RADIUNO V 1.5

This source file is under General Public License version 3.

Most source code are meant to be understood by the compilers and the computers.
Code that has to be hackable needs to be well understood and properly documented.
Donald Knuth coined the term Literate Programming to indicate code that is written be easily read and understood.

The Raduino is a small board that includes the Arduin Nano, a 16x2 LCD display and an Si5351a frequency synthesizer. This board is manufactured by Paradigm Ecomm Pvt Ltd.

To learn more about Arduino you may visit www.arduino.cc.

The Arduino works by first executing the code in a function called setup() and then it repeatedly keeps calling loop() forever. All the initialization code is kept in setup() and code to continuously sense the tuning knob, the function button, transmit/receive, etc. is all in the loop() function. If you wish to study the code top down, then scroll to the bottom of this file and read your way up.

Below are the libraries to be included for building the Raduino

The EEPROM library is used to store settings like the frequency memory, calibration data, callsign etc.

```
#include <EEPROM.h>
```

The main chip which generates up to three oscillators of various frequencies in the Raduino is the Si5351a. To learn more about Si5351a you can download the datasheet from www.silabs.com although, strictly speaking it is not a requirement to understand this code. Instead, you can look up the Si5351 library written by Jason Mildrum, NT7S. You can download and install it from

https://github.com/etherkit/Si5351Arduino

To compile this file.

NOTE: This sketch is based on version V2 of the Si5351 library. It will not compile with V1!

The Wire.h library is used to talk to the Si5351 and we also declare an instance of Si5351 object to control the clocks.

```
#include <Wire.h>
#include <si5351.h>
Si5351 si5351;
```

The Raduino board is the size of a standard 16x2 LCD panel. It has three connectors:

First, is an 8 pin connector that provides +5v, GND and six analog input pins that can also be configured to be used as digital input or output pins. These are referred to as A0, A1, A2, A3, A6 and A7 pins. The A4 and A5 pins are missing from this connector as they are used to talk to the Si5351 over I2C protocol.

A0	A1	A2	A3	+5v	GND	A6	A7
BLACK	BROWN	RED	ORANGE	YELLOW	GREEN	BLUE	VIOLET

Second is a 16 pin LCD connector. This connector is meant specifically for the standard 16x2 LCD display in 4 bit mode. The 4 bit mode requires 4 data lines and two control lines to work:

Lines used are:

```
RESET, ENABLE, D4, D5, D6, D7
```

We include the library and declare the configuration of the LCD panel too

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(8, 9, 10, 11, 12, 13);
```

The Arduino, unlike C/C++ on a regular computer with gigabytes of RAM, has very little memory. We have to be very careful with variables that are declared inside the functions as they are created in a memory region called the stack. The stack has just a few bytes of space on the Arduino if you declare large strings inside functions, they can easily exceed the capacity of the stack and mess up your programs. We circumvent this by declaring a few global buffers as kitchen counters where we can slice and dice our strings. These strings are mostly used to control the display or handle the input and output from the USB port. We must keep a count of the bytes used while reading the serial port as we can easily run out of buffer space. This is done in the serial_in_count variable.

```
char serial_in[32], c[30], b[30], printBuff[32];
int count = 0;
unsigned char serial_in_count = 0;
```

We need to carefully pick assignment of pin for various purposes. There are two sets of completely programmable pins on the Raduino. First, on the top of the board, in line with the LCD connector is an 8-pin connector that is largely meant for analog inputs and front-panel control. It has a regulated 5v output, ground and six pins. Each of these six pins can be individually programmed either as an analog input, a digital input or a digital output. The pins are assigned as follows:

A0,	A1,	A2,	A3,	+5v,	GND,	A6,	A7
BLACK	BROWN	RED	ORANGE	YELLOW	GREEN	BLUE	VIOLET

(while holding the board up so that back of the board faces you)

Though, this can be assigned anyway, for this application of the Arduino, we will make the following assignments:

A2 will connect to the PTT line, which is the usually a part of the mic connector.

A3 is connected to a push button that can momentarily ground this line. This will be used to switch between different modes, etc.

A6 is to implement a keyer, it is reserved and not yet implemented.

A7 is connected to a center pin of good quality 100K or 10K linear potentiometer with the two other ends connected to ground and +5v lines available on the connector. This implements the tuning mechanism.

```
#define ANALOG_KEYER (A1)
#define CAL_BUTTON (A2)
#define FBUTTON (A3)
#define PTT   (A6)
#define ANALOG_TUNING (A7)
```

The second set of 16 pins on the bottom connector are have the three clock outputs and the digital lines to control the rig.

This assignment is as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
~~+5V~~ +12	~~+5V~~ +12	CLK0	GND	GND	CLK1	ND	GND	CLK2	GND	D2	D3	D4	D5	D6	D7

NOTE: Pins 1 and 2 were incorrectly labeled as +5. These are +12 to power the 7805 +5V regulator on the Raduino board. Only pins 1-5 are connected to a connector. Pins 6-16 can have a proper connector added if required for future use.

These too are flexible with what you may do with them, for the Raduino, we use them to:

TX_RX line: Switches between Transmit and Receive after sensing the PTT or the morse keyer
CW_KEY line: turns on the carrier for CW

These are not used at the moment.

```
#define TX_RX (7)
#define CW_TONE (6)
#define CW_KEY (5)
#define TX_LPF_SEL (4)
```

The raduino has a number of timing parameters, all specified in milliseconds.

CW_TIMEOUT: how many milliseconds between consecutive keyup and keydowns before switching back to receive? The next set of three parameters determine what is a tap, a double tap and a hold time for the function button.

TAP_DOWN_MILLIS: upper limit of how long a tap can be to be considered as a button_tap

TAP_UP_MILLIS: upper limit of how long a gap can be between two taps of a button_double_tap.

TAP_HOLD_MILIS: many milliseconds of the button being down before considering it to be a button_hold.

```
#define TAP_UP_MILLIS (500)
#define TAP_DOWN_MILLIS (600)
#define TAP_HOLD_MILLIS (2000)
#define CW_TIMEOUT (600l) // in milliseconds, this is the parameter that
determines how long the tx will hold between cw key downs
```

The Raduino supports two VFOs: A and B and receiver incremental tuning (RIT). We define a variables to hold the frequency of the two VFOs, RITs the rit offset as well as status of the RIT.

To use this facility, wire up push button on A3 line of the control connector.

```
#define VFO_A 0
#define VFO_B 1
char ritOn = 0;
char vfoActive = VFO_A;
unsigned long vfoA = 7100000L, vfoB = 14200000L, ritA, ritB, sideTone = 800;
```

Raduino needs to keep track of current state of the transceiver. These are a few variables that do it.

```
char inTx = 0;
char keyDown = 0;
char isUSB = 0;
unsigned long cwTimeout = 0;
unsigned char txFilter = 0;
```

Tuning Mechanism of the Raduino

We use a linear pot that has two ends connected to +5 and the ground. the middle wiper is connected to ANALOG_TUNNING pin. Depending upon the position of the wiper, the reading can be anywhere from 0 to 1023. If we want to use a multi-turn potentiometer with a tuning range of 500 kHz and a step size of 50 Hz we need 10,000 steps which is about 10x more than the steps that the ADC provides. Arduino's ADC has 10 bits which results in 1024 steps only. We can artificially expand the number of steps by a factor 10 by oversampling 100 times. As a result we get 10240 steps.
The tuning control works in steps of 50Hz each for every increment between 10 and 10230. Hence the turning the pot fully from one end to the other will cover 50 x 10220 = 511 KHz. But if we use the standard 1-turn pot, then a tuning range of 500 kHz would be too much. (tuning would become very touchy). In the next few lines we can limit the tuning range depending on the potentiometer used and the band section of interest. Tuning beyond the limits is still possible by the 'scan-up' and 'scan-down' mode at the end of the pot. At the two ends, that is, the tuning starts stepping up or down in 10 KHz steps.
To stop the scanning the pot is moved back from the edge.
To rapidly change from one band to another, you press the function button and then move the tuning pot. Now, instead of 50 Hz, the tuning is in steps of 50 KHz allowing you rapidly use it like a 'bandset' control.
To implement this, we fix a 'base frequency' to which we add the offset that the pot points to. We also store the previous position to know if we need to wake up and change the frequency.

TUNING RANGE SETTINGS

Standard setting for 1-turn potentiometer: TUNING_RANGE 50, baseTune 7100000L (7.100 - 7.150).
For 10-turn pot: recommended value for TUNING-RANGE is 200, baseTune 7000000L (7.000 - 7.200).
If you are in ITU Region 2, you may want to use a TUNING_RANGE of 300 kHz to cover the entire 40m band (7.000 - 7.300). But of course you can change the settings to what suits you best.

```
#define TUNING_RANGE (50) // tuning range (in kHz) of the tuning pot
unsigned long baseTune =  7100000L; // frequency (Hz) when tuning pot is at
minimum position

#define INIT_BFO_FREQ (1199800L)
unsigned long bfo_freq = 11998000L;
int  old_knob = 0;

#define CW_OFFSET (800l)

#define LOWEST_FREQ  (6995000L) // absolute minimum frequency (Hz)
#define HIGHEST_FREQ (7500000L) //  absolute maximum frequency (Hz)

long frequency, stepSize = 100000;
```

The raduino can be booted into multiple modes:

MODE_NORMAL: works the radio normally
MODE_CALIBRATION: used to calibrate Raduino.

To enter this mode, hold the function button down and power up. Tune to exactly 10 MHz on clock0 and release the function button.

```
#define MODE_NORMAL (0)
#define MODE_CALIBRATE (1)
char mode = MODE_NORMAL;
```

Display Routines

These two display routines print a line of characters to the upper and lower lines of the 16x2 display.

```
void printLine1(char *c) {
  if (strcmp(c, printBuff)) {
    lcd.setCursor(0, 0);
    lcd.print(c);
    strcpy(printBuff, c);
    count++;
  }
}
void printLine2(char *c) {
  lcd.setCursor(0, 1);
  lcd.print(c);
}
```

Building upon the previous two functions, update Display paints the first line as per current state of the radio.

At present, we are not using the second line. You could add a CW decoder or SWR/Signal strength indicator.

```
void updateDisplay() {
  sprintf(b, "%08ld", frequency);
  sprintf(c, "%s:%.2s.%.4s", vfoActive == VFO_A ? "A" : "B" , b,  b + 2);
  if (isUSB)
    strcat(c, " USB");
  else
    strcat(c, " LSB");
  if (inTx)
    strcat(c, " TX");
  else if (ritOn)
    strcat(c, " +R");
  else
    strcat(c, "   ");
  printLine1(c);
}
```

To use calibration sets the accurate readout of the tuned frequency:

To calibrate, follow these steps:

1. Tune in a signal that is at a known frequency.
2. Now, set the display to show the correct frequency,
   the signal will no longer be tuned up properly
3. Press the CAL_BUTTON line to the ground (pin A2 - red wire)
4. Tune in the signal until it sounds proper.
5. Release CAL_BUTTON

In step 4, when we say 'sounds proper' then, for a CW signal/carrier it means zero-beat, and for SSB it is the most natural sounding setting.

Calibration is an offset that is added to the VFO frequency by the Si5351 library. We store it in the EEPROM's first four bytes and read it in setup() when the Radiuno is powered up.

```
    void calibrate() {
      int32_t cal;
```

The tuning knob gives readings from 0 to 1000.
Each step is taken as 10 Hz and the mid setting of the knob is taken as zero.

```
      cal = (analogRead(ANALOG_TUNING) * 10) - 5000;
```

If the button is released, we save the setting and delay anything else by 5 seconds to debounce the CAL_BUTTON.

Debounce: it is the rapid on/off signals that happen on a mechanical switch when you change its state.

```
      if (digitalRead(CAL_BUTTON) == HIGH) {
        mode = MODE_NORMAL;
        printLine1((char *)"Calibrated      ");
```

“Calibration fix" by Allard, PE1NWL
.
Calculate the correction factor in parts-per-billion (offset in relation to the osc. frequency).

```
        cal = (cal * -1000000000LL) / (bfo_freq - frequency) ;
```

Apply the correction factor.

```
        si5351.set_correction(cal);
```

Write the 4 bytes of the correction factor into the eeprom memory.

```
        EEPROM.write(0, (cal & 0xFF));
        EEPROM.write(1, ((cal >> 8) & 0xFF));
        EEPROM.write(2, ((cal >> 16) & 0xFF));
        EEPROM.write(3, ((cal >> 24) & 0xFF));
        printLine2((char *)"Saved.    ");
        delay(5000);
      }
      else {
```

While the calibration is in progress (CAL_BUTTON is held down), keep tweaking the frequency as read out by the knob, display the change in the second line.

```
        si5351.set_freq((bfo_freq - frequency) * 100LL, SI5351_CLK2);
        sprintf(c, "offset:%d ", cal);
        printLine2(c);
```

Calculate the correction factor in ppb and apply it.

```
        cal = (cal * -1000000000LL) / (bfo_freq - frequency) ;
        si5351.set_correction(cal);
      }
    }
```

The setFrequency is a little tricky routine, it works differently for USB and LSB.

The BITX BFO is permanently set to lower sideband, (that is, the crystal frequency is on the higher side slope of the crystal filter).

LSB: The VFO frequency is subtracted from the BFO. Suppose the BFO is set to exactly 12 MHz and the VFO is at 5 MHz. The output will be at 12.000 - 5.000 = 7.000 MHz.

USB: The BFO is subtracted from the VFO. Makes the LSB signal of the BITX come out as USB!!

Here is how it will work:

Consider that you want to transmit on 14.000 MHz and you have the BFO at 12.000 MHz. We set the VFO to 26.000 MHz. Hence, 26.000 - 12.000 = 14.000 MHz. Now, consider you are whistling a tone of 1 KHz. As the BITX BFO is set to produce LSB, the output from the crystal filter will be 11.999 MHz. With the VFO still at 26.000, the 14 Mhz output will now be 26.000 - 11.999 = 14.001, hence, as the frequencies of your voice go down at the IF, the RF frequencies will go up!

Thus, setting the VFO on either side of the BFO will flip between the USB and LSB signals.

```
    void setFrequency(unsigned long f) {
      uint64_t osc_f;
      if (isUSB) {
        si5351.set_freq((bfo_freq + f) * 100ULL, SI5351_CLK2);
      }
      else {
        si5351.set_freq((bfo_freq - f) * 100ULL, SI5351_CLK2);
      }
      frequency = f;
    }
```

The CheckTX toggles the T/R line. The current BITX wiring up doesn't use this. But, if you would like to make use of RIT, etc, you must wire up an NPN transistor to the PTT line as follows:

Emitter to ground, base to TX_RX line through a 4.7K resistor (as defined at the top of this source file) collector to the PTT line.

Now, connect the PTT to the control connector's PTT line (see the definitions on the top).

Yeah, surprise! We have CW supported on the Raduino.

```
    void checkTX() {
```

We don't check for ptt when transmitting cw as long as the cwTimeout is non-zero, we will continue to hold the radio in transmit mode

```
      if (cwTimeout > 0)
        return;

      if (digitalRead(PTT) == 0 && inTx == 0) {
        inTx = 1;
        digitalWrite(TX_RX, 1);
        updateDisplay();
      }

      if (digitalRead(PTT) == 1 && inTx == 1) {
        inTx = 0;
        digitalWrite(TX_RX, 0);
        updateDisplay();
      }
    }
```

CW is generated by injecting the side-tone into the mic line.

Watch http://bitxhacks.blogspot.com for the CW hack.

nonzero cwTimeout denotes that we are in cw transmit mode.

This function is called repeatedly from the main loop() hence, it branches each time to do a different thing.

There are three variables that track the CW mode:

inTX: true when the radio is in transmit mode
keyDown: true when the CW is keyed down, you may be in transmit mode (inTX true) and yet between dots and dashes and hence keyDown could be true or false.
cwTimeout: Figures out how long to wait between dots and dashes before putting the radio back in receive mode.

```
    void checkCW() {

      if (keyDown == 0 && analogRead(ANALOG_KEYER) < 50) {
```

Switch to transmit mode if we are not already in it

```
        if (inTx == 0) {
          digitalWrite(TX_RX, 1);
```

Give the relays a few ms to settle the T/R relays.

```
          delay(50);
        }
        inTx = 1;
        keyDown = 1;
        tone(CW_TONE, sideTone);
        updateDisplay();
      }
```

Reset the timer as long as the key is down.

```
      if (keyDown == 1) {
        cwTimeout = CW_TIMEOUT + millis();
      }
```

If we have a keyup,

```
      if (keyDown == 1 && analogRead(ANALOG_KEYER) > 150) {
        keyDown = 0;
        noTone(CW_TONE);
        cwTimeout = millis() + CW_TIMEOUT;
      }
```

If we are in cw-mode and have a keyuup for a longish time;

```
      if (cwTimeout > 0 && inTx == 1 && cwTimeout < millis()) {
```

Move the radio back to receive

```
        digitalWrite(TX_RX, 0);
        inTx = 0;
        cwTimeout = 0;
        updateDisplay();
      }
    }
```

A trivial function to wrap around the function button.

```
    int btnDown() {
      if (digitalRead(FBUTTON) == HIGH)
        return 0;
      else
        return 1;
    }
```

A click on the function button toggles the RIT.
A double click switches between A and B vfos.
A long press copies both the VFOs to the same frequency.

```
    void checkButton() {
      int i, t1, t2, knob, new_knob, duration;
```

The rest of this function is interesting only if the button is pressed.

```
      if (!btnDown())
        return;
```

We are at this point because we detected that the button was indeed pressed! We wait for a while and confirm it again so we can be sure it is not some noise.

```
      delay(50);
      if (!btnDown())
        return;
```

Note the time of the button going down and where the tuning knob was.

```
      t1 = millis();
      knob = analogRead(ANALOG_TUNING);
      duration = 0;
```

If you move the tuning knob within 3 seconds (3000 milliseconds) of pushing the button down then consider it to be a coarse tuning where you can move by 100 Khz in each step. This is useful only for multiband operation.

```
      while (btnDown() && duration < 3000) {
        new_knob = analogRead(ANALOG_TUNING);
```

Has the tuning knob moved while the button was down from its initial position?

```
        if (abs(new_knob - knob) > 10) {
          int count = 0;
```

Track the tuning and return.

```
          while (btnDown()) {
            frequency = baseTune = ((analogRead(ANALOG_TUNING) * 30000L) +
    1000000L);
            setFrequency(frequency);
            updateDisplay();
            count++;
            delay(200);
          }
          delay(1000);
          return;
        } /* end of handling the bandset
        delay(100);
        duration += 100;
      }
```

```
We reach here only upon the button being released. If the button has been down
for more than TAP_HOLD_MILLIS, we consider it a long press. Set both VFOs to
the same frequency, update the display and be done.

      if (duration > TAP_HOLD_MILLIS) {
        printLine2((char *)"VFOs reset!");
        vfoA = vfoB = frequency;
        delay(300);
        updateDisplay();
        return;
      }

t1 holds the duration for the first press.

      t1 = duration;

Now wait for another click

      delay(100);

If there a second button press, toggle the VFOs.

      if (btnDown()) {

Swap the VFOs on double tap.

        if (vfoActive == VFO_B) {
          vfoActive = VFO_A;
          vfoB = frequency;
          frequency = vfoA;
        }
        else {
          vfoActive = VFO_B;
          vfoA = frequency;
          frequency = vfoB;
        }
        //printLine2((char *)"VFO swap! ");
        delay(600);
        updateDisplay();
      }

No, there was not more taps.

      else {

On a single tap, toggle the RIT.

        if (ritOn)
          ritOn = 0;
        else
          ritOn = 1;
        updateDisply();
      }
    }
```

The Tuning mechanism of the Raduino works in a very innovative way. It uses a tuning potentiometer. The tuning potentiometer that a voltage between 0 and 5 volts at ANALOG_TUNING pin of the control connector. This is read as a value between 0 and 1000. By 100x oversampling this range is expanded by a factor 10. Hence, the tuning pot gives you 10,000 steps from one end to the other end of its rotation. Each step is 50 Hz, thus giving maximum 500 Khz of tuning range. The tuning range is scaled down depending on the limit settings. The standard tuning range (for the standard 1-turn pot) is 50 Khz. But it is also possible to use a 10-turn pot to tune across the entire 40m band. In that case you need to change the values for TUNING_RANGE and baseTune. When the potentiometer is moved to either end of the range, the frequency starts automatically moving up or down in 10 Khz increments

```
void doTuning() {
  long knob = 0;
```

The knob value normally ranges from 0 through 1023 (10 bit ADC).
In order to expand the range by a factor 10, we need 10^2 = 100x oversampling.

```
  for (int i = 0; i < 100; i++) {
    knob = knob + analogRead(ANALOG_TUNING) - 10;
```

Take 100 readings from the ADC.

```
  }
  knob = knob / 10L;
```

Take the average of the 100 readings and multiply the result by 10.
Now the knob value ranges from -100 through 10130.

The knob is fully on the low end, move down by 10 Khz and wait for 200 msec

```
  if (knob < -80 && frequency > LOWEST_FREQ) {
    baseTune = baseTune - 10000L;
    frequency = baseTune + (50L * knob * TUNING_RANGE / 500);
    updateDisplay();
    setFrequency(frequency);
    delay(200);
  }
```

The knob is full on the high end, move up by 10 Khz and wait for 200 msec.

```
  else if (knob > 10120L && frequency < HIGHEST_FREQ) {
    baseTune = baseTune + 10000L;
    frequency = baseTune + (50L * knob * TUNING_RANGE / 500);
    setFrequency(frequency);
    updateDisplay();
    delay(200);
  }
```

The tuning knob is at neither extremities, tune the signals as usual ("flutter fix" by Jerry, KE7ER).

```
      else if (knob != old_knob) {
        static char dir_knob;
        if ( (knob > old_knob) && ((dir_knob == 1) || ((knob - old_knob) > 5)) ||
             (knob < old_knob) && ((dir_knob == 0) || ((old_knob - knob) > 5)) )   {
          if (knob > old_knob) {
            dir_knob = 1;
            frequency = baseTune + (50L * (knob - 5) * TUNING_RANGE / 500);
          } else {
            dir_knob = 0;
            frequency = baseTune + (50L * knob * TUNING_RANGE / 500);
          }
          old_knob = knob;
          setFrequency(frequency);
          updateDisplay();
        }
      }
    }
```

Setup is called on boot up. It setups up the modes for various pins as inputs or outputs initializes the Si5351 and sets various variables to initial state. Just in case the LCD display doesn't work well, the debug log is dumped on the serial monitor. Choose Serial Monitor from Arduino IDE's Tools menu to see the Serial.print messages.

```
    void setup()
    {
      int32_t cal;
      lcd.begin(16, 2);
      printBuff[0] = 0;
      printLine1((char *)"Raduino v1.05");
      printLine2((char *)"             ");
```

Start serial and initialize the Si5351.

```
      Serial.begin(9600);
      analogReference(DEFAULT);
      Serial.println("*Raduino booting up\nv1.05\n");
```

Configure the function button to use the external pull-up.

```
      pinMode(FBUTTON, INPUT);
      digitalWrite(FBUTTON, HIGH);
      pinMode(PTT, INPUT);
      digitalWrite(PTT, HIGH);
      pinMode(CAL_BUTTON, INPUT);
      digitalWrite(CAL_BUTTON, HIGH);
      pinMode(CW_KEY, OUTPUT);
      pinMode(CW_TONE, OUTPUT);
      digitalWrite(CW_KEY, 0);
      digitalWrite(CW_TONE, 0);
      digitalWrite(TX_RX, 0);
      delay(500);
```

Fetch the correction factor from EEPROM

```
      EEPROM.get(0, cal);
      Serial.println("fetched correction factor from EEPROM:");
      Serial.println(cal);
```

Initialize the SI5351 and apply the correction factor

```
      si5351.init(SI5351_CRYSTAL_LOAD_8PF, 25000000L, cal);
      Serial.println("*Initiliazed Si5351\n");
      si5351.set_pll(SI5351_PLL_FIXED, SI5351_PLLA);
      si5351.set_pll(SI5351_PLL_FIXED, SI5351_PLLB);
      Serial.println("*Fixed PLL\n");
      si5351.output_enable(SI5351_CLK0, 0);
      si5351.output_enable(SI5351_CLK1, 0);
      si5351.output_enable(SI5351_CLK2, 1);
```

Increase the VFO drive level to 4mA to kill the birdie at 7199 kHz. You may try different drive strengths for the best results accepted values are 2,4,6,8 mA.

```
      si5351.drive_strength(SI5351_CLK2, SI5351_DRIVE_4MA);
      Serial.println("*Output enabled PLL\n");
      si5351.set_freq(500000000L , SI5351_CLK2);
      Serial.println("*Si5350 ON\n");
      mode = MODE_NORMAL;
      delay(10);
    }

    void loop() {
      if (digitalRead(CAL_BUTTON) == LOW && mode == MODE_NORMAL) {
        mode = MODE_CALIBRATE;
        si5351.set_correction(0);
        printLine1((char *)"Calibrating: Set");
        printLine2((char *)"to zerobeat.    ");
        delay(2000);
        return;
      }
      else if (mode == MODE_CALIBRATE) {
        calibrate();
        delay(50);
        return;
      }
      /*
        checkCW();
        checkTX();
        checkButton(); */
      doTuning();
      delay(50);
    }
```