599 Menlo Drive, Suite 100
Rocklin, California 95765, USA
Office: (916) 624-8333
Fax: (916) 624-8003

Sales:sales@parallax.com
Technical: support@parallax.com
Web Site: www.parallax.com

PE Kit Tools: Measure Resistance and Capacitance

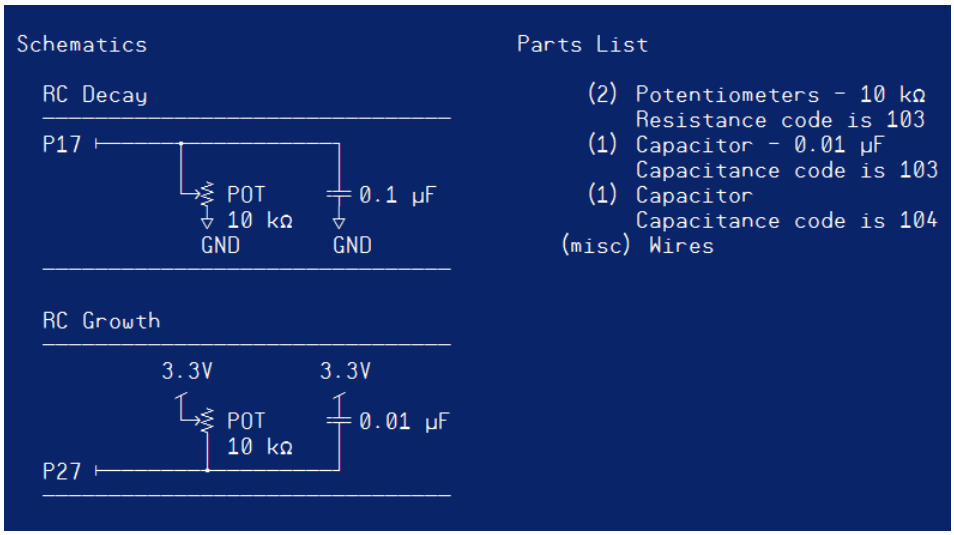
Propeller microcontroller applications that need to measure resistors or capacitors can use the MeasureTime object and a resistor capacitor (RC) circuit to determine their values. Since there's a myriad of sensors whose resistances or capacitances respond to physical properties such as light, rotation, humidity and pressure (to name a few), the simple, inexpensive circuits and easy-to-use object featured in this PE Kit Tools article opens up a whole world of measurement possibilities.

Parts and Circuit

One of the most common variable resistance sensors is the potentiometer, a.k.a "pot". As the knob on the pot is turned, its resistance varies. The Propeller microcontroller can use the MeasureTime object to measure the variable resistors (labeled POT) in Figure 1, which can in turn give the application accurate information about how far each potentiometer knob has been turned. The potentiometer can also be replaced by any number of other resistive sensors. For example, if the pot is replaced with a photoresistor, the circuit can instead be used to measure light intensity. If the pot is replaced with a fixed resistor, variable capacitor sensors that measure pressure or humidity can be measured. The examples in this article will just use a couple of pots with the fixed capacitors from the PE Kit Project parts to test the MeasureTime object. Just keep in mind that the pot is just one example of many sensors that can be monitored by the Propeller chip with an RC circuit and the MeasureTime object.

- ✓ Build the circuits shown in Figure 1.

Figure 1: RC Test Circuits

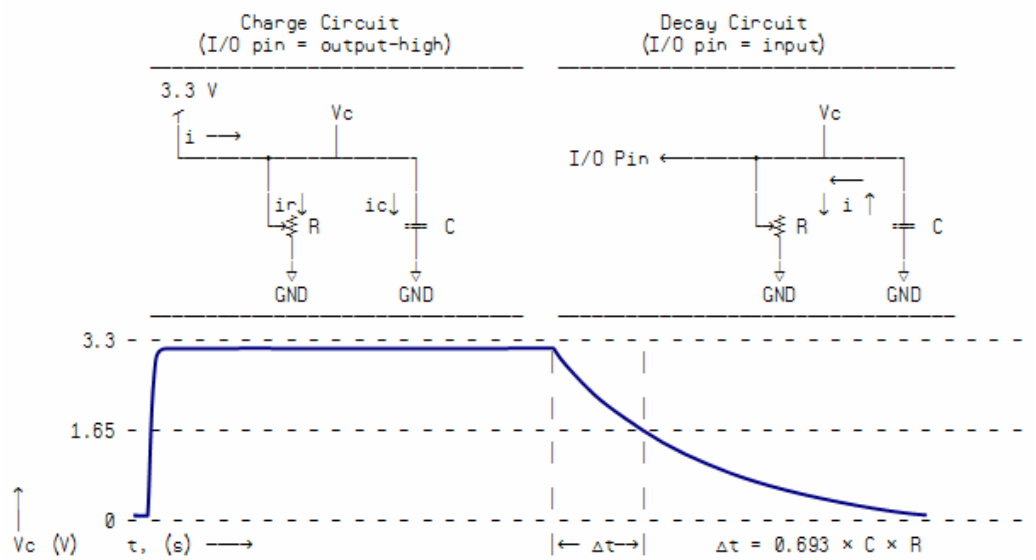


How it Works

The MeasureTime object can be used to determine a variable resistor value by treating the capacitor in the circuit like a small battery. It charges up this capacitor (left side of Figure 2) by sending an output-high signal to the I/O pin. Then, it changes the pin to input and measures the time it takes the capacitor's voltage to decay as it looses its charge through the variable resistor (right side of Figure 2). The decay measurement time (Δt) starts at 3.3 V, and stops when the voltage to decays below the

Propeller I/O pin's 1.65 V logic threshold. For larger resistances, it takes more time for the capacitor to lose its charge. For smaller the resistances, it takes less time for the capacitor to lose its charge.

Figure 2: Microcontroller RC Decay Measurement(1)



(1) Excerpt from *Propeller Education Kit Labs: Fundamentals*

The equation that describes the time it takes for the voltage to decay from 3.3 to 1.65 V is:

$$\Delta t = 0.693 \times C \times R$$

With a little algebra, the terms can be rearranged to solve for the value of R. This is great if the project is to make a simple resistance meter. On the other hand, if the application needs a sensor measurement, it may just scale the time measurement and compare it to some benchmark values. Other sensor applications need to compare the sensor measurement to complex equations, and others still use points from a graph in the sensors datasheet. The application can then check to find out which value in the list is closest to the measured value and so determine the value of the property the sensor measures.

!	**Actual threshold voltage varies** between I/O pins and Propeller chips. So, if you use this technique to make a resistance or capacitance meter, some calibration will be necessary.

Simple Test Code

The MeasureTime object has lots of tools for measuring RC voltage growth and decay in circuits. In its simplest form, code that measures the circuits in Figure 1 resembles PBASIC RCTIME commands for the Parallax BASIC Stamp microcontroller. After declaring the MeasureTime object, the code passes the pin, voltage state of the circuit at the start of the measurement, and the address of the variable where the Rc method should store the result. The MeasureTime object uses these parameters to charge the circuit, measures the decay, and store the result in the appointed variable, tDecay in the first method call, and tGrowth in the second.

```
OBJ
  time : "MeasureTime"

PUB Go | tDecay, tGrowth
  time.Rc(17, 1, @tDecay)
```

```
time.Rc(27, 0, @tGrowth)
```

MeasureTime.zip file has both Spin and ASM versions of the MeasureTime object along with several test code examples. The first code example you should try is "Test Simple RCTIME.spin".

- ✓ Download and unzip "PE Kit Tools – Measure Resistance and Capacitance.zip".
- ✓ Open "Test Simple RCTIME.spin" with the Propeller Tool software.
- ✓ Open the Parallax Serial Terminal, and set the COM Port to the Propeller chip's programming port. (You can use F7 in the Propeller Tool to find out which port that is.)
- ✓ In the Propeller Tool, load the Test Simple RCTIME object into the Propeller chip with F11
- ✓ Immediately after you have pressed the F11 key in the Propeller Tool, click the Parallax Serial Terminal's Enable button. The Parallax Serial Terminal will wait for the Propeller Tool software to finish loading code into the Propeller chip before it connects to the COM port.

"Test Simple RCTIME.spin" displays the decay and growth time measurements in the Parallax Serial Terminal. These growth and decay times are in terms of 12.5 ns units. That's because this program has the Propeller chip's system clock set to 80 MHz, and if the clock is ticking at 80 million times per second, the time between each tick is 12.5 ns.

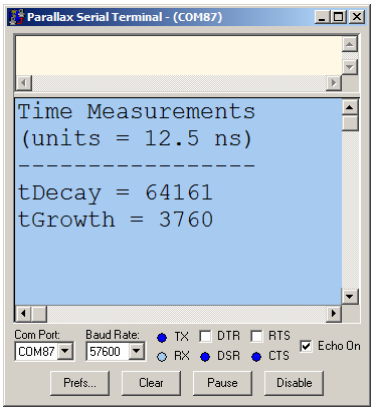


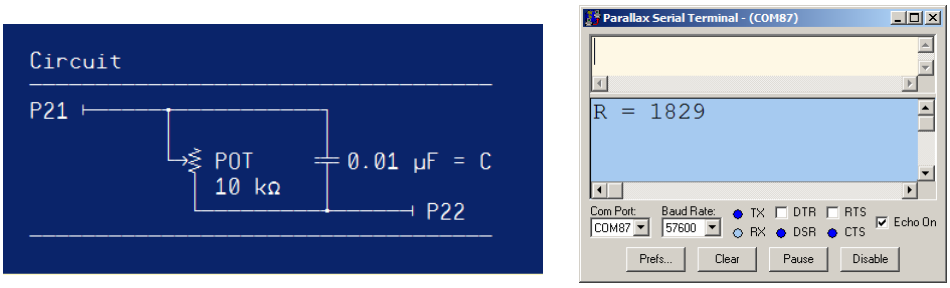
Figure 3: RC Decay and Growth

The range of decay times should be about ten times the range of growth times since the decay circuit has a capacitor that's ten times as large as the one in the growth circuit. Since the decay circuit's capacitor can store ten times the charge, the voltage will take ten times as long to decay through the same size resistor. This can be verified by setting the potentiometers to roughly the same position. The tDecay value should be about ten times the tGrowth value. There will be some variation, especially since the threshold voltage is not likely to be exactly 1.65 V. For example, if the threshold voltage instead 1.55 V, the decay time will be longer and the growth time will be shorter. Reason being, the decay will have to drop from 3.3 V down to 1.55 V, which is a 1.75 V decay. Meanwhile, measuring the growth will only be a 1.55 V voltage rise, and a very different growth time measurement.

Application Example: Resistance Meter

The "RC Resistance Meter.spin" application uses an RC circuit connected to two I/O pins to measure and display resistance measurements. The pins that would get connected to ground in a decay circuit get connected to another Propeller I/O pin instead. In Figure 4, P21 takes the two different RC measurements. First, the application grounds P22 and takes a decay measurement with P21. Then, it ties P22 to 3.3 V and takes a growth measurement with P21. The average of these measurements helps compensate for any deviations from the nominal I/O pin threshold voltage of 1.65 V.

Figure 4: Resistance Meter Circuit and Display (1.829 kΩ)



The application's RcResistance object uses the MeasureTime object to take one hundred of these growth/decay measurements and averages them to eliminate any noise. It then uses the Propeller Library's FloatMath object to calculate the resistance R of the pot.

$$R = \frac{\Delta t}{0.693 \times C} \times CF$$

In the case of this application, the term Δt is calculated by dividing the clock frequency (clkfreq) into the number of clock ticks in the averaged growth/decay measurement. C is the capacitance (0.01e-6 for 0.01 µF), and CF is a correction factor that can be used reduce any remaining error in the measured vs. actual resistance value.

$$CF = 1 + \frac{R(known) - R(measured)}{R(measured)}$$

- ✓ Open "RC Resistance Meter.spin" with the Propeller Tool software.
- ✓ Load RC Resistance Meter into the Propeller chip with F11.
- ✓ Immediately after you have pressed the F11 key, click the Parallax Serial Terminal's Enable button.
- ✓ Test the value of a known resistor.
- ✓ Solve for CF, and substitute that value in the RC Resistance Meter object's COR_FACTOR constant.
- ✓ Load the modified program into the Propeller chip and try a variety of resistance measurements.

Setting Timeout Values for RC Measurements

Let's say a circuit will decay under most circumstances, but not always. What happens then? Some objects that measure RC decay will wait indefinitely. In contrast, the MeasureTime object has a configurable timeout that defaults to 10 ms to prevent this problem. This configurable timeout also prevents the application from having to wait an unnecessarily long time for circuits that are responding slowly due to a large resistance or capacitance value. In many cases, a slow RC response indicates a throw-away measurement any how. A photoresistor in complete darkness is an example. It can really slow down an RC measurement due to large resistance values, but maybe the application only cares that it's beyond a certain level of darkness. At that point, the application might chose to hibernate until morning, or maybe turn on the lights!

Both the MeasureTime and MeasureTime.ASM objects have default timeout values of 10 ms. For MeasureTime.ASM, the timeout value is just as accurate as the decay measurement itself, good to the nearest clock tick. This is possible because MeasureTime.ASM uses assembly language to take the growth and decay measurements; whereas, MeasureTime takes its decay measurements in Spin.

Since Spin is an interpreted language, it does not provide the same degree of control over timing that assembly language does. Keep in mind, both objects' growth and decay measurements are good to the nearest clock tick. The difference between the two objects is that the MeasureTime object's timeout value is approximate; whereas, MeasureTime.ASM's timeout value is exact.

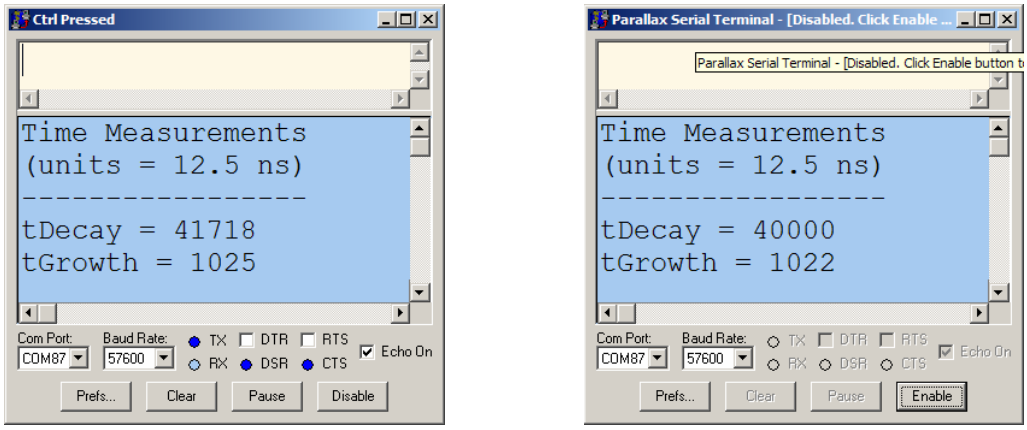
Changing either object's timeout value involves a simple method call to its TimeOut method. The examples below impose a 0.5 ms timeout on RC measurements for the Figure 1 circuits. Each code example below passes clkfreq/2000 to the MeasureTime object's TimeOut method, after which, all the measurements in the repeat loop are subject to that 0.5 ms timeout.

```
OBJ
  time : "MeasureTime"

PUB Go | tDecay, tGrowth
  time.TimeOut(clkfreq/2000)
  repeat
    time.Rc(17, 1, @tDecay)
    time.Rc(27, 0, @tGrowth)
    ' ...
```

```
OBJ
    time : "MeasureTime.ASM"

  PUB Go | tDecay, tGrowth
    time.TimeOut(clkfreq/2000)
    repeat
      time.Rc(17, 1, @tDecay)
      time.Rc(27, 0, @tGrowth)
      ' ...
```

Figure 5 shows how the Spin MeasureTime object's tDecay timeout is approximate while the assembly language version is accurate to the clock tick. The code above configures both versions of the object to time out at 40_000 clock ticks (0.5 ms at 80 MHz). Keep in mind that for most applications, the actual decay measurement is the important part, and that both objects will return the exact same value provided when the measurement is below the timeout.

Figure 5: Timeout Display for Spin (left) and Assembly Language (right)



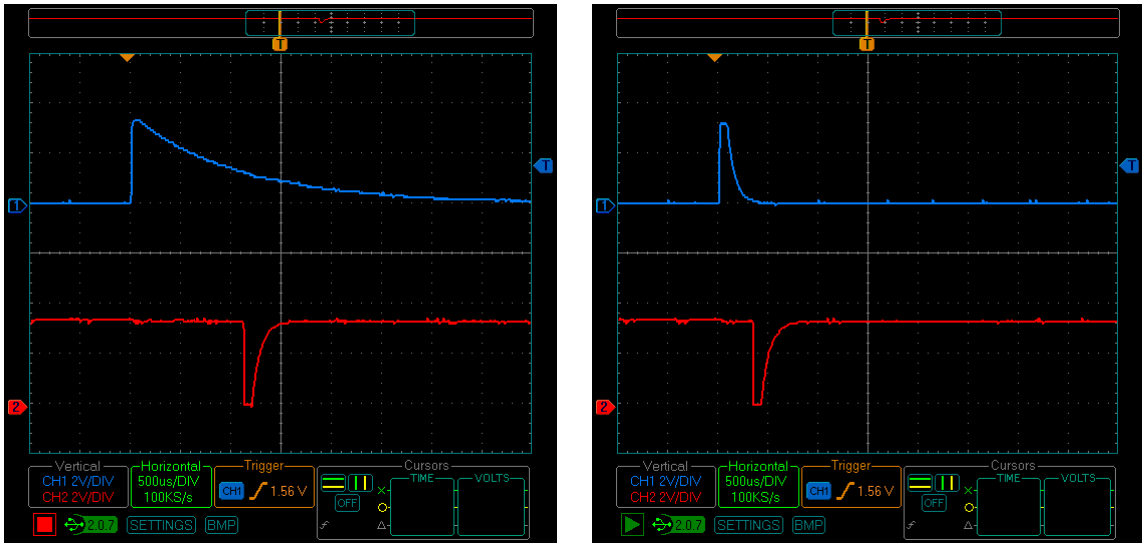
- ✓ Test both "Test RCTIME Timeout.spin" and "Test RCTIME Timeout.ASM.spin"objects, and verify that they take the same measurements when the timeout values are below the 0.5 ms threshold.
- ✓ Set the potentiometer connected to P17 so that it causes both objects to time out and verify that the assembly language version of the object can enforce the timeout.

Sequential vs. Parallel RC Measurements

The MeasureTime objects default to sequential measurements. In sequential mode, the MeasureTime object does not return from the Rc subroutine call until the measurement is complete. Figure 6 shows what happens when the MeasureTime object's Rc method gets called twice in immediate

succession like it does in "Test Simple RCTIME.spin". On the left, the first measurement takes just over 1 ms to complete, and the second measurement does not start until the first measurement finishes. On the right, the first measurement takes less than 250 µs to complete, so the application moves on to the second measurement more quickly.

Figure 6: Sequential Measurements



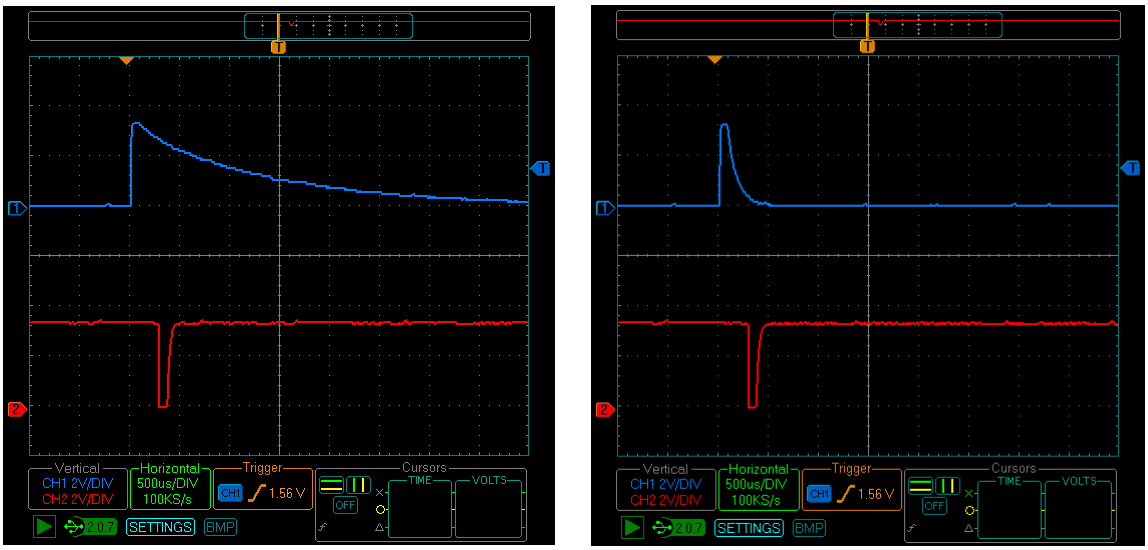
The MeasureTime object can be configured to launch multiple measurements in parallel. For parallel measurements, a copy of the MeasureTime has to be created for each simultaneous measurement, and then each copy of the object has to be configured to return immediately after starting its RC measurement.

Here is an excerpt from "Test Parallel RCTIME.spin". It declares two copies of the MeasureTime object and then configures each copy for taking parallel measurements by passing time#PARALLEL to its SetMode method.

```
OBJ
  time[2] : "MeasureTime"

PUB Go | tdecay, trise, i
  repeat i from 0 to 1
    time[i].SetMode(time#PARALLEL)
  '...
  time[0].Rc(17, 1, @tdecay)
  time[1].Rc(27, 0, @tGrowth)
  'Code here can work on other tasks while
  'waiting for the measurements complete...
```

With two copies of the MeasureTime object configured to take parallel measurements, the second measurement starts immediately after the first one does, so the duration of the first measurement no longer affects when the second measurement starts. This approach can be useful for making sure measurements start at approximately the same time, and it can also be useful for saving time by taking measurements in parallel. The drawback is that each measurement launches a separate cog for the duration of the measurement. This drawback is not severe because the MeasureTime object also shuts down a given cog immediately after the measurement is complete. Keep in mind though, that it your application launches four simultaneous RC decay measurements, there will be a brief period of time where there are four cogs used up by these simultaneous measurements.

Figure 7: Parallel Measurements



If you have an oscilloscope:

- ✓ Examine how the duration of the first measurement can delay the start of the second measurement in "Test Simple RCTIME.spin" while the second measurement in "Test Parallel RCTIME.Spin" starts a short, fixed time after the first measurement starts.

Assembly code optimization: The MeasureTime.ASM object is in its 0.65 revision, which is the first working version. The assembly code is currently a lot longer than it needs to be because it has not undergone any optimizations. Before it gets to v1.0, it will undergo additional testing and several iterations of assembly code optimization.

Establishing Sampling Rates

The MeasureTime object also has Start and Stop methods. The Start method makes it possible to make one or more copies of the MeaureTime object take growth/decay measurements at one or more different rates. When the application doesn't need any more measurements, the Stop method can be used to shut down the cod and make it available for other tasks.

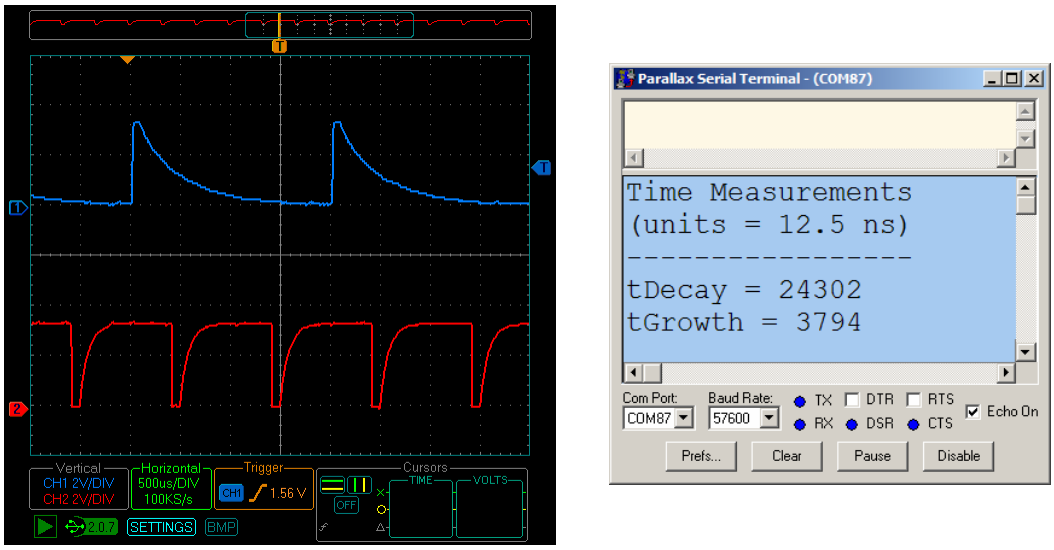
The MeasureTime object's Start method requires three additional parameters, charge time, timeout, and sample interval. The sample interval is the time between measurements, and it establishes the sampling rate. In the example code below, time[0] is configured with a 0.1 ms charge time, a 1.0 ms timeout, and a 2 ms sample interval (clkfreq/500). Time[1] also has a 0.1 ms charge time, but its time out is 0.5 ms and its sample interval is 1 ms (clkfreq/1000).

```
OBJ
  time[2] : "MeasureTime"
  display : "PstRcDisplay"

PUB Go | tdecay, trise
  time[0].start(17, 1, clkfreq/1_0000, clkfreq/1000, clkfreq/500, @tDecay)
  time[1].start(27, 0, clkfreq/1_0000, clkfreq/2000, clkfreq/1000, @tGrowth)

  repeat
    waitcnt(clkfreq/10 + cnt)
    display.Update(tDecay, tGrowth)
```

After the Start method for each MeasureTime instance gets called, the repeat loop can just send the latest measurement stored in each variable (tDecay and tGrowth) to the display object for viewing in the Parallax Serial Terminal. One of the parameters in the start method is the address of the variable that the MeasureTime object should store its result in. Since both instances are taking repeated measurements, the tDecay and tGrowth measurements both store the latest value. tDecay gets updated at 500 Hz, and tGrowth gets updated at 1 kHz.

Figure 8 shows how the two instances of the MeasureTime object, configured to independent sampling rates, and both repeatedly take RC measurements. The upper trace shows time[0], which repeats its measurements at a T = 2 ms sampling interval. Since sampling frequency is the inverse of sampling interval, (f = 1/T), f = 1/(2 ms) = 500 Hz. The lower trace shows the measurements time[1] takes and stores in tGrowth at a sample interval of 1 ms and a sampling frequency (or sampling rate) of 1 kHz.

Figure 8: Two Different Sampling Rates and the Parallax Serial Terminal Results



- ✓ Open "Test Repeated RCTIME.spin" with the Propeller Tool software.
- ✓ Open the MeasureTime object and examine the minimum times allowed for the start method's chargeTimeTicks, timeOutTicks, and sampleTicks parameters.
- ✓ Try a variety of sampling rates, and if you have an oscilloscope, use it to examine the repeated RC measurements.

200 kHz Sampling Rate Example

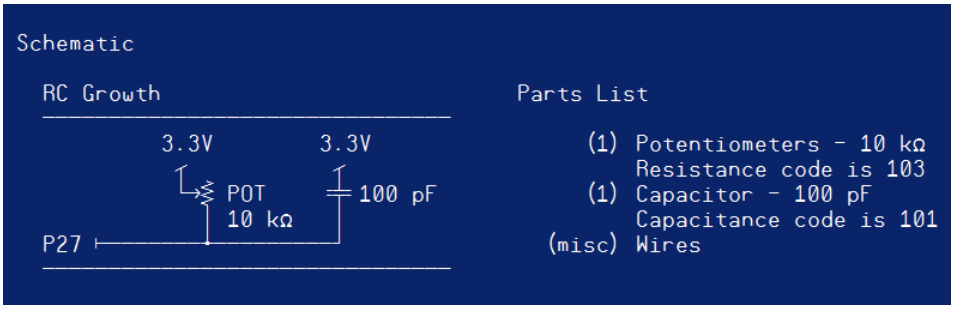
If you need a sampling rate that's faster than what MeasureTime.spin can provide, use MeasureTime.ASM.spin. It can support sampling intervals as short as a few hundred clock ticks, as opposed to the several thousand minimum in the MeasureTime.spin object's documentation. Let's say that your application requires a 200 kHz sampling rate. The sampling interval is T = 1/f = 1/200 kHz = 5 µs. Assuming the Propeller chip's system clock is running at 80 MHz, the number of clock ticks in the sampling interval would be:

```
sample interval clock ticks = ticks in 1 second × sample interval
                            = 80 MHz × 5 µs
                            = 400 clock ticks
```

Figure 9 shows an RC growth circuit that responds 100 times more quickly than the circuit in Figure 1. That's because the capacitor is 1/100 the size of the one in Figure 1. Instead of growth times in the 0 to 5000 clock tick range, the measurements will be in the 0 to 50 clock tick neighborhood.

- ✓ Modify the RC growth circuit connected to P27 according to Figure 9.

Figure 9: A Faster RC Circuit



These excerpts from "200 kHz Sampling rate.spin" configure the MeasureTime.ASM object to charge the circuit's capacitor for 50 clock ticks, and allow 125 clock ticks for the decay, repeating every 400 clock ticks.

```
OBJ
  time      : "MeasureTime.ASM"
  display   : "PstRcDisplay2"

PUB Go | tdecay, tgrowth, repsAddr

  display.Start

  repsAddr := time.start(27, 0, 50, 125, 400, @tgrowth)

  repeat
    waitcnt(clkfreq/10 + cnt)
    display.Update(long[repsAddr], tgrowth)
```

According to the MeasureTime.ASM object's documentation, its Start method returns the address of the object's repsAddr variable, which stores the number of samples the object has taken. (Unless all cogs were busy, in which case the Start method returns zero.) Example code in the "200 kHz Sampling Rate.spin" object uses this variable address along with a modified versions of the Display object to list the number of measurements (reps) that MeasureTime.ASM has taken. Every second, the repetitions display increments by 200,000. Figure 10 shows the display at about the 10 second mark.

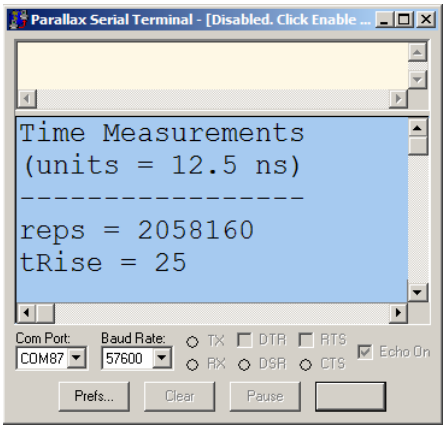
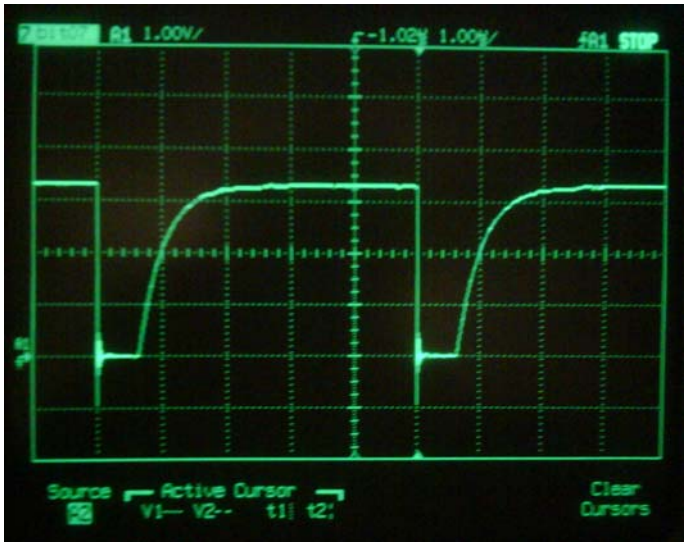


Figure 10: RC Decay and Growth

Figure 11 shows the RC measurements on a 100 MHz oscilloscope set to 1 µs/division. Note that the I/O pin switches to output-low to recharge the circuit every 5 µs demonstrating the 200 kHz sampling rate.

Figure 11: Oscilloscope View of 200 kHz Sampling Rate.Spin



- ✓ Open "200 kHz Sampling Rate.spin" with the Propeller Tool software.
- ✓ Try a variety of sampling rates, and if you have an oscilloscope, use it to examine the repeated RC measurements.