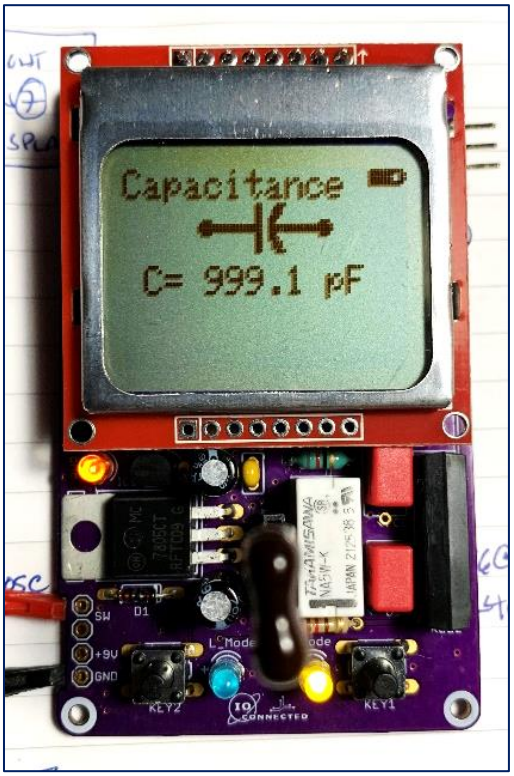
Build an LC Meter

Bruce Hall, W8BH



An LC meter is a handy tool for any electronics hobbyist or ham radio operator. There are ready-built devices which you can purchase, but what's the fun in that? You can make your own meter for less money and with better accuracy. Read on if you are interested in building your own LC meter based on the famous AADE design.

Introduction

Several DIY meters are available on the web. One of the most accurate and widely acclaimed was designed and sold by Neil Hecht and his company AADE (Almost All Digital Electronics). Neil passed away in 2015 but his design lives on.

An AADE-inspired design was recently implemented by "coreWeaver", using an ATmega328 for the MCU and a Nokia 5110 LCD for the display. The author designed the circuit, PCB, and 3D-printed case. And he provides a four-part video describing his device. The documentation for his project, including links to his videos, is found on his GitHub page at: https://github.com/coreWeaver/LC-Meter

The notes below describe my experience with the author's project. The author wrote his firmware in AVR Basic (BASCOM). He provides the source code and the .HEX file for anyone who would like to use his software.

I used the author's PCB but chose to write my own software. Why?

- I want to customize the software without buying a BASIC compiler.
- Arduino IDE is free and widely understood by the open-source community.

The source code for my LC meter is on my GitHub page at: https://github.com/bhall66/LC-meter

Hardware

The operation of this circuit is described elsewhere. Briefly, an LM311 comparator is configured as an oscillator. The frequency-determining components of this oscillator are a parallel 82 uH coil L1 and 1000pF capacitor C12, which resonate at approximately 500 kHz. The component under test is added

to the LC tank -- either a coil in series with L1 or a capacitor in parallel with C12 -- and the new frequency is measured. The change in frequency is mathematically related to the unknown component's value. The accuracy of the meter depends on an internal calibration capacitor, which is nominally a 1000pF component with 1% (or better) tolerance.

The oscillator frequencies are measured by the MCU, which calculates the component value and displays it on an LCD screen.

Power is supplied via a 9V battery. A 7805 voltage regulator is used to supply 5V to the MCU and oscillator. The regulator is large for the modest current requirement of the board. An LM317 is used to provide the 3.3V necessary for display. R1 and R3 control the voltage output of this regulator. R5 and R6 form a resistive divider which is used for measuring the battery voltage, converting the 9V input into a measurable 4.5V. (The MCU cannot measure voltages higher than 5V.)

The microcontroller is an ATmega328, the same device used in the Arduino UNO. Although this component has been around many years, it has more than enough I/O pins and computing power for the job. It uses an 8 MHz crystal clock, running half as fast as the 16 MHz UNO. The 100nF bypass caps C5, C8, C9 are the usual MLCC type and the values are not critical. C7, C8 are loading caps for the crystal oscillator; small ceramic discs of 20-27pF are OK for these. R4 is a 10K pullup resistor for the MCU/display reset line.

The display is a Nokia 5110 monochrome 84 x 48 pixel LCD. This display is very readable in daylight and has backlight LEDs for nighttime viewing. It requires very little power. R7-R14 are used to convert 5V signals from the MCU into 3.3V signals for the display.

There are three display LEDs and two standard 6mm pushbutton switches. The switches do not have external pullup resistors on them, so internal pullups should be enabled by software.

Construction

All components, except the battery, probes, and power switch, are mounted on a single PCB. The PCB pads and holes are small, so a small-tipped soldering iron and small-gauge solder (0.015") are recommended. The resistor pads are only 7mm apart; you must bend the leads tightly against the ¼ watt resistor bodies in order to place these components.

I like to complete the build in stages, testing as I go. First, I fitted all the power components and confirmed 5V and 3.3V output. Then I added the display and its support circuitry, creating the necessary solder bridge for the backlight. Next, I added the MCU parts and ran a diagnostic sketch to confirm that the MCU and display were working. Then I added the switches, LEDs, and current limiting resistors and ran a second sketch to test those components. In the final step I added the LM311 and all the remaining components.

Some of the components proved difficult to source. Here a *partial* list of the components and where I obtained them. You will also need to have a PCB made from the author's Gerber files.

Partial Parts List:

Part	Supplier/Link	Cost
ATmega328P MCU 28-pin DIP	Mouser 556-ATMEGA328P-PU	2.60
Nokia 5510 LCD display	eBay "Nokia 5110 LCD"	4.00
Fujitsu DPDT Relay NA-5W-K	Mouser 817-NA-5W-K	3.55
Meder Reed Relay SPST 5V	Mouser 876-SIL05-1A72-71L	3.64
LM317 Voltage Regulator	Mouser 595-LM317LCLPRE3	0.49
BAT42 200mA diode DO-35	Mouser 511-BAT42	0.38
WIMA 1000pf Film Cap, 2.5%	Mouser 505-FKP2D011001DHC00	0.61
WIMA 1000pF Film Cap, 1%	Mouser 505-FKP2D011001D00ES	2.51
Electrolyic Cap, 10uF 16V	Mouser 647-UMA1C100MDD1TP	0.28
LM311 Voltage Comparator	Mouser 926-LM311N/NOPB	1.38
Bourns 82uF Fixed Inductor 5%	Mouser 542-78F820J-RC	0.33
Yageo 1.2K Resistor, ¼ w, 5%	Mouser 603-CFR-25JR-521K2	0.10
Yageo 6.8K Resistor, ¼ w, 5%	Mouser 603-CFR-25JT-52-6K8	0.10
Yageo 120R Resistor, ¼ w, 5%	Mouser 603-CFR-25JT-52-120R	0.10

After building the board, it's time to upload software. The author's PCB does not have an ICSP (in circuit serial programming) port, so you must remove the MCU chip from your meter and insert it into a dedicated programmer. From the Tools menu of the Arduino IDE,

- Select Board "**Pro or Pro Mini**"
- Select Processor "ATmega328P (3.3V, **8 MHz**)". This choice is necessary to match the 8MHz clock of the LC meter.
- Select your programmer
- Select "Burn Bootloader" In addition to installing a bootloader onto your ATmega328, this step will also set the ATmega328 hardware fuses.

Next, install a library for the Nokia 5110 display. I am using the one from Adafruit called "PCD8544". Also install the Adafruit GFX library, if not already installed.

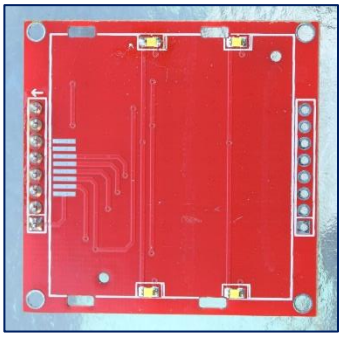
Finally, try uploading a simple sketch, and replace the programmed MCU into the LC meter.

The Nokia 5510 display

This module was originally developed in 2001 for the Nokia 5510 mobile phone. The module contains a monochrome 48 x 84 pixel LCD and a Philips PCD8544 driver IC. In its current form, the module is contained in a metal housing that clips to a PCB carrier.

This is a 3.3V device that uses SPI serial bus for communication. There are 8 interface pins: reset, chip enable, data/cmd, data in, clock, Vcc, backlight, and ground.

Back of the Nokia 5510 display



PCB carrier with display removed

The PCB carrier contains 4 LEDs, which function as the display backlight. On my unit, the anodes of LEDs connect directly to Vcc, and each cathode is tied to the "backlight" pin via a 150-ohm resistor. As a result, grounding the backlight pin causes all 4 LEDs to turn on. On the back of my display carrier there is a jumper position marked "JP". Create a solder bridge across this jumper to turn the backlight on permanently. (Soldering across this jumper is easier than soldering the small one found on the LC meter PCB.) The current consumption of my module is 11 mA for the backlight and less than 1 mA for the display. There are several varieties of this display on the market, and not all of them have the same backlight configuration.

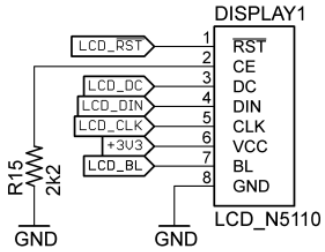
Hello World

The first and simplest diagnostic sketch for any display is to print "Hello World":

```
#include <Adafruit_GFX.h>
#include <Adafruit_PCD8544.h>

Adafruit_PCD8544 lcd =
 Adafruit_PCD8544(13, 11, 12, -1, 3);   // Pins for CLK,Din,DC,CS,RST


void setup() {
  lcd.begin();                          // initialize the display variable
  lcd.setContrast(60);                  // good values are between 40 and 70
  lcd.clearDisplay();                   // start with blank screen
  lcd.setTextSize(2);                   // use big and beautiful characters
  lcd.println("Hello");                 // say something profound
  lcd.println("World");
  lcd.display();                        // update the screeen
}

void loop() {}
```

The first two lines, prefaced with "#include", pull in two libraries needed by the Nokia 5510 display. You will need to install both in your Arduino IDE first. The next line instantiates the lcd variable, indicating which Arduino pins connect to the display Clock, Data, Data/Command, Chip Select, and Reset lines. Notice that the Chip Select pin is specified as "-1", which means that the display is selected all the time. Referring to the schematic, you will notice that the CE (same thing as CS) line is tied to ground via R15, thereby keeping the device always enabled.

In setup(), there are a few lines to specify the size of the text and contrast level for the display, followed by printing the words Hello and World. Notice the last line, lcd.display(). The Nokia display does not automatically show all its updates. These are written into memory but not shown until you explicitly say so. The display remains blank until you call lcd.display().



The source code for LC_Hello is on GitHub.

Buttons and LEDs

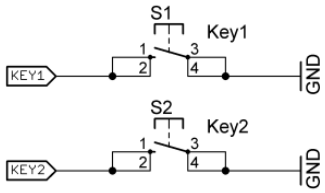
The 2 buttons and 2 LEDs are each tied to a microcontroller pin. I like starting my sketches with a list of the pins and the devices they are connected to. Compare these #defines with the schematic:

```
#define C_KEY           9                   // PB1: "KEY1" pushbutton
#define L_KEY           2                   // PD2: "KEY2" pushbutton
#define C_LED           5                   // PD5: LED for C_MODE
#define L_LED           6                   // PD6: LED for L_MODE
```

Now we can refer to these interface components by name, rather than the pin they are connected to. Next, at the start of the sketch, declare each of these MCU pins as input or output:

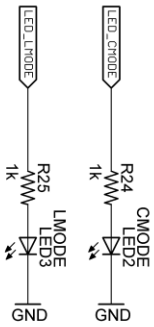
```
void initPorts() {                          // INITIALIZE MCU PINS
  pinMode(L_KEY,INPUT_PULLUP);              // L pushbutton
  pinMode(C_KEY,INPUT_PULLUP);              // C pushbutton
  pinMode(C_LED,      OUTPUT);              // C LED
  pinMode(L_LED,      OUTPUT);              // L KED
}
```

The Keys are on input pins and the LEDs are on output pins. INPUT_PULLUP means that the MCU internally applies a pullup resistor to the pin, ensuring that its value is logic 1 in the absence of any grounding input. Pressing a key pulls its corresponding pin to ground. Here is the code:




```
bool LKeyPressed() {            // is the L key pressed?
  return !digitalRead(L_KEY);   // pressed = pin is grounded
}

bool CKeyPressed() {            // is the C key pressed?
  return !digitalRead(C_KEY);   // pressed = pin is grounded
}
```

To control the LEDs, the MCU applies 5V to the corresponding pins. We can control both LEDs with a single routine:




```
void setLEDs(int led1, int led2) {    // control the LEDs
  digitalWrite(L_LED,led1);           // turn the L LED on/off
  digitalWrite(C_LED,led2);           // turn the C LED on/off
}
```

With these routines in place, we can write a full Arduino sketch to test the buttons and LEDs. You can download the full sketch "LC_Blinker" on GitHub.

```
void setup() {
  initPorts();                              // initialize MCU ports
  initDisplay();                            // initialize NOKIA display
}

void loop() {
  if (LKeyPressed() && CKeyPressed()) {     // both keys pressed?
      setLEDs(1,1);                         // ..light both LEDs
  } else if (LKeyPressed()) {               // only left key pressed?
      setLEDs(1,0);                         // ..light left LED
  } else if (CKeyPressed()) {               // only right key pressed?
      setLEDs(0,1);                         // .. light right LED
  }
  delay(700);                               // for a little bit
  setLEDs(0,0);                             // then turn off the LEDs
  delay(300);                               // repeat every second
}
```

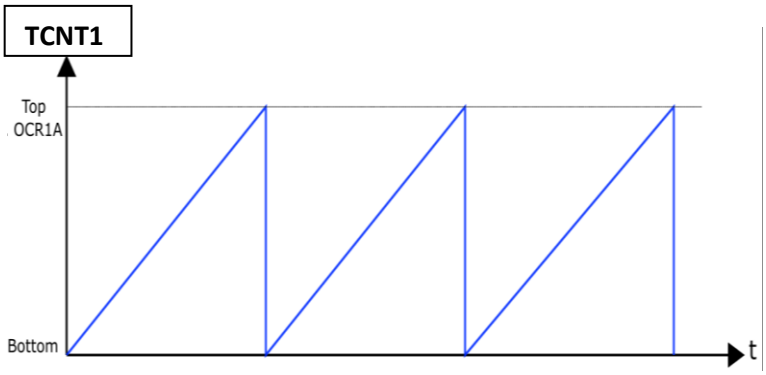
Hardware Timers

Any discussion about the internals of an MCU gets complicated - fast. So, with that in mind, you may keep your sanity and skip ahead to the next section. Or, grab a beverage and learn how to turn our trusty ATmega328 into a frequency counter. The basic idea is simple: count the number of pulses in exactly one second, and that number is the frequency in Hertz. 1000 pulses per second = 1000 Hz. We need two routines: one to count pulses, and one measure exactly one second. Let's consider the second-timer first.

The ATmega328 MCU contains three internal timer/counters, which I abbreviate TC0, TC1, and TC2. TC0 and TC2 are 8-bit counters, meaning that they can count to 256. TC1 is 16 bits in size. What to these devices count? Usually, they count the MCU's own clock pulses *or some fraction thereof*. And each timer/counter has several different modes of operation. We will only use the mode called "Clear Timer on Compare Match" (CTC), in which the timer counts to a certain value and then resets to zero.

Back to the topic at hand: how to we measure out a second? Set TC1 to count clock cycles until exactly one second has passed. If our clock is running at 8 MHz, then we will need to count to 8,000,000 to mark out 1 second. This is number is too large for our 16-bit counter, which only has enough bits to count to 65536. But, by using a prescalar, we can specify a much slower clock, and therefore won't have to count so high. For example, a prescalar of 256 will result in a clock of 8MHz/256 or 31.250 kHz. If we count to 31250 with this lower frequency clock, exactly 1 second will have passed.

You can set up TC1 with Arduino code. Each of the Timer/Counters has several named registers for this purpose. For example, the register containing the count is called TCNT1 and the compare match register is called OCR1A. TCNT1 starts at 0 and counts as high as the value in OCR1A before it resets. See diagram at right. The following code snippet will accomplish our 1 second timer:




```
TCNT1   = 0;                          // TIMER1 SETUP: interrupts at 1 Hz
TCCR1A  = 0;                          // no external outputs
TCCR1B  = bit(WGM12) + bit(CS12);     // CTC Mode; prescalar /256
OCR1A   = 31250-1;                    // compare match register 8 Mhz/256/1Hz
TIMSK1  = bit(OCIE1A);                // enable timer compare interrupt
```

Complicated, but bite sized. It puts TC1 into CTC mode, using a prescalar to reduce the counter input frequency to 31.25 kHz, and counts 31249 pulses. When the 31250th pulse comes in, an interrupt is generated and the counter resets.

TC1 is our one second timer. Now we need something to count the incoming pulses. I wrote above that Timer/Counters *usually* count MCU clock pulses. However, TC0 and TC1 have a special mode in which they count external pulses instead. The schematic and PCB connect the external oscillator to the input pin of Timer0.

The following code will put TC0 into CTC mode, counting external pulses on pin PD4:

```
  TCNT0   = 0;                              // TIMER0 SETUP: count external pulses
  TCCR0A  = bit(WGM01);                     // CTC Mode; no external outputs
  TCCR0B  = bit(CS00)+bit(CS01)+bit(CS02);  // use T0 (external) source = Digital 4
  OCR0A   = 256-1;                          // count to 256
  TIMSK0  = bit(OCIE0A);                    // enable timer overflow interrupt
```

Notice that this 8-bit timer can't count higher than 256. To get around this limitation, our Timer0 interrupt routine increments an overflow counter each time that we've counted to 256. And our 1-second interrupt routine for TC1 will multiply that count by 256 to get the total number of counts per second:

```
ISR(TIMER0_COMPA_vect) {                    // INTERRUPT SERVICE ROUTINE: Timer0 compare
  ovfCounter++;                             // increment overflow counter
}

ISR(TIMER1_COMPA_vect) {                    // INTERRUPT SERVICE ROUTINE: Timer1 compare
  int t0 = TCNT0;                           // save TCNT0
  TCNT0 = 0;                                // & reset TCNT0 as soon as possible
  frequency = (ovfCounter*256) + t0;        // calculate total pulses in 1 second
  ovfCounter = 0;                           // reset overflow counter
  seconds++;                                // increment seconds counter
}
```

After setting up the timer registers and associated interrupt routines, the frequency counter is running in the background, counting pulses on Arduino digital pin 4, and updating the measured frequency once per second. We don't have to tell it to start, or stop, or wait. It is always on.

As an aside, Timer/counter0 is used by the Arduino environment for several important timing functions: delay(), millis() and micros(). If we use TC0, our own sketch - including any libraries that it uses - can no longer call these important functions. So if we need a delay() routine, we must write our own. The remaining hardware timer, TC2, can be used for this purpose. Consider the following code:

```
TCNT2   = 0;                              // TIMER2 SETUP: interrupts at 1000 Hz
TCCR2A  = bit(WGM21);                     // CTC Mode; no external outputs
TCCR2B  = bit(CS22);                      // prescalar /64
OCR2A   = 125-1;                          // 8 Mhz/64/1000Hz = 125 (250 for 16MHz)
TIMSK2  = bit(OCIE2A);                    // enable timer compareA interrupt
```

The above lines will put TC2 into CTC mode and cause an interrupt to fire every millisecond. Our corresponding interrupt routine will keep track of the number of milliseconds that have passed in a global variable called *ticks*.

```
ISR(TIMER2_COMPA_vect) {                    // INTERRUPT SERVICE ROUTINE: Timer2 compare
  ticks++;                                  // increment millisecond counter
}
```

Here is a replacement routine for the Arduino delay() routine, called *wait()*. It just waits in an empty while loop until the specified number of milliseconds have passed:

```
void wait(int msDelay) {                    // substitute for Arduino delay() function
  long finished = ticks + msDelay;          // time finished = now + specified delay
  while (ticks<finished) { };               // spin your wheels until time is up
}
```

A test sketch illustrating all 3 timers is on my GitHub page as "LC_Timers".

Converting Oscillator frequency into Capacitance & Inductance

Neil Hecht is credited with the following method. I encourage you to watch coreWeaver's YouTube video, as he demonstrates the algebra of going from the resonant frequency equation ($F = \frac{1}{2\pi\sqrt{LC}}$) to equations that express the unknown value in terms of frequencies.

Using Neil's method,

$$C_x = \left[\frac{\left(\frac{F1}{F3}\right)^2 - 1}{\left(\frac{F1}{F2}\right)^2 - 1}\right] * C_{cal}$$

$$L_x = \left[\left(\frac{F1}{F3}\right)^2 - 1\right] * \left[\left(\frac{F1}{F2}\right)^2 - 1\right] * \left(\frac{1}{C_{cal}}\right) * \left(\frac{1}{2\pi F1}\right)^2$$

Where:
F1 = frequency with no calibration cap, no unknown
F2 = frequency with calibration cap in circuit, no unknown.
F3 = frequency with unknown in circuit, no calibration cap

The oscillator frequency is measured under three conditions. First, with an in-circuit inductor and capacitor. This is called F1. Next, with a calibration capacitor added in parallel with the LC tank. This is called F2. And finally, with the unknown component added to the LC tank. The resulting frequency is called F3.

Unknown inductors are added in series with the in-circuit inductor. Unknown capacitors are added in parallel to the in-circuit capacitor. The action of adding a series inductance or parallel capacitance is controlled by a DPDT relay.

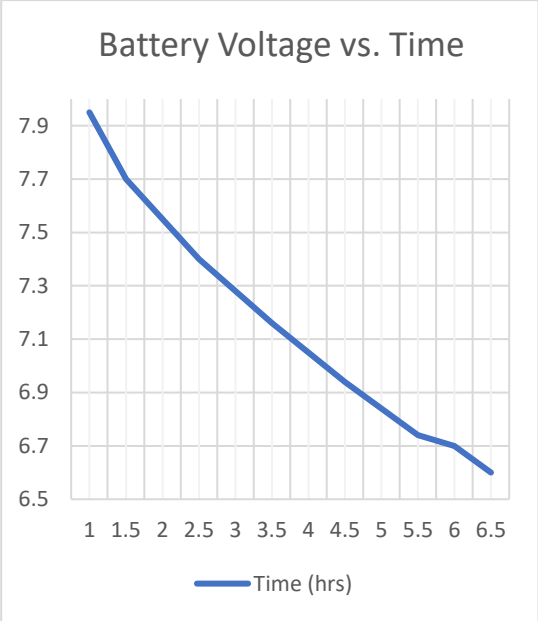
Note that L_x and C_x both depend on the calibration capacitor value. For accurate measurements, C_{cal} should have a tolerance of 1% or less.

As intimidating as these equations seem, coding them is not too difficult:

```
float getCapacitance() {                    // CAPACITANCE EQUATIONS:
  F3 = frequency;                           // save current frequency
  float p = (float)F1*F1/F3/F3;             // (F1^2)/(F3^2)
  float q = (float)F1*F1/F2/F2;             // (F1^2)/(F2^2)
  float r = (p-1)/(q-1)*CCAL;               // this is the capacitance equation
  return r;                                 // result in picoFarads
}

float getInductance() {                     // INDUCTANCE EQUATIONS:
  F3 = frequency;                           // save current frequency
  float p = (float)F1*F1/F3/F3;             // (F1^2)/(F3^2)
  float q = (float)F1*F1/F2/F2;             // (F1^2)/(F2^2)
  float s = 2 * 3.1415926 * F1;             // 2pi*F1
  float r = (p-1)*(q-1)/CCAL/s/s;           // this is the inductance equation
  return (r*1E24);                          // convert to picoHenries
}
```

Current Consumption & Battery Life

My meter consumes 40 mA of current in its idle state. Of this, 14 mA supplies the backlight, and 26 mA supplies the PCB and display circuitry. Your results may vary, depending on the backlight module you are using. When the DPDT relay is energized, which happens during calibration and capacitance measurements, an additional 33 mA of current is required. When the SPST is energized during calibration, an additional 11 mA is required.



Battery voltage gives us a good estimate of remaining battery life:

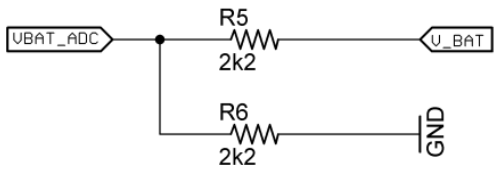
Battery Voltage (V)	Capacity Remaining (%)	Runtime Remaining (hr)
9.00	100	6.0
7.95	83	5.0
7.70	75	4.5
7.55	67	4.0
7.40	58	3.5
7.28	50	3.0
7.16	42	2.5
7.05	33	2.0
6.94	28	1.5
6.84	17	1.0
6.74	8	0.5
6.64	0	0.0

In the name of science, I sacrificed a new Duracell 9V battery for a battery rundown test. The battery powered the meter without difficulty for 6 hours. In the first hour of use, battery voltage dropped to 7-8 volts, and remained there for most of the test. For this meter, a useful lower limit of battery voltage is 6.6 volts. Below that level, voltage dropped quickly, the regulator could not maintain 5V output, and the meter stopped working. Without the backlight, battery runtime is increased from 6 hours to 10 hours.

We can check the battery voltage in software with the analogRead() function. Provide the function with the pin number, and it will return the voltage on that pin expressed as a value from 0 to 1023, where 0 is 0 volts and 1023 is 5 volts. For example, for a voltage input on pin VBAT:

```
float getVoltage() {                    // get the battery voltage
  int raw = analogRead(VBAT);           // read voltage input as 0-1023
  float voltage = raw*5.0/1024;         // convert to a voltage, 0-5V
  return voltage;                       // and return the result
}
```

The MCU and this function will only work for voltages less than or equal to 5 volts. If 9V from our battery is applied to the MCU directly, it will ruin the hardware. Notice that R5 and R6 are equal in value and convert the 9V input “V_BAT” from the battery into 4.5V “VBAT_ADC” which then connects to the MCU.



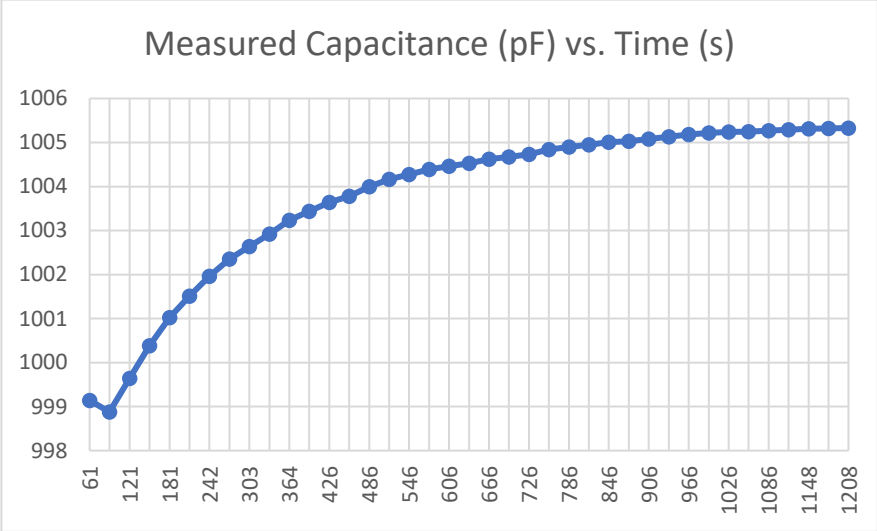
Oscillator Stability and Warm-up

This meter requires 10-15 minutes warmup to achieve its most accurate and stable results.

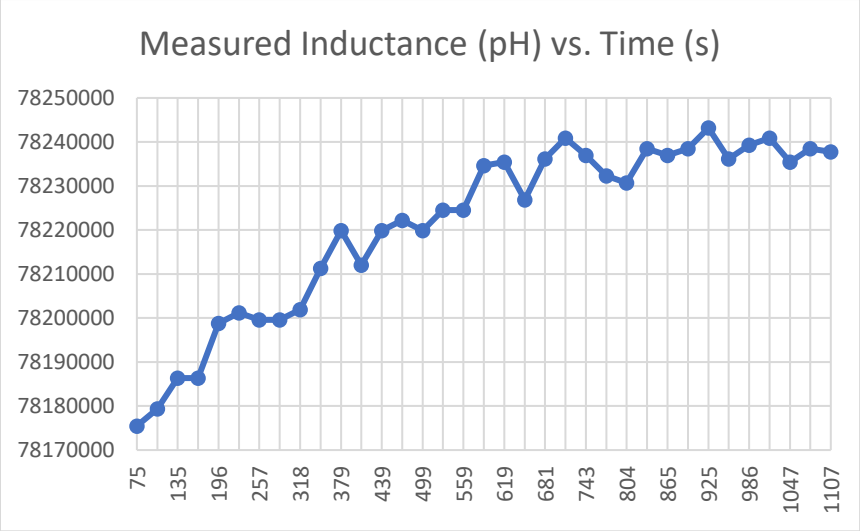
Measurements taken immediately after startup are slightly (<1%) less than those obtained after warm-up.

To measure the effects of warm-up, I started with a "cold" meter that had not been used for more than an hour. I turned on the meter, calibrated, and measured an unknown component every 30 seconds, recording the oscillator frequency and measured value. The data are shown below.

<u>Capacitance Measurement:</u> oscillator frequency peaked 91 seconds after startup, and then exponentially decreased 634 Hz over the next 15 minutes to a resting frequency of 402390 Hz. The resulting capacitance measurement changed from 998.9pF to 1005.3pF over the same time interval, a deviation of 6.4/1005.3 x 100% = +0.64%



<u>Inductance Measurement:</u> the oscillator frequency peaked 100 seconds after startup, and then decreased 94 Hz over then next 15 minutes to a resting frequency of 399074 Hz. The inductance value changed from 78.16 uH to 78.24 uH over the same time interval, a deviation of 0.064/78.24 x 100% = +0.08%.



Last edited: 12/30/2021 3:28 PM