AudioSDR – A Single-block SDR Processor for the Teensy Audio Library

Derek Rowell[1]

1 Introduction

`AudioSDR` is a single-block, Teensy Audio Library compatible, signal processor/demodulator for use in direct-conversion software-defined-radio (SDR) systems. `audioSDR` provides high quality demodulation for LSB, SSB, AM (using envelope detection or SAM), and CW (using both LSB and USB modes). It is designed to work at the standard Teensy Audio Library sampling rate of 44.1kHz, uses the standard 128 sample block size. It takes two inputs (the I and Q quadrature signals) and produces demodulated audio output (single channel). In its basic form it requires no other Audio block objects.

The `audioSDR` block has the following features:

- It uses floating point (f32) processing throughout the signal processing chain.
- It uses a dual-conversion approach with an intermediate frequency (IF) at approximately 7kHz.
- It contains demodulators for single sideband (USB and LSB), AM (using complex envelope, or PLL based SAM, and CW using USB or LSB).
- It includes an impulse noise-blanker with adjustable threshold.
- It has AGC with variable attack, release and hang times.
- It contains an ALS (adaptive-least-squares) automatic notch/peak filter.
- It (currently) has nine selectable output audio band-pass filters.

It should be understood that `audioSDR` is **not** a complete stand-alone SDR receiver - it is a software component that requires additional hardware to generate quadrature (I and Q) inputs in the range ±22.05 kHz. The `class audioSDR` includes an extensive set of public methods to control all the internal operation, but the user must provide the software for the interaction with the block, and control of a front panel and display. It should be noted that the current version uses floating point (f32) arithmetic for all processing in the signal chain, and is therefore only suitable with the Teensy 3.6.

Figure 1 shows the placement of the `audioSDR` block in an SDR processing chain. The figure shows two Audio Blocks: `AudioPreProcessor` and `AudioSDR`. `AudioPreProcessor` is an optional separate block (included in the package and described later in this document) that serves to correct errors in the Teensy raw quadrature data streams from the codec. The main processing block is `AudioSDR`. It accepts I and Q data at the standard Teensy sample rate (44.1 kHz) and outputs demodulated and processed audio, with demodulation/signal processing controlled through the public functions.

[1] August 23, 2019

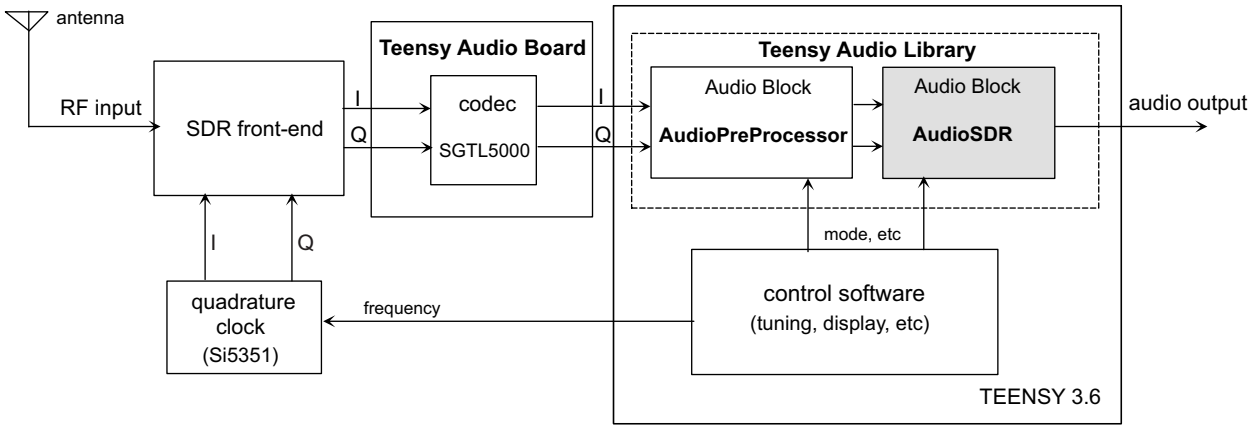

Figure 1: Placement of the `audioSDR` block in a minimal Teensy Audio Board based SDR.

`audioSDR` is (and always will be) a work-in-progress. I will be continually improving and adding features. I welcome anybody to join with me in making this a collaborative effort. Current efforts are directed toward migration to the Teensy T4.

1.1 A Brief Note on Performance

The performance of any SDR is, of course, very dependent on the receiver front-end. I use a largely "home-brew" design, with an *SDR Cube* rf pre-amp, a double-balanced QSD mixer driven by an SI5351 quadrature oscilator, and 4th-order active elliptic anti-aliasing filters.

While I have a reasonable set instruments for bench measurements, I live in a very noisy rf environment for on-air testing. In order to qualitatively evaluate the system, I use a commercial SDR (SDRplay RSP1) in conjunction with the *SDR-Console* PC software package as my gold standard. I use an indoor magnetic loop antenna, with an antenna switch to move between the `audioSDR` and the RSP1. After many hours of side-by-side comparison, I am hard-pressed to tell any qualitative difference in terms of sensitivity, SNR, sideband rejection, adjacent signal rejection, etc between the two systems.

In addition I regularly test the system as a receiver node on the WSPR (**W**eak **S**ignal **P**ropagation **R**eporter) network. See Appendix B for a minimal AudioSDR WSPR receiver included in this package. I get very good reception from Europe, and of course the USA, but also South Africa, Australia, and occasionally New Zealand. (See Fig. 3 at the end of this document for a typical overnight recording session.) These are extremely weak signals, as low as -30 dB below the receiver noise level!

2 System Description

Internally `audioSDR` is a **dual-conversion** SDR receiver. The quadrature input from the external RF front-end, sampled at 44.1 kHz, has a *complex* bandwidth of -22.05 kHz to +22.05kHz. The initial processing is done at a low IF (intermediate frequency) centered at

+6890 Hz [2] before being translated down to baseband for demodulation. The reason for this is three-fold: 1) to eliminate problems caused by the fact that the Teensy Audio Board codec is ac coupled, 2) to avoid noise issues associated with processing rf signals at (or around) 0 Hz, and 3) to avoid the possibility of direct feed-through of radiated rf from the external rf oscillator, into the tuning band of the receiver. The major signal flow is shown in Fig. 2.

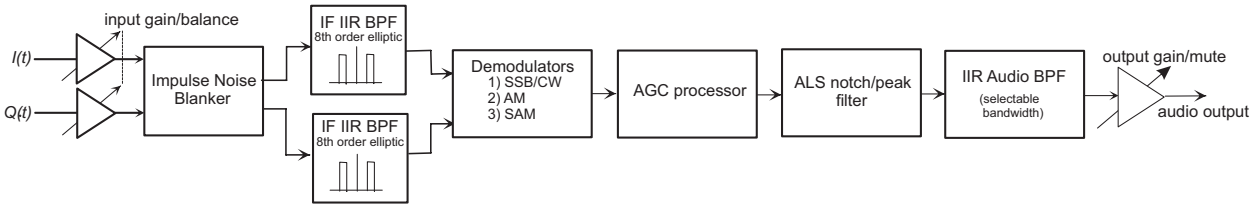

Figure 2: The signal-processing chain within the AuduiSDR block.

`AudioSDR` is a conventional Audio Library object with two inputs (the quadrature I and Q inputs), and a single output (the demodulated audio). `AudioSDR` input and output signals are in the standard q15 integer format, although internally the processing is done in 32-bit floating point (f32) format. There is no decimation, and all processing is done at the 44.1 kHz sampling frequency.

The input processing includes an adjustable input gain, and a differential gain to optimize sideband suppression. The processing chain then includes:

The Impulse Noise Blanker: The noise blanker is used to eliminate short, high-amplitude signal bursts such as generated by lightning, or electrical sparks from a faulty car ignition. It is placed first in the processing chain to provide easy impulse detection (without degradation by filtering operations). The I and Q channels are examined sample-by-sample for sudden increases in amplitude, and a binary mask is created (set to 0 if an impulse is detected). The mask is then processed to 1) include extra samples immediately before and after the impulse, and 2) to add smooth (raised cosine) transitions between 1's or 0's in the mask. Finally the data block is multiplied by the processed mask to effectively "blank" the signal for the duration of the impulse. The threshold for blanking is adjustable. Generally, with occasional pulse detections, the effect of blanking the signal is inaudible.

IF Band-Pass Pre-Filters: Each channel is filtered by an 8th order, elliptic IIR band-pass filter with a bandwidth of 1, 3, or 8 KHz, for CW. SSB, or AM respectively.

Demodulators: Three separate demodulation schemes are provided for SSB/CW, AM, and Synchronous AM (SAM):

[2]This particular frequency choice was based on other projects I am undertaking

SSB/CW Demodulation This version of `audioSDR` uses the Hartley (or phase-shift) method for SSB/CW demodulation (a Weaver demodulator is available but not included in this package). The signal is tuned so as to align the SSB signal within the IF pre-filter pass-band, with a USB signal aligned to the lower cut-off frequency, and a LSB signal aligned to the upper cut-off. The IF filter output is then shifted to baseband by complex multiplication, and the Q channel is shifted in phase by 90° by a length 257 Hilbert FIR filter (with delay compensation on the I channel). The demodulated audio output is then formed as the sum or difference of the I and Q channels.

AM demodulation The basic AM detector is a standard quadrature envelope detector. The IF signal is downshifted to baseband, passed through an anti-image low-pass IIR filter, and the and the demodulated audio is computed as the complex magnitude of the AM envelope.

$$f_{\text{audio}}(t) = \sqrt{I^2(t) + Q^2(t)}$$

SAM demodulation Synchronous AM (SAM) demodulation uses a software PLL (phase-locked-loop), locked to the AM carrier at the IF frequency, to down-shift the signal to baseband, so that the carrier is centered at 0 Hz. Should the PLL fail to gain lock, or lose lock, the demodulator reverts to the normal envelope detection described above.

Automatic Gain Control (AGC): `audioSDR` uses AGC based on 1) the strength of the demodulated audio for SSB/CW, or 2) the carrier amplitude for AM. The block acts as an audio compressor with three gain regions:

- A fixed gain if the signal strength is below the AGC threshold,
- An amplitude dependent gain reduction for strong signals, and
- A rounded "knee" transition region between the above weak/strong signal regions.

There are three standard setting for the attack/hang/release times ("fast", "medium", "slow"), but finer control is available through a set of public user functions.

ALS Notch/Peaking Filter An adaptive-least-squares (ALS) FIR filter is a filter whose coefficients are continually adjusted (automatically) to optimally achieve a given task. `audioSDR` contains such a filter that can be switched between two functions:

- Detect and minimize a narrow-band interference signal in the presence of a wide-band signal or noise, thus acting as an automatic "notch" filter.
- Detect and maximize a narrow-band signal in the presence of noise, thus acting as an automatic "peaking" band-pass filter which can be an effective noise-reduction filter.

The ALS action is based on "least-squares" statistical optimization and, depending on the relationship between the desired signal and the interference, the performance will vary. While not perfect in all situations, they can be very effective.

Audio Band-Pass Filters: The audio output is passed through an elliptic 8th-order IIR band-pass filter with selectable bandwidth. Default filter bandwidths are chosen for each demodulation mode (600Hz for CW, 2.7 kHz for SSB , and 3.9kHz for AM), but a total of nine different filters are provided, and may be selected through a public function.

Audio Output: The audio output has a variable gain (for use as a volume control), and a provision for muting the output.

3 Setting up and Using `audioSDR`

3.1 Requirements

The requirements for including `audioSDR` into a project are:

- A Teensy 3.6 and a Teensy Audio Board (or other codec board compatible with the Audio library).
- `audioSDR` uses the DSP functions in the `arm_math` library. It may be necessary for you to upgrade to a more recent version than that supplied with Teesnyduino. The `audioSDR` package includes the required files and instructions to make the upgrade.

3.2 Tuning Offset

All multi-conversion receivers have an inherent difference between the tuning oscillator frequency and the actual frequency tuned. For example, assume that we are tuning to an SSB signal at 7.150 MHz. With a direct-conversion receiver we would simply set the rf oscillator to 7.150 MHz to translate the signal to baseband. However, in the case of `audioSDR` the initial translation is to the IF, centered on 6890 Hz. To tune an AM station on audioSDR we want the carrier to end up at 6890 HZ, therefore we need to set the rf oscillator to 7,150,000 - 6890 = 7,143,110 Hz. Similarly for tuning an LSB signal it must be aligned with the **upper** edge of the SSB 3 kHz IF filter, that is 6890 + (3000/2) = 8390 Hz, and the rf oscillator must be set to 7,150,000 - 8390 = 7141610 Hz. There is, therefore, a **mode dependent tuning offset** that needs to be applied to the rf quadrature oscillator. `AudioSDR` computes the offset each time a demodulation mode is selected. To tune to a frequency f_0 you should

1. Set the front panel display to show f_0, and
2. Set the rf oscillator frequency to $f_0 - f_{\text{offset}}$.

where f_{offset} has been supplied by `audioSDR` by a call such as

```
tuningOffset = mySDR.setDemodMode(LSBmode)
```

3.3 Inclusion of `audioSDR`

`audioSDR` is a standard Teensy Audio Library object with two inputs and a single output. It is included into a project (sketch) as:

```
#include <Audio.h>
#include "AudioSDR.h"
//
AudioInputI2S         input;
AudioOutputI2S        ouput;
AudioSDR              mySDR;
AudioControlSGTL5000  codec;
//
AudioConnection c1(input, 0,  mySDR,  0);
AudioConnection c2(input  1,  mySDR,  1);
AudioConnection c1(mySDR, 0,  output, 0);
AudioConnection c1(mySDR, 0,  output, 1);
//
```

Note: The input quadrature I channel **must** be connected to I2S channel 0 (left), and the Q data is on channel 1 (right). Failure to meet this requirement will make it impossible to tune signals. This can be corrected in hardware connections, or software.

3.4 Control of AudioSDR

The `audioSDR` class includes a large number of **public** methods (functions) to set up, and control of its operation. These are invoked by prefix the function with the name of your instance, for example with the above header you might call:

```
tuning_offset = mySDR.selectDemodMode(LSBmode);
mySDR.setALSfilterNotch());
```

somewhere in your `setup()` or `loop()`.

3.4.1 Public Functions for General Set up

int32_t tuning_offset = selectDemodMode(int16_t mode) - Set the current demodulation mode as follows:

	`USBmode`	(or 0)	for USB
	`LSBmode`	(or 1)	for LSB
mode =	`CW_USBmode`	(or 2)	for CW_USB
	`CW_LSBmode`	(or 3)	for CW_LSB
	`AMmode`	(or 4)	for AM
	`SAMmode`	(or 5)	for synchronous AM

Return the tuning offset (in Hz)to applied to the RF quadrature oscillator for the new mode (See Sec. 3.2 above)..

int32_t currentMode = getDemodMode(void) - Return the current demodulation mode as one of the numeric values above.

float32_t currentTuningOffset = getTuningOffset(void) - Return the tuning offset (in Hz) for the current demodulation mode.

void setMute(bool muted) - Control the SDR audio output muting, mute if `muted = true`; release from mute if `muted = false`

boolean getMute() - Return the current audio output mute state of the SDR, `true` if muted.

void setInputGain(float32_t Gain) - Set the gains of the two quadrature (I and Q) input channels (default is 1.0).

void setIQgainBalance(float32_t GainBalance) - Set a differential gain between the two quadrature input channels so that $\text{gain}_I = \text{gainBalance} \times \text{gain}_Q$.

void setOutputGain(float) - Set the output audio gain. May be used as a volume control for the SDR, (default = 1.0).

float32_t IFbandLowerCutOff = getBPFlower(void) - and

float32_t IFbandUpperCutOff = getBPFupper(void) - return the upper and lower cut-off frequencies of the current IF band-pass filters. Useful for displaying the tuning band limits on a panadapter (spectrum) display.

3.4.2 Public Functions for the Synchronous AM (SAM) demodulator

float32_t getSAMfrequency(void) - Return the current frequency of the SAM phase-locked-loop (PLL).

boolean getSAMphaseLockStatus(void) - Return `true` if the SAM detector PLL is locked to the carrier of an AM signal, `false` otherwise.

3.4.3 Public Functions for the ALS Notch/Peak Filter

void enableALSfilter(void) - Enable the ALS filter, set in *adaptive* mode, with *notch* operation.

void disableALSfilter(void) - Set the ALS filter to *by-pass* mode.

boolean ALSfilterIsEnabled(void) - Return the current *enabled/disabled* status.

void setALSfilterNotch(void) - Use the ALS filter as a notch filter.

void setALSfilterPeak(void) - Use the ALS filter as a narrow-band peak filter. Useful for noise reduction.

boolean ALSfilterIsNotch(void) - Return `true` if the filter is in *notch* mode.

boolean ALSfilterIsPeak(void) - Return `true` if the filter is in *peak* mode.

void setALSfilterAdaptive(void) - Set the filter to automatically adjust its coefficients for optimal *notch/peak* response.

void setALSfilterStatic(void) - Continue filtering operation with the current filter coefficients, that is cease adaptive operation.

boolean ALSfilterIsAdaptive(void) - Return `true` if the filter is in *adaptive* mode.

3.4.4 Public Functions for the AGC processor

void setAGCmode(int16_t mode) - Set AGC operation to one of three standard settings:

	`AGCoff`	(or 0)	(AGC is disabled)
mode =	`AGCfast`	(or 1)	(t_{attack} = 2 ms, t_{hang} = 100 ms, $t_{release}$ = 100 ms)
	`AGCmedium`	(or 2)	(t_{attack} = 5 ms, t_{hang} = 250 ms, $t_{release}$ = 500 ms)
	`AGCslow`	(or 3)	(t_{attack} = 10 ms, t_{hang} = 500 ms, $t_{release}$ = 2000 ms)

void enableAGC(void) - Enable the AGC processor.

void disableAGC(void) - Disable the AGC processor.

bool AGCisEnabled(void) - Return `true` if the AGC processor is running.

bool AGCisActive(void) - Return *true* if the current signal strength exceeds the threshold for AGC processing.

void setAGCoutputGain(float gain) - Set a static gain to be applied to the AGC output to compensate for the compression.

void setAGCattackTime(float attackTime_msec) - (For advanced users) Defines the approximate time (in msecs) the AGC processor takes to fully respond to a sudden increase in signal strength. (Usually a small value.)

void setAGChangTime(float hangTime_msec) - (For advanced users) Defines the approximate time (in msecs) the AGC processor dwells at its current gain berore reacting to s sudden decrease in signal strength.

void setAGCreleaseTime(float releaseTime_msec) - (For advanced users) Defines the approximate time (in msecs) the AGC processor takes to fully respond to a sudden decrease in signal strength, after the hang time has expired. (Usually a much longer time than the attack time.)

3.4.5 Public Functions for the Impulse Noise Blanker

void enableNoiseBlanker(void) - Enables the noise blanking on the incoming signal.

void disableNoiseBlanker(void) - Disables the noise blanker.

bool NoiseBlankerisEnabled(void) - Returns the operational status of the noise blanker.

void setNoiseBlankerThreshold(float nbThreshold) - Sets the threshold for pulse detection. The noise blanker keeps a short-term running average of the signal strength. An impulse is detected when the signal suddenly exceeds a level of `nbThreshold` × `average_signal_strength`.

void setNoiseBlankerThresholdDb(float nbThreshold_dB) - Similar to the case above, but the threshold value is set to the decibel level above the running average signal level.

3.4.6 Public Functions for the IIR Audio Output Filters

void setAudioFilter(int16_t audioFilter) - select an output audio band-pass filter. The available choices are:

`audioAM`	(or 0):	300 - 3900 Hz for AM
`audioCW`	(or 1):	600 -1000 Hz for narrow-band CW reception
`audioWSPR`	(or 2):	1300 - 1700 Hz for WSPR reception
`audio2100`	(or 3):	300 - 2100 Hz
`audio2300`	(or 4):	300 - 2300 Hz
`audio2500`	(or 5):	300 - 2500 Hz
`audio2700`	(or 6):	300 - 2700 Hz
`audio2900`	(or 7):	300 - 2900 Hz
`audio3100`	(or 8):	300 - 3100 Hz
`audio3300`	(or 9):	300 - 3300 Hz
`audioByPass`	(or 10):	filter is inactive, set to "pass-through" mode.

The default filter is set by the demodulation mode: `CWfilter` for CW, `audio2700` for SSB, and `AMfilter` for AM.

int16_t getAudioFilter(void) - return the numeric identifier (see above) of the current active audio filter.

void enableAudioFilter(void - enable the audio filter.

void disableAudioFilter(void) - disable the audio filter, that is set it in "pass-through" mode.

4 Two Useful (included) Audio Library Objects

4.1 Class AudioSDRpreprocessor

Although not strictly a part of the `audioSDR` package, this audio block is **necessary** to detect and eliminate a nasty bug in the Teensy 3.6 two channel I2S input through an audio codec.[3]

The bug appears randomly (with about 50% probability) on program upload and (less frequently) on power-up, and once present remains until a reload or a power cycle.

[3] I do not mean to imply that the bug is in PJRC's software as it has been reported on many ARM MCUs with different codecs.

What happens is very simple: the system remains fully functional, **but the samples in the two I2S data streams from the codec become out alignment with each other, by a single sample**, or in other words a time delay of one sample is introduced between the I and Q data. While this may be not important for many audio/music processing applications, it completely destroys the essential phase relationships for quadrature SDR operation.

When the fault is present:

- SSB signals become impossible to demodulate.
- AM signals become extremely noisy.
- Image phantom signals appear in the tuning.
- The data spectrum on a panadapter display becomes symmetrical with a false peak appearing in the negative frequency spectrum (leading to the naming of the bug by some as the "twin peaks" problem.)

The main purpose of `AudioSDRpreprocessor` is to detect and compensate for the inter-channel delay, but it also includes a separate function to optionally swap the I and Q input channels to make sure the I signal is connected to I2S input 0.

Compensation for the single sample delay is easy: simply delay the non-delayed channel by a single element to bring them back into alignment, but auto-detection of the bug is more difficult. I have found that the most reliable method is to look for symmetry in the FFT of the data stream.

On each data block the error detection algorithm uses an FFT to look for symmetry about 0 Hz in the power spectrum (there should be none). If symmetry is detected (twin-peaks) there is an error, and it changes the compensation and tries again on the next block. If it does not find symmetry after 1000 successive attempts with the same compensation, the current compensation is declared the correct choice and the algorithm exits, leaving the compensation in the correct state.

The algorithm requires a "reasonably" strong coherent signal in the two input streams, and it will not work with just uncorrelated noise in the two channels. It maybe necessary to tune around a little.

The public functions for `AudioSDRpreprocessor` are:

void startAutoI2SerrorDetection(void) - Initiate automatic detection and compensation for the I2S error condition. See the note above.

void stopAutoI2SerrorDetection(void) - If the automatic error detection is active, stop it and set no compensation. If inactive take no action.

bool getAutoI2SerrorDetectionStatus(void) - Return `true` if the automatic error detection is in progress return `false` otherwise.

void setI2SerrorCompensation(int16_t compensation) - Set the error compensation mode according to `compensation`:

If `compensation` =		
	+1	- add a single sample compensating delay to the Q channel.
	0	- Do not add compensation to either channel.
	-1	- (This case has not been necessary in practice), add a single sample compensating delay to the I channel

int16_t getI2SerrorCompensation(void) - Return +1 if a compensation delay is currently add to the Q channel, return 0 if no compensation is currently applied, and return -1 if a compensation delay is currently added to the I channel.

void swapIQ(boolean swap) - Reverse the standard channel definition if `swap = true`, otherwise use the standard definition (I data on I2S channel 0). This function is not related to the I2S problem, but is useful while debugging hardware and software compatibilty,

4.2 Class AudioGrabberComplex256

I have found that this class, while non-essential, is very useful in my SDR work. It is designed to extract buffers of data from the Teensy Audio Library data stream to the external environment, say in `loop()`. I have several variants for various buffer lengths and data types, but I include here just one which returns IQ data in the arm_math.h complex data format (`real, imag, real, imag,....`) as 256 `int16_t` complex numbers in a length 512 buffer.

It is not intended as a source of contiguous data, but for extracting "occasional" blocks. In my own SDR I use the "grabber" (as I call it) within the `loop()` to extract a length 256 complex buffer of data every 50 msec, which is then processed through an FFT to update the panadapter (spectral) and signal strength display on the front panel.

Internally the grabber functions use a double-buffered output in the appropriate format and size, which can be transferred to the users environment. A set of flags prevent updating while a transfer is in progress.

See the example in the code fragments in the appendix.

4.2.1 Public User Functions in AudioGrabberComplex256

bool newDataAvailable(void) - Returns `true` if a new data block is available since the last "grab", and `false` the the available data is unchanged.

void grab(int16_t * buffer) - where `buffer` is a length 512 int16_t buffer, within the scope of the call, that will receive the new data record. Data is automatically transferred to the buffer.

Appendix A: Code Fragments from my AudioSDR System

1. `audioSDR` **definitions and connections:**

```
// Required "includes"
#include <Arduino.h>
#include <Audio.h>
#include "arm_math.h"
#include "arm_const_structs.h"
#include "audioSDRlib.h"
// etc....
// --- Audio Block elements
AudioInputI2S          IQinput;
AudioSDRpreProcessor   preProcessor;   // For I2S error detection and compensation
AudioGrabberComplex256 spectrumData;   // For extraction of data for spectrum display
AudioSDR               SDR;
AudioOutputI2S         audioOut;
AudioControlSGTL5000   codec;
//---
// Audio Block connections
AudioConnection  c1(IQinput,0,       preProcessor,0);
AudioConnection  c2(IQinput,1,       preProcessor,1);
AudioConnection  c3(preProcessor,0,  spectrumData,0);
AudioConnection  c4(preProcessor,1,  spectrumData,1);
AudioConnection  c5(preProcessor,0,  SDR,0);
AudioConnection  c6(preProcessor,1,  SDR,1);
AudioConnection  c7(SDR,0,           audioOut,0);
AudioConnection  c8(SDR,0,           audioOut,1);
//
```

2. SDR and codec set-up in setup(): (`frequency` and `tuningOffset` are defined globally)

```
void setup() {}
.
.
codec.inputSelect(AUDIO_INPUT_LINEIN);
codec.volume(0.7);
codec.lineInLevel(15);      // Set codec input voltage level to most sensitive
codec.lineOutLevel(13);     // Set codec output voltage level to most sensitive
codec.enable();
//
tuningOffset = SDR.setDemodMode(LSB); // Select LSB mode and return its tuning offset
frequency = 7150000.0;                      // Start listening at 7.15 MHz
displayFrequency(frequency);                // Update the front panel display
tuner.setFrequency(frequency-tuningOffset); // Set rf quadrature oscillator freq.
//
SDR.inputGain(2.0);
SDR.outputGain(5.0);
```

```
SDR.selectFilter(audio2900);  // Overide the default 2700 Hz LSB audio filter
SDR.mute(false);                    // Turn on the audio output
SDR.setNoiseBlankerThesholdDb(5.0); // Set threshold to 5dB above average level.
SDR.enableNoiseBlanker();
SDR.disableALSfilter();
SDR.setAGCmode(AGCmedium);
//
preProcessor.startAutoI2SerrorDetection();  // Start I2S error detection
preProcessor.swapIQ(false);
.
.
}
```

3 - Examples of polling front panel controls in loop(): .

```
.
//--- Check if the toggle MUTE tactile switch has been pressed
if (muteSW.update() && muteSW.fell()){
  MUTE_on = !MUTE_on;
  SDR.setMute(MUTE_on);
}
//--- Check if the i2Serror correction change tactile switch has been pressed
if (I2SerrorSW.update() && I2SerrorSW.fell()){
  // Manually increment and set the I2S error correction mode
  lag = preProcessor.getI2SerrorCompensation();
  if (++lag > 1) lag = -1;       // cycle modes  -1 -> 0 -> 1 -> -1...
  preProcessor.setI2SerrorCompensation(lag);
}
//--- Cycle around the AGC modes with presses of the AGC tactile switch  (AGCmode is decl
if (agcEnableSW.update() && agcEnableSW.fell()){
  AGCmode = (++AGCmode)%4;
  SDR.setAGCmode(AGCmode);
}
.
.

}  // end of loop
```

4 - Extraction of a data buffer to update spectrum/signal strength in loop(): .

```
     // --  SpectrumTimer is a "Metro" timer set to 50 msec.
     // --  SpectrumData is a a data grabber block connected to the pre-processor
    if (SpectrumTimer.check()) {
      //-- Fetch and process FFT for signal strength and spectral display
      if (spectrumData.newDataAvailable()){
        // -- Note: int16_t FFTbuffer[512] is defined globally
        spectrumData.grab(FFTbuffer);
      }
     ...   (process FFTbuffer and display signal strength and panadapter) .
    }
```

Appendix B: BareBonesWSPR

A minimal, but fully functional WSPR (Weak Signal Propagation Reporter) AudioSDR Receiver.

WSPR (pronounced "whisper") is a world-wide network of very low power (from about 10mW to 5w) amateur radio stations that transmit 2 minute duration encoded ID messages, at about 20 minute intervals, using very narrow band (about 6 Hz) FSK as an upper-sideband signal. The transmissions are synchronized to start on the even minutes UTC. While not transmitting, the stations act as receivers, and decode incoming WSPR signals which are then automatically uploaded to a world-wide database at WSPRnet.org. A very good introduction to WSPR can be fond at http://www.g4ilo.com/wspr.html

You must have an amateur radio license to operate a transmitting station (I do not), but anybody can own a receiver and upload reception reports to the database – but if you do you will have to give yourself a pseudonym for identification.

I have included a `BareBonesWSPR.ino` sketch in the AudioSDR examples folder. It is an attempt to create a minimal, stripped-down, but fully functional SDR example that has been working unattended for several weeks. Figure 3 shows the stations received on a single overnight reception period using `BareBonesWSPR`.

`BareBonesWSPR` deliberately has the following **non**-features:

- It has no user controls.
- It has no panel display.
- All parameters mut be set in the sketch before compiling.

The idea is that it is a framework upon which to build a more sophisticated system.

The output from BareBonesWSPR is the "audio" output of the demodulated USB WSPR signal, which contains multiple WSPR signals in a narrow band (200 Hz) centered at 1500 Hz. These will generally be inaudible. You will need two additional software components and a host computer to use BareBonesWSPR:

A WSPR decoder: WSPR signals are exteremely weak (as much as -30dB below the receiver noise level), Subsequent signal detection and decoding requires sophisticated DSP and must be done on a PC system. I use **WSJT-X** software as the decoder - it is a very powerful system and has a very nice narrow-band panadapter/waterfall display. It also handles the uploading to the WSPRnet database automatically.

A database interrogation and display program: to download and display maps and database search information. The map in Fig. 3 was generated by **http://wspr.vk7jj.com/**. Alternatively you can use **http://wsprnet.org/drupal** and click on "maps" or "database".

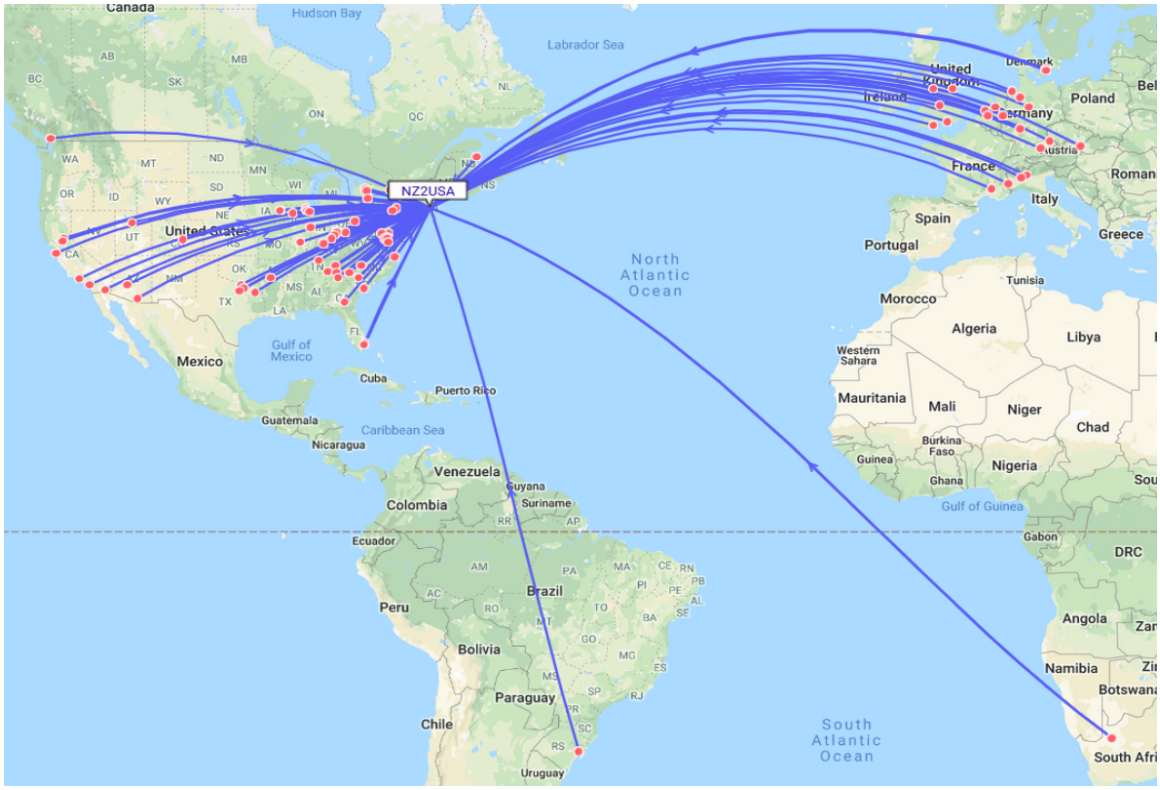

Figure 3: 40m (7,038,600 Hz) WSPR locations received in a 12 hour recording session using the `BareBonesWSPR.ino` with an indoor magnetic-loop antenna.