Measuring Duty Cycle with Nextest testers

By Dan Bullard, Senior Marketing Applications Engineer, Nextest Systems
5/30/2006

In previous applications notes I have covered frequency measurement and these techniques have allowed several users to measure frequency accurately and fairly quickly without adding hardware to the tester or the loadboard. However, those techniques only reported back the frequency of the captured clock, they did not attempt to measure duty cycle, nor were they able to. All of those routines assumed that the clock's duty cycle was 50%, if it deviated significantly from that there may be some difficulty in resolving the frequency. This applications note discusses how to measure the duty cycle of a free-running clock waveform and also how to get an accurate frequency measurement of a clock that is not a 50% duty cycle waveform.

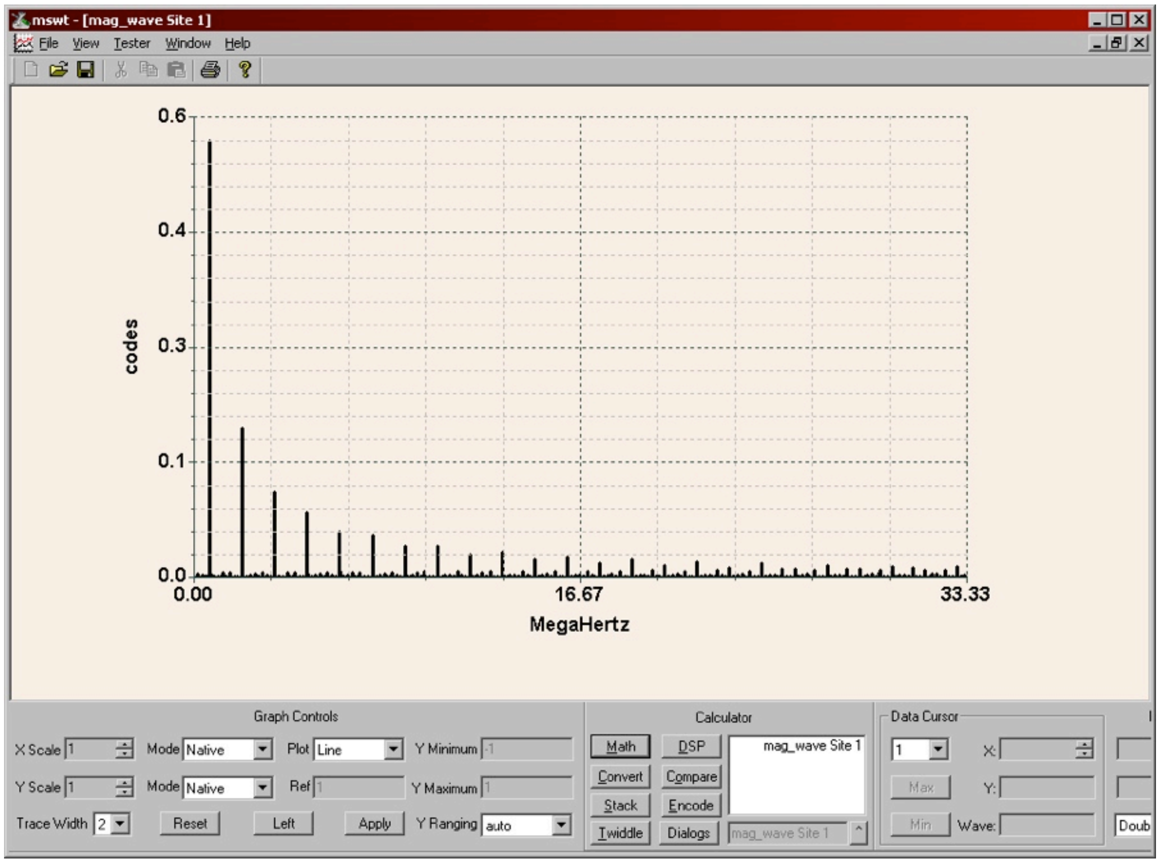
Refresh Time

To refresh your memory about Frequency measurements on Nextest Maverick and Magnum testers, we captured the clock using a digital pin and stored the resulting stream of 1s and 0s in the Error Catch Ram or ECR. In some cases we used the DCI which is the same thing, but just a little easier to use. The capture was done non-coherently, sometimes in normal mode, sometimes in DDR, and sometimes we captured twice, allowing us to deal with very high frequencies by correlating the aliased frequency bins back to the original, very high frequency. Note that we are capturing digital only data, so the captured waveform consists only of 1s and 0s, there is no amplitude information at all in the captured data.
In all cases our technique relied on the fact that the spectrum of a 50% duty cycle square wave consisting of only 1s and 0s would reveal the fundamental frequency as the highest amplitude bin in the spectrum. This is a perfectly reasonable assumption, unless the duty cycle deviates significantly from 50%.

Duty cycle versus Harmonic content

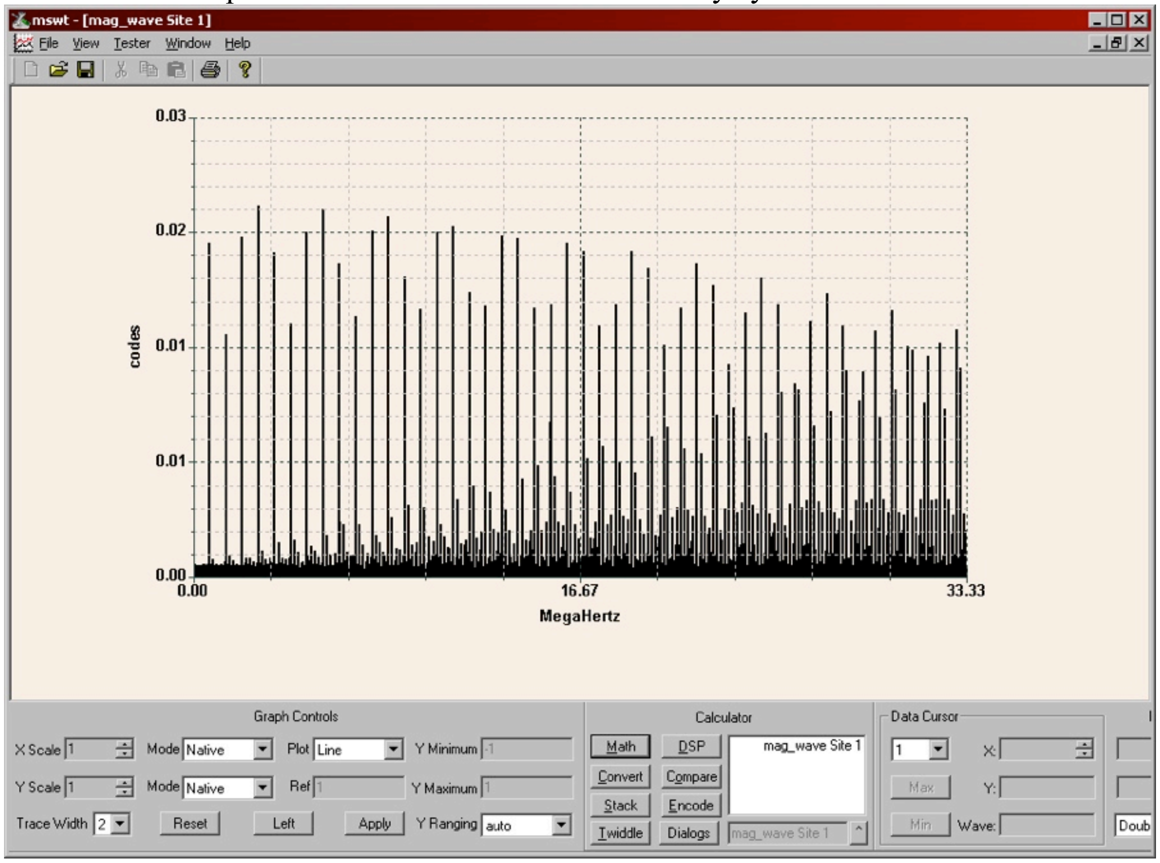
Everyone knows (or should know) that a squarewave consists of a fundamental frequency summed together with odd harmonics in amplitudes inversely proportional to their harmonic number. For example, if a 1MHz squarewave has a fundamental amplitude of 1.0V, Fourier analysis will reveal a spectrum that contains the 3rd harmonic, or 3MHz with an amplitude of 1/3 or 0.3333V, the 5th harmonic with an amplitude of 1/5 or 0.2V, the 7th harmonic with an amplitude of 1/7 or 0.142857v, and so on. There will be no even harmonic energy at all, as long as the duty cycle of the squarewave is exactly 50%. If however, the duty cycle is not exactly 50%, the 2nd, 4th, 6th, and subsequent even harmonics will start to appear and in some cases get quite high relative to the fundamental amplitude. What is worse (for us) is that if this waveform we are capturing is free-running, it is necessarily non-coherent. If the duty cycle gets very low, the second and/or fourth harmonic will become quite large, but generally they are not larger than the

fundamental. Remember that a VERY low duty cycle waveform is an impulse, and its spectrum is flat, all bins are at the same amplitude. However, because of the non-coherent nature of our capture, the individual bin magnitude of the some harmonic bins might overpower the bin magnitude of the fundamental, making it appear that the dominant frequency captured is a multiple of what it actually is. Since we generally use the waveform_min_max() function to tell us what the dominant (fundamental) frequency is, we can be fooled by this phenomenon.
Take for example, the spectrum of a 700KHz waveform with a 50% duty cycle captured in DDR mode on a Maverick.



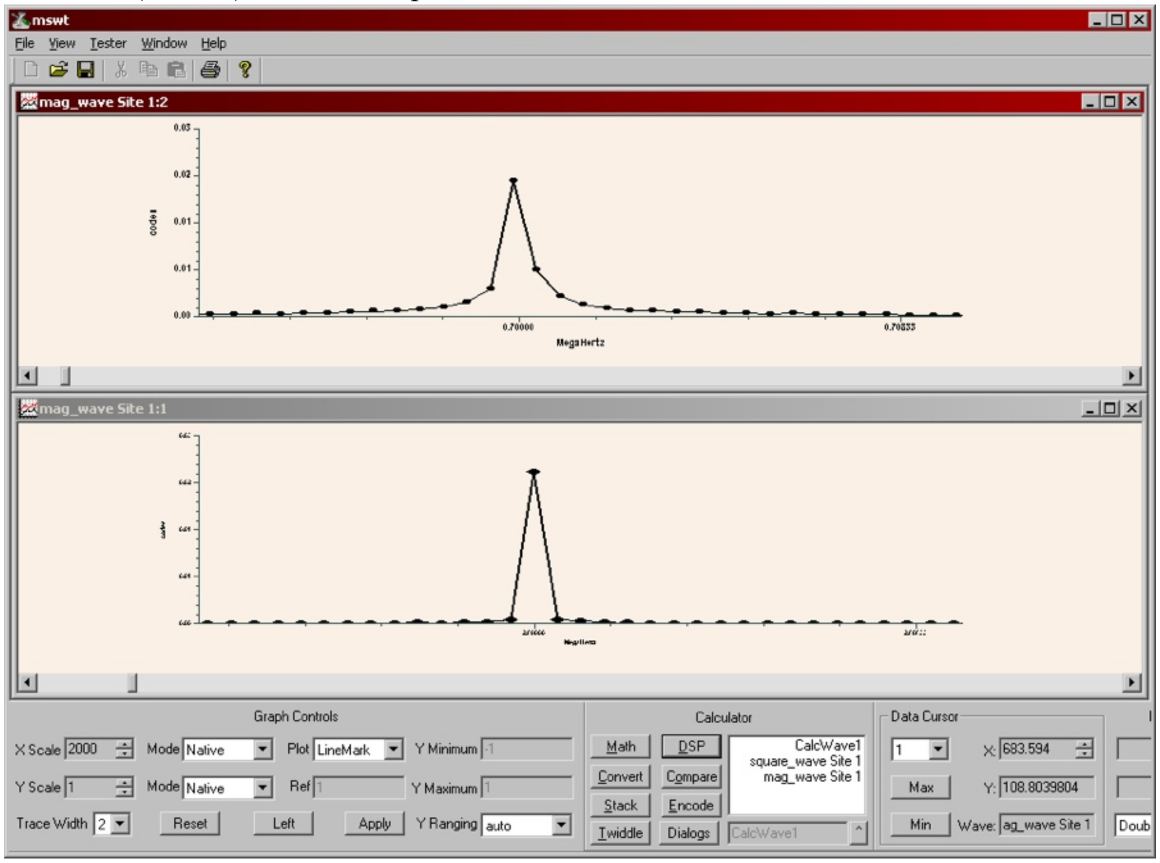
You can clearly see the fundamental on the left hand side, and you can see the odd harmonics decreasing in amplitude off to the right. What are those very small bins popping up between the odd harmonics? Are they even harmonics? Not at all, those are aliasing odd harmonics that bounced off the Nyquist frequency at 33.33MHz (we are sampling at 66.66MHz). In fact, they also bounced off of DC (frequency 0.00) and are headed back up. They keep bouncing up and down between DC and the Nyquist frequency forever and ever. Remember, there is no amplitude information in this waveform, so there is no noise, except for tiny amounts due to jitter and runt pulses that may have been captured because of noise tripping a comparator on a rising or falling edge.

Now here is the spectrum of a waveform with a 1% duty cycle.



You can clearly see the harmonics trailing off from left to right, (and right to left as the aliases trail off after bouncing off of 33.33MHz) but you can also tell that the lowest frequency bin (700KHz on the far left) appears to be smaller than many of its harmonics. If we were to use the waveform_min_max() function to find the maximum amplitude bin in our effort to find the dominant frequency, we would arrive at an answer of 2.8MHz, or four times 700KHz, since it appears that the fourth harmonic is larger than the fundamental at 700KHz. The fact is however, that the fourth harmonic is not necessarily larger in amplitude, it’s more likely that the energy in that bin location is more concentrated.

Look what happens when we zoom in on the fundamental (top) bin and the fourth harmonic (bottom) bin and compare them.



X Scale 2000 Mode Native Plot LineMark Y Minimum -1

Y Scale 1 Mode Native Ref 1 Y Maximum 1

Trace Width 2 Y Ranging auto

CalcWave1
square_wave Site 1
mag_wave Site 1

CalcWave1

1 X: 683.594

Y: 108.8039804

Wave: ag_wave Site 1

Notice how the fundamental bin is more spread out, whereas the fourth harmonic bin is nearly straight up and down. The reason for this is that just due to luck (or lack thereof) we captured 1376.25 cycles of the fundamental, but that means we captured 4 times that number, or 5505.0 cycles of the fourth harmonic. Since we captured a whole number of cycles of the fourth harmonic, most of its energy is in a single bin, whereas, even though the actual amount of 700KHz energy is probably larger, it's all spread out because it wasn't nearly as coherent.

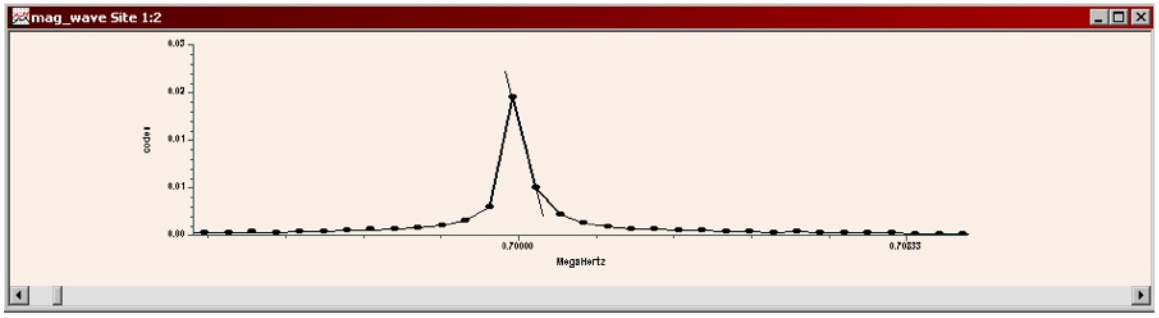
Bunnemann for interpolation

You might wonder how I knew that my capture contained 1376.25 cycles of the fundamental. I used a technique called Bunnemann to figure out how many cycles and partial cycles I captured. The Bunnemann technique consists of finding the two highest points in a spectrum near the frequency of interest and then performing a non-linear interpolation to find which *fractional* bin the signal belongs to. Since an FFT does not deal in fractional bins, only in integer bins, this technique can be very useful for getting a much more accurate frequency measurement than would normally be allowed with standard frequency domain techniques.

The Bunnemann formula looks like this:

```
double bin_num = (double)left_bin+atan(sin(pi/N)/(cos(pi/N)
+left_mag/right_mag))/(pi/N);
```

where left_bin is the lowest frequency bin (left-most) of the two highest amplitude bins near a peak, and the leftmag is the magnitude of the leftmost bin. Right_mag is the magnitude of the bin to the right (higher in frequency) of the left_bin. The code for this complete function is listed later so you don't have to worry too much about the details. Remember the spectrum of 700KHz fundamental we saw above? Well, here is another look at it.



What the Bunnemann function does is interpolate between the two highest points in the spread out bin based on the actual bin number and the ratio between the two highest bins. The highest amplitude bin here is bin 1376. The next highest bin is bin 1377. The Bunneman function tells us that if we could have fractional bins in our spectrum, this frequency would fall in bin 1376.25. That means that our capture contains 1376.25 cycles of the fundamental frequency, it also means that if we multiply 1376.25 by the Fourier Frequency, we can arrive at a much more accurate answer for our frequency measurement. How does that help us in determining duty cycle?

Finding whole cycles

Knowing that we captured non-coherently we can't use the most obvious method of estimating duty cycle. The most obvious solution is to find the arithmetic mean of the captured waveform. Remember that there is no amplitude information in this waveform, since it was digitized with a single bit. If the waveform was exactly 50% you would expect the average voltage to be 0.5V, with 50 percent of the samples high and 50 percent of the samples low. If the waveform had a 10% duty cycle you would expect that the average voltage would be 0.1V, with 10 percent of the samples high and 90 percent of the samples low.

The only problem with this idea is that because we captured non-coherently, we have a partial cycle and we don't know whether that partial cycle is all high, all low, or some percentage of either. If only we knew how many samples we had that contained partial cycles we could trim those samples out of the waveform and then we would have a waveform that contained almost exactly an integer number of cycles.

Well, since we can use Bunnemann to tell us how big our partial cycle is, we can calculate how many samples to trim off. It is pretty easy really, we work out the equivalence thus:

$$\frac{\text{Fractional Cycles}}{\text{Total Samples}} = \frac{\text{Whole Cycles}}{\text{Partial Samples}}$$

Since we know how many fractional cycles we have (1376.25) and the total number of samples (131072) and we know how many whole cycles we would like to have ((int) fractional_cycles) or 1376, then we can use the function waveform_subset() to cut the waveform down to 1376*131072/1376.25 or 131048. Now there is a remainder of .1904 samples, there isn't much we can do about that, so that will add to our error, but it will be quite small relative to our duty cycle versus the error we would have been dealing with of a quarter of a cycle. This error is 0.000138 cycles, since each cycle occupies 1376 samples and our error was 0.1904 samples. That comes out to a time error of 0.000138/(1/700KHz) or 197ps for a signal that has a 1.429us cycle time. That is such a small portion of the cycle it's not going to contribute hardly anything to our duty cycle measurement.

Now that we have trimmed our sampled waveform down to very close to a whole number of cycles, we perform an arithmetic mean operation which returns that average value of the waveform. That translates directly into duty cycle.

It's high time that we measured high time

Speaking of time measurements, maybe you wanted to know low time and high time. It is easy to convert duty cycle into high time and low time directly, assuming you know the frequency very accurately. Well, since you ran the Bunneman function you know the frequency with very high accuracy, it was the fractional bin number times the Fourier Frequency. Once you know that, you can use T=1/F to calculate the period of the waveform. Now, duty_cycle times period = high_time, and 1-duty_cycle times period is low_time. Remember, in our case our accuracy was very good, about two hundred pico-seconds for a 1.5 micro-second measurement, or about 0.013% error.

The final wrinkle

Earlier I made a big deal about the fact that one of the higher harmonics might fool the waveform_min_max() routine into coming up with the wrong frequency. As it turns out, it doesn't really matter to the duty cycle what the measured frequency is, it's only when you try to translate duty cycle into time that you may get the wrong answer, since you need the correct frequency number to calculate period.

So, if all you want is the duty cycle, there is no need to worry about what the fundamental frequency actually is. If the min_max routine picks up a harmonic, so what? The fourth harmonic is even nicer (in my case) because it is nearly coherent (purely by chance I might add). However, if you want to know the frequency (and you usually do) and you want to know the low time and the high time, you need another little trick up your sleeve.

That trick is pretty simple, you need to estimate the frequency before you go off and run the FFT so you know approximately where to look in the spectrum for the fundamental. How do you do that? It comes down to counting edges. You need to count how many edges you have using a brute force method of scanning the waveform data and counting each transition from a 1 to a 0 and a 0 to a 1. At the end, divide by 2.0 and then divide by the Unit Test Period (UTP). UTP is the reciprocal of the Fourier Frequency, it is the total amount of time you were sampling. You can calculate that by dividing the number of samples you took by the sample frequency. In my case I captured 131072 samples at a rate of 66.66666MHz, so my UTP was about 2ms.
So, who needs a TMU to measure pulse width, duty cycle, frequency or period. As long as the waveform you want to measure repeats, you can measure all these things, accurately and pretty quickly too. You pay a penalty when you have to measure high time and low time with a low-duty cycle waveform because you have to count edges and this takes time. But at least you can test the part, and you don't have force the customer to buy a Femto-2000.

Program Listing

Below is a complete program that uses the AWI on a Lightning to generate a variable duty cycle waveform with programmable frequency. The waveform is captured by a digital pin running in DDR mode and measured for frequency, duty cycle, high time and low time. It has options to skip the frequency estimation step so you can see how it works without that, and it has an option for accumulating stats on variance when the program is run over and over. It also has an option to step the duty cycle from 1% to 99% and back (if repeated enough times) so you can see how it behaves with various duty cycle waveforms.

```
#include "tester.h"
#include "patterns.h"
#include "pin_lists.h"
#define samples 131072

int round(double var)
{
      return (int)(var+0.5);
}

static double min_high_time = 1.0e19;
static double max_high_time = 0.0;
static double min_freq = 1.0e19;
static double max_freq = 0.0;

static int auto_dc;
static int num_fails = 0;


DOUBLE_VARIABLE(threshold, 1.0, "Threshold") {}
DOUBLE_VARIABLE(IOH_IOL, 1.0, "Load currents in mA") {}
INT_VARIABLE(duty_cycle, 50, "Duty Cycle") {}
```

```
DOUBLE_VARIABLE(FsubT, 700.0, "Test Frequency(in KHz)")
{
      if((FsubT>800.0) || (FsubT<10.0))
      {
            output("Error, allowed range of FsubT is 10.0 to 800.0
(KHz), resetting to %3.1f", oldval);
            FsubT = oldval;
      }
}

DOUBLE_VARIABLE(calibration_val, 1.000, "Calibration factor") {}
BOOL_VARIABLE(collect_stats, FALSE, "collect_stats")
{
      if(oldval &&!collect_stats)
      {

            min_high_time = 1.0e19;
            max_high_time = 0.0;
            min_freq = 1.0e19;
            max_freq = 0.0;
      }
}

BOOL_VARIABLE(use_dans_square, TRUE, "Use Dans squarewave generator"){}

BOOL_VARIABLE(auto_set_dc, FALSE, "Automatically increment duty cycle")
{
      if(!oldval && auto_set_dc)
            auto_dc = 1;
}

BOOL_VARIABLE(get_estimated_freq, TRUE, "Try to measure frequency
first"){}

AWI_ASSIGNMENTS( awis )
{
      INSTRUMENT( source1, t_mspc4, id_awi1 )
}

SINGLE_AWI_LIST( source1 )

WAVEFORM(captured_wave1){}
WAVEFORM(fft_wave){}
WAVEFORM(mag_wave){}
WAVEFORM(subset){}
WAVEFORM(square_wave){}

WAVEFORM(subset_wave){}

void dans_generate_squarewave(Waveform *wave,
                                            int wave_samples,
                                            MKSFrequency sample_freq,
                                            double low_level,
                                            double high_level,
                                            MKSFrequency test_freq,
                                            double duty_cycle )
{
      int i, trans_point;
      trans_point = (round((duty_cycle/100.0)*wave_samples))-1;

      if(waveform_get_size(wave) != wave_samples) waveform_generate_DC
(wave, wave_samples, sample_freq, low_level);
      for (i=0; i<wave_samples; i++)
      {
            if(i<=trans_point) waveform_set_element(wave, i,
high_level); else  waveform_set_element(wave, i, low_level);
      }
}

double dig_setup_1()
{
      double cycle_time, strobe_time;
      lvm_error_mode( TRUE );

      tgmode( 2 );
      system_clock_source(t_mspc);
      ioh(IOH_IOL MA, cap_pin1);
      iol(IOH_IOL MA, cap_pin1);
      vz(1.0, cap_pin1);
      vol(threshold V);
      voh(threshold V);
      cycle(TSET1, 30.0 NS);
      cycle_time = cycle(TSET1, t_actual);
      strobe_time = cycle_time/2.0;
      edge_strobe(cap_pin1, TRUE);
      settime( TSET1, cap_pin1, DDR_STROBE, strobe_time, cycle_time);
      return cycle_time;
}

double waveform_estimate_freq(Waveform *wave)
{
      int i, old_state, new_state, num_edges = 0;
      double estimated_freq, num_cycles, sample_cycle;

      sample_cycle = waveform_get_x_increment(wave);
      old_state = (int)waveform_get_element(wave, 0);
      i=1;
      do
      {
            new_state = (int)waveform_get_element(wave, i);
            if(old_state != new_state)
            {
                  num_edges++;
                  old_state = new_state;
            }
            i++;
      }
      while (i < waveform_get_size(wave));
      num_cycles = num_edges/2.0; // measure number of cycles
      estimated_freq = num_cycles/(sample_cycle*samples);

      return estimated_freq;
}

double bullard_buneman_bin_estimator(Waveform *wave, double
expected_freq)
```

```
{
      int max_idx, min_idx, expected_low_bin, expected_high_bin;
      int left_bin;
      double min, max;
      int num_samples = waveform_get_size(wave);
      waveform_set_element(wave, 0, 0.0);
      if(expected_freq > 0.0)
      {
            expected_low_bin = (int)((expected_freq/
waveform_get_x_increment(wave))*0.8);
            if(expected_low_bin < 0) expected_low_bin = 0;
            expected_high_bin = (int)((expected_freq/
waveform_get_x_increment(wave))*1.2);
            if(expected_high_bin > num_samples) expected_high_bin =
num_samples;

            waveform_subset(subset_wave, wave, expected_low_bin,
(expected_high_bin-expected_low_bin));
            waveform_min_max(subset_wave, &min, &max, &min_idx,
&max_idx);
            min_idx = min_idx+expected_low_bin;
            max_idx = max_idx+expected_low_bin;
      }
      else waveform_min_max(wave, &min, &max, &min_idx, &max_idx);

      if (max_idx == 1)
            left_bin = 1;
      else if (max_idx == num_samples-1)
            left_bin = max_idx-1;
      else if (waveform_get_element(wave, max_idx-1) >=
waveform_get_element(wave, max_idx+1))
            left_bin = max_idx-1;
      else
            left_bin = max_idx;
      double left_mag = waveform_get_element(wave, left_bin);
      double right_mag = waveform_get_element(wave, left_bin+1);
      // avoid division by zero and overflows
      double est_bin;
      if (left_mag < 1e-100)
            est_bin = (double) left_bin+1;
      else if (right_mag < 1e-100)
            est_bin = (double) left_bin;
      else
      {
            double pi_over_n = NEXTEST_PI/(double) num_samples;
            est_bin = (double) left_bin+atan(sin(pi_over_n)/(cos
(pi_over_n)+left_mag/right_mag))/pi_over_n;
      }
      return est_bin;
}

TEST_BIN(pass_bin){}

TEST_BIN(fail_bin){ num_fails ++;}

SEQUENCE_TABLE( seq1 )
{
      SEQUENCE_TABLE_INIT
      TEST( clk_capture,            lpass,
      lfail );
      BINL( lfail,                  fail_bin,               STOP );
      BINL( lpass,                  pass_bin,               STOP );
}

SITE_BEGIN_BLOCK( site1_begin )
{
      InitCntr();
}

SITE_END_BLOCK( site1_begin ){}

static bool going_up = TRUE;

BEFORE_TESTING_BLOCK(bt_block)
{
      if(auto_set_dc)
      {

            remote_set(duty_cycle, auto_dc, 1, TRUE, INFINITE);

            if(auto_dc == 99) going_up = FALSE;
            if(auto_dc == 1) going_up = TRUE;
            if (going_up) auto_dc++; else auto_dc--;
      }
}

AFTER_TESTING_BLOCK(at_block)
{
      if (num_fails > 0) output("NUMBER OF FAILS = %d", num_fails);
}


TEST_BLOCK( clk_capture )
{
      double F1, FsubS, cycle1, M_real, duty_cycle1, temp_val,
high_time, low_time, estimated_freq, real_FsubS, real_FsubT;
      int N_real, N_new, M_whole;
      StartCntr();

      cycle1 = dig_setup_1();
      dci_clear();
      awi_connect( source1, ms_pin16);

      // setup DCI
      if (!dci_set(cap_pin1, samples, TRUE))
            warning(" Error with dci_set()");

      FsubS = FsubT*1000.0*100.0;

      if(use_dans_square)     dans_generate_squarewave(square_wave,
100, FsubS HZ, 0.0, 2.0 V, FsubT KHZ, (double)duty_cycle );
      else waveform_generate_square_wave( square_wave, 100, FsubS HZ,
0.0, 2.0 V, FsubT KHZ, (double)duty_cycle );

      awi_set( source1, square_wave, t_awi_no_filter, 0.0,
t_awi_continuous );
```

```
      awi_set_frequency(source1, FsubS HZ, 0.0, t_new_pll);
      real_FsubS = awi_get_frequency(source1);
      real_FsubT = real_FsubS/100.0;

      awi_start(source1);


      count(21, samples/2);
      funtest(cap_clk_double, fullec);


      if (!dci_get_waveform_parallel( cap_pin1, 0, samples,
captured_wave1))
                              warning(" Error with
dci_get_waveform_parallel()");

      waveform_set_x_scale(captured_wave1, 0.0, cycle1/2.0,
SCALE_SECONDS);
      if(get_estimated_freq)
            estimated_freq = waveform_estimate_freq(captured_wave1);
            else estimated_freq = -1.0;
            output("Estimated frequency is %fKHz", estimated_freq/1e3);

      waveform_real_fft(fft_wave, captured_wave1);
      waveform_magnitudes(mag_wave, fft_wave);

      M_real = bullard_buneman_bin_estimator(mag_wave, estimated_freq);
      F1 = (M_real*waveform_get_x_increment(mag_wave))/calibration_val;
      output("Frequency = %fKHz, cycles = %f", F1/(1 KHZ), M_real);
      // now for duty cycle

      M_whole = (int)M_real;
      N_real = samples;
      temp_val = (double)N_real*(double)M_whole;
      N_new = round(temp_val/(double)M_real); // calculates number of
samples to get a whole number of cycles.


      waveform_subset(subset, captured_wave1, 0, N_new);
      duty_cycle1 = waveform_arithmetic_mean(subset);
      output("Waveform Duty Cycle is %f%%", duty_cycle1*100);
      high_time = duty_cycle1*(1.0/F1);
      low_time = (1.0/F1)-high_time;
      output("high_time = %fns, low_time = %fns", high_time*1e9,
low_time*1e9);

      StopCntr();
            // statistical testing

            if(collect_stats)
            {
                  if(high_time > max_high_time) max_high_time =
high_time;
                  if(high_time < min_high_time) min_high_time =
high_time;
                  if(F1 > max_freq) max_freq = F1;
                  if(F1 < min_freq) min_freq = F1;
                  output(" max_high_time = %fns min_high_time = %fns,
max_dev = %fps", max_high_time*1e9, min_high_time*1e9, (max_high_time-
min_high_time)*1e12);
                  output(" max_freq = %fMHz     min_freq = %fMHz",
max_freq/1e6, min_freq/1e6);
            }
      output("elapsed time = %3.2fms", TotalTime*1e3);

      if(fabs((duty_cycle1*100)-duty_cycle) > 0.1) { output("duty cyle
wrong, returning FAIL"); return FAIL; }
      if(fabs(F1-real_FsubT)/real_FsubT > 0.001) {output("frequency
wrong, returning FAIL"); return FAIL; }
      return PASS;
}
```