Upgraded Software for the W8TEE/K2ZIA Antenna Analyzer - Version 3.8

Hacker’s Guide

John Price – WA2FZW

License Information

Introduction

The W8TEE/K2ZIA antenna analyzer was originally developed as a club project for the Milford Amateur Radio Club. The original author of the software, Jack Purdum, published the design and code online on the Yahoo SoftwareControlledHamRadio group (which has now been moved to the SoftwareControlledHamRadio group on Groups.io). Jack also published an article about the project in the November 2017 issue of QST.

My main objective in modifying the software was to make it work on the 6 meter band (which requires replacing the AD9850 DDS with the higher frequency AD9851). In the process of going through the original code to figure out how to accomplish this, I did find a number of potential and actual bugs in the code (Definition: Working Software – Software with only undiscovered bugs) and came up with some enhancements to make it easier to use. Then I got a little carried away!

This document provides an overview of what each of the functions that make up the software do at a very high level for anyone who wants to attempt their own modifications. You will find rather detailed descriptions of what each function does in the comments in the software itself.

My_Analyzer.h

This header file contains all the definitions of things that anyone might want (or need) to change in order for the analyzer to operate with the various hardware options available.

The definitions of the Arduino pins used for the encoder and its built-in switch are near the beginning of the header file. Many, if not most of the encoders on eBay, including the dozen or so in my parts bin are wired backwards. If your encoder works backwards simply flip the numerical values of the definitions for “PINA” and “PINB”, and all will be right with the world.

```
#define PIN_A   18          // Encoder hookup; reverse A and B if
#define PIN_B   19          // it works backwards
#define SWITCH  20
```

The following are the definitions for the various DDS and hardware configuration options:

```
#define AD9850_DDS    // Using the AD9850 DDS board
#define AD9851_DDS    // Using the AD9851 DDS board
#define AD8307_SWR    // Using AD8307 detectors
#define PE1PWF_MOD    // Using Edwin’s modifications
```

Select one of the two DDS modules by un-commenting its definition, and make sure the other is commented out. It you’re using the AD8307 detector circuit or have installed the PE1PWF modifications, make sure you un-comment the appropriate definition.

Added in Version 03.3, we define 2 DDS calibration factors; one for the AD9850 module and one for the AD9851 module. The normal calibration factors for these are 125MHz and 180MHz respectively; however we already know there are a few AD9851 modules in use that have an issue with operating at 180MHz and will only work if the calibration is set to 120MHz. If this applies to your DDS, change the definition of "CAL_9851" to "120000000UL".

```
#define    CAL_9850   125000000UL
#define    CAL_9851   180000000UL
```

Whenever the “Freq Cal” function under the maintenance menu is used, the calibration factor displayed will revert back to the default value as set by one of the above definitions. Once you’ve done the calibration, you can write down the number and change the appropriate definition above to that value, which will then be used as the starting point the next time you do the calibration.

Note that the calibration factor is saved in the EEPROM and that value will be used each time the analyzer is started.

Definitions related to adding the 6 meter and/or “Custom” bands:

```
#define ADD_6_METERS              // Enable 6 meter operation
#define LOW_6M_EDGE   50000U      // 50 MHz / 1000
#define HIGH_6M_EDGE  54000U      // 54 MHz / 1000

#define ADD_CUSTOM                // Enable the "Custom" band
#define LOW_C_EDGE      100U      //  100KHz / 1000
#define HIGH_C_EDGE   65500U      // 65.5MHz / 1000
```

The details of how to enable the extra bands and set the band limits are explained in the “User Manual”. Note that in Version 03.3, the default limits of the “Custom” band were changed to a range of 100KHz to 65.5MHz. Do not try to set them any lower or any higher. Bad things will happen!

Definitions of the parameters that control the “Repeat Scan” function:

```
#define DEFAULT_COUNT 50          // Initial default count
#define REPEAT_INCREMENT 10       // How much to change the count
#define MIN_REPEAT_COUNT 10       // Minimum repetition count
#define MAX_REPEAT_COUNT 100      // Maximum repetition count
#define SCAN_PAUSE 1000           // Length of pause between scans
```

If the “Fine Tune” button had been added, un-comment the line:

```
#define    FT_INSTALLED
```

If you want the “Examine” function to operate automatically after using the “Single Scan”, “Repeat Scans” and “View Plot” functions, un-comment the following line.

```
#define AUTO_EXAMINE
```

If the definition is commented out, the “Examine” feature will still be activated by simply moving the encoder knob one click one way or the other.

Added in Version 03.2 is the ability to display labels on the “Single Scans”, “Repeat Scan”, “View Plot” and “Overlay” graphs. This feature can be turned on or off commenting out or un-commenting the line:

```
#define LABEL_SCANS
```

in the header file.

Added in Version 03.5 is an option associated with the “Examine” feature to display not only a vertical cursor line, but a horizontal one to indicate SWR as well. The horizontal cursor will change colors based on the SWR value at the scan point being examined. The line will be green if the SWR is less than 1.5:1, yellow if the SWR is less than 2:1 and red if the SWR is 2:1 or greater.

The option is enabled by un-commenting the line:

```
#define HORIZ_INDEX
```

Another new feature in Version 03.5 is the ability to skip over the requests for the current band and frequency range settings at startup. However, if you’ve erased the EEPROM or it’s the first time the analyzer has been used, you will be asked for the information.

To skip the band and frequency requests at startup, un-comment the line:

```
#define    SKIP_BAND_SELECT
```

Added in Version 03.6 is the option to display the forward and reverse Arduino pin readings on the "Frequency" and the "SWR Calibration" displays. Also by pressing the "Fine Tune" button for more than one second, the DDS can be toggled on and off when using these functions.

If you want to enable this option, un-comment the following definition:

```
#define    VIEW_PIN_DATA
```

Battery Check Function

Added in Version 03.8 is a model of how to implement a low battery detection function. There are three symbols associated with this capability in the header file:

```
#define DO_BATT_CK          // Un-comment to enable the capability
#define BATT_CHECK_PIN A0   // Whichever pin you use to check
#define LOW_BATTERY   512   // Low voltage limit
```

The “BatteryCheck” function will only be compiled if the “DO_BATT_CK” symbol is defined.

The “BATT_CHECK_PIN” is whichever analog pin you have hooked your method of measuring the power supply voltage to. Change the “A0” to the appropriate pin designation.

The value of “LOW_BATTERY” is the reading on the analog pin below which your unit doesn’t function properly. You’ll have to determine what the appropriate value is for your particular unit and set the number accordingly.

EPROM Address Map

The following shows the addresses of things stored in the EEPROM and the symbolic names used in the program to reference them.

Address	Symbol	Purpose in Life
0000 – 0001	SWR_MINS_SET	Indicates minimum SWRs have been set
0002 – 0003	SWR_MINS_ADDRESS	Saved SWR readings - 160M
0004 - 0005		Saved SWR readings - 80M

0006 - 0007		Saved SWR readings - 60M
0008 - 0009		Saved SWR readings - 40M
0010 - 0011		Saved SWR readings - 30M
0012 - 0013		Saved SWR readings - 20M
0014 - 0015		Saved SWR readings - 17M
0016 - 0017		Saved SWR readings - 15M
0018 - 0019		Saved SWR readings - 12M
0020 - 0021		Saved SWR readings - 10M
0022 - 0023		Saved SWR readings - 6M
0024 - 0025		Saved SWR readings – Custom
0026 – 0049		Reserved for additional bands
0050 – 0051	ACTIVE_BAND_SET	Indicates active band data is saved
0052 – 0053	ACTIVE_BAND_INDEX	Saved active band index setting
0054 – 0055	ACTIVE_BAND_BOTTOM	Saved active band low freq setting
0056 – 0057	ACTIVE_BAND_TOP	Saved active band high freq setting
0058 – 0061	FREQ_CALIBRATION	Saved calibration constant for the DDS
0090 – 0091	NEXT_SD_FILE_NUMBER	Saved next file sequence number
0092 – 0093	SWR_CALIBRATION	Calibration factor for the PE1PWF mods
0094 – 0094	DEBUG_MODE	Last setting of dynamic debug flag
0096 – 0097	DDS_IN_USE	Last used DDS type (9850 0r 9851)
0098 – 0101	SLOPE_CALIBRATION	Calibration factor for the VK3PE board
0102	EEPROM_NEXT	Next available EEPROM address

Main Program Functions

The first two functions are the standard Arduino functions:

setup()	Initializes all of the things needed to make it work
loop()	Runs continuously; basically processes the menu functions

The functions described in the following sections are grouped according to their primary role (although they may have secondary roles). The order they are listed in is the same order as which they appear in the .ino file.

Initialization Functions

The functions in this group primarily deal with setting various variables that make everything else work.

SetActiveBand()	Sets the active band an startup and may be invoked from the "Analysis" menu.
SetBandEdge()	Sets the upper and lower scan frequency range
GetBandEdge()	Gets the upper and lower legal band edge limits
GetActiveEdge()	Gets the upper and lower band edge settings set by the operator.
ReadActiveBandData()	Reads the saved band and frequency information from the EEPROM.
SaveActiveBandData()	Saves active band and frequency information to the EEPROM.
SetEEPROMMins()	Sets the saved minimum SWR readings in the EEPROM.
ReadEEPROMMins()	Reads the saved minimum SWR readings from the EEPROM.

Menu Processing Functions

These functions process the main and sub-menus

ShowMainMenu()	Display the top level menu.
AlterMenuOption()	Controls which main menu item is selected and makes a selection when the encoder switch is pushed.
ShowSubMenu()	Displays one of the sub-menus.
AlterMenuDepth()	Controls which sub-menu item is selected and makes a selection when the encoder switch is pushed.
DoAnalysis()	Processes selections in the "Analysis" menu.

DoOptions()	Processes selections in the “View/Save” menu.
DoMaintenance()	Processes selections in the “Maintenance” menu.
DoSetOptions()	Shows a submenu of program options the user can set which dynamically alter the program behavior and can reduce the need for changing compile-time options.

Command Processing Functions

The functions in this group are responsible for executing the individual commands initiated via sub-menu selections.

DoNewScan()	Performs and displays the results of a single scan between the preset frequency ranges.
RepeatScan()	Repeats a scan between the preset frequency ranges a specified number of times.
Examine()	This is not exactly a command processing function, but it’s called after “DoNewScan”, “RepeatScan” and “ViewOldPlot” to allow the operator to examine the SWR at frequencies determined by moving the encoder knob.
DoSingleFrequency()	Monitors the SWR at one specific frequency (which can be changed while monitoring). It also includes an “analog” SWR meter function. Modified in Version 03.6 to optionally display the raw Arduino forward and reverse pin readings and to toggle the DDS on and off.
DoCalibration()	As of Version 03.6, there are now two versions of this function; one for the PE1PWF modifications and one for the VK3PE modifications. Outwardly, they look the same, but the internal math is different. As is the case in “DoSingleFrequency”, the raw Arduino forward and reverse pin readings can be displayed and the DDS can be toggled on and off.

SaveScan()	Saves the data from the most recent scan to the SD card.
ViewOldPlot()	Displays the contents of a saved scan exactly as it was displayed when originally performed.
PlotOverlay()	Can be used to display the results of a saved scan on the same graph as a live scan or another saved one.
ViewTable()	Displays the contents of a saved scan file in a tabular format.
PlotToSerial()	Sends the contents of a saved scan file verbatim to the Arduino IDE’s serial monitor.
DrawBarChart()	Draws the bar chart of the saved minimum SWR values.
DeleteSingleFile()	Deletes a selected single file from the SD card.
DeleteAllFiles()	Deletes all of the files from the SD card.
ResetFileSeqNumber()	Resets the next file sequence number provided there are no saved scan files on the SD card.
EraseEEPROM()	Completely erases the contents of the EEPROM except the next file sequence number.
ReadEEPROM()	Displays the contents of the EEPROM on the Arduino IDE’s serial monitor.

Frequency Calibration Functions

Although a couple of these could be include in the “Formatting & Display Functions” section, I elected to keep them together in the code.

DoFreqCal()	Called when the “Freq Cal” option is selected from the “Maintenance” menu. It allows the operator to set the calibration frequency and calibration constant and saves the new value in the EEPROM.

DisplayCalibration()	Handles displaying the calibration constant on the screen with the usual carat (^) character under the digit which will change when the encoder is turned.
FormatCalConstant()	Used to format the calibration constant string.

Formatting & Display Functions

These functions either format specific data items or display specific things on the TFT display:

Splash()	Displays the startup information and credits.
FormatFrequency()	Formats the internal frequency into an ASCII string with either 2 or 3 decimal places.
FormatFloat()	This is a general purpose function for formatting any floating point number.
PaintSwrData()	Displays the readings on the Arduino forward and reverse pins and the difference between them. It also handles the ability to toggle the DDS on and off.
FormatSWR()	Formats the internal SWR into an ASCII string.
PaintText()	Writes text strings of various sizes on the screen.
EraseText()	Erases text from the screen.
GraphAxis()	Plots the background for the scan plots.
GraphPoints()	Plots the actual scan data.
MarkMinimum()	Puts the little red ‘+’ characters at the minimum SWR points on the scan graphs.
PaintMeter()	Paints the “analog” SWR meter on the display.
MovePointer()	Controls the movement of the pointer on the “analog” SWR meter.

ShowAndScroll()	Part of the “View Table” function; decides which page should be displayed and controls scrolling between pages.
DrawTable()	Displays a single page of the “View Table” output.
DisplayFrequency()	Used to display frequencies with the carat (‘^’) character under the digit that rotating the encoder will change.
PaintHeading()	Paints the headings on the plot and table outputs.
PaintFooter()	Displays current band and scan frequency limits at the bottom of the screen.

DDS Control Functions

These functions manipulate the DDS:

GetNextPoint()	Gets the next SWR/frequency pair when performing a live scan.
ReadSWRValue()	Reads and computes the VSWR.

SD Card Related Functions

These all have to do with things related to using the SD card:

Mount_SD()	Mounts the SD card at startup or if a card wasn’t installed at startup. Still has a bug (see the *User Manual*)
CountFiles()	Counts the numner of “SCANnn” files on the card.
ShowFiles()	Displays a list of the “SCANnn” files on the card.
SelectFile()	Allows the operator to select a single file from the displayed list.
SortFiles()	Sorts the files by name before displaying them.

ConfirmDelete()	Gives the operator the ability to cancel out of or confirm deletion of a single file.
ConfirmDeleteAll()	Gives the operator the ability to cancel out of or confirm deleting all files.
ReadScanDataFile()	Reads the scan data from a saved file.
WriteScanData()	Writes the current scan to an SD file.
Display_SD_Err()	General error display for SD problems.
Display_SD_Err_2()	Specific sequence of error messages when there are no “SCANnn” files on the card.
Display_SD_Err_4()	Specific sequence of error messages when there is no SD card present.

Interrupt Processing Functions

These functions handle the actual interrupts from the encoder and “Fine Tune” button (if installed) and the subsequent processing of those interrupts.

ReadEncoder()	Handles and processes interrupts generated by rotating the encoder knob.
ResetEncoder()	Clears the flags resulting from moving the encoder.
ReadFT()	Handles the “Fine Tune” button. In Version 03.6, the “Fine Tune” button is no longer interrupt driven. This function was also modified to be able to differentiate between short and long pushes of the button.
ResetFT()	Resets the variables associated with the “Fine Tune” button.

Miscellaneous Functions

These really don't neatly fit into any of the previous categories:

ConfirmAction()	Used where the operator is given the option to cancel out of a previously selected function such as deleting a file or erasing the EEPROM.
DisplayScanStruct()	Conditionalized on the definition of DEBUG, this function displays the contents of the "scan" structure on the Arduino IDE's serial monitor.
ShowDebugMode()	When dynamic debugging is enabled, this function displays a red "DB" in the upper right hand corner of most of the screens.
BatteryCheck()	This function is not intended to be a working function (although it is), but rather is included as a model for how one might implement the capability.

Statistics Calculations

New features in Version 3.6 calculate and send basic statistics on scan readings to the IDE's serial monitor. This data can be useful when debugging your hardware, or for comparing the influence of induced or mitigated noise on different hardware & software configurations. To make these tests, attach a resistive load, and set the desired logging level under Maintenance > Options.

Statistics calculations are implemented via a new C++ class: AAStats, which is defined in two new files, AAStats.cpp and AAStats.h.

The public methods of an object of class AAStats are:

AAStats()	Constructs objects of class AAStats.
InitAVGReadingsForScan()	Initializes the variables used in calculation of average means and standard deviations for all samples in a scan.
ResetStatsCounters()	Initializes the variables used in calculation of average means and standard deviations for a single scan.

CollectStatsDataPerPoint()	Populates arrays of forward and reflected readings for a scan point.
LogIndivReadings()	Logs the individual forward and reflected readings in a sample for a scan point. Unless you reduce the #defines SCAN_INTERVALS and MAX_POINTS_PER_SAMPLE in the. ino file, you will get TONS of output. I set each of these #defines to 5 for useful tests. Note that if MAX_POINTS_PER_SAMPLE is set to greater than 75, buffer overruns will occur, as the size of the arrays that hold individual forward & reverse readings is hardcoded to the default value of 75. (You can change this in AAStats.h. Setting the arrays to larger sizes will consume more dynamic memory (two bytes per additional point in the sample sets). This situation will be mitigated in a future release by more sophisticated memory allocation techniques.
LogSWRPointStats()	Logs the minimum, maximum, and standard deviation of the samples taken for a scan point.
CalculateStats()	Performs the calculations to determine the minimum, maximum, and standard deviation of the forward and reflected readings in the samples taken for a point in a scan.
LogScanStatsSummary()	Logs a summary containing the average minimum, maximum, and standard deviations of the forward and reflected readings in all the samples taken for all points in a scan. Also included are the minimum and maximum SWRs seen across all the points in a scan.

FindMaxMinSWR()	Tests for and saves the minimum and maximum SWR of all points in a scan. This is useful in determining and comparing baseline performance of the hardware when a resistive load is attached.