Build your own LC Meter

Bruce E. Hall, W8BH

Part 3: Under the Hood.

Part 1 and Part 2 of this series cover construction and basic operation of the LC Meter. Part 3 reviews the software and shows you how to customize your meter.

Customize your meter

You deserve credit for building your own LC meter, so proudly add your name/callsign to the startup display. To do this, open the LCmeter.ino file in the Arduino IDE and review the list of #define statements near the beginning of the sketch. One of these statements is "#define OWNER". Change the value from "W8BH" to your name or callsign, keeping in mind that only a dozen or so characters will fit comfortably on screen. Here is a quick review of how to upload changes to your meter:

1. Prepare your MCU programmer according to the manufacturer's instructions and connect it to your computer.
2. Confirm an intact USB link between the computer and programmer. If you are using a PC, open the device manager and make sure that your programmer is listed. For example, on my PC the programmer appears as "libusb-win32 devices -> AVRISP mkII"
3. From the Tools menu of the Arduino IDE,
   • Select Board > "Arduino UNO"
   • Select Programmer > AVRISP mkII (or your device)
4. Plug the programmers 6-wire cable onto to the programming header of your LC meter, making sure that all 6 pins are firmly seated. If the programming cable has a red "pin 1" wire it will be facing LEFT toward the DC jack.
5. Apply power to the LC meter. Slide the switch right to the ON position. (At this point, an AVRISP-mkII's LED will turn from red to green, confirming power to the board.)
6. Compile and upload the sketch, using the IDE command "Upload using programmer" <Ctrl><Shift>U.

The startup screen contains 4 customizable items. Here are the defines for all four:

```
#define DEVICE_NAME   "LC meter"            // device name
#define VERSION       "2.0"                 // device version
#define OWNER         "W8BH"                // Your name or call here
#define WARMUP_TIME      8                  // warm up time, in seconds
```

Try changing their values and observing the effect on the startup screen.

Two other #defines deserve mention:

```
#define SHOW_BATTERY   false                 // if true, show battery icon on screen
#define USE_AUDIO      true                  // if true, use piezo element
```

The first of these puts a small battery icon in the top-right corner of the screen, updated every 3 minutes. It is off by default but is useful if you are powering your meter with a 9V battery. The second #define determines whether the meter emits audio feedback. If you don't like your meter beeping at you, feel free to set this #define to false.

Finally, the LC meter uses display color to indicate which mode it's in. By default, startup is white, capacitance mode is yellow, inductance mode is blue, and calibration mode is green. These are all changeable by editing the following defines:

```
#define C_COLOR        TFT_YELLOW            // color for capacitance screen
#define L_COLOR        TFT_CYAN              // color for inductance screen
#define CAL_COLOR      TFT_GREEN             // color of calibration screen
#define STARTUP_COLOR  TFT_WHITE             // color of startup screen
#define BATT_COLOR     TFT_WHITE             // color of battery icon
```

How does it work?

This meter is controlled by two keys. The meter spends most of its time idling, waiting for the user to press one of the keys. The corresponding main code loop is only two lines long:

```
void loop() {                                // wait for user to press a button...
  if (LKeyPressed()) handleLKey();           // do something when L key is pressed
  else if (CKeyPressed()) handleCKey();      // do something when C key is pressed
}
```

When a key is pressed, the handler for the key is called. For this discussion we will focus on inductance, keeping in mind that the capacitance routines are very similar. Here is the L-key (inductance) handler:

```
void handleLKey() {                          // do something when L key is pressed:
  newScreen("Inductance",L_COLOR);           // start new titled screen
  setLEDs(1,0);                              // turn on the L LED
  wait(500);                                 // how long did user press the L button?
  if (LKeyPressed()) doData();               // L long press = Engineering screen
  else doInductance();                       // L short press = Inductance screen
  setLEDs(0,0);                              // turn off the LEDs
}
```

The handler waits for 500 milliseconds, then checks to see if the LKey is *still* pressed. This is how button taps are distinguished from button holds. If the button is held down more than half a second, the meter goes into Engineering mode. Otherwise, it goes into Inductance Mode and measures the unknown device as an inductor.

The work of measuring an inductor is done by the doInductance() routine. After setting up the screen, it does 4 consecutive measurements, one second apart:

```
void doInductance() {
  setMode(L_MODE);                           // relay off when measuring L
  stabilize();                               // allow oscillator time to stabilize
  for (int i=0; i<4; i++) {                  // do 4 measurements, 1 sec apart
    float msmt = getInductance();            // calculate L value
```

```
      showResult(msmt);                          // show msmt on screen
      showUnits(msmt);                           // and show units in title bar
      sendData(msmt);                            // and send it to serial port
      dit();                                     // announce each measurement
      wait(1000);                                // wait one second before next msmt
    }
  }
```

The important lines are highlighted above. The "for" statement creates a loop that is executed four times. The getInductance() routine does all the hard work, calculating the inductance of the device under test. It returns a floating-point number named msmt. That measurement value is shown on the screen by the showResult() routine. The measurements are spaced at 1 second intervals by the wait(1000) statement.

Converting Oscillator frequency into Capacitance & Inductance

To summarize so far: pressing the L-key results in a call to LKeyHandler(), which in turn calls doInductance(). That routine displays the Inductance screen and then makes several calls to getInductance(). So far, so good. But how do we *measure* Inductance, you ask. How does the getInductance() routine work? It works by considering the frequency of the onboard oscillator. The frequency changes when the device under test is added to the oscillator circuit. We can determine the device's value by clever algebraic manipulation of the oscillator frequencies.

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
```

And that's how it's done.

Hardware Timers

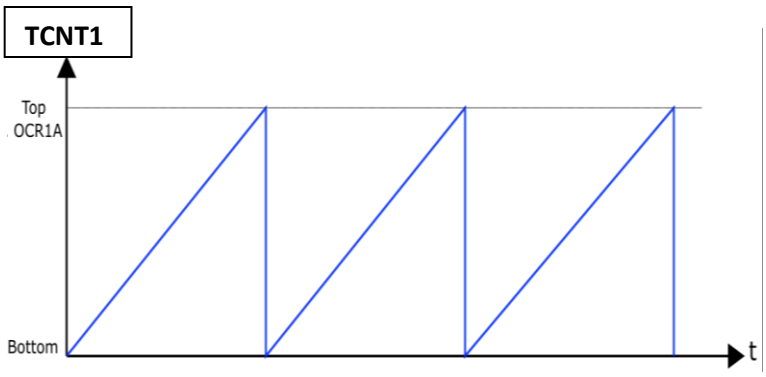
Those calculations are a lot to digest. If you are satisfied, stop here, and continue living your sane and happy life. But you might have a nagging thought: how does this meter determine the oscillator frequencies? It's not a frequency counter. The answer is that we program a poor-man's frequency counter into our LC meter sketch. Continue reading if you are interested in learning how.

Here is the basic idea. 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. Both routines require use of special hardware counters that reside *inside* the MCU.

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 do 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 can we measure a second? Set TC1 to count clock cycles until exactly one second has passed. If our clock is running at 8 MHz, then we must 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
```

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 timing 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 frequency counter sketch using all 3 timers is on my GitHub page as "LCM_Test5".

Last edited: 4/12/2022 11:13 PM