NJAD VFO - Hacker's Guide
Version 1.1
John Price – WA2FZW

Table of Contents

Introduction

This document is designed to walk the builder through modifying the software for the ESP32 based VFO that I have been working on along with Glenn (VK3PE) and Jim (G3ZQC) to work with one's particular hardware implementation (described in the *NJAD VFO - Hardware Manual)*.

It's not just another digital VFO. This project is a heavily modified version of the VFO originally created by T.J. Uebo (JF3HZB).

Modifications made so far include:

- Multiband capability
- Allow a PCF8574 chip to read a band switch.
- Allow use of a PCF8574 chip to read a mode switch; note, different PCF8574s can be used for the band and mode switches or they can be read from a single chip.
- Allow use of either the ST7735 based display or the ILI9341 based display and allow a display size of up to 240x320 pixels. Both displays now use a standard library (*TFT_eSPI*), which can work with a number of other displays as well however we've only tested with the IL9431 and ST7735 displays.
- Eliminated all of TJ's code dealing with the rotary encoder and replaced it with a standard library. At the same time added the ability to use very high speed encoders and adjust the effective PPR rate in the software.
- Addition of a clarifier which can be implemented via a second encoder or a potentiometer (still having issues with the later).
- Added the ability to read the TX/RX status of the radio. This is needed for the clarifier operation and to prevent changing the frequency while the radio is transmitting.
- Many modifications to make the code easier to understand and to allow a high degree of user customization.
- Added CAT control.
- Added the ability for the software to key the transmitter when commanded to do so via CAT control.
- Added 2nd VFO and all the stuff that goes with that capability like split frequency (but only in the same band, at least for now).

- Added a “function button” which allows the dual VFOs to be manipulated manually (as opposed to only via CAT control).
- Added an optional battery monitor function for those that chose to run this from a battery as might be the case in a QRP radio.
- Added an optional button to cycle through frequency change increments of 10Hz, 100Hz and 1KHz.
- Modify the carrier oscillator to use only CLK0 or only CLK1 or both in quadrature mode (as it was originally in TJ’s code), or in quadrature mode with CLK1 inverted and out of phase with CLK0.

Something to be aware of; this is not your typical build it, program it and plug it in project. First of all, the hardware configuration needed is going to depend a lot on the radio you intend to use it with.

I’ve tried to make it possible to interface this with pretty much any radio, but you might find that in places, you may have to use a HIGH indication for something where the current software expects a LOW or vice-versa. Such things are going to require changes in the code itself as opposed to just changing the options described here.

There are no “default” settings for things. The *config.h* file contains over 130 definitions that the user can modify! Many of these ***MUST*** be set to match your particular hardware implementation. Failure to do so can result in program exceptions which will cause the program to constantly re-boot or at least unexpected operation. The other definitions control what is shown on the display and how it is displayed.

As so many of the definitions that must be set in the software are directly related to the hardware configuration, it is best to first read the *NJAD VFO - Hardware Manual* and make the appropriate implementation decisions for how you want to build the VFO for your application before going any further here.

Version 1.1 Modifications

In version 1.1, we added the capability to add a pushbutton switch that can be used to cycle through the operation modes. It functions in the same manner as the increment pushbutton in the original version.

The advantage of using this approach to changing the mode over the use of a rotary type switch read by the PCF8574 is that this can be used in conjunction with CAT control.

Another minor software change was to change all the “#include <arduino.h>” to “#include <Arduino.h>”. This is for those who are running the Arduino IDE on Linux systems on which file names are case sensitive.

Architecture

If you’re familiar with programming on the Arduino family of processors, you know that the basic architecture of all programs running on those have two main functions; *setup*, which initializes everything the program needs to run and *loop*, which runs forever and handles all of the tasks that need to be performed.

The ESP32 is a dual-core processor capable of running at speeds up to 240MHz. It also allows interrupts to be generated from any GPIO pin unlike the Arduino which has a very limited interrupt handling capability.

The beauty of the dual core processor is that there can essentially be two *loop* functions running simultaneously. So, in the VFO program, we have the usual *loop* function running in core #1 and a second *loop* function (named *task0*) running in core #0.

As the ESP32 can generate interrupts from any GPIO pin, a lot of the required tasks are handled via interrupts such as the two encoders and the transmit/receive indication from the radio.

TJ’s approach in the original code was brilliant (although IMHO, some of the implementation was troublesome). At any given time, there are essentially 2 copies of what’s to be displayed on the screen; one copy that is under construction and a second copy that gets sent to the display after the construction process has completed.

The *task0* function handles everything related to maintaining the operating frequencies and sending the correct frequency to the Si5351 module.

The *loop* function, along with the *dial* and *graph* functions handles maintaining the display.

The peripherals include the frequency encoder (although one of the mechanical encoders can be used, the use of a high-speed optical encoder is recommended), either a second encoder (mechanical is ok) or potentiometer for the clarifier, the Adafruit Si5351 module which provides the necessary frequencies to the radio and an optional PCF8574(A) GPIO expander chip (or two) for reading a band switch, mode switch or other things as needed, and of course, the display.

TJ's original code included support for the SEPS525 based OLED display and the ST7735 based TFT display. We eliminated support for the OLED display and replaced TJ's hand crafted code with a standard library (*TFT eSPI*), the code can theoretically now support a number of TFT displays with various driver chips. More information regarding the display support can be found in the *TFT eSPI Display Library* section below.

The initial version of this program was developed on the ESP32-DEVKIT3 board. The processor has since been upgraded to the TTGO T8 – V1.7 board. As of this writing, I believe there is a V1.8 version available. We haven't tested with the newer version, but I would suspect it would work ok.

The TTGO board has an additional 4Mb of PSRAM and that allowed 240x320 pixel displays to be used. Larger displays can theoretically be used; however the amount of data that has to be transferred from the processor to the display and the time that the transfer takes will degrade the analog-like motion of the dial. That degradation is already slightly noticeable using the 240x320 display.

The way the display is managed is truly clever! It's a three step process and really depends on the speed of the ESP32 to work. It's also a memory hog, which is why we went to the processor with the additional 4Mb of PSRAM; the memory requirement for the 240x320 display is 432Kb.

As things change (operating frequency, clarifier, Tx/Rx status, etc.), a map of the display is created in three arrays; *R_GRAM*, *G_GRAM* and *B_GRAM* which contain the red, green and blue values of each pixel (8 bits).

When the construction process is completed, the contents of those three arrays are combined into a single array (*GRAM65k*) which contains the 16 bit color value of each pixel. Once that is completed, the entire *GRAM65k* array is sent to the display in one big block. Thus, the display is working essentially like a TV screen in that it is being updated an entire frame at a time as opposed to being updated a pixel at a time. This is what gives the illusion of the dial being an analog type dial.

Rough measurements indicate that when using the 128x160 pixel display the screen can be updated about 14 times per second; for the 240x320 display, only about 4 updates per second are possible as the amount of data being transferred is roughly 4 times that needed for the small display.

The CAT control capability is covered in detail in the *NJAD VFO - CAT Control Manual*.

Arduino IDE Setup

This software is designed to be compiled using the Arduino IDE (Version 1.8.10 was the current version when this was written) as opposed to the Espressif ESP-IDF. Before you can use the Arduino IDE, there are a number of things you will need to do to allow it to work with the ESP32.

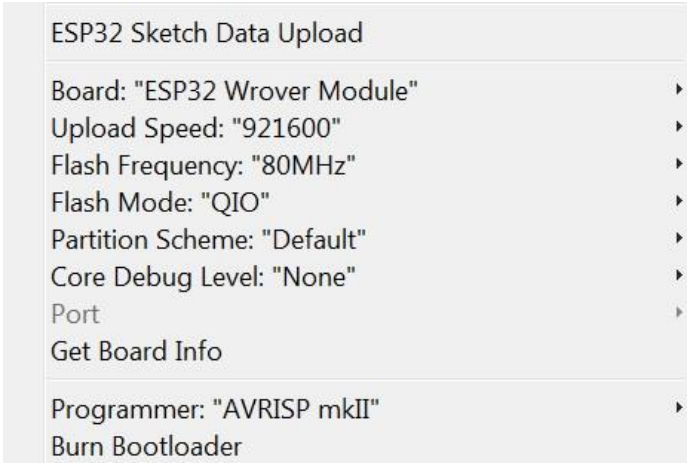
Step #1 is to add the support for the ESP32 to the IDE. Detailed instructions for doing that can be found here, so I won't repeat them.

Secondly, you will need three additional libraries to compile the VFO software:

- Rotary – Handles the encoders. There are a number of libraries for this out there. The one you want can be found here. Download the ZIP file and use the Arduino IDE's "Sketch – Include Library – Add ZIP Library" menu item to install it.
- TFT_eSPI – This library can be added using the Arduino IDE's "Tools – Manage Libraries" menu item. Enter "TFT_eSPI" in the search box, then click "Install".

- FT891_CAT – I originally developed this library as an integral part of the VFO program code, but then divorced it from the program. I have used it in a couple of other projects as well. It can be obtained from GitHub. Download the ZIP file and use the Arduino IDE's "Sketch – Include Library – Add ZIP Library" menu item to install it.

Finally, using the IDE's "Tools – Board" menu item you need to select the "ESP32 Wrover Module" from the list of available boards. Once you do that, there are a number of options for the ESP32 configuration that will be listed in the "Tools – Board" menu. Select the options as shown here:



Of course, you also need to select the correct COM port and you should be good to go!

As of this writing, there is talk of a new Arduino PRO IDE on the Software Controlled Ham Radio group. At the present time this IDE is an Alpha release and only supports Arduino boards, so it cannot be used for this application. This software has been successfully compiled with Version 1.8.9 and 1.8.10 of the Arduino IDE. I haven't used version 1.8.11 yet, but don't think that would cause any issues.

User Customization

One of the major things we've done with this version of the program is to make as many behaviors of the program user customizable. Almost everything that the user should need to change (there are a couple of exceptions) can be found in the *config.h* file. The only things you should need to change that aren't in the *config.h* file are the *bandData* and *modeData* arrays which are in the *.ino* file.

If you change things in other files, you're on your own!

Again note that there are no "default" values for many of the settings in the *config.h* file. You *MUST* make the appropriate changes to make the software match your hardware implementation or bad things will happen!

One of the nice things about the ESP32 is that almost any GPIO pin can be used for anything and all of the pins can be set to generate interrupts (unlike the Arduino). This means that within limits, the individual user can modify the pin assignments. This web page describes what can and cannot be used. Note that if you are using my PCBs or Glenn's, the pin assignments match the boards.

One issue that is noted on the page but not clearly is that GPIO-12 must be in a low state for the processor to reboot. The pin can be used as an output pin, but can only be used as an input if it is guaranteed to be in a low state when trying to reboot the processor (or load software into it). We've also had problems using GPIO-19 for anything; we're not sure why.

Optional Settings in *config.h*

The first section of the *config.h* file contains a number of definitions that enable or disable the optional hardware related features. In some cases, the hardware features can be implemented in a couple of different ways in the hardware. The *ESP32 Based VFO - Hardware Manual* provides more information about how to configure the hardware.

The pin number assignments defined in *config.h* and included here match the way my PCBs and Glenn's PCBs are designed, so unless you're using completely different hardware, you shouldn't change pin numbers.

The following are the optional hardware related features:

BIT_RATE – Sets the baud rate for the serial monitor and/or CAT control interface.

BAND_SWITCH – If this definition is set to *GPIO_EXPNDR*, a physical switch is read by a PCF8574 GPIO extender chip. If *BAND_SWITCH* is set to *CAT_CONTROL*, the band can only be changed via CAT control.

MODE_SWITCH – If this is set to *GPIO_EXPNDR*, the switch is read by a PCF8574 GPIO extender chip. If *MODE_SWITCH* is set to *CAT_CONTROL*, the mode can only be changed via CAT control. In the Version 1.1 software, this symbol can also be set to *PUSH_BUTTON* indicating that a momentary pushbutton is going to be used to cycle through the operating modes.

MODE_BUTTON (36) – Defines the GPIO pin number for the optional mode select pushbutton.

BAND_SW_SENSE & MODE_SW_SENSE – These two symbols define whether the active band or mode switch selection is defined by a HIGH (3V3) or LOW (grounded) reading on the PCF8574 pins.

CLARIFIER – If *CLARIFIER* is set to *NOT_AVAIL*, no clarifier is installed. If set to *ENCODER*, the clarifier is installed and implemented using a 2nd encoder (a mechanical one is ok). If *CLARIFIER* is set to *POTENTIOMETER*, the clarifier is implemented using a potentiometer (I still haven't managed to get the POTENTIOMETER implementation to work nicely, but you can try).

CLAR_SW_RESET – If this is set to *true* the clarifier offset will be reset to zero when the clarifier is turned off; if set to *false*, the offset will not be reset. Note, this applies only to the 2nd encoder implementation. If you are using the potentiometer, set this to *false* or strange things happen!

CLAR_FA_RESET – If this is set to *true* the clarifier offset will be reset to zero when the VFO-A frequency is changed either by moving the encoder or operating the function button or via CAT control; if set to *false*, the offset will not be reset. Note, this applies only to the 2nd encoder implementation. If you are using the potentiometer, set this to *false* or strange things happen!

PTT_LINE – Set this to *AVAILABLE* to indicate that the PTT indication from the transceiver is connected; set it to *NOT_AVAIL* if not. Note that if the clarifier is used, the PTT line *MUST* be connected or the clarifier offset will be applied to the transmit frequency as well as to the receive frequency! If the PTT is connected, this also prevents the operating frequencies from changing while transmitting.

PTT_PIN (4) – This pin provides the transmit/receive indication from the radio.

XMIT_PIN (12) – Used by the CAT control module to key the transmitter.

BATT_CHECK – Set this to *AVAILABLE* or *NOT_AVAIL* if you want to display the battery voltage.

BATTERY_PIN (35) – If *BATT_CHECK* is *AVAILABLE*, this defines the pin number which is used to read the battery voltage. Note that this capability is built into the TTGO ESP32 board, so the pin number cannot be changed. Note also, that the accuracy of the voltage readings is not all that great and if no battery is attached, the readings are meaningless.

Also note that this measures the voltage going into the ESP32 board, not perhaps a 12V battery one might be powering an entire rig from.

BATTERY_ADJ – The number associated with this symbol is a fudge factor to compensate for the non-linearity of the A/D conversion used on the battery monitor GPIO pin. Feel free to play with the value. There is a graph in the *ESP32 Based VFO - Operator's Manual* that shows the non-linearity.

INCR_BUTTON – If *AVAILABLE*, the button is installed. This button cycles through frequency change increments of 10Hz, 100Hz and 1KHz.

INCR_PIN (34) – If *INCR_BUTTON* is *AVAILABLE*, this is the pin number used for the button.

FUNCN_BUTTON – If *AVAILABLE*, the button is installed. This button cycles through the entries in the *modeData* array.

FUNCN_PIN (39) – If *INCR_BUTTON* is *AVAILABLE*, this is the pin number used for the button.

DISP_SIZE – If set to *SMALL_DISP* or *VK3PE_DISP*, the display size is assumed to be 128x160 pixels; if set to *LARGE_DISP*, the display size is assumed to be 240x320 pixels. A setting of *CUSTOM_DISP* can be used to implement a 'logical' display size (assumed to physically be a 240x320 screen). The positions of various items on the display and the font sizes used on the splash screen are controlled by this definition.

It should be noted that the software contains some conditionals that are specific to the *VK3PE_DISP* setting.

Band Switch Pin Assignments

The next section of *config.h* defines the PCF8574 pin assignments for the band switch based on the setting of the definition of *BAND_SWITCH*.

If *BAND_SWITCH* is set to *GPIO_EXPNDR* indicating that a PCF8574 is to be used to read it, the assignments are the pin designations on the chip used to read the switch.

In earlier versions of the software there was a requirement that when using the PCF8574 to read the band switch, the first band in the *bandData* array could have been assigned to any pin on the PCF8574 (0 to 7), however the pin assignments for the rest of the bands had to be on consecutive pins and in the same order as the bands are listed in the *bandData* array. This is no longer the case.

If no physical band switch is installed, then the *BS_nn* values compile to dummy numbers needed in the *bandData* array.

As distributed, we only have 6 bands defined; feel free to add more:

```
BS_80          80 Meters
BS_40          40 Meters
BS_20          20 Meters
BS_15          15 Meters
BS_10          10 Meters
BS_6M           6 Meters
```

If using the PCF8574 to read a physical switch, you can only define eight bands. If using CAT control to manage the frequencies and bands the limit is 255!

BS_I2C_ADDR – If using the PCF8574 to read the band switch, this definition is the I2C bus address of the PCF8574 used to read the switch. Note that there can be multiple PCF8574s on the bus to read other things (for example, the mode switch). Note also that if using a PCF8574, the address range is 0x28 to 0x2F; for the PCF8574A, the address range is 0x38 to 0x3F.

A single PCF8574 can be used to read both a band switch and a mode switch provided the maximum number of pins required is eight or less (and make sure the pin numbers don't overlap).

F_STEP – Defines the minimum allowable frequency change per encoder (pseudo) interrupt. It is set to '10', which is the lowest order digit displayed in the numerical frequencies.

Mode Switch Pin Assignments

The next section of *config.h* defines the PCF8574 pin assignments for the mode switch based on the setting of the definition of *MODE_SWITCH*.

If *MODE_SWITCH* is set to *GPIO_EXPNDR* indicating that a PCF8574 is to be used to read it, the assignments are the pin designations on the chip used to read the switch.

In earlier versions of the software there was a requirement that when using the PCF8574 to read the mode switch, the first mode in the *modeData* array could have been assigned to any pin on the PCF8574 (0 to 7), however the pin assignments for the rest of the modes had to be on consecutive pins and in the same order as the bands are listed in the *bandData* array. This is no longer the case.

If no physical mode switch is installed, then the *MS_aaa* values compile to dummy numbers needed in the *modeData* array.

As distributed, we only have 5 modes defined; you can define up to eight if using the PCF8574 or up to 255 if only using CAT control:

MS_LSB	Lower sideband
MS_USB	Upper sideband
MS_CW	CW
MS_AM	AM
MS_DIG	Digital

If you want to add more modes, the definitions for the *MS_aaa* definitions are in the *VFO_Defs.h* module. Read the comments in there regarding adding new definitions.

MS_I2C_ADDR – If using the PCF8574 to read the mode switch, this definition is the I2C bus address of the PCF8574 used to read the switch. Note that there can be multiple PCF8574s on the bus to read other things (for example, the band switch).

A single PCF8574 can be used to read both a band switch and a mode switch provided the maximum number of pins required is eight or less (and make sure the pin numbers don’t overlap).

SDA & **SCL (21 & 22)** – These pins are the standard I2C bus pins (21 & 22) and are used for the PCF8574 GPIO expander chip (if used).

Si5351 Associated Definitions

These items are all associated in some way with the operation of the Si5351 clock generator module:

VFO_FACTOR – This was something I needed to add for my Swan-250C project. The radio expects a VFO frequency in the 13MHz to 14Mhz range, but triples it in the radio. Thus I needed to use the tripled frequency in the *bandData* array and divide that by 3 to send to the Si5351. If you have a similar issue, set the divisor accordingly; if not, set it to ‘1’.

SI_XTAL – The Adafruit Si5351 module's nominal crystal frequency is 25MHz, however some other Si5351 modules may use a 27MHz crystal. If this is the case, change the value. In either case, the actual crystal frequency is most likely not going to be exact. The section titled *Frequency Calibration* explains how to calibrate the Si5351.

VFO_DRIVE & C_OSC_DRIVE – These set the output level of the VFO and carrier oscillators. The choices are: *CLK_DRIVE_2MA*, *CLK_DRIVE_4MA*, *CLK_DRIVE_6MA* and *CLK_DRIVE_8MA*. They define the output current delivered to an 85 ohm load (the output impedance of the Si5351).

SI_SDA & **SI_SCL (26 & 27)** – These define an alternate I2C bus used for the Si5351 clock generator. For some reason, TJ's Si5351 code causes some conflict with other devices on the normal I2C bus.

SI_I2C_ADDR – This is the I2C address of the Si5351 module.

Clarifier Associated Definitions

There are a few things that need to be defined if the clarifier option is installed. They are slightly different depending on the hardware implementation.

CLAR_ENCDR_SW (25) – Defines the GPIO pin used for the encoder switch. If either type of clarifier is installed, there needs to be a switch to turn it on and off. The assumption is that the switch is a momentary on pushbutton (like the switch on the mechanical encoders) that is normally HIGH and LOW when operated.

CLAR_ENCDR_A & **CLAR_ENCDR_B (32 & 33)** – If a 2nd encoder is used for the clarifier, these define the GPIO pins that it is connected to. If the encoder seems to work backwards, simply flip-flop the numbers.

CLAR_POT (32) – If a potentiometer is being used for the clarifier this defines the pin that it is connected to.

Frequency Encoder Definitions

FREQ_ENCDR_A & **FREQ_ENCDR_B (13 & 15)** – These pins are for the main frequency encoder. If it seems to work backwards, flip-flop the numbers.

ENCDR_FCTR – This is a divisor for the high-speed optical encoders, which are rated on the order of 400 PPR (Pulses per Revolution). As the code actually receives interrupts on the HIGH and LOW transitions of each of the two pins, they actually generate about 1,600 interrupts per revolution! By dividing the actual number of interrupts by this number the amount of movement of the dial per interrupt is reduced; if you are using a low-speed mechanical encoder, set this to '1'. I found a setting of '40' to work nicely for the high-speed encoder.

VSPI Bus Pins

These are listed in *config.h*, but commented out. They are not used in the program itself; however they are used in the *TFT_eSPI* library. They are not available for other uses.

```
SDI   -1    // MISO (master input slave output) not used
RESET -1    // Reset pin (connected to ESP32 RESET)
DC     2    // Data/Command pin
CS     5    // Chip Select pin
SCLK  18    // SPI clock pin
SDO   23    // MOSI (master output slave input) pin
```

Timer Definitions

It is not necessary to read the band switch, mode switch or other stuff on every pass through either of the looping functions. These definitions control how often those components will actually be read:

```
BS_READ_TIME     25UL  // Read the band switch every 25mS
MS_READ_TIME     25UL  // Read the mode switch every 25mS
CLAR_READ_TIME   25UL  // Read the clarifier every 25mS
CAT_READ_TIME    25UL  // Check for new CAT messages every 25mS
FB_READ_TIME     25UL  // Check the function button every 25mS
INCR_READ_TIME   25UL  // Check the increment button every 25mS
BATT_READ_TIME   60000UL  // Check the battery once per minute
```

Splash Screen Text

These define the four lines of text displayed on the splash screen. The vertical positions for each line are set dependent on the setting of *DISP_SIZE*, however, for now, the horizontal positions are hard coded in the software. If one changes the length of any line, adjustments will have to be made in the *PaintSplash* function in the *display.cpp* file to keep things centered.

```
SPLASH_1   "NJAD VFO"
SPLASH_2   "Version 1.1"
SPLASH_3   "Original:  JF3HZB"
SPLASH_4   "Modified:  WA2FZW"
```

Accelerator Parameters

One of the neat things about TJ's original code is his algorithm for increasing the speed at which the dial turns as the encoder knob is turned faster. Using his algorithm, it truly accelerates and decelerates.

We also provide a liner algorithm that simply multiplies the normal frequency increment by a constant.

These definitions control how this all works:

ACCELERATE – If set to *true*, the accelerator is turned on; if set to *false* it is disabled.

ACC_FACTOR – If set to '0', TJ's non-liner algorithm is used. If set to anything other than '0', the number is used as a multiplier for the currently selected frequency increment per encoder interrupt.

V_TH – If this many (or more) encoder interrupts are seen in a single read of the encoder, the accelerator kicks in.

Display Parameters

Because we can now deal with different size displays, a number of the parameters related to how things are displayed are conditionalized based on the display size.

See the section *Displays* for information on how to set up the TFT_eSPI library for your particular display.

PAINT_BOX – This definition controls whether or not the box around the numerical frequency is displayed or not. Whether the box itself is displayed, the numerical frequencies can still be displayed without the box.

PAINT_VFO_A & **PAINT_VFO_B** – These definitions control whether or not the numerical frequencies for VFO-A and VFO-B are displayed or not.

PAINT_MODE – This determines whether or not to show the operating mode (USB, LSB, etc.) on the display

PAINT_SPLIT – This determines whether or not to show the split mode status on the screen or not.

PAINT_UL – This determines whether or not to place a cursor under the VFO-A frequency that corresponds to the active frequency change increment.

TFT_MODE – This sets the display rotation. The value must be ‘0’ or ‘2’, both of which are landscape mode settings. Which you choose would depend on the physical mounting of the display in your particular application.

Display Size Dependent Parameters

The following definitions are set differently based on the setting of *DISP_SIZE*. They are repeated 4 times in conditionalized sections based on the definition of *DISP_SIZE*.

Note that the ‘X’ and ‘Y’ coordinates are relative to the lower left hand corner of the screen. When you want to display something near the top of the display, it is best to use a ‘Y’ coordinate definition like “*DISP_H – 20*”, which places something 20 pixels below the top of the screen or to place something relative to the right side of the screen, something like “*DISP_W – 30*”, which says 30 pixels from the right side of the screen.

DISP_W & DISP_H – These define the screen size in landscape mode. The values will be determined by the setting of *DISP_SIZE*. The numbers need not be the physical size of the display, but must be equal to or less than the actual size. For example, in the settings for the *CUSTOM_DISPLAY*, I have the screen height set to 180 pixels even though the actual display height is 240 pixels.

BOX_X & BOX_Y – These define the ‘X’ and ‘Y’ coordinates of the lower left hand corner of the numerical frequency box.

BOX_W & BOX_H – These define the width and height of the numerical frequency box. As distributed, these are set to the same values regardless of the setting of *DISP_SIZE*, even though they appear within the conditionalized settings for both display sizes.

VFO_A_X & VFO_A_Y – These define the ‘X’ and ‘Y’ coordinates of the numerical frequency for VFO-A. They are defined relative to the location of the surrounding box.

UL10X, UL100_X and **UL1K_X** – Horizontal coordinates for the active frequency increment cursor positions. They are defined relative to the position of the VFO-A numerical frequency and shouldn’t be changed.

UL_Y – The vertical coordinate for the cursors

UL_W – The width of the cursor.

VFO_B_X & VFO_B_Y – These define the ‘X’ and ‘Y’ coordinates of the numerical frequency for VFO-B. They are defined relative to the location of the surrounding box.

TR_X & **TR_Y** – These define the ‘X’ and ‘Y’ coordinates of the Tx/Rx indicator; again relative to the location of the frequency box.

MODE_X & MODE_Y – These define the ‘X’ and ‘Y’ coordinates of the current operating mode; in previous releases, these were also defined relative to the frequency box, but are now defined relative to the screen dimensions.

CLAR_X & CLAR_Y – These define the ‘X’ and ‘Y’ coordinates of the clarifier status; in previous releases, these were also defined relative to the frequency box, but are now defined relative to the screen dimensions.

SPLIT_X & **SPLIT_Y** – These define the ‘X’ and ‘Y’ coordinates of the split mode status; again relative to the display dimensions.

BATT_X & **BATT_Y** – These define the ‘X’ and ‘Y’ coordinates of the battery voltage reading.

D_HEIGHT – Defines the vertical position of the dial proper.

D_R – Defines the radius of the dial arc.

DIAL_FONT – The font size used for the numbers on the dial scales; can be ‘0’, ‘1’ or ‘2’.

DIAL_SPACE – The number of pixels between the arcs for the main and sub dials.

DP_WIDTH – The width of the dial pointer in pixels.

DP_LEN – The length of the dial pointer. Note that in the definitions for the *SMALL_DISP*, the length is conditionalized on whether or not the battery voltage is being displayed or not.

DP_POS – The vertical position of the top of the dial pointer relative to the top center of the outer dial arc. A setting of ‘0’ makes the top of the pointer equal to the top of the outer arc.

SPLASH_Y1 through **SPLASH_Y4** – The ‘Y’ coordinates for lines 1 through 4 of the splash screen. The values depend on the setting of *DISP_SIZE*. The ‘X’ values are currently hard coded in the software.

Other Display Parameters

There are many definitions that control how the dial itself is constructed. The comments in the *config.h* file explain what each does reasonably well, so I’ll not spell them out in detail here.

Color Definitions

The final section of the *config.h* file defines some basic colors that are used for various parts of the display and the assignment of colors to the various parts of the display. Feel free to add to the list if you want to use different colors than the ones listed here.

The 24 bit numerical values are in RED-GREEN-BLUE order. Note, however that although the definitions are 24 bit values, the color actually sent to the display is a 16 bit value, so some accuracy will be lost.

Here’s a link to a handy web page that allows you to visually select colors and get the proper 24 bit RGB value for them.

Following the definitions of the colors themselves is another set of definitions that associates specific parts of the display with specific colors. For example:

```
#define  CL_FREQ_BOX  CL_GREY    // Numerical frequency box
```

which says that the color used for the numerical frequency box will be grey.

The TFT_eSPI Display Library

Both the IL9431 and ST7735 displays are supported using the *TFT_eSPI* library, which can be installed into the Arduino IDE using the "Tools – Manage libraries" menu item. Search for *TFT_eSPI* and tell the IDE to install it.

There is one file that is part of the *TFT_eSPI* library itself that needs to be modified to set the display type and some other parameters associated with the particular display.

This file is located in the *TFT_eSPI* directory wherever your user-added library directory (go to "File – Preferences" in the Arduino IDE and the path should be shown as "Sketchbook location") is located and the file is named *User_Setup.h*.

There are a few things that need to be modified in this file:

- Which driver (aka display) you are using
- For the ST7735 only, the display size (in portrait mode)
- For the ST7735 only, there are a number of options that can be enabled if the display does not look right.

- There are several lines where you have to define the pins being used to control the display. Assuming you're using the same pin numbers as in the code as distributed and as the PCBs are constructed, these should be as follows:

```
//#define TFT_MISO 19     // Not used
  #define TFT_MOSI 23     // aka SDO
  #define TFT_SCLK 18
  #define TFT_CS    5     // Chip select control pin
  #define TFT_DC    2     // Data Command control pin
//#define TFT_RST  15     // Reset pin (could connect to RST pin)
  #define TFT_RST  -1     // RESET is connected to ESP32 board RST
```

- Finally, near the bottom of the file, you need to select the proper definition for the SPI bus speed. In testing, I found that the IL9431 can run at 40MHz, however, the specifications for the ST7735 dictate a maximum speed of 27MHz.

When trying to set up the parameters in this file, it is best to use some of the example programs that are distributed with the library itself. These can be found by going to the "File – Examples" menu item and scrolling down to the entry for *TFT_eSPI*. There are lots of programs that you can test with there before trying the VFO program.

Glenn and I have been using the *TFT_Meter5* example as a way of testing to ensure that the library is set up correctly. When that program is working correctly, it should look like this (without the slightly blue tint on the meter face that I couldn't eliminate; it should be white):

Other than the fact that the meter moves and all the text looks correct, the colors are important also, and should match the picture. If not, there are other options that can be set in *User_Setup.h* to make the colors correct. That file is fairly well commented as to what options do what.

There is another program in the *TFT_eSPI* library *examples/Tests and diagnostics* directory named *Read_User_Setup* that will display the settings in the library.

We've only tested the VFO program on the ILI9431 and ST7735 displays but as the *TFT_eSPI* library supports a number of other display types, there should be no reason that any display that the library supports couldn't be used.

In theory, displays larger than 240x320 could be used, however because of the way the software updates the display, this is not really a great idea. The software builds a complete image of the screen in memory and then transfers that image to the display in one large block. When the display size is greater than 240x320 the amount of time that the transfer takes causes the dial movement to lose the illusion of being an analog dial. This effect is noticeable even on the 240x320 pixel display.

Setting Up The *bandData* and *modeData* Arrays

Other than the things that need to be modified in the *config.h* and the TFT library *User_Setup.h* files, the only other things that you will need to modify are the *bandData* and *modeData* arrays in the *.ino* file.

bandData Array and *band_data* Structure

The *bandData* array contains an entry for each band available in the radio. If you're using it for a single-band radio, it must contain one and only one entry and the *BAND_SWITCH* symbol in *config.h* should be set to either *NOT_AVAIL* or *CAT_CONTROL*.

If you have a multi-band radio and are using the PCF8574 to read a physical band switch, there can be a maximum of eight bands defined.

If you have a multi-band radio that is using CAT control for the frequencies and bands, you can define up to 255 separate bands.

The entries in the *bandData* array are *band_data* structures, which contain a number of elements needed to control the frequency limits of each band and the VFO behavior for each band.

The structure is defined in the *VFO_Defs.h* file. Here's the structure definition as currently implemented:

```
typedef struct
{
  uint32_t   vfoA;      // VFO-A frequency (receive in split mode)
  uint32_t   vfoB;      // VFO-B frequency (transmit in split mode)
  uint32_t   refFreq;   // Freq at which actual VFO freq is at the reference value
  uint32_t   vfoRef;    // The VFO reference frequency
  uint32_t   lowLimit;  // The lower band edge
  uint32_t   topLimit;  // The upper band edge
  int16_t    incr;      // The frequency change increment index for this band
  uint8_t    bandSW;    // Band switch pin number
  int8_t     vfoDir;    // +1 - VFO frequency increases; -1 it decreases
  uint8_t    opMode;    // Default mode (subscript to modeData) for this band
} band_data;
```

Most of the entries should be self-explanatory except the things relating to the reference frequencies. So here's how that works:

The *refFreq* corresponds to the *vfoRef*, and may or may not be one of the band limits (although it should be). What these dictate is that when the frequency displayed on the display is at the *refFreq*, the actual frequency being sent to the Si5351 will be the *vfoRef* frequency.

The *vfoDir* entry is set to plus or minus one and indicates whether the Si5351 frequency increases or decreases with respect to the *refFreq* as the operating frequency (*txFreq*) increases or decreases.

In the case of Glenn's FT-7 implementation, the reference frequencies are set to 500 KHz above the lower edge of the band and the VFO reference frequency is set to 5.5 KHz on all bands. The VFO direction indicator is set to '-1' on all bands indicating that the Si5351 frequency decreases as the displayed frequency decreases.

The *incr* entry is an index to the *incrList* array which contains the amount to change the frequency for each encoder click.

If using the PCF8574 to read a physical band switch, the *BS_nn* definitions have to be set appropriately in the *config.h* file. If not using a physical band switch, the *BS_nn* symbols have to be assigned to dummy numbers in *config.h*.

modeData Array and *mode_data* Structure

The *modeData* array contains an entry for each operating mode available in the radio.

The entries are *mode_data* structures, which contain a few things we need to know about each operating mode. Here's the current implementation:

```
typedef struct
{
   uint32_t  coFreq;      // Carrier oscillator frequency
   uint8_t   coMode;      // Carrier oscillator mode
   uint8_t   modeSW;      // Mode switch pin number
   uint8_t   catMode;     // Mode selection value used by the "MD" CAT command
   int8_t    vfoAdjust;   // VFO frequency offset to compensate for CO frequency
   char*     modeString;  // String to be displayed on screen
} mode_data;
```

The *catMode* value corresponds to the number used in the Yaesu FT-891 "MD" message to indicate what the operating mode should be. Consult the *Yaesu FT-891 CAT Manual* for the specific values.

The array should look something like this:

```
mode_data modeData[] =
{
   { 10904000UL, C_OSC_CLK0, MS_LSB, 1, -1600, "LSB" },
   { 10898100UL, C_OSC_CLK0, MS_USB, 2,  1600, "USB" },
   { 10899400UL, C_OSC_CLK0, MS_CW,  3,     0, "CW"  },
   { 10898000UL, C_OSC_CLK0, MS_AM,  5,     0, "AM"  },
   { 10898000UL, C_OSC_CLK0, MS_DIG, 8,     0, "DIG" }
};
```

The carrier oscillator frequencies shown are those for my Swan-250C.

The choices for the carrier oscillator mode are:

C_OSC_OFF	No carrier oscillator
C_OSC_CLK0	Carrier oscillator output only on CLK0
C_OSC_CLK1	Carrier oscillator output only on CLK1
C_OSC_QUAD	Quadrature mode (CLK1 lags CLK0 by 90°)
C_OSC_QUAD_R	Reverse quadrature mode (CLK1 leads CLK0 by 90°)

The quadrature modes are used for direct conversion radios as a means of switching sidebands. Please note that we haven't actually tested the *C_OSC_QUAD_R* mode in a radio, however, the *C_OSC_QUAD* mode was tested in a direct conversion receiver. On an oscilloscope, the *C_OSC_QUAD_R* mode seems to be working.

The *vfoAdjust* value is used to adjust the actual frequency sent to the VFO (Si5351 – CLK2). In my Swan-250C, the carrier oscillator frequency is shifted 3.2KHz to put the carrier frequency on the high or low side of the crystal filter depending on whether the mode is USB or LSB. By offsetting the actual VFO frequency by the *vfoAdjust* value, the dial and numerical frequencies will display the actual (suppressed) carrier frequency which will be the same for both USB and LSB (I haven't figured out the other modes yet).

Depending on how your particular radio works, you may have to use different values even if not using the carrier oscillator output of the Si5351. The numbers shown above (+/-1600) would apply on my Swan even if using the internal crystal controlled carrier oscillator.

If using the PCF8574 to read a physical mode switch, the *MS_aaa* pin number definitions have to be set appropriately in the *config.h* file. If not using a physical band switch, the *MS_aaa* symbols have to be assigned to dummy numbers in *config.h*.

Physical Switches versus CAT Control

This is as good a place as any to add a few words about the use of physical band and/or mode switches versus using CAT control to manage them.

Even if you specify that the program is supposed to read physical switches to select the band and/or mode that does not disable the ability to set either using CAT commands, but here's the problem. If the physical switches are enabled (*BAND_SWITCH* and/or *MODE_SWITCH* set to *GPIO_EXPNDR*), that does not disable the CAT control capability!

Suppose you have the *MODE_SWITCH* definition set to *GPIO_EXPNDR* and the program receives a CAT command to switch to another mode; it will actually do that. BUT, in *MS_READ_TIME* milliseconds or less the program is going to read the physical mode switch again and set the mode back to whatever is selected by the switch!

Which to use? Well, that's going to depend on your radio. In my Swan-250C, there are actually two (for whatever reason) switches that control the mode; USB and LSB are on one switch and CW is on another. Those switches change the frequency of the carrier oscillator in the radio to put the carrier on one side or the other of the filter, or in the passband for CW operation, so as long as I'm using the radio's carrier oscillator, I'm stuck using the physical switches and I'm not yet sure if I can interface them to the VFO hardware, thus for the time being, for my application, I have *MODE_SWITCH* set to *CAT_CONTROL*.

My plan is to replace the internal carrier oscillator with one of the Si5351 outputs, in which case, I can easily control the mode via CAT commands.

This is just one example of the thought process you'll need to go through to figure out how to set up the hardware and software.

Other Files

The following sections contain a brief description of each of the files that make up the whole program and list the functions contained in each along with a short description of what those functions do in the grand scheme of things.

The NJAD_VFO_Vn.n.ino File

This is the main file for the program and controls everything that happens. It contains the following functions in order of appearance:

setup – The standard function to initialize all the variables used throughout the program, setup GPIO pins, interrupts, etc.

loop – This is one of two loop functions (remember, the ESP32 is a dual core processor) and its job is to handle everything related to painting the display. It runs in core #1.

task0 – This is the second loop function if you will. It runs in core #0 and handles everything related to managing the VFO and carrier oscillator frequencies.

ReadBandSwitch – Does just what the name implies; if there is a physical band switch installed, this function finds which switch setting is active. If the active band changes, it handles setting the new band and appropriate frequencies.

ReadModeSwitch – If there is a physical mode switch, this function reads the switch and sets the proper operating mode.

CheckModeButton – If the *BAND_SWITCH* symbol is set to *PUSH_BUTTON*, this function checks to see if the button has been operated and changes the operating mode appropriately.

ReadBattery – If the option to monitor the battery voltage is enabled, this reads the GPIO pin that gives the battery reading and converts the pin reading to an actual voltage; again, this is the voltage as seen by the ESP32 board, not the voltage that might be powering the entire radio.

CheckCAT – This function handles all command messages that arrive from the CAT interface. Those commands can change the frequency, band, mode, split status and transmit status. Commands can also swap the two VFOs or copy one to the other.

CheckFreq – This function is used by the CheckCat function to validate whether or not a new frequency falls into one of the defined bands and whether or not a band change is needed.

FrequencyISR – Interrupt service routine for the frequency encoder. It just maintains a +/- counter of the number of interrupts received. The counter goes plus or minus based on which direction the encoder is turning.

PTT_ISR – Interrupt handler for the transmit/receive indicator. It looks at the state of the *PTT* pin and changes the transmit/receive status flag appropriately.

InitClarifier – Determines if an encoder based or potentiometer based clarifier is installed and sets up the proper pins and interrupt handlers.

ClarifierISR – If an encoder based clarifier is installed, this interrupt handler keeps track of the clarifier offset.

printBandData – Is a function that is available for debugging purposes. It displays the contents of the *bandData* array on the serial monitor.

CheckFcnButton – This function reads the function button and performs any required actions based on the sequence of button operations or if it is being held down.

CheckIncrButton – If the increment button is installed, this function cycles through frequency change increments of 10Hz, 100Hz and 1KHz each time the button is operated.

CheckClarSwitch - If the clarifier is installed, this function looks for the switch to be operated and when it is, toggles the state of the *clarifierOn* variable.

SwapVFOs – Swaps the frequencies in VFO-A and VFO-B.

Delay – Is a non-interrupt blocking delay function.

VFO_defs.h

This file contains definitions of symbols that the user should never change along with the definitions of many of the structures used in other parts of the program.

dial.cpp & dial.h

Together, these two files contain all the functionality to draw the actual VFO dial. There are only 2 functions that you might be interested in (there are others only used internally):

InitDial – Initializes everything needed to subsequently update the dial image.

Dial – Constructs the dial image based on the current value of the *rxFreq* variable (the dial always displays the receive frequency even when split mode is turned on).

display.cpp & display.h

These two files contain all the functions used to paint the display.

A word about how the display image is maintained is in order here.

The code in the *NJAD_VFO_Vn.n.ino* file builds a pixel by pixel map of the screen in three arrays, *R_GRAM*, *G_GRAM* and *B_GRAM*. These three arrays contain the red, green and blue components of each pixel in an 8 bit format.

When the image is completed, the red, green and blue components are merged into a 16 bit color code in the *GRAM65k* array. When it's time to actually update the display, the contents of the *GRAM65k* array are sent to the display in one huge block.

Here are the functions in these files:

InitDisplay – Initializes the TFT display.

Transfer_Image – Sends the contents of the *GRAM65k* array to the display.

trans65k – Combines the red green and blue pixel data into the *GRAM65k* array.

PaintSplash – Paints the splash screen at startup.

font.h files

There are four font files:

font.h
font12.h
font16.h
font20.h

These contain the bit values of the various fonts used throughout the program and I don't have a clue as to how they work!

graph.cpp & graph.h

These files handle drawing strings, characters, lines and boxes on the screen; well, they don't actually paint the screen, but rather construct the pixel maps for those items.

The included functions are:

ClearGRAM – Sets the contents of the pixel map to all black (for clearing the screen on the next display update).

Line – Draws a line on the screen.

BoxFill – Draws a box filled with a specified color.

Box – Draws an unfilled box

disp_str – There are actually 4 *disp_str* functions for different sized fonts. They paint entire strings of characters.

disp_chr – There are actually 4 *disp_chr* functions for different sized fonts. They paint individual characters.

si5351.cpp & si5351.h

These two files contain all the functions to operate the Adafruit Si5351 clock generator module:

Si5351_Init – Initializes the module.

Set_VFO_Freq – Sets the VFO frequency in CLK2 of the Si5351.

Set_Carrier_Freq – Sets the carrier frequency in CLK0 and CLK1 of the Si55351 in a single clock or quadrature mode (signals 90° out of phase).

DoTheMath – This is used internally to compute the actual bit patterns that must be loaded into the Si5351 to generate a specified frequency.

SetXtalFreq – Sets the nominal Si5351 crystal frequency (as defined by the value of the *SI_XTAL* symbol).

SetCorrection – Sets the calibration factor for the Si5351. See the *Frequency Calibration* section to learn how to calibrate the module.

Frequency Calibration

The Si5351 crystal frequency as defined by the value of the *SI_XTAL* symbol is a nominal value. The actual crystal frequency is almost never going to be exact, so the Si5351 module that you are going to install in your radio must be calibrated, or your transmit/receive frequencies will not be accurate.

We've provided a separate program to perform this function. The *Calibrate Si5351* program can be found on GitHub along with the instructions on how to use it.

The *Calibrate_Si5351* program saves a calibration factor in the ESP32's EEPROM memory and the VFO program will retrieve that calibration factor and feed it to the Si5351 module of the VFO program.

GPIO Pin Definitions

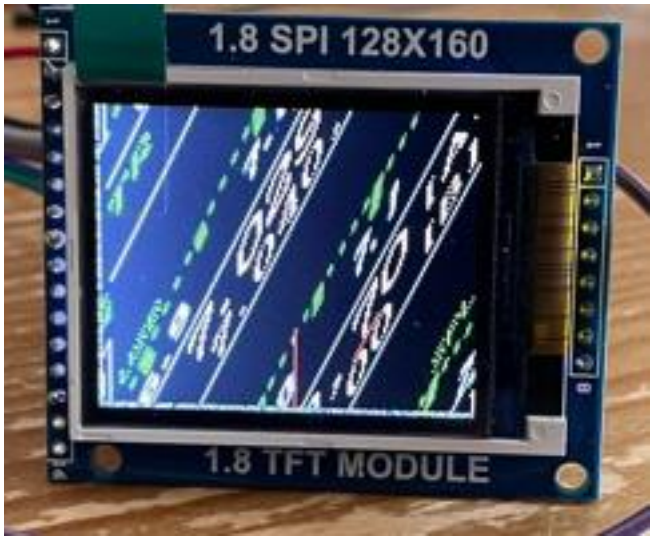
The following is a summary of the GPIO pin assignments. Note that these represent how I have them assigned on my PCBs and on Glenn's PCB. If you are using either of those PCB designs, you should NOT change any of the assignments other than flipping the numerical assignments for the encoder pins if the encoders operate backwards.

If you are using a PCB of your own design, you can change the assignments subject to the limitations described on this web page.

2	Display DC
4	Transceiver PTT line (LOW indicates transmitting)
5	Display - CS (VSPI Bus)
12	For CAT control - Keys the transmitter
13	Frequency Encoder – A
15	Frequency Encoder - B
18	Display - SCLK (VSPI Bus)
19	Reserved - Display MISO (VSPI bus - for reads)
21	PCF8572 - SDA (Standard I2C bus)
22	PCF8572 - SCL (Standard I2C bus)
23	Display - MOSI/SDO (VSPI Bus)
25	Clarifier Switch
26	Si5351 - SDA (Alternate I2C bus)
27	Si5351 - SCL (Alternate I2C bus)
32	Clarifier Encoder - B (or optional Potentiometer Clarifier)
33	Clarifier Encoder - A
34	Increment adjustment button (optional)
35	Battery monitor (optional)
36	Mode selection pushbutton (optional)
39	Function button (optional)

Known Issues

Bob (G3PJT) had a problem using the 12x x 160 display which was causing the screen to be distorted as shown here:



We couldn’t find a solution to the problem in the VFO software, however it turned out that Bob was using version 2.3.61 of the TFT_eSPI library (latest version at the time). When Glenn, Jim and I developed the software, we were using version 2.2.14 of the library.

When Bob reverted back to the older version it worked correctly. Although even though the display has a green tab, Bob had to tell the library that it was a display with a black tab.

If you are having display problems using a version of the library newer than version 2.2.14, that might be your problem.

Suggestion Box

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