & Outputs) assigns Port bits as inputs or outputs. In MPLAB, this is accomplished by assigning a logical value 1 for inputs or a logical value 0 for outputs. Each bit may be individually assigned, which typically is used if there's a mix of input & outputs on the same port. In this case, all bits on Port A are assigned as inputs (logical value 1). An example of assigning an entire port as outputs (logical value 0) is shown for Port B, having "TRISB = 0x00". TRIS stands for Tri-State, meaning that the port bits can have three different states: LOW, HIGH, and high impedance. LATB is assigning all output pins an initial value of 0 (LOW). Alternatively, port A could've had all bits assigned as inputs by using "TRISA = 0xFF" – 0xFF being hex for all 1.

```
void init_io(void) {
    TRISAbits.TRISA0 = 1; //Set Port A, bit0 as an input by assigning a logical value of 1
    TRISAbits.TRISA1 = 1; //Set Port A, bit1 as an input by assigning a logical value of 1
    TRISAbits.TRISA2 = 1; //Set Port A, bit2 as an input by assigning a logical value of 1
    TRISAbits.TRISA3 = 1; //Set Port A, bit3 as an input by assigning a logical value of 1
    TRISAbits.TRISA4 = 1; //Set Port A, bit4 as an input by assigning a logical value of 1
    TRISAbits.TRISA5 = 1; //Set Port A, bit5 as an input by assigning a logical value of 1
    TRISAbits.TRISA6 = 1; //Set Port A, bit6 as an input by assigning a logical value of 1
    TRISAbits.TRISA7 = 1; //Set Port A, bit7 as an input by assigning a logical value of 1
    TRISB = 0x00; // Set RB0-RB7 as outputs
    LATB = 0x00; // Initialize all output pins to 0
}
```

The code block below contains the main WHILE loop, which continuously checks (or polls) the state of each pin. To simplify the code, a ternary operator is implemented via the "?". A ternary operator is a shorthand way of writing an if-else statement in a single line. It's called ternary because it takes three operands.

```
void main(void) {
    init_io();

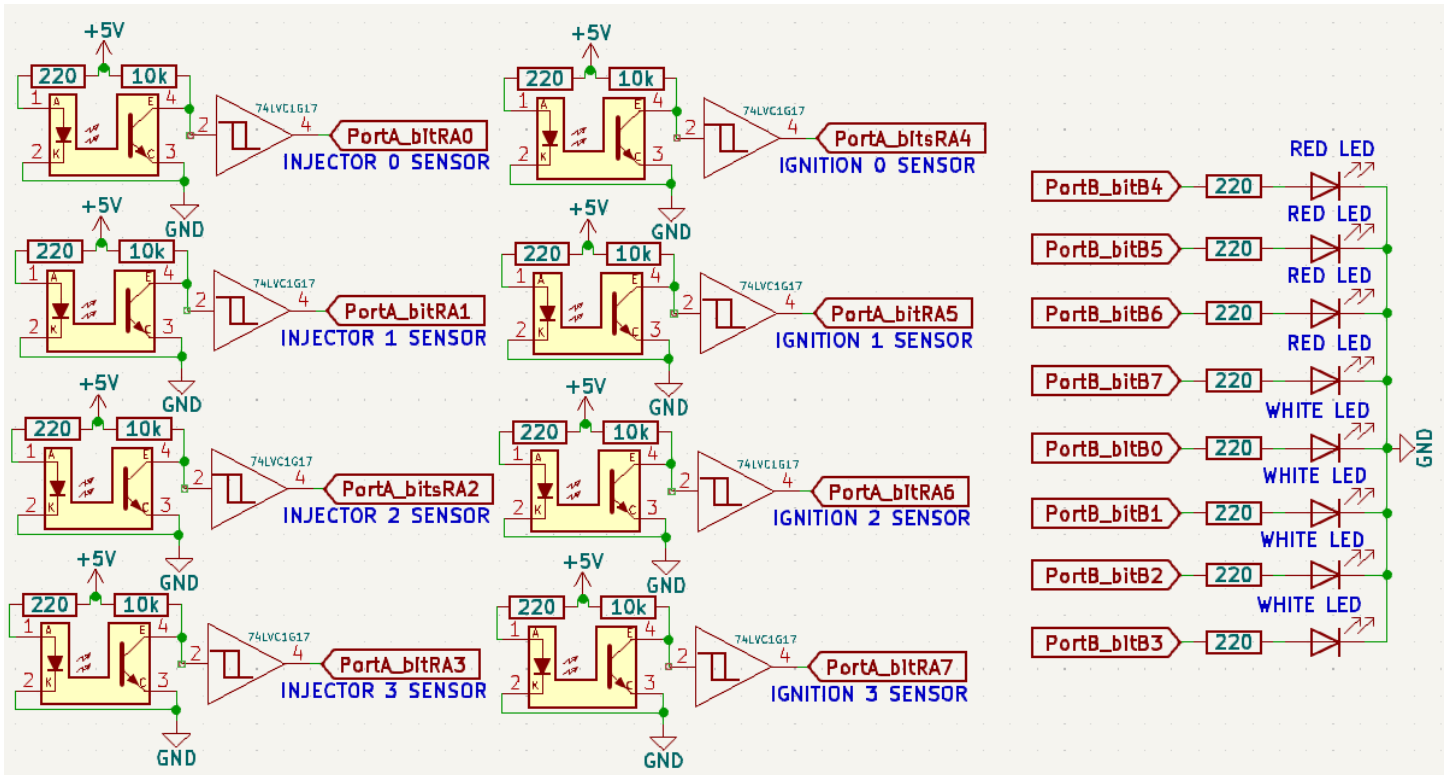
    while (1) { //Continuously checks the state of each pin in a polling fashion

        INJ_OUT_0 = (INJ_SENSOR_0 == 0) ? 1 : 0;
        //Ternary operator [?] meaning: If INJ_SENSOR_0 is 0 (ACTIVE LOW), then set INJ_OUT_0 to 1 (ON), otherwise set it to 0 (OFF)
        INJ_OUT_1 = (INJ_SENSOR_1 == 0) ? 1 : 0;
        //Ternary operator [?] meaning: If INJ_SENSOR_1 is 0 (ACTIVE LOW), then set INJ_OUT_1 to 1 (ON), otherwise set it to 0 (OFF)
        INJ_OUT_2 = (INJ_SENSOR_2 == 0) ? 1 : 0;
        //Ternary operator [?] meaning: If INJ_SENSOR_2 is 0 (ACTIVE LOW), then set INJ_OUT_2 to 1 (ON), otherwise set it to 0 (OFF)
        INJ_OUT_3 = (INJ_SENSOR_3 == 0) ? 1 : 0;
        //Ternary operator [?] meaning: If INJ_SENSOR_3 is 0 (ACTIVE LOW), then set INJ_OUT_3 to 1 (ON), otherwise set it to 0 (OFF)

        IGN_OUT_0 = (IGN_SENSOR_0 == 0) ? 1 : 0;
        //Ternary operator [?] meaning: If IGN_SENSOR_0 is 0 (ACTIVE LOW), then set IGN_OUT_0 to 1 (ON), otherwise set it to 0 (OFF)
        IGN_OUT_1 = (IGN_SENSOR_1 == 0) ? 1 : 0;
        //Ternary operator [?] meaning: If IGN_SENSOR_1 is 0 (ACTIVE LOW), then set IGN_OUT_1 to 1 (ON), otherwise set it to 0 (OFF)
        IGN_OUT_2 = (IGN_SENSOR_2 == 0) ? 1 : 0;
        //Ternary operator [?] meaning: If IGN_SENSOR_2 is 0 (ACTIVE LOW), then set IGN_OUT_2 to 1 (ON), otherwise set it to 0 (OFF)
        IGN_OUT_3 = (IGN_SENSOR_3 == 0) ? 1 : 0;
        //Ternary operator [?] meaning: If IGN_SENSOR_3 is 0 (ACTIVE LOW), then set IGN_OUT_3 to 1 (ON), otherwise set it to 0 (OFF)

        __delay_ms(1); // Optional debounce or timing buffer
    }
}
```

## Schematic:



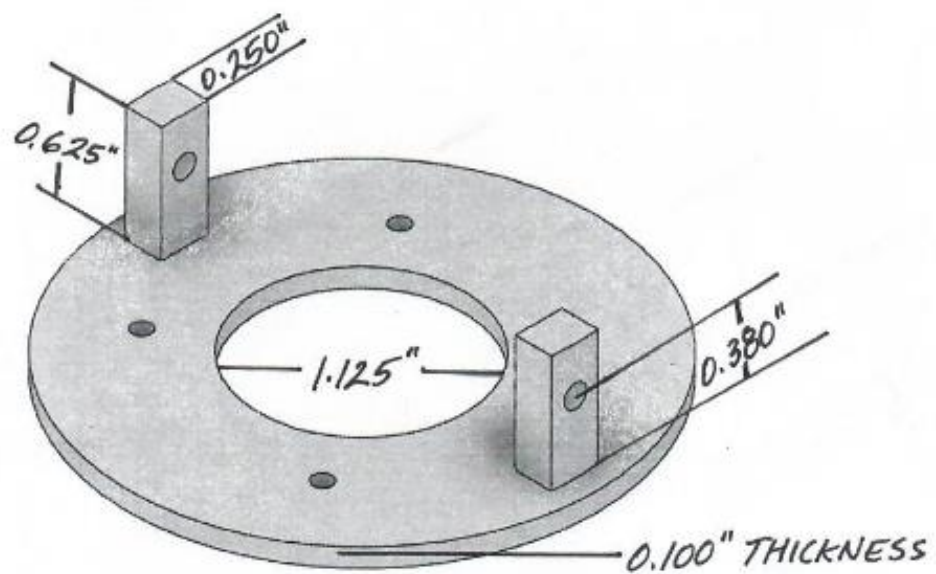
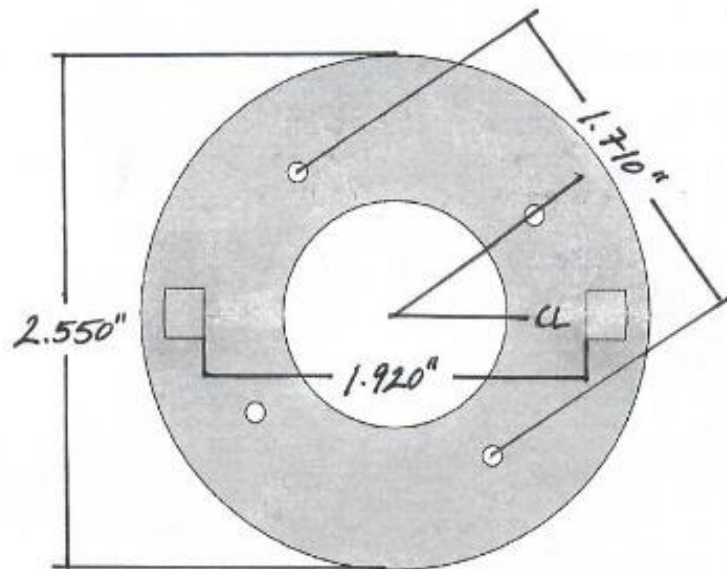
## Modernization

While Stan's circuitry was in accordance with acceptable methods for engine ignition and injection management, he would've needed to develop a strategy to retrofit vehicles that were eliminating the use of distributors. One of the easiest methods, which is also popular with open-source platform engine control units (ECUs), uses a hall sensor in conjunction with a toothed timing wheel. In these arrangements, the hall sensor acts as an inductive-type encoder, which, when used with a hardware interrupt, allows precise counting increments that correlate with the crankshaft angular displacement. This provides a time base to trigger both injection and ignition cycles. An example is shown below in the animated picture:

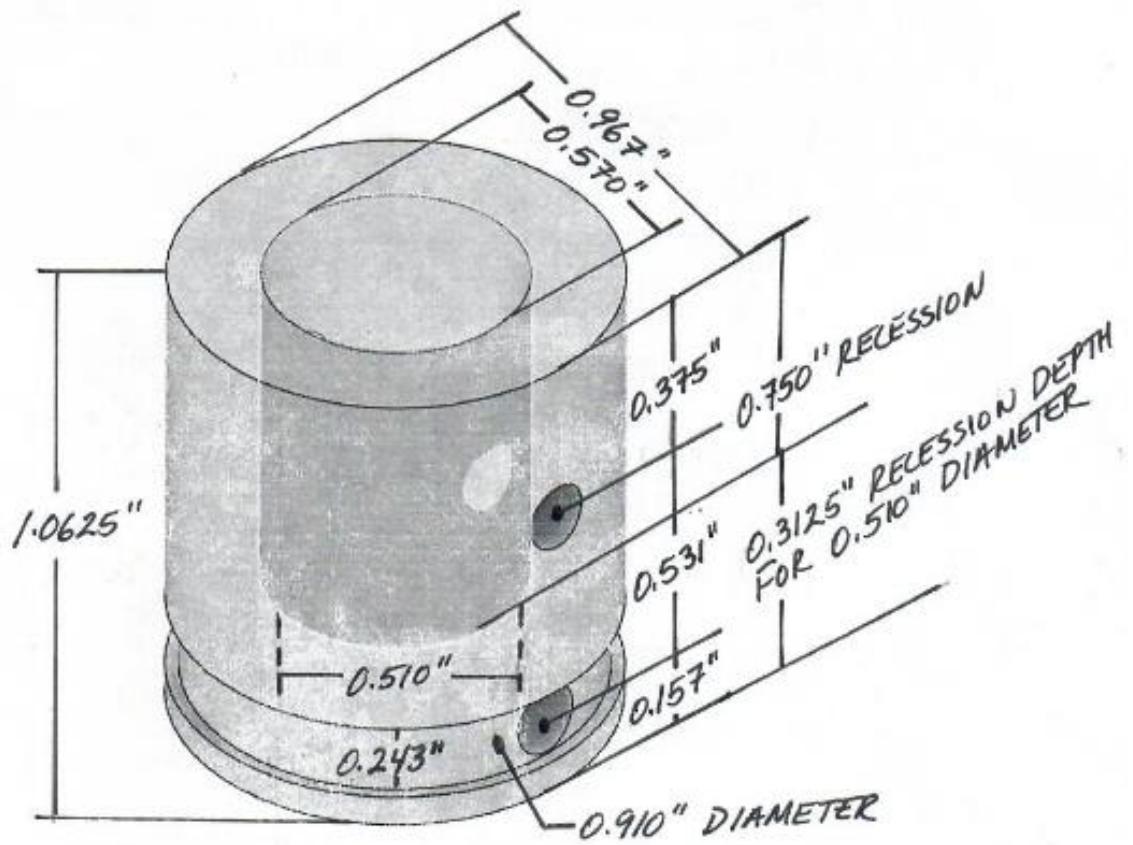
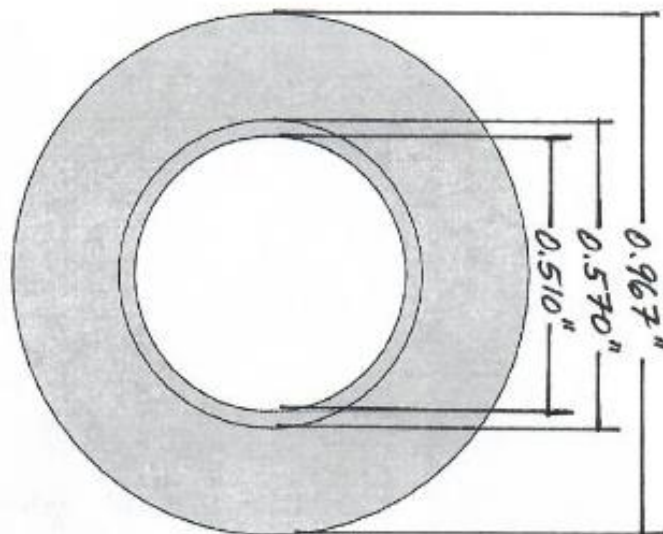


## Appendix A: Dimensions of the replication:

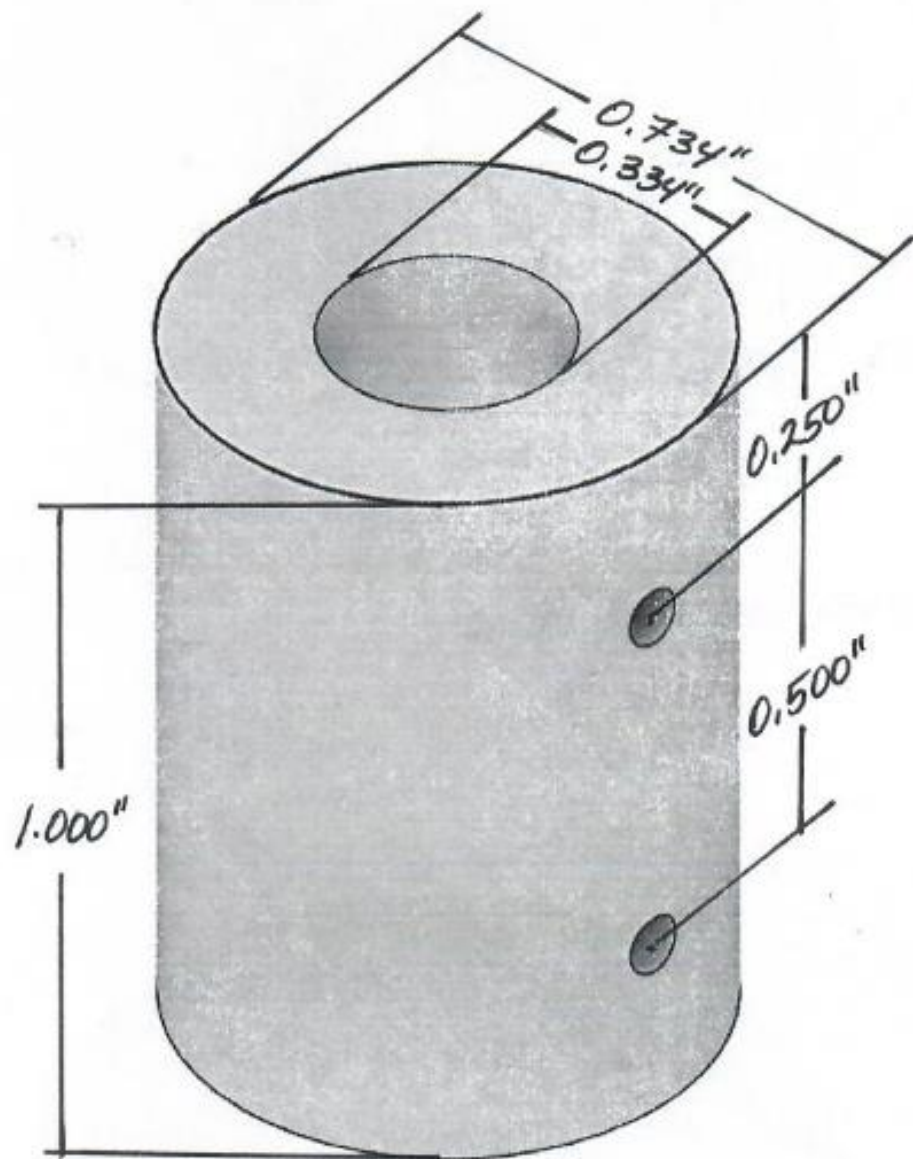
Bottom Mounting Plate



Drive Motor / Alternator Drive Shaft Coupler

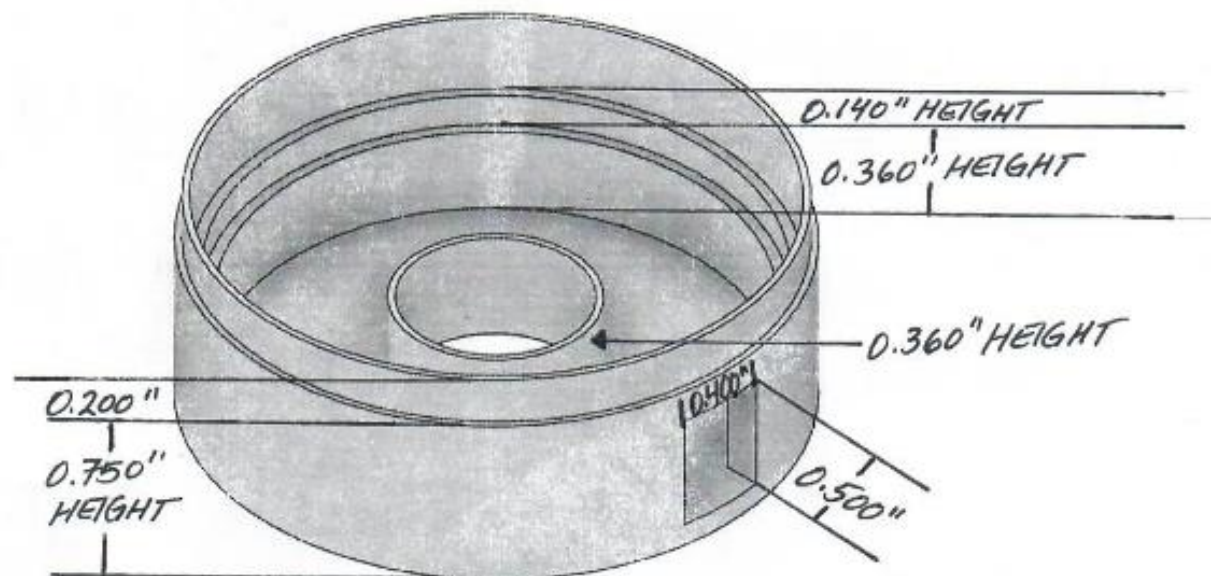
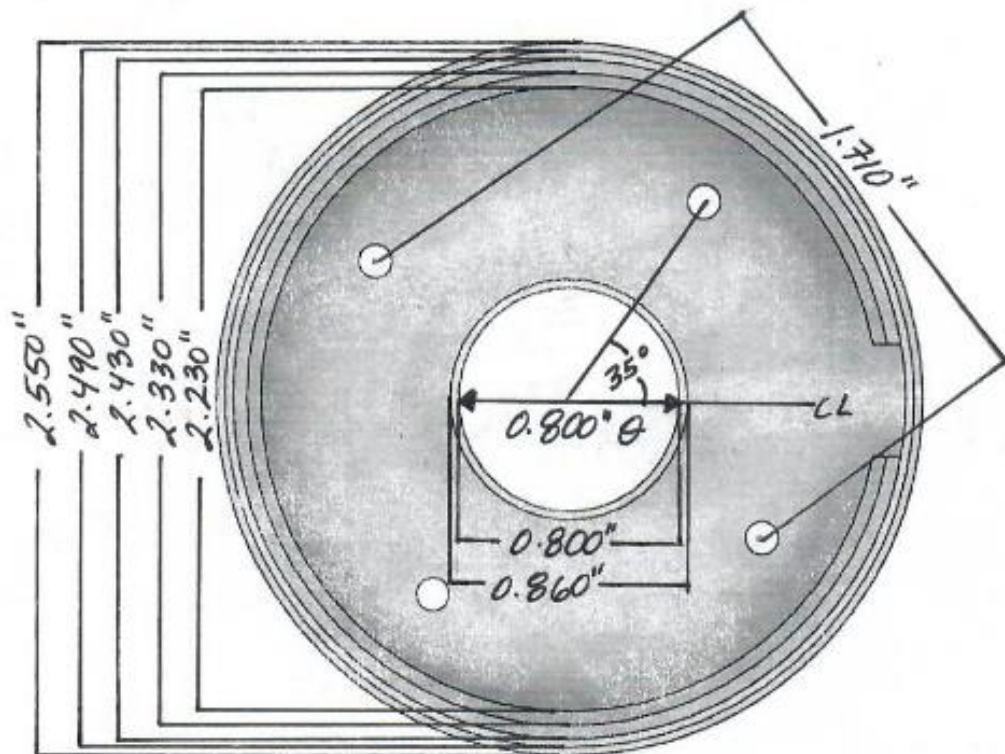


Internal (5/16" / 8mm) Shaft Coupler:

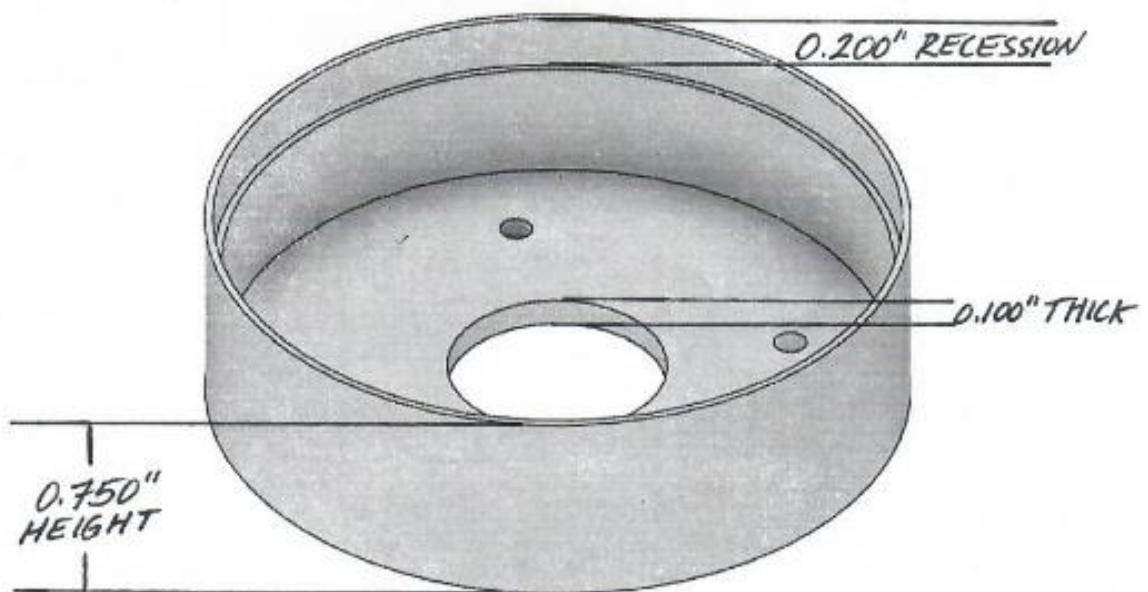
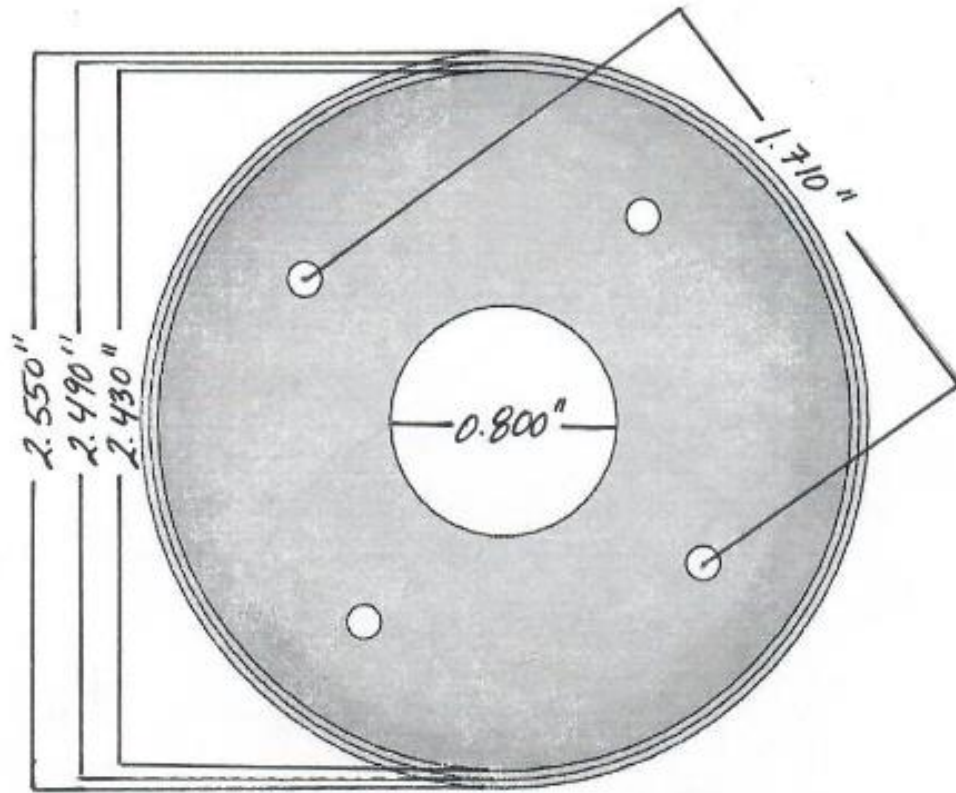




Part A - Midsection

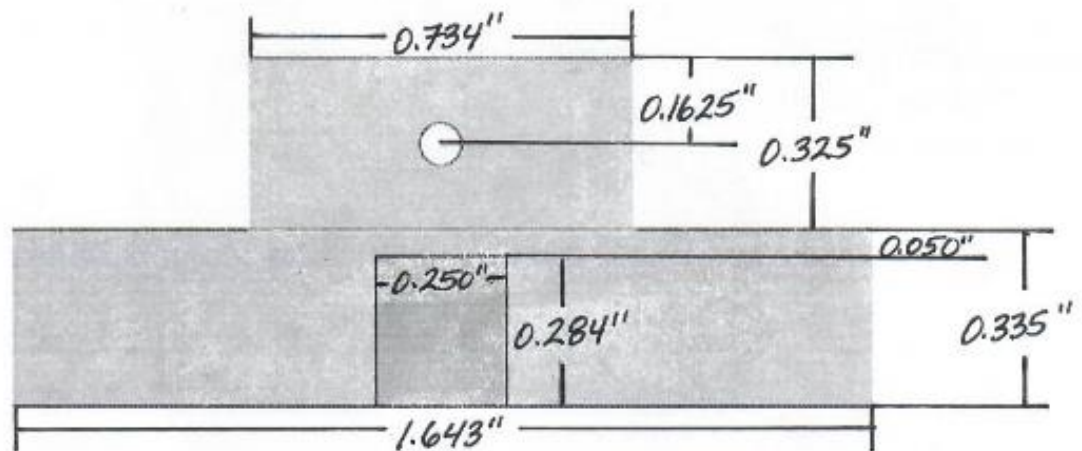
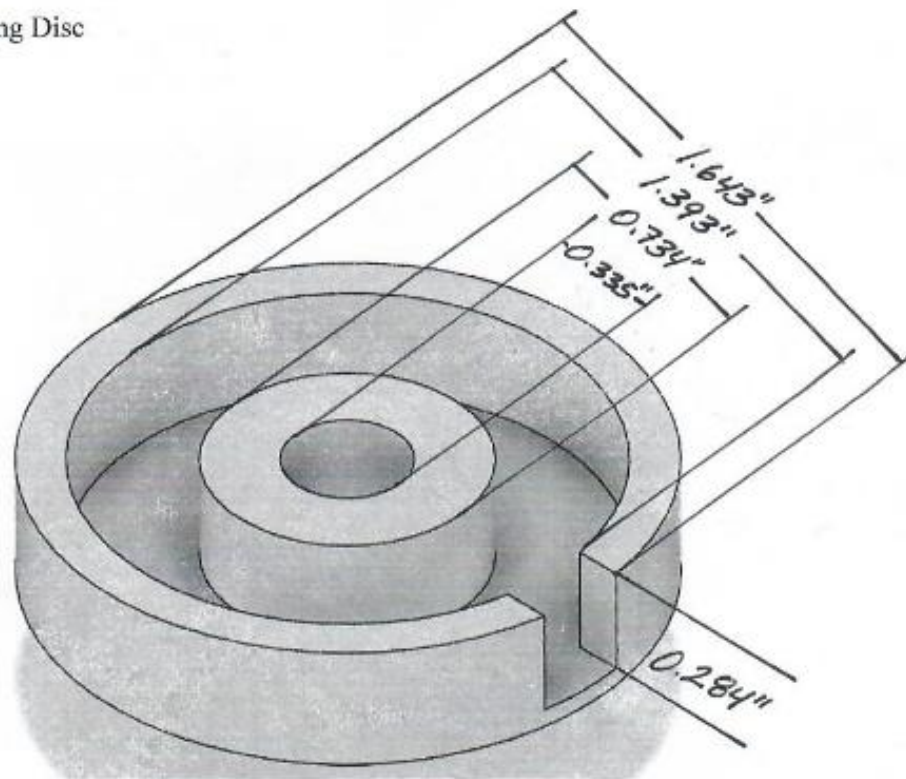


Part B – Midsection



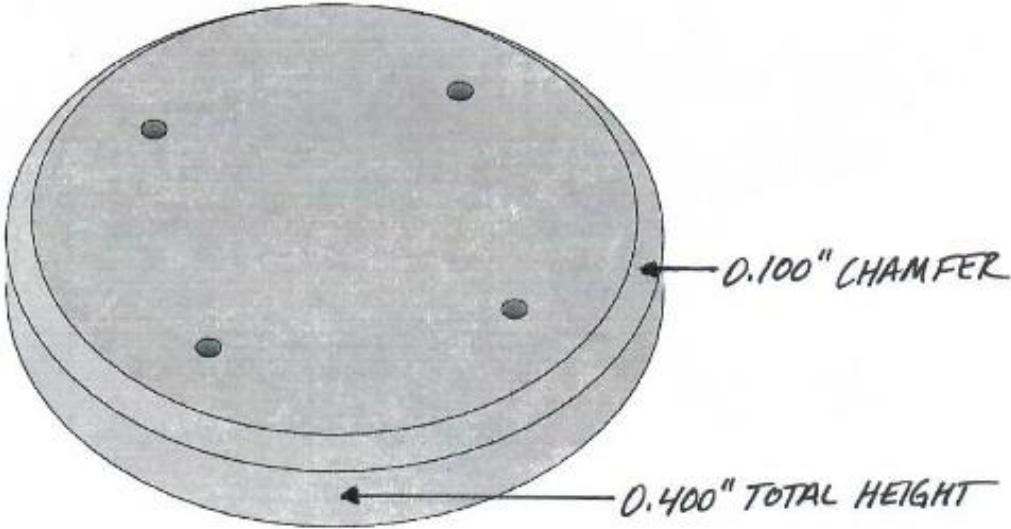
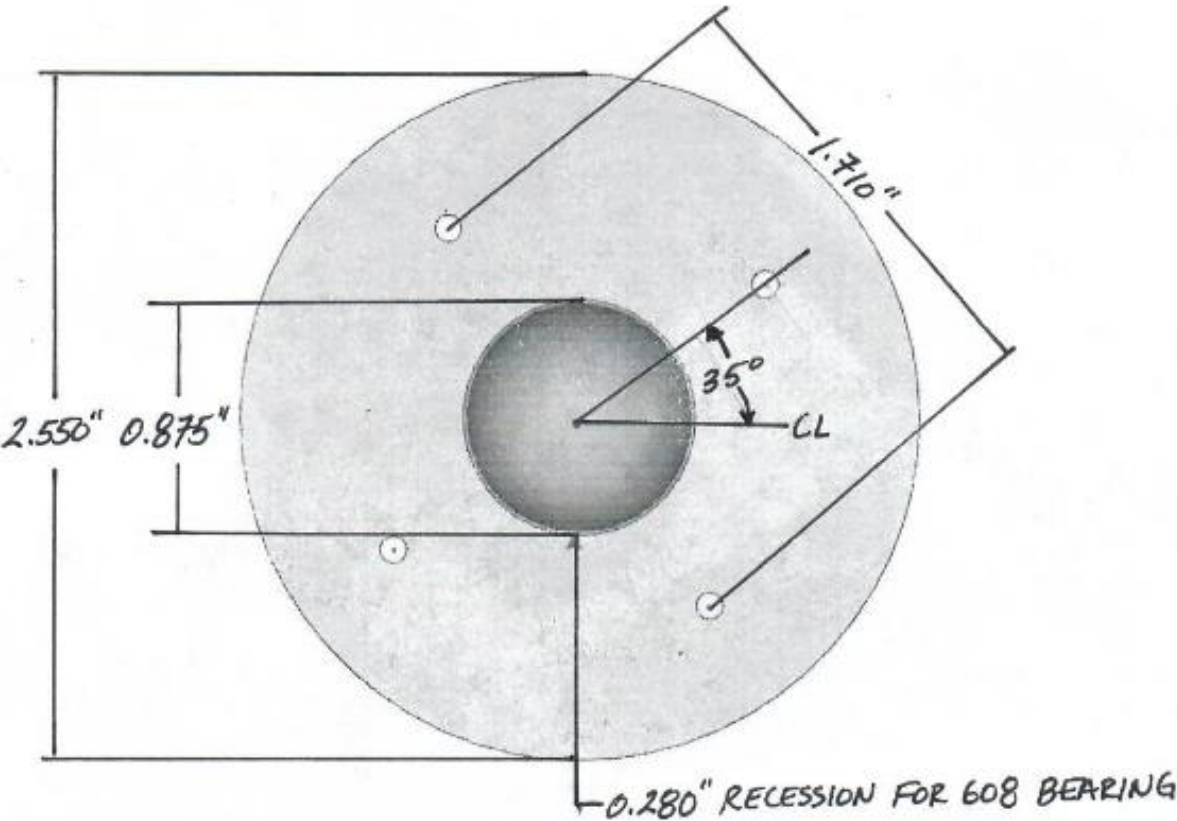


Optical Slot-Switch Rotating Disc





Top Cap:





### **About the Author:**



### **About the author:**

Ethan Crowder has a background in Electromechanical Engineering Technology, with diverse skills in CAD modeling, additive manufacturing design, PCB layout, and electronic analysis. Over the course of his career, Ethan has studied the work of numerous influential inventors in the field of electronic/electrical technology. Driven by a passion for innovation and collaboration, Ethan's work revolves around open-sourcing alternative energy and agricultural technologies. His aim is to empower others by providing accessible solutions that promote environmental consciousness and drive the development of technological independence