Description of the source code for the mcHF transceiver

Updated for firmware version 0.0.219.26 by KA7OEI

The mcHF firmware has its original foundation based on the STM32F405/STM32F407 "Discovery board" and uses various ST Micro and Keil libraries as well as other (attributed) sources for the different modules.

Please note that this document is not complete and does not describe, in detail, all of the modules – only those that have been extensively studied and "touched" by its author. There are many other modules – many of them provided by the manufacturer and from other sources – that have not been exhaustively studied and their inner-workings remain unclear.

Directories within the source code file, in alphabetical order:

cmsis, cmsis_boot, cmsis_lib:

These contain the definitions for the CMSIS functions, including I/O and arithmetic. These should not be touched.

drivers:

This directory contains the majority of the core files that define the fundamention operation of the mcHF transceiver and where most of the development work is done.

firmware_Target_Flash:

This containes the compiled binaries, object and various debug files.

hardware:

The files contained herein define the majority of the I/O related to the mcHF transceiver. The file "mchf_board.h" is where EEPROM locations are defined by name and also where the main "transceiver state" (ts) structure is defined: It is the "ts" structure that contains the majority of the working variables in the transceiver.

libs:

This contains the compressed math libraries related to the math functions. Care should be taken when updating these files to newer versions as dependencies/incompatibilities are likely to arise!

misc:

This directory contains only the definitions for the addresses used by the "Virtual EEPROM" – a portion of the processor's FLASH memory that is used to store/recall nonvolatile settings. These functions are modified versions of those provided by ST Micro as detailed in the

documentation/source.
stdio:

This contains only "printf.c" which is used solely for debug purposes to send asychronous data via the "Service connector".

syscalls:

Not much is known about the purpose of these functions.

usb:

These are the various USB functions, supplied by ST Micro, that interface with a variety of devices that may be connected. At present, the mcHF utilizes very little of its USB capability and considerable work remains to be done to further development to support things like keyboard, memory, network, remote serial (e.g. "CAT" interface) and other devices.

The root directory:

The root directory contains only a few files that might be touched in the maintenance of the code:

- **main.c** – Being "C"-based, this is, in fact, the "main" function and in it may be found initialization, etc. related to startup and various low-level function calls. *This will be discussed in more detail later in this document.*
- **link.ld** – These are configurations for the compiler (at least with CooCox) and care must be taken to preserve the original settings if these are modified!
- **memory.ld** – These contain information related to the memory map/configuration and care must be taken to preserve the original settings if these are modified!

Most of the other files are environment/compiler/linker related. In particular, the file "firmware.coproj" is the main configuration used for by the CooCox environment. If you find that CooCox has inexplicably "blown up" and will not restart properly it is often the case that copying this file from an older backup that is known to work into the directory of the one that is not working will "fix" these issues: Always remember to save the old, non-working version – just in case!

Description of the important mcHF-related modules:

The file main.c:

This file contains a number of low-level functions and definitions, described below generally in the order that they appear:

VirtAddVarTab[] - This is the main array that defines the "Virtual EEPROM" variables used by the system to store nonvolatile information such as radio configuation, frequency/band, etc. Because of the construct of the ST Micro function it is necessary to manually define each and every address location used, hence the manner of its appearance. If more Virtual EEPROM memory locations are to be added please note that variable "NB_OF_VAR" must be modified as well as the file "eeprom.h".

Yes, this is a kludge, but it works! (No, I didn't write the "Virtual EEPROM" code!)

* * *

There are also a number of low-level handlers for things like CritialError, NMI, HardFault, problems with the memory manager, bus, etc. Most of these simply output a message via STDIO and set the defined "CriticalError" value.

* * *

SysTick_Handler() - This is an interrupt (low-priority) that was originally used to sequence the various user-interface tasks. It is still called, but it is no longer used as it was possible for re-entry to occur and corrupt memory if tasks were taking longer than expected.

EXTI0_IRQHandler – This services the edge-triggered "EXT1" interrupt that is assigned to **PE0**, the "PADDLE_DAH" and "PTT" line. It was discovered after much head-banging that this interrupt could also be triggered if the signals on PC4 and PC5 (Step- and Step+) had slow rise/fall times or multple bounces *(this is <u>**NOT**</u> in the errata for this chip as far as is known!)* so this function includes additional validation to make ***absolutely sure*** that ***ONLY*** the **PE0** line was the source of the interrupt: All other causes of this interrupt are ignored.

EXTI1_IRQHandler – This services the edge-triggered "EXT1" interrupt that is assigned to pin **PE1**, the "PADDLE_DIT" line. As with the case of **PE0**, above, it was discovered after much hair-pulling that this interrupt could also be triggered by unrelated IO pins – also a hardware bug that appears not to be properly documented! To prevent problems, the source of the interrupt is validated to make sure that it was **PE1** that was operated and then ***ONLY*** while in CW mode since it is only in that mode that "PADDLE_DIT" is actually used.

EXTI5_10_IRQHandler – This services the interrupt related to the POWER button. Note that pressing the POWER button does not always mean a power-off (e.g. it may be used to select display brightness) and this activation is further validated elsewhere.

TransceiverStateInit – This function sets the variables contained within the "Transceiver State" (ts) structure to their initial states at power-up. Note that this is called ***before*** variables are read from the EEPROM so many of those related to user-stored settings are later overwritten.

IT IS IMPERATIVE that **ALL** variables contained within the "ts" structure be initilized in this function – even if you think that they are supposed to be later loaded with variables from the EEPROM! Additionally, make absolutely certain that you use "reasonable" variables for initialization – such as "off" or defined "default" settings.

MiscInit – Presently, this initilizes only the "softdds", a function used to generate (quadrature) sine waves at defined frequencies for the CW and TUNE functions.

wd_reset - This initializes the watchdog timer, which is not fully implemented.

main – Yes, this is the "main" function for the entire mcHF transceiver. It performs the following functions:

- mchf_board_init – This sets up the processor's hardware, etc. To get the board running.
- mchf_board_green_led – it turns on the green LED
- TransceiverStateInit – this sets the initial states of the "ts" structure's variables
- EE_Init – This gets the current status of the EEPROM initialization to indicate if it will function.
- UiLcdHy28_ShowStartUpScreen – this shows the start-up splash screen for a specified amount of time. This screen also contains various messages to indicate warnings, configuration, etc.
- MiscInit – This starts the "softdds" as noted above.
- UiCalcRxPhaseAdj – This is the function that is used to adjust the RX IQ phasing, but it also loads the (default) coefficients for the receiver's Hilbert Transformer. This is necessary before we initialize the audio driver.
- UiCalcTxPhaseAdj – This is the function that is used to adjust the TX IQ phasing, but it also loads the (default) coefficients for the transmitter's Hilbert Transformer. This is necessary before we initialize the audio driver.
- UiDriverLoadFilterValue – This gets the (default) audio filter value so that we have **something** with which we can initialize our audio filters. This is necessary before we initialize the audio drivers.
- audio_driver_init – This initializes the variables for the "ads" structure and starts the audio driver so that the receive/transmit functions may begin.
- keyb_driver_init – not yet implemented
- ui_driver_init – This does the initial configuration of the user interface which includes reading of stored values from EEPROM and drawing the screen.
- audio_driver_init – we re-initialize the audio driver using the settings (filters, AGC, DSP and MANY others) that we just loaded from EEPROM since the default settings – necessary for initial start-up – are probably not the same as what is stored.
- uiCodecMute – This is called to make sure that the audio codec (and audio) is un-muted.

At this point the transceiver is now initialized. The "Transceiver main loop" is now entered and the function "ui_driver_thread" is continually called.

Note: The audio processing is handled in the background by an ISR that pulls data from and pushes data to the audio codec chip via DMA and, when full, signals that this data be processed. This ISR has highest priority and oversubscription in the audio processing ISR will cause the transceiver's GUI operation to slow to a crawl/halt!

Underdeveloped code within the "drivers" directory:

cat – The "cat" subdirectory contains the functions related to the remote control via the USB interface, called "CAT" by Yaesu and possibly others. These functions are not fully-developed and at this point only "PTT" indications are conveyed via this interface.

keyboard – As with the "cat", the USB keyboard interface functions – and a user interface that might go along with it – are not developed.

The "hardware" subdirectory:

This directory contains a number of files related to the definition and configuration of the mcHF hardware.

The file mchf_board.c:

This file contains the majority of the low-level hardware definitions and some of the basic control functions as follows:

- mchf_board_led_init – This function configures the I/O pins for the red and green LEDs on the front panel.
- mchf_board_debug_init – This function configures the USART that may be used when debugging the transceiver when output is sent to STDIO via the "printf" statement. The default baud rate is 921600 bits per second and is interfaced via the "service" connector.
- mchf_board_keypad_init – This function configures the inputs related to the input buttons – except for the POWER button.
- mchf_board_ptt_init – This initializes the main PTT (transmit) control pin (e.g. PB1).
- mchf_board_keyer_rq_init – This function sets up the interrupts related to the DAH and DIT pins (PE0, PE1, respectively).
- mchf_board_power_button_irq_init – This function sets up the I/O and interrupt for the POWER button.
- mchf_board_dac0_init – This function sets up the onboard DAC for the JFET RF attenuator on pin PA4. (This I/O pin is unused by the current firmware.)
- mchf_board_dac1_init – This function sets up the onboard DAC for the PA bias on pin PA5.
- mchf_board_adc1_init – This function sets up the A/D converter for reading the power supply voltage via pin PA6.
- mchf_board_adc2_init – This function sets up the A/D converter for reading the forward RF power via pin PA3.
- mchf_board_adc3_init – This function sets up the A/D converter for reading the reverse RF power via pin PA2.
- mchf_board_power_down_init – This function sets up I/O pin PC8 to control the power on/off for the transceiver.
- mchf_board_band_ctr_init – This function sets up I/O pins PA7, PA8 and PA10 for control of the band-switch relays.
- mchf_board_watchdog_init – This function sets up the MCU watchdog timer. *(Not currently used).*
- mchf_board_set_system_tick_value – This sets up a hardware counter as a system source for timing events.
- mchf_board_green_led – This function controls the front-panel green LED: 0 = off, 1 = on, other value = toggle.
- mchf_board_red_led – This function controls the front-panel red LED: 0 = off, 1 = on, other value = toggle.
- mchf_board_switch_tx – This function switches the transceiver hardware to TX mode and activates the red LED, turning off the green LED if the passed value is TRUE; It switches to RX mode and activates the green LED, turning off the red LED if the value is FALSE.
- mchf_board_power_off – This function will, clear the screen, display a countdown, mute audio,

save parameters to EEPROM and then signal to allow the transceiver to be powered down once the POWER button is released.

- mchf_board_init – This function initializes the various I/O and hardware modules, the Si570, etc.
- mchf_board_post_init – This function initializes the "tick" timer, the power button interrupts, the PTT states and the keyer interrupts.
- ReadVirtEEPROM – This function interfaces with the ST Micro functions that read from the virtual EEPROM. ***This interposing function is necessary for the proper functioning of the virtual EEPROM!***
- Write_VirtEEPROM – This function interfaces with the ST Micro functions that write to the virtual EEPROM. ***This interposing function is necessary for the proper functioning of the virtual EEPROM!***
- WriteVirtEEPROM_Signed – This is a version of the above function that writes a signed variable (int) to the EEPROM by passing using the pointer of an unsigned to pass the value. This results in a compiler warning, but it is, thusfar, the only way that I've found to reliably write signed values to the virtual EEPROM! *(Yes, I suppose that I could try the "wrap-around" method using unsigned vars, but this was easier.)* ***This interposing function is necessary for the proper functioning of the virtual EEPROM!***

The file mchf_board.h:

This file contains the definitions/names of the variables that define the aspects of the hardware. It also contains the version information of the firmware and the attributions seen on power-up.

There are many hardware-related definitions in this file, including those related to the "bottom" frequencies of the defined amateur bands and their "sizes", definitions related to audio filters – bandwidths and center frequencies, default band-gain values, many default settings used in the EEPROM Read functions and the menu system, all EEPROM addresses/locations, the definition of the main "transceiver state" structure ("ts") and may others.

Be absolutely certain that you understand the functions and related dependencies <u>before</u> you modify any of the values contained in this file!

The file mchf_hw_i2c.c:

This file contains the low-level I2C functions used to communicate with the Si570 and the MPC9801, the temperature sensor associated with it. The I2C functions to communicate with the audio codec, U1, are NOT located in this file.

The file mchf_hw_i2c2.c:

This file contains additional code related to the low-level I2C functions used to communicate with the Si570 and the MPC9801, the temperature sensor associated with it. The I2C functions to communicate with the audio codec, U1, are NOT located in this file.

Files and subdirectories within the "audio" directory:

The drivers->audio subdirectory contains those functions and definitions related to audio and DSP operations.

The "codec" subdirectory:

In the "codec" subdirectory are those functions that communicate with U1, the audio codec via the I2C interface.

i2s.c - This contains the functions that initialize the DMA and interrupt servicing related to moving the data in and out between the codec and the MCU. Note that bidirectional audio is required in both transmit and receive: A/D data from the receive mixer goes into the MCU and is processed and then out to the D/A to the speaker on receive while A/D data from the microphone is input to the MCU and then output to the D/A to the modulator for transmit.

codec.c – This contains functions used to configure the hardware of the codec itself and none of them should be modified unless you have a complete and thorough understanding of the W8731's registers and the hardware!

The functions contained include:

- Codec_Reset – This initializes the codec hardware to an inial state
- Codec_RX_TX – This is a rather complicated function that is very easy to break unless you know *exactly* what you are doing. This function is called when the transceiver goes between receive and transmit – in either direction – and reconfigures the codec's hardware and settings appropriate to the current mode. Proper sequencing and configuration is absolutely essential to prevent squeaks, clicks and other disruptions. This function also sets the microphone and LINE IN gains as appropriate as well as calling the function that calculates the proper setting for the CW sidetone amplitude.
- Codec_SidetoneSetgain – This function considers the "scaling factor" used for the current band (e.g. the amount of D/A gain applied to set the current power level) and calculates the appropriate codec gain setting to achieve the amount of sidetone that is commensurate with the user's "STG" (SideTone Gain) setting.
- Codec_Volume – This sets the receive audio volume based on the inputted value. Note that the "Right Headphone Out" on the codec chip is actually the "Line Out" and is always fixed, so this setting affects only the "Left Headphone Out" on the codec. Also note that this is ***ONLY*** a hardware gain adjustment: The actual user-interface volume control operates a combination of ***BOTH*** this hardware gain adjustment and software audio level adjustment, together.
- Codec_Mute – This is used to "clicklessly" mute the audio codec. Note that adjustment of the Codec Volume may not automatically un-mute it: Use this function ONLY if you can keep track of the fact that you muted so that you can un-mute it later!
- Codec_WriteRegister – As you might guess, this writes to the codec hardware.
- CodecAudioInterfaceInit – This configures the codec for the sample rate and audio format to be used.
- Codec_GPIO_INIT – These are configurations for the processor itself in setting up the IP and the 12.288 sample clock for the codec.
- Codec_Line_Gain_Adj – This adjusts the gain of the "LINE" inputs for the codec which are

used for receive or transmit if the "LINE" (not Microphone) input is not used. Do not use this unless you know exactly the format that is to be used!

"softdds" subdirectory:

In the "softdds" subdirectory contains functions that generate arbitrary sine waves – in quadrature – to generate the baseband energy required to produce CW tones (e.g. unmodulated carriers) for the generation of Morse keying and when the TUNE button is pressed.

The functions contained include:

- softdds_setfreq – This function sets the frequency of the audio tone being generated, based on the sample rate. If the variable "smooth" is TRUE, the starting amplitude of the sine wave will be forced to zero: This is something that would be done for the generation of a CW signal – which also includes envelope chaping – but it would NOT be done for the production of audio signals that require phase-continuous tone generation such as RTTY or WSPR! If this function is called while tone generation is active, the change will take instant effect.
- sofdds_run – This produces quadrature sine waves in the "i" and "q" buffers passed to it of size "size". Note that this sine generation does not use "sine" and "cosine" functions, but rather a much faster lookup table.

The "cw" subdirectory:

In the "cw" subdirectory may be found the functions that generate the CW, based on input from a straight key or paddle. These function call the "softdds" function and apply amplitude (envelope) shaping to assure appropriate rise/fall time to minimize "key clicks."

The functions contained include:

- cw_gen_init – This sets the appropriate keyer mode. ("Straight" is not a keyer mode and bypasses this function.) In this function the "dit time" is calculated, where 12 WPM = 10 baud = dit = 100 milliseconds.
- cw_gen_remove_click_on_rising_edge – This function applies an envelope to the rising edge ("key down") portion of the audio CW waveform to go gradually from zero amplitude to full in a controlled manner. This "shaping" prevents key clicks on the rising edge.
- cw_gen_remove_click_on_falling_edge – This function applies an envelope to the falling edge ("key up") portion of the audio CW waveform to go gradually from full to zero amplitude in a controlled manner. This "shaping" prevents key clicks on the falling edge.
- cw_gen_check_keyer_state – Based on which of the paddles are pressed, or if the "reversal" mode is activated – sets the state machine to produce a dit or dah. This is not involved in the straight-key mode.
- cw_gen_process – This is the "high speed" function for generating CW as called by a function slaved to the main audio IRQ. Different portions of code are called depending on whether the mode is straight key or iambic paddle.
- cw_gen_process_strk – This function is used for the generation of straight key CW: If the key is released and the element has been generated, it is returned to receive mode. If the key has been pressed, the softdds function is called to generate a tone. If the key was *just* pressed, shaping is applied at the beginning but if the key was just released, shaping is applied at the end.
- cw_gen_procsess_iamb – This is a state machine that is called by "cw_gen_process" and the timing/sequence is very important to avoid improper operation. The state process is approximately as follows:
  - CW_IDLE: If nothing pressed, go back to receive mode. If either DIT or DAH is pressed, go to "CW Wait".
  - CW_WAIT: This is just a delay of one state in the machine, but it is absolutely necessary to prevent a "glitch" on the first DIT in a string of iambic elements as this gives other parts of the state machine/transceiver to synchronize/initialize. After this wait, "CW_DIT_CHECK" is called on the next invocation of this function.
  - CW_DIT_CHECK: If a DIT was pressed, set up the timers and, if necessary, enter tansmit mode and indicate a key-down condition and then bail-out. If not a dit, go to "CW_DAH_CHECK" next time.
  - CW_DAH_CHECK – If the DAH was pressed, set the timers appropriate for a DAH, indicate a key-down condition and then switch to transmit mode if necessary before bailing out. If nothing pressed, indicate a "CW_IDLE" condition.
  - CW_KEY_DOWN – If the state machine is called in this state (in the middle of generating a DIT or DAH) the softdds is called to generate the tone and the click is removed on the keading edge – and then the state is changed to "CW_KEY_UP"
  - CW_KEY_UP – If the timer for generating the element has NOT expired, generate more

    tone, removing the "click" at the end of the element. If the time has expired for the element, set to "CW_PAUSE" and bail out.
    - CW_PAUSE – If the element has completed, this checks to see if either dit or dah has been pressed to generate more and sets the state to "CW_DAH_CHECK" or it sets the state to "CW_IDLE" to go back to receive if nothing has been pressed.
- cw_gen_dah_IRQ – In Iambic mode, this switches to transmit mode if not disabled: In straight key mode, this sets the timers and other variables needed to generate a "clean" CW element. (Note that the CW "DAH" line is the same as the Straight Key and PTT lines.)
- cw_gen_dit_IRQ – This switches to transmit mode if not disabled.

The "filters" subdirectory:

This is where the filter coefficients are stored.

There are at least THREE types of filters used on the mcHF transciever:

- **IIR** – Infinite-Impules Response. These are the types of filters most generally used in the mcHF for ***post-demodulation*** audio filtering and are thus the main bandwidth-determining elements in the receiver. These types of filters are ***not*** unconditionally stable unless carefully designed, but they are used in the mcHF because they can be capable of better filtering with lower processor "horsepower" than an FIR with similar bandwidth characteristics.
- **Hilbert Transformer** – This is really an FIR filter with a bandpass (but mostly low-pass) response. In the mcHF the "Phase Added" variety is used accomplish a 0 and 90 degree audio phase shift to produce the I (In-Phase) and Q (Qudrature) channels, respectively. The low-pass response of the Hilbert is leveraged for use as anti-aliasing for the following decimation function, eliminating the need for filtering in that step and thus saving processor time.
- **FIR Lowpass –** For AM reception the I and Q (quadrature) audio channels are not needed and the FIR function that which would otherwise be the Hilbert transformer is simply a low-pass filter to set the AM detection bandwidth. In this instance both channels are identical and present a zero relative phase shift with respect to each other.

Two main tools are used to generate the filters used by the mcHF transceiver:

- **Iowa Hills Software** – This (free) suite has tools suitable for producing the needed FIR Hilbert transformers and AM Lowpass (FIR) filters. There are also tools suitable for producing (limited) IIR filters and general-purpose FIR filters as well as other things.
- **MatLab** – An add-on package (which includes "fdatool") is available that allows the development of digital filters and this is the best tool available for the generation of custom IIR filters that maximize the filter performance with the fewest mathematical operations and processor load and more importantly, verifying their stability!

As noted in the source code, the coeffients for the IIR filters must be derived to be of the ARMA type, and for both FIR and IIR elements used the coefficients ***MUST BE TIME-REVERSED*** – that is, the order of the coeffients must be back-to-front: A spreadsheet can be used to quickly "reverse" these coefficients and put them into a format that may be cut-and-pasted.

Note that the compiler makes it a bit tricky to include the "const" (constant) values that are the filter coefficients in multiple locations within the code, hence the avoidence of these filter coeffient files being utilized in multiple locations.

Phase adjustment:

In SDR implementations where CPU power is plentiful, on-the-fly transformations are applied to effect fractional-degree adjustments of phase. Because of the limited processing power available on the mcHF a "straight-line" approximation was utilized in which three "Q"-channel filters' coefficients were derived for the Hilbert transformations: One at 90 degrees, another at 89.5 degrees and a third at 90.5 degrees. Fractional phase adjustment was done via linear interpolation over a range of +/- 0.5 degrees, with the maximum error occurring at 89.75 and 90.25 degrees – but resulting degradation of the Hilbert transformers due this error was found to be negligable in this application. The "new" coefficients are calculated at the time of start-up (or adjustment of phase) and loaded into RAM where they are used by the DSP/Hilbert transformer.

Audio filter passband adjust:

Another feature of the mcHF is the ability to adjust the center frequency of the passband of various filters. To simplify matters this was done in a "brute force" manner and multiple sets of filter coefficients – each with a different center frequency – were generated.

If more "resources" are available, this may be done more elegantly: On-the-fly calculation of filter coefficients (particularly for FIR) or the use of frequency translation to effect a passband shift in a manner not unlike in an analog receiver, but both of these – particularly the latter – require much more processor power. In the case where we are using IIR filters – which ironically take less processor power to execute – it would be "code-prohibitive" to derive new filters of these types "on-the-fly while also verify that they were unconditionally stable.

Finally, a full explanation of the proper design and implementation of filters is beyond the scope of this document, although aspects are touched upon in the discussions of the various source code elements that use them.

Files in the main "audio" directory:

The file "audio_driver.c" contains the functions that initialize and process the receive and transmit audio. In the front part of this file it will be noticed that the audio filters are included here – but not the Hilbert transformers which are included in the "ui_driver.c" filter.

If one looks even briefly it will also be apparent that there are some "kludgy" work-arounds for a long-standing memory allocation problem in the form of the "test" variables (e.g. "test_a[]", "test_b[]", etc.) that grab large blocks of RAM without actually using it. For reasons yet unknown, failure to grab RAM in this manner will result in some of the audio-related buffers and variables overwriting each other and causing system crashes and/or the inability to boot up: ***Certain compiler optimization settings can also cause this as well!***

To be sure, the proper course of action would be to determine the actual cause of this anomaly and fix it, but many hours have already been spent by several people in a fruitless effort to erradicate this particular bug, so this is the "temporary" and surely ultimately unsatisfactory work-around!

Functions within "audio_driver.c":

- audio_driver_config_nco – This function (depricated) sets the constants used for the "on-the-fly" version of "audio_rx_freq_conv". The original function, while very flexible, took considerable processor time and was unsuitable for anything other than initial testing.
- audio_driver_init – As the name implies this initializes the audio drivers, particularly the variables associated with the "ads" structure containing audio-related variables and the "sm" structure related to the S-meter. Additionally, a number of additional functions are called to initialize the AGC, RF gain, ALC, noise blanker, and which sideband is used for CW mode. After this is done the hardware for the codec itself is initialized and the DMA/Interrupts are started.
- audio_driver_set_rx_audio_filter – This function uses the current audio filter setting (ts.filter_id) and loads the IIR filter coefficients for it. This function also determines the "offset" for the specific filter selected (e.g. center frequency) and loads the appropriate coefficients. Another task performed in this function is the initialization of the DSP noise reduction and notch filters along with the anti-aliasing filters their associated decimation/interpolation functions.
- audio_check_nr_dsp_state – This function determines if the coefficients of the DSP noise reduction are within bounds. ***It has been depricated.***
- Audio_Init – This function sets up the IIR filter used for transmit as well as calls the set up for the receive audio filter.
- audio_rx_noise_blanker – This function operates as a broadband impulse-type noise blanker. It works by "looking" at the entire (+/- 48 kHz) passband before the audio filtering and determining that "average" signal amplitude. If an impulse appears that exceeds this level this noise pulse is quashed – along with a few samples following this pulse – to remove it, preventing the following audio filters from "ringing" and exaggerating the pulse. The "blanking" adjustment, when increased, moves the threshold of detection closer to average signal level, making it more sensitive to pulses, but potentially triggering falsely on legitimate signals within the passband. Even though this sort of noise blanker is based, in general function, on broadband noise blankers found in many analog radios it turns out to be much less effective owing to the fact that there are distinct limitations as to the amount of pulse detection

and suppression that can be done on a temporally-limited signal that is already in the digital domain. It is because of this that this noise blanker is demonstrably inferior to its analog counterparts. This noise blanker algorithm also takes considerable processor resources to operate, hence blanker being disabled if in AM mode, if a "wide" filter is selected *(due to less decimation – see below)* or if "ts.nb_disable" is set to TRUE as done when the MENU mode is entered.

- audio_rx_freq_conv – This function performs on-they-fly spectral conversion of both the I and Q audio channels, offsetting the receive frequency from "zero". In the early part of this function is commented-out code that was originally used for testing that demonstrates its fundamental functioning and math, operating much like USB/LSB conversion in a sense. The "original" code is disused because it takes far too much processor power to be practical. The lower portion of this function contains code optimized for the mcHF and its operational environment. The first portion is a section of code that is called ***once*** that produces sine and cosine tables in RAM at angular rates that correspond to conversion that will yield an offset of 6 kHz from "zero" – a value chosen because its an integral divisor of the 48 kHz sample rate and "far enough" from zero to be useful. The final section is the actual frequency conversion: The code present in that section is functionally equivalent to the "old", commented-out code but represented using optimized ARM mathematical functions. The ultimate result is a frequency conversion that takes relatively little processor overhead, sacrificing only the flexibility of an easily-varied offset frequency. There are two versions of this function: One that mixes "high", requiring the local oscillator on the receiver to be tuned 6 kHz low, and that which mixes "low", requiring the local oscillator to be tuned 6 kHz high.
- audio_rx_agc_processor – This function is called by the main receive processor function ("audio_rx_processor") and it analyzes the receive signal and processes that in the audio buffer to adjust the audio gain to maintain an contant audio output despite a widely varying level. This function works by looking at the absolute amplitude of the signal and then adjusting "ads.agc_val" up or down, depending on whether or not the sample being examined is lower or higher than the normalized value expected. If the signal is stronger than expected, the gain (ads.agc_val) is reduced quickly, but if the signal is lower than expected the gain is increased – the rate of increase depending of the current AGC setting (e.g. slow, medium, fast.) A proportional calculation is done at the "decision point" (e.g. about the agc "knee" value) to prevent the inevitable low-level oscillation that would otherwise result. The progressive AGC value is stored in a buffer on a sample-by-sample basis (ads.agc_valbuf) and after the values are examined, the audio is delayed very slightly. To this slightly-delayed audio the stored values in agc buffer are applied, effecting a "look-ahead" techniques where the AGC is applied to the audio samples "before" they seem to occur, almost completely eliminating the problem of over/undershoot. The value of "ads.agc_val" is used elsewhere for an S-meter calculation as it represents the amount of gain applied to the signal path in AGC operation. Note that while the recovered audio level is used for AGC in SSB mode, a separate value ("ads.am_agc") is calculated and used for the AGC in AM demodulation. In FM modulation no audio processing is performed since it is not necessary.
- audio_demod_fm – This is the demodulator for Frequency Modulation. This is based on an optimized "atan2" fuction originally described by Jim Shima and placed in the public domain and is MUCH faster than the normal library ATAN2 function. In short, this calculates the angular change of the current sample pair (I, Q) with respect to the previous and outputs the difference. Because FM is a form of angle modulation, the result of this change is, in fact audio. Because the input to the ATAN2 function is a ratio, the absolute amplitude is irrelevant and "limiting" is not required. After the demodulation, additional functions are performed:

    - Integration is performed on the demodulated audio with a "knee" set at a few hundred Hz to fit the de-emphasis curve used in amateur radio communications (e.g. -6dB/octave). The audio from this is stored in the "c" buffer for possible tone detection and to the differentiator.
    - If the squelch is open (by carrier squelch or tone being detected) the audio is differentiated to function as a high-pass filter, but this has a "knee" of a few hundred Hz – approximately the same as the knee of the integrator. The result of this functions as a sort of filter for the subaudible tone decoder, but it also removes the low-frequency "rumble", that would be the inevitable result of the integration, that could cause saturation of the audio amplifier and/or distortion in the speaker.
    - After this the magnitude of one sample of the I/Q data is taken and used to measure the carrier level. This value is passed to the AGC processor, but in FM the AGC is NOT operating on the signal, but is used ONLY for the S-meter.
    - Audio from the "b" buffer, taken prior to the integration, is applied to an IIR high-pass filter with an approximate 15 kHz cut-off. This is used to remove the modulation and leave only the triangle noise for the squelch detector.
    - The absolute value of one (representative) sample from the output of the high-pass "squelch noise" filter is square-rooted and applied to an IIR low-pass filter.
    - The next section is triggered occasionally by the variable "count", taking the "running average" from the IIR low-pass of the squelch energy. The squelch energy from the IIR is "clamped" at a maximum level to keep it from going extremely high during high-noise (e.g. no signal) conditions which would cause it to act slowly when a signal appears. This clamped noise signal is then rescaled, limited again and then "inverted" so that a smaller number now represents a noisier signal. This "cooked" noise level is then compared with the squelch setting, with hysteresis, and used to open/close the squelch. The flag "ads.fm_squelched" is set if the squelch is closed and is used to signal this condition. When the audio is to be muted due to squelch closure and/or the lack of tone detection the audio buffer is "zeroed out" to to mute it.
    - The final section is for the detection of subaudible tones and it uses three single-frequency Goertzel tone detectors: One at the desired frequency, one slightly above, and another slightly below. The amplitude output of the on frequency is divided by the average of the two "off-frequency" detectors to yield a ratio that is independent of the amplitude of the detected signal. This ratio is IIR low-pass filtered to reduce the variation and this result is then compared with a fixed threshold. The output of this threshold comparison is applied to a debounce algorithm that is slightly biased toward the optimistic with a bit of hysteresis – that is, it will be more inclined to detect a tone and maintain detection, this to minimize "bouncing" in the presence of noisy signals. The flag "ads.fm_subaudible_tone_detected" is used to signal the presence of a tone.
    - Note: Within the function "audio_rx_processor" decimation/interpolation is disabled when FM demodulation is active since no audio filtering or DSP is done.
- audio_demod_am – This is the AM demodulator. This has been rewritten and streamlined as of 0.0.219.25. First, the I/Q data is interleaved in the "b" audio buffer and once in this format, the optimized ARM function is used to calculate complex magnitude – which is the audio. For the AGC, the mean of the demodulated audio is taken and rescaled for proper S-meter calibration.
- audio_lms_notch_filter – This is the DSP notch filter and it uses a "normalized" LMS (Least Mean Squares) function to produce the coefficients for an FIR filter – or putting it another way, if there is coherent spectral energy (say, a "tuner-upper" or the carrier of an AM broadcast station) within the passband, the LMS function will cause the filter coeffients to be generated

such that it will produce a filter that is optimized to pass that spectral energy. Because this is exactly what we *don't* want we utilize the "error" output of the LMS function which contains everything *but* that which is to be filtered, and then throw away the rest. It should be noted that in order for the LMS function to work it needs a "de-correlated" input that is representative of a signal with noise, and *not* containing the "desired" signal. Since this is impractical this is simulated by delaying the input signal slightly to make it look different and use it for the reference. *(Yes, it really works!)*

- audio_lms_noise_reduction – This is the DSP noise reduction and it works very much like the DSP notch filter: The LMS function detects "coherent" energy (voice) and quickly adapts an FIR filter to conform to it – except in this case we are keeping the filtered output instead of throwing it away as in the case of the notch filter, where we keep everything *but* the filtered output. As it turns out, the vowels of the human voice consists of discrete tonal frequencies and their harmonics and these can be generally detected by the LMS function: Noise, on the other hand, is completely random and the LMS function never "locks on". Because voice is comparatively slow-changing, and most peoples' voices don't change pitch extremely quickly it is possible to pick the LMS parameters so that it can react to voice fairly quickly. The main difference between the function of the LMS in the notch and noise reduction is that for the notch, it operates very slowly, by comparison since an interfering tone stays present for a long time whereas a voice changes fairly quickly. One can increase the "strength" of the DSP noise reduction by slowing the response and somewhat reduce the "hollow barrel" sound on the noise but it makes it slower to adapt to changing voice characteristics.
    - Contained within the DSP noise reduction processing is an algorithm that detects if the audio output of the DSP filter goes too low or too high, indicating that that it has crashed. If this state is detected some variables are used to detect this so that the DSP may be automatically reset (e.g. "ads.dsp_zero_count" and "ads.dsp_nr_sample").
- audio_rx_processor – This is the main audio processor.
    - First, "psize" is the sample size of the internally-processed audio data – after decimation.
    - Before anything else is done, the noise blanker is called as this must operate on audio samples before any filtering.
    - The next function is that which accumulates audio samples for the spectrum scope/waterfall display: Once the necessary number of samples have been accumulated, further acquisition of samples is halted until after the buffer is processed and the results displayed.
    - The next step, performed concurrently with accumulating samples for the spectral display, is to take the interleaved 16 bit audio samples and convert them to a pair of floating-point audio buffers for further audio processing. Within this process the absolute value of the A/D samples is examined and compared fractionally to the constant "ADC_CLIP_WARN_THRESHOLD" at three different levels (/4, /2 and at that level) and if the sample exceeds that value the appropriate flag (ads.adc_quarter_clip, ads.adc_half_clip, ads.adc_clip) is set: These flags are used elsewhere to set the input gain fo the codec to maximize dynamic range and to provide and on-screen indication of high signal levels.
    - The next step is to apply corrections for I/Q gain balancing: The audio buffers are multiplied by equal-and-opposite values to compensate for slight differences in the gains of the I and Q receive channels.
    - The next step is the frequency conversion. If enabled, the a flag is passed to indicate whether or not a conversion is to be "high" or "low".
    - For SSB reception the Hibert transformer comes next: This uses an 89-tap FIR "phase-

added" where the "I" channel is set to 0 degrees and the "Q" channel is set to approximately 90 degrees: The use of 0, 90 degrees rather than -45,+45 was done (initially) because the mcHF originally used a "zero" offset receiver (e.g. no frequency translation) and only the 0, 90 degree Hilbert transformations offer a resonable degree of performance at low audio frequencies with an acceptable number of taps. As noted in the section about the "filter" subdirectory, the "90 degree" filter coefficients are linearly interpolated amongst three filters (89.5, 90.0, 90.5) to produce a fractional-degree adjustment of the "Q" channel's angular offset, minimizing processor overhead. These Hilbert transformers are also used for low-pass filtering of the audio prior to the decimation, eliminating the need for strong anti-aliasing filters in that step.

- For AM reception this same 89-tap FIR is used simply as a low-pass filter, the cut-off frequency setting the effective front-end bandwidth (e.g. 5 kHz low-pass = +/- 5 kHz bandwidth, or 10 kHz total bandwidth.)
- The next step is demodulation using optimized ARM functions:
  - LSB is the difference of the I and Q channels
  - USB is the sum of the I and Q channels
  - AM is the square root of the sum of the squares of the I and Q channels. For AM, a "mean" value is calculated and used for carrier-based AM AGC.
- [IF FM is NOT enabled] At this point it is time to do decimation which involves "throwing away" most of the audio samples: We can do this because Nyquist tells us that if our highest audio frequency is less than half of our sample rate, we can adequately represent that frequency. Having already designed the Hilbert Transformer as a suitable anti-aliasing filter we need not really do any low-pass filtering here at all, but since the Keil DSP function requires at least a minimum, the simplest-possible (and most effective, given the number of taps) FIR low-pass is implemented within the decimation. For the filters 3.6 kHz and narrow decimation-by-four is implemented to reduce the 48 kHz sample rate to just 12 kHz: For "wide" filters the decimation is by two, reducing the sample rate to 24 kHz. In either case, the amount of "number crunching" is significantly reduced!
- [IF FM is NOT enabled] At this point, the LMS notch filtering function is called, if enabled.
- [IF FM is NOT enabled] At this point the LMS noise reduction function is called (the "pre-AGC version), if enabled.
  - Note: There are actually two DSP noise reduction algorithms: One before the AGC, and one after – but only one is enabled at a time. If the one before the AGC is enabled the S-meter reading is affected by the DSP noise reduction since it "sees" less signal, but the audio output to the speaker remains more or less constant.
- [IF FM is NOT enabled] Audio filtering is the next to be applied. As noted in the "filters" section, above, these are IIR-based using time-reversed ARMA lattice coefficients.
- The next to be called is the AGC, described in the previous section.
- [IF FM is NOT enabled] As noted before, the DSP noise reduction may be used either before or after AGC: In the latter case it is applied now, along with "crash" detection.
- Now, audio gain scaling is calculated. This is necessary because the decimation/interpolation process has the effect of reducing the audio, so this "fixes" the level so that it remains constant. Another scaling factor that is applied is that related to AM demodulation and the values are chosen so that the output level of a 1 kHz CW note and an AM carrier 100% modulated with a 1 kHz tone are the same. Once the constants are derived, the audio buffer is multiplied by that scaling factor.
- [IF FM is NOT enabled] The next step is the interpolation to return the audio sample rate to

48 kHz. As with the decimation, this includes low-pass filtering to remove aliasing effect from the previous, lower sample rate. Unlike with the decimation where we can "pre-filter" with the Hilbert transformer, we must actually make some effort to get rid of the aliasing: Unlike the receive case, we need only reduce the level of this aliasing enough to prevent it from being "annoying" and thus the filtering was designed to be as minimal as possible to reduce processor loading, but also to reducing the unwanted energy by at least 25 dB, making it essentially inaudible for audio filters 3.6 kHz and narrower: For the "wide" filter, the alias frequency is higher and not as well reduced – but you have to have both good ears and good speakers to hear it!

- The next function performs two functions:
  - A "muting" by filling the audio buffer with zeros: This is different than muting the speaker in that it purges the buffer and doesn't require messing with the codec which can sometimes be... well, messy...
  - If the audio isn't muted, the next portion does the audio gain control. The "LINE OUT" level is fixed here while the speaker volume is adjusted in software: For audio levels from 0-16 there is no "software" amplification and only the hardware (codec) is used. For values above 16 the variable "ts.audio_gain_active" is used to "amplify" the audio in software.
  - The final step is to transfer the floating-point audio data to the DMA buffer, converting it to 16 bit integer data for the D/A converter in the process.
- audio_dv_rx_processor – This is a stripped-down receive demodulator function with no DSP or noise blanker capability or audio filtering, relying only the low-pass filtering of the Hilbert transformer – all to reduce processor overhead to an absolute minimum. As of firmware version 0.0.219.22 it is incomplete, lacking the necessary decimation/interpolation for the sample rate required for typical digital-mode modules.
- audio_tx_compressor – This functions as both the ALC and transmit speech processor, working in the same way as the receive AGC (audio_rx_agc_processor), using "look-ahead" processing with the variable "ads.alc_val" is used to display the "ALC" parameter on the main display and instead of the AGC mode, the parameter "ALC Release Time" coupled with the audio gain applied in front of this module is used to adjust the amount of compression. In the case of the ALC, no gain is applied in the process – only attenuation as required to prevent the audio from ever exceeding a pre-set level. It is this function that places an absolute limit on the amplitude of the audio that may be applied to the transmitter's modulator(s). It is worth noting that the audio compression is based only on one audio channel, but the gain correction (compression) is applied to both channels in quadrature.
- audio_tx_processor – This function generates the audio that is applied in quadrature to the modulator with various sub-modules being used for the different transmit modes:
  - Tune mode: Using the "softdds" this generates an audio tone in quadrature, rescaling it for the power setting and band-gain ("ts.tx_power_factor") and then loads this audio information into the DMA buffer, converting to 16 bit data: Which of the generated quadrature audio tones is placed into the transmit I or Q output D/A channel will dictate whether or not the generated RF is above or below the carrier frequency (USB/LSB.) In this TUNE mode, frequency translate is disabled.
  - In CW mode, if "cw_gen_process" returns a NULL, the audio buffer is filled with zeros to mute transmitter output. If a CW element is to be generated the quadrature in the audio buffer its amplitude is rescaled according to the power setting and band-gain and copied/converted to the DMA buffer. In CW transmit mode, frequency translate is

disabled.

- In USB/LSB mode:
  - If "tune" mode is active while in an SSB mode, an audio carrier is generated in the same way as the TUNE mode, above.
  - If not in TUNE mode audio is copied from the A/D DMA buffer to the floating-point audio buffer.
  - The transmit audio gain parameter is then determined (whether LINE or Microphone input).
  - The absolute (peak) value and this value is made available to be displayed as the "AUDio" meter with the variable "ads.peak_audio".
  - If enabled, the transmit audio is shaped with an IIR filter that has been specially tailored to provided a (nearly) brick-wall, flat response over the 275-2500 Hz frequency range with special emphasis at the lower end to compensate for some of the (minor) low-frequency deficiencies of the "Phase-Added" Hilbert transformer.
  - At this point in the process the Hibert transformation is performed, generating the I and Q audio channels. As with the receive processor, the phase of the "Q" channel is fractionally adjustable to provide precise phase adjustment. The Hilbert transformer also provides a degree of audio filtering as well.
  - The "pre-compression" gain is applied to both the I and Q channels: The higher the gain, the more opportunity there is to reduce the gain and "level" the audio to reduce the peak-to-average ratio.
  - The next step is the audio compression. Because this step follows the audio filtering, it reflects the audio content to be applied to the transmitter: If the compression had been placed prior to the transmit audio filtering it would to frequency content that was not being transmited such as very low (fundamental) voice/vowel frequencies and sillbance (consonants).
  - If frequency conversion is to be enabled a logic matrix is applied along with mode flag (LSB/USB) to assure that the proper sideband is transmitted.
  - The audio is then scaled according the transmit power setting and band-gain.
  - Finally, the floating-point audio buffer is copied to the DMA buffer, the order determining whether USB or LSB energy is generated – and converting to 16 bit audio data for the D/A converter, to be applied to the modulator circuityry.
- In AM transmit mode:
  - The audio data is converted from 16 bit A/D format to floating point and placed into the audio buffer.
  - As with the SSB audio the gain adjustments for the Mic/LINE are performed.
  - The peak/absolute values within the audio buffer are determined and made available as variable "ads.peak_audio" for the "AUDio" meter on the main display.
  - If it is enabled, the same "brick wall" transmit audio filter (275-2500 Hz) is applied to the transmit audio.
  - The transmit Hilbert transformation is applied, both producing the quadrature (I/Q) channels and providing a degree of filtering, applied if the "brick wall" filter, above is disabled.
  - The "pre-compression" gain is applied to both the I and Q channels: The higher the gain, the more opportunity there is to reduce the gain and "level" the audio to reduce the peak-to-average ratio.
  - The next step is the audio compression. Because this step follows the audio

filtering, it reflects the audio content to be applied to the transmitter: If the compression had been placed prior to the transmit audio filtering it would to frequency content that was not being transmited such as very low (fundamental) voice/vowel frequencies and silibance (consonants).

    - At this point a duplicate of the I and Q channels is created as for AM, a separate LSB and USB signal must be generated.
    - To generate the LSB AM signal using the original copy of the I and Q channels a differential bias is added, effectively injecting the carrier, producing a reduced-carrier single-sideband AM signal.
    - Frequency translation is the next step, shifting the AM signal by 6 kHz from "zero" – a necessary step with the hardware since there is a "dead spot" at that frequency.
    - The translated energy is then scaled according to the transmit power level, band-gain and a fixed constant to set the overal output power level.
    - Finally, the quadrature signal is stored in the DMA buffer, having been converted back to 16 bit format.
    - The I and Q channels previous copied are now restored to produce the USB AM signal: The same steps are performed as for the LSB signal except that the I and Q buffers are swapped.
    - At the end of the generation the USB AM data is simply added to the already-stored LSB data in the DMA buffer to produce full-carrier, double-sideband AM.

- In FM transmit mode:
    - The audio data is converted from 16 bit A/D format to floating point and placed into the audio buffer.
    - As with the SSB audio the gain adjustments for the Mic/LINE are performed.
    - The peak/absolute values within the audio buffer are determined and made available as variable "ads.peak_audio" for the "AUDio" meter on the main display.
    - A transmit audio filter (215-2700 Hz) is applied to limit the range of frequencies that may be applied to the modulator to limit bandwidth.
    - The "pre-compression" gain is applied to both the I and Q channels: The higher the gain, the more opportunity there is to reduce the gain and "level" the audio to reduce the peak-to-average ratio.
    - The next step is the audio compression. Because this step follows the audio filtering, it reflects the audio content to be applied to the transmitter: If the compression had been placed prior to the transmit audio filtering it would to frequency content that was not being transmited such as very low (fundamental) voice/vowel frequencies and silibance (consonants). At this point the audio level is strongly limited to a maximum value.
    - The audio is now differentiated to apply a 6dB/octave pre-emphasis as is standard for amateur FM communications.
    - If it is enabled, DDS techniques are used to generated a subaudible tone which is summed with the post pre-emphasis audio.
    - If it enabled by the timer, a tone burst (to activate repeaters, etc.) is generated using DDS techniques, summed with post pre-emphasis audio. The subaudible tone generation is disabled while the tone burst is being generated.
    - The audio (with subaudible tone or tone burst) is then applied to the "frequency word" of a DDS oscillator (NCO) with quadrature outputs that is operating at 6 kHz to produce an FM signal.

        - The quadrature output from the "FMed" DDS is scaled according to the FM TX I/Q phase adjustment and band-power setting.
        - Finally, the quadrature signal is stored in the DMA buffer, having been converted back to 16 bit format. The order (which sample comes from which buffer) is selected according to the "frequency translate" mode so that the transmitted signal is shifted above or below the LO as needed.
    - audio_dv_tx_processor - This is a stripped-down transmit modulator function with no compression or added audio filtering, relying only the low-pass filtering of the Hilbert transformer – all to reduce processor overhead to an absolute minimum. As of firmware version 0.0.219.22 it is incomplete, lacking the necessary decimation/interpolation for the sample rate required for typical digital-mode modules.
    - I2S_RX_CallBack – This is the main DMA function that, depending on TX or RX mode, will call either the RX or TX processors and is called at approximately 375 Hz. When switching between transmit and receive this function includes code to purge the audio buffers (fill with zeroes) to prevent/minimize a "click" from transmit/data that it had previously contained. This function also contains timing functions for timing the TX-to-RX delay, the debounce of keys, the generation of the PWM signal for dimming of the LCD backlight, the generation of a 10 millisecond (100 Hz) master timer for sequencing events and finally, update of a timer for scheduling the occurrance of the spectrum scope/waterfall update.

Important definitions in "audio_driver.h":

This file contains the definitions for the structure that contains the majority of the audio-related variables (ads.xxx) as well as those used for the Spectrum and Waterfall Displays (sd.xxx) and S-meter (sm.xxx).

This file also contains the many constants used in the AGC and RX/TX audio gain processing: These should ***NOT*** be adjusted unless one has a completely thorough understanding of the algorigthms involved and has also done a gain-block analysis of the affected stages: For example, a modfication of an RX gain and/or AGC value can wreck both the AGC and S-meter calibration unless other, related values are compensated! Constants related to the ALC processing and calibration of the various audio/carrier levels related to SSB and AM transmit are also included, along with DSP system parameters – generally the the defaults and GUI-related parameters.

Files in the "ui" directory:

As you might expect, the "UI" directory contains files that are related to the UI (User Interface), but also additional hardware elements directly/indirectly related to the user interface.

The "encoder" subdirectory:

The two files "ui_rotary.c" and "ui_rotary.h" contain the hardware definitions for the four rotary encoders used on the mcHF front panel.

The "oscillator" subdirectory:

The file "ui_si570.c" contains the low-level drivers for the frequency synthesizer. These functions take the inputted frequency, which is 4x the receive/transmit frequency, and calculate the register values for the Si570 to set it to frequency.

It should be noted that there are two methods of adjusting the output frequency of the Si570:

- "Fine Tuning". This is a limited-range tuning method that provides a smooth adjustment of the output frequency. At HF frequencies it has a maximum "swing" of only a few hundred kHz before a "Coarse Tuning" is required.
- "Coarse Tuning". This requires a major reconfiguration of the internal registers of the Si570 and causes a brief (several milliseconds) interruption in the output when it occur. From the calculated frequency of the "Coarse Tuning" one can "Fine Tune" up and down a short range without interruption.

Recently, additional code (e.g. the parameter "test" in the function "ui_si570_change_frequency") that does a "faux" frequency change (e.g. does not actually change the frequency) but, instead, returns a flag to indicate of a "coarse" retune is required by the new tuning frequency. If a coarse retune is required it will then be known that the local oscillator output will be muted, causing a loud "click" and steps may be taken to prevent this (e.g. halt audio processes temporarily) from appearing as an annoying artifact – particularly if DSP noise reduction enabled and/or a narrow audio filter is switched in – either of which can exacerbate strong impulse noise responses owing to the nature of those elements.

At the moment the code is written such that the "coarse tune" bondaries are fixed which means that one can tune back-and-forth over a particular frequency and repeatedly force a "coarse tune" event to occur. Because of the aforemention facility, an audible "click" has been supressed, but a very brief, audible muting does occur. In theory it should be possible to make the placement of these "coarse tune boundaries" a bit more "intelligent" such that they never occur if one moves back-and-forth over just a few kHz, but rather slightly "elastic" (e.g. hysteresis added in their locations) to avoid this (admittedly minor) phenomenon.

The file "ui_soft_tcxo.h" contains the frequency-to-temperature dependencies for a typical Si570 with each entry corresponding to the frequency offset, in Hz – referenced to 14.000 Mhz – at a particular temperature reported on the main display. These values are typical for an average "AT" cut quartz element and should generally fit the "lazy 'S' curve" exhibited by most Si570 units. It should be noted that between the 1 C entries in the table the calibration values are interpolated.

If you modify the table in "ui_soft_tcxo.h" for your own purposes, be certain that you DO NOT cause it to be distributed in future releases of the code! If you modify this table, DO NOT leave any values blank, but rather interpolate/extrapolate the missing values.

The "misc" subdirectory:

This contains the small file "hamm_wnd.h". I have no idea what this is for.

The "lcd" subdirectory:

This contains the files that relate text and graphical presentation on the LCD, including the drivers, fonts and colors.

The file "ui_lcd_hy26.c" contains the low and medium-level drivers for the LCDs and the various functions are described as follows:

- UiLcdHy28_Delay – This performs a delay that might be used during initialization to allow hardware to "settle".
- UiLcdHy28_BacklightInit – This configures pin PD2 for the control of the LCD backlight.
- UiLcdHy28_SpiInit – This configures the MCU I/O hardware to interface with the LCD with an SPI interface at the highest-possible data rate. *(This has been emperically verified!)*
- UiLcdHy28_SpiDeInit – This deconfigures the MCU I/O hardware's SPI hardware if it had been previously configured for and SPI interface. This may be done if an SPI interface to the LCD was *not* detected.
- UiLcdHy28_ParallelInit – This configures the MCU I/O hardware for a 16 bit wide, bidirectional parallel interface to the LCD at the highest possible data rate.
- UiLcdHy28_Reset – This forces a complete hardware reset of the LCD module.
- UiLcdHy28_FSMCConfig – This configures the internal memory manager of the MCU to map the LCD's display RAM as internal memory, implementing WAIT states, etc. as needed to conform to the timing limitations.
- UiLcdHy28_SendByteSpi – This sents a single byte via the SPI to the LCD.
- UiLcdHy28_ReadByteSpi – This reads a single byte via the SPI interface from the LCD.
- UiLcdHy28_WriteIndexSpi – This addresses the index register of the Hy28 LCD. (See data sheet).
- UiLcdHy28_WriteDataSpi – This writes 16 bits of data to the LCD via the SPI interface – appropriate for color/pixel information because the RGB data is stored in packed (5,6,5) format.
- UiLcdHy28_WriteDataSpiStart – This sets up the LCD to begin writing of data.
- UiLcdHy28_WriteDataOnly – This writes 16 bit data to the LCD, using either parallel or SPI format, depending on which interface is being used.
- UiLcdHy28_ReadDataSpi – This reads 16 bits of data from the LCD via the SPI interface.
- UiLcdHy28_WriteReg – This writes the value to the specified LCD register using either parallel or SPI format, depending on which interface is being used.
- UiLcdHy28_ReadReg – This reads the value from the specified LCD register using either parallel or SPI format, depending on which interface is being used.
- UiLcdHy28_SetCursorA – This sets the X, Y coordintes of the LCD's cursor, irrespective of the interface format.
- UiLcdHy28_LcdClear – This clears the LCD, filling it with the specified color, irrespective of the interface format.
- UiLcdHy28_SetPoint – This puts a point on the LCD at the specified coordinates, using the specified color, irrespective of interface format. This is a somewhat slower method of accessing a particular pixel.
- UiLcdHy28_WriteRAM_Prepare – This configures the LCD to write to display memory, irrespective of interface format.
- UiLcdHy28_DrawColorPoint – This puts a point on the LCD at the specified coordintes, using the specified color, irrespective of interface format. This is slightly faster than

"UiLcdHy28_SetPoint.

- UiLcdHy28_DrawStraightLine – This draws a straight line from coordinate x, y of specified length and color. If "direction" is TRUE, the line is vertical, otherwise horizontal. It ALWAYS draws in the direction of INCREASING coordinates. This command operates irrespective of interface format.
- UiLcdHy28_DrawHorizLineWithGrad – This draws a horizontal line starting at x, y of specified length with a gradient, the position of which is specified. The resulting line is brightest in the middle. This command operates irrespective of interface format.
- UiLcdHy28_DrawEmptyRect – This draws an empty box at x, y of specified height and width and color using single pixel-width lines. This command operates irrespective of interface format.
- UiLcdHy28_DrawBottomButton – This draws the bottom portion of a button at the specified x, y coordintes of defined height and width and color. This command operates irrespective of interface format.
- UiLcdHy28_DrawFullRect – This draws a filled rectangle at the specified x, y coordinates of defined height width and color. This command operates irrespective of interface format. Note that in SPI mode this uses a hardware-defined window and "blind writes" to minimize the amount of data to be written.
- UiLcdHy28_OpenBulk – This configures the LCD for a "bulk write" into a hardware-defined sub-window on the LCD at position x, y and specified width and height. Further writes to the LCD need only consist of color information, but must be pixel-sequential: The elimination of individual coordinate information for each pixel greatly acellerates the throughput, particularly in SPI mode. This command operates irrespective of interface format.
- UiLcdHy28_BulkWrite – After the "UiLcdHy28_OpenBulk" command (above) has been issued, this command is used to sequentially fill the area defined with colors. Note that the data itself must be of the proper row/column format in order for this data to be represented properly on the display. This command operates irrespective of interface format.
- UiLcdHy28_CloseBulkWrite – This command ends the "bulk write" and removes the hardware-defined "sub window" on the display. After having invoked the "OpenBulk" function his command MUST be issued before "normal" LCD screen writes may resume! This command operates irrespective of interface format.
- UiLcdHy28_DrawChar – This places the specified character/symbol at position x, y using the chosen foreground and background colors: Not also that a pointer to the font table must be included. It is not intended that this function be called directly, but rather by "UiLcdHy28_PrintText", below. This function does "blind writes" to display memory in defined sub-windows to maximize speed – particularly in SPI mode. This command operates irrespective of interface format.
- UiLcdHy28_PrintText – This prints the text pointed to by the string at the coordintes x, y using the specified foreground and background colors and using the desired font. There are five fonts available: A 16x24, 12x12, 8x12, 8x8, and 8x12 bold font. This command operates irrespective of interface format.
- UiLcdHy28_DrawSpectrum – DEPRICATED. This function updates ONE HALF of the spectrum scope display, using the amplitude data passed to it, using the drawing color specified.
- UiLcdHy28_DrawSpectrum_Interleaved – This is an optimized version of the function that updates the Spectrum Display, using both the "new" (current) data and "old" (previous) data. The spectrum data is represented by a series of vertical lines, the height of each representing the amplitude at that frequency, and as such it is often the case that from one update to the next, the majority of this vertical line will remain unchanged – each one being somewhat shorter or

longer than before. Because of this, it is only the ***difference*** that is updated, rather than the entire line which greatly reduces the amount of data that is required to update the spectrum scope's display – particularly important when using an SPI interface. In addition to updating the vertical lines representing the signals, the graticules must also be repainted as necessary as well as the optional vertical line that represents the display frequency to which the receiver is tuned. This command operates irrespective of interface format.

- UiLcdHy28_InitA – This function attempts to initializes the various vertions of the HY28 LCE modules (A, B, SPI, Parallel). If it detects that the parallel interface exists, it always prefers it, instead. This function is used by "UiLcdHy28_Init", below.
- UiLcdHy28_Test – This is a generic test function used for debugging – normally not used.
- UiLcdHy28_Init – This does the search for the various LCD types. The result made available as the system variable "sd.use_spi" which is TRUE if the SPI interface is to be utilized: This is important because there are a number of dependent functions that require known which interface is being used as the methods/settings/timings can vary.
- UiLcdHy28_ShowStartUpScreen – This shows the start-up "splash" screen which includes version number, attribution, and various bits of configuration information and warnings – including those related to the inability to access the Virtual EEPROM and if the mcHF has been configured NOT to use a "Frequency Translate" mode.

The file "ui_lcd_hy28.h" contains, amonst other items, the definitions for the various colors used. As noted above, the RGB information is represented using 16 bits in 5, 6, 5 format (64k colors). This file also contains some definitions related to the spectrum width (e.g. used for the "block" write) as well as definitions for RAM and hardware allocations for the various LCD types – plus the externally-available definitions of the various functions.

The file "ui_lcd_hy28_fonts.c" contain the bit maps of the various fonts defined. Refer also to the files contained in the subdirectory "fonts" for further attribution and information on how these fonts were developed.

The file "waterfall_colours.h" contain the color palettes for the waterfall display. Each of these palettes is defined such that location [0] is the weakest-possible signal and [63] is strongest with a 16 bit RGB value (in 5, 6, 5 format) being defined for each. Note that there is, in fact, a 65th entry in the table that is reserved for the vertical marker line, typically filled with "0xff" for each color. Additional waterfall palettes may be added to this file provided that they follow the defined format and are added to supporting code, elsewhere.

Main "ui" directory:

The "ui_eeprom.c" and "ui_eeprom.h" files:

The files "ui_eeprom.c" and "ui_eeprom.h" are leftovers from an attempt to separate the Virtual EEPROM read/write functions from the "ui_driver" files. Unfortunately issues with multiple definitions of CONST variables complicated matters and this effort was abandoned for now. ***NOTE: These files are not complete as many EEPROM variables have been added/changed: They should be used only as a representation of what was attempted!***

The "ui_driver.c" file:

Aside from the core DSP files related to "audio_driver.c", the "ui_driver.c" file contains the most important and complex functions in the entirety of the mcHF code, comprising nearly all aspects of the user interface and DSP control.

Please note: There are a huge number of strong interdependencies contained within this file and it is (unfortunately) very easy to "break" things by accident! Be certain that you keep a progressive backup of your various modifications so that you can identify at which point you "broke" something – and then have some hope of figure out what it was and then fix it!

It is in this file that the various "front end" filters are defined, namely the Hilbert and "Not-Hilbert Lowpass filtering" (the latter used only for AM demodulation) as the relevant coefficients are loaded in to RAM and modified on-the-fly by the user interface.

This file also contains the reads from and writes to the Virtual EEPROM that are used to restore and save user-configurable options to nonvolatile memory.

Tuning steps, Band Definitions and Band sizes:

Several CONST arrays are defined that declare the tuning step sizes along with the starts (bottoms) fo the various amateur bands and their defined sizes: The actual definitions are elsewhere. Note that there are provisions for a "general purpose" band (not yet used) that is intended to store "non-ham" frequencies (e.g. general coverage.)

S-meter calibration table:

The table "S_Meter_Cal" correlates each of the S-meter pixels (dots) with a value from the AGC subsystem. It is this table that is used to calibration the S-meter to the industry IARU standard (e.g. S-9 = 50 microvolts into 50 ohms) and modification of this table should NOT be done unless you thoroughly understand all aspects of the AGC system!

Functions in detail:

- ui_driver_init – This function calls the various initialization functions, reads stored EEPROM values and some of the utility functions used to set up the hardware – during power-up of the radio as well as loading some initial defaults. Once the various initializations have been perofrmed the desktop is then drawn, the softdds is configured, the audio volume control is set to an initial (but not necessarily correct) state, the band-gain values and TX/RX state machines initialized, and the frequency display updated. If something must be done at startup, it probably needs to be done here – but beware the order!

- ui_driver_thread – This is the main "round-robin" function that gets everything related to the UI done with the first order of business being the servicing of the Waterfall or Spectrum Display state machines. *It is important to note that if the main state machine is not serviced frequently enough, the user interface itself will become sluggish: Button-presses will become less-responsive and the adjustments of knobs will have apparent lags. On the other hand, servicing the state machine too frequently can give an erratic appearance to the Waterfall or Spectrum Scope in its update rate.*
- ui_driver_irq – This function has been depricated as it could result in function re-entry and memory corruption if the task had not completed before it was called again! It has been replaced by "ui_driver_thread", above.
- ui_driver_toggle_tx – **This is one of those functions that, if messed with without completely understanding all aspects of the hardware and software, can completely screw up the operation of the entire transceiver.** Based on the state of the "ts.txrx_mode" flag this will switch the hardware between transmit and receive modes, ***properly*** sequencing the hardware tasks required to do this – the precise procedure depending on the operational mode (CW versus voice.) This function also enforces the maximum power limit in AM mode, configures the bias for the PA stage – and disables it when in receive to minimize quiescent power consumption – and assure proper configuration of the "softdds" when in CW mode. In addition, it makes sure that the proper VFO is updated to the synthesizer and display (VFO A/B, Split) as well as handles the frequency control related to RIT and other necessary frequency shifts – and a few other things as well.
- UiDriverPublicsInit – This is part of the initialization for variables related to the UI, such as button presses, encoders, SWR/Power metering and the voltmeter.
- UiDriverProcessKeyboard – This function is the main handler for keyboard presses and the changing of modes/functions related to those buttons. The first half is for the "brief" presses of SINGLE buttons – EXCEPT for buttons F1-F5. Note that each button function is multiply-qualified, depending on whether menu mode is active, whether in TX or RX, and also based on the specific dependencies related to those functions. For example, changing the operational mode may have implications related to the displayed frequency, the DSP functions have dependencies based on the currently-selected mode, etc. The second half of this function are for those "press-and-hold" functions and these may involve one ore MORE buttons. Note that for any function involving multiple buttons being held down, the same qualification/operation must be replicated for each possible entry of the button as it cannot be known which button will be pressed first at the instant the multiple buttons are pressed!
- UiInitRxParms – This function calls the dependent functions necessary to update/change the mode. This is used when changing band, filter or mode as MANY parameters may need to be modified in that instance.
- UiDriverPressHoldStep – This function is called by UiDriverProcessKeyboard and it *temporarily* changes the step size while the STEP- or STEP+ button is being held down to facilitate the tuning, and the updates the step size display.
- UiDriverProcessActiveFilterScan – This function utilizes a state machine to determine which filters are active, as selected in the menu system, and appropriate for the selected mode.

Because the number of available filter may vary depending on mode and configuration, it is necessary to utilize a somewhat complicated function to reliably determine the next-available filter and skip to/over filters as needed.

- UiDriverProcessFunctionKeyClick – This is the function that processes the F1-F5 "short press" actions: The "long-press" functions involving these keys are handled in "UiDriverProcessKeyboard". As with the aforementioned function, there are many dependencies with these key actions, depending on the current mode and configuration of the transceiver.
- UiDriverShowMode – This function updates portion of the main display to indicate the mode contained in the variable "ts.dmod_mode". Additionally, it uses the variable "ts.cw_lsb" to determine which of the CW modes (CW-U, CW-L) should be indicated. If in FM mode the state of the squelch/tone detector is indicated by changing the color of the text surrounding "FM".
- UiDriverShowStep – This shows the tuning step size and, if enabled, draws/updates the line under the digit that visually indicates the step size.
- UiDriverShowBand – This updates the display to show the currently-tuned amateur band. There is an additional, non-ham band that is displayed as "Gen".
- UiDriverChangeBandFilter – This function selects the bandpass and lowpass filters based on the band information (number) passed to it. If the variable "bpf_only" is set, the lowpass filter relays are NOT updated, only the bandpass filters.
- UiDriverInitMainFreqDisplay – This function initializes the main frequency display, based on the mode (Split). It does not paint the numbers, just the decimal places and general formatting.
- UiDriverCreateDesktop – This function calls the various functions needed to "build" the main display, including the initialization of the main and sub frequency displays, buttons, S-Meter, waterfall or spectrum scope, initialization of the encoders, temperature, filters, band, DSP mode, bandwidth, initialization of CW state machines, voltmeter and more!
- UiDriverCreateFunctionButtons – This function draws the buttons below the Spectrum/Waterfall displays, based on the currently selected modes (VFO, TUNE, etc.)
- UiDriverDrawWhiteSMeter – This draws the "white" (S-9 and lower) part of the S meter. This is used in the initial build of the display and to restore this portion of the S-Meter to white after it has been turned red – *see below.*
- UiDriverDrawRedSMeter – This draws the same portion of the S-Meter as above, but in RED, instead: This is done when the receive signals present exceed a certain A/D threshold to indicate that the receiver is either decreasing its input CODEC gain or potentially going into clipping.
- UiDriverDeleteSmeter – This deletes the S-Meter, as you would expect!
- UiDriverCreateSMeter – This draws the S-Meter – lines, numbers, dots. It also draws the POWER meter as well as the meter below it which, depending on mode, could indicate ALC, SWR, or AUDio.
- UiDriverUpdateTopMeterA – This updates the "top" meter – the S-meter. If the meter exceeds full-scale or has not changed, it is not updated, saving time.
- UiDriverUpdateBtmMeter – This function updates the "lower" meter, which could be the ALC, SWR or AUDio indicator. A second parameter, "warn", is the threshold above which the dots being drawn will be indicated in RED which can vary, depending on meter mode.
- UiDrawSpectrumScopeFrequencyBarText – This function draws the frequency information below the Spectrum Scope and Waterfall Display. At the point representing the receiver's center frequency, the text below will indicate the frequency rounded to the nearest kHz, but above and

below it will show only the 1's and 10's of kHz. Note that the scaling of this information is dependent upon the setting of the "magnify" mode and the positioning of the text when the magnify mode is off is also dependent on the frequency translate mode (high, low, off).

- UiDriverCreateSpectrumScope – This function initializes the spectrum scope, first determining the "working colors" involved and then placing the various text as necessary and then drawing the graticule lines.
- UiDriverClearSpectrumDisplay – This sets to black the area of the screen occupied by the Spectrum Scope/Waterfall Display. This is used in preparation to present the menu or other text or to switch between the Waterfall and Spectrum Scope.
- UiDriverCreateDigiPanel – Not yet implemented.
- UiDriverInitFrequency – This does the inital set-up of the frequency synthesizer and band values.
- UiDriverCheckFilter – This function checks compares the frequency applied to it and if the filter required for this new frequency does not match the currently-selected one, it sets the bandpass and lowpass filters to it.
- UiDriverCheckBand – This function determines which band the current frequency lies and returns the value, taking into account the LO offset caused by the "frequency translate" function and updates the on-screen band indicator if the "Update" flag is set.
- UiDriverUpdateFrequency – This function is rather complex and is very easy to break so it must be thoroughly understood before being modified! The steps include:
  - When called, this will check for an update of the tuned frequency via the tuning encoder unless "skip_encoder_check" is TRUE: If not, it will simply process the current frequency. The "mode" variable is also present, setting the way the display is to be updated. Normally, mode "0" (automatic) is used, but additional modes are available to enforce the display of the "large" display, only (non-split mode), or the "upper" display (RX) or lower (TX) display as these can change independently of each other.
  - This function takes into account the offset caused by the frequency translate and whether this is active in TX mode or not (e.g. the case with CW) as well as the various display/frequency shifts involved in the different CW offset modes that are available.
  - This function also checks the tuning limits for the Si570 (synthesizer) and enforces them as needed and operates the RIT. All of these factors must include whether or not the transceiver is in RX or TX mode as aforementioned factors will determine the precisely-needed local oscillator frequency for the specific band, mode and transceiver state.
  - This function also does a preliminary check to see if the tuning of the Si570 will require a "coarse tune" event to occur: If it does, it schedules the future un-blanking of the system audio by momentarily disabling the various audio chains (DSP, speaker, etc.) and mutes them at that moment.
  - The tuning is performed, retrying once if it fails the first time: If it fails a second time the "col" variable is changed to cause the digit that was changed to be displayed in red.
  - If the LCD is using SPI mode the updates of the spectrum scope/waterfall display are halted with the time to resume updates rescheduled for a future time – this, because of the comparative slowness of the SPI mode that would otherwise make the radio nearly unusable if such updates were not halted when tuning was performed.
  - The frequency display is them updated, the precise manner depending on whether or not SPLIT mode was enabled. Depending on the circumstances, the secondary (smaller) frequency display may/may not be updated at this time as well.
  - A flag is set to indicate to the spectrum display/waterfall that the frequency was changed:

    This is used to determine the manner of future updates.
  - Finally, the new frequency is saved in the system variable that contains such information.
- UiDriverUpdateFrequencyFast – This function does only an update of the receiver's LO frequency, but it does not update the display. The same factors (frequency translate, CW offset mode, TX/RX, etc.) must also be considered.
- UiDriverUpdateLcdFreq – This function updates the LCD frequency using the supplied color and mode value: This "mode" value is the same as that used in the function "UiDriverUpdateFrequency" (e.g. large, and upper/lower for SPLIT). For this, the CW offset modes must also be considered when determining the frequency to be displayed as some of them are offset by varying amounts from the LO frequency. Additional consideration is given for the "transverter" mode which may be used to modify the displayed frequency with a user-supplied linear offset and multiplication value permitting frequencies up to just short of 1 GHz to be displayed. If the dial is locked, the frequency is displayed in gray color. Note that this function updates only those digits that have actually changed: If the flag "ts.refresh_freq_disp" is TRUE, ALL of the frequency digits are refreshed: *This flag is used only in those specific instances where it is absolutely necessary to force an update of ALL digits as redrawing all digits upon every frequency-tune update will significantly slow down the user interface of a radio that uses an LCD with an SPI interface!*
- UiDriverUpdateSecondFreq – This is the smaller frequency display that is used primarily for showing the actual receive frequency after the RIT offset has been applied. In the case of this frequency, the offsets imposed by the RIT and the CW offset modes is also taken into account. As with the main display, *only those digits that change are actually updated as always updating all digits will slow down the user interface response of a radio that uses an LCD with an SPI interface.*
- UiDriverChangeTuningStep – This changes to the "next" tuning step: If the flag "is_up" is TRUE, the next, larger, tuning step is selected. If the maximum tuning step is already selected when an "up" command occurs, a "wrap-around" is done to the smallest tuning step and vice-versa if the smallest tuning step had been selected when a "down" command occurred. Note that there are special provisions to allow larger-sized steps when the transverter mode is active to permit the quick changing of frequency in steps larger than the normal 100 kHz. Once the step size has been changed the on-screen indicator is also updated.
- UiDriverButtonCheck – This function scans and checks the main buttons, providing the debouncing and the timing for the press-and-hold functions.
- UiDriverTimeScheduler – This is one of those functions that must be thoroughly understood before it is modified as any screw-up here can badly wreck the operation of the transceiver! The various sub-functions are as follows:
  - The un-muting of the transceiver as it returns to RX from TX. Depending on the mode, the timing of the sequences that occur must be handled very precisely in order to both avoid clicks and pops as well as preventing the ruining of the cadence during CW operation when semi break-in operation is used. A number of different aspects of the hardware receive system must be considered when returning to receive including the proper handling of the restoration of the AGC system and audio gain settings.
  - The DSP is re-enabled after a scheduled delay upon return to receive.
  - Automatic change of on-screen parameters between TX and RX is handled.
  - Timing of delay of muting of transmit audio upon the beginning of a voice transmission is processed.
  - Timing the duration of the tone burst (in FM)
  - Update of the on-screen squelch/tone decode information (on the "FM" text).

  - Analysis of statistics from the DSP Noise Reduction is performed and a reset of the DSP NR engine is executed as necessary.
  - The delay of the enabling of the DSP upon power-on startup is handled.
  - The detection of a new firmware version and subsequent reinitialization of EEPROM is performed.
  - The un-muting of audio done during power-up and to suppress Si570 tuning clicks is done.
  - Timing of LCD backlight auto-off.
  - The processing of the button-press state machine to debounce and time button-presses: This includes the enforcement of minimum hold and release times.
- UiDriverChangeDemodMode – This function increments the current operating mode, skipping those modes that have been disabled by either deactivation (e.g. AM) or because of automatic sideband setting (e.g. auto USB above 10 Mhz). If the flag "noskip" is TRUE the modes are not skipped. Note that there are quite a few dependencies related to mode changes, namely the reinitialization of the RX and TX Hilbert transformers and the loading of the TX/RX amplitude adjustments.
- UiDriverChangeBand – This function, if "is_up" is TRUE, will increment the band, decrement if FALSE. This may not be done during transmit, and the volume must be set to minimum during this process in order to supress a loud "click" that would otherwise result. Note that there are quite a few dependencies related to band changes, namely the setting of the stored audio filter setting, the mode and the related reinitialization of the RX and TX Hilbert transformers and the loading of the TX/RX amplitude adjustments – not to mention proper handling of the band-gain settings, bandpass and lowpass filters and on-screen frequency displays.
- UiDriverCheckFrequencyEncoder – This function checks the state of the frequency tuning encoder, incrementing/decrementing as necessary – if the tuning dial has not been locked. The state of the display blanking is also updated, if enabled.
- UiDriverCheckEncoderOne – This function checks the state of encoder #1, incrementing/decrementing as appropriate. If the LCD is operating in SPI mode, updates of the spectrum scope/waterfall are momentarily halted in order to permit reasonable response of the user interface. In this function the audio gain, the primary function of this encoder is handles, along with the setting of the CW sidetone gain and audio compression level.
- UiDriverCheckEncoderTwo – This function checks the state of encoder #2, incrementing/decrementing as appropriate. If the LCD is operating in SPI mode, updates of the spectrum scope/waterfall are momentarily halted in order to permit reasonable response of the user interface. In this function the selection of the item is done in menu mode, otherwise it sets the RF gain, FM Squelch, Noise Blanker or DSP strength setting.
- UiDriverCheckEncoderThree – This function checks the state of encoder #3, incrementing/decrementing as appropriate. If the LCD is operating in SPI mode, updates of the spectrum scope/waterfall are momentarily halted in order to permit reasonable response of the user interface. In menu mode this encoder changes the selected menu item, otherwise it sets the RIT, CW speed or Line/Microphone gain.
- UiDriverChangeEncoderOneMode – Using button M1, this allows the selection between Audio Gain and CW sidetone gain/Audio Compressor settings, depending on transmit mode.
- UiDriverChangeEncoderTwoMode – Using button M2, this allows the selection between RF gain and DSP/Noise blanker settings.
- UiDriverChangeEncoderThreeMode – Using button M3, this allows the selection between RIT and the CW speed or Line/Microphone gain, depending on the transmit mode.

- UiDriverChangeAfGain – This function updates the on-screen display of the audio gain. If the "enabled" flag is not TRUE the indication is grayed-out.
- UiDriverChangeStGain – This function updates the CW Sidetone Gain. If the "enabled" flag is not TRUE the indication is grayed-out.
- UiDriverChangeCmpLevel – This function updates the display of the audio compression level. If the "enabled" flag is not TRUE the indication is grayed-out.
- UiDriverChangeDSPMode – This function updates the display of the DSP mode (Off, Notch, NR, NR+Notch).
- UiDriverChangePowerLevel – This function updates the power level display (e.g. 5W, 1W, etc.) and also calls the function that sets the band-gain.
- UiDriverChangeKeyerSpeed – This function updates the keyer speed. If the "enabled" flag is not TRUE the indication is grayed-out.
- UIDriverChangeAudioGain – This function displays the Microphone or Line input gain – depending on whether the Line or Microphone input is selected. If the "enabled" flag is not TRUE the indication is grayed-out.
- UiDriverChangeRfGain – This function displays the RF gain or, if in FM mode, the squelch setting. If the "enabled" flag is not TRUE the indication is grayed-out.
- UiDriverChangeSigProc – This function displays the setting of the DSP Signal Level or the Noise Blanker, depending on which is selected for display, providing color-based warnings of the settings. *Note that this function includes dependencies based on certain noise blanker/DSP modes being available or not, contingent on the selected demodulator mode and/or bandwidth.*
- UiDriverChangeRIT – This function updates the displayed RIT setting. If the "enabled" flag is not TRUE the indication is grayed-out.
- UiDriverChangeFilter – This function changes to the next audio filter UNLESS the "ui_only_update" flag is TRUE, in which case only the display is updated. *Note that this function includes dependencies based on the "wide" filter mode.*
- UiDriverDisplayFilterBW – This function determines the currently-selected audio filter, its operational bandwidth and center-frequency offset. It then determines various parameters related to USB/LSB of the selected mode and then calculates the size and position of the line to be drawn under the Spectrum Scope/Waterfall Display, erasing the "old" line and then drawing the new one. *This function has a large number interdepencies between modes, filter settings, filter offsets, etc.*
- UiDriverFFTWindowFunction – This function applies a "Window Function" (the DSP mathematical type!) to the sample data, prior to to the FFT being applied, to reduce "bin leakage" and adjust the "look and feel" of the Spectrum Scope/Waterfall display. The mathematics of contained in this function are beyond the scope of this document.
- UiDriverInitSpectrumDisplay – This function initializes the variables and functions required for either the Spectrum Scope or Waterfall display including initializing the palette RAM based on the selected colors as well as configuring the vertical gain factor for the Spectrum Scope, various settins related to the waterfall display, and the initialization of the FFT function used for both the Waterfall and Spectrum Scope.
- UiDriverReDrawSpectrumDisplay – This is function contains the state machine that produces the Spectrum Scope display. ***DO NOT** modify this function unless/until you have a thorough and complete understanding of its operation!* Its various sub-processes function as follows:
  - If in TX mode, the transceiver is powering down, in menu mode, the spectrum scope is inhibited by the timer because of a recent button-press/knob adjustment, etc. this function is immediately exited.

  - A "gain factor" is calculated based on the current CODEC gain setting ("gcalc") that is used to compensate for on-the-fly adjustments.
  - State 1 of the machine rescales the input samples based on the CODEC gain ("gcalc") and an emperically-derived calibration factor to provide optimal dynamic range. Once this is done the data is then operated upon by the selected window function (e.g. "UiDriverFFTWindowFunction") and then the FFT is performed.
  - State 2 of the machine first saves the "old" amplitude data from the previous screen display update and then calculates the complex magnitude from the processed FFT data. At this point the buffer ("sd.FFT_MagData") contains 256 useful amplitude-versus-frequency points, but ***they are not in linear frequency order,*** but rather fragmented into several individual segments!
  - State 3 of the machine applies a low-pass filter to the data using IIR techniques to "smooth" the running data based on past scans – the equivalent of a "Video" filter on a conventional spectrum analyzer: The "strength" (low-pass frequency cut-off) of this filter is adjustable. If the frequency has been recently adjusted, the frequency bar under the spectrum scope is updated, and if the frequency has been moved with a step size of greater than 1kHz, the contents of the filter are cleared and the display builds anew.
  - State 4 performs the "de-linarization", specifcally the application of the selected "dB-per-division" vertical scaling. At this point, things start to get complicated: If the "magnify" mode is enabled (e.g. 2x horizontal magnification) the portion of the spectrum that will be visible on the screen is analyzed to determine the minimum, maximum, and mean amplitude levels, after dB scaling: Because the spectrum is not contiguous, only the portions to be displayed are thus-analyzed – this being necessary because the mean/min/max values are used for the automatic scaling of the "reference" level in the appliation of the spectrum scope's AGC and it is imperative that *only* those amplitude levels that will be visible on the screen will be used to determine this information. If the "magnify" mode is disabled, all of the data is analyzed. If it is determined that the "maximum" value will exceed the maximum "Height" permitted on the display, the gain (spectrum scope AGC) is reduced, effectively shifting the display down, but if the "average" height of the display is too high, this is also taken into account as well to prevent the average level of a "blank" display (e.g. no "spikes" representing signals) from rising to the top of the display! If the signals present are on the low-ish side – say, upon changing to a "weaker" band, disconnecting the antenna or a due to the cessation of a strong signal, the gain (AGC) is increased somewhat slowly. This also includes a "sanity check" to detect when the entirety of the spectrum display is below the bottom of the screen and quickly bring it back up. After this is done, if the "magnify" mode is on, the various "segments" of the spectrum scope are "magnified" in the horizontal scale (in piecemeal!) to expand it to the needed 256-bin width and saved in the "sd.FFT_DspData" array whereupon the data is clipped to prevent illegal values.
  - State 5. At this point we have available the an array of 256 amplitude values regardless of whether the "magnify" mode is on or not. The second quarter of this arry is applied to the function "UiLcdHy28_DrawSpectrumInterleaved" along with the "old" data from the previous scan to udpate the left-hand side of the display, then the first quarter of the display is processed by the same function to update the right-hand half. The state machine is then reset to begin anew.
- UiDriverReDrawWaterfallDisplay – This is function contains the state machine that produces the Waterfall display. ***DO NOT*** *modify this function unless/until you have a thorough and complete understanding of its operation!* Its various sub-processes function as follows:
  - If in TX mode, the transceiver is powering down, in menu mode, the waterfall display

update is inhibited by the timer because of a recent button-press/knob adjustment, etc. this function is immediately exited.

- A "gain factor" is calculated based on the current CODEC gain setting ("gcalc") that is used to compensate for on-the-fly adjustments.
- State 1 of the machine rescales the input samples based on the CODEC gain ("gcalc") and an emperically-derived calibration factor to provide optimal dynamic range. Once this is done the data is then operated upon by the selected window function (e.g. "UiDriverFFTWindowFunction").
- State 2 of the machine first performs the FFT and then calculates the complex magnitude from the processed FFT data. At this point the buffer ("sd.FFT_MagData") contains 256 useful amplitude-versus-frequency points, but ***they are not in linear frequency order,*** but rather fragmented into several individual segments!
- State 3 of the machine applies a low-pass filter to the data using IIR techniques to "smooth" the running data based on past scans – the equivalent of a "Video" filter on a conventional spectrum analyzer: The "strength" (low-pass frequency cut-off) of this filter is adjustable. If the frequency has been recently adjusted, the frequency bar under the spectrum scope is updated, and if the frequency has been moved with a step size of greater than 1kHz, the contents of the filter are cleared and the display builds anew.
- State 4 performs the "de-linarization", specifcally the application of a fixed "10 dB-per-division" vertical scaling. At this point, things start to get complicated: If the "magnify" mode is enabled (e.g. 2x horizontal magnification) the portion of the spectrum that will be visible on the screen is analyzed to determine the minimum, maximum, and mean amplitude levels, after dB scaling: Because the spectrum is not contiguous, only the portions to be displayed are thus-analyzed – this being necessary because the mean/min/max values are used for the automatic scaling of the "reference" level in the appliation of the spectrum scope's AGC and it is imperative that *only* those amplitude levels that will be visible on the screen will be used to determine this information. If the "magnify" mode is disabled, all of the data is analyzed. If it is determined that the "maximum" value will exceed the maximum "Height" permitted on the display, the gain (spectrum scope AGC) is reduced, effectively shifting the display down, but if the "average" height of the display is too high, this is also taken into account as well to prevent the average level of a "blank" display (e.g. no "spikes" representing signals) from rising to the top of the display! If the signals present are on the low-ish side – say, upon changing to a "weaker" band, disconnecting the antenna or a due to the cessation of a strong signal, the gain (AGC) is increased somewhat slowly. The "threshold" of this AGC level is affected by the user-adjustable waterfall "brightness" control which, along with the "contrast" control, provides scaling to suit the operator's taste. This also includes a "sanity check" to detect when the entirety of the spectrum display is below the bottom of the screen and quickly bring it back up.
- State 5. At this point the amplitude data is multiplied by the user-adjustable waterfall "contrast" value which is used with the above "brightness" value to suit the user's tates. If the "magnify" mode is on, the various "segments" of the spectrum scope are "magnified" in the horizontal scale (in piecemeal!) to expand it to the needed 256-bin width and saved in the "sd.FFT_DspData" array whereupon the data is clipped to prevent illegal values. At this point we have available the an array of 256 amplitude values regardless of whether the "magnify" mode is on or not. This data is then saved in a circular buffer. Depending on the magnify mode and frequency translate mode, the position within the currently active line of the circular buffer is modified to include the vertical graticule that indicates the dial frequency of the receiver.

    - State 6 updates the circuilar buffer for the display, opens up a "bulk write" window via the LCD hardware and then does an indirect write to the display buffer using the stored amplitude value to index the palette – the latter being the value that is actually sent to the display. This "blind write" – which consists of all the columns and rows of the circular buffer and thus the waterfall display – complete, the "bulk write" is closed with the waterfall having been moved "up" by the specified number of steps and the state machine is then reset.
- UiDriverGetScopeTraceColour – This function simply indexes the scope trace color.
- UiInitSpectrumScopeWaterfall – This is a general-purpose function that clears the display in the location of the spectrum scope/waterfall, creates the display background, initializes the spectrum display and then draws the line that depicts the filter bandwidth/position.
- UiDriverUpdateUsbKeyboardStatus – *Not yet used.*
- UiDriverHandleSmeter – Thus function updates the S-meter and controls the gain of the CODEC chip. ***DO NOT*** *modfity this function unless/until you have a thorough and complete understanding of its operation!* The general operation of this function is as follows:
    - The setting of "RFG" (RF Gain) is obtained: If set anything other than "9" the gain of the codec is automatically controlled.
    - The codec gain "rfg_calc" is first offset and then doubled – then offset again to put scale/offset it such that when applied to the gain-control variable of the CODEC chip, it represents the useful range of the A/D converter input: This was determined by noting the maximum output voltage of the receiver front end (e.g. input to the codec) at the clipping level versus the full-scale A/D indication: Adjusting beyond this range would be pointless and wasteful. This value "rfg_calc", when in "auto" mode, is actually derived from the value "auto_rfg" which will be discussed shortly.
    - The "rfg_calc" value is then applied to the CODEC hardware: A manually-input value is applied directly – after the aforementioned rescaling/offsetting.
    - This "rfg_calc" value is now converted to a power ratio so that based on the gain value in front of the A/D converter we can keep track of that effect: Failure to do so would not only skew the AGC and the recovered audio, but also the spectrum scope/waterfall display and S meter!
    - This power ratio is applied to the value from the AGC to provide a correction that is used to compensate for the A/D gain adjustment. This value is then used as part of a sequential search to determine the corresponding S-meter value: The S-meter itself is now updated.
    - From the main audio processor, the flag "ads.agc_half_clip" is examined: If it is TRUE, an A/D sample had recently exceeded that value and that it is possible that signals are present that could cause A/D clipping. This triggers a decrease of the "auto_rfg" value that, on the next call, will decrease the CODEC gain in front of the A/D converter. The rate of the gain decrease is intentionally limited to prevent a very sudden and rapid change which can cause rather obvious and annoying artifacts.
    - If the flag "ads.adc_quarter_clip" has NOT been set – which indicates that the maximum A/D value is *half* that of the above – then the value of "audo_rfg" is increase to raise the gain. As with the case of the gain decrease there is a timer that limits the rate of adjustment – also to prevent artifacts. After both of these checks, the "half clip" and "quarter clip" flags are unconditionally reset.
    - If the flag "ads.adc_clip" is set this means that the A/D value recently see was twice that (6dB or 1 s-unit higher than) that which caused "ads.adc_half_clip" to be set. If this occurs a flag is set and the bottom-half of the S-meter is redraw in in red to indicate that the clipping level is being approached and the flag is then reset. It should be noted that the

actual threshold for "ads.adc_clip" is a ways below the A/D clipping level so the incidence of clipping is actually quite rare.
  - If the flag indicating possible clipping was set (above) the next time-around the bottom of the S-meter is redrawn to change it from red back to white.
- UiDriverHandleLowerMeter – This function handles the processing of "lower" portion of the S-meter which includes both the power indicator and SWR, ALC and AUDio indicators. Note that this section has some rather delicate calculations so be very careful if you feel as though you need to mess with it. This function works as follow:
  - If not in transmit mode, none of these lower meter functions are used and this is immediately exited.
  - The function processes the data only every-so-often, bailing out otherwise.
  - The A/D inputs are sampled numerous times to effect an average. This is absolutely necessary since not only are the signals varying rapidly due to modulation but also due to the fact that forward and reverse readings cannot be taken simultanously as there is only one A/D converter on the main MCU and they are delayed from each other due to the lag caused by the time to do the conversion and acquire the new A/D channel after switching its MUX.
  - Once the A/D samples have been accumulated, they are averaged and then the "sensor null" offset values previously obtained by the user are applied so that the readings are "zero" with no power input.
  - At very low power levels (low A/D values) the RF power is calculated using a second-order polynomial equation: At higher power levels (above a few 10's of milliwats) a third-order polynomial equation used. These nonlinear equations are required to compensate for the nature of the diodes so that the readings 'track" resonably well over the wide power range. ***It is <u>absolutely necessary</u> to use the two sets of equations, particularly for the accurate calculation of VSWR over wide power ranges!***
  - The readings from the above equations are then multiplied by user-inputted coupling calibration coefficients that compensate for differences related to operating frequency to maintain maximum accuracy across all bands.
  - The power readings are then converted to logarithmic values, dBm. From there, the power is converted to actual watts and using the dBm values, the return loss is then calculated, from which the SWR may also be directly calculated: Since the SWR is calculated from dBm the power level is irrelevant!
  - There is an optional digital display, selectable from the menu, that will show the forward and reverse power in milliwatts. This is used primarily for set-up and calibration of the power sensors.
  - The forward power is now rescaled such that there are 3 dots power watt of RF power and this is used to update the "top" meter to indicate transmitter action.
  - At this point there are three possible meter modes: SWR, ALC and AUDio.
    - If the SWR meter is to be displayed (ts.tx_meter_mode == METER_SWR) the above VSWR calculation is used to update the meter. The calculated VSWR is displayed ***only*** if the forward power is above a minimum value defined by "SWR_MIN_CALC_POWER" for two very important reasons:
      - There can be no meaningful VSWR calculation if there is no RF power present (key-up, silent portions of speech).
      - There needs to be a a sufficient amount of foward power such that with reasonably low VSWR values the reverse RF power detector will be able to discern some power. For example, if the lowest power that could be reliably measured by the

power detector was 20 milliwatts and the VSWR was known to be 1.2:1, there would need to be at least 2.4 watts forward. In actuality, if properly calibrated the actual lower-limit of the reverse power detector is just a few milliwatts. If the reverse power detector were not "zeroed" properly and/or there was not a lower limit on the RF power required to trigger an SWR reading the VSWR would appear to go up with lower power levels!

        - The VSWR indicator is never "zero" for the simple reason that the lowest-possible reading is 1:1.
        - The resolution of the display is four dots per "SWR" unit (e.g. 2:1 = 8 dots.)
        - An IIR low-pass filter is applied to "smooth" the reading – particularly important with the rapidly varying readings (and uncertainty) that inevitably results from the fact that the A/D sampling of the forward and reverse channels cannot be truly simultaneous and that there will be slight variances in the calculations due to inacurracies in the detected forward and reverse power measurements and the subsequent, instantaneous power calculations.
    - If the ALC meter is to be displayed (ts.tx_meter_mode == METER_ALC) the transmit ALC is obtained (variable "ads.alc_val") and is logarithmically converted and scaled to provide a more useful visible range on the display.
    - If the AUDio metering is to be displayed (ts.tx_meter_mode == METER_AUDIO) the peak audio level (ads.peak_audio) is retrieved, rescaled, logarithmically converted and then scaled and offset for display.
        - **IMPORTANT:** The values, threshold and scaling for the ALC and AUDio metering have been very carefully determined based on the analysis of the hardware (codec, modulator) to maximize the dynamic range of the D/A converter while keeping a "safe" distance from actual clipping of the converter and modulator.
- UiDriverHandlePowerSupply – This function will shut off the power to the transceiver if the "power down" command has been issued once the power-off delay has expired. This function also accumulates A/D readings from the voltage divider that reads the power supply voltage and scales it. As with the frequency display, only those digits that need to be updated are actually redrawn when the metering is updated – a process that occurs only once every few seconds. If the voltage is below 9.5 volts, it is displayed in red to indicate that transceiver operation may be compromised.
- UiDriverUpdateLoMeter – This function updates the bar graph associated with the TCXO temperature.
- UiDriverCreateTemperatureDisplay – This function draws the box and text associated with the TCXO temperature display in the upper-left corner of the screen, setting the appropriate color if enabled, the units of temperature and the current operational mode.
- UiDriverRefreshTemperatureDisplay – This function updates the operational status displayed in the TCXO temperature display in the upper-left corner of the screen as well as the current temperature, converting from centigrade to Fahrenheit if so-configured.
- UiDriverHandleLoTemperature – This function takes the reading from the sensor that is thermally-bonded to the Si570 – if present – and then converts it to centigrade*10000 and calls for an update of the on-screen display. If temperature compensation of the Si570 is enabled (e.g. "TCXO") two data points in the temperature compensation table are read: The integer degree below the current temperature and the next one above – and a linear interpolation of that data is performed using the current temperature to provide a smoother frequency compensation than would otherwise occur if such happened only on even-degree thresholds: If the temperature is beyond that included in the table, a "believable" value is synthesized. With the

frequency offset based on the temperature calculated, the function "UiDriverUpdateFrequencyFast" is called to update the Si570's operational frequency.

- UidriverEditMode – Not completed.
- UiDriverSwitchOffPtt – This function handles activation or deactivation of the PTT: The handling of this in CW mode is done elsewhere. *Be VERY careful if you modify this function as you can really break things if you mess with it!*
- UiCodecMute – This function simply makes the "Codec_Mute" function available to "main.c": We don't include the file containing that function because of compiler conflicts – and it was easy to do it this way!
- UiDriverSetBandPowerFactor – This function looks up the band-gain and power-setting values, derived from EEPROM, based on the user-selected band and power mode. Note that "FULL" power has one set of value – used only for "FULL" power – while the 5-watt power level and others (2 watts, 1 watt, 0.5 watts) are all derived from the 5-watt values. It is these values by which the signal amplitude in the modulator functions is multiplied to achieve the appropriate power output for each band and power setting.
- UiLDDBlankTiming – This function schedules the LCD blanking time for the backlight auto-off (if that feature is enabled).
- UiCalcAGCDecay – This function calculates the AGC decay value appropriate to the selected mode. Because the nature of the AGC action is exponential, this value is as well.
- UiCalcALCDecay – This function calculates the ALC decay value appropriate to compression setting. Because the nature of the ALC action is exponential, this value is as well.
- UiCalcRFGain – This function calculates the value related to the RF gain value appropriate to the RF Gain setting. Because the nature of the AGC is exponential, this value is as well.
- UiCWSidebandMode – This function determines the appropriate sideband for CW operation based on the selected CW mode and, in certain cases, the operating frequency.
- UiCalcNB_AGC – This function calculates the parameter for the noise blanker's own AGC time constant. Because the nature of the AGC action is exponential, this value is as well.
- UiCalcRxIqGainAdj – This function calculates the amount of receive system I/Q gain adjustment appropriate for to the selected mode, determined and saved in EEPROM during user calibration. The amount of amplitude is applied equally and opposite to the I and Q channels.
- UiCalcTxIqGainAdj - This function calculates the amount of transmit system I/Q gain adjustment appropriate for to the selected mode, determined and saved in EEPROM during user calibration. The amount of amplitude is applied equally and opposite to the I and Q channels.
- UiCalcRxPhaseAdj – This function loads and modifies the 89-tap FIR present in the "front end" of the receive function: If AM demodulation is selected, the FIR filters in the I and Q channels are simply low-pass but in all other modes they comparise a pair of "phase-added" (0-90 degree) Hilbert transformers to produce the quadrature audio. For fractional-degree phase adjustment the FIR coefficients of the "Q" channel are linearly interpolated using three available sets of Q-channel models, 89.5, 90.0 and 90.5 degrees, to simulate phase shifts between these values.
- UiCalcTxCompLevel – This function looks up the pre-set values for the pre-ALC gain and the ALC decay rate needed to synthesize the various "strengths" of audio compression. The highest value, "SV", loads user-defined settings from EEPROM.
- UiCalcSubaudibleDetFreq – This function looks up and calculates the DDS frequency word for the generation of the subaudible tone used for FM transmit, if any.
- UiCalcSubaudibleGenFreq – This function looks up and calculates the frequency of the subaudible tone used for FM receive (to be detected) and calculates the Goertzel values for the

tone detection algorithms.

- UiLoadToneBurstMode – This function loads the constants used for the tone burst generator based on user settings.
- UiLoadBeepFreq – This function loads and calculates the DDS frequency word for the keyboard (and CW Sidetone test) beep. It also calculates/loads the variables to set the loudness of the keyboard and CW sidetone test beeps.
- UiSideToneRef – This function causes a tone of the currently-set CW sidetone frequency to be generated and sounded in the speaker (but not on the "Line Out").
- UiCalcTxPhaseAdj – This function loads and modifies the 89-tap FIR that comparise a pair of "phase-added" (0-90 degree) Hilbert transformers to produce the quadrature audio for transmit. For fractional-degree phase adjustment the FIR coefficients of the "Q" channel are linearly interpolated using three available sets of Q-channel models, 89.5, 90.0 and 90.5 degrees, to simulate phase shifts between these values.
- UiLoadFilterFalue – This function loads the audio filter value from EEPROM. While this same functionality is included in the master function that loads EEPROM values ("UiDriverLoadEepromValues", below) this is used during startup to get only this parameter to facilitate initialization of the audio processes.
- UiCheckForEEPROMLoadDefualtRequest – This function will look for the key combination of "F1+F3+F5" on power-up and if this occurs – and a new firmware version has NOT been loaded – it will cause the loading of default values instead of the stored EEPROM values. The user has the option of either power down to keep the "old" values, or powering down using the POWER button and saving the newly-loaded defaults, erasing ALL old values (configuration, adjustments, frequencies, modes, etc.)
- UiCheckForPressedKey – This function detects if a key (other than POWER) is pressed on power-up. If so, it enters a mode in which the key (of highest precedence) is displayed on the display. The only way to exit this is to disconnect power.
- UiDriverLoadEepromValues – This is the master function that reads the user-stored data from EEPROM, validates it and then stores it in the appropriate system variables: If the data that was read from EEPROM was outside the expected range (e.g. read failure, newly-initialized EEPROM, new firmware) the "default" value is used, instead. There are a few important notes related to this function:
    - Use only the indirect function "Read_VirtEEPROM" to read virtual EEPROM data: It will fail otherwise!
    - Several of the parameters restored use signed variables and this is done by passing the pointer of an unsigned variable to that representing a signed variable: Thusfar, this is the only way that has been found to get this to work reliably – even though it causes a compiler warning.
    - **PLEASE READ** the comments in the source code, particularly for variables "ts.band", "tune_bands[]", and "ts.filter_id". It is recommended that you do NOT change these formats unless you have a damn good reason as doing so will forever break any backwards compatibility with other firmware versions!
    - There was an attempt to separate the EEPROM load/save function into its own source file but some compiler dependencies prevented this from being easily done. If this can be done while maintaining compatibility with both the CooCox and Eclipse compilers, then that would be good!
    - Do not "re-use" or reassign EEPROM addresses – at least not unless there are many versions between where an address was last-used! The reasons for this should be obvious.
    - The frequency/band/mode/filter settings for the VFOs are saved three times: One for the

    main tuning (working) variables and then also for the separate VFOA and VFOB variables. It is recommended that you leave them this way unless you are willing to completely rewrite all dependent functions all throughout the code!
  - Don't do anything too clever or stupid – which often boils down to the same thing!
- UiDriverSaveEepromValuesPowerDown – This function saves the various user-defined settings to EEPROM. Contrary to the name, this is not used only during power-down, but for those instances where all variables are to be saved to EEPROM (e.g. press-and-hold of the "F1" button.) If a variable has not been used in the virtual EEPROM before, it is created by this function and if necessary, the default value for that parameter is written to it. There are a few important notes related to this function:
  - Use only the indirect functions "Read_VirtEEPROM" to read virtual EEPROM data and "Write_VirtEEPROM" to write virtual EEPROM data: It will fail otherwise!
  - Several of the parameters restored use signed variables and this is done by passing the pointer of an unsigned variable to that representing a signed variable: Thusfar, this is the only way that has been found to get this to work reliably – even though it causes a compiler warning.
  - **PLEASE READ** the comments in the source code, particularly for variables "ts.band", "tune_bands[]", and "ts.filter_id". It is recommended that you do NOT change these formats unless you have a damn good reason as doing so will forever break any backwards compatibility with other firmware versions!
  - There was an attempt to separate the EEPROM load/save function into its own source file but some compiler dependencies prevented this from being easily done. If this can be done while maintaining compatibility with both the CooCox and Eclipse compilers, then that would be good!
  - Do not "re-use" or reassign EEPROM addresses – at least not unless there are many versions between where an address was last-used! The reasons for this should be obvious.
  - The frequency/band/mode/filter settings for the VFOs are saved three times: One for the main tuning (working) variables and then also for the separate VFOA and VFOB variables. It is recommended that you leave them this way unless you are willing to completely rewrite all dependent functions all throughout the code!

The "ui_driver.h" file:

This file contains many configuration parameters/definitions related to the fundamental operation of the transceiver.

DO NOT MESS WITH ANY OF THEM UNLESS YOU HAVE A THOROUGH UNDERSTANDING OF WHAT THEY CONTROL! Specifically, there are quite a few definitions that are related to the characteristics of the mcHF hardware that have already been carefully optimized: Little would be gained with additional "tweaking" as the charactaristics of the hardware involved (e.g. CODEC specifications, measured capabilities of the mcHF signal paths, etc.) have been taken into account. Additional parameters (AGC, etc.) have been carefully measured using a wide variety of test equipment to verify performance and operation such as the look-ahead delay of the AGC/ALC algorithms, CODEC gain control, derivation of the polynomials involved in the forward/reverse power metering, etc.

A large number of these definitions also relate to the absolute positions of the graphical elements on the

LCD display itself: Unless/until you are prepared to take the implied dependencies into account, one should change these only with due care!

The "ui_menu.c" file:

This file contains the menu display/drivers for the "adjustment" and "configuration" menus. This particular subsystem has a very large number of dependencies as the change of one menu item can have an impact of quite a few others, often requiring that hardware be directly updated, immediately – or in some cases, not!

The function "UiDriverUpdateMenu" has several modes: 0 – Update the display of all items, 1 – update the currently-selected item, 2 – go to the next screen, or 3 – restore to default the settings for the selected item.

The menu system is driven primarily by five user controls:

- ENC2 – This moves the cursor up and down to select menu items.
- ENC3 – This changes the options of the currently-selected menu item.
- F2 – This restores to default the currently-selected menu item.
- F3 – This moves to the previous screen or, if pressed-and-held, the first screen of the menu.
- F4 – This moves to the next screen or, if pressed-and-held, the last screen of the menu.

Because the radio is "generally" operational when in menu mode, ENC1 is available for volume control and button F5 is available for TUNE functions. Of course, F1 will exit the menu system at any time.

The currently-selected menu item is held in the variable "ts.menu_item" and based on this up to six items on that particular menu page are displayed along with the current settings of those parameters: In general, all parameters on the visible page are updated any time the cursor is moved amonst items using ENC2 or the page is changed.

There are currently two separate menu systems: The "adjustment" menu which contains the most frequently-used parameters and the "configuration" menu which has lesser-used parameters mostly related to hardware: The later menu must be specifically enabled in the "adjustment" menu in order to appear.

It should be noted that because of the many dependencies, menu items may need to be updated based on mode (which implies band!) as well as filter settings as well as whether or not the unit is in transmit or receive mode – this being necessary because some menu items are "locked out" to prevent accidental adjustment of unrelated parameters (e.g. you cannot adjust band-gain settings unless you are actually in transmit mode, on that band). Because of this the transceiver's "turn-around" time between TX and RX is quite slow when in the menu mode and CW mode is generally not possible and a warning to this effect is displayed at the bottom of the screen.

This particular function could likely be rewritten to be easier to maintain. In its present form it is necessary to cut-and-paste many times if one inserts a new menu item amongst others as everything below it must be moved down – a real PITA! For the moment, observe that the menu display items – and the subsequent sections that jump to the "UiDriverUpdatexxxLines" functions are arranged in groups of six – this being the maximum number of menu items that can be displayed on the screen with the current layout: If it is required that another "screen" be added, you would be wise to pay very close attention to how these portions are configured, keeping a copy of what was there before you started: The code is not complicated, but you may lose hair unless/until you figure out exactly how it works!

The menu system is extremely CPU intensive owing to the tremendous amount of graphical updating that is required. While not so much a problem with a parallel-interface LCD, this is quite critical with SPI-interface LCDs and a careful study of the code will reveal that some aspects of the UI related to the menu system are different when using an SPI-based LCD – mostly causing fewer updates to occur on-screen: This is likely not noticed unless an A/B comparison is made. In order to reduce processor overhead while in the menu system, the noise blanker – if enabled – is always forced off, but if you have the receive system configured such that it already consumes a large amount of processor power (e.g. AM reception with a WIDE filter and DSP active) you will definitely notice that the menu system slows down whether you have an SPI or Parallel LCD interface!

The function "UiDriverUpdateMenuLines" drives the "adjustment" menu, what is operated by ENC3 to change the actual menu item and what is displayed. If any menu item that is saved in EEPROM (not all of them are!) is changed one must remember to set the " ts.manu_var_changed" flag to 1 (TRUE) to cause an indication to the user that it is necessary to save to EEPROM via power-down or press-and-hold of F1 if those changes are to be kept: If the changes are not to be kept one would simply disconnect the power from the transceiver.

In each of the **case** instances within this function – each one corresponding to a menu selection – may be found customized code related to that particular parameter and how it is displayed. A few things to note:

- In the current menu convention, items that are not available in the current operational mode are typically displayed in orange and, in most cases, cannot be adjusted.
- There are many parameters that will change color as they are adjusted (example: DSP NR strength) from, say, white to yellow, orange then red to indicate that one is entering a "danger zone" for a value – which is to say, exceeding a recommended setting: The rationale is that the user may experiment with such settings, but fully expect that something may either go wrong or, in the case of the "Maximum Volume" setting, actually be dangerous!
- Remember that many paramters have a lot of dependencies. For example, if you change the TX IQ phase you must remember to reload the Hilbert transformer for that change to take effect – and this is also a change that one would ONLY allow if one was currently transmitting in a mode, using that specific filter, anyway!
- You must be aware of the limitations and configurations of the hardware if you choose to modify any parameters related to it: There are quite a few functions present that may be used to "talk" to the various pieces of hardware already that probably do what you need to do! Having said that, there are still a few functions within the menu system that need to be moved to their own functions as their "function" is already being duplicated elsewhere and it would be a good idea to get everything into one place.
- The color of the parameter displayed may be modified as needed using the "clr" parameter.
- If the parameter to be displayed is "large" one may set the parameter "disp_shift" to TRUE and shift it to the left several characters – but this works only if the description for that menu item itself is not too long.

- Don't be too cute! It is very easy to screw things up when adding new menu items, so add only one-at-a-time, if you can help it.

The function "UiDriverUpdateConfigMenuLines" is much the same as the above function except that it is for the "configuration" menu: The same rules apply.

The functions "UiDriverMemMenu" and "UiDriverUpdateMemLines" are in a state of developmental flux.

The "ui_menu.h" file:

This file contains the numerical positions, using the "enum" directive, of each of the menu items – one set for the "adjustment" menu and the other for the "configuration" menu: **The order and count of the menu items must be the same as the display on the screen as it is in the enumeration!**

It should be noted that two of the items must always be maintained at the end of the list:

MAX_MENU_ITEM – always at the end of the "adjustment" menu.
MAX_RADIO_CONFIG_ITEMS – always at the end of the "configuration" menu.

These are used to determine the size and scope of the two menu subsystems and by placing them at the end, they represent how many items are present in each menu.

Versions:
20150918 – FW Version 0.0.219.22 - CT

[END OF DOCUMENT]