Calibrate_Si5251 – V1.1 – Operator’s Manual
John Price – WA2FZW

Table of Contents

Introduction

This program was originally created in conjunction with the ESP32 based VFO program that I have been developing along with Glenn (VK3PE) and Jim (G2ZQC). The Si5351 software is a modified version of the code used in TJ Uebo's (JF3HZB) digital VFO.

Before using the Si5351 module, its output frequencies should be calibrated. The crystal frequency of the Si5351 is nominally 25 MHz (27 MHz on some), but all of the Si5351 modules we've tested have been off by a little bit (some much more than others).

The original version of the program was designed to only work on the ESP32, but realizing that the application has more general uses, I modified it to also run on an Arduino (I've only tested it on a UNO, but there's no reason it won't work on other versions that I can see).

The program's user interface is the Arduino IDE's Serial Monitor.

How the Calibration Works

It took us a number of attempts to figure out how to compensate for the off-frequency crystals and the answer was found in the Etherkit Si5351 library by Jason Milldrum (NT7S).

Unfortunately, although the Etherkit library seems to work fine on the Arduino, we could not make it work on the ESP32.

This program is used to set a *correction factor* which is used in the math that converts the crystal frequency to the output frequency. I'm not going to try to explain in detail how all that math works; heck, I barely understand it myself, but in a nutshell, the crystal frequency is multiplied by some number to set a PLL frequency in the range of 600 MHz to 900 MHz, then that number is divided back down to get the output frequency.

The *correction factor* set by this program is used to adjust the output frequency prior to setting up the PLL.

The value of the *correction factor* is the frequency error of the crystal expressed in parts per billion (PPB). I'll explain how to actually do this in the program in the section titled [Determining The Starting Point](), below, but here's the math:

> Error = Nominal Crystal Frequency – Actual Crystal Frequency
> Error Factor = Error / Nominal Crystal Frequency
> Correction Factor = Error Factor x 10^9

Using some real numbers from one of my Si5351s:

> Error = 25,000,000 – 25,001,010 = -1,010
> Error Factor = -1,010 / 25,000,000 = -0.0000404
> Correction Factor = -0.0000404 x 10^9 = -40,400

Note, the actual crystal frequency in this example was higher than the nominal frequency and the *correction factor* is negative. This is correct. The actual *correction factor* for this particular Si5351 after some tweaking was determined to -40,450.

The Program

Symbols in the *Cal_Config.h* File

The *Calibrate_Si5351* program includes a file named *Cal_Config.h* which contains a few definitions that are needed by the *Calibrate_Si5351* program.

There are a few things in the *Cal_onfig.h* file that you may need to change or must change.

The main difference between the Arduino and the ESP32 is the EEPROM library that is used for each. The symbol *USE_EEPROM* must be set to either *ESP32* or *ARDNO* if you want the program to be able to save the correction in the EEPROM. If you do not want to use the EEPROM, set the *USE_EEPROM* symbol to *DON'T_USE*.

Most Si5351 breakout boards come with a 25MHz crystal however they can also use a 27MHz crystal. If you have such a device, change the frequency.

```
#define SI_XTAL       25000000UL      // Default crystal frequency
```

The setting of the *TEST_FREQ* symbol defines the default frequency at which to perform the calibration. If the user just hits ’Enter’ when asked for the frequency at which to perform the calibration, this value will be used. It is currently set to 10MHz.

The Si5351 module used in the program does not use the standard Wire libraries for either the Arduino or the ESP32, thus there are no default pin numbers for the I2C bus. These definitions define which pins are used (the settings shown are those used on my ESP32):

```
#define SI_SDA        26              // Non-standard I2C bus GPIO
#define SI_SCL        27              // pins used by the VFO program
```

One thing to be aware of: if there are other I2C devices on the processor, the Si5351 ***MUST*** use alternate pins as there is some conflict between TJ’s code and the Wire library which is the normal vehicle for handling I2C devices.

The standard I2C address for the Si5351 is 0x60, but we have seen one with an address of 0x62. If you need to change the address, do so.

```
#define SI_I2C_ADDR   0x60            // I2C address of the Si5351
```

The VFO_DRIVE symbol defines the Si5351’s output current into an 85Ω load. You shouldn’t need to change it, but you can.

```
#define VFO_DRIVE     CLK_DRIVE_8MA   // Can be _2MA, -4MA or _6M
```

I have no idea what the real expected range of the correction factor can be, but for the time being, the CAL_LIMIT is set to 1,000,000. The highest value we’ve seen in testing is in the -204K range, with -35K to -40K being the more usual value.

```
#define CAL_LIMIT 1000000L            // Correction factor range limit
```

There are a few other definitions in the file that you shouldn’t mess with. They are not used in the calibration program, but are needed to correctly compile the program.

Running the Calibration Program

If the ESP32 is not already connected to your PC, connect it. If it's already connected and you are using the USB interface for some CAT control program, you will have to terminate any programs using the USB port.

To run the program, open the *Calibrate_Si5351.ino* program in the Arduino IDE. Make sure that under the IDE's "Tools" menu you have the correct board type and port selected, then type "Ctrl-Shift-M" to open the Serial Monitor window. At the bottom of that window are buttons to set the baud rate and the input termination character(s). The baud rate should be set to 115200 and the line termination should be set to "Newline".

The CLK2 output of the Si5351 module is used for the output frequency of the calibration program. You can use an accurate frequency counter to measure the output frequency or you can zero-beat the signal against a known accurate received signal in your receiver (such as WWV).

It is best to perform the calibration at a frequency which is near the mid-range of the VFO for the radio you intend to use the VFO in if possible. For example in the case of the Yaesu FT-7, the VFO range is from 5MHz to 5.5MHz, so the ideal frequency to perform the calibration at would be 5.25MHz. For my Swan-250C, I used 39.352MHz, which is the expected VFO frequency for an operating frequency of 50.250MHz.

Once everything is set up, compile the program and download it to the ESP32.

When it starts up, you may see some nonsense from the ESP32 bootloader in the Serial Monitor window followed by:

```
     Si5351 Calibration Program – V1.1

     Initializing EEPROM        🡐 You won't see this on an Arduino

     Enter calibration test frequency in Hertz
```

Determining the Starting Point

This is easier if you are using a counter to measure the output frequency than if you are using a receiver as the exact output frequency is going to be a bit more difficult to determine using a radio.

Make sure the counter and Si5351 have had ample time to warm up and stabilize.

In the line at the top of the Serial Monitor window, enter the frequency in Hertz that you wish to use for the calibration. Commas or periods may be included in the number for clarity. For this example, we’ll use a frequency of 10MHz, so enter:

```
     10,000,000      (or just 10000000, or 10.000.000)
```

The program confirms the test frequency and displays the instructions:

```
     Calibration will be performed at: 10,000,000 Hz
```

If you simply hit ‘Enter’ when asked for the test frequency, the program will use the default frequency defined by the value of the TEST_FREQ symbol and you will dee the following:

```
     Using default test frequency.
     Calibration will be performed at: 10,000,000 Hz
```

When asked to enter a new correction factor, you may enter:

```
    nnnnn (or =nnnnn) to set a new value
    +nnnn to add an offset to the current value
    -nnnn to subtract an offset from the current value
    The enter key to use the current value
    or 'R' or 'r' to reset the correction factor to zero
    or 'S' or 's' to save the correction and get a new test frequency
    or 'N' or 'n' to get a new test frequency without saving
    or 'Q' or 'q' to quit and save the value in the EEPROM
```

```
Checking for previous correction factor.
```

If you have the *USE_EEPROM* symbol set to *DON'T_USE*, the 'S' option will not be included in the option list and the message after the option list will be:

```
EEPROM is not available; correction factor set to zero!
```

If using the EEPROM, the program is looking to see if you've already saved a *correction factor* in the EEPROM. If there is a valid *correction factor* already set in the EEPROM, the program will start with that setting and you will see a message like:

```
EEPROM contains correction factor: 7,000
We will start with that value.
```

If there is not a valid *correction factor* already in the EEPROM, you will see:

```
EEPROM does not contain a valid correction factor.
Initializing the correction factor to:   0
```

Next, you will be asked to enter a new correction factor:

```
Enter a new correction factor or offset or
hit 'Enter' to use the current correction factor
```

At this point, in order to determine the initial *correction factor*, you want to enter the 'R' command to reset the *correction factor* to zero if it isn't already.

Observe the actual output frequency. For the Si5351 I used above in the section titled *How it Works*, the actual output frequency was 10,000,41x with the test frequency set at 10,000,000 (note, my counter doesn't display units, so the low order digit is unknown; I assumed it was zero).

Let’s do the math:

```
Error = 10,000,000 – 100,000,410 = -410
Error Correction = -410 / 10,000,000 = -0.000041
```

Correction Factor = -0.000041 x 10^9 = -41,000

Again the actual *correction factor* for this Si5351 after tweaking was -40,450.

At this point the program should still be asking you to enter a new *correction factor*:

```
Enter a new correction factor or offset or
hit 'Enter' to use the current correction factor
```

Enter -41,000

Tweaking the Correction Factor

After entering the initial guess at the correction factor, the program will confirm your entry and you will be asked again:

```
Correction factor set to: -41,000

Enter a new correction factor or offset or
hit 'Enter' to use the current correction factor
```

Now it’s just a matter of entering offsets to the initial correction factor until the Si5351 produces the correct output frequency, although I’ve only had to do this on one Si5351; again most likely due to the fact that on my counter I have to guess at the unit’s digit.

If the output frequency is higher than the desired frequency, enter negative offsets; if the output frequency is lower than the desired output frequency, enter positive offsets.

The offsets are entered by entering the desired offset preceded by either a ‘+’ sign or a ‘-‘ sign, for example:

```
-100        (lower the correction factor by 100)
+100        (raise the correction factor by 100)
```

Make sure to include the sign; if omitted, the program will interpret the number as a totally new *correction factor* as opposed to being an offset.

The program will again inform you as to the new correction factor and then prompt you to do it again:

```
Correction factor set to: -41,100     (I entered -100)

Enter a new correction factor or offset or
hit 'Enter' to use the current correction factor
```

You can continue tweaking the correction factor until your counter shows the correct output frequency or until a zero-beat is heard in your receiver. Each time you change the *correction factor* the output frequency will change by some amount. The amount of change is going to depend on how far off frequency the crystal actually is. For the three Si5351s that we’ve tested, a change of about 35 to 40 in the *correction factor* (in either direction) resulted in a 1 Hz change in the output frequency.

Once you are satisfied with the calibration, there are three more actions you can take if use of the EEPROM is enabled:

If you enter ‘S’ or ‘s’, the program will save the current *correction factor* in the EEPROM and cycle back to ask for a new calibration test frequency.

If you enter ‘N’ or ‘n’, the program will cycle back to ask for a new calibration test frequency without saving the *correction factor*.

If you enter ‘Q’ or ‘q’ on the command line the program will save the *correction factor* and terminate; you will see:

```
Correction factor is saved in the EEPROM
Execution terminated!
```

If the EEPROM is not being used, entering the ‘S’ command or the ‘Q’ command will produce the message:

```
EEPROM is not available; correction factor not saved!
```

Error Messages

There a couple of possible error messages that you may see.

Should there be a problem initializing the EEPROM (on the ESP32 only), you will get the message:

```
Failed to initialize EEPROM
Execution terminated!
```

This indicates there is some problem with the ESP32 itself, not the software; try a different processor.

Of the three Si5351s that we’ve tested this with, one had a *calibration factor* of ~204,000; the other two were ~40,000. The program in fact allows for a much wider range but considers anything over plus or minus 1,000,000 to be bogus. If the *calibration factor* is moved beyond that range, you will get the message:

```
Correction factor: n,nnn,nnn
Is out of the legal range; not changed.
```

If this is a problem in your application, you can change the value of the symbol *LIMIT* in the *config.h* file, although I would me more inclined to question the reliability of the Si5351 in this case.

Now That I Know the Correction Factor

What do I do with it?

If you look at line 397 in the *Si5351.cpp* file you will see the formula that adjusts the desired output frequency prior to computing the *ClockBuilder* numbers to set up the PLL. Huh?

ClockBuilder is a program available from Silicon Labs that allows one to compute all the numbers that need to be fed to the Si5351 in order for it to produce a desired output frequency. The code in the *DoTheMath* function in the *Si5351.cpp* module mimics the *ClockBuilder* code.

You can use the Si5351 code in this program in other programs; the original code was used in TJ Uebo's (JF3HZB) digital VFO. I added the function *SetCorrection* to TJ's code which you can use to set the *correction factor* for some other application. If you use the Etherkit Si5351 library, it contains the function *set_correction*, which does the same thing. In either case, you need only call that function once in your *setup* code and the library code will use it any time you set an output frequency.

Here's a walkthrough of the formula on line 397 in the *Si5351.cpp* module using the values for one of Glenn's Si5351 modules which happens to be using a 25MHz TCXO in place of the crystal (more accurate than a crystal, but still not exactly on frequency).

Frequency adjustment formula (with the typecasting removed) is:

```
freq = freq + ((((( xtalCorr ) << 31 ) / 1000000000LL ) * freq ) >> 31 );
```

Glenn's correction factor (using the procedure described above in the section *Determining the Starting Point* is:

```
xtalCorr = 970

xtalCorr = 0x3CA

xtalCorr = 0011 1100 1010
```

Shifting that number 31 places to the left gives:

```
( xtalCorr << 31 ) =
        0001 1110 0101 0000 0000 0000 0000 0000 0000 0000 0000

( xtalCorr << 31 ) = 1E500000000

( xtalCorr << 31 ) = 2,083,059,138,560
```

Dividing by one billion:

```
      2,083,059,138,560 / 1,000,000,000 = 2,083    (note loss of precision)
```

Assume calibrating at 10MHz:

```
      freq = 10,000,000

      2,083 * freq = 20,830,000,000

      2,083 * freq = 4D9909380

      2,083 * freq =
              0100 1101 1001 1001 0000 1001 0011 1000 0000 0000 1001
```

Shifting that result 31 places to the right:

```
      ( 2,083 * freq ) >> 31 = 0000 1001 = 9   (note loss of precision)
```

End result:

```
      (Adjusted) freq + (all that) = 10,000,009
```

The other popular library for the Si5351 is available from Adafruit. It's not obvious to me how to incorporate the *correction factor* into that library as it seems to expect the using program to set up the *ClockBuilder* data as opposed to how the Etherkit library works.

Accuracy

Because of the convoluted math used to set up the multipliers and dividers in the Si5351, this calibration process is not an exact science. The math in some cases loses a bit of precision (as can be seen in the above example) so the calibration will be reasonably close over a narrow range it might not be 100% accurate over a wide range of frequencies.

That being the case, it's best to perform the calibration at a frequency close to the range over which you intend to use the Si5351.

Example Output

The following is a complete sample of what you can expect to see as the program progresses (with the EEPROM turned on):

```
Si5351 Calibration Program - V1.1

Initializing EEPROM

Enter calibration test frequency in Hertz        🡐 Entered 10,000,000
Calibration will be performed at: 10,000,000 Hz

When asked to enter a new correction factor, you may enter:

    nnnnn (or =nnnnn) to set a new value
    +nnnn to add an offset to the current value
    -nnnn to subtract an offset from the current value
    The enter key to use the current value
    or 'R' or 'r' to reset the correction factor to zero
    or 'S' or 's' to save the correction and get a new test frequency
    or 'N' or 'n' to get a new test frequency without saving
    or 'Q' or 'q' to quit and save the value in the EEPROM

Checking for previous correction factor.


EEPROM contains correction factor: 7,000
We will start with that value.


Enter a new correction factor or offset or
hit 'Enter' to use the current correction factor 🡐 Entered -4,000

Correction factor set to: 3,000

Enter a new correction factor or offset or
hit 'Enter' to use the current correction factor 🡐 Entered 'r'

Correction factor set to:   0

Enter a new correction factor or offset or
hit 'Enter' to use the current correction factor 🡐 Entered 3,000
```

```
Correction factor set to: 3,000

Enter a new correction factor or offset or
hit 'Enter' to use the current correction factor ← Entered 's'

Correction factor is saved in the EEPROM

Enter calibration test frequency in Hertz        ← Entered 5000000
Calibration will be performed at:  5,000,000 Hz
```

Instructions are repeated here

```
Correction factor set to: 3,000                  ← Saved value

Enter a new correction factor or offset or
hit 'Enter' to use the current correction factor ← Entered 'q'

Correction factor is saved in the EEPROM
Execution terminated!
```

Suggestion Box

I welcome any suggestions for further improvements. Please feel free to email me at WA2FZW@ARRL.net.