

Contents

- 1 - Announcements
- 1 - Regression Testing in Analog Verification, Part 1

Greetings

Dear friends and colleagues,

What is different about today's mixed-signal designs is they are much larger than their predecessors, and they implement a large number of modes and settings. Both of these factors lead to expensive simulations. To control the simulation time, designers generally do two things.

1. They test the blocks individually.
2. They choose a representative set of modes and settings and test only those.

We have found that both act to limit one's ability to find functional errors in the design. The only way to have confidence that the design is completely functional before it is fabricated is to exhaustively test the design. Furthermore, it is not sufficient to just do it once. One must exhaustively test the design repeatedly during the design process. The early testing helps to find problems when they are still easily and inexpensively fixed. The late testing assures the design will actually function when fabricated. This repeated exhaustive testing is referred to as regression testing, and it is the subject of this newsletter.

This month, we give all of our attention to the feature article, "Regression Testing in Analog Verification, Part 1." Regression testing is the primary means by which bugs are found in analog verification. It is what makes analog verification truly verification.

As always, please let us know what you think about the article.

Sincerely,
Henry Chang and Ken Kundert

Announcements

Designer's Guide Community Opportunities

Analog/RF/Mixed-Signal engineers are currently highly in demand. We receive at least one call a day from recruiters asking for engineers. We have decided to start an opportunities page that lists jobs for analog/RF/mixed-signal professionals. If you are interested in looking for a job, our goal is to give you a great starting point to see the wide variety of jobs available. We are just starting, so we do not have too many listings yet. We do charge a fee for the posting. This helps us support the website, and also ensures that the people who post are serious about looking. If you are interested, please don't hesitate to contact them. Even if you don't send in a resume, I think many of the people who post would be interested in just talking to you.

If you have suggestions for how to improve our opportunities section of our site, please send us your comments.

For more information visit www.designers-guide.com/partners/dg-opportunities.html.

Regression Testing in Analog Verification, Part 1

By Henry Chang and Ken Kundert

Introduction

The most important aspect of analog verification is the writing of regression tests. It is through the regression tests that the entire function of the analog block is tested and verified. Recall that our initial goal in analog verification is functional verification. We are verifying that the outputs respond in a functionally correct manner as any of the input or control signals change (analog or digital). In this first of a series of articles on regression testing, we introduce the key concepts in regression testing by writing tests to verify the static behavior of a 10 bit digital-to-analog converter (DAC) at the transistor level. We apply the tests to the transistor implementation to clearly show how modeling and regression testing are two distinct concepts. Where a model of the block is required, we would run the same regression test on the model as on the transistor level implementation. If both pass, then we have confirmed that the model and the transistor level implementation are functionally equivalent. If no model is required, this regression test only needs to be applied to the transistor level implementation.

We begin with a relatively simple example. Our DAC has 10 inputs, 5 control bits, 1 bias input, 1 supply line, and 1 output. Though simple, over 2,500 automated checks will be performed to verify that it fully functions. In this article, we describe the circuit and its specifications, all of the components in the regression test, the results, and show what happens if there is an error in the circuit. As regression testing is a large topic, we dedicate a series of articles to the subject. In Part 2 of the series, we will add more complexity to the DAC such as adding a programmable RC filter at the output. We will extend the regression tests to go beyond static tests to test the filter settings. We will also add tests for power consumption.

Circuit Description

The block diagram of the DAC example is shown in Figure 1. There are three blocks. The bias block sets the current range of the DAC. This can be tuned in linear increments from 1 to 16 times the reference bias current. The DAC core converts the 2 most significant bits (MSB) in a linearly weighted fashion, and the 8 least significant bits (LSB) in a binary weighted fashion. The digital block has a 4 to 15 bit thermometer decoder for the bias reference, and a 2 to 3 bit thermometer decoder for the 2 MSBs of the DAC. It also inverts the enable line and buffers the 8 least significant bits of the DAC.

The pin descriptions for the DAC are shown in Table 1.

Regression Test

Listing 1 shows the testbench. This is the first part of the file that contains the regression tests. The testbench consists of parameter, signal, and variable declarations; the instantiation of the device under test (the DUT); and the analog connections to the DAC.

In this example, the entire testbench is in a single Verilog-AMS module. In this case, we are not using Verilog-AMS as a modeling language but as a verification language. As a superset of Verilog-HDL (digital) and Verilog-A (analog), Verilog-AMS gives the flexibility to sequence through



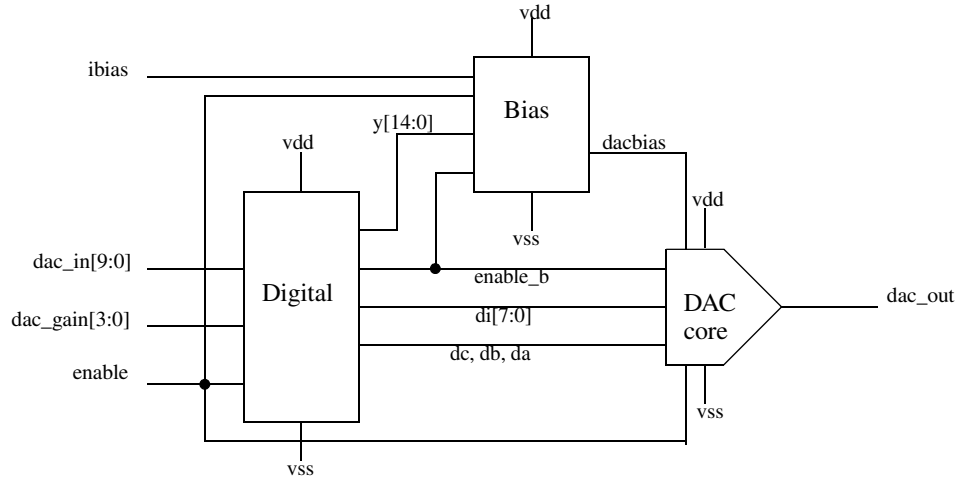


Figure 1: DAC block diagram.

Name	Pin	Direction / Type	Description
DAC input	dac_in[9:0]	input / logic	10 bit input to the DAC
gain control	dac_gain[3:0]	input / logic	Sets the DAC LSB current from 1× to 16× the reference current
DAC output	dac_out	output / electrical	Current output of the DAC. The functional behavior of the DAC is: $I(\text{dac_out}) = \text{dac_in}(\text{dac_gain}+1)\text{ibias}$
enable	enable	input / logic	enable=1 turns on the DAC and enable=0 turns off the DAC
bias current	ibias	input / electrical	The nominal bias current is 1μA.
supply/gnd	vdd, vss	input / electrical	Vdd = 1.8V, Vss= 0V

Table 1: DAC pin descriptions.

the digital codes using standard Verilog-HDL digital techniques, and observe the analog response of the DUT using Verilog-A techniques. This is a very powerful combination as will be shown when we walk through the code in detail.

The actual regression tests are contained in an *initial* block that Verilog runs as the simulation starts. The first few tests are shown in Listing 2. These tests make an initial assessment of the overall gain of the DAC. This value is used through out the tests. It then exercises each input code for the DAC and checks to see if the output is as expected.

We begin by setting the DAC into its default state. We then wait to allow the DAC to settle. Unlike in an ideal model, a real DAC will exhibit nonideal settling behavior. The delay is provided by the Verilog # operator, and the amount of delay is controlled by *gain_settle*, a parameter of the testbench that was set to 2 μs (this is defined at the beginning of the module).

Because we need to check the output of the DAC with an expected value, we first run a calibration test. In this example, a primary source of error is gain error. We test that DAC with *dac_in*=0, then with *dac_in*=1023. In both cases, before taking the measurement we wait an amount of time, *code_settle*, which is also a testbench parameter and is set to 100ns. We also calculate what the cur-



```

`include "disciplines.vams"
`timescale 1ns/1ps
`define tick 1e-9

module dac_testbench ( );

    // Testbench parameters
    parameter real supply_voltage = 1.8;           // supply voltage (V)
    parameter real bias_current_nominal = 1u;      // bias current (A)
    parameter real bias_r = 1G;                   // resistance of bias source (Ohms)
    parameter real load_r = 100;                  // load resistance (Ohms)
    parameter real load_c = 100p;                 // load capacitance (F)
    parameter real code_settle = 100n;           // settling time for code change (s)
    parameter real gain_settle = 2u;              // settling time for a gain change (s)
    parameter real abstol = 0.5u;                 // absolute tolerance (A)

    // Signal declarations
    electrical dac_out;
    reg [9:0] dac_in;
    reg [3:0] dac_gain;
    reg enable;
    electrical ibias;
    electrical vdd, vss;
    ground vss;
    branch (vdd, dac_out) load;
    branch (ibias, vss) bias;
    branch (vdd, vss) supply;

    // Variable declarations
    integer tests, failures, i, j;
    real bias_current, prev, tol, normalized_gain, expected, lsb, out_low, out_high;

    // Instantiation of the DAC
    dac dut (
        .dac_out (dac_out),
        .dac_in (dac_in),
        .dac_gain (dac_gain),
        .enable (enable),
        .ibias (ibias),
        .vdd (vdd),
        .vss (vss)
    );

    // Analog connections to the DAC
    analog begin
        V(supply) <+ supply_voltage;
        I(bias) <+ bias_current + V(bias)/bias_r;
        V(load) <+ V(load)/load_r + load_c*ddt(V(dac_out_c));
    end

    // Tests
    ...

```

Listing 1: The test bench.

rent corresponding to an LSB for the DAC given the DAC gain and bias current settings. With this data, we calculate the normalized gain. If the DAC were ideal, we would expect the normalized gain to be 1. As our first test, we record a failure if the normalized gain is not within 2% of the expected value. The idea is that a normalized gain sufficiently different from 1 indicates that there is likely something wrong with the circuit. All tolerances settings should be reviewed with the designer.

The actual check of the normalized gain is performed by *checkPercent*. This is a user-defined Verilog task shown in Listing 3. It determines whether the achieved value is close in a relative sense



```

...
// run the tests
initial begin
    // initialize testbench
    tests = 0;
    failures = 0;

    // initialize the DUT
    set_dac_default; // user defined task to set DAC inputs to default values
    #(gain_settle/tick); // Let the DAC fully settle

    // Determine the overall gain of the DAC
    dac_in = 0;
    #(code_settle/tick);
    out_low = I(load);
    dac_in = 1023;
    #(code_settle/tick);
    out_high = I(load);
    lsb = bias_current*(dac_gain+1);
    normalized_gain = (out_high-out_low)/(1023*lsb);
    checkPercent(1.0, normalized_gain, "DAC normalized gain", 2);

    // Test all of the DAC codes
    $display ("Test all DAC codes");
    tol = 4.0*lsb;
    for (i = 0; i < 1024; i= i+1) begin
        dac_in = i;
        #(code_settle/tick);
        checkAbs(i*(dac_gain+1)*bias_current*normalized_gain, I(load),
            "DAC codes (abs value)", tol);
        if (i > 0) begin
            checkMonotonicity (prev, I(load), 1,
                "DAC codes (monotonicity)", 0.95*lsb);
        end
        prev = I(load);
    end
end
...

```

Listing 2: The initial set of tests.

```

task checkPercent;
    input expected, actual;
    input description;
    input tolerance;
    real expected, actual;
    reg [30*8:0] description;
    real tolerance;
    real error;
    integer failed;

    begin
        error = abs((actual-expected)/expected)*100;
        failed = (error > tolerance) && (abs(actual-expected) > abstol);
        $display("%s @ %0.2fus: %0s: expected=%0.2g actual=%0.2g error=%0.2f%%.",
            (failed ? "FAIL" : "Pass"), $abstime/1u,
            description, expected, actual, error);
        ranTest(failed);
    end
endtask

```

Listing 3: The task that determines if an output is close in a relative sense to the expected value.



to the expected value. The fact that a test was run, and whether the test passed or failed, is recorded by yet another task, *ranTest*, shown in Listing 4. This task sets variables that are declared in the top-level module, *dac_testbench*.

```

task ranTest;
    input failed;

    begin
        if (failed)
            failures = failures + 1;
            tests = tests + 1;
        end
    endtask

```

Listing 4: A task for recording tests and whether they passed or failed.

We now run through all 1,024 codes of the DAC. We set the input to each value, and again wait for the DAC to settle. We then measure the output current and see if it is within a certain tolerance of the expected value, which has been adjusted by the gain error we had previously calculated. In this case, we set the tolerance to 4 LSBs. Unfortunately, even with the gain error adjustment, we find that if we set the tolerance to tighter than 4 LSBs, we get spurious errors due to the deterministic integral nonlinearity (INL) of the DAC. Recall again that we are testing function. We are checking to see that as the input changes, the output changes in a functionally correct manner. Because our tolerance is greater than an LSB, we need to apply a second test that has better resolution. Without such a test it is possible that if there is an error in the least significant bit, such as it being tied to supply or ground, we would miss it. Here, we choose to apply a monotonicity test. We check that as we change the DAC codes from one to the next that the change is at least 75% of an LSB. This value should be adjusted to accommodate the deterministic differential nonlinearity (DNL) of the DAC under test. With the test on the gain error, the absolute value test on each of the codes, and this monotonicity test, we can be certain that the DAC is functional if it passes all of the tests. These tests use *checkAbs* and *checkMonotonicity*. These are also user-defined Verilog tasks that are similar to *checkPercent* and are not included for the sake of brevity.

There are a few important things to note about these tests. First, the test code is executed as the simulation is running. We are not post processing the results, but detecting errors as we run the simulation. In fact, no signals need be saved while running our tests. The main reason to save signals is for debugging purposes if an error is found. Second, the messages are printed to the simulation output. Thus, without looking at a waveform, errors in the design can be quickly seen as the simulation is running.

These tests hint at the potential of the dynamic nature of regression tests written in Verilog-AMS. The 1,024 input tests are parameterized in terms of the measured gain. Although in this case, these tests could have been conducted in a post processing fashion, we have written tests where post processing techniques cannot be used. For example, we often adjust settling times on-the-fly. An advantage of this self calibration method is that this test can now be run on an ideal model view and this transistor level view and have both pass. Had we hard coded the gain error, an ideal functional model view would not pass.

The tests shown in Listing 2 are a subset of the tests needed to fully verify the function of the DAC. In particular, more tests are needed to test the gain settings for the DAC and the enable line. These are not included as they are similar to what has already been shown.



Results

The full suite of regression tests includes over 2500 tests. When all the tests pass, the following summary statement is printed by the test bench,

```
VERIFICATION COMPLETE: 0 failures found in 2543 tests.
```

In this case the circuit is relatively small and very simple transistor models are used, so the tests run very quickly (less than a minute). A more typical design with foundry models will of course take much longer. However, it is our experience that one can generally arrange to complete the verification of even complex RF designs overnight.

Now to illustrate the power of this regression test, we introduce a few common errors in the transistor level design. The first error we introduce is an inverted enable signal. The output for this case is shown below. The initial gain error test finds this error. We omit the many other errors detected subsequently. There were 1,102 errors in total. Each has a "FAIL" message printed in the simulation log.

```
FAIL @ 2.2us: DAC normalized gain: expected=1.00 actual=3.81e-06 error=100.00%.  
...  
VERIFICATION COMPLETE: 1102 failures found in 2543 tests.
```

A more subtle error is if the `dac_gain[2]` and `dac_gain[1]` bits are switched. This error is caught as the gain setting bits are tested.

```
FAIL @ 116us: Gain (binary array): expected=3.07uA actual=5.12uA error=66.74%.  
FAIL @ 116us: Gain (binary array): expected=6.14uA actual=10.3uA error=68.30%.  
FAIL @ 116us: Gain (binary array): expected=3.07uA actual=5.16uA error=68.16%.  
FAIL @ 116us: Gain (binary array): expected=12.3uA actual=20.7uA error=68.26%.  
...  
VERIFICATION COMPLETE: 186 failures found in 2543 tests.
```

The output when `dac_in[1]` and `dac_in[0]` are swapped is shown next. Here as described earlier, the monotonicity test catches this error.

```
FAIL @ 2.5us: DAC codes (monotonicity): previous=20.56uA current=10.26uA diff=-10.3uA.  
FAIL @ 2.9us: DAC codes (monotonicity): previous=61.64uA current=51.34uA diff=-10.3uA.  
FAIL @ 3.3us: DAC codes (monotonicity): previous=102.7uA current=92.43uA diff=-10.3uA.  
FAIL @ 3.7us: DAC codes (monotonicity): previous=143.8uA current=133.5uA diff=-10.3uA.  
...  
VERIFICATION COMPLETE: 393 failures found in 2543 tests.
```

Suppose in the digital logic, we used an OR gate instead of an AND gate. Here again, the gain error test finds this immediately.

```
FAIL @ 2.2us: DAC normalized gain: expected=1.00 actual=0.92 error=8.05%.  
FAIL @ 2.4us: DAC codes (monotonicity): previous=0.04uA current=9.26uA diff=9.22uA.  
FAIL @ 2.5us: DAC codes (monotonicity): previous=9.26uA current=18.5uA diff=9.25uA.  
FAIL @ 2.6us: DAC codes (monotonicity): previous=18.5uA current=27.8uA diff=9.26uA.  
...  
VERIFICATION COMPLETE: 280 failures found in 2543 tests.
```

Finally, supposed that one of the bias input bits is tied to ground by mistake. Here, there are fewer failures.

```
FAIL @ 106.7us: Gain (binary array): expected=1.02uA actual=2.04uA error=99.58%.  
FAIL @ 106.8us: Gain (binary array): expected=2.05uA actual=4.15uA error=102.98%.  
FAIL @ 106.9us: Gain (binary array): expected=1.02uA actual=2.08uA error=103.18%.  
FAIL @ 107.0us: Gain (binary array): expected=4.09uA actual=8.30uA error=102.91%.  
...  
VERIFICATION COMPLETE: 24 failures found in 2543 tests.
```



As illustrated in this example, sometimes there are a spectacular number of failures. Sometimes there are few. Care must be taken when writing the regression tests to think through all that might go wrong and test for those cases. We find that once the regression test module is set up, adding additional tests is a relatively straightforward process requiring usually only a few lines of code.

In several of the examples, the first test, the gain error test caught the error. Since these errors are printed as the simulation is running, within a small fraction of the time for the entire simulation, the error would be known to the verification engineer. As is more typical when the regression test requires overnight to run, this method has a great advantage over post processing approaches in that debugging can begin immediately rather than waiting for the simulation to complete.

Finally, to give an idea of the tests that are run, we show the output waveform of all of the tests applied to the working circuit in Figure 2 and Figure 3. Without a regression test based methodology, a designer would need to visually inspect this waveform to validate that the circuit is working. Given that checking 2,500 points on every design iteration is not realistic, it is likely that designers would not apply such a comprehensive test. This is where errors creep in.

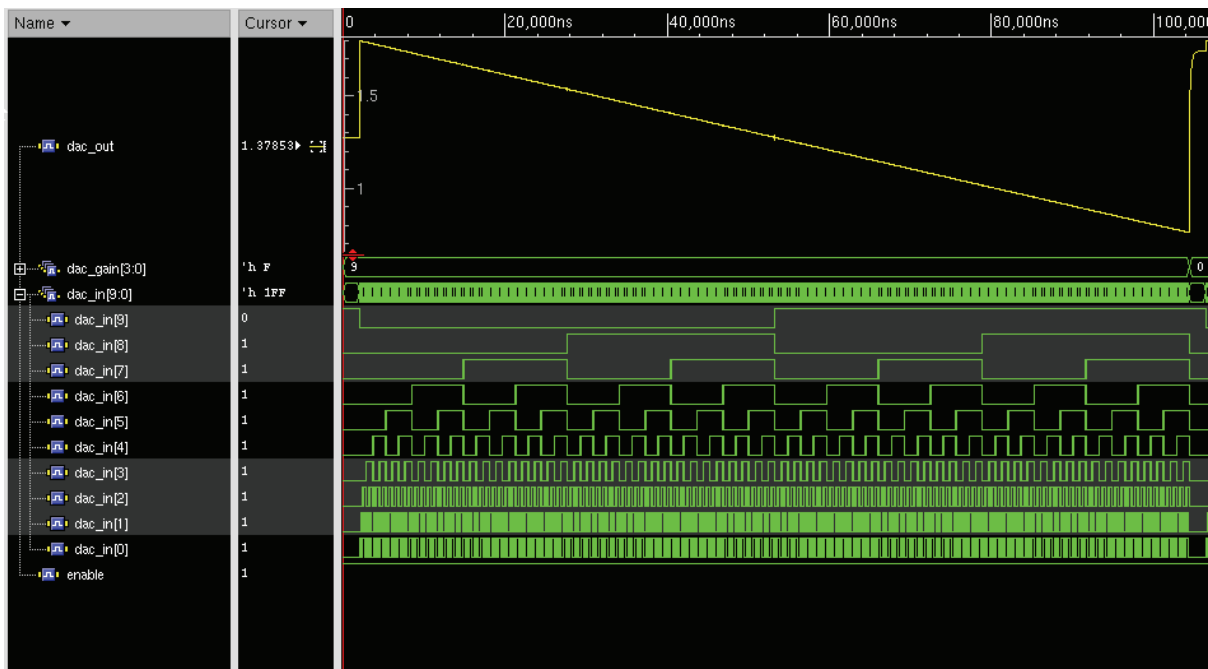


Figure 2: Output waveform for DAC input code tests.

Conclusion

Running the regression tests on the transistor level implementation is where analog verification comes together. For the analog verification engineer, this is a powerful tool for finding bugs in the circuit design. In future articles, we will build on this example to show more advanced techniques that can be applied to test circuits using this methodology.

You can access a fully complete and executable version of this circuit and testbench at www.designers-guide.com/newsletter/0711/example.tgz.



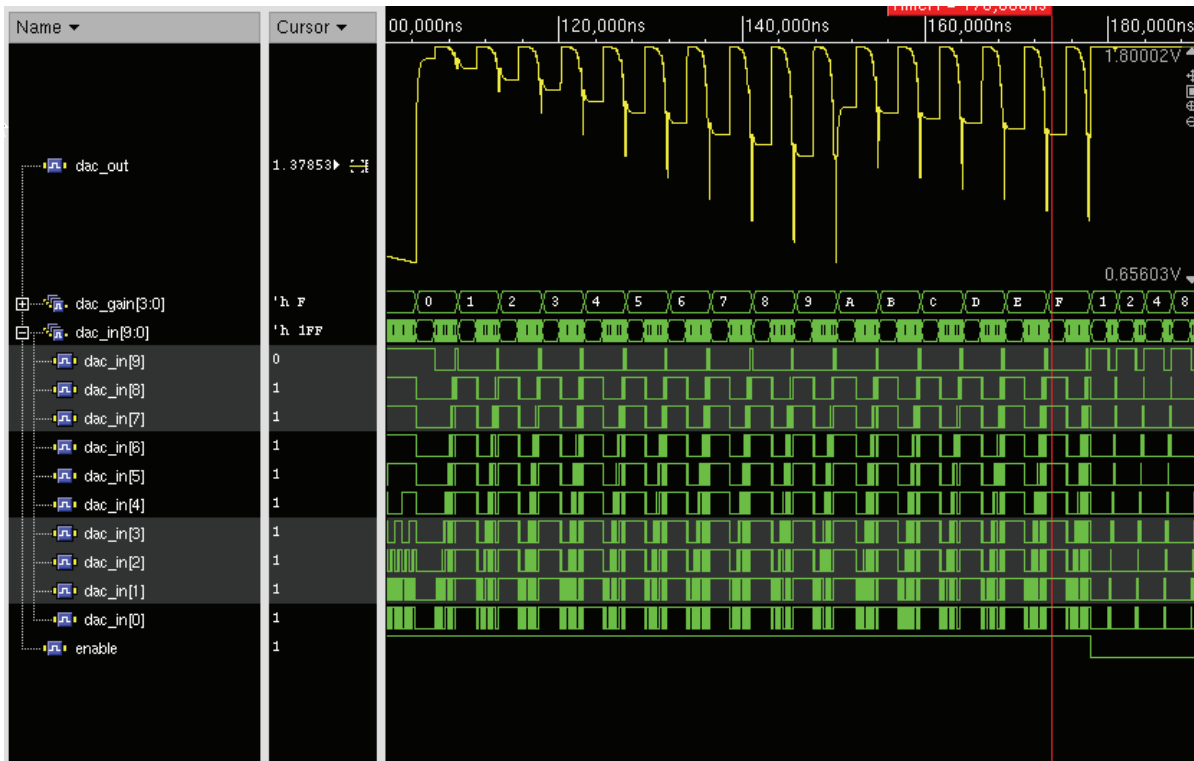


Figure 3: Output waveform for DAC gain and enable tests.

Disclaimer: We strive to provide information that is both helpful and accurate. Designer's Guide Consulting, Inc., the creators of this newsletter, makes no representation or guarantees on the newsletter contents and assume no liability in connection with the information contained on it.

Copyright, 2007 © Designer's Guide Consulting, Inc. No reproductions of this newsletter can be made without the expressed permission of Designer's Guide Consulting, Inc.

