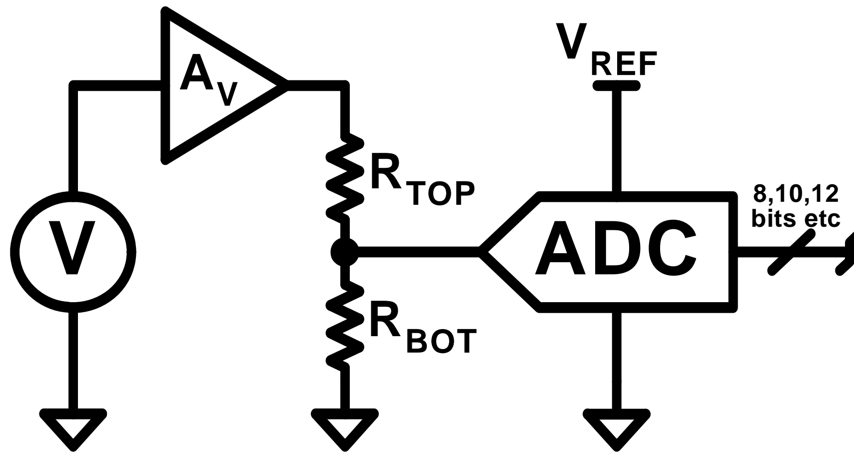
ADC SCALER METHOD

The purpose of this scaler calculation is to eliminate long division in the conversion of raw ADC data into a human readable decimal value using only integer math. This scale factor will account for the voltage gain Av, the voltage division of the Rtop-Rbot divider network, the ADC Vref , the ADC resolution and the sample count when averaging is used. A computation result equal to Volts x 1000 (in the examples used below) is computed with a minimal loss of accuracy due to rounding errors while using just a single multiplication operation followed by a fast binary byte shift division. By adjusting this scaler to reflect the actual system, all of the various error sources can be calibrated out at once. This is best used for circuits with linear transfer functions; non-linear systems such as thermistor curves are better converted using multi-point linear interpolation techniques.



The main object of this exercise is to multiply up information to the right of the decimal point, moving as many significant digits to the left of the decimal point as possible at the beginning, saving any division to be done for the very last step. To begin with the ratio below shows generically how an ADC reading is converted to a "Voltage".

$$\frac{ADC_{COUNTS}}{ADC_{FULLSCALE}} = \frac{VOLTS_{IN}}{VOLTS_{FULLSCALE}}$$

Or re-written...

$$VOLTS_{IN} = \frac{ADC_{COUNTS} \times VOLTS_{FULLSCALE}}{ADC_{FULLSCALE}}$$

For the following examples, a 10 bit ADC with an input at half its full scale limit would compute out to a value of 2.4975V. Note that there is no way to get a result of 2.5000V because there is no integer that is exactly half of 1023. Adding more bits will only make the distance from exactly half progressively smaller.

$$VOLTS_{IN} = \frac{511 \times 5.000}{1023} = 2.4975$$

This calculation is fine as is when done on a PC with unlimited processing power and memory space to do floating point math on every calculation, but in the limited resources of an embedded processor you can simply move the decimal to the left to prevent the need for floating point math using the much faster and compact integer math that can be done in hardware, all except of course the division.

$$\mathbf{VOLTS_{IN} = \frac{511 \times 5000}{1023} = 2497(.5)}$$

Note that the decimal in the result is displayed parenthetically: Integer math will essentially round every result down to the lower integer value, discarding anything to the right of the decimal point. When displaying this result, the first digit is displayed, then a decimal point and then the next three digits! By doing the math in the order shown above we are still doing long, non-binary division which tends to eat a lot of instruction cycles and a lot of RAM so a short-cut is to create a "Scale factor" or "Scaler" by doing the division ahead of time. We can then simply multiply the ADC reading by the scaler.

$$\mathbf{Scaler = \frac{5000}{1023} = 4.8875855;\ VOLTS_{IN} = 511 \times 4.8875855 = 2497(.6)}$$

Of course, now we have a decimal in the math again and this is not good for an embedded CPU! This is where binary math comes to the rescue... If you want to divide a number by 2, you can simply shift all of the bits to the right by 1 space where the least significant bit drops into oblivion. Likewise, division by any power of 2 is done by shifting to the right by that power of 2; n/2=1 shift, n/4 = 2 shifts, n/8=3 shifts and so on.

546 / 2 = 1000100010 >> 1 = 0100010001 = 273

When shifts are done 8 and 16 at a time as when dividing by 256 and 65536, you can simply discard the lower bytes and get your result without shifting. In an 8 bit RAM processor, this works out really well as all data is already stored as groups of 8 bits.

4532 / 256 = 00010001 ~~00000000~~ = 00010001 = 17

1114112 / 65536 = 00010001 ~~00000000 00000000~~ = 00010001 = 17

How does this help us? If we take the decimal scaler of **4.8875855** from the previous example and multiply it by any power of 2 ^ (n x 8) bits such as 256, 65536 or 16777216, we can do some simple byte shift division where the accuracy gets progressively better as n is increased since we are throwing away fewer and fewer digits of the scaler constant after the decimal point. From the previous example of an ADC reading of 511 and a multiplier of 2 ^ (2 x 8) and using hex to show the byte-wise division operation:

Scaler = 4.8875855 x 65536 = 320312.73 (use 320313)
$\mathbf{VOLTS_{IN}}$ = (511 x 320313) / 65536 = 0x09C1~~8EC7~~ = 0x09C1 = 2497

This is a good time to note the one major limitation of this method: If for instance 32 bit math is used to convert a number down to a 16 bit value with a 16 bit divider, the multiplication result MUST NOT exceed 32 bits! Just like with shifts to the right, overflows to the left go into oblivion!

This may seem like a lot of work to end up with the same result, but on a RISC processor, long division can consume hundreds of microseconds and this can add up when a lot of different inputs are converted continuously. Another benefit of this method is that the scaler can be adjusted to simultaneously perform the averaging of an accumulated number of readings in addition to scaling the result. Even when the number of samples is an even power of 2 such as 16, just the shift dividing to get the average of the raw ADC data in a 16 bit variable using an 8 bit processor requires about 40 instructions... by combining the averaging and the scaler by this method, the whole operation can be done in about 8 instructions!

Scaler = 4.8875855 x 65536 / 11 = 29119.3 (use 29119)
Average = (511 x 5) + (512 x 6) = 5621
$VOLTS_{IN}$ = (5627 x 29119) / 65536 = 0x09C4~~3145~~ = 0x09C4 = 2500

The accumulation of samples for the purpose of averaging is a very fast process as it simply requires addition of each successive sample, something any micro-processor is really good at. As can be seen in the above example, another advantage of averaging is that averaged ADC readings can have more resolution than the ADC is actually capable. From the example above for an input of exactly 1/2 the full scale range, any single reading can only consist of 511 or 512 counts; never the 511.5 counts of exactly 1/2 full scale (One half of 1023 for the 10 bit example). However, by averaging a signal that is right at the boundary of counts 511 and 512 where some number of samples read 511 and some number reads 512, we can actually achieve a result equal to the elusive 2.500V for a virtual resolution of 11, 12 and more bits for a 10 bit ADC!

When writing the firmware for ADC conversions the optimal scaler computed from the ideal circuit elements can be used but what do you do when the real world errors that exist at every stage of a circuit are considered? This question brings us to the final level of refinement for this method because reference voltages are never EXACTLY what they are supposed to be and resistor dividers are never operate EXACTLY as the values printed on the parts would suggest; everything must allow for some level of error or "Tolerance" in order to be manufacturable. With a scaler, the procedure to calibrate out these errors is quite simple: First you use the optimal scaler as a default and apply a known voltage to the input to be calibrated. Next, you take the value you wanted and divide it by the value you got using the optimal scaler and multiply the optimal scaler by the result. This becomes the calibrated scaler that can then be saved and the firmware can be written such that if a calibrated scaler is available then use it, otherwise use the optimal scaler that has been hard coded as a default.

As an example of a calibration step, a scale factor of 40039 was computed to give an optimal reading of 5.000V for a 5.000V input, however because of the various sources of errors, the ADC read function returned 4.879V when the input was actually 5.012V: 41131 is the calibrated scaler where 40039 would be the uncalibrated hard coded default scaler.

$$\frac{5.012V}{4.879V} \times 40039 = 41131$$

The equations used in the scaler calculator are shown below. VOUTfullscale is the maximum

input into the network that will result in the input to the ADC being equal to the ADC reference voltage, i.e. to output its maximum code where a resistor divider has a gain <= 1, an amplifier will have a gain >= 1. Keep in mind that this could apply just to "Amps" full scale so that for instance if you are measuring current with a 2 ohm resistor, the gain would be 2 where a 0.1 ohm resistor would have a gain of 1/10.

$$\mathbf{VOLTS_{FULLSCALE}} = \frac{\left(\frac{V_{REF}}{R_{BOT}}\right) \times (R_{BOT} + R_{TOP})}{\mathbf{Gain}}$$

$$\mathbf{Scaler} = \left(\frac{\mathbf{VOLTS_{FULLSCALE}}}{\mathbf{ADC_{FULLSCALE}}}\right) \times \left(\frac{65536}{\mathbf{Samples}}\right) \times 1000$$

When making the math fit into a 32 bit unsigned integer, the two numbers that will have the greatest effect are the PostDivider (65536) and the ResultMultiplier (1000). Remember that the ResultMultiplier is the distance to shift the decimal point to the right and so should match the resolution of the ADC: 6 decimal places for an 8 bit ADC result does not have a lot of meaning and 3 decimal places for a 16 bit result throws away useful information. The ResultMultiplier should therefore be established first based on maintaining the maximum information from your ADC and then the PostDivider should be as large as possible such that the scaler multiplied by the raw ADC data still fits into the available 32 bit integer.

Below is sample code from my switcher efficiency sweeper that reads 16 samples from 4 inputs and then scales them to volts or amps using calibrated scalers. The post-calibrated accuracy achieved by using the scalers is better than 0.2% for all channels.

```
///////////////////////////////////////////////////
//global variables: calibration constants
//Vref is 3.05V
//Vin and Vout use a 75K/30.1K divider
//Iin has a gain of 5 and a 20K/30.1K divider
//Iout has a gain of 2 and no divider (Rtop=0)
unsigned    int16   r_vin_cal=42835;  //Opt 42640
unsigned    int16   r_iin_cal=4882;   //     4065
unsigned    int16   r_vout_cal=42946; //    42640
unsigned    int16   r_iout_cal=6106;  //     6106

//global variables:  function return variables
unsigned    int16   in_volts;
unsigned    int16   in_amps;
unsigned    int16   out_volts;
unsigned    int16   out_amps;
///////////////////////////////////////////////////
//function to read ADCs and compute results
void read_all(void){
unsigned int8 i;           //loop counter
unsigned int32 temp32;     //32 bit math variable

//initialize averages to zero (Block averages)
```

```
in_volts=0;
in_amps=0;
out_volts=0;
out_amps=0;

//interleave read 16 samples from each input
for (i=16;i>0;i--)
    {
    set_adc_channel(0);    //set the ADC channel
    delay_us(15);          //mux settling time
    in_volts+=read_adc();  //add to the average

    set_adc_channel(1);
    delay_us(15);
    in_amps+=read_adc();

    set_adc_channel(2);
    delay_us(15);
    out_volts+=read_adc();

    set_adc_channel(4);
    delay_us(15);
    out_amps+=read_adc();
    }

//each average (16bits) is multiplied by its
//calibrated scaler (16bits) and then the 32bit
//result is byte shift divided to a 16 bit
//scaled result.

temp32=(int32) in_volts * (int32) r_vin_cal;
in_volts=make16(make8(temp32,3),make8(temp32,2));

temp32=(int32) in_amps * (int32) r_iin_cal;
in_amps=make16(make8(temp32,3),make8(temp32,2));

temp32=(int32) out_volts * (int32) r_vout_cal;
out_volts=make16(make8(temp32,3),make8(temp32,2));

temp32=(int32) out_amps * (int32) r_iout_cal;
out_amps=make16(make8(temp32,3),make8(temp32,2));
}
///////////////////////////////////////////////////
```