Jacob Sundstrom
DXARTS 565 Final Project Documentation

The Brain and Sound:
an investigation into the application of EEG data in electronic music and beyond

Abstract:

This project is serves as a preliminary investigation into the potential and applicability of EEG data in electronic music. The goal of the project is to acquire EEG data through the hacking of NeuroSky's TGAM1 EEG chip, found in a Mattel Mindflex headset, analyze the data in ATS, then apply it to a simple algorithm in SuperCollider, thus demonstrating the wide potential for application not only in a basic parameter mapping context, but also as data to use in traditional contemporary musical composition. The project also attempts to bring to light several latent issues in the application of EEG data to art in general, as well as issues specific to a sonic medium.

Stage One: acquiring the raw wave

The first stage of research was to understand how the TGAM1 outputs data and how to acquire EEG data in its raw wave form. As it stands, there are three output modes on the NeuroSky TGAM1: first, an output of Poor Quality, eSense attention, and eSense meditation; second, an output of the same information plug EEG powers (delta, theta, low alpha, high alpha, low beta, high beta, low gamma, mid gamma); third, the same as the second output plus a 10-bit raw EEG wave, sampled at 512Hz. The EEG power outputs are unit-less; that is to say, they are only useful when compared to one another over time. The mode of interest here is the third; that is, the raw wave data.

According to the TGAM1 documentation, there are two ways to get the correct output: first, a hardware hack, disconnecting and re-soldering a jumper on the board itself. This has been achieved by a few others in years past. Second, a serial command sent to the chip in order for it to output in the correct mode; this is the method used in this project. Despite extensive research, I found nothing indicating that anyone had used this method before, most likely due to the fact that the Arduino Brain library requires the use of the serial port and the design of the Arduino prohibits the use of two way communication on the serial port; that is, anything sent back from the serial port on the Arduino is also written to the serial port. Due to this, the author had to use the Arduino SoftwareSerial library and this rewrite most of the Arduino Brain library, though many of the methods reimplemented were used almost verbatim from the original library (as an aside, it is the desire of the author to update the library over the summer for use with the SoftwareSerial library). After much lucubration, the raw wave data (fig. 1) was being streamed to the Arduino serial monitor.

At this point, a strange issue emerged: the author measured his brain for a period of time, 30 seconds, then, using a SuperCollider program, wrote the data to a signal. However, when the signal was written, it was found that it only lasted less than 10 seconds at 512Hz; the number of data items was then counted and compared to what would be expected for 30 seconds and it was found to be about one fourth of the quantity expected. After additional research, it was found that the Arduino is indeed too slow using the traditional digital.read/write() commands used by the SoftwareSerial library. Hence, the author acquired data at about 128Hz, potentially aliasing and corrupting the data (EEG gamma waves run from about 30Hz to 100Hz, though the are generally very low in amplitude when compared to other, more robust waves found in the brain). There is a way around this: true 'c' style commands reading directly from the Arduino's Atmega328 chip; however, given the time constraints of the project, it was decided that it should move ahead with the 128Hz sampling rate instead of using valuable time rewriting the complex SoftwareSerial library.

Stage Two: data analysis and implementation

After the raw data was acquired, a SuperCollider program, written by the author, was used to transform the textfile data into a .wav file, capable of being analyzed by Juan Pampin's ATS software. Seeing that the numbers do not reach into negative numbers, the average of the file was computed and taken to be the zero-crossings; it was then subtracted from every value, then the entire file was normalized to appear in the standard range of -1 to +1. A sample waveform appears in Fig. 2.

The file was then analyzed in ATS to then be ported into SuperCollider for use as control data. It was found early on, however, that ATS is likely not the best tool to use for this purpose. ATS was designed for higher frequencies than EEG waves, as well as higher sampling rates. Additionally, ATS is designed for the analysis and resynthesis of sound into another sound; the author desired to use the data as a control. This proved to be difficult to use in making meaningful control structures in SuperCollider, since the data acquired had to be gathered in SynthDefs on the server and then sent back to the client in the language. For the purposes of this project, the author decided to make simple sonification of the data directly in synths reading from busses to which the data was written (only delta, theta, alpha, and beta waves were used). Moreover, ATS requires the file to be analyzed offline, thus prohibiting real-time use, which is, at present, the ultimate goal of the author.

For the SuperCollider implementation, a convolution synthesizer was implemented. It utilized an FIR filter; that is, kernel convoluted with a decorrelated signal generated by the Dust UGen. The signal was then branched into two paths: one, in which the convoluted signal was able to pass from the Convolution2 UGen directly to the output; and the second, in which the convoluted signal was routed through a series of resonant filters tuned to an arbitrary frequency. The mix of the two signals was modulated by a 'mix' value, such that the signal could become more or less noisy depending on the value (the unadulterated convolution signal was generally more noisy than its filtered counterpart).

The data was gathered from a recording session where the author listened to a sample or piece of music for a given set of time; the kernels for the FIR filter were then extracted from the same sample listened to at the time of recording. In an effort to get a clear signal, the author mentally prepared himself before each session, sitting quietly and ensuring no disturbance, lightly meditating beforehand.

Mapping the data onto sonic parameters proved to be quite difficult, not only in terms of what waves to map to what parameter, but also in terms of implementation in SuperCollider itself, mostly due to difficulties mentioned above. In the end, it was decided to map the parameters as follows: the inverse of the delta value determined the amplitude, such that the lower the delta value, the higher the amplitude; the theta value was mapped to distortion gain, such that a higher theta value resulted in more distortion; the alpha value was mapped to the mix parameter, such that the higher the alpha value, the less noisy and more pitched the signal became; and the beta value was mapped to stretch the resonant filter frequencies, such that as the beta value rose, the distance between the frequencies of the resonant filters increased. The parameters were chosen after extensive investigation into the nature of the brain waves and their relationships to one another in the author-acquired signal, such that as one meditates and focuses, the resulting signal becomes louder (delta falls), more distorted (theta rises), more concentrated (beta falls), and noisiness decreases (alpha rises). The association of the waves with certain mental activity can be found in Larsen, pages 10 and 11. At present, the author is unable to maintain a steady state due to 1) inexperience in meditation, and 2) unconscious and involuntary changes in brain activity resulting from the auditory stimulus during the recording sessions. However, due to the offline nature of the analysis, the involuntary changes were desired, though in a real-time situation, the involuntary changes would be the result of direct feedback from the resulting audio. Interestingly, during the playback of the resulting sonification it was discovered that there exist larger pattens of periodicity within wave bands themselves; whether or not this is due directly to the auditory stimulus present during recording or a function of the brain is yet to be discovered by the author and will require additional experimentation in the future.

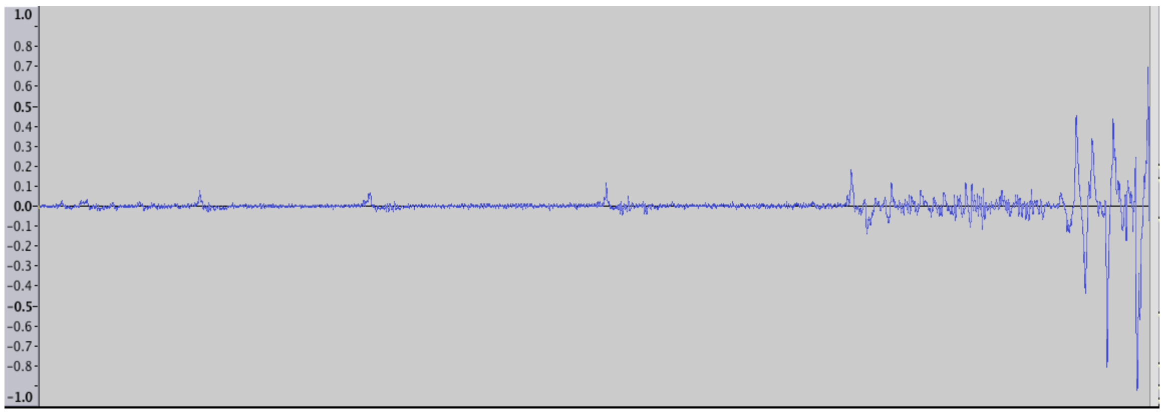
Epilogue: going further

The results of the project are an exciting look into the potential of EEG data in the artistic sphere, not only in terms of sonification and biofeedback, but also in generating signals for use in the authors own experimental noise performances, as well as generating data and patterns for use in musical composition for acoustic instruments. Moreover, sonification of EEG data could prove useful to scientists searching for a more comprehensive way to analyze data, especially larger patterns, in long and complex signals. There are certainly key issues still to be resolved, but, at present, the potential for EEG data in the artistic sphere is a vast and relatively untapped reservoir.

Fig. 1:

467
471
473
472
473
471
472
472
470
471
469
472
472
473
471
474
473
474
473
471
472
475
etc...

Fig. 2



Jacob Sundstrom
DXARTS 565 Final Project Documentation

Bibliography

Larsen, Erik A. *Classification of EEG Signals in a Brain- Computer Interface System*. Diss. Norwegian U of Science and Technology, 2011. N.p.: n.p., n.d.

“Brain Wave Signal (EEG) of NeuroSky, Inc.” Manual. NeuroSky, Inc. 15 December 2009.

“TGAM1 Spec Sheet.” Manual. NeuroSky, Inc. 24 March 2010.

“Mindset Communications Protocol.” Manual. NeuroSky, Inc. 28 June 2010.

Pampin, Juan. "ATS Manual." *ATS*. Stanford, n.d. Web. 10 May 2014.

Whippler, Jean-Claude. "Computing Stuff Tied to the Physical World." *JeeLabs RSS*. N.p., n.d. Web. 13 Apr. 2014.

"Frontier Nerds: An ITP Blog." *How to Hack Toy EEGs*. N.p., n.d. Web. 25 April 2014.

```
//////////////////
// Arduino code
//////////////////

#include <Brain.h>
#include <SoftwareSerial.h>
#define BAUDRATE 9600

Brain headset(Serial);
SoftwareSerial tgam(10,11); // RX, TX on other device
int latestByte, lastByte;
signed int checkSumAccum;
int packetIndex = 0, packetLength;
boolean isNewPacket = false;
int tgamPacket[32];

int signalQuality, meditation, attention, eegPowerLength;
int eegPower[8];
int rawLength, rawLo, rawHi, raw;

int led = 13;

void setup() {
  // set stuff up
  Serial.begin(9600);
  tgam.begin(9600);
  pinMode(led, OUTPUT);

  // send command to output raw wave value, then change baud
  delay(1000);
  tgam.write(2); // actual command
  Serial.println("stuff written");
  blinkLED();

  // close serial
  delay(100);
  tgam.end();
  Serial.end();

  // reopen serial @ 57.6k baud
  delay(100);
  tgam.begin(57600);
  Serial.begin(57600);
}


void loop() {
  // do something
  newPacket();
}


// debug
void printMyPacket() {
  while(tgam.available() > 0) {
    latestByte = tgam.read();
    Serial.println(latestByte);
  }
}

void newPacket() {
  while((tgam.available()) > 0) {
    latestByte = tgam.read();

    if(isNewPacket) {

      if(packetIndex==0) {
        packetLength = latestByte;
      }
      else if(packetIndex <= packetLength) {
        tgamPacket[packetIndex - 1] = latestByte;

        checkSumAccum += latestByte;
      }
      else if(packetIndex > packetLength) {
        // we're at the end
        int checkSumVal = latestByte;
        checkSumAccum = checkSum(checkSumAccum);

        if(checkSumVal == checkSumAccum) {
          // parse it out
          parseMyPacket();
        }
        else {
          // Checksum mismatch, send an error.
          Serial.println("ERROR: Checksum");
        }

        isNewPacket = false;
      }
      packetIndex++;
    }

    if((latestByte==170) && (lastByte==170)) {  // sync bytes in place (0xAA)
      isNewPacket = true;
      packetIndex = 0;
      checkSumAccum = 0;
    }

    lastByte = latestByte;
  }
}

int checkSum(int sum) {
  // mask the upper bits (>8), then take one's complement
  sum &= 0xFF;
  sum = 255 - sum; // ~sum???

  return sum;
}

int getRaw(int localRawLo, int localRawHi) {
  int tempVal;

  tempVal = (localRawLo<<8) | localRawHi;
  return tempVal;
}

// some stuff here ..oOo..oOo..oOo..oOo..oOo..oOo..oOo..oOo..oOo..oOo..oOo..oOo
void parseMyPacket() {
  for (byte i = 0; i < (packetLength-1); i++) {
    switch (tgamPacket[i]) {
    case 2:
      signalQuality = tgamPacket[++i];
      break;
    case 4:
      attention = tgamPacket[++i];
      break;
    case 5:
      meditation = tgamPacket[++i];
      break;
    case 131:
      // ASIC_EEG_POWER: eight big-endian 3-byte unsigned integer values representing delta, theta, low-alpha
high-alpha,
      // low-beta, high-beta, low-gamma, and mid-gamma EEG band power values.
      // The next byte sets the length, usually 24 (Eight 24-bit numbers... big endian?)
      eegPowerLength = tgamPacket[++i];

      // Extract the values. Possible memory savings here by creating three temp longs?
      for(int j = 0; j < 8; j++) {
        eegPower[j] = ((unsigned long)tgamPacket[++i] << 16) | ((unsigned long)tgamPacket[++i] << 8) |
(unsigned long)tgamPacket[++i];
      }
      break;
    case 128:
      rawLength = tgamPacket[++i];
      rawLo = tgamPacket[++i];
      rawHi = tgamPacket[++i];
      raw = getRaw(rawLo, rawHi);
      Serial.println(raw);
    }
  }
}

void blinkLED() {
  for(byte i = 0; i < 10; i++) {
    digitalWrite(led, HIGH);
    delay(75);
    digitalWrite(led, LOW);
    delay(75);
  }
}
```

```
//////////////////////////////
// SuperCollider Code /////
//////////////////////////////

Server.local.makeWindow;
Server.local.options.memSize = 16384;

// do this first
(
Routine.run({
	Server.local.boot;
	0.1.wait;
	Server.local.scope;
});
)

// main
(
var infoArray = [
	[thisProcess.nowExecutingPath.dirname +/+ "sounds/cunningham2.wav", 120.144, 4096],
	[thisProcess.nowExecutingPath.dirname +/+ "sounds/mable.wav", 2.1, 4096],
	[thisProcess.nowExecutingPath.dirname +/+ "sounds/tcp_d2_09_phonetic_alphabet_nato_irdial.wav", 8.0,
4096],
	[thisProcess.nowExecutingPath.dirname +/+ "sounds/piano.wav", 27.8, 4096],  // 27.8
	[thisProcess.nowExecutingPath.dirname +/+ "sounds/piano.wav", 130.2, 4096],
	[thisProcess.nowExecutingPath.dirname +/+ "sounds/earth.wav", 7.0, 8192],
	[thisProcess.nowExecutingPath.dirname +/+ "sounds/piano.wav", 130.2, 8192]
];
var pollVals, testSynth, convSynthKlank, convSynth, conv, conv2, conv3;
var makeNotes, makeKernalArray, loadBuffs, freeBuffs, runMain;
var buffArray, kernalArray;
var deltaBus, thetaBus, alphaBus, betaBus;
var pollRoutine;
var debug = 0, plotKernals = false;

// ATS file
~atsFile = AtsFile.new(thisProcess.nowExecutingPath.dirname +/+ "lib/ats/two.ats").load;

// data busses
deltaBus = Bus.control(s, 1);
thetaBus = Bus.control(s, 1);
alphaBus = Bus.control(s, 1);
betaBus = Bus.control(s, 1);


// synthDefs...
pollVals = SynthDef(\pollVals, {arg atsBuff, totalDur = 120.0, timeThresh = 1.0, ampThresh = 0.3, inBus = 2;
	var delta, theta, alpha, beta, gamma, waves, waveSum;
	var deltaPartials, thetaPartials, alphaPartials, betaPartials;
	var in, inAmp, inTrig, timer;
	var pointer = LFSaw.kr(~atsFile.sndDur.reciprocal, 1, 0.5, 0.5);

	deltaPartials = Array.series(38,1,1);
	thetaPartials = Array.series(15,39,1);
	alphaPartials = Array.series(20,55,1);
	betaPartials = Array.series(41,76,1);

	delta = (AtsAmp.kr(~atsFile.bufnum, deltaPartials, pointer)).sum;
	theta = (AtsAmp.kr(~atsFile.bufnum, thetaPartials, pointer)).sum;
	alpha = (AtsAmp.kr(~atsFile.bufnum, alphaPartials, pointer)).sum;
	beta = (AtsAmp.kr(~atsFile.bufnum, betaPartials, pointer)).sum;

	// normalizeSum the values
	waveSum = [delta, theta, alpha, beta].sum;
	waves = [delta, theta, alpha, beta]/waveSum;

	// write the values to a bus
	Out.kr(deltaBus, waves[0]);
	Out.kr(thetaBus, waves[1]);
	Out.kr(alphaBus, waves[2]);
	Out.kr(betaBus, waves[3]);
}).send(s);

convSynth = CtkSynthDef(\convSynth, {arg dur = 10.0, amp, buffer, lpCutoff = 2000, den = 2000, panFreq = 1.0,
centerFreq = 50, outBus = 2, mix = 1.0, gain = 1.0, mS = 1, alpha = 0.5, beta = 0.5, delta = 0.5, theta = 0.5;
	var ampEnvGen;
	var sigL, sigR, convSig, klankSig, out;
	var convSigMono, klankSigMono;
	var klankSigExciter, convSigExciter;
	var panPos;
	var harm, amps, ring;
	var deltaAmpGood, deltaAmp;
	var mixGood, centerFreqGood, gainGood, stretchGood, stretch;
	var ampStretchGood, ampStretch;

	// in
	mixGood = (alpha/1)* 3;
	mix = Gate.kr(mixGood, CheckBadValues.kr(mixGood)<=0);
	// mix = Clip.kr(mix, 0, 1);
	// mix = mix*Rand(0.5, 1.5);

	deltaAmpGood = Lag.kr(delta.reciprocal, 2.0);
	deltaAmp = Gate.kr(deltaAmpGood, CheckBadValues.kr(deltaAmpGood)<=0);

	gainGood = theta * 4.5;
	gain = Gate.kr(gainGood, CheckBadValues.kr(gainGood)<=0);

	stretchGood = beta + 0.8;
	stretch = Gate.kr(stretchGood, CheckBadValues.kr(stretchGood)<=0);


	ampStretchGood = Lag.kr(alpha/beta, 5.0);
	ampStretch = Gate.kr(ampStretchGood, CheckBadValues.kr(ampStretchGood)<=0);
	ampStretch = ampStretch*Rand(1.0, 2.0);

	// sounds lame... could do better
	// centerFreqGood = Lag.kr((theta/delta)*2*centerFreq, 1.0);
	// centerFreq = Gate.kr(centerFreqGood, CheckBadValues.kr(centerFreqGood)<=0);

	// note envelope
	ampEnvGen = EnvGen.kr(
		Env([0,1,1,0], [1,15,1].normalizeSum),
		levelScale: amp*0.2,
		timeScale: dur
	);

	// complex source
	sigL = LPF.ar(Dust2.ar(den), lpCutoff);
	sigR = LPF.ar(Dust2.ar(den), lpCutoff);

	// filter!
	convSig = Convolution2.ar([sigL, sigR], buffer, framesize: 4096);

	// klank stuff
	harm = [0.51, 1, 2.01, 2.98, 4.02, 5.01, 6.02, 6.99, 8.0, 9.03, 9.97] * stretch;
	amps = [0.7, 1.0, 1.0, 0.9, 0.8, 0.71, 0.82, 0.9, 0.7, 0.6, 0.6, 0.55] * (0.015*ampStretch);
	ring = Array.rand(11, 0.2, 1.5);

	// panning..
	panPos = LFNoise1.kr(panFreq);

	// klank me
	klankSig = DynKlank.ar(`[harm, amps, ring], HPF.ar(HPF.ar(convSig, 50), 50), centerFreq);
	klankSigExciter = HPF.ar(
		(HPF.ar(klankSig, (centerFreq*64).min(SampleRate.ir*0.5)) * 3.0).softclip,
		(centerFreq*128).min(SampleRate.ir*0.5),
		2.0);
	klankSig = Mix.new([klankSig, klankSigExciter]) * mix;
	klankSig = Rotate2.ar(klankSig[0], klankSig[1], panPos);
	klankSigMono = Pan2.ar(klankSig[0], panPos);
	klankSig = Select.ar(mS, [klankSigMono, klankSig]);

	// conv me
	convSigExciter = HPF.ar(
		(HPF.ar(convSig, (centerFreq*64).min(SampleRate.ir*0.5)) * 3.0).softclip,
		(centerFreq*128).min(SampleRate.ir*0.5),
		2.0
	);
	convSig = Mix.new([convSig, convSigExciter]) * (1.0-mix);
	convSig = Rotate2.ar(convSig[0], convSig[1], panPos);
	convSigMono = Pan2.ar(convSig[0], panPos);
	convSig = Select.ar(mS, [convSigMono, convSig]);

	// make me stereo
	out = Mix.new([convSig*1.5, klankSig]);
	out = (out*gain).softclip;
	out = Mix.new([FreeVerb2.ar(out[0],out[1], 1.0, 1.0, 0.5), out]);
	Out.ar(outBus, (out*ampEnvGen*deltaAmp).softclip);
}).send(s);


// kernal stuff.......................
// make kernal function
makeKernalArray = {arg localBuffArray, localSize;
```

```
	var win, makeKernalFunc;
	var localKernal;
	var localArray;
	var normalize;

	// this function must live inside this in order for it to work?!
	normalize = { arg kernel, maxAmp = 1.0;
		var realSignal, imagSignal, cosTable, fftSignal, fftMagnitude, fftPhase, maxMag;
		var scale;
		var ifftKernel;

		// FFT analysis here!
		realSignal = kernel.as(Signal);
		imagSignal = Signal.newClear(kernel.size);
		cosTable = Signal.fftCosTable(kernel.size);
		fftSignal = fft(realSignal, imagSignal, cosTable);
		fftMagnitude = fftSignal.magnitude;

		// reset phase to linear
		fftPhase = Array.series(kernel.size, 0, -pi);

		// compute max magnitude and display
		maxMag = fftMagnitude.maxItem;
		"un-normalised maximum magnitude = ".post;
		maxMag.ampdb.post;
		" dB".postln;

		// normalise phase
		ifftKernel = ifft(
			fftMagnitude.as(Signal) * (fftPhase.cos.as(Signal)),
			fftMagnitude.as(Signal) * (fftPhase.sin.as(Signal)),
			cosTable
		);
		ifftKernel = ifftKernel.real;

		// normalise gain
		scale = maxMag.reciprocal * maxAmp;
		(scale * ifftKernel).as(Array);
	};

	// make the kernal
	win = Signal.hanningWindow(localSize);
	win = win.as(Array);
	localBuffArray.loadToFloatArray(action: {arg array;
		~localKernal = win*array;
		~localKernal = normalize.value(~localKernal);

		// debug me
		if(debug == 2, {'inFunction: '.post; ~localKernal.postln;}, {});
	});

	// return
	~localKernal;
};

// buffers ..oOo..oOo..oOo..oOo..oOo..oOo..oOo
buffArray = Array.fill(infoArray.size, {arg i;
	CtkBuffer.playbuf(
		infoArray[i][0],			// path
		(infoArray[i][1] * 44100).floor,	// startFrame
		infoArray[i][2],			// size
		channels: 0				// just load one channel (left)
	);
});

loadBuffs = {arg localBuffArray;
	localBuffArray.do({arg localBuff;  // load kernal buffers
		localBuff.load;
	});
};

freeBuffs = {arg localBuffArray;
	localBuffArray.do({arg localBuff;
		localBuff.free;	// free only the kernal buffers
	})
};


///////
runMain = {
	Routine.run({
		var ratio;
```

```
				0.5.wait;
				Synth(\pollVals, addAction: \addToTail);

				conv = Synth(\convSynth, [
					\dur, ~atsFile.sndDur,
					\amp, 0.8,
					\buffer, [b,c,d].choose.bufnum,
					\den, 15000,
					\panFreq, 0.05,
					\centerFreq, 30,
					\gain, 1.0,
					\mS, 0,
					\lpCutoff, 15000,
					\outBus, 0,
					\mix, 0.0
					]
				);
				conv.map(\alpha, alphaBus);
				conv.map(\beta, betaBus);
				conv.map(\delta, deltaBus);
				conv.map(\theta, thetaBus);

				conv2 = Synth(\convSynth, [
					\dur, ~atsFile.sndDur,
					\amp, 0.7,
					\buffer, [b,c,d].choose.bufnum,
					\den, 15000,
					\panFreq, 0.33,
					\centerFreq, 46,
					\gain, 1.0,
					\mS, 1,
					\lpCutoff, 15000,
					\outBus, 0,
					\mix, 0.0
					]
				);
				conv2.map(\alpha, alphaBus);
				conv2.map(\beta, betaBus);
				conv2.map(\delta, deltaBus);
				conv2.map(\theta, thetaBus);

				conv3 = Synth(\convSynth, [
					\dur, ~atsFile.sndDur,
					\amp, 0.7,
					\buffer, [b,c,d].choose.bufnum,
					\den, 15000,
					\panFreq, 0.33,
					\centerFreq, 33,
					\gain, 1.0,
					\mS, 1,
					\lpCutoff, 15000,
					\outBus, 0,
					\mix, 0.0
					]
				);
				conv3.map(\alpha, alphaBus);
				conv3.map(\beta, betaBus);
				conv3.map(\delta, deltaBus);
				conv3.map(\theta, thetaBus);
			});
		};

		// do things in the right order
		Routine.run({
			( // allocate three buffers
				b = Buffer.alloc(s,4096);
				c = Buffer.alloc(s,4096);
				d = Buffer.alloc(s,4096);
				b.zero;
				c.zero;
				d.zero;
			);
			0.5.wait;
			// do stuff
			loadBuffs.value(buffArray); // load buffers

			// make the kernals
			0.5.wait;
			kernalArray = Array.fill(buffArray.size, {arg i;
				0.2.wait;  // absolutley necessary!!
				makeKernalArray.value(buffArray[i], infoArray[i][2]);
			});
```

```
	// unforunate hack...
	kernalArray[5].do({arg val, i;
		0.0001.wait;
		b.set(i, val);
	});
	kernalArray[4].do({arg val, i;
		0.0001.wait;
		d.set(i, val);
	});
	kernalArray[2].do({arg val, i;
		0.0001.wait;
		c.set(i, val);
	});
	\done.postln;

	// wait a sec to free, then free, otherwise things go wrong
	1.0.wait;
	if(plotKernals, {
		kernalArray.size.do({arg i;
			{kernalArray[i].plot}.defer;
			'inArray: '.post; kernalArray[i].postln;
		});
		}, {}
	);
	freeBuffs.value(buffArray); "buffers freed!!".postln;

	// wait again, then do everything
	0.1.wait;
	runMain.value;
});

)

this.clearAll;
```