Using the NT7S Library With the Si5351 Synthesizers Versions, Continuous Fractions and the Arduino IoT33

05/17/24 1st Draft

These notes have arisen from some recent efforts to make improvements in an original disciplined reference for use in broadband frequency converters for ham radio use. This unit was a three-output synthesizer disciplined either by a GNSS satellite timing pulse or an external 10 MHz standard such as an OCXO or rubidium signal sported by an Arduino controller and on-board phaselock loop.

For many users of the Si5351 none of this may be relevant, existing performance of the A series part, perhaps from a breakout board and used only for wide modulation systems on HF may be entirely satisfactory as things stand. For those people this might be a good time to stop reading and go do something more useful.

I desired better close-to-carrier phase noise, more outputs and better frequency precision than the original unit which used a Si5351 A series part. The C series version in a 20 pin package offered 5 additional outputs and potentially lower phase noise thorough use of af an external reference input.

Previously I had used the ADAFruit Si5351 library which sufficed for the simpler A version part but did not support the C part as well as Jason Meldrum’s NT7S library appeared to do. Most of Jason’s work appears to have been aimed at somewhat less precise ham radio applications than mine and had not been extensively developed on the C version nor at all on Arudino boards other than the UNO.

In adapting and extending this library I ran into several difficulties which may apply to some users with more demanding applications such as mine. As of the time of this writing there are several issues that I think might be worth documenting. They are

- Support for Arduino IoT33 with the SAMD21 processor
- Higher precision for the available outputs using continued fractions algorithm
- Lower phase noise for the outputs through use of an off-chip VCXO
- Library problems when using the IoT33 board environment

I’ll address these in separate sections. Here is a description of my goals.

Version Differences

The Si5351 is not a single part but rather a family of parts with different strengths and capabilities. As can be found in Skyworks Si5351-B.pdf data sheet and expanded upon in AN619.pdf the family comes in three package types, 10-MSOP, 16-QFN and 20-QFN, with only the first being hand-solderable and the 16 and 20 pin versions requiring reflow assemby.

In these packages there are also three versions, A, B and C,

Si5351A available in all three package types

Si53551B in 16-QFN and 20-QFN

Si5351C available in 16-QFN and 20-QFN

Furthermore, each version has particular capabilities.

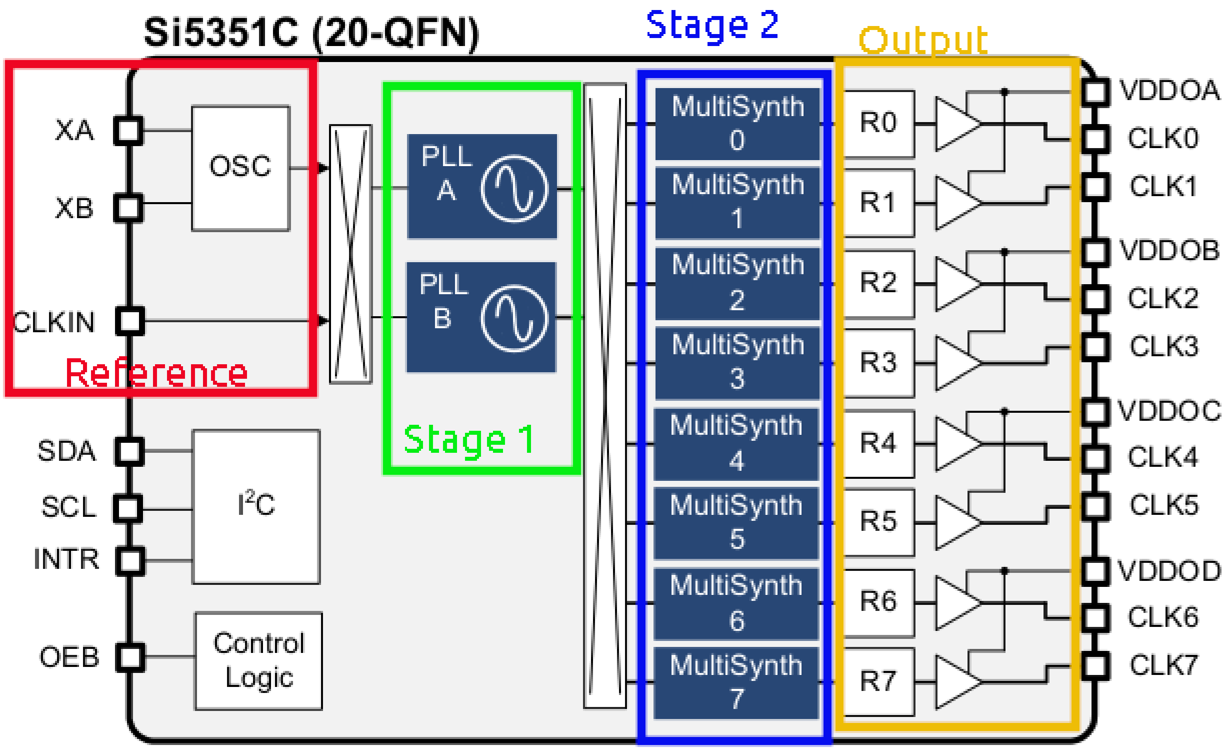
Si5351A 10-MSOP has three outputs and an external Crystal using a special on-board oscillator, though it can also be used with an external clock as well. The 16-QFN version has four outputs while the 20-QFN package haseight.

The Si5351B in 16-QFN is like the corresponding 16 pin A version but replaces one of its two internal PLLs with a VCXO which can be used to create a PLL from an external reference

The Si5351C packages both offer a separate external clock input, not using the on-board oscillator circuitry. The 16-QFN has four outputs while the 20-QFN has eight.

Thus no single version and package has all the features of the aggregate. However all have at least one PLL stage which generates a programmable output in the 600-900 MHz range using a ‘multisynch’ design which allows for either integer of fractional N relationship with an input reference, crystal, external or VCXO as described above. There are two of these PLL’s in the A and C version but only one in the B.   The PLLs apparently are actually operating at 2.4-3.6 GHz internally but this is of no real consequence to the user or programmer.

The PLL stage, the first stage can then be applied to a second stage that is used in generating outputs. This stage also uses the multisynch integer/Frac-N capability and in combination with an additional output divider, with some restrictions, is capable of producing at least one output from 2.5 kHz to over 200 MHz. The constraints are not simple and make application and library design more difficult.



The B version has a VCXO with external tune input replacing PLLB and other variation such as I2C address selection is available on the 16 and 20 pin A versions but this block gives a general idea of family.

A block diagram of the 20-QFN C version is shown above. This doesn't completely represent all the versions but it shows most of the features. There are four sections,

1) **Reference** input, XO, external or VCXO (along with an available divider that is not shown)
2) **Stage 1** Int or Frac-N divider PLL or VCXO
3) **Stage** 2 Int or Frac-N divider
4) **Outputs** with integer dividers

where each section can be routed by way of a cross-point switch to the next and in the case of the outputs directly to the the reference input section as well.

As can be appreciated, this multiplicity of packages, versions and capability makes understanding and applying the part very complex. This complexity is reflected in any library that attempts to make good use of the full capability of the family almost impossible.   Not only is management of part difficult but good understanding of the trade-offs and characteristics and advantages of the various features make choosing a good design to match a requirement even more difficult.

User Requirements

For my own purposes with a second revision disciplined reference for amateur use and for more precsision signal generation for propagation and ionospheric measurement systems using narrowband modulation techniques such as WSPR and FST4W I had these particular empahses:

1. Frequency precision
2. Multiple outputs
3. from external reference input instead of XO pin
4. low to moderate cost and convenient SMD assembly

The 20-QFN C version along with Jason's library seemed good to consider to meet these goals since it provided somewhat more support for eight outputs, external reference input and frequency management for the outputs of this complex part. However in trying to apply it to my existing environment, an Arduino Iot33 board and external VCXO I ran into several difficulties.

Frequency Precsion – Continued Fractions

The NT7S library can program the PLL and output stages with either integer or Frac-N ratios however the precision of the Frac-N, that is, it's resolution is limited by the precision of the floating point library used to request a non-integer result as well as by the algorithm.

Output/input relationship is accomplished within the multisynch registers by either multiplying or dividing the input by (a + b/c) where generally 6 <= a <= 90 and b/c < 2^{20} =1,048,576. In the library a a fixed c = 1e6 is used. For the PLL stage which has an output that is a multiple of the [25 MHz] reference, this limits precision to no better than 1e6/6 or about .17 ppm or 170 ppb (parts-per-billion). While this may be more than adequate for many amateur applications being only one sixth of a ppm, less than .00002% or 5 Hz for, say, a 10m SSB signal, it is nowhere nearly good enough for a narrowband WSPR or FST4W signal being used as part of a measurement of the ionosphere at HF measured in the small mHz (not this is a lower-case 'm'' meaning thousandths of a Hertz not a 'M' meaning MegaHertz). Even if the signal is needed to reference, say, a 10 GHz SSB or CW system it would be 1.6 kHz in error which is unacceptable.

A second source of imprecision is the machine epsilon of the standard Arduino UNO floating point library. While both 'float' and 'double' precsion are valid types, in the Arduino they are the <u>same</u>. In the Uno they are both single precision with a machine epsilon, the smallest value different from 1 that can be detected, of about 60 ppb.

So together the two problems, a fixed denomoinator of 'c' and a single position floating point library in a standard Arduino UNO limit applications which can tolerate 230 pbb setting resolution.

The solution I used to improve this precision was two-fold, use double precsion floating point precision in the Arduino and add an algorithm called "continued fractions" (CF) to the frequency setting algorithm. This turned out to be rather easy for two reasons, one the IoT33 with SAMD21 CPU already had double precision for its floating point library. This provided a machine epsilon of almost 10^{-16} a value so small that it exceeds even the precision of the NIST definition of time/frequency. I was able to request a setting so precise that I had no way of telling that the result wasn't perfect.

The second part of this solution was the use of continued fractions which to my great delight was already coded by Franco Venturi, KV4Z, from an algorithm described in Wikipedia. I am greatly appreciative of Franco's contribution, it made a drop-in improvement to the NT7S library ridiculously simple. This code was used in place of the calculation of the multisynch dividers for the b and c values of the Frac-N synthesis and result in errors of about the same magnitude as the double precision floating precision of the IoT33. See Franco's Github site for the Arduino code. Franco's rational_approximation() method is shown at the end of this note.

This improvement in precision absolutely requires double precision floating accuracy in the language. As noted, a stock Uno library does not provide this but a 64 bit float library is available. If you are considering using the high-precision algorithm it is pointless to try it without also using a float library that can *request* a result to the precision you require. Don't bother otherwise.

Multiple outputs

While all the Si5351 versions and packages offer multiple outputs, at least three, these are not independent and complex relationships are involved. With no more than two shared PLLs, at least one of the outputs is constrained by another output's selection of PLL frequency. But with the use of continued fraction this is problem is greatly mitigated since virtually any desired output frequency may be obtained from any PLL. This is a great advantage of the CF solution since it essentially provides complete output independence. Although the si5351 documentation encourages using integer divides, in practice you may be hard pressed to find an issue using Frac-N everywhere, at least for amateur HF uses.

Jason’s library already manages much of the other dependent factors, particularly for clock outputs 6 and 7 as well as managing power setting.

Lower Phase noise

Another reason for choosing the C version Si5351 was the presence of an external reference input which did not limit me to the use of either a crystal or one of the crystal oscillator pins (Xa) as an external input. Though not proven yet, this matters because of increased phase noise in the outputs.

Before I detail this, I want to say while I write in terms of phase noise rather than ‘jitter’ or ‘Alan Variance’ when describing spectral purity.

The ultimate goal of a precision reference is to produce a perfectly regular sinusoid at the output having no modulation of any kind imposed on it. This is impossible. Every signal has both some angular and some amplitude modulation imposed upon it. An attempt to summarize this complex condition is made by using simple models either in time or frequency domain. The method chosen is influenced by the ultimate application. The resulting signal can be described in terms of frequency or time. The spectrum that results from these unwanted modulation has finite bandwidth in the frequency domain and has angle, or phase, that varies in the time domain. While amateurs may tend to think of frequency accuracy I find that thinking in terms of phase is more useful for most of our applications.

Frequency is the rate-of-change, the time derivative, of phase. A signal that is offset in time from an ideal can have no frequency error but only a phase offset, if that offset is stable. For some applications that offset of time/phase matters greatly even if it doesn’t in others. Additionally the spectrum produced by angular modulation imperfection, referred to as “phase noise” may not imply any frequency error at all but be extremely detrimental to amateur communications. These communications always require non-zero bandwidth and finite signal-to-noise-ratio (SNR).

Typically the use of jitter or Alan Variance to describe an output isn’t as appicable to amateur communications systems. Jitter is often stated in terms of imperfections at higher rates – perhaps from 12 kHz to 20 MHz offset from an ideal carrier- while Alan Variance is especially useful for describing the performance of a signal over very long intervals and related to energy very close to a carrier.

On the other hand, phase-noise provides a representation of unwanted noise energy at any offset from the carrier and provides insight into the SNR consequences of imperfect signals to communications.

For wider bandwidth amateur modes such as SSB or CW, noise power within the information bandwidth, perhaps 50 to 3000 Hertz is most relevant since these impact commumications the most. For narrowerband signals such as the newere digital modes, noise from perhaps less than .01 Hz to a few Hz is most influential. Phase noise representations easily and directly allows examination, comparison and optimization of a system in these regions of interest.

As a practical example, it is possible to generate a signal with a relative large 50 or 60 Hz angular component, one that would be unacceptable for many amateur uses but which might show near zero jitter and not obviously bad Alan Variance. A phase noise plot would clearly show a large spike at this offset from the carrier and immediately identify an important issue.

The Si5351 is a complex part. This is evident by the crystal or ceramic resonator support in its on-board oscillator circuit. To appreciate how important these circuits are it is worth considering what happens if there is only a small amount of unwanted modulation, either noise or coherent signal from limited power supply rejection of the chip. A signal that is, say, 60 dB below the carrier generated by a 25 MHz reference is only -20 dBC at 2.5 GHz. Silicon Labs/Skyworks has designed the Si5351 to essentially eliminate power supply related modulation of this oscillator circuit while still providing one that can operate with a wide variety of external crystal resonator components. Even inexpensive and relatively poor quality crystals can generate spectrally decent outputs because of this. The circuit has provision to adjust oscillation conditions to avoid over-driving crystals while providing suitable SNR.

However the generality of the circuit comes at a cost. The phase noise floor of this on-board circuit is not particualy excellent. The Si5351 C version provides an ability to source a reference in a manner not requiring the use of this circuitry and input pin by providing a separate external clock input. Though not thorougly measured yet, use of this input has the potential to produce significantly lower output phase noise, particularly noise close to the carrier, than the Xa input can provide. More on this if/when I measure it.

SMD assembly

For the above reasons I chose to try the 20-QFN C version part. It has provided advantages stated and potentially available. Using it does have a down side though. The part is ‘so good’ that it is presently sanctioned for sale in China where inexpensive SMD PCB board assembly is most inexpensively performed. To date evey part I have needed to have a finished PCB created and assembled there except for the C version synthesizer has been possible. I’ve had to add and reflow solder the C version synthesizer myself to a complete an otherwise finished PCB.

```
const double epsilon = 1.0e-16;
const uint32_t SI5351_MAX_DENOMINATOR = SI5351_MULTISYNTH_C_MAX;
const double CLOCK_TOLERANCE = 1e-10;
void rational_approximation(double value, uint32_t max_denominator, uint32_t *a, uint32_t *b,
uint32_t *c) {


  double af;
  double f0 = modf(value, &af); // f0 is fractional part of target value
  *a = (uint32_t) af; //  counting on cast to uint32_t to strip up fraction and leave integer portion
  *b = 0;
  *c = 1;
  double f = f0;
  double delta = f0;
  /* we need to take into account that the fractional part has a_0 = 0 */
  uint32_t h[] = {1, 0};
  uint32_t k[] = {0, 1};
  for (int i = 0; i < 100; ++i) {
    if (f <= epsilon) {
      break;
    }
    double anf;
    f = modf(1.0 / f, &anf);
    uint32_t an = (uint32_t) anf;
    for (uint32_t m = (an + 1) / 2; m <= an; ++m) {
      uint32_t hm = m * h[1] + h[0];
      uint32_t km = m * k[1] + k[0];
      if (km > max_denominator) {
        break;
      }
      double d = fabs((double) hm / (double) km - f0);
      if (d < delta) {
        delta = d;
        *b = hm;
        *c = km;
      }
    }
    uint32_t hn = an * h[1] + h[0];
    uint32_t kn = an * k[1] + k[0];
    h[0] = h[1]; h[1] = hn;
    k[0] = k[1]; k[1] = kn;
  }
  return;
}
```