MSP430 Breathalyzer Photo Booth

Kelton Whiteaker

April 8, 2018

Abstract

A rudimentary breathalyzer was made using an alcohol gas sensor, the Texas Instruments MSP430 microcontroller and a four-digit 7-segment display. The real-time BAC of the user was displayed on the four-digit 7-segment display and the microcontroller was connected to a digital camera, taking a picture if the user's BAC was above a certain threshold.

1 Introduction

This project is, in a way, inspired by various instructables on building motion-detector camera traps [8] [10]. However, given that alcohol sensors were inexpensive and widely available from sites such as Sparkfun or Seeed Studio, the idea of an alcohol-triggered camera was derived naturally. Breathalyzers themselves are undoubtedly a very important piece of technology, and familiarity with their inner workings, and the limits of their accuracy, can be useful in unexpected places [2]. A breathalyzer that takes pictures at a certain blood alcohol concentration ("BAC") can in and of itself be a fun device to have, but its functionality can also be extended to many practical uses like facilitating traffic stops or restricting admission into concert venues.

This paper describes the process of making the breathalyzer camera, then presents ways in which the final product could be improved, as well as possible improvements upon the design of the device itself.

2 Apparatus

The main components of the device are: MQ-3 gas sensor, piezoelectric beeper, 7-segment display, Vivitar Mini Digital Camera. A full list of required materials follows:

- breadboard with power source and 7-segment display
- 3x 1kΩ resistors
- two 2N4123 NPN transistors

- Vivitar Mini Digital Camera
- MQ-3 gas sensor
- piezoelectric beeper
- insulated wires
- soldering iron, solder
- electrical tape (for reassembling camera)
- tiny Philips-head screwdriver

Figure 1 shows the components of the device and the connections between them. A more detailed circuit diagram can be seen in Figure 2. A description of the calibration and implementation of each component follows.

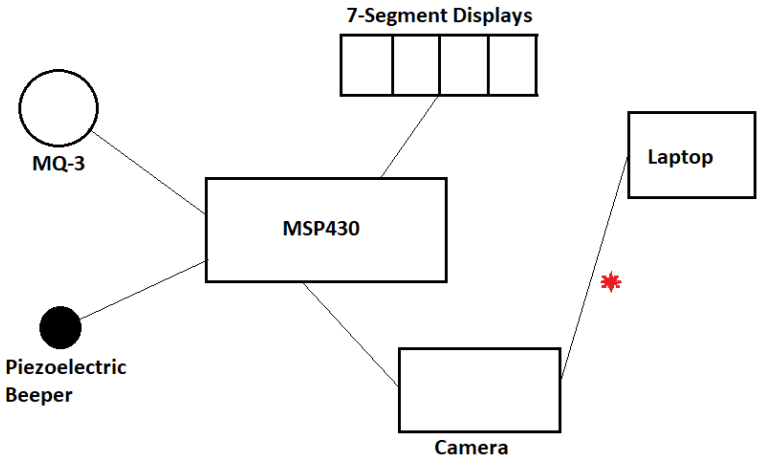


Figure 1: High-level diagram of circuit components; note that connecting the camera to the laptop should only be done when the device is not in use.

2.1 MQ-3 Gas Sensor

The MQ-3 is a resistive sensor; it decreases its internal resistance in response to increased alcohol gas levels. According to its data sheet [5], the internal resistance of the device ranges from 1MΩ to 8MΩ. Figure 3 shows the pin connections of the MQ-3 (H pins power the heater and the internal variable resistance runs from pin A to pin B), as well as its recommended test circuit with a load resistance R_L=200kΩ. Michael from Nootropicdesign [7] recommends a load resistance of 10kΩ to lower and widen the output voltage, but in the end the lowest baseline ADC readings were received at 1.090kΩ; at 10kΩ, ADC

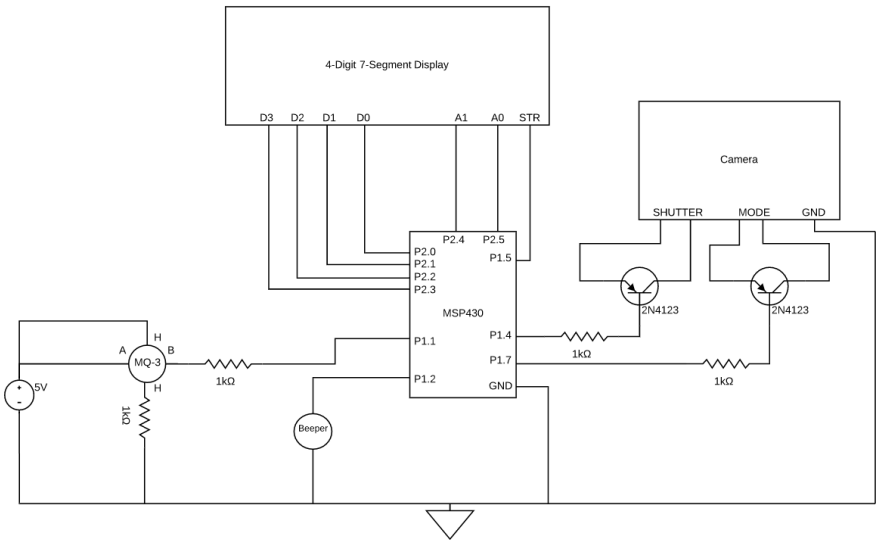


Figure 2: Full circuit diagram of device.

readings were at a baseline of 600. It's possible that my MQ-3 is different from the one used on Nootropicdesign as multiple datasheets exist, all with different interal sensing resistances. It should be noted that there is a 24- to 48-hour warmup period during which the sensor does not return stable readings [5].

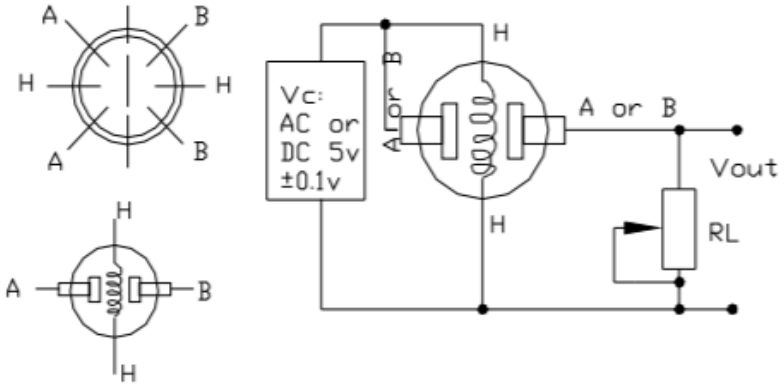


Figure 3: Recommmended test circuit for MQ-3.

In order to calibrate the MQ-3 to properly recognize a 0.08 BAC, the

datasheet recommends exposing it to a gas of 0.4mg/L alcohol concentration, which by the standard law enforcement conversion, % BAC = breath alcohol concentration [mg/L] × 0.21, corresponds to a BAC of roughly 0.08 [7]. An environment with an alcohol concentration of 0.4mg/L was created using Henry's law on the vapour concentration of a volatile (ie. creates vapour) solute in solvent [6]:

$$k_H^{cc} = C_{aq}/C_g \text{ where } k_H^{cc} = \frac{T}{12.2} \times k_H$$

Here, $k_H = 140$ M/atm for isopropanol (the alcohol used for testing), T is room temperature, C_{aq} is the concentration of the aqueous solution and C_g is the gas concentration [12]. Note that there are many forms for Henry's Law, each with their own constant (for example, k_H is the constant of the relation between aqueous concentration and gas *pressure*), so it's important to ensure the correct one is being used.

Once 0.4mg isopropanol/L was converted to mol isopropanol/L (molar mass of isopropanol obtained from the PubChem database [9]), Henry's law was used to find the air concentration in M, which was then converted to mL isopropanol/L (density of isopropanol again obtained from the PubChem database [9]). The final result for concentration was 1.15mL/L. Interestingly enough, the same result was obtained within 2 decimal places using just k_H and the ideal gas law.

Using this known solution, the ADC value corresponding to a 0.08% BAC and to a 0% BAC can be obtained, and then a linear relation between ADC and BAC can be constructed. In order to find the corresponding ADC value, made a program was written that sends the current ADC value to the computer in real-time, the code for which can be found in Appendix Section 2.

I found that the baseline ADC reading was roughly 100, and the 0.08% BAC reading was roughly 320 (I will speak more on the precision of the device in Section 4; the uncertainty in the readings was definitely disappointing considering the effort put into making a solution with a known gas concentration).

The final relation obtained between ADC and BAC was $ADC = 2750 \times BAC + 100$, leading us to the following conversion from ADC to BAC:

$$BAC = (ADC - 100)/2750$$

This was changed into an integer, so that my DisplayNum() subroutine (see Section 2.3) could display it, by multiplying it by 1000. Thus a display of 0080 corresponded to a BAC of 0.08

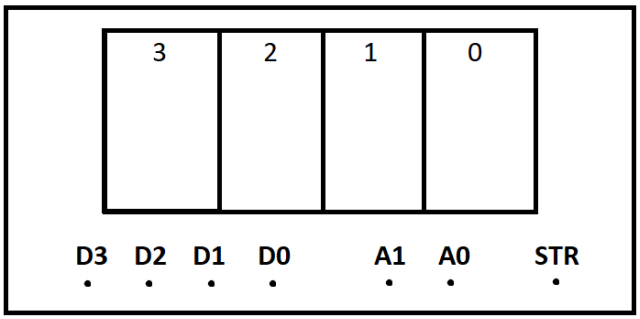
Given the above relation between BAC and ADC value, the maximum BAC readable by the device (ADC = 1023) is 0.33%. One would hope that this value would not be reached by any user given that, according to Wikipedia, primary symptoms of a BAC near between 0.3 and 0.4% are "lapses in and out of consciousness" and "low possibility of death". [14].

2.2 Piezoelectric

In order to get the PWM working, a PWM wave had to be set up. This process is described in the lab manual [3]: setting P1.2 to output, selecting its PWM option, choosing the period and duty cycle (for frequency of noise) and making TA count in up mode. The details of my implementation can be seen in Appendix Section 1.

2.3 7-Segment Displays

Figure 4 reviews the basics of changing the value displayed on the 4-digit 7-segment display.



For any one of the four displays, D0-3 should correspond to the binary representation of the number displayed as follows:

D3 D2 D1 D0

Which of position 0-3 to edit is selected by the values of A1 and A0:

A1 A0

STR must be raised high before a change in number, and lowered after the change.

Figure 4: 7-segment display connections [3].

The code written for the 7-segment displays, found in Appendix Section 1 under the subroutine "DisplayNum()", took an integer as input and cycled through the number using the modulus operator, saving each digit in an array and then projecting that digit onto the necessary 7-segment. It was important to connect P2.0 to the D0 on the 7-segment, P2.1 to D1, and so on, because

when translating each decimal digit of the subroutine's decimal input into a four-digit binary number, one had to be able to vary A0 and A1 without altering the four-digit number itself. An in-depth explanation of the code can be found commented in the aforementioned Appendix section.

2.4 Camera

The process of disassembling the camera is explained by Solomon Bisker ("smb") on instructables.com [1]. There are several stages of disassembly, the most important parts of which are documented in Figure 5. After disassembly, connections must be soldered onto the points indicated in Figure 6: across the mode button, across the shutter button and at the ground of the battery.

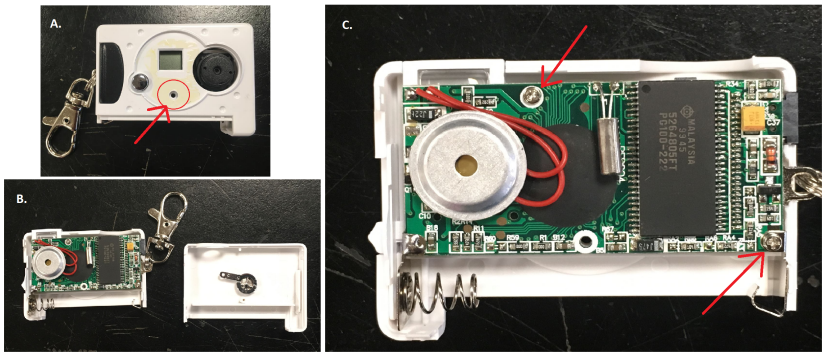


Figure 5: Screws to remove from camera indicated by red arrows; more details on disassembly available on instructables.com [1].

The camera was then put back together. To test whether the soldering was done correctly, the camera should turn on/take a picture when the mode/shutter wires are shorted together, respectively.

It's also very important to ensure that the wires are not interconnected or touching the camera's internal circuitry before reassembly. Black electrical tape on the outside of the camera can be used to stabilize the wires.

Before implementing the camera into the circuit, a multimeter should be used to determine in which direction current flows across the buttons, as that will determine the orientation of the transistors in the circuit. Once that has been determined, Figure 7 shows how the connections across each transistor are made. These transistors will make a short from camera high to camera low upon receiving a signal from the MSP430. A 1kΩ resistor at the base of each transistor is needed to protect the MSP.

Bisker provided Arduino code for the operation of the camera [1], the translated-to-MSP version of which can be found in Appendix Section 1 un-

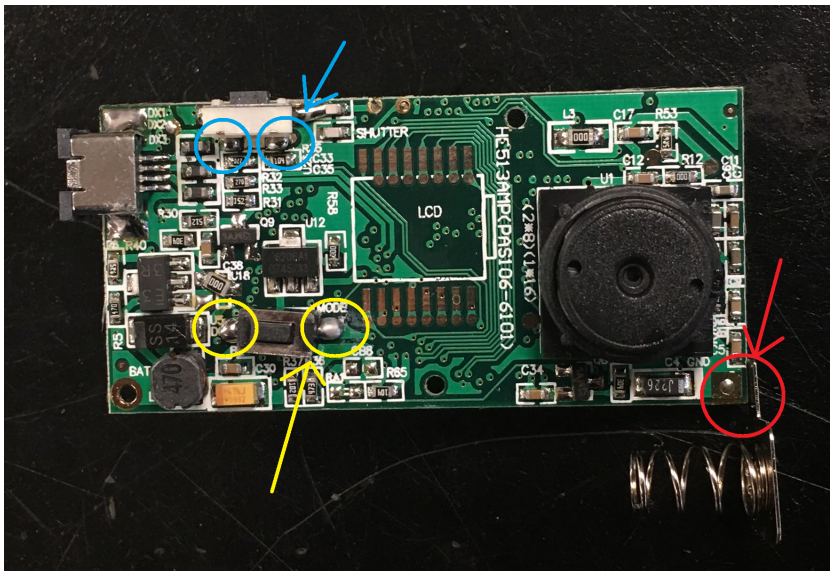


Figure 6: Solder points on the camera's circuit board: blue indicates shutter, yellow indicates mode and red indicates ground [1].

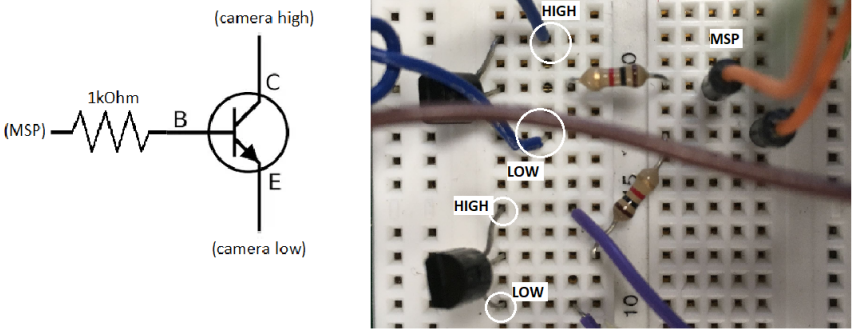


Figure 7: Theoretical and real transistor connections [1].

der the subroutine "takePhoto()". The duration of some pauses had to be changed to account for differences in responsiveness of the camera being used and Bisker's camera.

A common issue among post-2000's Vivitar Mini Digital Camera users was accessing the pictures from the camera. The comments on the aforementioned instructable suggested gphoto2 for Linux [1], but after many attempts neither

gphoto2 nor any of its additional packages, such as gtkam, worked. However, drivers for the Vivitar Mini Digital Camera were available online [11], but only for Windows XP, so a virtual machine (VirtualBox) was downloaded a tutorial [4] was followed to install Windows XP onto it. After downloading the drivers onto the virtual machine, the camera's internal memory could be accessed.

3 Procedure

The final device is shown in Figure 8, and a flowchart of its functionality is shown in Figure 9.

Figure 8: Final device upon presentation. The display shows the approximate baseline ADC reading.

For testing purposes, the button mentioned in Figure 9 was the MSP's built-in P1.3 button, though the code works just as well with an external push-button connected (through a resistor) to P1.3 on the MSP (which was needed when testing while the device was inside the box). The cylindrical container surrounding the MQ-3 in Figure 8 is to isolate it from external stimuli (for example, the alcohol in the Sharpie used to decorate the box). The code does not need to be reset after a measurement is taken and the program can run indefinitely.

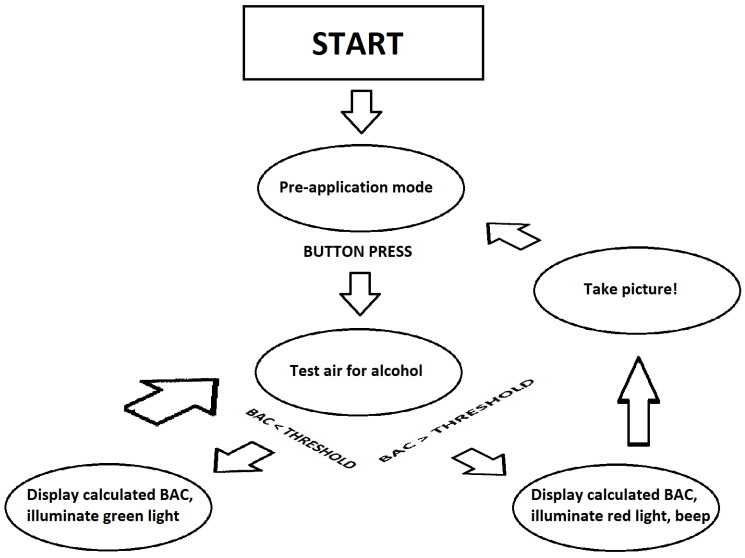


Figure 9: Flowchart of device functionality.

4 Results

The device did work as planned, turning the camera off/on and taking pictures upon receiving the predetermined ADC reading (sample photographs shown in Figure 10), but the precision of the BAC measurements are absolutely in question: Krumpus [7] notes that the MQ-3 is very sensitive to environmental changes, and my own observations show that the ADC reading continues to decay, albeit slowly, for hours after turning on the device. This decay could be because of the device's sensitivity to heat, as it continues to get hotter the longer it's turned on. However, the decay in baseline reading does seem to be exponential, the rate of decay approaching zero as a function of time.

At any given time, the ADC readings would fluctuate $\pm$ 10 bits, leading to an uncertainty in the BAC reading of $\pm$ 0.005%. However, it should be noted that if too much time passes between measurements the baseline reading changes. This makes it harder to construct a reliable linear relation between ADC and BAC, so to get the most stable y-intercept for the relation it's important to leave the device turned on for hours before calibrating.

5 Discussion

While the device functioned as planned, there are improvements that could be made. As it stands, the program works, but not as intuitively and efficiently as

Figure 10: Test trial results; both photos taken on April 6.

it could given more time to perfect it.

Firstly, the accuracy of the instrument could be improved by taking more data points for calibration; given more time, more solutions with known vapour concentrations could have been prepared using the method described in Section 2.1. Once the instrument's accuracy has been improved, its precision could also be improved by acquiring constants with more significant figures and doing chemical conversions more precisely. Then more decimal places on the display could be used and an analog-to-digital converter that allows for more decimal places, such as the MSP430's ADC12 unit, could be used in conjunction with a larger display.

For this device, the sensor was isolated with a simple plastic container (as seen in Figure 8 from Section 3). However, a more sophisticated breathing chamber could be employed, such as the one made by Michael Krumpus from Nootropicdesign, as shown in Figure 11.

In addition, the execution path shown in Figure 9 could be changed so that, instead of pausing and returning to pre-application mode after a higher-than-threshold reading, the program continues to display the alcohol concentration in the air, keeping the red LED illuminated until the alcohol level drops back well below the threshold (0.005% below the threshold to protect against triggering by random fluctuations). See Figure 12 for this improved execution path. This would help not only with more precise calibration and further optimization, letting us measure how long it takes for readings to fall back below threshold, but it would also make for a funner and more dynamic device. I wrote the code for this improvement, attached in Appendix Section 3, but was not able to get it working in time.

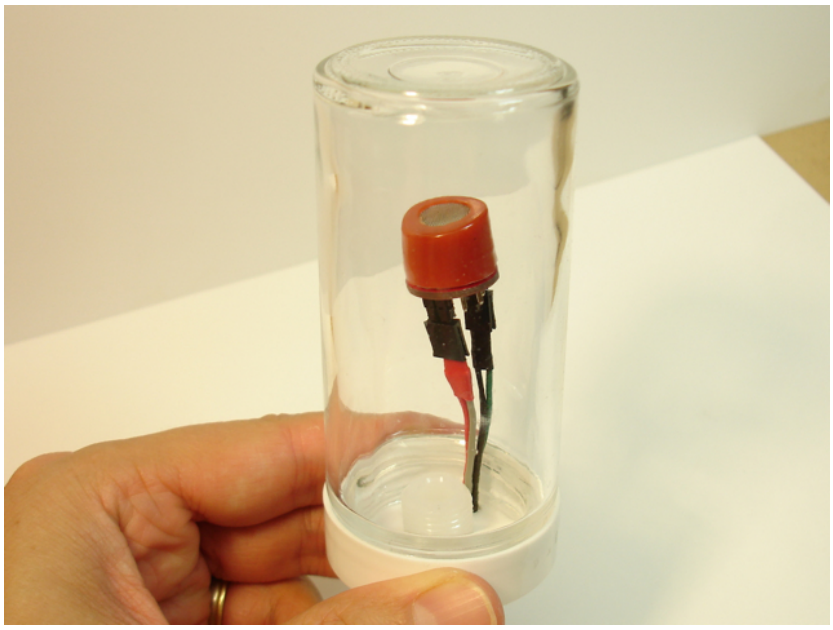


Figure 11: Glass MQ-3 isolation chamber [7]. A breathing tube is attached.

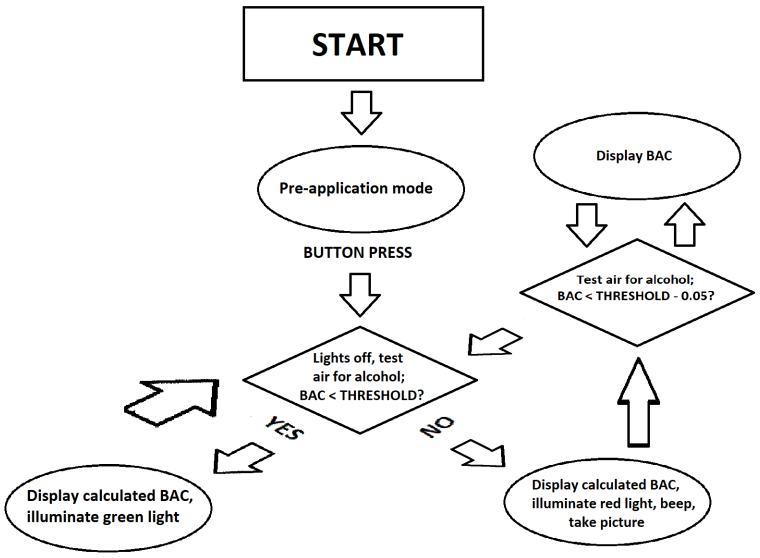


Figure 12: Execution flowchart for more dynamic BAC measurements.

6 Conclusions

The end goal of the project, to make a breathalyzer that displayed BAC in real-time and took pictures at a certain blood alcohol level, was achieved. The blood alcohol threshold can be changed as desired by a trivial adjustment of the code, and given the adjustments described in Section 5, the device could be made more accessible and easy-to-use for a wider market. However, by the end of the project, there were additions to its primary programming that I imagined making but had no time to do so.

One addition that could have been made is further camera control via the MSP430. From the camera's operation manual [13] we can see that, given connections to the camera's shutter and mode buttons, we can also easily access camera functions such as deleting photos, toggling high/low-resolution and accessing video mode. These features can be accomplished by simple interrupts triggered by button presses, and they would make for a more reusable and dynamic device.

Another feature I wished I had time to add was a water gun that fired at the user before taking the picture. Water guns in previous projects had been accomplished by hydraulic pumps, but in this case my idea was to use a single servo attached to a cord wrapped around the trigger of the gun. Of course, extra care would have had to be taken to ensure the water gun was properly sealed and would not leak!

Finally, pictures could be sent directly and processed through the MSP430. This would require soldering direct access to the camera's memory and bit-by-bit information transmission, or an indirect method of bit-by-bit transmission via sound or radio waves as such as is used in slow-scan TV [15]. An improvement like this would circumvent the need for a Windows XP OS and make the device more accessible! It would also allow for more immediate viewing of the photographs; if the breathalyzer photo booth were to be set up as a game at a bar, for example, then one would need to be able to see the photographs right away instead of downloading them later!

In conclusion, the project was a success in the goals that were set out for it, and with a few improvements to the integration of the discrete components (and with a more reliable sensor than the MQ-3) it could even be marketable! That being said, there are many ways in which the idea of the device could be improved upon to make it an all-around better and more enjoyable system. As it stands, though, it's a fun and intuitive device, though its accuracy when employing the MQ-3 should indicate its unsuitability for any legal tests. Indeed, even professionally-made breathalyzers suffer (albeit less frequently) from sensitivity to temperature, humidity and other external factors [2].

References

[1] Bisker, S. (2009, December 15). Hacking a Digital Keychain Camera for Arduino Control. Retrieved March 16, 2018, from http://www.instructables.com/id/Hacking-A-Keychain-Digital-Camera-for-Arduino-Cont/

[2] The Canadian Press. (2011, December 26). Lawyer questions accuracy of B.C. breathalyzers. Retrieved April 6, 2018, from https://www.ctvnews.ca/lawyer-questions-accuracy-of-b-c-breathalyzers-1.745362

[3] Chow, S., Kotlicki, A., Wicks, R., and Michal, C. (2018, January). Introduction to the Programming and Use of Microprocessors. Retrieved January 5, 2018, from http://www.phas.ubc.ca/~kotlicki/Physics_319/Lab_Manual-2017AK.pdf

[4] EverythingEpan. (2015, Jan 5). Windows XP Professional Installation in Virtualbox. Retrieved April 3, 2018, from https://www.youtube.com/watch?v=gjQiKTgeiqk

[5] Hanwei Electronics Co. (n.d). Technical Data MQ-3 Gas Sensor. Retrieved March 28, 2018, from http://nootropicdesign.com/projectlab/downloads/mq-3.pdf

[6] Knowino contributors. (2010, December 12). Henry's law. Retrieved March April 2, 2018, from http://www.tau.ac.il/~tsirel/dump/Static/knowino.org/wiki/Henry%27s_law.html

[7] Krumpus, M. (2010, September 17). Arduino Breathalyzer: Calibrating the MQ-3 Alcohol Sensor. Retrieved March 20, 2018, from http://nootropicdesign.com/projectlab/2010/09/17/arduino-breathalyzer/

[8] Krumpus, M. (2017, September 9). Wi-Fi Camera Trap. Retrieved March 20, 2018, from https://nootropicdesign.com/projectlab/2017/09/09/wifi-camera-trap/

[9] National Center for Biotechnology Information. (2018, March 31). PubChem Compound Database; CID=3776. Retrieved April 2, 2018, from https://pubchem.ncbi.nlm.nih.gov/compound/isopropanol#section=Vapor-Density

[10] Paradis, D. (2010, November 20). Cheap Motion Detection Wildlife Camera. Retrieved March 15, 2018, from http://www.instructables.com/id/Cheap-Motion-Detection-Wildlife-Camera/

[11] Sakar International Inc. (n.d.). Vivitar Camera Drivers Download. Retrieved March 28, 2018, from https://www.driverguide.com/driver/company/Vivitar/Camera/index.html

[12] Sander, R. (1999, February 17). Compilation of Henry's Law Constants for Inorganic and Organic Species of Potential Importance in Environmental Chemistry. Retrieved April 2, 2018, from http://www.henrys-law.org/henry-3.0.pdf

[13] Vivitar (2007). Vivitar Mini Digital Camera Operation Manual. Retrieved March 26, 2018, from https://www.camerahacker.com/Forums/DisplayComments.php?file=Digital%20Camera/Vivitar/Vivitar_Mini_Digital_Camera_User_Manual

[14] Wikipedia contributors. (2018, March 10). Blood alcohol content. In Wikipedia, The Free Encyclopedia. Retrieved 10:14, April 7, 2018, from https://en.wikipedia.org/wiki/Blood_alcohol_content

[15] Wikipedia contributors. (2018, March 22). Slow-scan television. In Wikipedia, The Free Encyclopedia. Retrieved 10:16, April 7, 2018, from https://en.wikipedia.org/wiki/Slow-scan_television

7 Appendix

The full code for the program, the code for the ADC level tester, and the almost-functional code for the improved BAC display all follow.

7.1 Full Code

```
#include "msp430.h"

#define    LED1                BIT0
#define    LED2                BIT6

#define    BUTTON              BIT3

#define    BEEP                BIT2

#define    STROBE              BIT5

#define    MODE                BIT4
#define    SHUTTER             BIT7

#define    PreAppMode          0
#define    RunningMode         1

#define    Threshold           0080

volatile unsigned int Mode;
volatile unsigned int High;
```

```
volatile unsigned int BAC;

void InitializeButton(void);
void PreApplicationMode(void);

void DisplayNum(int num);
void takePhoto(void);


void main(void)
{
    WDTCTL = WDTPW + WDTHOLD;                   // Stop WDT

    // ADC10 on and choose input P1.1 ("A1")
    ADC10CTL0 = ADC10SHT_2 + ADC10ON + ADC10IE; // turns on ADC with
        interrupts
    ADC10CTL1 = INCH_1;                         // input A1 (AKA p1.1)
    ADC10AE0 |= BIT1;                           // PA.1 ADC option select

    // set up button to interrupt preappmode
    InitializeButton();

    // set up ports for leds:
    P1DIR |= LED1 + LED2;
    P1OUT &= ~(LED1 + LED2);

    // set up ports for ADC display
    P2DIR = 63;
    P1DIR |= STROBE;

    // flash lights and wait for button press
    Mode = PreAppMode;
    PreApplicationMode();

    __enable_interrupt();

    for (;;)
    {

        High = 0;
        // lower-than-threshold alcohol level is assumed at the start of
            the main loop; this may change upon measurement
        P1OUT |= LED2;

        // ADC sampling and conversion start
        ADC10CTL0 |= ENC + ADC10SC;
        __bis_SR_register(CPUOFF + GIE);    // LPM0 with interrupts
            enabled; turn cpu off.
        // an interrupt is triggered when the ADC result is ready.
```

```
        // the interrupt handler performs one of two tasks based on air
            alcohol concentration, then restarts the cpu

        __delay_cycles(1000);        // wait for ADC Ref to settle

        if(High == 1)                // if more than threshold was
            detected
        {
            InitializeButton();
            Mode = PreAppMode;
            PreApplicationMode();
        }
        else
        {
            __delay_cycles(500000);  // wait half a second between
                measurements

        }

    }
}


/***********************************SUBROUTINES*********************************/


// toggles LEDs in low power mode until interrupted by button press
// from TI's MSP430G2553 temperature demo program
void PreApplicationMode(void)
{
    P1OUT |= LED1;               // enable LED toggling effect
    P1OUT &= ~LED2;

    // configure ACLK (not using SMCLK because SMCLK can't run in LMP3
        but ACLK can)
    BCSCTL1 |= DIVA_1;           // ACLK is 1/2 speed of source (VLO)
    BCSCTL3 |= LFXT1S_2;         // ACLK = VLO

    // make timer interrupt periodically; ISR will toggle lights
    TACCR0 = 1200;               // period
    TACTL = TASSEL_1 | MC_1;     // TACLK = ACLK, Up mode.
    TACCTL1 = CCIE + OUTMOD_3;   // TACCTL1 Capture Compare
    TACCR1 = 600;                // duty cycle
    __bis_SR_register(LPM3_bits + GIE); // LPM3 with interrupts
}


// configure push button: enable pullup resistor and interrupts on port 3
// from TI's MSP430G2553 temperature demo program
void InitializeButton(void)
```

```
{
    P1DIR &= ~BUTTON;
    P1OUT |= BUTTON;            // set internal resistor as pull-up
    P1REN |= BUTTON;            // enable internal resistor
    P1IES |= BUTTON;            // HIGH-to-LOW transition triggers P1IFG
    P1IFG &= ~BUTTON;           // clear interrupt flag
    P1IE |= BUTTON;             // enable interrupts
}


// takes an integer as input and displays its 4 least significant digits
    on the 7-segment displays
void DisplayNum(int num)
{
    int d[4] = {0,0,0,0};
    // initialize an array to hold the digits of num, from least
        significant to most significant
    int i = 0;
    while(num > 0)
    {
        d[i] = num % 10;
        // will return the least significant digit
        num = (num-(num % 10))/10;
        // equivalent to num = floor(num/10), but our makefile doesn't
            understand floor()
        i++;                            // continue to iterate through
            num
    }

    // change position 0 on display
    P1OUT |= STROBE;
    // d[0], num's least significant digit, is a one-digit decimal
        number and a 4-digit binary number
    P2OUT = d[0] + 0;
    // the four binary digits are stored in P2.0-P2.3; P2.4 and P2.5
        control A0 and A1, respectively
    P1OUT &= ~STROBE;
    // in this case, A0 = A1 = 0 means we're selecting position 0, which
        is the rightmost 7-segment display

    // change position 1 on display
    P1OUT |= STROBE;
    P2OUT = d[1] + 16;
    // here, P2OUT = 00(D3)(D2)(D1)(D0) + 010000, where D0-3 are the
        binary digits of d[1] and 010000b = 16
    P1OUT &= ~STROBE;
    // this means that the number (D3)(D2)(D1)(D0) = d[1] is displayed
        at position (A1)(A0) = 01b = 1

    // change position 2 on display
```

```
    P1OUT |= STROBE;
    P2OUT = d[2] + 32;
    // similiarly, here 32 = 100000b, so P2OUT = 32 + d[2] plots d[2] on
        position (A1)(A0) = (1)(0) = 2
    P1OUT &= ~STROBE;

    // change position 3 on display
    P1OUT |= STROBE;
    P2OUT = d[3] + 48;                  // 48 = 110000b
    P1OUT &= ~STROBE;
}


// turns on the camera, takes a single photo, then turns off the camera
void takePhoto(void)
{
    // set up ports for camera
    P1DIR |= SHUTTER + MODE;

    // button combinations for accessing different modes of the camera
        can be found in the operation manual
    // here, we just employ the on and off button sequences

    // turn camera on
    P1OUT |= MODE;
    __delay_cycles(250000);
    P1OUT &= ~MODE;
    __delay_cycles(1000000);

    // take picture
    P1OUT |= SHUTTER;
    __delay_cycles(250000);
    P1OUT &= ~SHUTTER;
    __delay_cycles(4000000);

    // turn camera off
    P1OUT |= MODE;
    __delay_cycles(250000);
    P1OUT &= ~MODE;
    __delay_cycles(250000);
    P1OUT |= SHUTTER;
    __delay_cycles(250000);
    P1OUT &= ~SHUTTER;
    __delay_cycles(1000000);

}

/*************************************ISRS*************************************/
```

```
// responds to button press by switching to running mode; uses WDT
    interrupt to de-bounce button
// from TI's MSP430G2553 temperature demo program
#if defined(__TI_COMPILER_VERSION__)
#pragma vector=PORT1_VECTOR
__interrupt void port1_isr(void)
#else
  void __attribute__ ((interrupt(PORT1_VECTOR))) port1_isr (void)
#endif
{
    P1IFG = 0;                          // clear interrupt flag
    P1IE &= ~BUTTON;                    // Disable port 1 interrupts
    WDTCTL = WDT_ADLY_250;              // set up watchdog timer duration
    IFG1 &= ~WDTIFG;                    // clear interrupt flag
    IE1 |= WDTIE;                       // enable watchdog interrupts

    TACCTL1 = 0;                        // turn off timer 1 interrupts
    P1OUT &= ~(LED1+LED2);              // turn off the LEDs
    Mode = RunningMode;

    __bic_SR_register_on_exit(LPM3_bits);
}


// WDT interrupt to de-bounce button press; prevents MSP from responding
    to two button presses close together
// from TI's MSP430G2553 temperature demo program
#if defined(__TI_COMPILER_VERSION__)
#pragma vector=WDT_VECTOR
__interrupt void wdt_isr(void)
#else
  void __attribute__ ((interrupt(WDT_VECTOR))) wdt_isr (void)
#endif
{
    IE1 &= ~WDTIE;                      // disable watchdog interrupt
    IFG1 &= ~WDTIFG;                    // clear interrupt flag
    WDTCTL = WDTPW + WDTHOLD;           // put WDT back in hold state
    P1IE |= BUTTON;                     // enable port 1 interrupts
}


// toggle lights if in preappmode; clear interrupts and turn CPU back on
    if not
// from TI's MSP430G2553 temperature demo program
#if defined(__TI_COMPILER_VERSION__)
#pragma vector=TIMER0_A1_VECTOR
__interrupt void ta1_isr (void)
#else
  void __attribute__ ((interrupt(TIMER0_A1_VECTOR))) ta1_isr (void)
#endif
```

```
{
    TACCTL1 &= ~CCIFG;                  // reset interrupt flag
    if (Mode == PreAppMode){
      P1OUT ^= (LED1 + LED2);           // toggle LEDs
    }
    else{
      TACCTL1 = 0;                      // prevent further interrupts
      __bic_SR_register_on_exit(LPM3_bits);
    }

}


// ADC10 interrupt service routine; calculates blood alcohol
    concentration from ADC result when ADC is ready
#if defined(__TI_COMPILER_VERSION__)
#pragma vector=ADC10_VECTOR
__interrupt void adc10_isr(void)
#else
  void __attribute__ ((interrupt(ADC10_VECTOR))) adc10_isr (void)
#endif
{

    // calculates BAC from ADC if ADC at base reading; if sensor voltage
        is fluctuating below base reading, it returns zero
    if(ADC10MEM > 100)
    {
        BAC = (int) 1000*(ADC10MEM-100)/2750;
    }
    else
    {
        BAC = 0;
    }

    // update 7-segment display
    DisplayNum(BAC);

    if (BAC < Threshold)
    {
        High = 0;                       // above-threshold indicator
            remains low
        P1OUT |= LED2;                  // green on if it wasn't already
        __bic_SR_register_on_exit(CPUOFF);
    }
    else
    {
        High = 1;                       // above-threshold indicator high
        P1OUT &= ~(LED2);               // green off
        P1OUT |= LED1;                  // red on if it wasn't already
```

```
        // Beep
        P1DIR |= BEEP;                    // P1.2 to output
        P1SEL |= BEEP;                    // P1.2 to TA0.1

        CCR0 = 800;                       // PWM Period
        CCTL1 = OUTMOD_7;                 // CCR1 reset/set
        CCR1 = 420;                       // CCR1 PWM duty cycle
        TACTL = TASSEL_2 + MC_1;          // SMCLK, up mode
        // Beep

        takePhoto();

        __delay_cycles(500000);           // wait half a second between
            measurements

        P1DIR &= ~BEEP;                   // disable PWM signal

        __bic_SR_register_on_exit(CPUOFF);
    }
}
```

7.2 ADC Level Tester

The following program is to be run in conjunction with python-serial-plot.py from TI's temperature demo.

```
// Credit to P319 for base code except for sprintf functionality.

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

#define RXD      BIT1
#define TXD      BIT2

#define BUTTON   BIT3

#define TXLED    BIT0
#define RXLED    BIT6

void UART_TX(char * tx_data);    // Function Prototype for TX
int adcLevel;

void main(void)
{
   // Stop Watch dog timer
   WDTCTL = WDTPW + WDTHOLD;
```

```
// Set DCO to 1 MHz
BCSCTL1 = CALBC1_1MHZ;
DCOCTL = CALDCO_1MHZ;

// Ensure button is input and enable resistor
P1DIR &=~BUTTON;
P1OUT |= BUTTON;
P1REN |= BUTTON;

// set up LEDs:
P1DIR |= TXLED + RXLED;
P1OUT &= ~(TXLED + RXLED);

// Select TX and RX functionality for P1.1 & P1.2
P1SEL = RXD + TXD ;
P1SEL2 = RXD + TXD;

// Have USCI use System Master Clock: AKA core clk 1MHz
UCA0CTL1 |= UCSSEL_2;
UCA0BR0 = 104;
UCA0BR1 = 0;

// Modulation UCBRSx = 1 and start USCI state machine
UCA0MCTL = UCBRS0;
UCA0CTL1 &= ~UCSWRST;

// ADC10 on and choose input P1.1 ("A1")
ADC10CTL0 = ADC10SHT_2 + ADC10ON;    // turns on the ADC!
ADC10CTL1 = INCH_1;                 // input A1 (p1.1) for conversion
ADC10AE0 |= 0x02;                   // PA.1 ADC option select

// Enable interrupts
__bis_SR_register(GIE);

while(1)
{

 ADC10CTL0 |= ENC + ADC10SC;      // Sampling and conversion start
 while (ADC10CTL1 &ADC10BUSY);    // wait until ADC no longer busy

 adcLevel = ADC10MEM;

 P1OUT ^= TXLED;                  // toggle LED

 char str[80];
 sprintf(str, "Current ADC Level = %i", adcLevel);

 UART_TX(str);                    // transmit data over UART
```

```
    __delay_cycles(500000);          // delay for half a second

   }
}

/*
 * Accepts a character pointer to an array and prints in over serial
 */
void UART_TX(char * tx_data) //
{
    unsigned int i=0;
    while(tx_data[i]) // Increment through array, look for null pointer
        (0) at end of string
    {
        while ((UCA0STAT & UCBUSY)); // Wait if line TX/RX module is
            busy with data
        UCA0TXBUF = tx_data[i]; // Send out element i of tx_data array
            on UART bus
        i++; // Increment variable for array address
    }
}
```

7.3 Improved BAC Display

Only the parts that have changed from the original program are included in the following code; changes were made to the main loop and the ADC interrupt service routine (all ISR functionality was moved to the main loop and ADC interrupts removed), and the subroutine waitUntilClear() was added.

```
// main loop
for (;;)
    {

        ADC10CTL0 |= ENC + ADC10SC;     // ADC sampling and conversion
            start
        while (ADC10CTL1 &ADC10BUSY);

        __delay_cycles(1000);           // Wait for ADC Ref to settle

        if(ADC10MEM > 100)
        {
            BAC = (int) 1000*(ADC10MEM-100)/2750;
        }
        else
        {
            BAC = 0;
        }
```

```
        // update 7-segment display
        DisplayADC(BAC);

        if (BAC < Threshold)
        {
            High = 0;
            P1OUT &= ~(LED1);       // red off
            P1OUT |= LED2;          // green on if it wasn't already
        }
        else
        {
            High = 1;
            P1OUT &= ~(LED2);        // green off
            P1OUT |= LED1;           // red on if it wasn't already

            // Beep
            P1DIR |= BEEP;           // P1.2 to output
            P1SEL |= BEEP;           // P1.2 to TA0.1

            CCR0 = 800;                  // PWM Period
            CCTL1 = OUTMOD_7;            // CCR1 reset/set
            CCR1 = 420;                  // CCR1 PWM duty cycle
            TACTL = TASSEL_2 + MC_1;     // SMCLK, up mode
            // Beep

            takePhoto();

            __delay_cycles(500000);

            P1DIR &= ~BEEP;

            while(High == 1)
            {
                waitUntilClear();
            }

        }

        __delay_cycles(500000);

    }


/**********************************SUBROUTINES*********************************/

void waitUntilClear(void)
{
    ADC10CTL0 |= ENC + ADC10SC;     // again, ADC sampling and
        conversion start
```

```
    while (ADC10CTL1 &ADC10BUSY);

    __delay_cycles(1000);

    // update 7-segment display
    BAC = 1000*(ADC10MEM-100)/2750;
    DisplayADC(BAC);

    if(BAC < Threshold - 0.005)
    {
        High = 0;
        P1OUT &= ~(LED1);
        P1OUT |= LED2;
    }
    else
    {
        High = 1;
    }

    __delay_cycles(500000);

}
```