Smart Bike Accessory

Team 15: Martin Romo, Cameron Donovan, Qingyuan Li
EE 459 Final Report
Spring 2020

Table of Contents

Overview

Project Description

The smart bike accessory is a multifunctional bike attachment designed to promote better health and safety for the rider. It has the ability to warn riders if there is a vehicle dangerously approaching from behind and can automatically toggle the bike light based on ambient light levels. It also provides the rider with important ride information such as the current time, location, speed, altitude, direction, and time travelled.

Inputs

There are several sensors which sample and transmit data needed to make this possible. There is an on/off switch, ultrasonic sensor, light sensor, and GPS module.

On/Off Switch: The on/off switch is used to reduce power because this is a battery powered device.

Ultrasonic Sensor: The ultrasonic sensor is mounted at the back of the bike to determine if there is a car dangerously approaching from behind.

Light Sensor: The light sensor can detect the level of ambient light in order to determine if the bike lights need to be toggled on for safety or off to save battery life.

GPS Module: The GPS module is used for multiple purposes. It offers a clock to determine travel time. It also offers positional data to determine distance travel and elevation change.

Outputs

Buzzer: The buzzer is necessary to warn the rider if there is danger. It has a sharp piercing sound which can be easily identified and heard of the sound of traffic.

LED Bike Light: The LED bike light is mounted at the front and back of the bike for the safety of the rider and drivers.

LCD Display: The LCD display shows all the information to the user. It will display current time, GPS information, bike speed, and total time traveled. Because of the backlight, it can change the background to red or yellow in order to provide a visual warning to the rider.

Block Diagram

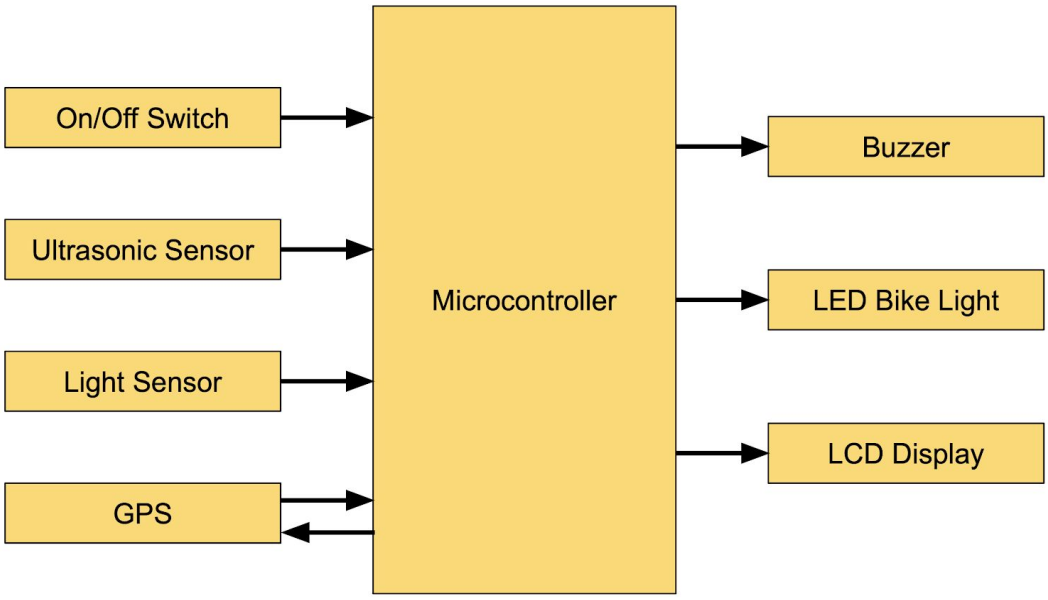

Figure 1: Full block diagram showing how all components interact

Schematic

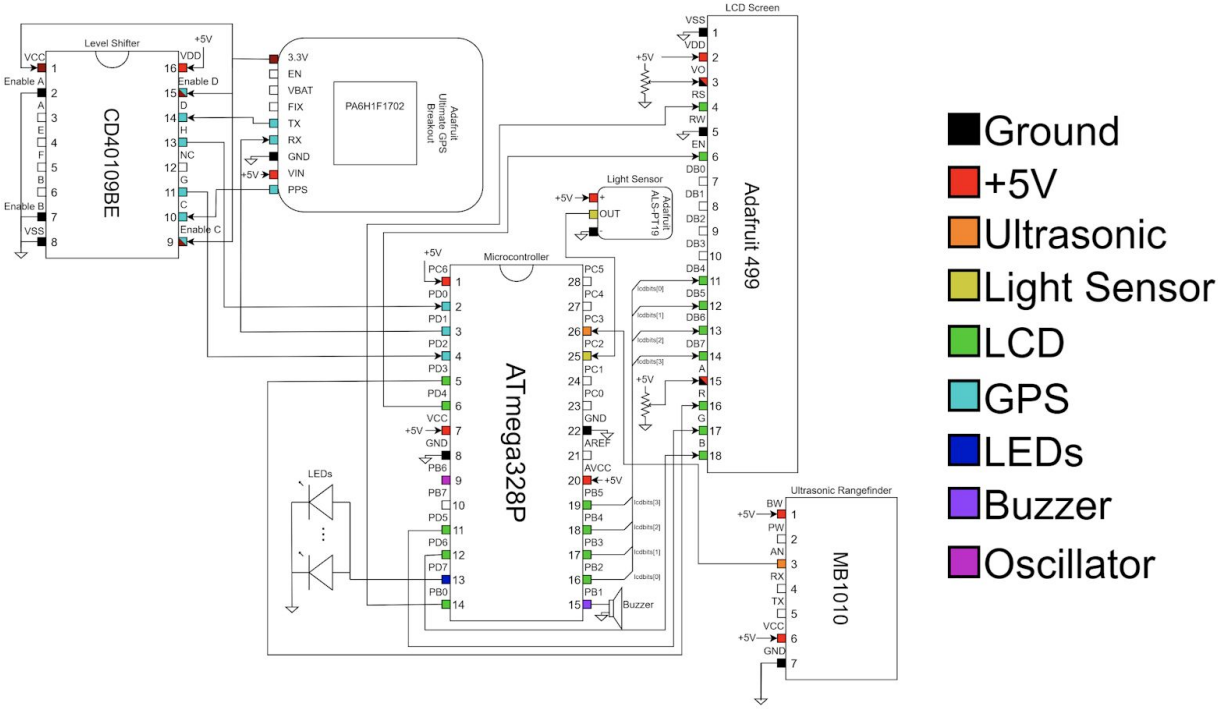

Figure 2: Full schematic detailing all wired connections

Components

Atmel ATmega328p - 8-bit Microcontroller

Description

Figure 3: Atmel ATmega328p - 8-bit Microcontroller

This component controls all the logic of the device. Every peripheral is connected and controlled by the microcontroller. This chip was chosen because it was the smallest, least expensive microcontroller with the necessary number of IO pins. Some pins were left free in order to allow future functionality. Through careful selection of the other components discussed below, we were able to simplify the data collection process and fit all the components with the limited number of pins.As shown in Fig. 4, we are utilizing the majority of pins. Most are regular IO pins; however, we are also using the chip's ability to perform analog-to-digital conversions and the designated UART lines. The computing power is also sufficient to perform the necessary computations for processing the ultrasonic sensor data.

Hardware Pinouts

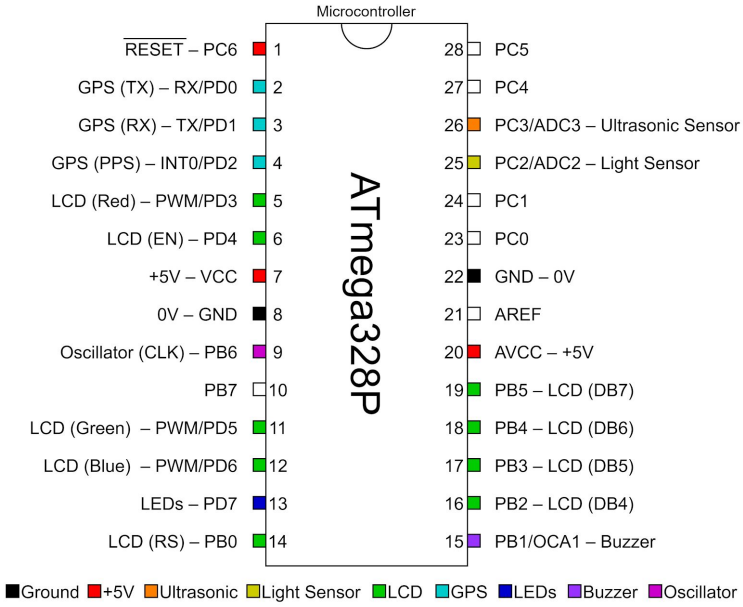

Figure 4: Pin Designations for the ATmega328p

Because of our choice of peripherals, most pins are used as generally IO pins, but there are some notable exceptions. Regular IO pins will be detailed in the **Software Integration** section of the respective peripheral. Pins used for non-general I/O purposes are detailed below:

Pin 1 - $\overline{RESET}$. This pin is an active-low reset signal for the microcontroller, it is connected to the 5V bus line on the board to keep the program running as long as the board is powered.

Pin 2 - RX. This pin accepts serial transmissions. It is the only pin on the microcontroller with this capability. It limited our peripherals to one serial device unless we attempted to multiplex the serial communications.

Pin 3 - TX. This pin transmits serial communication. Like the RX serial pin, it is the only TX serial pin.

Pin 7 - VCC. This pin powers the microcontroller. It is connected directly to the 5V power supply.

Pin 8 - GND. This pin is connected directly to ground.

Pin 9 - CLK. This pin is connected to the crystal oscillator so that the microcontroller has an accurate clock signal.

Pin 20 - AVCC. This pin is the reference voltage for the microcontroller's built in ADC. All the peripherals that use ADC have a max voltage of 5V, so it is connected directly to the supply voltage.

Pin 22 - GND. This pin is connected directly to ground.

Pin 25 - PC2/ADC2. This pin is connected to the analog light sensor's OUT pin. This pin is capable of analog to digital conversion and is used to convert the analog value from the light sensor to a 10-bit digital value.

Pin 26 - PC3/ADC3. This pin is connected to the ultrasonic rangefinder's AN pin. This pin is able to convert the analog signal from the ultrasonic to a 10-bit digital value.

Other Pins on microcontroller not used in this project:

Pins 23 & 24 - PC0/ADC0 & PC1/ADC1. These pins are also capable of doing analog to digital conversions. We chose to leave these free instead of using them for general I/O in case we desire to include additional analog devices in the future.

Pins 27 & 28 - PC4/ADC4/SDA & PC5/ADC5/SCL. These pins are capable of doing ADC as well, but they can also be used for I2C communication. None of our modules required I2C, but we left these unused to allow for I2C devices to be added in the future.

Software Integration

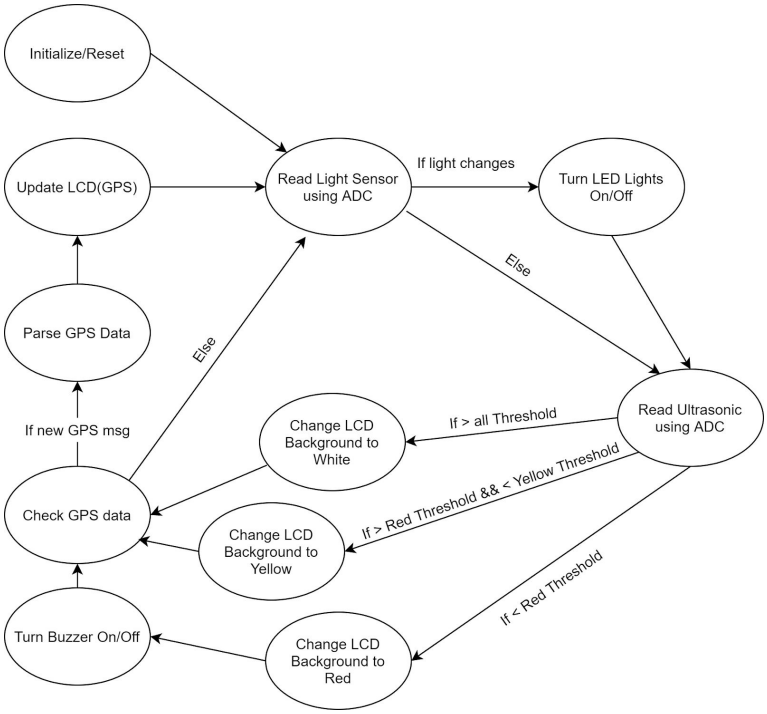

Figure 5: General State Machine of the program

Brief code snippets from smart_bike.c, implementation of function calls and communication with other modules will be shown in later sections.

- Initializing LCD, GPS, ADC modules, LEDs, and Buzzer

```
40:  void init(void) {
41:      lcd_init();
42:      gps_init();
43:      adc_init();
44:      PORTD &= ~LED_BIT;
```

```
 45:      PORTB &= ~BUZZER_BIT;
 46:      sei(); // Enable interrupts
 47:  }
```

- Function running state machine described above

```
 58:  void loop(void) {
 59:      light_update();
 60:      sonar_update();
 61:      if(gps_check()) {
 62:          lcd_update();
 63:      }
 64:      _delay_ms(10);
 65:  }
```

- The program's main() function

```
190:  int main(void)
191:  {
192:      init();
193:      splash();
194:      while(1) {
195:          loop();
196:      }
197:      return 0;   /* never reached */
198:  }
```

- Initializing the ADC and taking a 10-bit sample

```
//From adc.c
  6:  void adc_init() {
  7:   // Initialize the ADC
  8:      ADMUX &= ~(1<<REFS1);  //Set REF bits to AVCC (REFS1=0 and
REFS0=1)
  9:      ADMUX |= (1<<REFS0);
 10:
 11:      ADMUX &= ~(1<<ADLAR);   //Clear ADLAR to enable 10-bit ADC
 12:
 13:      ADCSRA |= (1<<ADPS2)|(1<<ADPS1);  //Set prescaler bits to 64
 14:      ADCSRA &= ~(1<<ADPS0);            // 7.37MHz clock turns to
115kHz which is within 50kHz-200kHz clock for ADC
 15:
 16:      ADCSRA |= (1<<ADEN);    //Enable ADC
 17:  }
 18:
 19:  uint16_t adc_sample(uint8_t channel) {
 20:      ADMUX &= 0xF0; //Clear mux bits
```

```
21:       ADMUX |= channel&0x0F; //Set channel
22:       ADCSRA |= (1<<ADSC); //Start conversion
23:       while(ADCSRA & (1<<ADSC));
24:       uint16_t result=ADC;
25:       return result;
26:   }
```

Adafruit 749 - Ultimate GPS Breakout &
Texas Instruments CD40109BE CMOS Low-to-High Voltage Level Shifter

Description

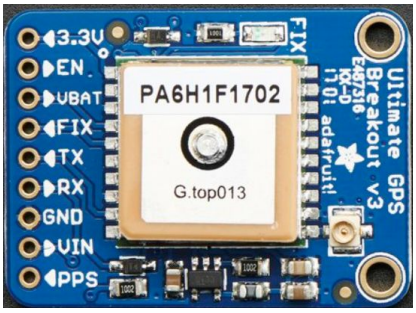

Figure 6: Adafruit 749 - Ultimate GPS Breakout

The Adafruit Ultimate GPS Breakout board is built on MediaTek's MTK3339 GPS module. This module is capable of tracking up to 22 satellites using 66 different channels and has a receiver with a high sensitivity of -165 dBm. This GPS module also has a low current draw of only 25 mA when tracking and 20 mA when navigating. Location updates can be transmitted at a rate of 10 times per second. The board includes a built in antenna which is suitable for outdoor use, but an external antenna can be connected if needed for testing inside if the internal antenna is unable to find a fix.

An internal voltage regulator allows the microcontroller to use 5V logic levels for inputs to the GPS. However, the GPS will have outputs in 3.3V which are not guaranteed to be read correctly by the microcontroller so a voltage level shifter is used to ensure that signals meet the Atmega328p's logic level requirements.

The Texas Instruments CD40109BE level shifter has 4 available channels for signals to be shifted up from one CMOS logic level to another. For this project we only need 2 channels to shift the GPS modules' Tx and PPS signals from 3.3V up to 5V to properly communicate with the microcontroller.

The GPS module includes a real time clock that can be powered using a 3V CR1220 coin battery as the power source when the device is powered off. The RTC allows the GPS module to output the correct time in its data transmissions after the board is powered on and before a GPS fix is obtained which allows the LCD to always display the correct time.

Transmitting location updates is done with TTL serial communication at a default baud rate of 9600. Updates are transmitted in the form of NMEA sentences which is a standard used by GPS receivers that specifies how data should be communicated using marine electronics. The start of a new sentence is indicated with a '$' character followed by the sentence type, data values, checksum, and terminated with the new line character '\n'. The data values are given as a list of comma separated values where the meaning of each value is based on the sentence type. For this project the 2 sentence types that provide relevant information are GGA and RMC.

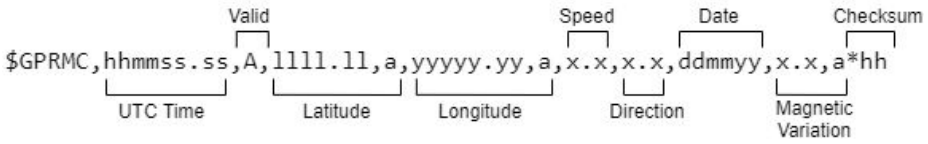

Figure 7: Example of data contained in RMC type sentences

GGA Sentences

These sentences provide GPS fix data and 3D location data. Important data values that we extract from these sentences include: time, latitude, longitude, fix quality, and altitude.

RMC Sentences

These sentences provide the recommended minimum data needed for navigation. The important values that we use from RMC sentences are: time, active status, latitude, longitude, speed, direction, and date.

Hardware Pinouts

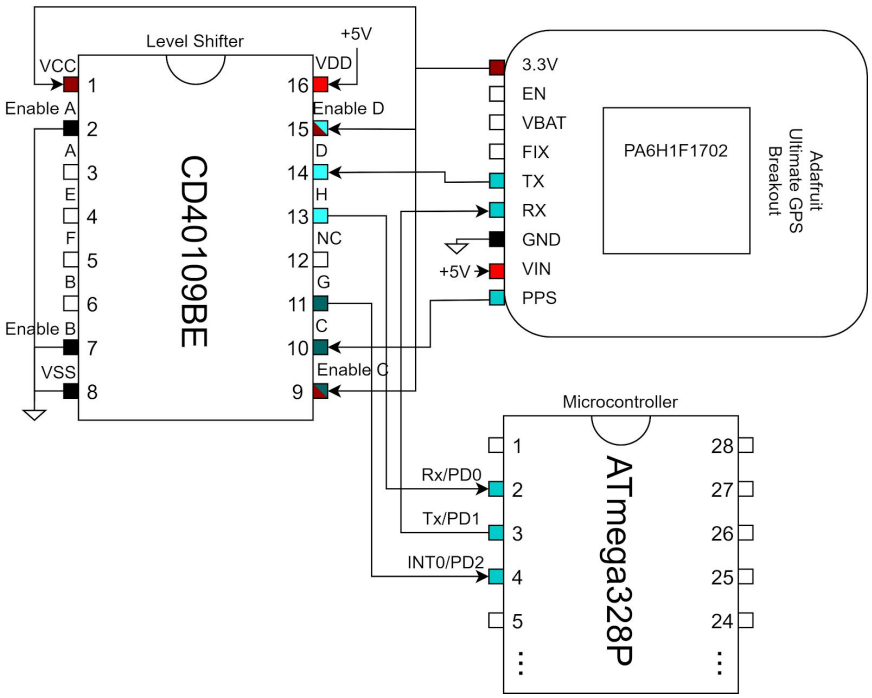

Figure 8: Schematic for GPS and Level Shifter

VIN - Voltage Input. The MTK3339 module used on the board requires a 3.3V source, but the board's built in voltage regulator allows this pin to be connected to our 5V power source.

GND - Ground Input. This can simply be connected to the ground bus line.

RX - Receive serial pin for the GPS module. This module can receive messages from the microcontroller using TTL serial communication to modify some of the default settings such as baud rate, types of NMEA sentences transmitted, frequency of transmissions, etc. This will be connected to the TX pin of the microcontroller. An internal logic level shifter allows the microcontroller to use 5V logic when transmitting which will be shifted down to 3.3V logic for the GPS module to read.

TX - Transmit serial pin for the GPS module. Data is transmitted using TTL serial communication with 8 data bits, no parity, 1 stop bit, asynchronous communication, and at a default baud rate of 9600. The output logic is 3.3V so we will inject a voltage level translator to convert the 3.3V CMOS logic levels to 5V CMOS logic levels. Thus, the Tx pin is connected to the input pin D on the level

shifter and the corresponding output pin H is connected to the microcontroller's RX pin.

PPS - A "pulse per second" output. This output creates a short 50ms-100ms 3.3V pulse each second. We use this output to update our elapsed time by creating an interrupt handler on the pin and updating the time when the interrupt is a rising edge. This is connected to the microcontroller's PD2 pin with the voltage level translator in between. Here the PPS signal is connected to the level shifter's input pin C and the corresponding output pin G is connected to the microcontroller's PD2 pin.

3.3V - This pin provides a 3.3V output signal from the board's 3.3V board regulator. This 3.3V source is used as an input to the voltage level translator's VCC pin to indicate that the inputs to the translator are using 3.3V CMOS logic levels. The signal is also connected to the level shifter's active-high Enable C and Enable D inputs to allow the level shifter to work on the two channels used.

Other Pins on GPS module not used in this project:

FIX - This is an output that pulses to indicate if the GPS has obtained a fix. If no fix has been obtained then a 200ms 3.3V pulse is created once every second. If a fix is obtained then a 200ms 3.3V pulse is created once every 15 seconds. This is not required because when the GPS transmits GGA and RMC messages it will contain information on whether a fix is obtained. For GGA messages a non-zero fix quality value indicates a fix is found and that the data is accurate. For RMC messages an active status of "A" indicates a fix and "V" indicates no fix.

VBAT - This is an input power source for the RTC battery backup. This line is connected to the coin battery spot on the back of the module, the pin is only needed if it needs to be connected to an external power source. In this project a coin battery will be used in the battery holder spot so we do not need to connect this pin input to an external power source.

EN - Enable pin for the GPS module. The board connects this line with a 10K pull-up resistor. If turning off the GPS is desired the pin can be pulled to ground, but for this project we desire the GPS to stay turned on when the device is powered on.

Software Integration

Code snippets from gps.c

- 2 string buffers are used, 1 to contain the last fully received message and another to buffer the currently transmitted message, when a message is finished being received the pointers will swap

```
18:  volatile char _line1[MAX_SENTENCE_LEN];
19:  volatile char _line2[MAX_SENTENCE_LEN];
20:  volatile char* _buff_line = _line1;
21:  volatile char* _last_line = _line2;
22:  volatile uint8_t _buff_idx = 0;
```

- Global array of strings to store what will be printed on the LCD
- When parsing the NMEA sentences theses strings get directly modified

```
53:  // 0xDF is the character code for displaying a degree symbol on the
LCD
54:  char display_screen[6][21] = {
55:      "--/--/--   --:-- GMT",
56:      "    --\xDF --.----' -  ",
57:      "   ---\xDF --.----' -  ",
58:      "Altitude: ----.- m  ",
59:      "Speed:    ----.- mph",
60:      "Direction:    --    "
61:  };
```

- Initializing the serial communication
- Communicating to the GPS module to only transmit RMC and GGA sentences to ensure each transmission contains useful information
- Limiting transmission rate to 1 Hz to make the GPS only transmit RMC and GGA sentences once a second and prevent frequent updates on LCD
- Enabling the rising edge interrupt for incrementing elapsed time.

```
70:  void gps_init(void) {
71:     UBRR0 = UBRR; // Set baud rate
72:     UCSR0B |= (1 << TXEN0) | (1 << RXEN0); // Enable RX and TX
73:     UCSR0C = (3 << UCSZ00); // Async., no parity, 1 stop bit, 8 data
bits
74:     gps_stringout("$PMTK314,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0*28\n"); // Only receive RMC and GGA sentences
75:     gps_stringout("$PMTK220,1000*1F\n"); // Update at 1 Hz frequency
76:     UCSR0B |= (1 << RXCIE0); // Enable RX Interrupt
77:     //Enable interrupt for PPS pin
78:     EIMSK |= (1 << INT0); // Enable the INT0 interrupt located on PD2
```

```
 79:     EICRA |= (1 << ISC01)|(1 << ISC00); //Interrupt only on rising
edge
 80: }
```

- Called from main() to copy the fully received message to ensure that the gps_parse() function is done on a string that will not get modified from the Rx interrupt

```
 93:  uint8_t gps_readline(char* str) {
 94:      if(_new_msg) {
 95:          strcpy(str, (char*)_last_line);
 96:          _new_msg = 0;
 97:          return _line_len;
 98:      }
 99:      return 0;
100:  }
```

- Called from main() in smart_bike.c to parse a newly received message and update the display_screen array
- Returns a code to indicate which fields contain valid information in case display_screen was updated, but the sentence is later found to be invalid
- If a 30 seconds have passed since the last successfully parsed message then a special code is sent to indicate to the LCD to display a waiting message

```
 31:  #define PARSE_ERROR_CODE (_msgs_elapsed >= 60)?-1:_valid_data
107:  int8_t gps_parse(char* str, uint8_t length) {
108:      //Check if checksum is accurate
109:      if(!gps_checksum_check(str, length))
110:          return PARSE_ERROR_CODE;
111:      //ptr1 will be at the start of a field, ptr2 will be at the end
112:      char* ptr1 = str;
113:      char* ptr2 = str;
114:      ptr2 = strchr(ptr1, ',');
115:
116:      if(strncmp(ptr1+2, "GGA", 3) == 0) { // Parse GGA
117:          // Time field
118:          // Format: hhmmss.ss
119:          // hh - hours, mm - minutes, ss.ss - seconds
120:          ptr1 = ptr2+1;
121:          ptr2 = strchr(ptr1, ',');
122:          if(!ptr2 || ptr2-ptr1 < 6)
123:              return PARSE_ERROR_CODE;
124:          if(gps_set_time(ptr1))
125:              _valid_data |= VALID_TIME;
126:          else
127:              _valid_data &= ~VALID_TIME;
```

```
---:         ...
183:     } else if(strncmp(ptr1+2, "RMC", 3) == 0) { // Parse RMC
---:         ...
267:         return PARSE_ERROR_CODE;
268:     }
269:     return PARSE_ERROR_CODE;
270: }
```

- Transmitting to GPS module

```
436: void gps_stringout(const char* str) {
437:     while(*str != '\0') {
438:         gps_charout(*str);
439:         str++;
440:     }
441: }

446: void gps_charout(char ch) {
447:     while ((UCSR0A & (1<<UDRE0)) == 0);
448:     UDR0 = ch;
449: }
```

- Validating checksum of the received sentence by XORing everything between the '$' and '*' characters

```
455: uint8_t gps_checksum_check(const char* str, uint8_t length) {
456:     if(length < 10 || str[length-3] != '*')
457:         return 0;
458:     int8_t cs1 = hex_to_int(str[length-2]);
459:     int8_t cs2 = hex_to_int(str[length-1]);
460:     if(cs1 < 0 || cs2 < 0)
461:         return 0;
462:     uint8_t checksum = (cs1<<4)|cs2;
463:     uint8_t i = 0;
464:     uint8_t parity = 0;
465:     while(i < length-3) {
466:         parity ^= str[i++];
467:     }
468:     return (checksum == parity);
469: }

475: int8_t hex_to_int(char c) {
476:     if(c >= '0' && c <= '9')
477:         return c-'0';
478:     if(c >= 'A' && c <= 'F')
479:         return 10+(c-'A');
480:     if(c >= 'a' && c <= 'f')
```

```
481:          return 10+(c-'a');
482:      return -1;
483:  }
```

- Interrupt handler for receiving characters from the GPS

```
488:  ISR(USART_RX_vect) {
489:      char ch = UDR0;
490:      if(ch == '\n') {
491:          ch = '\0';
492:      }
493:
494:      if(ch == '$') {
495:          _start_flag = 1;
496:          _buff_idx = 0;
497:          if(!(_valid_data & VALID_LOC)) {
498:              _msgs_elapsed++;
499:              if(_msgs_elapsed > 250) // Prevent overflowing to 0
500:                  _msgs_elapsed = 200;
501:          }
502:      } else if(_start_flag) {
503:          _buff_line[_buff_idx++] = ch;
504:          if(ch == '\0') { // End of transmitted line
505:              _start_flag = 0;
506:              _new_msg = 1;
507:              _line_len = _buff_idx-1;
508:
509:              // Swap _buff_line and _last_line pointers
510:              char* temp = (char*)_buff_line;
511:              _buff_line = _last_line;
512:              _last_line = temp;
513:          } else if(_buff_idx >= MAX_SENTENCE_LEN) { // Discard line if
longer than expected
514:              _start_flag = 0;
515:          }
516:      }
517:  }
```

- Interrupt handler for updating elapsed time when PPS pulses

```
519:  ISR(INT0_vect) {
520:      elapsedTime++;
521:  }
```

Adafruit 499 - RGB backlight positive LCD 20x4

Description

Figure 9: LCD Display Example

This LCD is a 20-character wide, 4-line character display that uses the common HD44780 LCD controller. The LCD displays black text on a RGB background. The RGB backlight is controlled by 3 LEDs (red, green, and blue) with pulse width modulation. The LCD communicates using a parallel interface and allows for either a 4-bit mode or 8-bit mode interface. To save input pins for the microcontroller, the 4-bit mode is used for this project.

Hardware Pinouts

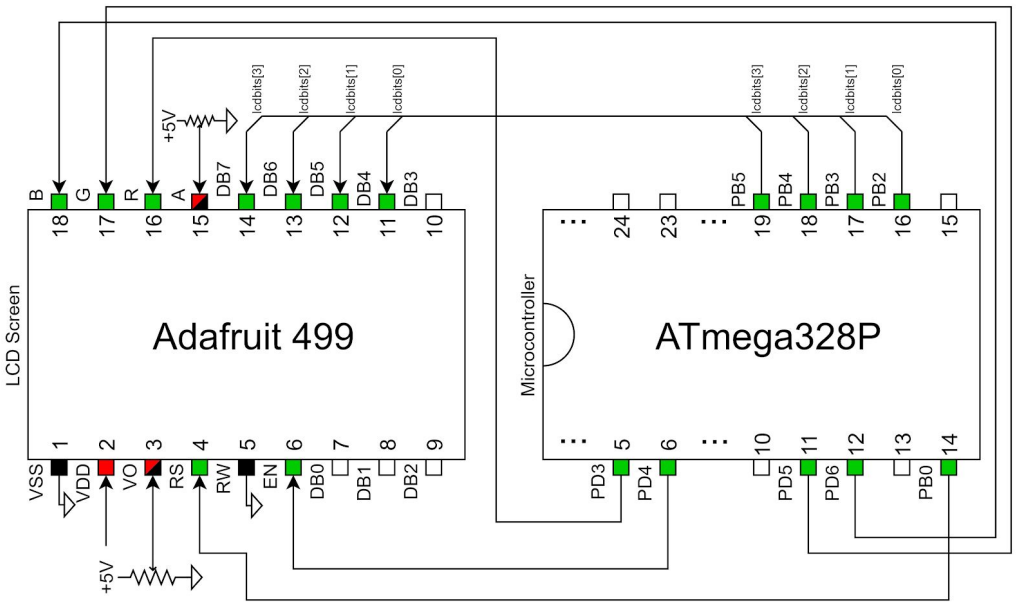

Figure 10: Schematic of LCD Display

Pin 1 - V_{SS}. This pin is connected to the ground bus line on the board.

Pin 2 - V_{DD}. This pin for the source voltage for logic operating. This pin is connected to the +5V bus line on the board because the LCD requires a 5V source.

Pin 3 - VO. Operating voltage for the LCD. This should be connected to the 10K Ohm potentiometer included with the LCD. The potentiometer can then be adjusted to change the contrast as needed.

Pin 4 - RS. This pin is for the register select signal and controls whether the LCD is receiving data to write or an instruction to execute. A low voltage input selects the instruction register for writing commands, while a high voltage selects the data register for print characters on the display. This pin is connected to the Atmega328p's PB0 pin.

Pin 5 - R/W. This pin controls whether the LCD is in read or write mode. If a low voltage input is given then write mode is selected, whereas a high voltage input selects the read mode. For this project there is no need to read from the LCD, so this pin is connected directly to the ground bus to keep the write mode active.

Pin 6 - EN. This pin provides the enable signal to the LCD and notifies the LCD that the data/instruction on the data pins are valid and ready to be executed. This pin should remain idle low until a byte is ready to be sent to the LCD. When this occurs then the enable signal should pulse high for at least 140ns. This is connected to PB4 on the microcontroller.

Pins 7, 8, 9, 10 - DB[3:0]. These pins are for the four least significant bits of the data byte when using the 8-bit interface. Pin 7 is DB0, pin 8 is DB1, pin 9 is DB2, and pin 10 is DB3. Since this project uses the 4-bit interface, these pins are not needed and can be left floating and unconnected.

Pins 11, 12, 13, 14 - DB[7:4]. These pins are for the four most significant bits of the data byte when using the 8-bit interface. Pin 11 is DB4, pin 12 is DB5, pin 13 is DB6, and pin 14 is DB7. Since a 4-bit interface is used, whenever the microcontroller needs to send a byte it first transmits the upper four bits to these pins and later transmits the lower four bits. Connections to the microcontroller are as follows: pin 11 to PB2, pin 12 to PB3, pin 13 to PB4, pin 14 to PB5.

Pin 15 - A. This pin is for the positive LED backlight power supply. This can be connected to the +5V power bus on the board. If this causes the display to be too bright then the brightness can be modified by adding a resistor or potentiometer to lower the brightness to an appropriate level.

Pin 16, 17, 18 - R, G, B. These pins control the RGB color display on the backlight. Pin 16 is for red, pin 17 is for green, and pin 18 is for blue. To adjust the color of the backlight display pulse width modulation is required to allow for various voltage levels. The intensity of each color can be modified by changing the duty cycle of the signal connected to each pin. Increasing the duty cycle of the signal will cause an increase in the color intensity. Since these pins require PWM they needed to be connected to pins on the microcontroller that have timer modules. As such, pin 16 (Red) is connected to PD3 using the OC2B output of TIMER2, pin 17 (Green) is connected to PD5 using the OC0B output of TIMER0, and pin 18 (Blue) is connected to PD6 using the OC0A output for TIMER0.

Software Integration

- Initializing LCD to use 4-bit interface, enabling PWM mode on TIMER2 and TIMER0 on the microcontroller to modify LCD backlight

```
17:  void lcd_init(void) {
18:      // Set output pins
19:      DDRB |= LCD_DATA_BITS;
20:      DDRD |= LCD_RS_BIT;
21:      DDRB |= LCD_EN_BIT;
22:
23:      // From HD44780 datasheet figure 24 on page 46
24:      _delay_ms(50);  // Delay at least 40ms after 2.7V is reached
25:      PORTD &= ~LCD_RS_BIT;   // Set to command mode
27:      lcd_writenibble(0x03);
28:      _delay_ms(5);           // Delay at least 4ms
29:      lcd_writenibble(0x03);
30:      _delay_ms(5);           // Delay at least 4ms
31:      lcd_writenibble(0x03);
32:      _delay_us(120);         // Delay at least 100us
33:      lcd_writenibble(0x03);
34:
35:      lcd_writenibble(0x02);  // Use 4-bit interface
36:      _delay_ms(2);           // Delay at least 2ms
38:      lcd_writecommand(0x28); // Function Set: 4-bit interface, 2
lines, 5x8 chars
40:      uint8_t display_cmd = 0x08;
```

```
43:     display_cmd |= 0x04; // Enable display on
47:     lcd_writecommand(display_cmd); // Set up display mode
48:     lcd_writecommand(0x06);        // Set Entry Mode
49:
50:     // Setup RGB backlight
51:     DDRD |= (LCD_R_BIT | LCD_G_BIT | LCD_B_BIT);
52:
53:     // Enable PWM for TIMER2 for R
54:     TCCR2A |= (1 << WGM01) | (1 << WGM00);
55:     TCCR2B |= (1 << CS22);   // Prescaler 64
56:     TCCR2A |= (1 << COM2B1); // R
57:
58:     // Enable PWM on TIMER0 for G and B
59:     TCCR0A |= (1 << WGM01) | (1 << WGM00);
60:     TCCR0B |= (1 << CS01) | (1 << CS00); // Prescaler 64
61:     TCCR0A |= (1 << COM0B1); // G
62:     TCCR0A |= (1 << COM0A1); // B
63:
64:     lcd_set_rgb(LCD_COLOR_WHITE);   //Set white backlight
65:     lcd_clear();                    // Clear the LCD screen
66: }
```

- Moving the cursor to a specific point on the LCD by calculating the address of the desired position

```
88: void lcd_moveto(uint8_t row, uint8_t col) {
89:     uint8_t pos;
90:     pos = row_offset(row) | col;
91:     lcd_writecommand(0x80 | pos); // Send move command
92: }
172: uint8_t row_offset(uint8_t row) {
173:     switch(row) {
174:         case 0:
175:             return 0x00;
176:         case 1:
177:             return 0x40;
178:         case 2:
179:             return 0x14;
180:         case 3:
181:             return 0x54;
182:     }
183:     return 0x00;
184: }
```

- Sending commands/data to the LCD

```
139: void lcd_writecommand(uint8_t cmd) {
```

```
140:      PORTB &= ~LCD_RS_BIT;       // Set to command mode
141:      lcd_writenibble(cmd >> 4);  // Send upper 4 bits
142:      lcd_writenibble(cmd);       // Send lower 4 bits
143:  }

148:  void lcd_writedata(uint8_t dat) {
149:      PORTB |= LCD_RS_BIT;        // Set to data mode
150:      lcd_writenibble(dat >> 4);  // Send upper 4 bits
151:      lcd_writenibble(dat);       // Send lower 4 bits
152:  }

157:  void lcd_writenibble(uint8_t lcdbits) {
158:      lcdbits &= 0x0F;
159:      PORTB &= ~LCD_DATA_BITS;
160:      PORTB |= (lcdbits << LCD_DATA_OFFSET);
161:
162:      //Send enable pulse
163:      PORTD |= LCD_EN_BIT;
164:      _delay_us(1);
165:      PORTD &= ~LCD_EN_BIT;
166:      _delay_us(50);
167:  }
```

- Updating the LCD's backlight color by changing the corresponding OCR registers to change the PWM pulse width

```
//From lcd.h
 17:  #define LCD_COLOR_WHITE  255, 255, 255
 18:  #define LCD_COLOR_YELLOW 255, 255, 100
 19:  #define LCD_COLOR_RED    255, 0,   0

//From lcd.c
122:  void lcd_set_rgb(uint8_t r, uint8_t g, uint8_t b) {
123:      OCR2B = r;
124:      OCR0B = g;
125:      OCR0A = b;
126:  }
```

- Not all of the information can be displayed neatly at once so we rotate what information gets displayed every 20 seconds. The top line always displays current date and time, the middle two rows switch between current location and elapsed time, and the last row rotates between speed, altitude, and direction.
- Code to determine what information gets displayed on the LCD and when rows on the LCD rotate to display other fields

```
//From smart_bike.c
```

```
102:  void lcd_update() {
103:      line_len = gps_readline(last_line);
104:      int8_t result = gps_parse(last_line, line_len);
105:      if(result == -1) { // Over a minute since last valid message
106:          if(line12_displayed == 0) { // If showing location switch
to elapsed time
107:              line12_displayed = 1;
108:              line12_counter = 0;
109:          }
110:          display_elapsed();
111:          display_wait();
112:      } else {
113:          if(result & VALID_LINE_0) {
114:              display_time();
115:          }
116:
117:          if(line12_displayed == 0) {
118:             if((!(result & VALID_LINE_1)||!(result & VALID_LINE_2)))
     { // Don't update LCD if error occured when parsing the last message
119:                  line12_counter++;
120:              } else {
121:                  display_location();
122:              }
123:          } else {
124:              display_elapsed();
125:          }
126:          if(line12_counter == LINE_CHANGE_INTERVAL) { // Toggle
between whether location or elapsed time is displayed
127:              line12_counter = 0;
128:              line12_displayed ^= 1;
129:          }
130:
131:          if(result & VALID_LINE_3) {
132:              if((line3_displayed == 0 && !(result & VALID_ALT)) ||
133:                  (line3_displayed == 1 && !(result & VALID_SPD))||
134:                  (line3_displayed == 2 && !(result & VALID_DIR))) {
135:                  line3_counter++;
136:              } else {
137:                  display_misc();
138:              }
139:              if(line3_counter == LINE_CHANGE_INTERVAL) { // Toggle
between whether altitude, speed, or direction are displayed
140:                  line3_counter=0;
141:                  line3_displayed++;
142:                  if(line3_displayed == 3) line3_displayed = 0;
143:              }
144:          } else {
```

```
145:                display_wait();
146:            }
147:        }
148:    }
```

```
04/01/20   16:00 GMT
    34°  1.2356' N
   118° 17.3638' W
Altitude:   62.8 m
```

```
04/01/20   16:00 GMT
Elapsed Time:
      00:15:24
Speed:      10.2 mph
```

```
04/01/20   16:00 GMT
Elapsed Time:
      00:15:24
Direction:    NW
```

Figure 11: Example LCD display layouts

Maxbotix MB1010 - Ultrasonic Rangefinder LV-MaxSonar-EZ1

Description

Figure 12: Maxbotix MB1010 - Ultrasonic Rangefinder LV-MaxSonar-EZ1

This rangefinder uses a transducer to emit ultrasonic pulses and determine the distance from the time it takes to receive the signal back. It can determine distances from 6 to 254 inches with a resolution of 1 inch. It can output the distance as pulse width, analog voltage, and RS232 serial. Because it can sample at 20 Hz, we will be using analog output and perform an ADC conversion to be able to sample when desired on the microcontroller side without interrupts.

Hardware Pinouts

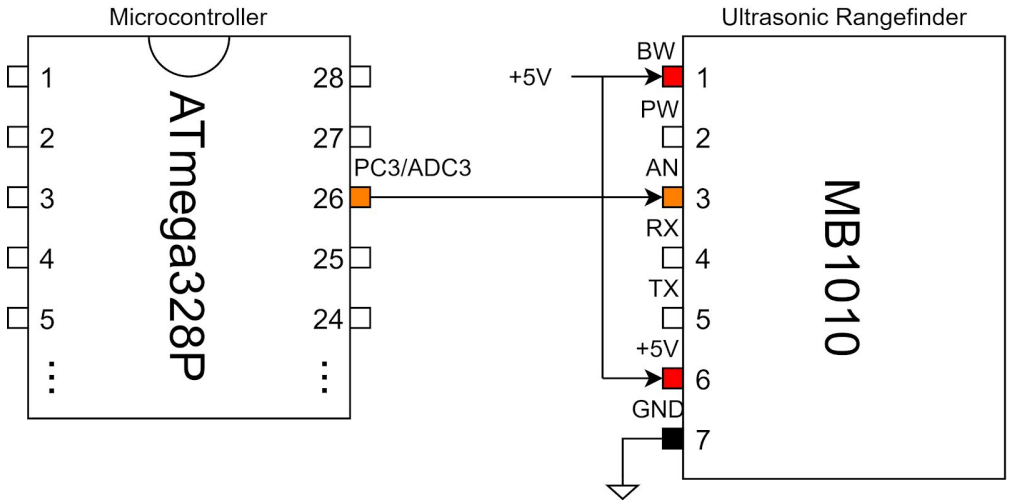

Figure 13: Schematic of Ultrasonic Rangefinder

Pin 1 - BW. This pin enables serial output when low or unconnected. It will be connected to the 5V source to prevent serial output.

Pin 2 - PW. This pin outputs a pulse width wave to represent the distance. Every 147 microsecond length represents an inch. For this project we use the analog result so this pin is left unconnected.

Pin 3 - AN. This pin outputs an analog voltage. With the 5V supply, every 9.8mV represents an inch. The ranger internally buffers this output with the most recent data. This will be connected to the microcontroller for ADC readings.

Pin 4 - RX. This pin is internally held high. If pulled low, it stops measuring. Individual samples can be taken by raising it high for more than 20 uS. For this project there is no need to disable the ultrasonic measurements because constant readings are desired so this pin will be unconnected.

Pin 5 - TX. This pin delivers asynchronous serial output when BW is held low or floating. It will be unconnected for this project.

Pin 6 - +5V. This pin provides the power. The ranger operates anywhere from 2.5V - 5.5V. It will be connected directly to the +5V bus line on the board.

Pin 7 - GND. This pin is connected to ground.

Software Integration

- Taking a 10-bit ADC sample and converting to number of inches
- The analog value from the sensor increments by 1/512*Vcc for each inch so dividing the 10-bit ADC sample by 2 converts the result to inches

```
//From adc.c
 32:  uint16_t sonar_reading() {
 33:      return adc_sample(SONAR_CHAN)/2;
 34:  }
```

- Determining if the distance is below a given threshold to update the lcd and buzzer

```
//From smart_bike.c
 15:  #define RED_THRESH      10*12 //Set Red warning at 10 feet
 16:  #define YELLOW_THRESH   20*12 //Set Yellow warning at 20 feet
 17:  #define BUZZER_BIT (1 << PB1)
 79:  void sonar_update() {
 80:      uint16_t distance = sonar_reading();
 81:      if(distance < RED_THRESH) {
 82:          if(lcd_color != 2) { // Only change if not already red
 83:              lcd_set_rgb(LCD_COLOR_RED);
 84:              lcd_color = 2;
 85:          }
 86:          PORTB |= BUZZER_BIT;
 87:      } else if(distance < YELLOW_THRESH) {
 88:          if(lcd_color != 1) { // Only change if not already yellow
 89:              lcd_set_rgb(LCD_COLOR_YELLOW);
 90:              lcd_color = 1;
 91:          }
 92:          PORTB &= ~BUZZER_BIT;
 93:      } else {
 94:          if(lcd_color != 0) { // Only change if not already white
 95:              lcd_set_rgb(LCD_COLOR_WHITE);
 96:              lcd_color = 0;
 97:          }
 98:          PORTB &= ~BUZZER_BIT;
 99:      }
100:  }
```

Adafruit 2748 - Analog Light Sensor ALS-PT19 & LEDs

Description

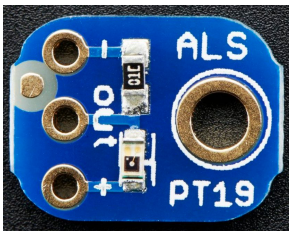

Figure 14: Adafruit 2748 - Analog Light Sensor ALS-PT19

This light sensor utilizes a phototransistor and voltage divider to determine light intensity. Its response is fairly linear and responds to the spectrum of light viewable by the human eye.

Figure 15: LED

The LEDs are used to simulate a bike light that automatically turns on or off based on the ambient light. 2 LEDs in series are recommended to signify a front and back light, but multiple LEDs can also be added in series as well.

Hardware Pinouts

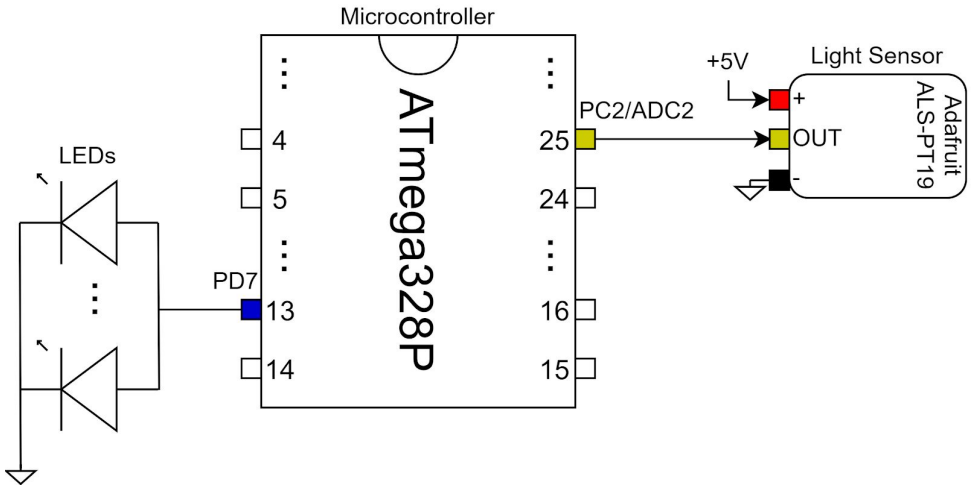

Figure 16: Schematic of Light Sensor and LEDs

Pin + - Vcc. This pin provides the power. It can work anywhere from 0.5V to 6V. It will be connected with the +5V bus line for the best definition from the ADC reading.

Pin OUT - Vout. This pin outputs an analog voltage linearly related to light intensity. The quick response time of the sensor allows accurate real time sensing. This is connected to the microcontrollers PC2 pin to allow for ADC conversions.

Pin – - GND. This pin is connected to ground.

LEDs:
Cathodes are connected to ground, anodes are connected in series to the PD7 pin of the microcontroller.

Software Integration

- 10-bit ADC sample of the light sensor is taken

```
//From adc.c
 28:  uint16_t light_reading() {
 29:      return adc_sample(LIGHT_CHAN);
 30:  }
```

- Logic for determining when to turn lights on or off. Different thresholds are used depending on whether the lights are already on or off to prevent sudden flickering

```
//From smart_bike.c
 11:  #define DARK_THRESH  400 // When lights are off, turn them on when
below this threshold
 12:  #define LIGHT_THRESH 600 // When lights are on, turn them off when
above this threshold
 13:  #define LED_BIT (1 << PD7)
 67:  void light_update() {
 68:      uint16_t light_lvl = light_reading();
 69:      //Two separate thresholds are used to avoid LED from flickering
if light level fluctuates around single threshold
 70:      if((PORTD & LED_BIT)==0 && light_lvl < DARK_THRESH) {
 71:          //Turn on lights when ambient light is dark enough
 72:          PORTD |= LED_BIT;
 73:      } else if((PORTD & LED_BIT) && light_lvl > LIGHT_THRESH) {
 74:          //Turn off lights when ambient light is bright enough
 75:          PORTD &= ~LED_BIT;
 76:      }
 77:  }
```

Digikey 102-1285 - Buzzer CX-0905C

Description

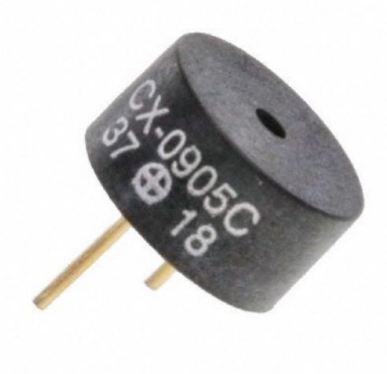

Figure 17: Digikey 102-1285 - Buzzer CX-0905C

This is a magnetic buzzer that is rated to output a sound at 2730 Hz frequency. The operating voltage for the buzzer is between 4-7V.

Hardware Pinouts

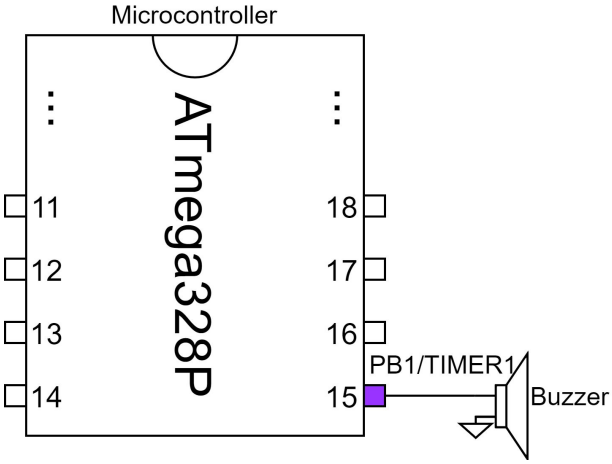

Figure 18: Schematic of Buzzer

Pin + - Vcc. This pin provides the power. It is rated for a 5V input. It will be connected to PB1 so that the microcontroller can toggle it on and off.

Pin – - GND. This pin is connected to ground.

Software Integration

See **Software Integration** section for the Ultrasonic Rangefinder above

Cost Analysis

Manufacturer	**Model Name/ Product ID**	**Description**	**Cost (Single)**	**Cost (Bulk)**
Atmel	ATmega328P	Microcontroller	$2.08	$1.73
Adafruit	499	20x4 Character LCD Screen with RGB-backlight	$24.95	$19.96
Adafruit	746	GPS Module	$34.95	$31.96
Maxbotix	MB1010 LV-MaxSonar-EZ	Ultrasonic Range Finder	$24.95	$18.06
Adafruit	2748	Analog Light Sensor	$2.50	$2.00
CUI	CX-0905C	Buzzer	$1.90	$1.05
Texas Instruments	CD40109BE	Low-to-High Level Shifter	$0.17	$0.15
Misc.		LEDs, wires, solder, etc.	~$8.00	~$8.00
Total			**$99.50**	**$82.91**

Table 1: Cost Breakdown

Table 1 outlines the cost of all the parts used in our product. This chart did not include the fee for labor. The assembly fee depends on the material and the technique we use to create the box containing our project. In a final version of the product, the bike light would not be cheap LEDs, so the cost would be greater. In estimate, the additional fees should be no more than $15 in bulk order. If this product was produced commercially, we would use parts that are significantly cheaper because they are not breakout boards or hobbyist parts.

Future Additions

There are two main features that we would like to add in the future: buttons for more user control and a solar panel.

While the product can operate without buttons, the product could be bettered with them. Possible features include the ability to change time zones, set distance goals and timers, and personal information to calculate approximate heart rate zones and calories expended like exercise bikes.

Because the product is meant to be attached to the bike, it must be battery powered. A solar panel which can recharge the batteries would be a feature that greatly improves the user's experience. Replacing batteries is costly and annoying.

Source Code

Full source code was submitted along with the the report and is also available on GitHub at:

github.com/mromo681/EE459_Smart_Bike

Division of Work

Task	**Martin**	**Cameron**	**Qingyuan**
System Design	33.33%	33.33%	33.33%
Hardware Design	33.33%	33.33%	33.33%
Software Design	90%	5%	5%
Documentation	33.33%	33.33%	33.33%
Project Report (oral)	40%	30%	30%
Project Report (written)	40%	30%	30%
Signature