BITX40 and the ARDUINO - Hardware build & Software Upgrades

Oct 13 2017

Building notes:
1. The holes in the arduino board are SO CLOSE to the display that I have given up on cutting an aperture in thin metal directly. Futhermore, all my efforts in cardboard boxes and plywood were unsuccessful because their thickness seems to always be too great for the screws and materials of this kit.
2. Building the radio inside a moderate sized thin aluminum or steel cookware intended to make bread or cakes seems to be the best.
3. Create a bezel made out of thin plastic (I used a portion of a Wendy Salad Plate) – I have a stencil of exactly how to cut (using a utility knife) the outline of the Arduino display) – then you affix the Arduino to this thin plastic, and you affix the thin plastic to a larger, simpler hole you can cut in the cookware using a couple of 1/4" drilled holes and then use a metal-cutting saber saw (fine toothed) to cut that opening. Use a couple of screws to hold the thin plastic to your cookware.
4. The holes for the other controls are as follows:

a) tuning potentiometer requires 3/8" hole, and if possible, a small hole off to the side for the index pin.
b) volume control requires slightly larger than 1/4" (you may have to use pliers to enlarge a 1/4" hole) and if possible a small hole off to the side for the index pin
c) The BNC connector requires 3/8"
d) The power connector requires 1/2" hole

Oct 10 2017

"sketch": The c-code that gets compiled to run the Arduio

1. Download the Integrated Development Environment (IDE) from the Arduino web site: https://www.arduino.cc/en/Main/Software Drop down the page until you find the one you can download for your operating system.....I found this much much easier than the web version for some reason.

They have it for many different operating systems, for the plain vanilla Windows installer is here: https://www.arduino.cc/download_handler.php?f=/arduino-1.8.5-windows.exe

At some point you'll need to pick thetype of Arduino you have (usually the Nano – it is a very little card on the Raduino; and also the port on your computer you'll be using (it takes a certain type of USB

card to fit the Raduino.

2. This fellow explains the basic idea, but I found a better one below:
http://oe6avd.net/category/projects/

3. This fellow has the very best update for the Raduino I've seen. His current version at the time of my writing is 1.25 https://github.com/amunters/bitx40. He adds all kinds of functions with one added pushbutton switch, such as receiver incremental tuning, A/B vfo, upper and lower sideband, ability to choose the VFO driver current (needs to be more for the 19 MHz upper side band operation), better calibrate routines, and even an electronic keyer.

You can find that on his page (or an even better one) but the link at the moment is this:
https://github.com/amunters/bitx40/blob/master/raduino_v1.25.1.ino

You will need to get his .ino project into your Integrated Development Environment and I don't remember exactly how I did this but you'll figure it out.

4. Inside theIDE, you'll compile it and upload it to the Arduino. It may create a new "project" for this.

5. These later versions of his code do NOT require the si5351a library, which means you have a lot less thatyou have to figure out in the Integrated Development Environment. He has put all the subroutines you need, right into his sketch.

6. To read about the clock frequency creator ship that the Arduino commands:
https://www.silabs.com/documents/public/data-sheets/Si5351-B.pdf

In order to get the Uppersideband 19MHz signal not to be reduced in strength, you'll need to (carefully unsolder c91 and C92 from the board.

see commentary here: https://groups.io/g/BITX20/topic/adding_switchable_usb_and_lsb/5650021?p=,,,20,0,0,0::recentpostdate%2Fsticky,,,20,2,0,5650021

and this photo: https://i0.wp.com/miscdotgeek.com/wp-content/uploads/2017/04/c9192.png

BitX40 schematic in black and white: http://qsl.net/nf4rc/bitx40SchematicBW.pdf

Additional knowledge: ubitx: http://www.phonestack.com/farhan/ubitx/ubitx.html

APPENDIX ONE: The Version 1.25 code

1.25 code:

```
/
**
   Raduino_v1.25.1 for BITX40 - Allard Munters PE1NWL (pe1nwl@gooddx.net)

   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 Arduino 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.

   First we define all user parameters. The parameter values can be changed by the user
   via SETTINGS menu, and are stored in EEPROM.
   The parameters values will be set to initial 'factory' settings after each
   version upgrade, or when the Function Button is kept pressed during power on.
   It is also possible to manually edit the values below. After that, initialize the
   settings to the new values by keeping the F-Button pressed during power on, or by
   switching the CAL wire to ground.
   */

   // *** USER PARAMETERS ***

   // tuning range parameters
   #define MIN_FREQ 7000000UL // absolute minimum tuning frequency in Hz
   #define MAX_FREQ 7300000UL // absolute maximum tuning frequency in Hz
   #define TUNING_POT_SPAN 50 // tuning pot span in kHz [accepted range 10-500 kHz]
   // recommended pot span for a 1-turn pot: 50kHz, for a 10-turn pot: 100 to 200kHz
   // recommended pot span when radio is used mainly for CW: 10 to 25 kHz
```

```
// USB/LSB parameters
#define OFFSET_LSB 0 // LSB offset in Hz [accepted range -10000Hz to 10000Hz]
#define OFFSET_USB 1500 // USB offset in Hz [accepted range -10000Hz to 10000Hz]
#define VFO_DRIVE_LSB 4 // VFO drive level in LSB mode in mA [accepted values 2,4,6,8 mA]
#define VFO_DRIVE_USB 8 // VFO drive level in USB mode in mA [accepted values 2,4,6,8 mA]

// CW parameters
#define CW_SHIFT 800 // RX shift in CW mode in Hz, equal to sidetone pitch [accepted range 200-1200 Hz]
#define SEMI_QSK true // whether we use semi-QSK (true) or manual PTT (false)
#define CW_TIMEOUT 350 // time delay in ms before radio goes back to receive [accepted range 10-1000 ms]

// CW keyer parameters
#define CW_KEY_TYPE 0 // type of CW-key: 0 (straight key), 1 (paddle), 2 (reverse paddle)
#define CW_SPEED 16 // CW keyer speed in words per minute [accepted range 1-50 WPM]
#define AUTOSPACE false // whether or not auto-space is enabled [accepted values: true or false]
#define DIT_DELAY 5 // debounce delay (ms) for DIT contact (affects the DIT paddle sensitivity)
#define DAH_DELAY 15 // debounce delay (ms) for DAH contact (affects the DAH paddle sensitivity)

// frequency scanning parameters
#define SCAN_START 7100 // Scan start frequency in kHz [accepted range MIN_FREQ - MAX_FREQ, see above]
#define SCAN_STOP 7150 // Scan start frequency in kHz [accepted range SCAN_START - MAX_FREQ, see above]
#define SCAN_STEP 1000 // Scan step size in Hz [accepted range 50Hz to 10000Hz]
#define SCAN_STEP_DELAY 500 // Scan step delay in ms [accepted range 0-2000 ms]

// Function Button
#define CLICK_DELAY 750 // max time (in ms) between function button clicks

// Capacitive touch keyer
#define CAP_SENSITIVITY 0 // capacitive touch keyer OFF (accepted range 0-25)

/**

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, etc.
*/

#include <EEPROM.h>

/**
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>

/**
The main chip which generates upto 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 requirment to understand this code.

We no longer use the standard SI5351 library because of its huge overhead due to many unused
features consuming a lot of program space. Instead of depending on an external library we now use
Jerry Gaffke's, KE7ER, lightweight standalone mimimalist "si5351bx" routines (see further down
the
code). Here are some defines and declarations used by Jerry's routines:
*/

#define BB0(x) ((uint8_t)x) // Bust int32 into Bytes
#define BB1(x) ((uint8_t)(x>>8))
#define BB2(x) ((uint8_t)(x>>16))

#define SI5351BX_ADDR 0x60 // I2C address of Si5351 (typical)
#define SI5351BX_XTALPF 2 // 1:6pf 2:8pf 3:10pf

// If using 27mhz crystal, set XTAL=27000000, MSA=33. Then vco=891mhz
#define SI5351BX_XTAL 25000000 // Crystal freq in Hz
#define SI5351BX_MSA 35 // VCOA is at 25mhz*35 = 875mhz

// User program may have reason to poke new values into these 3 RAM variables
uint32_t si5351bx_vcoa = (SI5351BX_XTAL*SI5351BX_MSA); // 25mhzXtal calibrate
uint8_t si5351bx_rdiv = 0; // 0-7, CLK pin sees fout/(2**rdiv)
uint8_t si5351bx_drive[3] = {1, 1, 1}; // 0=2ma 1=4ma 2=6ma 3=8ma for CLK 0,1,2

uint8_t si5351bx_clken = 0xFF; // Private, all CLK output drivers off

/**
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 GND +5V A6 A7
BLACK BROWN RED ORANGE YELLOW GREEN BLUE VIOLET (same color coding as used for resistors)

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 c[17], b[10], printBuff[2][17];

/**
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, GND, +5V, A6, A7
pin 8 7 6 5 4 3 2 1 (connector P1)
BLACK BROWN RED ORANGE YELLW 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
assignment:

A0 (digital input) for sensing the PTT. Connect to the output of U3 (LM7805) of the BITX40.
```

```
This way the A0 input will see 0V (LOW) when PTT is not pressed, +5V (HIGH) when PTT is pressed.
A1 (digital input) is to connect to a straight key, or to the 'Dit' contact of a paddle keyer.
Open (HIGH) during key up, switch to ground (LOW) during key down.
A2 (digital input) can be used for calibration by grounding this line (not required when you have the Function Button at A3)
A3 (digital input) is connected to a push button that can momentarily ground this line. This Function Button will be used to switch between different modes, etc.
A4 (already in use for talking to the SI5351)
A5 (already in use for talking to the SI5351)
A6 (analog input) is not currently used
A7 (analog input) 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 PTT_SENSE (A0)
#define KEY (A1)
#define CAL_BUTTON (A2)
#define FBUTTON (A3)
#define ANALOG_TUNING (A7)

bool PTTsense_installed; //whether or not the PTT sense line is installed (detected automatically during startup)

/**
The second set of 16 pins on the bottom connector P3 have the three clock outputs and the digital lines to control the rig.
This assignment is as follows :
Pin 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 (connector P3)
+12V +12V CLK2 GND GND CLK1 GND GND CLK0 GND D2 D3 D4 D5 D6 D7
These too are flexible with what you may do with them, for the Raduino, we use them to :

output D2 - PULSE : is used for the capacitive touch keyer
input D3 - DAH : is connected to the 'Dah' contact of an paddle keyer (switch to ground).
input D4 - SPOT : is connected to a push button that can momentarily ground this line. When the SPOT button is pressed a sidetone will be generated for zero beat tuning.
output D5 - CW_TONE : Side tone output
output D6 - CW_CARRIER line : turns on the carrier for CW
output D7 - TX_RX line : Switches between Transmit and Receive in CW mode
*/

#define PULSE (2)
```

```
#define DAH (3)
#define SPOT (4)
#define CW_TONE (5)
#define CW_CARRIER (6)
#define TX_RX (7)

/**
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, RIT, SPLIT
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 (Function Button)
*/

unsigned long vfoA, vfoB; // the frequencies the VFOs
bool ritOn = false; // whether or not the RIT is on
int RIToffset = 0; // offset (Hz)
int RIT = 0; // actual RIT offset that is applied during RX when RIT is on
int RIT_old;
bool splitOn; // whether or not SPLIT is on
bool vfoActive; // which VFO (false=A or true=B) is active
byte mode_A, mode_B; // the mode of each VFO

bool firstrun = true;
char clicks; // counter for function button clicks
bool locked = false;
byte param;

/**
We need to apply some frequency offset to calibrate the dial frequency. Calibration is done in
LSB mode.
*/
int cal; // LSB frequency offset in Hz (value is set in the SETTINGS menu)

/**
In USB mode we need to apply some additional frequency offset, so that zerobeat in USB is same as
in LSB
*/
int USB_OFFSET; // USB frequency offset in Hz (value is set in the SETTINGS menu)

/**
We can set the VFO drive to a certain level (2,4,6,8 mA)
*/
```

```
byte LSBdrive; // VFO drive level in LSB mode (value is set in the SETTINGS menu)
byte USBdrive; // VFO drive level in USB mode (value is set in the SETTINGS menu)

// scan parameters

int scan_start_freq; // lower scan limit (kHz) (value is set in the SETTINGS menu)
int scan_stop_freq; // upper scan limit (kHz) (value is set in the SETTINGS menu)
int scan_step_freq; // step size (Hz) (value is set in the SETTINGS menu)
int scan_step_delay; // step delay (ms) (value is set in the SETTINGS menu)

/**
Raduino has 4 modes of operation:
*/

#define LSB (0)
#define USB (1)
#define CWL (2)
#define CWU (3)

/**
Raduino needs to keep track of current state of the transceiver. These are a few variables that do it
*/
byte mode = LSB; // mode of the currently active VFO
bool inTx = false; // whether or not we are in transmit mode
bool keyDown = false; // whether we have a key up or key down
unsigned long TimeOut = 0; // time in ms since last key down
bool semiQSK; //whether we use semi QSK or manual PTT (value is set in the SETTINGS menu)
int QSK_DELAY; // in milliseconds, this is the parameter that determines how long the tx will hold between cw key downs (value is set in the SETTINGS menu)

//some variables used for the autokeyer function:

byte key_type = 0; // straight key (0), paddle (1) or reversed paddle (2) (value is set in the SETTINGS menu)
bool keyeron = false; //will be true while auto-keying
unsigned long released = 0;
bool ditlatch = false;
bool dahlatch = false;
byte wpm; // keyer speed (words per minute) (value is set in the SETTINGS menu)
byte gap = 1; // space between elements (1) or characters (3)
bool autospace = false;
```

```
unsigned long dit;
unsigned long dah;
unsigned long space = 0;

// some variable used by the capacitive touch keyer:

bool CapTouch_installed = true; // whether or not the capacitive touch modifiaction is installed (detected automatically during startup)
byte base_sens_KEY; // base delay (in us) when DIT pad is NOT touched (measured automatically by touch sensor calibration routine)
byte base_sens_DAH; // base delay (in us) when DAH pad is NOT touched (measured automatically by touch sensor calibration routine)
byte cap_sens; // additional delay to both touch pads (for controlling the touch sensitivity)
bool capaKEY = false; // true when DIT pad is touched
bool capaDAH = false; // true when DAH pad is touched

/**
Tks Pavel CO7WT:
Use pointers for reversing the behaviour of the dit/dah paddles, this way we can simply
refer to them as two paddles regardless it's normal or reversed
Macros don't like pointers, so we create two byte vars to hold the values
**/

byte _key = KEY;
byte _dah = DAH;
byte *paddleDIT = &_key; // Paddle DIT bind to KEY
byte *paddleDAH = &_dah; // Paddle DAH bind to DAH

/** 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 covering 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 0 and 10000.
Hence the turning the pot fully from one end to the other will cover 50 x 10000 = 500 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 SETTINGS menu we can limit the pot span
depending on the potentiometer used and the band section of interest. Tuning beyond the
```

```
limits is still possible by the fast '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.
*/

int POT_SPAN; // span (in kHz) from lower end to upper end of the tuning pot (value is set in the SETTINGS menu)
unsigned long baseTune = 7100000UL; // frequency (Hz) when tuning pot is at minimum position

#define bfo_freq (11998000UL)

int old_knob = 0;

int CW_OFFSET; // the amount of offset (Hz) during RX, equal to sidetone frequency (value is set in the SETTINGS menu)
int RXshift = 0; // the actual frequency shift that is applied during RX depending on the operation mode

unsigned long LOWEST_FREQ; // absolute minimum frequency (Hz) (value is set in the SETTINGS menu)
unsigned long HIGHEST_FREQ; // absolute maximum frequency (Hz) (value is set in the SETTINGS menu)

unsigned long frequency; // the 'dial' frequency as shown on the display
int fine = 0; // fine tune offset (Hz)

/**
The raduino has multiple RUN-modes:
*/
#define RUN_NORMAL (0) // normal operation
#define RUN_CALIBRATE (1) // calibrate VFO frequency in LSB mode
#define RUN_DRIVELEVEL (2) // set VFO drive level
#define RUN_TUNERANGE (3) // set the range of the tuning pot
#define RUN_CWOFFSET (4) // set the CW offset (=sidetone pitch)
#define RUN_SCAN (5) // frequency scanning mode
#define RUN_SCAN_PARAMS (6) // set scan parameters
#define RUN_MONITOR (7) // frequency scanning mode
#define RUN_FINETUNING (8) // fine tuning mode

byte RUNmode = RUN_NORMAL;

// ************* SI5315 routines - tks Jerry Gaffke, KE7ER **********************

// An minimalist standalone set of Si5351 routines.
// VCOA is fixed at 875mhz, VCOB not used.
// The output msynth dividers are used to generate 3 independent clocks
```

```
// with 1hz resolution to any frequency between 4khz and 109mhz.

// Usage:
// Call si5351bx_init() once at startup with no args;
// Call si5351bx_setfreq(clknum, freq) each time one of the
// three output CLK pins is to be updated to a new frequency.
// A freq of 0 serves to shut down that output clock.

// The global variable si5351bx_vcoa starts out equal to the nominal VCOA
// frequency of 25mhz*35 = 875000000 Hz. To correct for 25mhz crystal errors,
// the user can adjust this value. The vco frequency will not change but
// the number used for the (a+b/c) output msynth calculations is affected.
// Example: We call for a 5mhz signal, but it measures to be 5.001mhz.
// So the actual vcoa frequency is 875mhz*5.001/5.000 = 875175000 Hz,
// To correct for this error: si5351bx_vcoa=875175000;

// Most users will never need to generate clocks below 500khz.
// But it is possible to do so by loading a value between 0 and 7 into
// the global variable si5351bx_rdiv, be sure to return it to a value of 0
// before setting some other CLK output pin. The affected clock will be
// divided down by a power of two defined by 2**si5351_rdiv
// A value of zero gives a divide factor of 1, a value of 7 divides by 128.
// This lightweight method is a reasonable compromise for a seldom used feature.

void si5351bx_init() { // Call once at power-up, start PLLA
uint8_t reg; uint32_t msxp1;
Wire.begin();
i2cWrite(149, 0); // SpreadSpectrum off
i2cWrite(3, si5351bx_clken); // Disable all CLK output drivers
i2cWrite(183, SI5351BX_XTALPF << 6); // Set 25mhz crystal load capacitance
msxp1 = 128 * SI5351BX_MSA - 512; // and msxp2=0, msxp3=1, not fractional
uint8_t vals[8] = {0, 1, BB2(msxp1), BB1(msxp1), BB0(msxp1), 0, 0, 0};
i2cWriten(26, vals, 8); // Write to 8 PLLA msynth regs
i2cWrite(177, 0x20); // Reset PLLA (0x80 resets PLLB)
// for (reg=16; reg<=23; reg++) i2cWrite(reg, 0x80); // Powerdown CLK's
// i2cWrite(187, 0); // No fannout of clkin, xtal, ms0, ms4
}

void si5351bx_setfreq(uint8_t clknum, uint32_t fout) { // Set a CLK to fout Hz
uint32_t msa, msb, msc, msxp1, msxp2, msxp3p2top;
if ((fout < 500000) || (fout > 109000000)) // If clock freq out of range
```

```
si5351bx_clken |= 1 << clknum; // shut down the clock
else {
msa = si5351bx_vcoa / fout; // Integer part of vco/fout
msb = si5351bx_vcoa % fout; // Fractional part of vco/fout
msc = fout; // Divide by 2 till fits in reg
while (msc & 0xfff00000) {
msb = msb >> 1;
msc = msc >> 1;
}
msxp1 = (128 * msa + 128 * msb / msc - 512) | (((uint32_t)si5351bx_rdiv) << 20);
msxp2 = 128 * msb - 128 * msb / msc * msc; // msxp3 == msc;
msxp3p2top = (((msc & 0x0F0000) << 4) | msxp2); // 2 top nibbles
uint8_t vals[8] = { BB1(msc), BB0(msc), BB2(msxp1), BB1(msxp1),
BB0(msxp1), BB2(msxp3p2top), BB1(msxp2), BB0(msxp2)
};
i2cWriten(42 + (clknum * 8), vals, 8); // Write to 8 msynth regs
i2cWrite(16 + clknum, 0x0C | si5351bx_drive[clknum]); // use local msynth
si5351bx_clken &= ~(1 << clknum); // Clear bit to enable clock
}
i2cWrite(3, si5351bx_clken); // Enable/disable clock
}

void i2cWrite(uint8_t reg, uint8_t val) { // write reg via i2c
Wire.beginTransmission(SI5351BX_ADDR);
Wire.write(reg);
Wire.write(val);
Wire.endTransmission();
}

void i2cWriten(uint8_t reg, uint8_t *vals, uint8_t vcnt) { // write array
Wire.beginTransmission(SI5351BX_ADDR);
Wire.write(reg);
while (vcnt--) Wire.write(*vals++);
Wire.endTransmission();
}

// ********** End of Jerry's si5315bx routines
*******************************************************

/**
Display Routine
```

```
This display routine prints a line of characters to the upper or lower line of the 16x2 display
linenmbr = 0 is the upper line
linenmbr = 1 is the lower line
*/
void printLine(char linenmbr, char *c) {
if (strcmp(c, printBuff[linenmbr])) { // only refresh the display when there was a change
lcd.setCursor(0, linenmbr); // place the cursor at the beginning of the selected line
lcd.print(c);
strcpy(printBuff[linenmbr], c);
for (byte i = strlen(c); i < 16; i++) { // add white spaces until the end of the 16 characters line is reached
lcd.print(' ');
}
}
}
/**
Building upon the previous function,
update Display paints the first line as per current state of the radio
*/
void updateDisplay() {
// tks Jack Purdum W8TEE
// replaced fsprint commmands by str commands for code size reduction
memset(c, 0, sizeof(c));
memset(b, 0, sizeof(b));
if (locked || RUNmode == RUN_FINETUNING) {
ultoa((frequency + fine), b, DEC);
strcpy(c, "");
c[0] = b[0];
}
else {
ultoa((frequency + 50), b, DEC);
if (!vfoActive) // VFO A is active
strcpy(c, "A ");
else
strcpy(c, "B ");
c[2] = b[0];
```

```
}
strcat(c, ".");
strncat(c, &b[1], 3);
strcat(c, ".");
if (locked || RUNmode == RUN_FINETUNING)
strncat(c, &b[4], 3); // show two more digits
else
strncat(c, &b[4], 1);

switch (mode) {
case LSB:
strcat(c, " LSB");
break;
case USB:
strcat(c, " USB");
break;
case CWL:
strcat(c, " CWL");
break;
case CWU:
strcat(c, " CWU");
break;
}

if (inTx)
strcat(c, " TX");
else if (splitOn)
strcat(c, " SP");

printLine(0, c);
}

// function to generate a bleep sound (FB menu)
void bleep(int pitch, int duration, byte repeat) {
for (byte i = 0; i < repeat; i++) {
tone(CW_TONE, pitch);
delay(duration);
noTone(CW_TONE);
delay(duration);
}
```

```
}

bool calbutton = false;

/**
To use calibration sets the accurate readout of the tuned frequency
To calibrate, follow these steps:
1. Tune in a LSB 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. Use the "LSB calibrate" option in the "Settings" menu (or Press the CAL_BUTTON line to the ground (pin A2 - red wire))
4. tune in the signal until it sounds proper.
5. Press the FButton (or Release CAL_BUTTON)
In step 4, when we say 'sounds proper' then, for a CW signal/carrier it means zero-beat
and for LSB it is the most natural sounding setting.

Calibration is an offset value that is added to the VFO frequency.
We store it in the EEPROM and read it in setup() when the Radiuno is powered up.

Then select the "USB calibrate" option in the "Settings" menu and repeat the same steps for USB mode.
*/

int shift, current_setting;
void calibrate() {
int knob = analogRead(ANALOG_TUNING); // get the current tuning knob position

if (RUNmode != RUN_CALIBRATE) {

if (mode == USB)
current_setting = USB_OFFSET;
else
current_setting = cal;

shift = current_setting - knob + 500;
}

// The tuning knob gives readings from 0 to 1000
// Each step is taken as 1 Hz and the mid setting of the knob is taken as zero

if (mode == USB) {
USB_OFFSET = constrain(knob - 500 + shift, -10000, 10000);

if (knob < 5 && USB_OFFSET > -10000)
shift = shift - 10;
```

```
else if (knob > 1020 && USB_OFFSET < 10000)
shift = shift + 10;
}
else {
cal = constrain(knob - 500 + shift, -10000, 10000);
if (knob < 5 && cal > -10000)
shift = shift - 10;
else if (knob > 1020 && cal < 10000)
shift = shift + 10;
}
// if Fbutton is pressed again (or when the CAL button is released), we save the setting
if (!digitalRead(FBUTTON) || (calbutton && digitalRead(CAL_BUTTON))) {
RUNmode = RUN_NORMAL;
if (mode == USB) {
printLine(1, (char *)"USB Calibrated!");
//Write the 2 bytes of the USB offset into the eeprom memory.
EEPROM.put(4, USB_OFFSET);
}
else {
printLine(1, (char *)"LSB Calibrated!");
//Write the 2 bytes of the LSB offset into the eeprom memory.
EEPROM.put(2, cal);
}
delay(700);
bleep(600, 50, 2);
printLine(1, (char *)"--- SETTINGS ---");
shiftBase(); //align the current knob position with the current frequency
}
else {
// while offset adjustment is in progress, keep tweaking the
// frequency as read out by the knob, display the change in the second line
RUNmode = RUN_CALIBRATE;
if (mode == USB) {
si5351bx_setfreq(2, (bfo_freq + frequency + cal / 5 * 19 - USB_OFFSET));
itoa(USB_OFFSET, b, DEC);
}
```

```
else {
si5351bx_setfreq(2, (bfo_freq - frequency + cal));
itoa(cal, b, DEC);
}
strcpy(c, "offset ");
strcat(c, b);
strcat(c, " Hz");
printLine(1, c);
}
}
/**
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.
In addition we add some offset to USB mode so that the dial frequency is correct in both LSB and
USB mode.
The amount of offset can be set in the SETTING menu as part of the calibration procedure.
Furthermore we add/substract the sidetone frequency only when we receive CW, to assure zero beat
between the transmitting and receiving station (RXshift)
The desired sidetone frequency can be set in the SETTINGS menu.
*/
void setFrequency(unsigned long f) {
if (mode & 1) // if we are in UPPER side band mode
```

```
si5351bx_setfreq(2, (bfo_freq + f + cal * 19 / 5 - USB_OFFSET - RXshift + RIT + fine));
else // if we are in LOWER side band mode
si5351bx_setfreq(2, (bfo_freq - f + cal - RXshift - RIT - fine));
updateDisplay();
}

/**
The checkTX toggles the T/R line. If you would like to make use of RIT, etc,
you must connect pin A0 (black wire) via a 10K resistor to the output of U3
This is a voltage regulator LM7805 which goes on during TX. We use the +5V output
as a PTT sense line (to tell the Raduino that we are in TX).
*/

void checkTX() {
// We don't check for ptt when transmitting cw in semi-QSK mode
// as long as the TimeOut is non-zero, we will continue to hold the
// radio in transmit mode
if (TimeOut > 0 && semiQSK)
return;

if (digitalRead(PTT_SENSE) && !inTx) {
// go in transmit mode
inTx = true;
RXshift = RIT = RIT_old = 0; // no frequency offset during TX

if (semiQSK) {
mode = mode & B11111101; // leave CW mode, return to SSB mode

if (!vfoActive) { // if VFO A is active
mode_A = mode;
EEPROM.put(24, mode_A);
}
else { // if VFO B is active
mode_B = mode;
EEPROM.put(25, mode_B);
}
}

setFrequency(frequency);
shiftBase();
updateDisplay();

if (splitOn) { // when SPLIT is on, swap the VFOs
```

```
swapVFOs();
}
}
if (!digitalRead(PTT_SENSE) && inTx) {
//go in receive mode
inTx = false;
updateDisplay();
if (splitOn) { // when SPLIT was on, swap the VFOs back to original state
swapVFOs();
}
if (mode & 2) { // if we are in CW mode
RXshift = CW_OFFSET; // apply the frequency offset in RX
setFrequency(frequency);
shiftBase();
}
}
}
/* CW is generated by unbalancing the mixer when the key is down.
During key down, the output CW_CARRIER is HIGH (+5V).
This output is connected via a 10K resistor to the mixer input. The mixer will
become unbalanced when CW_CARRIER is HIGH, so a carrier will be transmitted.
During key up, the output CW_CARRIER is LOW (0V). The mixer will remain balanced
and the carrrier will be suppressed.
The radio will go into CW mode automatically as soon as the key goes down, and
return to normal LSB/USB mode when the key has been up for some time.
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 maybe in transmit mode (inTX true)
and yet between dots and dashes and hence keyDown could be true or false
TimeOut: Figures out how long to wait between dots and dashes before putting
the radio back in receive mode
When we transmit CW, we need to apply some offset (800Hz) to the TX frequency, in
order to keep zero-beat between the transmitting and receiving station. The shift
depends on whether upper or lower sideband CW is used:
In CW-U (USB) mode we must shift the TX frequency 800Hz up
In CW-L (LSB) mode we must shift the TX frequency 800Hz down
```

```
The default offset (CW_OFFSET) is 800Hz, the default timeout (QSK_DELAY) is 350ms.
The user can change these in the SETTINGS menu.
*/
void checkCW() {
if (!keyDown && ((cap_sens == 0 && !digitalRead(KEY)) || (cap_sens != 0 && capaKEY) || (key_type
> 0 && ((cap_sens == 0 && !digitalRead(DAH)) || (cap_sens != 0 && capaDAH))))) {
keyDown = true;
if (key_type > 0) { // if we are using the keyer
keyeron = true;
released = 0;
// put the paddles pointers in the correct position
if (key_type & 1) {
// paddle not reversed
paddleDAH = &_dah;
paddleDIT = &_key;
}
else {
// paddle reversed
paddleDAH = &_key;
paddleDIT = &_dah;
}
if ((cap_sens == 0 && !digitalRead(*paddleDIT)) || (cap_sens != 0 && capaKEY))
dit = millis();
if ((cap_sens == 0 && !digitalRead(*paddleDAH)) || (cap_sens != 0 && capaDAH))
dah = millis();
}
if (!inTx && semiQSK) { // switch to transmit mode if we are not already in it
digitalWrite(TX_RX, 1); // activate the PTT switch - go in transmit mode
delay(5); // give the relays a few ms to settle the T/R relays
inTx = true;
if (splitOn) // when SPLIT is on, swap the VFOs first
swapVFOs();
mode = mode | 2; // go into to CW mode
if (!vfoActive) { // if VFO A is active
mode_A = mode;
```

```
EEPROM.put(24, mode_A);
}
else { // if VFO B is active
mode_B = mode;
EEPROM.put(25, mode_B);
}
RXshift = RIT = RIT_old = 0; // no frequency offset during TX
setFrequency(frequency);
shiftBase();
}
}
//keep resetting the timer as long as the key is down
if (keyDown)
TimeOut = millis() + QSK_DELAY;
//if the key goes up again after it's been down
if (keyDown && ((cap_sens == 0 && digitalRead(KEY)) || (cap_sens != 0 && !capaKEY))) {
keyDown = false;
TimeOut = millis() + QSK_DELAY;
}
// if we are in semi-QSK mode and have a keyup for a "longish" time (QSK_DELAY value in ms)
// then go back to RX
if (TimeOut > 0 && inTx && TimeOut < millis() && semiQSK) {
inTx = false;
TimeOut = 0; // reset the CW timeout counter
RXshift = CW_OFFSET; // apply the frequency offset in RX
setFrequency(frequency);
shiftBase();
if (splitOn) // then swap the VFOs back when SPLIT was on
swapVFOs();
digitalWrite(TX_RX, 0); // release the PTT switch - move the radio back to receive
delay(10); //give the relays a few ms to settle the T/R relays
}
if (key_type == 0 && keyDown && mode & B00000010) {
digitalWrite(CW_CARRIER, 1); // generate carrier
tone(CW_TONE, CW_OFFSET); // generate sidetone
```

```
}
else if (key_type == 0 && digitalRead(SPOT) == HIGH) {
digitalWrite(CW_CARRIER, 0); // stop generating the carrier
noTone(CW_TONE); // stop generating the sidetone
}
}
void keyer() {
static bool FBpressed = false;
static bool SPOTpressed = false;
if (!digitalRead(FBUTTON)) // Press and release F-Button to increase keyer speed
FBpressed = true;
if (FBpressed && digitalRead(FBUTTON) && wpm < 50) {
FBpressed = false;
wpm++;
EEPROM.put(1, wpm);
}
if (!digitalRead(SPOT)) // Press and release SPOT button to reduce keyer speed
SPOTpressed = true;
if (SPOTpressed && digitalRead(SPOT) && wpm > 1) {
SPOTpressed = false;
wpm--;
EEPROM.put(1, wpm);
}
if (key_type > 2) { // bug mode
if ((cap_sens == 0 && !digitalRead(*paddleDAH)) || (cap_sens != 0 && capaDAH))
dah = millis();
else
dah = 0;
}
unsigned long element = 1200UL / wpm;
if (space == 0 && (millis() - dit < element || millis() - dah < 3 * element)) {
digitalWrite(CW_CARRIER, 1); // generate carrier
tone(CW_TONE, CW_OFFSET); // generate sidetone
keyDown = true;
}
else {
```

```
digitalWrite(CW_CARRIER, 0); // stop generating the carrier
noTone(CW_TONE); // stop generating the sidetone
if (space == 0) {
space = millis();
}
if (millis() - space > gap * element) {
if (dit < dah) {
if (ditlatch || (cap_sens == 0 && !digitalRead(*paddleDIT)) || (cap_sens != 0 && capaKEY)) {
dit = millis();
keyeron = true;
ditlatch = false;
keyDown = true;
gap = 1; //standard gap between elements
space = 0;
released = 0;
}
else {
if (dahlatch || (cap_sens == 0 && !digitalRead(*paddleDAH)) || (cap_sens != 0 && capaDAH)) {
dah = millis();
keyeron = true;
dahlatch = false;
keyDown = true;
gap = 1; //standard gap between elements
space = 0;
released = 0;
}
else {
if (autospace)
gap = 3; // autospace - character gap is 3 elements
keyeron = true;
keyDown = true;
if (millis() - space > gap * element) {
keyeron = false;
keyDown = false;
gap = 1; //standard gap between elements
space = 0;
released = 0;
```

```
}
}
}
}
else {
if (dahlatch || (cap_sens == 0 && !digitalRead(*paddleDAH)) || (cap_sens != 0 && capaDAH)) {
dah = millis();
keyeron = true;
dahlatch = false;
keyDown = true;
gap = 1; //standard gap between elements
space = 0;
released = 0;
}
else {
if (ditlatch || (cap_sens == 0 && !digitalRead(*paddleDIT)) || (cap_sens != 0 && capaKEY)) {
dit = millis();
keyeron = true;
ditlatch = false;
keyDown = true;
gap = 1; //standard gap between elements
space = 0;
released = 0;
}
else {
if (autospace)
gap = 3; // autospace - character gap is 3 elements
keyeron = true;
keyDown = true;
if (millis() - space > gap * element) {
keyeron = false;
keyDown = false;
gap = 1; //standard gap between elements
space = 0;
released = 0;
}
}
}
```

```
}
}
}
if (released == 0) {
if (space == 0 && millis() - dit < element && ((cap_sens == 0 && digitalRead(*paddleDIT)) ||
(cap_sens != 0 && !capaKEY)))
released = millis();
if (space == 0 && millis() - dah < 3 * element && ((cap_sens == 0 && digitalRead(*paddleDAH)) ||
(cap_sens != 0 && !capaDAH)))
released = millis();
if (space > 0 && ((cap_sens == 0 && digitalRead(*paddleDIT)) || (cap_sens != 0 && !capaKEY)) &&
((cap_sens == 0 && digitalRead(*paddleDAH)) || (cap_sens != 0 && !capaDAH)))
released = millis();
}
if (cap_sens == 0) { // if standard paddle is used
if (released > 0 && millis() - released > DIT_DELAY && !digitalRead(*paddleDIT)) { // DIT_DELAY
optimized timing characteristics - tks Hidehiko, JA9MAT
ditlatch = true;
dahlatch = false;
}
else if (space > 0 && released > 0 && millis() - released > DAH_DELAY && !
digitalRead(*paddleDAH)) { // DAH_DELAY optimized timing characteristics - tks Hidehiko, JA9MAT
dahlatch = true;
ditlatch = false;
}
}
else { // if touch keyer is used
if (released > 0 && capaKEY) {
ditlatch = true;
dahlatch = false;
}
else if (released > 0 && capaDAH) {
dahlatch = true;
ditlatch = false;
}
}
if (keyeron) {
itoa(wpm, b, DEC);
```

```
strcpy(c, "CW-speed ");
strcat(c, b);
strcat(c, " WPM");
printLine(1, c);
}
else if (locked)
printLine(1, (char *)"dial is locked");
else if (clicks >= 10)
printLine(1, (char *)"--- SETTINGS ---");
else {
RIT_old = 0;
printLine(1, (char *)" ");
}
}

/**
The Function Button is used for several functions
NORMAL menu (normal operation):
1 short press: swap VFO A/B
2 short presses: toggle RIT on/off
3 short presses: toggle SPLIT on/off
4 short presses: toggle LSB/USB
5 short presses: start freq scan mode
5 short presses: start A/B monitor mode
long press (>1 Sec): VFO A=B
VERY long press (>3 sec): go to SETTINGS menu
SETTINGS menu:
1 short press: LSB calibration
2 short presses: USB calibration
3 short presses: Set VFO drive level in LSB mode
4 short presses: Set VFO drive level in USB mode
5 short presses: Set tuning range
6 short presses: Set the 3 CW parameters (sidetone pitch, semi-QSK on/off, CW timeout)
7 short presses: Set the 4 scan parameters (lower limit, upper limit, step size, step delay)
long press: exit SETTINGS menu - go back to NORMAL menu
*/
void checkButton() {
```

```
static byte action;
static long t1, t2;
static bool pressed = false;

if (digitalRead(FBUTTON)) {
t2 = millis() - t1; //time elapsed since last button press
if (pressed)
if (clicks < 10 && t2 > 600 && t2 < 3000) { //detect long press to reset the VFO's
bleep(600, 50, 1);
resetVFOs();
delay(700);
clicks = 0;
}

if (t2 > CLICK_DELAY) { // max time between button clicks (ms)
action = clicks;
if (clicks >= 10)
clicks = 10;
else
clicks = 0;
}
pressed = false;
}
else {
delay(10);
if (!digitalRead(FBUTTON)) {
// button was really pressed, not just some noise

if (locked && digitalRead(SPOT)) {
bleep(600, 50, 1);
printLine(1, (char *)"");
delay(500);
locked = false;
shiftBase();
clicks = 0;
pressed = false;
return;
}

else {
```

```
if (!digitalRead(SPOT)) {
bleep(1200, 50, 1);
locked = true;
updateDisplay();
printLine(1, (char *)"dial is locked");
delay(500);
clicks = 0;
pressed = false;
return;
}
}
if (ritOn) {
toggleRIT(); // disable the RIT when it was on and the FB is pressed again
old_knob = knob_position();
bleep(600, 50, 1);
delay(100);
return;
}
if (!pressed) {
pressed = true;
t1 = millis();
bleep(1200, 50, 1);
action = 0;
clicks++;
if (clicks > 17)
clicks = 11;
if (clicks > 6 && clicks < 10)
clicks = 1;
if (!PTTsense_installed) {
if (clicks == 2)
clicks = 4;
if (clicks == 16)
clicks++;
}
switch (clicks) {
//Normal menu options
case 1:
```

```
printLine(1, (char *)"Swap VFOs");
break;
case 2:
printLine(1, (char *)"RIT ON");
break;
case 3:
printLine(1, (char *)"SPLIT ON/OFF");
break;
case 4:
printLine(1, (char *)"Switch mode");
break;
case 5:
printLine(1, (char *)"Start freq scan");
break;
case 6:
printLine(1, (char *)"Monitor VFO A/B");
break;

//SETTINGS menu options
case 11:
printLine(1, (char *)"LSB calibration");
break;
case 12:
printLine(1, (char *)"USB calibration");
break;
case 13:
printLine(1, (char *)"VFO drive - LSB");
break;
case 14:
printLine(1, (char *)"VFO drive - USB");
break;
case 15:
printLine(1, (char *)"Set tuning range");
break;
case 16:
printLine(1, (char *)"Set CW params");
break;
case 17:
```

```
printLine(1, (char *)"Set scan params");
break;
}
}
else if ((millis() - t1) > 600 && (millis() - t1) < 800 && clicks < 10) // long press: reset the VFOs
printLine(1, (char *)"Reset VFOs");
if ((millis() - t1) > 3000 && clicks < 10) { // VERY long press: go to the SETTINGS menu
bleep(1200, 150, 3);
printLine(1, (char *)"--- SETTINGS ---");
clicks = 10;
}
else if ((millis() - t1) > 1500 && clicks > 10) { // long press: return to the NORMAL menu
bleep(1200, 150, 3);
clicks = -1;
pressed = false;
printLine(1, (char *)" --- NORMAL ---");
delay(700);
}
}
}
if (action != 0 && action != 10) {
bleep(600, 50, 1);
}
switch (action) {
// NORMAL menu
case 1: // swap the VFOs
swapVFOs();
EEPROM.put(26, vfoActive);
delay(700);
break;
case 2: // toggle the RIT on/off
toggleRIT();
delay(700);
break;
case 3: // toggle SPLIT on/off
```

```
toggleSPLIT();
delay(700);
break;

case 4: // toggle the mode LSB/USB
toggleMode();
delay(700);
break;

case 5: // start scan mode
RUNmode = RUN_SCAN;
TimeOut = millis() + scan_step_delay;
frequency = scan_start_freq * 1000L;
printLine(1, (char *)"freq scanning");
break;

case 6: // Monitor mode
RUNmode = RUN_MONITOR;
TimeOut = millis() + scan_step_delay;
printLine(1, (char *)"A/B monitoring");
break;

// SETTINGS MENU

case 11: // calibrate the dial frequency in LSB
RXshift = 0;
mode = LSB;
setFrequency(frequency);
SetSideBand(LSBdrive);
calibrate();
break;

case 12: // calibrate the dial frequency in USB
RXshift = 0;
mode = USB;
setFrequency(frequency);
SetSideBand(USBdrive);
calibrate();
break;

case 13: // set the VFO drive level in LSB
mode = LSB;
SetSideBand(LSBdrive);
```

```
VFOdrive();
break;
case 14: // set the VFO drive level in USB
mode = USB;
SetSideBand(USBdrive);
VFOdrive();
break;
case 15: // set the tuning pot range
param = 1;
set_tune_range();
break;
case 16: // set CW parameters (sidetone pitch, QSKdelay, key type, capacitive touch key, auto space)
param = 1;
set_CWparams();
break;
case 17: // set the 4 scan parameters
param = 1;
scan_params();
break;
}
}
void swapVFOs() {
if (vfoActive) { // if VFO B is active
vfoActive = false; // switch to VFO A
vfoB = frequency;
frequency = vfoA;
mode = mode_A;
}
else { //if VFO A is active
vfoActive = true; // switch to VFO B
vfoA = frequency;
frequency = vfoB;
mode = mode_B;
}
if (mode & 1) // if we are in UPPER side band mode
```

```
SetSideBand(USBdrive);
else // if we are in LOWER side band mode
SetSideBand(LSBdrive);

if (!inTx && mode > 1)
RXshift = CW_OFFSET; // add RX shift when we are receiving in CW mode
else
RXshift = 0; // no RX shift when we are receiving in SSB mode

shiftBase(); //align the current knob position with the current frequency
}

void toggleRIT() {

ritOn = !ritOn; // toggle RIT
if (!ritOn)
RIT = RIT_old = 0;
shiftBase(); //align the current knob position with the current frequency
firstrun = true;
if (splitOn) {
splitOn = false;
EEPROM.put(27, 0);
}
updateDisplay();
}

void toggleSPLIT() {

splitOn = !splitOn; // toggle SPLIT
EEPROM.put(27, splitOn);
if (ritOn) {
ritOn = false;
RIT = RIT_old = 0;
shiftBase();
}
updateDisplay();
}

void toggleMode() {
if (PTTsense_installed && !semiQSK)
mode = (mode + 1) & 3; // rotate through LSB-USB-CWL-CWU
else
mode = (mode + 1) & 1; // switch between LSB and USB only (no CW)
```

```
if (mode & 2) // if we are in CW mode
RXshift = CW_OFFSET;
else // if we are in SSB mode
RXshift = 0;

if (mode & 1) // if we are in UPPER side band mode
SetSideBand(USBdrive);
else // if we are in LOWER side band mode
SetSideBand(LSBdrive);
}

void SetSideBand(byte drivelevel) {

set_drive_level(drivelevel);
setFrequency(frequency);
if (!vfoActive) { // if VFO A is active
mode_A = mode;
EEPROM.put(24, mode_A);
}
else { // if VFO B is active
mode_B = mode;
EEPROM.put(25, mode_B);
}
}

// resetting the VFO's will set both VFO's to the current frequency and mode
void resetVFOs() {
printLine(1, (char *)"VFO A=B !");
vfoA = vfoB = frequency;
mode_A = mode_B = mode;
updateDisplay();
bleep(600, 50, 1);
EEPROM.put(24, mode_A);
EEPROM.put(25, mode_B);
}

void VFOdrive() {
static byte drive;
int knob = analogRead(ANALOG_TUNING); // get the current tuning knob position

if (RUNmode != RUN_DRIVELEVEL) {
```

```
if (mode & 1) // if UPPER side band mode
current_setting = USBdrive / 2 - 1;
else // if LOWER side band mode
current_setting = LSBdrive / 2 - 1;

shift = knob;
}

//generate drive level values 2,4,6,8 from tuning pot
drive = 2 * ((((knob - shift) / 50 + current_setting) & 3) + 1);

// if Fbutton is pressed again, we save the setting

if (!digitalRead(FBUTTON)) {
RUNmode = RUN_NORMAL;
printLine(1, (char *)"Drive level set!");

if (mode & 1) { // if UPPER side band mode
USBdrive = drive;
//Write the 2 bytes of the USBdrive level into the eeprom memory.
EEPROM.put(7, drive);
}
else { // if LOWER side band mode
LSBdrive = drive;
//Write the 2 bytes of the LSBdrive level into the eeprom memory.
EEPROM.put(6, drive);
}
delay(700);
bleep(600, 50, 2);
printLine(1, (char *)"--- SETTINGS ---");
shiftBase(); //align the current knob position with the current frequency
}
else {
// while the drive level adjustment is in progress, keep tweaking the
// drive level as read out by the knob and display it in the second line
RUNmode = RUN_DRIVELEVEL;
set_drive_level(drive);

itoa(drive, b, DEC);
strcpy(c, "drive level ");
strcat(c, b);
strcat(c, "mA");
```

```
printLine(1, c);
}
}
/* this function allows the user to set the tuning range depending on the type of potentiometer
for a standard 1-turn pot, a span of 50 kHz is recommended
for a 10-turn pot, a span of 200 kHz is recommended
*/
void set_tune_range() {
int knob = analogRead(ANALOG_TUNING); // get the current tuning knob position
if (firstrun) {
switch (param) {
case 1:
current_setting = LOWEST_FREQ / 1000;
break;
case 2:
current_setting = HIGHEST_FREQ / 1000;
break;
case 3:
current_setting = POT_SPAN;
break;
}
shift = current_setting - 10 * knob / 20;
}
switch (param) {
case 1:
//generate values 7000-7500 from the tuning pot
LOWEST_FREQ = constrain(10 * knob / 20 + shift, 1000UL, 30000UL);
if (knob < 5 && LOWEST_FREQ > 1000UL)
shift = shift - 10;
else if (knob > 1020 && LOWEST_FREQ < 30000UL)
shift = shift + 10;
break;
case 2:
//generate values 7000-7500 from the tuning pot
HIGHEST_FREQ = constrain(10 * knob / 20 + shift, (LOWEST_FREQ / 1000 + POT_SPAN), 30000UL);
if (knob < 5 && HIGHEST_FREQ > (LOWEST_FREQ / 1000 + POT_SPAN))
shift = shift - 10;
```

```
else if (knob > 1020 && HIGHEST_FREQ < 30000UL)
shift = shift + 10;
break;
case 3:
//generate values 10-500 from the tuning pot
POT_SPAN = constrain(10 * knob / 20 + shift, 10, 500);
if (knob < 5 && POT_SPAN > 10)
shift = shift - 10;
else if (knob > 1020 && POT_SPAN < 500)
shift = shift + 10;
break;
}
// if Fbutton is pressed again, we save the setting
if (!digitalRead(FBUTTON)) {
switch (param) {
case 1:
LOWEST_FREQ = LOWEST_FREQ * 1000UL;
//Write the 2 bytes of the minimum frequency into the eeprom memory.
EEPROM.put(39, LOWEST_FREQ);
bleep(600, 50, 1);
delay(200);
break;
case 2:
HIGHEST_FREQ = HIGHEST_FREQ * 1000UL;
//Write the 2 bytes of the maximum frequency into the eeprom memory.
EEPROM.put(43, HIGHEST_FREQ);
bleep(600, 50, 1);
delay(200);
break;
case 3:
//Write the 2 bytes of the pot span into the eeprom memory.
EEPROM.put(10, POT_SPAN);
printLine(1, (char *)"Tune range set!");
RUNmode = RUN_NORMAL;
bleep(600, 50, 2);
delay(700);
printLine(1, (char *)"--- SETTINGS ---");
shiftBase(); //align the current knob position with the current frequency
```

```
break;
}
param ++;
firstrun = true;
}
else {
RUNmode = RUN_TUNERANGE;
firstrun = false;
switch (param) {
case 1:
ultoa(LOWEST_FREQ, b, DEC);
strcpy(c, "min ");
strcat(c, b);
strcat(c, " kHz");
printLine(1, c);
break;
case 2:
ultoa(HIGHEST_FREQ, b, DEC);
strcpy(c, "max ");
strcat(c, b);
strcat(c, " kHz");
printLine(1, c);
break;
case 3:
itoa(POT_SPAN, b, DEC);
strcpy(c, "pot span ");
strcat(c, b);
strcat(c, " kHz");
printLine(1, c);
break;
}
}
}
/* this function allows the user to set the two CW parameters: CW-OFFSET (sidetone pitch) and CW-TIMEOUT.
*/
void set_CWparams() {
```

```
int knob = analogRead(ANALOG_TUNING); // get the current tuning knob position
if (firstrun) {
switch (param) {
case 1:
mode = mode | 2; //switch to CW mode
updateDisplay();
RXshift = CW_OFFSET;
QSK_DELAY = 10; // during CW offset adjustment, temporarily set QSK_DELAY to minimum
current_setting = CW_OFFSET;
shift = current_setting - knob - 200;
break;
case 2:
current_setting = key_type;
shift = knob;
break;
case 3:
current_setting = cap_sens;
shift = knob;
break;
case 4:
current_setting = autospace;
shift = knob;
break;
case 5:
current_setting = semiQSK;
shift = knob;
break;
case 6:
current_setting = QSK_DELAY;
shift = current_setting - 10 * (knob / 10);
break;
}
}
switch (param) {
case 1:
//generate values 500-1000 from the tuning pot
CW_OFFSET = constrain(knob + 200 + shift, 200, 1200);
```

```
if (knob < 5 && CW_OFFSET > 200)
shift = shift - 10;
else if (knob > 1020 && CW_OFFSET < 1200)
shift = shift + 10;
break;
case 2:
//generate values 0-1-2-3-4 from the tuning pot
key_type = ((((knob - shift) + 4 + current_setting * 26) & 127) / 26);
break;
case 3:
//generate values 0-25 from the tuning pot
cap_sens = constrain((knob - shift) / 40 + current_setting, 0, 25);
if (knob < 5 && cap_sens > 0)
shift = shift + 10;
else if (knob > 1020 && cap_sens < 25)
shift = shift - 10;

//configure the morse keyer inputs
if (cap_sens == 0) { // enable the internal pull-ups if capacitive touch keys are NOT used
pinMode(KEY, INPUT_PULLUP);
pinMode(DAH, INPUT_PULLUP);
}
else { // disable the internal pull-ups if capacitive touch keys are used
pinMode(KEY, INPUT);
pinMode(DAH, INPUT);
}
break;
case 4:
//generate values 0-1-0-1 from the tuning pot
autospace = ((((knob - shift + 4) & 64) / 64) + current_setting) & 1;
break;
case 5:
//generate values 0-1-0-1 from the tuning pot
semiQSK = ((((knob - shift + 4) & 64) / 64) + current_setting) & 1;
break;
case 6:
//generate values 10-1000 from the tuning pot
QSK_DELAY = constrain(10 * (knob / 10) + shift, 10, 1000);
```

```
if (knob < 5 && QSK_DELAY >= 20)
shift = shift - 10;
else if (knob > 1020 && QSK_DELAY < 1000)
shift = shift + 10;
break;
}
// if Fbutton is pressed again, we save the setting
if (!digitalRead(FBUTTON)) {
switch (param) {
case 1:
EEPROM.get(36, QSK_DELAY); // restore QSK_DELAY to original value
//Write the 2 bytes of the QSK_DELAY into the eeprom memory.
EEPROM.put(12, CW_OFFSET);
bleep(600, 50, 1);
delay(200);
break;
case 2:
EEPROM.put(38, key_type);
//Write 1 byte of the key-type into the eeprom memory.
bleep(600, 50, 1);
delay(200);
if (!CapTouch_installed) {
param++;
if (key_type == 0)
param++;
}
break;
case 3:
calibrate_touch_pads(); // measure the base capacitance of the touch pads while they're not being
touched
EEPROM.put(48, cap_sens);
//Write 1 byte of cap_sens into the eeprom memory.
bleep(600, 50, 1);
delay(200);
if (key_type == 0)
param++;
break;
```

```
case 4:
EEPROM.put(47, autospace);
//Write 1 byte of autospace into the eeprom memory.
bleep(600, 50, 1);
delay(200);
break;
case 5:
EEPROM.put(8, semiQSK);
//Write 1 byte of semiQSK into the eeprom memory.
if (semiQSK) {
bleep(600, 50, 1);
delay(200);
}
else {
QSK_DELAY = 10; // set CW timeout to minimum when manual PTT is selected
EEPROM.put(36, QSK_DELAY);
//Write 2 bytes of the QSK_DELAY into the eeprom memory
printLine(1, (char *)"CW params set!");
RUNmode = RUN_NORMAL;
delay(700);
bleep(600, 50, 2);
printLine(1, (char *)"--- SETTINGS ---");
shiftBase(); //align the current knob position with the current frequency
}
break;
case 6:
//Write the 2 bytes of the CW Timout into the eeprom memory.
EEPROM.put(36, QSK_DELAY);
printLine(1, (char *)"CW params set!");
RUNmode = RUN_NORMAL;
delay(700);
bleep(600, 50, 2);
printLine(1, (char *)"--- SETTINGS ---");
shiftBase(); //align the current knob position with the current frequency
break;
}
param ++;
firstrun = true;
```

```
}
else {
RUNmode = RUN_CWOFFSET;
firstrun = false;
switch (param) {
case 1:
itoa(CW_OFFSET, b, DEC);
strcpy(c, "sidetone ");
strcat(c, b);
strcat(c, " Hz");
break;
case 2:
strcpy(c, "Key: ");
switch (key_type) {
case 0:
strcat(c, "straight");
break;
case 1:
strcat(c, "paddle");
break;
case 2:
strcat(c, "rev. paddle");
break;
case 3:
strcat(c, "bug");
break;
case 4:
strcat(c, "rev. bug");
break;
}
break;
case 3:
if (cap_sens == 0)
strcpy(c, "touch keyer OFF");
else {
printLine(0, (char *)"Touch sensor");
itoa((cap_sens), b, DEC);
```

```
strcpy(c, "sensitivity ");
strcat(c, b);
}
break;
case 4:
strcpy(c, "Auto-space: ");
if (autospace)
strcat(c, "ON");
else
strcat(c, "OFF");
break;
case 5:
strcpy(c, "Semi-QSK: ");
if (semiQSK)
strcat(c, "ON");
else
strcat(c, "OFF");
break;
case 6:
itoa(QSK_DELAY, b, DEC);
strcpy(c, "QSK delay ");
strcat(c, b);
strcat(c, "ms");
break;
}
printLine(1, c);
}
}

/* this function allows the user to set the 4 scan parameters: lower limit, upper limit, step size and step delay
*/

void scan_params() {

int knob = analogRead(ANALOG_TUNING); // get the current tuning knob position
if (firstrun) {
switch (param) {

case 1: // set the lower scan limit
```

```
current_setting = scan_start_freq;
shift = current_setting - knob / 2 - LOWEST_FREQ / 1000;
break;
case 2: // set the upper scan limit
current_setting = scan_stop_freq;
shift = current_setting - map(knob, 0, 1024, scan_start_freq, HIGHEST_FREQ / 1000);
break;
case 3: // set the scan step size
current_setting = scan_step_freq;
shift = current_setting - 50 * (knob / 5);
break;
case 4: // set the scan step delay
current_setting = scan_step_delay;
shift = current_setting - 50 * (knob / 25);
break;
}
}
switch (param) {
case 1: // set the lower scan limit
//generate values 7000-7500 from the tuning pot
scan_start_freq = constrain(knob / 2 + 7000 + shift, LOWEST_FREQ / 1000, HIGHEST_FREQ / 1000);
if (knob < 5 && scan_start_freq > LOWEST_FREQ / 1000)
shift = shift - 1;
else if (knob > 1020 && scan_start_freq < HIGHEST_FREQ / 1000)
shift = shift + 1;
break;
case 2: // set the upper scan limit
//generate values 7000-7500 from the tuning pot
scan_stop_freq = constrain(map(knob, 0, 1024, scan_start_freq, HIGHEST_FREQ / 1000) + shift,
scan_start_freq, HIGHEST_FREQ / 1000);
if (knob < 5 && scan_stop_freq > scan_start_freq)
shift = shift - 1;
else if (knob > 1020 && scan_stop_freq < HIGHEST_FREQ / 1000)
shift = shift + 1;
break;
```

```
case 3: // set the scan step size

//generate values 50-10000 from the tuning pot
scan_step_freq = constrain(50 * (knob / 5) + shift, 50, 10000);
if (knob < 5 && scan_step_freq > 50)
shift = shift - 50;
else if (knob > 1020 && scan_step_freq < 10000)
shift = shift + 50;
break;

case 4: // set the scan step delay

//generate values 0-2500 from the tuning pot
scan_step_delay = constrain(50 * (knob / 25) + shift, 0, 2000);
if (knob < 5 && scan_step_delay > 0)
shift = shift - 50;
else if (knob > 1020 && scan_step_delay < 2000)
shift = shift + 50;
break;
}

// if Fbutton is pressed, we save the setting

if (!digitalRead(FBUTTON)) {
switch (param) {

case 1: // save the lower scan limit

//Write the 2 bytes of the start freq into the eeprom memory.
EEPROM.put(28, scan_start_freq);
bleep(600, 50, 1);
break;

case 2: // save the upper scan limit

//Write the 2 bytes of the stop freq into the eeprom memory.
EEPROM.put(30, scan_stop_freq);
bleep(600, 50, 1);
break;

case 3: // save the scan step size

//Write the 2 bytes of the step size into the eeprom memory.
EEPROM.put(32, scan_step_freq);
bleep(600, 50, 1);
```

```
break;
case 4: // save the scan step delay
//Write the 2 bytes of the step delay into the eeprom memory.
EEPROM.put(34, scan_step_delay);
printLine(1, (char *)"Scan params set!");
RUNmode = RUN_NORMAL;
delay(700);
bleep(600, 50, 2);
printLine(1, (char *)"--- SETTINGS ---");
shiftBase(); //align the current knob position with the current frequency
break;
}
param ++;
firstrun = true;
}
else {
RUNmode = RUN_SCAN_PARAMS;
firstrun = false;
switch (param) {
case 1: // display the lower scan limit
itoa(scan_start_freq, b, DEC);
strcpy(c, "lower ");
strcat(c, b);
strcat(c, " kHz");
break;
case 2: // display the upper scan limit
itoa(scan_stop_freq, b, DEC);
strcpy(c, "upper ");
strcat(c, b);
strcat(c, " kHz");
break;
case 3: // display the scan step size
itoa(scan_step_freq, b, DEC);
strcpy(c, "step ");
strcat(c, b);
```

```
strcat(c, " Hz");
break;

case 4: // display the scan step delay

itoa(scan_step_delay, b, DEC);
strcpy(c, "delay ");
strcat(c, b);
strcat(c, " ms");
break;
}
printLine(1, c);
}
}

// function to read the position of the tuning knob at high precision (Allard, PE1NWL)
int knob_position() {
unsigned long knob = 0;
// the knob value normally ranges from 0 through 1023 (10 bit ADC)
// in order to increase the precision by a factor 10, we need 10^2 = 100x oversampling
for (byte i = 0; i < 100; i++) {
knob = knob + analogRead(ANALOG_TUNING); // take 100 readings from the ADC
}
knob = (knob + 5L) / 10L; // take the average of the 100 readings and multiply the result by 10
//now the knob value ranges from 0 through 10,230 (10x more precision)
knob = knob * 10000L / 10230L; // scale the knob range down to 0-10,000
return (int)knob;
}

/* Many BITX40's suffer from a strong birdie at 7199 kHz (LSB).
This birdie may be eliminated by using a different VFO drive level in LSB mode.
In USB mode, a high drive level may be needed to compensate for the attenuation of
higher VFO frequencies.
The drive level for each mode can be set in the SETTINGS menu
*/

void set_drive_level(byte level) {
si5351bx_drive[2] = level / 2 - 1;
setFrequency(frequency);
}

void doRIT() {
```

```
if (millis() - max(dit, dah) < 1000)
return;
int knob = knob_position(); // get the current tuning knob position

if (firstrun) {
current_setting = RIToffset;
shift = current_setting - ((knob - 5000) / 2);
firstrun = false;
}

//generate values -2500 ~ +2500 from the tuning pot
RIToffset = (knob - 5000) / 2 + shift;
if (knob < 5 && RIToffset > -2500)
shift = shift - 50;
else if (knob > 10220 && RIToffset < 2500)
shift = shift + 50;

RIT = RIToffset;
if (RIT != RIT_old) {
setFrequency(frequency);
itoa(RIToffset, b, DEC);
strcpy(c, "RIT ");
strcat(c, b);
strcat(c, " Hz");
printLine(1, c);

RIT_old = RIT;
old_knob = knob_position();
delay(30);
}
}

/**
Function to align the current knob position with the current frequency
If we switch between VFO's A and B, the frequency will change but the tuning knob
is still in the same position. We need to apply some offset so that the new frequency
corresponds with the current knob position.
This function reads the current knob position, then it shifts the baseTune value up or down
so that the new frequency matches again with the current knob position.
*/

void shiftBase() {
```

```
setFrequency(frequency);
unsigned long knob = knob_position(); // get the current tuning knob position
baseTune = frequency - (knob * (unsigned long)POT_SPAN / 10UL);
}

/**
The Tuning mechansim 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 1023. 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 POT_SPAN value.
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 accross the entire 40m band. The desired POT_SPAN can be set via the Function Button in the SETTINGS menu.
When the potentiometer is moved to either end of the range, the frequency starts automatically moving
up or down in 10 Khz increments, so it is still possible to tune beyond the range set by POT_SPAN.
*/
void doTuning() {
int knob = analogRead(ANALOG_TUNING) * 100000 / 10230; // get the current tuning knob position
// tuning is disabled during TX (only when PTT sense line is installed)
if (inTx && clicks < 10 && abs(knob - old_knob) > 20 && !locked) {
printLine(1, (char *)"dial is locked");
shiftBase();
firstrun = true;
return;
}
else if (inTx || locked)
return;
knob = knob_position(); // get the precise tuning knob position
// the knob is fully on the low end, do fast tune: move down by 10 Khz and wait for 300 msec
// if the POT_SPAN is very small (less than 25 kHz) then use 1 kHz steps instead
```

```
if (knob == 0) {
if (frequency > LOWEST_FREQ) {
if (POT_SPAN < 25)
baseTune = baseTune - 1000UL; // fast tune down in 1 kHz steps
else
baseTune = baseTune - 10000UL; // fast tune down in 10 kHz steps
frequency = baseTune + (unsigned long)knob * (unsigned long)POT_SPAN / 10UL;
if (clicks < 10)
printLine(1, (char *)"<<<<<<<"); // tks Paul KC8WBK
delay(300);
}
if (frequency <= LOWEST_FREQ)
baseTune = frequency = LOWEST_FREQ;
setFrequency(frequency);
old_knob = 0;
}

// the knob is full on the high end, do fast tune: move up by 10 Khz and wait for 300 msec
// if the POT_SPAN is very small (less than 25 kHz) then use 1 kHz steps instead

else if (knob == 10000) {
if (frequency < HIGHEST_FREQ) {
if (POT_SPAN < 25)
baseTune = baseTune + 1000UL; // fast tune up in 1 kHz steps
else
baseTune = baseTune + 10000UL; // fast tune up in 10 kHz steps
frequency = baseTune + (unsigned long)knob * (unsigned long)POT_SPAN / 10UL;
if (clicks < 10)
printLine(1, (char *)" >>>>>>>"); // tks Paul KC8WBK
delay(300);
}
if (frequency >= HIGHEST_FREQ) {
baseTune = HIGHEST_FREQ - (POT_SPAN * 1000UL);
frequency = HIGHEST_FREQ;
}
setFrequency(frequency);
old_knob = 10000;
}

// the tuning knob is at neither extremities, tune the signals as usual
```

```
else {
if (abs(knob - old_knob) > 4) { // improved "flutter fix": only change frequency when the current knob position is more than 4 steps away from the previous position
knob = (knob + old_knob) / 2; // tune to the midpoint between current and previous knob reading
old_knob = knob;
frequency = constrain(baseTune + (unsigned long)knob * (unsigned long)POT_SPAN / 10UL, LOWEST_FREQ, HIGHEST_FREQ);
setFrequency(frequency);
delay(10);
}
}
if (!vfoActive) // VFO A is active
vfoA = frequency;
else
vfoB = frequency;
}
/**
"CW SPOT" function: When operating CW it is important that both stations transmit their carriers on the same frequency.
When the SPOT button is pressed while the radio is in CW mode, the RIT will be turned off and the sidetone will be generated (but no carrier will be transmitted).
The FINETUNE mode is then also enabled (fine tuning +/- 1000Hz). Fine tune the VFO so that the pitch of the received CW signal is equal to the pitch of the CW Spot tone.
By aligning the CW Spot tone to match the pitch of an incoming station's signal, you will cause your signal and the other station's signal to be exactly on the same frequency.
When the SPOT button is pressed while the radio is in SSB mode, the radio will only be put in FINETUNE mode (no sidetone will be generated).
*/
void checkSPOT() {
if (digitalRead(SPOT) == LOW && !inTx) {
RUNmode = RUN_FINETUNING;
TimeOut = millis() - 1000UL;
if (ritOn) {
toggleRIT(); // disable the RIT if it was on
old_knob = knob_position();
}
}
}
```

```
void finetune() {

// for SPOT tuning: in CW mode only, generate short side tone pulses every one second
if ((mode & 2) && millis() - TimeOut > 1000UL) {
tone(CW_TONE, CW_OFFSET);
if (millis() - TimeOut > 1150UL) {
noTone(CW_TONE);
TimeOut = millis();
}
}

int knob = knob_position(); // get the current tuning knob position
static int fine_old;

if (digitalRead(SPOT) == LOW && !inTx && !locked) {
if (firstrun) {
firstrun = false;
fine = 0;
fine_old = 9999;
shift = (5000 - knob) / 5;
}

//generate values -1000 ~ +1000 from the tuning pot
fine = (knob - 5000) / 5 + shift;
if (knob < 10 && fine > -1000 && frequency + fine > LOWEST_FREQ)
shift = shift - 10;
else if (knob > 10220 && fine < 1000 && frequency + fine < HIGHEST_FREQ)
shift = shift + 10;

if (fine != fine_old) {
setFrequency(frequency); // apply the finetuning offset
updateDisplay();
}

if (mode & 2)
printLine(1, (char *)"SPOT + FINE TUNE");
else
printLine(1, (char *)"FINE TUNE");

fine_old = fine;
}

else { // return to normal mode when SPOT button is released
```

```
firstrun = true;
TimeOut = 0;
noTone(CW_TONE);
RUNmode = RUN_NORMAL;
frequency = frequency + fine; // apply the finetuning offset
fine = 0;
setFrequency(frequency);
shiftBase();
old_knob = knob_position();
if (clicks == 10)
printLine(1, (char *)"--- SETTINGS ---");
}
}

byte raduino_version; //version identifier

void factory_settings() {

printLine(0, (char *)"loading standard");
printLine(1, (char *)"settings...");
EEPROM.put(0, raduino_version); //version identifier
EEPROM.put(1, CW_SPEED); //CW keyer speed
EEPROM.put(2, OFFSET_LSB); //LSB offset value
EEPROM.put(4, OFFSET_USB); //USB offset
EEPROM.put(6, VFO_DRIVE_LSB); //VFO drive level in LSB/CWL mode
EEPROM.put(7, VFO_DRIVE_USB); //VFO drive level in USB/CWU mode
EEPROM.put(8, SEMI_QSK); // semi QSK
EEPROM.put(10, TUNING_POT_SPAN); //tuning pot span
EEPROM.put(12, CW_SHIFT); //CW offset / sidetone pitch
EEPROM.put(16, 7125000UL); // initial VFO A frequency (7125 kHz)
EEPROM.put(20, 7125000UL); // initial VFO B frequency (7125 kHz)
EEPROM.put(24, 0); // initial mode VFO A (LSB)
EEPROM.put(25, 0); // initial mode VFO B (LSB)
EEPROM.put(26, false); // initial vfoActive (VFO A)
EEPROM.put(27, false); // SPLIT initially off
EEPROM.put(28, SCAN_START); // scan_start_freq
EEPROM.put(30, SCAN_STOP); // scan_stop_freq
EEPROM.put(32, SCAN_STEP); // scan_step_freq
EEPROM.put(34, SCAN_STEP_DELAY); // scan_step_delay
EEPROM.put(36, CW_TIMEOUT); // CW timout
```

```
EEPROM.put(38, CW_KEY_TYPE); // CW key type
EEPROM.put(39, MIN_FREQ); // absolute minimum frequency
EEPROM.put(43, MAX_FREQ); // absolute maximum frequency
EEPROM.put(47, AUTOSPACE); // CW keyer autospace ON/OFF
EEPROM.put(48, CAP_SENSITIVITY); // Capacitive touch sensor sensitivity (0=OFF)
delay(1000);
}

// save the VFO frequencies when they haven't changed more than 500Hz in the past 30 seconds
// (EEPROM.put writes the data only if it is different from the previous content of the eeprom location)
void save_frequency() {
static long t3;
static unsigned long old_vfoA, old_vfoB;
unsigned long aDif, bDif; // tks Richard Blessing
if (vfoA >= old_vfoA)
{
aDif = vfoA - old_vfoA;
}
else
{
aDif = old_vfoA - vfoA;
}
if (vfoB >= old_vfoB)
{
bDif = vfoB - old_vfoB;
}
else
{
bDif = old_vfoB - vfoB;
}
if ((aDif < 500UL) && (bDif < 500UL)) {
if (millis() - t3 > 30000UL) {
EEPROM.put(16, old_vfoA); // save VFO_A frequency
EEPROM.put(20, old_vfoB); // save VFO_B frequency
t3 = millis();
}
}
```

```
else
t3 = millis();
if (aDif > 500UL)
old_vfoA = vfoA;
if (bDif > 500UL)
old_vfoB = vfoB;
}
void scan() {
int knob = knob_position();
if (abs(knob - old_knob) > 8 || (digitalRead(PTT_SENSE) && PTTsense_installed) || !
digitalRead(FBUTTON) || (cap_sens == 0 && !digitalRead(KEY)) || (cap_sens != 0 && capaKEY) || !
digitalRead(SPOT)) {
//stop scanning
TimeOut = 0; // reset the timeout counter
RUNmode = RUN_NORMAL;
shiftBase();
delay(400);
}
else if (TimeOut < millis()) {
if (RUNmode == RUN_SCAN) {
frequency = frequency + scan_step_freq; // change frequency
// test for upper limit of scan
if (frequency > scan_stop_freq * 1000UL)
frequency = scan_start_freq * 1000UL;
setFrequency(frequency);
}
else // monitor mode
swapVFOs();
TimeOut = millis() + scan_step_delay;
}
}

// Routine to detect whether or not the touch pads are being touched.
// This routine is only executed when the related touch keyer mod is installed.
// We need two 470K resistors connected from the PULSE output to either KEY and DAH inputs.
// We send LOW to the PULSE output and as a result the internal capacitance on the KEY and DAH
// inputs will start to discharge. After a short "threshold" delay, we test whether the
```

```
// KEY and DAH inputs are still HIGH or already LOW.
// When the KEY or DAH input is already LOW, it means the internal base capacitance has rapidly
// discharged, so we can conclude the touch pad wasn't touched.
// When the KEY or DAH input is still HIGH, it means that the pad was touched.
// When the pad is touched, the human body adds extra capacitance to the internal base capacity,
// and he increased capacity will not rapidly discharge within the given threshold delay time.
// Finally we send HIGH to the pulse output allowing the capacitance to recharge for the next cycle.
// We can control the touch sensitivity by varying the threshold delay time.
// The minimum delay time is the base_sens value as determined by the calibration routine which
// is executed automatically during power on.
// With minimum delay time we will have maximum touch sensitivity. If we want to reduce the sensitivity
// we add some extra delay time (cap_sens). The user can set the desired touch sensisitivy value in the
// SETTINGS menu (1-25 µs extra delay).

void touch_key() {
static unsigned long KEYup = 0;
static unsigned long KEYdown = 0;
static unsigned long DAHup = 0;
static unsigned long DAHdown = 0;

digitalWrite(PULSE, LOW); // send LOW to the PULSE output
delayMicroseconds(base_sens_KEY + 25 - cap_sens); // wait few microseconds for the KEY capacitance to discharge
if (digitalRead(KEY)) { // test if KEY input is still HIGH
KEYdown = millis(); // KEY touch pad was touched
if (!capaKEY && millis() - KEYup > 1) {
capaKEY = true;
released = 0;
}
}
else { // KEY touch pad was not touched
KEYup = millis();
if (capaKEY && millis() - KEYdown > 10)
capaKEY = false;
}

digitalWrite(PULSE, HIGH); // send HIGH to the PULSE output
```

```
delayMicroseconds(50); // allow the capacitance to recharge

digitalWrite(PULSE, LOW); // send LOW to the PULSE output

delayMicroseconds(base_sens_DAH + 25 - cap_sens); // wait few microseconds for the DAH
capacitance to discharge

if (digitalRead(DAH)) { // test if DAH input is still HIGH

DAHdown = millis(); // DAH touch pad was touched

if (!capaDAH && millis() - DAHup > 1) {

capaDAH = true;

released = 0;

}

}

else { // DAH touch pad was not touched

DAHup = millis();

if (capaDAH && millis() - DAHdown > 10)

capaDAH = false;

}

digitalWrite(PULSE, HIGH); // send HIGH to the PULSE output

delayMicroseconds(50); // allow the capacitance to recharge

// put the touch sensors in the correct position

if (key_type == 2 || key_type == 4) {

// swap capaKEY and capaDAH if paddle is reversed

capaKEY = capaKEY xor capaDAH;

capaDAH = capaKEY xor capaDAH;

capaKEY = capaKEY xor capaDAH;

}

}

// Routine to calibrate the touch key pads

// (measure the time it takes the capacitance to discharge while the touch pads are NOT touched)

// Even when the touch pads are NOT touched, there is some internal capacitance

// The internal capacitance may vary depending on the length and routing of the wiring

// We measure the base capacitance so that we can use it as a baseline reference

// (threshold delay when the pads are not touched).

void calibrate_touch_pads() {

// disable the internal pullups

pinMode(KEY, INPUT);

pinMode(DAH, INPUT);
```

```
bool triggered;
// first we calibrate the KEY (DIT) touch pad
base_sens_KEY = 0; // base capacity of the KEY (DIT) touch pad
do {
base_sens_KEY++; // increment the delay time until the KEY touch pad is no longer triggered by the base capacitance
triggered = false;
for (int i = 0; i < 100; i++) {
digitalWrite(PULSE, HIGH); // bring the KEY input to a digital HIGH level
delayMicroseconds(50); // wait a short wile to allow voltage on the KEY input to stabalize
digitalWrite(PULSE, LOW); // now bring the KEY input to a digital LOW value
delayMicroseconds(base_sens_KEY); // wait few microseconds
if (digitalRead(KEY)) { // check 100 times if KEY input is still HIGH
triggered = true; // if KEY is still high, then it was triggered by the base capacitance
i = 100;
}
}
} while (triggered && base_sens_KEY != 255); // keep trying until KEY input is no longer triggered

// Next we calibrate the DAH pad
base_sens_DAH = 0; // base capacity of the DAH touch pad
do {
base_sens_DAH++; // increment the delay time until the DAH touch pad is no longer triggered by the base capacitance
triggered = false;
for (int i = 0; i < 100; i++) {
digitalWrite(PULSE, HIGH); // bring the KEY input to a digital HIGH level
delayMicroseconds(50); // wait a short wile to allow voltage on the DAH input to stabalize
digitalWrite(PULSE, LOW); // now bring the DAH input to a digital LOW value
delayMicroseconds(base_sens_DAH); // wait few microseconds
if (digitalRead(DAH)) { // check 100 times if DAH input is still HIGH
triggered = true; // if KEY is still high, then it was triggered by the base capacitance
i = 100;
}
}
} while (triggered && base_sens_DAH != 255); // keep trying until KEY input is no longer triggered

if (base_sens_KEY == 255 || base_sens_DAH == 255 || base_sens_KEY == 1 || base_sens_DAH == 1) {
```

```
// if either input is still triggered even with max delay (255 us)
CapTouch_installed = false; // then the base capacitance is too high (or the mod is not installed) so we can't use the touch keyer
cap_sens = 0;
EEPROM.put(48, 0); // turn capacitive touch keyer OFF
printLine(0, (char *)"touch sensors");
printLine(1, (char *)"not detected");
}
else if (cap_sens > 0) {
printLine(0, (char *)"touch key calibr");
strcpy(c, "DIT ");
itoa(base_sens_KEY, b, DEC);
strcat(c, b);
strcat(c, ", DAH ");
itoa(base_sens_DAH, b, DEC);
strcat(c, b);
strcat(c, " us");
printLine(1, c);
}
else
printLine(1, (char *)"touch keyer OFF");

delay(2000);
updateDisplay();

//configure the morse keyer inputs
if (cap_sens == 0) { // enable the internal pull-ups if touch keyer is disabled
pinMode(KEY, INPUT_PULLUP);
pinMode(DAH, INPUT_PULLUP);
}
}

/**
setup is called on boot up
It setups up the modes for various pins as inputs or outputs
initiliaizes 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() {
raduino_version = 25;
strcpy (c, "Raduino v1.25.1");

lcd.begin(16, 2);

// Start serial and initialize the Si5351
Serial.begin(9600);
analogReference(DEFAULT);
//Serial.println("*Raduino booting up");
;
//configure the function button to use the internal pull-up
pinMode(FBUTTON, INPUT_PULLUP);
//configure the CAL button to use the internal pull-up
pinMode(CAL_BUTTON, INPUT_PULLUP);
//configure the SPOT button to use the internal pull-up
pinMode(SPOT, INPUT_PULLUP);

pinMode(TX_RX, OUTPUT);
digitalWrite(TX_RX, 0);
pinMode(CW_CARRIER, OUTPUT);
digitalWrite(CW_CARRIER, 0);
pinMode(CW_TONE, OUTPUT);
digitalWrite(CW_TONE, 0);
pinMode(PULSE, OUTPUT);
digitalWrite(PULSE, 0);

// when Fbutton or CALbutton is kept pressed during power up,
// or after a version update,
// then all settings will be restored to the standard "factory" values
byte old_version;
EEPROM.get(0, old_version); // previous sketch version
if (!digitalRead(CAL_BUTTON) || !digitalRead(FBUTTON) || (old_version != raduino_version)) {
factory_settings();
}

//configure the PTT SENSE to use the internal pull-up
pinMode(PTT_SENSE, INPUT_PULLUP);
// check if PTT sense line is installed
PTTsense_installed = !digitalRead(PTT_SENSE);
pinMode(PTT_SENSE, INPUT); //disable the internal pull-up
```

```
printLine(0, c);
delay(1000);

//retrieve user settings from EEPROM
EEPROM.get(1, wpm);
EEPROM.get(2, cal);
EEPROM.get(4, USB_OFFSET);
EEPROM.get(6, LSBdrive);
EEPROM.get(7, USBdrive);
EEPROM.get(8, semiQSK);
EEPROM.get(10, POT_SPAN);
EEPROM.get(12, CW_OFFSET);
EEPROM.get(16, vfoA);
EEPROM.get(20, vfoB);
EEPROM.get(24, mode_A);
EEPROM.get(25, mode_B);
EEPROM.get(26, vfoActive);
EEPROM.get(27, splitOn);
EEPROM.get(28, scan_start_freq);
EEPROM.get(30, scan_stop_freq);
EEPROM.get(32, scan_step_freq);
EEPROM.get(34, scan_step_delay);
EEPROM.get(36, QSK_DELAY);
EEPROM.get(38, key_type);
EEPROM.get(39, LOWEST_FREQ);
EEPROM.get(43, HIGHEST_FREQ);
EEPROM.get(47, autospace);
EEPROM.get(48, cap_sens);

if (PTTsense_installed) {
calibrate_touch_pads(); // measure the base capacitance of the touch pads while they're not being touched
}

//initialize the SI5351
si5351bx_init();
//Serial.println("*Initialized Si5351\n");
si5351bx_setfreq(2, 4900000L);
//Serial.println("*Si5350 ON\n");

if (!vfoActive) { // VFO A is active
```

```
frequency = vfoA;
mode = mode_A;
}
else {
frequency = vfoB;
mode = mode_B;
}

if (mode & 1) // if UPPER side band
SetSideBand(USBdrive);
else // if LOWER side band
SetSideBand(LSBdrive);

if (mode > 1) // if in CW mode
RXshift = CW_OFFSET;

shiftBase(); //align the current knob position with the current frequency

//display warning message when VFO calibration data was erased
if ((cal == 0) && (USB_OFFSET == 1500)) {
printLine(1, (char *)"VFO uncalibrated");
delay(1000);
}

bleep(CW_OFFSET, 60, 3);
bleep(CW_OFFSET, 180, 1);
}

void loop() {
switch (RUNmode) {
case 0: // for backward compatibility: execute calibration when CAL button is pressed
if (!digitalRead(CAL_BUTTON)) {
RUNmode = RUN_CALIBRATE;
calbutton = true;
factory_settings();
printLine(0, (char *)"Calibrating: Set");
printLine(1, (char *)"to zerobeat");
delay(2000);
}
else {
if (cap_sens != 0)
touch_key();
```

```
if (!inTx && millis() - max(dit, dah) > 1000) {
checkButton();
checkSPOT();
save_frequency();
if (clicks == 0 && !ritOn && !locked)
printLine(1, (char *)"");
}
if (PTTsense_installed) {
checkCW();
checkTX();
if (keyeron)
keyer();
}
if (ritOn && !inTx)
doRIT();
else if (millis() - max(dit, dah) > QSK_DELAY)
doTuning();
}
return;
case 1: //calibration
calibrate();
break;
case 2: //set VFO drive level
VFOdrive();
break;
case 3: // set tuning range
set_tune_range();
break;
case 4: // set CW parameters
set_CWparams();
if (cap_sens != 0)
touch_key();
checkCW();
checkTX();
if (keyeron)
keyer();
return;
case 5: // scan mode
```

```
scan();
break;
case 6: // set scan paramaters
scan_params();
break;
case 7: // A/B monitor mode
scan();
break;
case 8: // Fine tuning mode
finetune();
break;
}
delay(100);
}
```