|  | Detector Building C |
| --- | --- |
| Specification Sheet – Version 1 |

| **SCHOOL NAME** | Bothell High School |
| --- | --- |
| **TEAM NAME** | Bothell Blue |
| **TEAM NUMBER** | C05 |
| **STUDENT NAME(S)** | Alexander Metzger |
| Samarth Venkatesh |

| **Instructions:** In the following sections, follow the instructions for either inserting photographs or typing your responses. Please type your responses only in the white boxes. Any responses in gray boxes will not be read. To ensure any formatting, images, and/or figures (e.g. arrows, boxes) you add are preserved, please save and upload your completed specification sheet as a PDF with filename in the format *TeamNumber\_DetectorBuilding\_SpecSheet.pdf*.  For the event **Detector Building C**, you will record your device testing video with the “test” in Scilympiad open, and you will document your device and results from your device testing in this specification sheet. You will submit this specification sheet with your device testing video via a Google Form. |
| --- |

| Device Photographs |
| --- |
| In the space below, insert photographs of your Device. You should include photographs that show the entire Device from multiple angles (top, each side, and isometric views) as well as ones that are close-up to show details so that the Event Supervisor can verify the compliance of your Device with the event’s rules. |
|  |

| Design Log |
| --- |
| In the space below, insert images (e.g. scans, photographs, screenshots) of your entire Design Log. |
| Detector.py import RPi.GPIO as GPIO  import LCD\_I2C  import NAU7802\_I2C  import smbus2  import time  # Create the bus  bus = smbus2.SMBus(1)  # initialize the NAU7802 ADC  adc = NAU7802\_I2C.NAU7802()  if adc.begin(bus):  print("Connected ADC!\n")  print("Calibrated ADC!\n")  else:  print("Can't find the adc, exiting ...\n")  exit()  # initialize LCD display + verify it works  lcd = LCD\_I2C.lcd(bus)  lcd.lcd\_display\_string("Calibrating ADC", 1)  # Constants  R1 = 214.3  R2 = 215.8  R3 = 9770  VIN = 3.29  VREF = VIN / 2  VDEVIATION = (0.0166036203409, -0.0384329033931, 1.00958174262, 0.010941954783)  # unknown voltage divider seems slightly off but it varies with voltage?  EPSILON = 1e-19  ADC\_MAX = 2\*\*23  SAMPLES = 8 # how many adc readings are averaged  H = 416.125  P = -554.131  D = -0.918042  CONVERSION = lambda volt: min(max(0, H / (volt + D) + P),5040)  def main():  # LEDs  GPIO.setmode(GPIO.BCM)  RedLED = LED(22, 0, 1600)  BlueLED = LED(27, 1600, 3300)  GreenLED = LED(17, 3300, float('Inf'))  LEDs = [RedLED, BlueLED, GreenLED]    # cool lightshow to verify LEDs work  letThereBeLight(LEDs)  minVout = float('Inf'))  try:  print('Reading detector values')  while True:  # Read the ADC for SAMPLES and return the average reading  value = adc.getAverage(SAMPLES)  # convert ADC value to the desired values (voltage, resistance, conductivity)  vout, resistance, conductance = interpretAdc(value)  if vout < minVout: minVout = vout  else: minVout = vout  conductivity = CONVERSION(vout)    # print values  printValues(value, vout, resistance, conductivity)    # show values on LCD  lcd.lcd\_clear()  lcd.lcd\_display\_string("Voltage (adc value):", 1)  lcd.lcd\_display\_string(str(round(vout,5)) + " (" + str(value) + ")", 2)  lcd.lcd\_display\_string("Concentration (ppm):", 3)  lcd.lcd\_display\_string(str(round(conductivity,3)), 4)    # activate LEDs  for l in LEDs:  l.update(conductivity)    # Pause for a second  time.sleep(1)  finally:  print("Clean Exit \n")  GPIO.cleanup() # ensure clean exit  def printValues(value, vout, resistance, conductivity):  print('ADC: ' + str(value) + #' ' + format(int(value), 'b') +  ' | Vol: ' + str(round(vout,5)) +  ' | Res: ' + str(round(resistance)) +  ' | Con: ' + str(round(conductivity,4))  )  def interpretAdc(value):  return interpretWheatstone(value)  def interpretWheatstone(value):  vout = VREF \* value / ADC\_MAX  v12 = VIN \* R2 / (R1 + R2) # voltage out at known voltage divider  vout += VDEVIATION[0] \* (vout + v12) \*\* 3 + VDEVIATION[1] \* (vout + v12) \*\* 2 + VDEVIATION[2] \* (vout + v12) + VDEVIATION[3] - vout - v12  # experimentally determined adjustment factor lol    print('known voltage divider: ' + str(v12) +  ' vout: ' + str(vout)  )  v3x = vout + v12 # voltage out at unknown voltage divider  resistance = R3 \* (R2 \* VIN + (R1 + R2) \* vout) / (R1 \* VIN - (R1 + R2) \* vout + EPSILON)  conductance = (R1 \* VIN - (R1 + R2) \* vout) / (R3 \* (R2 \* VIN + (R1 + R2) \* vout) + EPSILON)  return v3x, resistance, conductance  def letThereBeLight(LEDs):  for l in LEDs:  l.on()  time.sleep(0.3)  l.off()  for l in LEDs:  l.on()  time.sleep(0.5)    class LED():  def \_\_init\_\_(self, pin, lower, upper):  self.pin = pin  self.lower = lower  self.upper = upper  GPIO.setup(pin, GPIO.OUT, initial=GPIO.LOW)    def on(self):  GPIO.output(self.pin, GPIO.HIGH)    def off(self):  GPIO.output(self.pin, GPIO.LOW)    def update(self, value):  if value >= self.lower and value < self.upper:  self.on()  else:  self.off()    # runs main() when program is run from terminal  if \_\_name\_\_ == '\_\_main\_\_':  main() NAU7802\_I2C.py '''  RPI code to read the NAU7802 ADC values.  The NAU7802 is a 24-Bit Dual-Channel ADC  that uses the I2C protocol. Datasheet:  https://www.nuvoton.com/resource-files/NAU7802%20Data%20Sheet%20V1.7.pdf  Based on similar (arduino) code for a scale:  https://github.com/sparkfun/SparkFun\_Qwiic\_Scale\_NAU7802\_Arduino\_Library  '''  import smbus2  import time  ###########################################  # Constants  ###########################################  """ Address """  DEVICE\_ADDRESS = 0x2A  """ Register Map """  NAU7802\_PU\_CTRL = 0x00  NAU7802\_CTRL1 = 0x01  NAU7802\_CTRL2 = 0x02  NAU7802\_OCAL1\_B2 = 0x03 # CH1 OFFSET Calibration[23:16]  NAU7802\_OCAL1\_B1 = 0x04 # CH1 OFFSET Calibration[15:8]  NAU7802\_OCAL1\_B0 = 0x05 # CH1 OFFSET Calibration[7:0]  NAU7802\_GCAL1\_B3 = 0x06 # CH1 GAIN Calibration[31:24]  NAU7802\_GCAL1\_B2 = 0x07 # CH1 GAIN Calibration[23:16]  NAU7802\_GCAL1\_B1 = 0x08 # CH1 GAIN Calibration[15:8]  NAU7802\_GCAL1\_B0 = 0x09 # CH1 GAIN Calibration[7:0]  NAU7802\_OCAL2\_B2 = 0x0A # CH2 OFFSET Calibration[23:16]  NAU7802\_OCAL2\_B1 = 0x0B # CH2 OFFSET Calibration[15:8]  NAU7802\_OCAL2\_B0 = 0x0C # CH2 OFFSET Calibration[7:0]  NAU7802\_GCAL2\_B3 = 0x0D # CH2 GAIN Calibration[31:24]  NAU7802\_GCAL2\_B2 = 0x0E # CH2 GAIN Calibration[23:16]  NAU7802\_GCAL2\_B1 = 0x0F # CH2 GAIN Calibration[15:8]  NAU7802\_GCAL2\_B0 = 0x10 # CH2 GAIN Calibration[7:0  NAU7802\_I2C\_CONTROL = 0x11  NAU7802\_ADCO\_B2 = 0x12 # ADC\_OUT[23:16]  NAU7802\_ADCO\_B1 = 0x13 # ADC\_OUT[15:8]  NAU7802\_ADCO\_B0 = 0x14 # ADC\_OUT[7:0]  # No idea why this is like this lol:  NAU7802\_ADC = 0x15 # Shared ADC and OTP 32:24  NAU7802\_OTP\_B1 = 0x16 # OTP 23:16 or 7:0?  NAU7802\_OTP\_B0 = 0x17 # OTP 15:8  NAU7802\_PGA = 0x1B # ?????  NAU7802\_PGA\_PWR = 0x1C # ?????  NAU7802\_DEVICE\_REV = 0x1F # Device Revision Code  """ Bits within the PU\_CTRL register """  NAU7802\_PU\_CTRL\_RR = 0  NAU7802\_PU\_CTRL\_PUD = 1  NAU7802\_PU\_CTRL\_PUA = 2  NAU7802\_PU\_CTRL\_PUR = 3  NAU7802\_PU\_CTRL\_CS = 4  NAU7802\_PU\_CTRL\_CR = 5  NAU7802\_PU\_CTRL\_OSCS = 6  NAU7802\_PU\_CTRL\_AVDDS = 7  """ Bits within the CTRL1 register """  NAU7802\_CTRL1\_GAIN = 2  NAU7802\_CTRL1\_VLDO = 5  NAU7802\_CTRL1\_DRDY\_SEL = 6  NAU7802\_CTRL1\_CRP = 7  """ Bits within the CTRL2 register """  NAU7802\_CTRL2\_CALMOD = 0  NAU7802\_CTRL2\_CALS = 2  NAU7802\_CTRL2\_CAL\_ERROR = 3  NAU7802\_CTRL2\_CRS = 4  NAU7802\_CTRL2\_CHS = 7  """ Bits within the PGA register """  NAU7802\_PGA\_CHP\_DIS = 0  NAU7802\_PGA\_INV = 3  NAU7802\_PGA\_BYPASS\_EN = 4  NAU7802\_PGA\_OUT\_EN = 5  NAU7802\_PGA\_LDOMODE = 6  NAU7802\_PGA\_RD\_OTP\_SEL = 7  """ Bits within the PGA PWR register """  NAU7802\_PGA\_PWR\_PGA\_CURR = 0  NAU7802\_PGA\_PWR\_ADC\_CURR = 2  NAU7802\_PGA\_PWR\_MSTR\_BIAS\_CURR = 4  NAU7802\_PGA\_PWR\_PGA\_CAP\_EN = 7  """ Allowed Low dropout regulator voltages """  NAU7802\_LDO\_2V4 = 0b111  NAU7802\_LDO\_2V7 = 0b110  NAU7802\_LDO\_3V0 = 0b101  NAU7802\_LDO\_3V3 = 0b100  NAU7802\_LDO\_3V6 = 0b011  NAU7802\_LDO\_3V9 = 0b010  NAU7802\_LDO\_4V2 = 0b001  NAU7802\_LDO\_4V5 = 0b000  """ Allowed gains """  NAU7802\_GAIN\_128 = 0b111  NAU7802\_GAIN\_64 = 0b110  NAU7802\_GAIN\_32 = 0b101  NAU7802\_GAIN\_16 = 0b100  NAU7802\_GAIN\_8 = 0b011  NAU7802\_GAIN\_4 = 0b010  NAU7802\_GAIN\_2 = 0b001  NAU7802\_GAIN\_1 = 0b000  """ Allowed samples per second """  NAU7802\_SPS\_320 = 0b111  NAU7802\_SPS\_80 = 0b011  NAU7802\_SPS\_40 = 0b010  NAU7802\_SPS\_20 = 0b001  NAU7802\_SPS\_10 = 0b000  """ Select between channel values """  NAU7802\_CHANNEL\_1 = 0  NAU7802\_CHANNEL\_2 = 1  """ Calibration state """  NAU7802\_CAL\_SUCCESS = 0  NAU7802\_CAL\_IN\_PROGRESS = 1  NAU7802\_CAL\_FAILURE = 2  ###########################################  # Classes  ###########################################  class NAU7802:  """ Class to communicate with the NAU7802 """  \_i2cPort: smbus2.SMBus = None  # Sets up the NAU7802 for basic function  # If initialize is true (or not specified), default init and calibration is performed  # If initialize is false, then it's up to the caller to initalize and calibrate  # Returns true upon completion  def begin(self, wire\_port: smbus2.SMBus = smbus2.SMBus(1), initialize: bool = True) -> bool:  """ Check communication and initialize sensor """  # Get user's options  self.\_i2cPort = wire\_port  # Check if the device ACK's over I2C  if not self.isConnected():  # There are rare times when the sensor is occupied and doesn't ACK. A 2nd try resolves this.  if not self.isConnected():  return False  result = True # Accumulate a result as we do the setup  if initialize:  result &= self.reset() # Reset all registers  result &= self.powerUp() # Power on analog and digital sections  result &= self.setGain(NAU7802\_GAIN\_1) # Set gain to 1  result &= self.setSampleRate(NAU7802\_SPS\_10) # Set samples per second to 10  result &= self.setRegister(NAU7802\_ADC, 0x30) # Turn off CLK\_CHP. From 9.1 power on sequencing.  result &= self.calibrateAFE() # Re - cal analog frontend when we change gain, sample rate, or channel  return result  def isConnected(self) -> bool:  """ Returns true if device ACK's at the I2C address """  try:  self.\_i2cPort.read\_byte(DEVICE\_ADDRESS)  return True # All good  except OSError:  return False # Sensor did not ACK  def available(self) -> bool:  """ Returns true if Cycle Ready bit is set (conversion is complete) """  return self.getBit(NAU7802\_PU\_CTRL\_CR, NAU7802\_PU\_CTRL)  def getReading(self) -> int:  """ Returns 24 bit reading. Assumes CR Cycle Ready bit  (ADC conversion complete) has been checked by .available() """  try:  value\_list = self.\_i2cPort.read\_i2c\_block\_data(DEVICE\_ADDRESS, NAU7802\_ADCO\_B2, 3)  except OSError:  return False # Sensor did not ACK  value = int.from\_bytes(value\_list, byteorder='big', signed=True)  return value  def getAverage(self, average\_amount: int) -> int:  """ Return the average of a given number of readings """  total = 0  samples\_acquired = 0  start\_time = time.time()  while samples\_acquired < average\_amount:  if self.available():  total += self.getReading()  samples\_acquired += 1  if time.time() - start\_time > 1.0:  return 0 # Timeout - Bail with error  time.sleep(0.001)  total /= average\_amount  return total  def setGain(self, gain\_value: int) -> bool:  """ Set the gain.x1, 2, 4, 8, 16, 32, 64, 128 are available """  if gain\_value > 0b111:  gain\_value = 0b111 # Error check  value = self.getRegister(NAU7802\_CTRL1)  value &= 0b11111000 # Clear gain bits  value |= gain\_value # Mask in new bits  return self.setRegister(NAU7802\_CTRL1, value)  def setLDO(self, ldo\_value: int) -> bool:  """ Set the on-board Low - Drop - Out voltage regulator to a given value.  2.4, 2.7, 3.0, 3.3, 3.6, 3.9, 4.2, 4.5 V are available """  if ldo\_value > 0b111:  ldo\_value = 0b111 # Error check  # Set the value of the LDO  value = self.getRegister(NAU7802\_CTRL1)  value &= 0b11000111 # Clear LDO bits  value |= ldo\_value << 3 # Mask in new LDO bits  self.setRegister(NAU7802\_CTRL1, value)  return self.setBit(NAU7802\_PU\_CTRL\_AVDDS, NAU7802\_PU\_CTRL) # Enable the internal LDO  def setSampleRate(self, rate: int) -> bool:  """ Set the readings per second. 10, 20, 40, 80, and 320 samples per second is available """  if rate > 0b111:  rate = 0b111 # Error check  value = self.getRegister(NAU7802\_CTRL2)  value &= 0b10001111 # Clear CRS bits  value |= rate << 4 # Mask in new CRS bits  return self.setRegister(NAU7802\_CTRL2, value)  def setChannel(self, channel\_number: int) -> bool:  """ Select between 1 and 2 """  if channel\_number == NAU7802\_CHANNEL\_1:  return self.clearBit(NAU7802\_CTRL2\_CHS, NAU7802\_CTRL2) # Channel 1 (default)  else:  return self.setBit(NAU7802\_CTRL2\_CHS, NAU7802\_CTRL2) # Channel 2  def calibrateAFE(self) -> bool:  """ Synchronous calibration of the analog front end of the NAU7802.  Returns true if CAL\_ERR bit is 0 (no error) """  self.beginCalibrateAFE()  return self.waitForCalibrateAFE(1000)  def beginCalibrateAFE(self) -> None:  """ Begin asynchronous calibration of the analog front end of the NAU7802.  Poll for completion with calAFEStatus() or wait with waitForCalibrateAFE(). """  self.setBit(NAU7802\_CTRL2\_CALS, NAU7802\_CTRL2)  def waitForCalibrateAFE(self, timeout\_ms: int = 0) -> bool:  """ Wait for asynchronous AFE calibration to complete with optional timeout. """  timeout\_s = timeout\_ms/1000  begin = time.time()  cal\_ready = self.calAFEStatus()  while cal\_ready == NAU7802\_CAL\_IN\_PROGRESS:  if (timeout\_ms > 0) & ((time.time() - begin) > timeout\_s):  break  time.sleep(0.001)  cal\_ready = self.calAFEStatus()  if cal\_ready == NAU7802\_CAL\_SUCCESS:  return True  else:  return False  def calAFEStatus(self) -> int:  """ Check calibration status. """  if self.getBit(NAU7802\_CTRL2\_CALS, NAU7802\_CTRL2):  return NAU7802\_CAL\_IN\_PROGRESS  if self.getBit(NAU7802\_CTRL2\_CAL\_ERROR, NAU7802\_CTRL2):  return NAU7802\_CAL\_FAILURE  # Calibration passed  return NAU7802\_CAL\_SUCCESS  def reset(self) -> bool:  """ Resets all registers to Power Of Defaults """  self.setBit(NAU7802\_PU\_CTRL\_RR, NAU7802\_PU\_CTRL) # Set RR  time.sleep(0.001)  return self.clearBit(NAU7802\_PU\_CTRL\_RR, NAU7802\_PU\_CTRL) # Clear RR to leave reset state  def powerUp(self) -> bool:  """ Power up digital and analog sections, ~2 mA """  self.setBit(NAU7802\_PU\_CTRL\_PUD, NAU7802\_PU\_CTRL)  self.setBit(NAU7802\_PU\_CTRL\_PUA, NAU7802\_PU\_CTRL)  # Wait for Power Up bit to be set - takes approximately 200us  counter = 0  while not self.getBit(NAU7802\_PU\_CTRL\_PUR, NAU7802\_PU\_CTRL):  time.sleep(0.001)  if counter > 100:  return False # Error  counter += 1  return True  def powerDown(self) -> bool:  """ Set low power 200 nA mode """  self.clearBit(NAU7802\_PU\_CTRL\_PUD, NAU7802\_PU\_CTRL)  return self.clearBit(NAU7802\_PU\_CTRL\_PUA, NAU7802\_PU\_CTRL)  def setIntPolarityHigh(self) -> bool:  """ Set Int pin to be high when data is ready(default) """  return self.clearBit(NAU7802\_CTRL1\_CRP, NAU7802\_CTRL1) # 0 = CRDY pin is high active (ready when 1)  def setIntPolarityLow(self) -> bool:  """ Set Int pin to be low when data is ready """  return self.setBit(NAU7802\_CTRL1\_CRP, NAU7802\_CTRL1) # 1 = CRDY pin is low active (ready when 0)  def getRevisionCode(self) -> int:  """ Get the revision code of this IC.Always 0x0F. """  revisionCode = self.getRegister(NAU7802\_DEVICE\_REV)  return revisionCode & 0x0F  def setBit(self, bit\_number: int, register\_address: int) -> bool:  """ Mask & set a given bit within a register """  value = self.getRegister(register\_address)  value |= (1 << bit\_number) # Set this bit  return self.setRegister(register\_address, value)  def clearBit(self, bit\_number: int, register\_address: int) -> bool:  """ Mask & clear a given bit within a register """  value = self.getRegister(register\_address)  value &= ~(1 << bit\_number) # Set this bit  return self.setRegister(register\_address, value)  def getBit(self, bit\_number: int, register\_address: int) -> bool:  """ Return a given bit within a register """  value = self.getRegister(register\_address)  value &= (1 << bit\_number) # Clear all but this bit  return bool(value)  def getRegister(self, register\_address: int) -> int:  """ Get contents of a register """  try:  return self.\_i2cPort.read\_byte\_data(DEVICE\_ADDRESS, register\_address)  except OSError:  return -1 # Sensor did not ACK  def setRegister(self, register\_address: int, value: int) -> bool:  """ Send a given value to be written to given address.Return true if successful """  try:  self.\_i2cPort.write\_byte\_data(DEVICE\_ADDRESS, register\_address, value)  return True  except OSError:  return False LCD\_I2C.py # Original code found at:  # https://gist.github.com/DenisFromHR/cc863375a6e19dce359d  """  Compiled, mashed and generally mutilated 2014-2015 by Denis Pleic  Made available under GNU GENERAL PUBLIC LICENSE  # Modified Python I2C library for Raspberry Pi  # as found on http://www.recantha.co.uk/blog/?p=4849  # Joined existing 'i2c\_lib.py' and 'lcddriver.py' into a single library  # added bits and pieces from various sources  # By DenisFromHR (Denis Pleic)  # 2015-02-10, ver 0.1  """  '''  Further Mutilated by Alex Metzger  '''  # i2c bus (0 -- original Pi, 1 -- Rev 2 Pi)  I2CBUS = 1  # LCD Address  ADDRESS = 0x27  import smbus2 as smbus  from time import sleep  class i2c\_device:  def \_\_init\_\_(self, addr, bus=smbus.SMBus(I2CBUS)):  self.addr = addr  self.bus = bus  # self.bus = smbus.SMBus(port)  # Write a single command  def write\_cmd(self, cmd):  self.bus.write\_byte(self.addr, cmd)  sleep(0.0001)  # Write a command and argument  def write\_cmd\_arg(self, cmd, data):  self.bus.write\_byte\_data(self.addr, cmd, data)  sleep(0.0001)  # Write a block of data  def write\_block\_data(self, cmd, data):  self.bus.write\_block\_data(self.addr, cmd, data)  sleep(0.0001)  # Read a single byte  def read(self):  return self.bus.read\_byte(self.addr)  # Read  def read\_data(self, cmd):  return self.bus.read\_byte\_data(self.addr, cmd)  # Read a block of data  def read\_block\_data(self, cmd):  return self.bus.read\_block\_data(self.addr, cmd)  # commands  LCD\_CLEARDISPLAY = 0x01  LCD\_RETURNHOME = 0x02  LCD\_ENTRYMODESET = 0x04  LCD\_DISPLAYCONTROL = 0x08  LCD\_CURSORSHIFT = 0x10  LCD\_FUNCTIONSET = 0x20  LCD\_SETCGRAMADDR = 0x40  LCD\_SETDDRAMADDR = 0x80  # flags for display entry mode  LCD\_ENTRYRIGHT = 0x00  LCD\_ENTRYLEFT = 0x02  LCD\_ENTRYSHIFTINCREMENT = 0x01  LCD\_ENTRYSHIFTDECREMENT = 0x00  # flags for display on/off control  LCD\_DISPLAYON = 0x04  LCD\_DISPLAYOFF = 0x00  LCD\_CURSORON = 0x02  LCD\_CURSOROFF = 0x00  LCD\_BLINKON = 0x01  LCD\_BLINKOFF = 0x00  # flags for display/cursor shift  LCD\_DISPLAYMOVE = 0x08  LCD\_CURSORMOVE = 0x00  LCD\_MOVERIGHT = 0x04  LCD\_MOVELEFT = 0x00  # flags for function set  LCD\_8BITMODE = 0x10  LCD\_4BITMODE = 0x00  LCD\_2LINE = 0x08  LCD\_1LINE = 0x00  LCD\_5x10DOTS = 0x04  LCD\_5x8DOTS = 0x00  # flags for backlight control  LCD\_BACKLIGHT = 0x08  LCD\_NOBACKLIGHT = 0x00  En = 0b00000100 # Enable bit  Rw = 0b00000010 # Read/Write bit  Rs = 0b00000001 # Register select bit  class lcd:  #initializes objects and lcd  def \_\_init\_\_(self, i2cbus = smbus.SMBus(I2CBUS)):  self.lcd\_device = i2c\_device(ADDRESS, i2cbus)  self.lcd\_write(0x03)  self.lcd\_write(0x03)  self.lcd\_write(0x03)  self.lcd\_write(0x02)  self.lcd\_write(LCD\_FUNCTIONSET | LCD\_2LINE | LCD\_5x8DOTS | LCD\_4BITMODE)  self.lcd\_write(LCD\_DISPLAYCONTROL | LCD\_DISPLAYON)  self.lcd\_write(LCD\_CLEARDISPLAY)  self.lcd\_write(LCD\_ENTRYMODESET | LCD\_ENTRYLEFT)  sleep(0.2)  # clocks EN to latch command  def lcd\_strobe(self, data):  self.lcd\_device.write\_cmd(data | En | LCD\_BACKLIGHT)  sleep(.0005)  self.lcd\_device.write\_cmd(((data & ~En) | LCD\_BACKLIGHT))  sleep(.0001)  def lcd\_write\_four\_bits(self, data):  self.lcd\_device.write\_cmd(data | LCD\_BACKLIGHT)  self.lcd\_strobe(data)  # write a command to lcd  def lcd\_write(self, cmd, mode=0):  self.lcd\_write\_four\_bits(mode | (cmd & 0xF0))  self.lcd\_write\_four\_bits(mode | ((cmd << 4) & 0xF0))  # write a character to lcd (or character rom) 0x09: backlight | RS=DR<  # works!  def lcd\_write\_char(self, charvalue, mode=1):  self.lcd\_write\_four\_bits(mode | (charvalue & 0xF0))  self.lcd\_write\_four\_bits(mode | ((charvalue << 4) & 0xF0))    # put string function with optional char positioning  def lcd\_display\_string(self, string, line=1, pos=0):  if line == 1:  pos\_new = pos  elif line == 2:  pos\_new = 0x40 + pos  elif line == 3:  pos\_new = 0x14 + pos  elif line == 4:  pos\_new = 0x54 + pos  self.lcd\_write(0x80 + pos\_new)  for char in string:  self.lcd\_write(ord(char), Rs)  # clear lcd and set to home  def lcd\_clear(self):  self.lcd\_write(LCD\_CLEARDISPLAY)  self.lcd\_write(LCD\_RETURNHOME)  # define backlight on/off (lcd.backlight(1); off= lcd.backlight(0)  def backlight(self, state): # for state, 1 = on, 0 = off  if state == 1:  self.lcd\_device.write\_cmd(LCD\_BACKLIGHT)  elif state == 0:  self.lcd\_device.write\_cmd(LCD\_NOBACKLIGHT)  # add custom characters (0 - 7)  def lcd\_load\_custom\_chars(self, fontdata):  self.lcd\_write(0x40)  for char in fontdata:  for line in char:  self.lcd\_write\_char(line) |

| Construction Parameters | | |
| --- | --- | --- |
| **Respond to the following statements for your Device by typing either “TRUE” or “FALSE” in the space to the right of each statement.** | | |
| 3.a. | Our Device is built using a microcontroller or microcontroller board, a display, LED lights, and a participant-built sensor/probe. The sensor produces a voltage which varies according to the concentration of the water. WiFi/Internet connection is not used at any time during competition. | TRUE |
| 3.b. | Our sensor is student-constructed from fundamental electronic components such as resistors, capacitors, wire, and DIP package integrated circuits. All circuits are assembled on a breadboard. No preassembled integrated circuit PCB boards are used. The sensor and wires/cables, together, are ≥ 30.0 cm in length, and narrow enough to fit through an opening of 7.0 cm and the end is immersible up to 5.0 cm in water. | TRUE |
| 3.d. | Our Device has a digital display that clearly shows voltage, and the salt concentration in ppm to the nearest unit value. If a laptop is used for display purposes, it is not used for the Written Test portion. | TRUE |
| 3.e. | Our Device is able to indicate the specific concentration zone using three separate LEDs – one red, one green, and one blue. RGB LEDs, if used, is wired for only one color. | TRUE |
| 3.f. | We did not use electrical outlets at any time during the competition. If our Device is not powered by a connected laptop or calculator, our Device is powered by commercially available batteries. | TRUE |
| 3.g. | Our Device is clearly labeled with our team’s name and team number. | TRUE |
| 5.Part I.a. | Once my partner and I entered the event area, we did not leave or receive outside assistance, materials, or communication. | TRUE |

| Device Dimensions | | |
| --- | --- | --- |
| 2.e. | **Device length:** In the space below, insert a photograph of a ruler held up to your Device, showing the measurement of the length of the sensor and wires/cables together. | |
|  | | |
| 2.e. | Is the length, in centimeters, of your Device’s sensor and wires/cables together ≥ 30.0 cm? | YES |
| 2.e. | **Device power:** In the space below, insert a photograph or schematic showing how your Device is powered. Specify input voltages and the output voltage of your probe. | |
| The primary power supply of the device is two commercial 3.7 V batteries in series. They are fed through three 5 V voltage regulators before reaching the device so the power supplied never exceeds 5 V. This 5 V potential difference is applied to the 5 V rail of the Raspberry Pi (microcontroller) and powers the circuitry.  The secondary power supply is from a laptop through a Micro USB to facilitate screen sharing (i.e. using the laptop as a display). As per USB standards, this power never exceeds 5 V so total power to the device never exceeds 5 + 5 = 10 V.  The probe is connected to the 3.3 V rail on the Raspberry Pi which is voltage regulated so the input and output voltage of the probe never exceeds 3.3 V. | | |
| 2.e. | What is the total input voltage, in volts, supplied to your Device by your laptop or batteries? | 10 V |
| 2.e. | What is the expected output voltage, in volts, of your Device’s probe? | 3.3 V |

| Device Testing | |
| --- | --- |
| **You will complete this section when you test your device during the tournament.** | |
| **Station 1:** What was the voltage, to the nearest whole number, displayed by your Device? |  |
| **Station 1:** What was the concentration (ppm), to the nearest whole number, displayed by your Device? |  |
| **Station 1:** What LED colors were displayed by your Device? |  |
| **Station 2:** What was the voltage, to the nearest whole number, displayed by your Device? |  |
| **Station 2:** What was the concentration (ppm), to the nearest whole number, displayed by your Device? |  |
| **Station 2:** What LED colors were displayed by your Device? |  |
| **Station 3:** What was the voltage, to the nearest whole number, displayed by your Device? |  |
| **Station 3:** What was the concentration (ppm), to the nearest whole number, displayed by your Device? |  |
| **Station 3:** What LED colors were displayed by your Device? |  |
| **Station 4 (State Only):** What was the voltage, to the nearest whole number, displayed by your Device? |  |
| **Station 4 (State Only)::** What was the concentration (ppm), to the nearest whole number, displayed by your Device? |  |
| **Station 4 (State Only):** What LED colors were displayed by your Device? |  |

*You have reached the end of this document. Please do not add anything beyond this point.*