Barry Chambers, G8AGN • b.chambers@sheffield.ac.uk

A soundcard-less SDR panadapter based on the Arduino

Live spectrum and waterfall display without a PC

INTRODUCTION. Over the years, I have experimented with several types of software defined radio receivers including the SoftRock series and the Finningley SDR. These provide I and Q output signals that are usually processed and displayed using a PC and its associated soundcard, the software to do this being freely available on the web **[1]**. Having recently purchased a KX3, I sought an inexpensive route to acquiring an accompanying standalone panadapter and so first looked at the design by AA6E, which is based on the Raspberry Pi or Beaglebone boards **[2]**. This design again follows the traditional route of using a soundcard, with the Pi or Beaglebone dealing mainly with information display. Although I have been developing computer software for 50 years (I wrote my first code in 1964, using Ferranti Mercury Autocode), I found the Pi route frustrating, to say the least! Having worked my way through an eleven page guide to setting up the AA6E software, I found that it didn't run because the Pi's OS has been changed and there was now a missing dependency between the various software libraries. A search of the Pi's Forum pages did not reveal a solution to this problem so I decided instead to see what I could do with an Arduino-based system that was entirely software-based, ie it did not require a soundcard. It is worth pointing out that the resulting hardware will only display signals; unlike some PC and soundcard-based systems, it will not demodulate them, primarily because of processor and memory limitations. Nevertheless, it can still provide useful information to an operator in terms of band occupancy and may also be capable of use as a data logger for propagation monitoring.

HARDWARE. The success of a project such as this depends crucially on the choice of computer board and display module.

Superficially, the Raspberry Pi might seem to win hands down over the Arduino because of its much faster processor; however, Pi software is normally written in an interpretive language, Python, whereas the Arduino uses a compiled version of C++. When referring to the Arduino, most people normally think of the ubiquitous Uno board but its performance is inadequate for this project, not only in terms of processor speed (16MHz) and working memory (2kB SRAM) but also in terms of available IO pins (14). Instead, I used an Arduino Due board, which has an Atmel SAM3X8E ARM Cortex-M3 32 bit processor running at 84MHz, together with 96kb of SRAM and a total of 54 IO pins. A Due board can also do 12 bit ADC sampling, yet is available on eBay for less than £20.

The 5 inch TFT colour display module used in the panadapter has 800x480 RGB pixels, a touch screen and even a SD card port. Nevertheless, it too is available via eBay for less than £20. The module has a standard 40 pin header that is also used on several smaller and cheaper displays advertised on eBay but these typically have only 320x240 pixels. Initial software development was done using one of the smaller displays but because of the common pinout of the header, it was a simple matter to upgrade the display when the feasibility of the project was confirmed.

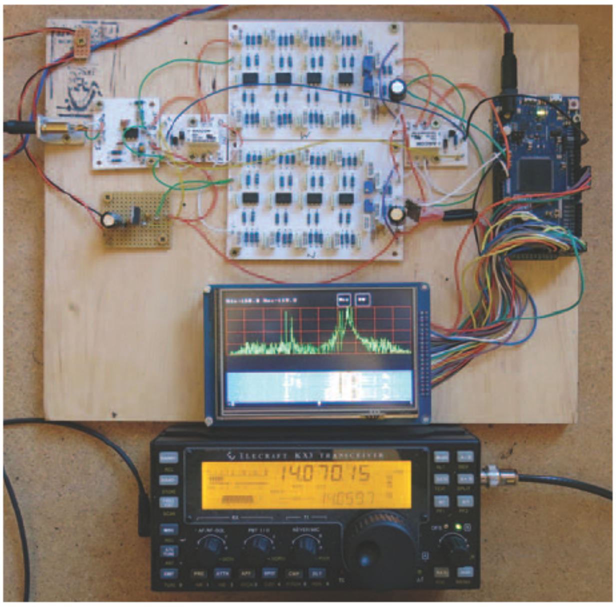
PHOTO 1: Breadboard prototype of Arduino based panadapter.

A block diagram of the panadapter is shown in **Figure 1**. As well as the Due board and TFT display, the only other hardware needed comprises single stage high gain audio amplifiers for each of the I and Q channels and their associated software-switchable active low pass filters. The latter are needed to reduce aliasing errors in the sampled I and Q data.

Figure 2 shows the circuit of the I and Q input amplifiers, each of which provides a nominal gain of about 50. **Figure 3** shows the generic circuit of one of the 8 pole active low pass filters. In total, four of these are needed: two for the I data channel and two for the Q data channel. The wideband filters have a nominal cut-off frequency of about 20kHz, the narrowband ones about 4kHz. The cut-off frequencies are set by changing the values of several capacitors and resistors **[3]** and the appropriate values for each filter are listed in **Table 1**. The signal paths through the appropriate filters are switched by software controlled relays using the circuit shown in **Figure 4**. If required, the individual I and Q channel gains can be set by multi-turn potentiometers connected between the outputs of the active low pass filters and

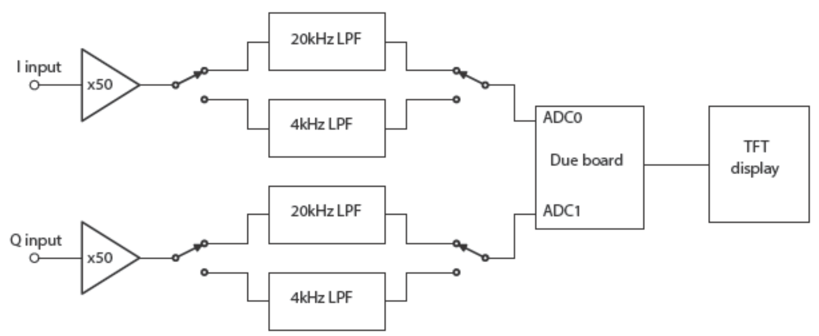

FIGURE 1: Block diagram of Arduino based panadapter.

Barry Chambers, G8AGN • b.chambers@sheffield.ac.uk

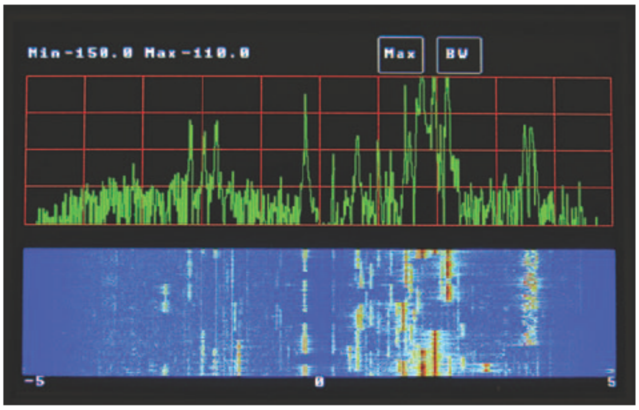

PHOTO 2: The panadapter displaying signals from the KX3 receiver tuned to 10.140715MHz (bandwidth set at ±5kHz).

the I and Q ADC inputs on the Due board. The connections between the latter and the TFT display are shown in Table 2. The display backlight is powered via the LED A connection, pin 19, on the 40 pin header. Preferably this should be connected to a separate 3.3V supply, but in practice the on-board Due 3.3V regulator seems able to cope with the required current demand. As supplied, the TFT display has two options for controlling the backlight brightness and these can be selected by bridging appropriate pairs of pads on the display board. However, I have only tried the option in which the backlight is fully on all the time.

SOFTWARE. Arduino programming is facilitated by the fact that there are many tutorials and code examples that can be downloaded from the web, together with code libraries to do or interface with almost anything. In the interest of brevity, I will discuss in detail only a few of the more important parts of the panadapter code, but the full listing contains many comments to enable a fuller understanding for those readers wishing to make modifications. The full code listing is available from me on request via e-mail.

The first part of the code is concerned with sampling the incoming low frequency I and Q data from the receiver. This data will be in the form of two AC waveforms but the ADC inputs on the Due board can only deal with uni-directional signals. The solution is thus to DC bias the ADC input pins at their midway point of 1.65V via a potential divider and to feed in the I or Q signal via a DC blocking capacitor. By default, the Due's ADC is configured to output 10 bits of data to represent each data sample, but this can be extended to 12 bits using the command *analogReadResolution (12);* Samples of the incoming I and Q data are acquired using the code loop shown in Listing 1.

Since the I and Q data samples are each represented by a 12 bit word (specifying one of 4096 voltage levels), these are then converted into two numbers each lying within the range ±1; thus the polarity of the original sampled AC waveforms is preserved.

Although this code fragment looks simple, two further factors need to be considered and these are to do with the rate at which the incoming I and Q data is sampled, since according to the Nyquist criterion, a signal with a bandwidth of B Hz needs to be sampled at a rate of at least 2B samples/sec. Thus the maximum signal bandwidth that can be displayed by the panadapter will depend on how fast the Due's ADC can acquire samples. There is also another requirement for fast sampling and this is to do with the fact that the I and Q data are not sampled simultaneously but rather sequentially since only a single ADC is used, together with a multiplexer. The resulting time delay between sampling first the I channel and then the Q channel will give rise to a phase error for the Q channel sample and this, together with any unbalance in the amplitude of the I and Q channel samples will give rise to an unwanted mirror image on the spectrum display. The code fragment in Listing 1 tries to minimise the phase error by first sampling the I and Q channels and then doing any processing that is required before saving the data in the array ***IQdata[ ].*** The second way in which the phase error can be reduced is by fine tuning certain constants that control the way the ADC operates. By default, a single ***analogRead()*** command takes about 39μs to implement but by manipulating the Cortex-M3 ADC MR (Mode Register) constants in software, this time can be reduced to about 4μs. The code to do this is shown in Listing 2.

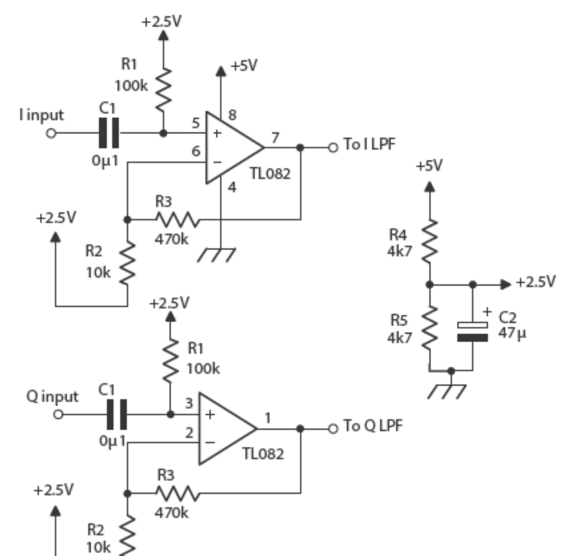

FIGURE 2: I and Q channel amplifiers.

Further details on this technique can be found in the Atmel SAM3X8E ARM Cortex-M3 CPU datasheet [4].

The time taken to acquire the desired number of samples of I data and Q data is noted and from this an estimate of the equivalent sampling frequency can be calculated. This is then trimmed to the required figure using additional delay inserted into the sampling loop using the command ***delayMicroseconds (Delay).*** The integer number ***Delay*** associated with this command must be determined empirically but, as a guide, a value of 65 in my code gave an effective sampling frequency of 13kHz (equivalent to a displayed spectrum of ±5kHz) and a value of 4 gave a sampling frequency of 62kHz (equivalent to a displayed spectrum of ±24kHz). It will be noted that the displayed spectrum width is less than might be expected from the quoted sampling frequencies. The reason for this will be discussed later.

Before further discussion of the software, we first need to consider what information will be displayed to the user. I chose to make both a spectrum display and a waterfall available simultaneously, but it would be a simple matter to alter the software to choose either one or the other. Irrespective of this, we should aim to utilise the TFT display pixels as effectively as possible. Hence I chose to use the display in landscape mode, ie the display axis corresponding to 800 pixels was used for the frequency axis. The TFT display can display up to 480 pixels of information along the other axis, so I chose to use 200 of these for the spectrum display amplitude axis, 200 for the waterfall display time axis and the remaining 80 for axis labels and soft keys.

A decision now has to be made about how many I and Q samples are to be processed each time a frequency spectrum or waterfall line is displayed. The I and Q samples represent snapshots of voltage waveforms in the time domain and these need to be transformed into information about the amplitudes of the individual frequency components making up the spectrum display. The most efficient way of doing this is by using the fast Fourier transform (FFT) but this can only deal with data sets made up of 2^N pairs of numbers. N is an integer, so for example if N=10, a data set would consist of 1024 I data samples and 1024 Q data samples, thus giving rise to 1024 frequency components. The TFT display,

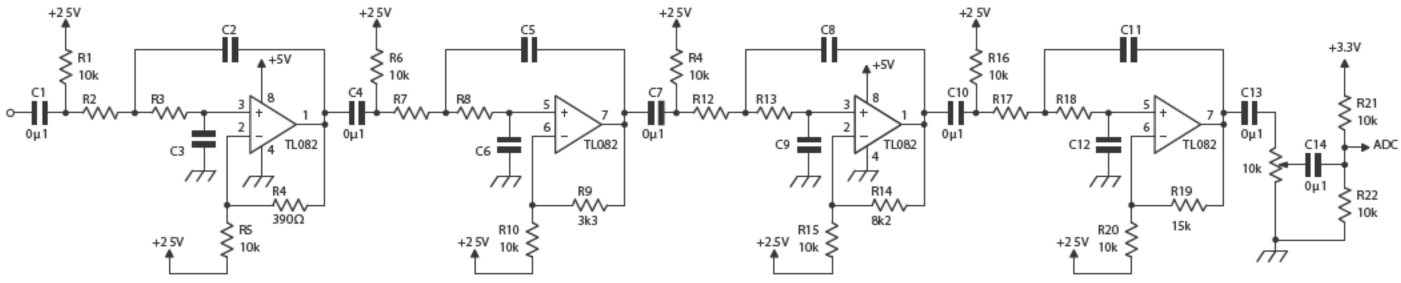

FIGURE 3: Generic low pass filter (see Table 1).

however, can only display 800 frequency components at a time so we either have to choose a value of N=9 (equivalent to 512 frequency points) or N=10. In the first case, not all the available pixels on the display will be used (512 out of 800) whereas in the second case, we can only display 800 out of the available 1024 frequency components. There are advantages in choosing the second of these alternatives; firstly, by using 1024 rather than 512 points in the FFT algorithm, the frequency resolution of the displayed spectrum is increased, even though we cannot display the full spectrum that a given sampling frequency provides. Also, the displayed I and Q data is effectively being oversampled and this will ease the task of the I and Q low pass filters; hence the reason why in the earlier discussion, it was mentioned that the displayed frequency spectrum did not seem to match up with the quoted sampling frequency. So for example, if the nominal sampling frequency were 50kHz, this would ordinarily result in a potential spectrum display of ±25kHz (the left and right hand sides of the spectrum are independent since we have sampled both the I and Q data at 50kHz). In practice, the actual displayed spectrum width would then be (800/1024) x 50kHz ≈ 39kHz.

Having finally decided on the number of I and Q data samples to be acquired, 1024, these are assembled into an array of complex numbers of the form real part=*IQdata[0],* imaginary part=*IQdata[1],* real part=*IQdata[2]*, etc. This array is then processed using the FFT to find the frequency spectrum of the sampled I and Q time waveforms. The resulting data is an array of complex voltage amplitudes for each frequency point in the spectrum. Conveniently, the Arduino IDE for the Due board contains a set of DSP algorithms specially written for the Due's ARM Cortex M3 processor, **[5]**, and so these are utilised wherever possible in the panadapter code. Further details are given in the first sidebar. The complex voltage amplitudes are then turned into powers by squaring and are normalised with respect to the maximum signal power that could be measured by the Due's ADC. This maximum power is that resulting from an ADC reading of ±2048. Finally, since a logarithmic amplitude display is required, each data point is processed accordingly.

A simplified code to carry out these operations is shown in **Listing 3**.

Turning now to the TFT display, the software for driving this is handled by a library called ***UTFT*** that can be downloaded from **[6]**. The library can drive a large number of different displays and contains routines for drawing text, figures, lines, circles etc as well as full control of colour and font. Since the TFT display also has an integrated touch screen, provision has been made to incorporate a number of soft keys to control such things as displayed spectrum width and display amplitude range. The handling of the soft keys is controlled by another library called ***UTouch*** that can also be downloaded from **[6]**.

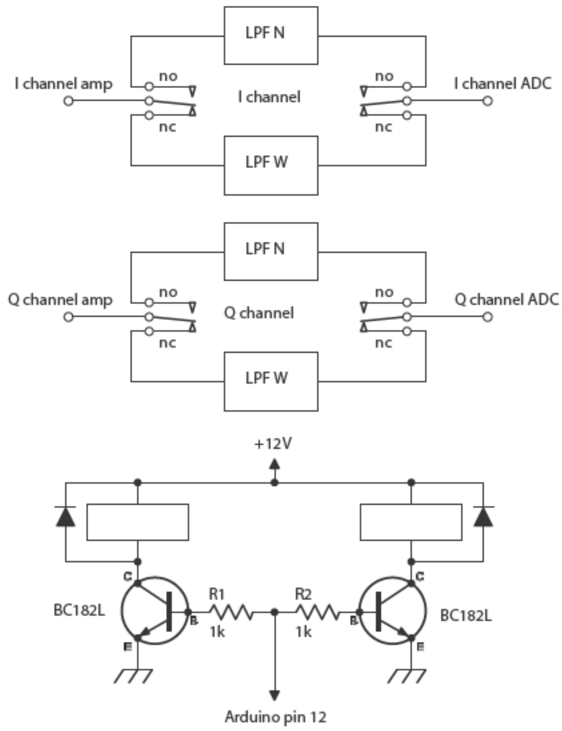

FIGURE 4: Relay driver for switching low pass filters.

SOFTWARE REFINEMENTS. If the software and hardware are used as described thus far, two artefacts will be evident on the display. Firstly, a large spike will always be seen at the centre of the display and this corresponds to a signal at DC. This arises from the fact that the I and Q waveform voltages will most probably have some associated DC shift associated with them. If this is not removed in the data processing then it will show as a central frequency spike.

The other artefact on the display will manifest itself as a mirror image of a wanted spectral component, which appears in the other half of the display. So for example, if a real frequency component appears at frequency +f in the right hand half of the display, an unwanted image of this may appear at –f in the left hand side of the display, or vice versa. This effect is caused by amplitude and phase errors occurring in an I and Q data pair. The amplitude error arises when the gains of the I and Q signal paths are unequal; the phase errors arise because the I and Q data channels are not sampled simultaneously but sequentially. Both the amplitude and phase errors will vary with the frequency being displayed so a correction at one frequency will not hold exactly at another. There are schemes that can provide correction at all displayed frequencies but these take up far more computing resources than we have at our disposal; hence I adopted a correction scheme that uses the received spectral component with the largest amplitude at any given time as a test signal **[7]**. The resulting amplitude and phase corrections to the displayed spectrum and waterfall are thus only valid, in theory, at a single frequency but in practice the results are quite acceptable over the full displayed frequency range. Furthermore, the corrections are made automatically and are dynamic, ie they change from one data set to the next. Because the mathematics of the error correction process is rather involved, it will not be discussed here but interested readers will find fuller details in the box on page 30.

IMPLEMENTATION. A breadboard prototype of the complete panadapter is shown in **Photo 1**. The I and Q channel amplifiers, low pass filters and relay switching circuits were all built on single-sided printed circuit boards using wire-ended components. As can be seen from the photo, the resulting boards are rather large and would benefit from a redesign using SMD parts.

The panadapter needs voltage supplies of 5V, 3.3V and 2.5V. The first two of these can be obtained from the Arduino board itself and the third is provided from the 5V supply via a simple potential divider, as shown in **Figure 2**.

Photo 2 shows a close up view of the TFT screen when the panadapter was connected to my KX3 and the latter was being used to monitor signal activity around 10.14MHz. In this case, the bandwidth of the display was set at ±5kHz. From left to right, the soft keys at the top of the display can be used to change the amplitude range of the displayed spectrum and waterfall displays and the displayed bandwidth (toggled between ±12kHz and ±5kHz). The current value of the former is displayed to the left of the soft keys.

CONCLUSIONS. When I started this project, I totally underestimated the time and effort it would consume but the end result has made it worthwhile. Although not yet boxed-up, the panadapter is already proving to be a useful addition to my KX3. Possible modifications might include increasing or decreasing the displayed frequency bandwidth, adding more soft keys for increased functionality or utilisation of the SD card slot on the TFT display. The latter would then enable the panadapter to be used as a beacon monitor and data logger for propagation studies.

TABLE 1: Component values for low-pass filters (all other values are common to both filter types)

Component	Narrow band 4kHz	Wide band 20kHz
R2, R3, R7, R8, R12, R13, R17, R18	8k2	8k2
C2, C3, C5, C6, C8, C9, C11, C12	4n7	1n0

TABLE 2: Connections between the Arduino Due board and the TFT display.

TFT pin	TFT function	Arduino Due pin
1	0V	
2	VCC 3.3V	
3		
4	RS	D38
5	WR	D39
6	RD Connect to 3.3V	
7	DB8	D22
8	DB9	D23
9	DB10	D24
10	DB11	D25
11	DB12	D26
12	DB13	D27
13	DB14	D28
14	DB15	D29
15	CS	D40
16		
17	RST	D41
18		
19	LED A Connect to 3.3V. Link 'always on' pads on back side of TFT module	
20		
21	DB0	D37
22	DB1	D36
23	DB2	D35
24	DB3	D34
25	DB4	D33
26	DB5	D32
27	DB6	D31
28	DB7	D30
29	T CLK	D6
30	T CS	D5
31	T DIN	D4
32		
33	T DOUT	D3
34	T IRQ	D2

WEBSEARCH
[1] Rocky SDR software www.dxatlas.com/rocky/
[2] A Tiny *Python* Panadapter, Martin Ewing, AA6E, *QST* April 2014
[3] *Active Low Pass Filter Design*, Texas Instruments www.ti.com/lit/SLOA049
[4]. Atmel SAM3X8E ARM Cortex M3 CPU data sheet www.atmel.com/Images/doc11057.pdf
[5] CMSIS DSP library for ARM Cortex M3 processor www.keil.com/pack/doc/CMSIS/DSP/html/index.html
[6] *UTFT* and *UTouch* libraries http://henningkarlsen.com/electronics/library.php?id=52
[7] IQ correction Churchill F E, Ogar G W and Thompson B J, The correction of I and Q errors in a coherent processor, *IEEE Trans Aerospace and Electronic Systems*, AES 17, No 1, Jan 1981, pp 131 137
[8] Enabling the Arduino Cortex M3 DSP functions http://forum.arduino.cc/index.php?PHPSESSID=t46sbolt2km849unr8jgfmjdc1&topic=140107.0

Enabling the Cortex M3 DSP functions

The following information has been taken from **[8]**.

1. Find the text file ***platform.txt*** at ***C:/Program Files (x86)/Arduino/hardware/arduino/sam/platform.txt***

2. Using Notepad, amend the file section called

Combine gc sections, archives, and objects

so it now includes a reference to a file called ***libarm_cortexM3l_math.a***

Combine gc sections, archives, and objects
recipe.c.combine.pattern="{compiler.path}{compiler.c.elf.cmd}" {compiler.c.elf.flags} mcpu={build.mcu} " T{build.variant.path}/{build.ldscript}" " Wl, Map,{build.path}/{build.project_name}.map" o "{build.path}/{build.project_name}.elf" " L{build.path}" lm lgcc mthumb Wl, cref Wl, check sections Wl, gc sections Wl, entry=Reset_Handler Wl, unresolved symbols=report all Wl, warn common Wl, warn section align Wl, warn unresolved symbols Wl, start group "{build.path}/syscalls_sam3.c.o" {object_files} "{build.variant.path}/{build.variant_system_lib}" "{build.variant.path}/libarm_cortexM3l_math.a" "{build.path}/{archive_file}" Wl, end group

3. Navigate to the IDE folder ***C:/Program Files (x86)/Arduino/hardware/arduino/sam/system/CMSIS/CMSIS/Lib/GCC/***

In that folder, you should find a file called ***libarm_cortexM3l_math.a***

Make a copy of this file and place it in the IDE folder ***C:/Program Files (x86)/Arduino/hardware/arduino/sam/variants/arduino_due_x/***

The Cortex M3 DSP functions should now be accessible.

Barry Chambers, G8AGN • b.chambers@sheffield.ac.uk

Correction of I and Q errors

This discussion is an abridged version of that found in [7].

A simplified and error free version of the I and Q data being passed into the panadapter can be written as

$$I(t) = A\cos\omega t \text{ and } Q(t) = A\sin\omega t$$

This pair of signals can be thought of as a complex signal

$$I(t) + jQ(t) = A\,e^{j\omega t}$$

In practice the received I and Q data will contain errors such that

$$I_1(t) = (1+\epsilon)A\cos\omega t + a \text{ and } Q_1(t) = A\sin(\omega t + \varphi) + b$$

where

ε is the fractional amplitude imbalance

φ is the phase imbalance

a is the DC offset in the I channel, and

b is the DC offset in the Q channel.

a and *b* can be corrected by subtracting the average level from the signal in each channel.

Then we have

$$I_2(t) = (1+\epsilon)A\cos\omega t \text{ and } Q_2(t) = A\sin(\omega t + \varphi)$$

The I_2 and Q_2 signals are treated as vectors and two correction coefficients P and E_1 are required, P for rotating one vector and E_1 for scaling the other. Then the corrected signals I_3 and Q_3 are related to I_2 and Q_2 by

$$I_3 = E_1 I_2 \text{ and } Q_3 = P\,I_2 + Q_2$$

It can be shown that the required form for E_1 and P is

$$E_1 = \frac{\cos\varphi}{(1+\epsilon)} \text{ and } P = \frac{-\sin\varphi}{(1+\epsilon)}$$

After application of the correction process, the final signals are

$$I_3(t) = A\cos\varphi\cos\omega t \text{ and } Q_3(t) = A\cos\varphi\sin\omega t$$

An amplitude scaling factor cos φ has been introduced but since it is common to both I_3 and Q_3 this is not of importance.

The procedure to obtain the correction coefficients is as follows:

The I and Q signals are each sampled 1024 times and the data is stored in array ***IQdata1[]***. The 2048 data values are also stored in array ***IQdata2[]*** for future use. The data in ***IQdata1[]*** is next FFTed as outlined in the previous discussion of the panadapter software. Now the ***IQdata1[]*** array contains 1024 complex frequency values arranged in pairs (real part, imaginary part, real part, imaginary part…) in array locations [0] to [2047].

The DC offsets in the original I and Q sampled time waveforms are obtained from

$$a = \frac{IQdata1[0]}{1024} \text{ and } b = \frac{IQdata1[1]}{1024}$$

and these values will later be subtracted from all the data contained in array ***IQdata2[]***.

To obtain the correction coefficients E_1 and P, we need to locate the complex frequency component in array ***IQdata1[]*** which has the largest magnitude and to note its location within the array as a pair of indices, eg ***IQdata1[2*i]*** real part of frequency, ***IQdata1 [2*i+1]*** imaginary part of frequency. This frequency component can be thought of as our test signal and let us assume that the values of its real and imaginary parts are

$$IQdata1[2*i] = W \text{ and } IQdata1[2*i+1] = X$$

Then the unwanted image of this test signal will be located within the array ***IQdata1[]*** at

IQdata1[2048 2*i] real part of image frequency and

IQdata1[2048 2*i+1] imaginary part of image frequency.

The associated data values are

$$IQdata1[2048 - 2*i] = Y \text{ and } IQdata1[2048 - 2*i+1] = Z$$

It can be shown that the correction coefficients E_1 and P are related to W, X, Y and Z by

```
denom = 2./((W+Y)*(W+Y)+(X Z)*(X Z));
e1 = 1. (Y*(W+Y) Z*(X Z))*denom;
p =  (Z*(W+Y)+Y*(X Z))*denom;
```

Finally, we can apply these correction factors to the copy of the original sampled I and Q data which was previously stored in array ***IQdata2[]***.

```
// IQ correction to copy of original sampled I and Q data
 for (uint16_t i = 0; i < fftSize; i++)
 {
// remove DC shift
IQdata2[2*i]=IQdata2[2*i] a;
IQdata2[2*i+1]=IQdata2[2*i+1] b;

// scale and rotate I and Q vectors
IQdata2[2*i+1]=p*IQdata2[2*i]+IQdata2[2*i+1];
IQdata2[2*i]=e1*IQdata2[2*i];
 }
```

This data is now FFTed to obtain the corrected frequency spectrum which will subsequently be displayed.

Listing 1: Simple code loop to sample I and Q data.

```
Const uint32_t fftSize = 1024;
const float weight = 1. / 2047.;
for(uint16_t i=0;i<fftSize; i++) {
rawI = analogRead (0);
rawQ = analogRead (1);
IQdata[2*i] = float(rawI   2047) * weight;
IQdata[2*i+1] = float(rawQ  2047) *weight;
}
```

Listing 2: Reducing the ADC read time.

```
// modify ADC_MR register to make ADC read
      faster
// change STARTUP from 8 to 2 (512 to 16
      periods of ADC clock)
 REG_ADC_MR = (REG_ADC_MR &
      0xFFF0FFFF) | 0x00020000;
// change PRESCAL from 2 to 1
 REG_ADC_MR = (REG_ADC_MR &
      0xFFFF00FF) | 0x00000100;
```

Listing 3: Converting the sampled I and Q data to a frequency spectrum.

```
const uint32_t fftSize = 1024;
const uint32_t ifftFlag = 0;
const uint32_t doBitReverse = 1;
// Initialize the CFFT/CIFFT module
arm_cfft_radix4_init_f32(&S, fftSize, ifftFlag,
      doBitReverse);
/* Process the complex IQ data in array
      IQdata[ ] through the CFFT/CIFFT module
      to implement FFT */
arm_cfft_radix4_f32(&S, IQdata);
/* Process the data through the Complex
      Magnitude Module for
calculating the magnitude at each bin.
Return data in array Spectrum[]; This
      contains 1024 points */
arm_cmplx_mag_f32(IQdata, Spectrum,
      fftSize);
/* Normalise the frequency data and convert
      to log form */
for (uint16_t i=0; i<fftSize; i++){
Spectrum[i]=20.0*log10(Spectrum[i])
      maxSignal;
}
```