Arduino Engine Monitor

Concept to Implementation

Overview

Analog Input Ports

Like many projects, this project started with basic needs; that is, to monitor various parameters around a Rotax engine which would be used in an Europa Airplane Kit. As the engine was purchased without instrumentation, the basic needs would need to be addressed: oil pressure, coolant temperature, and other running characteristics indicative of an airplane piston engine – in code, these are defined as:

```
#define WaterTemp_AnalogIn        A0  // Coolant Temperature
#define OilTemp_AnalogIn          A1  // Oil Temperature
#define OilPress_AnalogIn         A2  // Oil Pressure
#define FrontCylTemp_AnalogIn     A3  // Cylinder Head Front Temperature 1
#define RearCylTemp_AnalogIn      A4  // Cylinder Head Rear Temperature 2
```

The above requirements are easily implemented on the chosen Atmel microcontroller (Arduino Mega2560) using the built-in 10-bit analog to digital conversion circuitry. The electrical implementation is essentially a sensor (“sender”) which varies resistance with the physical quantity being monitored. The sensor is connected in series with a precision resistor and the resulting two resistances create a simple circuit that the microcontroller can monitor the changing voltage at the resistances node.

Two analog inputs are reserved for battery voltage and battery current:

```
#define Amperes_AnalogIn          A10 // Amperage from ACS758LCB-050B
#define Voltage_AnalogIn          A12 // Battery voltage
```

The amperage is measured by a Hall-effect device, essentially a low-resistance copper shunt, an integrated magnetic flux sensor, and a low-noise amplifier which provides a current to voltage transformation. The battery voltage is calculated by use of a resistance divider network.

Other analog sensor requirement focused on the fuel pressure and remaining fuel volume. Thus:

```
#define FuelQty_AnalogIn          A13 // Engine Fuel Level
#define FuelPres_AnalogIn         A15 // Engine Fuel Pressure

int GFLevl_1    =  0;                 // Gasoline Fuel Level
float GFPress   =  0.0;               // Fuel Pressure PSI
```

I2C Ports

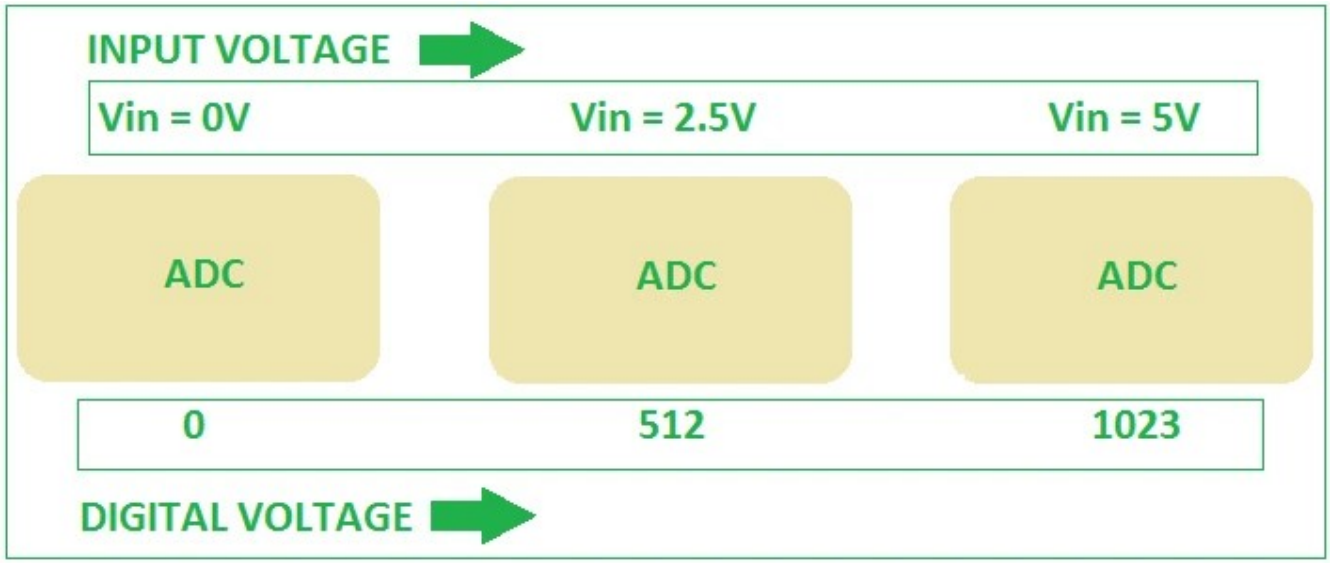
Another need is the exhaust temperature of which there are two exhaust headers, thus two separate sensors are required. Early on, this was a critical design concern as the most common sensor for such high temperatures are specialized thermocouples. Accurate design of thermocouple measurement electronics is equally as much one of physics as electronics. Simple R-R implementation as used for oil temperature would be inadequate. Therefore, a specialized integrated circuit was the central design element: the MAX31855 IC is capable of operating as a trans-conductance amplifier with internal microcontroller to compensate for cold-junction requirements. The IC is connected to the Mega2560 board via the I2C bus. An output port on the 2560 is used to implement an effective "chip-select" as two temperature channels are required and two MAX31855 ICs are utilized. In code, this appears as:

```
#include "./MAX31855.h"              // Local (embedded) library

int thermoDO    = 51;                // common SPI data to both 31855s
int thermoCS_1  = 52;                // first EGT thermocouple amp Chip Select
int thermoCLK   = 53;                // common SPI clock to both 31855s
int thermoCS_2  = 50;                // second EGT thermocouple amp Chip Select

MAX31855 thermocouple_EGT1(thermoCLK, thermoCS_1, thermoDO);
MAX31855 thermocouple_EGT2(thermoCLK, thermoCS_2, thermoDO);
```

To standardize the software implementation, the (mature, commercial) Adafruit MAX31855 library was chosen, some changes were made as required, and the entire library was encapsulated within the Arduino codebase so that any future changes made by Adafruit and automated updated by the Arduino Library Manager would not have any negative effects. The code line **`#include "./MAX31855.h"`** in the previous "snippet" instructs the compiler to look in the current directory for the library and thus prevents the normal Arduino library search methodology.

Analog Input to Display Conversion

On the Atmel 2560 microcontroller, an input (analog) voltage is converted to a number between 0 and 1023 (1024 steps == 2^10) as shown below:



To make “human sense” of the number, could can utilize a mathematical equation to derive a number in some reference frame, or one could utilize a “look-up table” to convert the digital voltage representation of 0 – 1023 into some real, understood number. In the Engine Monitor, both techniques are utilized.

As the display mechanism in the cockpit utilizes two (2) LCD displays, the display area is limited. Traditionally, engine monitors were analog gauges:

As such, the 1024 possible values for look-up quantities is an overkill. For look-up arrays, the 1024 result is further divided by 4 to map the output of the A/D into a 0 – 255 space (8-bits == 2^8). In these array elements, the engine owner can place appropriate/representative values for the quantity being measured even if the results of the reading is non-linear. This lookup process is ultra-fast and uses little microcontroller resources and is ideally suited to such quantities as fuel-level (non-linear tank.)

Display Formats

The LCD display chosen for the cockpit is a high contrast OLED display in the standard 1602 format. Two displays are used, and data is written to each as shown below in the graphics:

OLED 1

```
1 2 3    4 5 6 7 8 9 1 1 1 1 1 1 1
                     0 1 2 3 4 5 6
FUEL       OIL      CHT      EGT
QTY        TEMP     FRONT    LEFT
X X X      X X X    X X X    X X X X
X X X      X X X    X X X    X X X X
FUEL       OIL      CHT      EGT
PRES       PRES     REAR     RIGHT
1 2 3    4 5 6 7 8 9 1 1 1 1 1 1 1
                     0 1 2 3 4 5 6
```

OLED 2

```
1 2 3    4 5 6 7 8 9 1 1 1 1 1 1 1
                     0 1 2 3 4 5 6
Volts                Amps

XX. X                XX.X
1 2 3    4 5 6 7 8 9 1 1 1 1 1 1 1
                     0 1 2 3 4 5 6
```

Display Requirements

Within the program code, provision is made to allow any analog calculated or look-up value to be compared to an established *High* and *Low* value (ceiling/floor.) If the value being returned by the code function is ***higher than the High*** or ***lower than the Low***, then the display will blink to indicate an out-of-bounds condition (error.) The flashing condition will automatically correct to non-blink once the parameter being monitored returns to a range between *High* and *Low **(non-inclusive.)***

Additionally, any of the monitored functions that are in error status will cause a single output port to be driven to the Low (L) state. This is implemented to allow for a single LED on I/O pin 2 to be used as a warning indicator. In code, this appears as:

```
int   errLED      =  2;             // Error LED for blinking OLED
// ……..
if (err_count > 0) {
 digitalWrite(errLED, LOW);
 delay(5);
```

Specific Implementation

Fuel

Fuel level and fuel pressure are two functions but are implemented completely differently, fuel level (quantity) is a look-up function and fuel pressure is formula-based. These two methodologies are reused in a similar manner for all analog inputs to the engine monitor. Note that one function is implemented as an integer and one function is implemented as a float.

```
#define FuelQty_AnalogIn          A13 // Engine Fuel Level
#define FuelPres_AnalogIn         A15 // Engine Fuel Pressure
int GFLevl_1    =  0;               // Gasoline Fuel Level
float GFPress   =  0.0;             // Fuel Pressure PSI
const int GFL[] PROGMEM = { 0, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7,
7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 10, 10, 10, 10,
10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10,
11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11,
11, 11, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13,
13, 13, 13, 13, 13, 13, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14,
14, 14, 14, 14, 14, 14, 14, 14, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 16, 16,
16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18 };
int fuelRemain()
{
    // int therm = (analogRead(FuelQty_AnalogIn) >> 2) ;
    // refererence https:  //forum.arduino.cc/index.php?topic=356064.0
    // Array boundary is 0 to 255
    // Scale input value by 4 as raw read is 10-bit == 0-1023 and the
    // desired is 8-bit resolution to match lookup arrays
    // therm = therm / 4;  == shift right 2 places, that is 2^2
    int ROW = 7;           // averaging matrix index
    int therm = ( analogAverage ( ROW, FuelQty_AnalogIn ) >> 2) ;
    // Gasoline Fuel Level
    GFLevl_1 = pgm_read_word(&GFL[therm]);
    return (GFLevl_1);
}
```

```
float fuelPressure()
{
    float temp;
    int ROW = 8;
    int voltage = analogAverage ( ROW, FuelPres_AnalogIn ) ;
    temp = float( float(voltage) / 1023) * 5.0 ;
    // map(value, fromLow, fromHigh, toLow, toHigh)
    // Following taken from Graph of Pressure Transducer showing min & max voltages
    // low = 103 (1/10 of 1024) == 0.5 Volts
    // high= 921 (9/10 of 1024) == 4.5 Volts
    // temp = map ( voltage, 103, 921, 0.5, 4.5) ;    // linear, 1::1 slope
    // Gasoline Fuel Pressure calculated from slope of line: f(x) == r^2 = 1.0000
    GFPress = (temp * 3.75) - 1.875 ;
    if (GFPress < 0.0)
{
   GFPress = 0.0;
}
    return (GFPress);
}
```

Adding the Error Wrappers

In code as shown above in the fuel implementation, values are returned by special functions written to either calculate by formula or look-up appropriate values from per-populated tables. These formulas are unaware of any *High* and *Low* requirements.

Rather, the error indication is specifically for the display visual (or audio alarm) of the pilot. As such, the acceptable operating range is implemented in the OLED display code. Specifically:

```
/* A13 Gasoline fuel remaining */
vLow = 00.0; vHigh = 00.0;  if (oDISP1_1_0(vLow, vHigh)) ++err_count;

/* A1  Oil temperature */
vLow = 00.0; vHigh = 00.0;  if (oDISP1_1_4(vLow, vHigh)) ++err_count;

/* A3  CylHead Front */
vLow = 00.0; vHigh = 00.0;  if (oDISP1_1_8(vLow, vHigh)) ++err_count;

/*     Exhaust Gas left */
vLow = 00.0; vHigh = 00.0;  if (oDISP1_1_12(vLow, vHigh)) ++err_count;

/* A15 Gasoline fuel pressure */
vLow = 02.0; vHigh = 05.0;  if (oDISP1_2_0(vLow, vHigh)) ++err_count;

/* A2  Engine Oil Pressure */
vLow = 00.0; vHigh = 00.0;  if (oDISP1_2_4(vLow, vHigh)) ++err_count;

/* A4  CylHead Rear temperature */
vLow = 00.0; vHigh = 00.0;  if (oDISP1_2_8(vLow, vHigh)) ++err_count;

/*     Exhaust Gas right */
vLow = 00.0; vHigh = 00.0;  if (oDISP1_2_12(vLow, vHigh)) ++err_count;

/* A12 Battery voltage measured */
vLow = 00.0; vHigh = 00.0;  if (oDISP2_2_1(vLow, vHigh)) ++err_count;

/* A10 Amps measured */
vLow = 00.0; vHigh = 00.0;  if (oDISP2_2_10(vLow, vHigh)) ++err_count;
```

Variable **`err_count`** starts with a value of 0 and ends with a value of 0 or > 0; that is, if >0 then there was a warning in one or more of the displayed items. At this time the specific function sending the warning is unknown, but the OLED display will handle the display blinking, if appropriate. Note in the complete code that **`err_count`** is set to 0 in the loop() function.

Anatomy of the OLED display functions

In an effort to keep the code-base clean, the multiple OLED display functions (10 total) are written almost identically: each being a logical function, returning ***true*** or ***false***. Within the function, a static variable is utilized to maintain state between each call from the main loop(). In this manner, a kind of flip-flop action can be implemented to make the display appear to blink by alternately displaying the actual value and a series of spaces. If *High* and *Low* are equal (as H =0 and L=0), then the function will not execute a blink routine.

```
bool oDISP1_1_0(float L, float H) {         /* Gasoline fuel remaining (gals) */
  bool error = false;                       // local error condition returned
  bool bypass = ( (int (L)) == (int (H)));  //disable flash when L==H
  static bool flash = false;                // local variable for blinking display
  float temp  = float(fuelRemain() );
  // below both conditions must be satisfied or out-of-range is detected
  if ( ! (bypass)) {            // error is initially false, so will remain so on bypass
    if ( (temp > L) && (temp < H)) {
        error = false;
      } else {
        error = true;
    }
  }
  if ( error ) flash = !flash;              // this will alternate on each pass
  oLED1.cursorPosition(1, 0);
  // cute blink value function only works when L <> H
  if (flash && error) {
    oLED1.print("    ");
  } else {
    oLED1.printFloat(temp, 3, 0);
  }
  Serial.print("{\"Gas_Lvl\":"); Serial.print(temp); Serial.print(",");
  return(error);
}
```

Averaging Matrix

The main software routine is a loop which repeats approximately on a 700 mS schedule. This refreshes all of the OLED display readings, the EGT readings,, and updates both the battery voltage and the current usage. Reading the analog sensor values and updating the display at a fast interval can result in some “display fluctuations.” While the fluctuations are in the least-most significant digit, the effect can still be annoying.

To smooth the display readings, an average is calculated from the current reading plus the last 4 recent readings (oldest reading is dropped); thus 5 values are averaged over slightly under 3 seconds. The averaging matrix consists of a multidimensional array of (0 – 4) x (0 – 8) elements. The 9 row indexes are represented by the following table:

Function index	Analog Input	col indx 0	col indx 1	col indx 2	col indx 3	col indx 4
0	**A0** (Water temp)	-	-	-	-	-
1	**A1** (Oil temp)	-	-	-	-	-
2	**A2** (Oil pressure)	-	-	-	-	-
3	**A3** (CHT 1)	-	-	-	-	-
4	**A4** (CHT 2)	-	-	-	-	-
5	**A10** (Amps)	-	-	-	-	-
6	**A12** (Voltage)	-	-	-	-	-
7	**A13** (Gas level)	-	-	-	-	-
8	**A15** (Gas pressure)	-	-	-	-	-

As seen in the table, two indexes are used, but the column index is private to the averaging function and thus the calling function only needs to pass the Function index so that the generic averaging routine can insert the roll the row dropping the oldest value, insert the new value, and then calculate the average over the 5 stored numbers. As this routine is intended to only handle analog inputs, if the calling routine also passes the Arduino analog pin number, the averaging routine can perform that function on behalf of the caller; thus simplifying the analog access.

In the Arduino community, some programmers suggest that doing a “double read” in a back-to-back fashion provides a higher quality of analog readings. Other suggest that a small 10mS delay between readings will provide better quality analog readings. Lab testing did not indicate a quality or stability improvement; however, both techniques were implemented as the Mega2560 microcontroller is not being resourced challenged in this application.

At the end of the average calculation, the value is returned to the calling function. Note that the calling function must still perform specific function calculations or lookup values appropriate to the function implementation. However the approach taken here allows for the storage and averaging of integers regardless of the integer or float type of the calling routine.

In code, this appears as:

```
const int Average_Depth = 5 ;                // can be changed at expense of loop{} total time
int AnalogAverageMatrix [9][Average_Depth];  // 9 is based on A0, A1, A2, A3, A4, A10, A12, A13, A15


int analogAverage ( int ROW, int ApinNo )
{
    int ReadVal = 0;
    // Rollup ... small matrix, just assign for testing, could be a for() if desired
    // Oldest value is [0], newest value [4]
    AnalogAverageMatrix [ROW][0] = AnalogAverageMatrix [ROW][1];
    AnalogAverageMatrix [ROW][1] = AnalogAverageMatrix [ROW][2];
    AnalogAverageMatrix [ROW][2] = AnalogAverageMatrix [ROW][3];
    AnalogAverageMatrix [ROW][3] = AnalogAverageMatrix [ROW][4];
    // do two reads ... as a test, but one should really be enough IMO
    ReadVal = analogRead( ApinNo ); // delayMicroseconds( 10 );
    AnalogAverageMatrix [ROW][4] = analogRead( ApinNo );
    if ( DIAG) Serial << ROW  << ": " << AnalogAverageMatrix [ROW][0] << ", " <<
AnalogAverageMatrix [ROW][1] << ", " << AnalogAverageMatrix [ROW][2] << ", " << AnalogAverageMatrix
[ROW][3] << ", " << AnalogAverageMatrix [ROW][4] << endl ;
    // now perform an average [0] -> [4]
    ReadVal = AnalogAverageMatrix [ROW][0] +
              AnalogAverageMatrix [ROW][1] +
              AnalogAverageMatrix [ROW][2] +
              AnalogAverageMatrix [ROW][3] +
              AnalogAverageMatrix [ROW][4] ;
    ReadVal = ReadVal / Average_Depth;
    if (DIAG) Serial << "Average for " << ROW << ": " << "is " << ReadVal << endl;
    return ReadVal;     // The average is returned to calling function
```

Appendix

Serial Outputs

Diagnostics

The Arduino serial port is used to stream all calculated readings; mostly what is seen on the OLED displays. The default Mega2560 serial port is utilized:

```
Serial.begin(9600); // Hardware serial port to PC terminal via USB
```

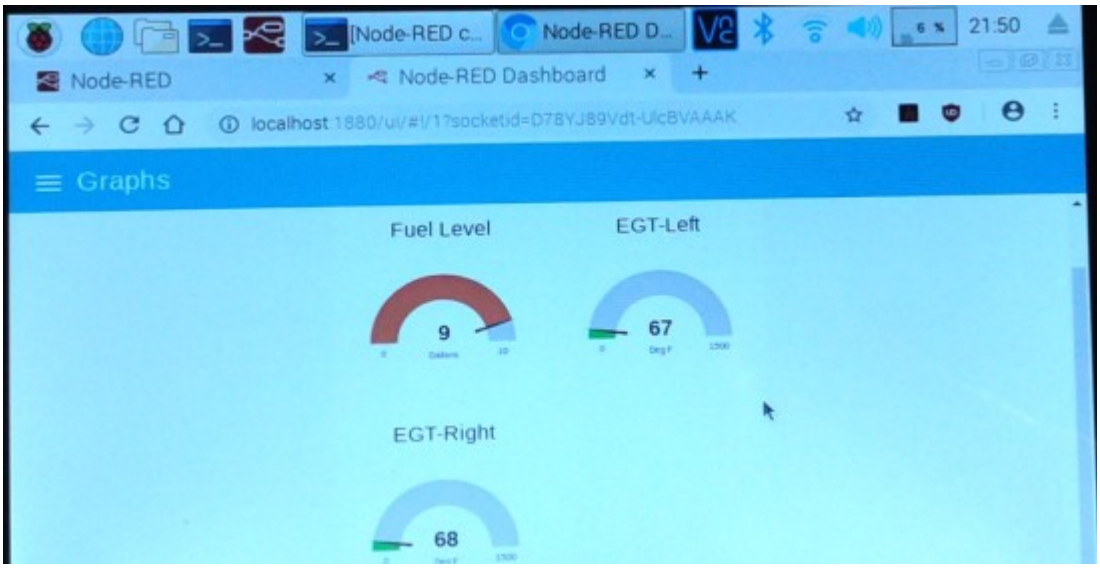
There is no requirement to utilize the serial data stream; no handshaking is performed, so leaving this digital output disconnected presents no issues to other functions.

Serial diagnostic formatting

The serial stream is output in a pseudo-JSON format suitable for serial input directly into Node Red for creating a graphical dashboard. For example, fuel level on the OLED display is handled by function **oDISP1_1_0()** and the associated JSON format serial print appears as:

```
Serial.print("{\"Gas_Lvl\":"); Serial.print(temp); Serial.print(",");
```

Within Node Red, graphic screens can be designed to represent conventional analog meters, for example:



OLED displays

The display used for the cockpit is a 2-line x 16-character OLED. Two of these displays are utilized. The specific display is here: https://www.picaxestore.com/axe133y

The output instructions (code) is scattered between the Arduino setup() routine, the loop() routine, and all of the functions that produce OLED display, which themselves are on serial ports as shown in the following code:

```
int   OLED_Tx     =  3;    // Serial out for OLED_1 (Upper display)
int   OLED_Tx2    =  5;    // Serial out for OLED_2 (Lower display)
// Instantiate the OLED display objects
AXE133Y oLED1 = AXE133Y(OLED_Tx);
AXE133Y oLED2 = AXE133Y(OLED_Tx2);
```

To simplify the OLED display code, an existing library was utilized:

http://stompville.co.uk/?p=303 and https://playground.arduino.cc/Main/AXE133Y/

The library is installed in the Arduino system library and is identified in the code as:

```
#include <AXE133Y.h>                // System library
```

Note that this library, unlike the MAX31855, is not encapsulated in the source code. Therefore, some care should be exercised if the Arduino IDE wishes to update this system library; always keep a backup copy of older libraries which are known to work correctly.