SBATMS
(Small Boxes Arranged To Make Sound)
Danny Clarke

1 Introduction

It would be almost disingenuous to refer to SBATMS as a NIME (New Interface for Musical Expression). The project is not particularly interested in facilitating expression. Indeed, on some level it is a reaction against the obsession with fine-grained control and gestural expressivity that permeates that genre. Instead, this project is an attempt to embed behavioral systems into physical objects and to provide an interface that facilitates exploration of the emergent properties of those systems through the physical rearrangement of its parts. So, SBATMS may not be a NIME in the traditional sense of the term, but it does sit nicely in an adjacent family of NIMEs: New Interfaces for Musical Exploration.

2 Influences

In his lecture *The Influence of Peasant Music on Modern Music*[1], the composer Belá Bartók proposes three ways that a composer can incorporate folk music into her work. Bartók lays the methods out in an order of increasing synthesis. He describes the final method as the composer having, "completely absorbed the idiom of peasant music which has become his musical mother tongue." Since I have begun writing electronic, specifically computer, music I have been trying to figure out what the analog of "mother tongue" might be for the genre. What is the culture of craft for computers? As tools, what do they facilitate that other tools do not? Why should I use a computer to make music instead of finding humans (who would be both easier and more fun to work with!)?

SBATMS is one attempt to create an answer for myself and it brings together four recent influences on my work: texts from the early years of artificial intelligence research, ecological psychology, the music of Henry Brant and the music of John Luther Adams.

2.1 Artificial Intelligence

Modern approaches to artificial intelligence emphasize statistical learning and linear algebra. Such methods are effective and easy to optimize. However, in the early days of the field, researchers preferred to build solutions out of logical and linguistic systems; the leap from McCarthy's *Programs with Common Sense*[2] to Chomsky's *Three Models*[3] is small indeed. Unfortunately, these systems were both too difficult to optimize and too difficult to generalize. Despite their shortcomings, the influence of these approaches to artificial intelligence has permeated the way that we structure computer systems today (not least because McCarthy's work directly resulted in the generation of one of the programming world's most mystical languages). The thought that complexity can arise through the recursive application of provable rules that are text-encodable and that build in layers of increasing abstraction and power, is arguably the fundamental belief of programming as a craft.[4]

2.2 Ecological Psychology

Ecological psychology is a branch of psychology that attempts to work at the border of the organism and its environment. Oftentimes that border is thoroughly blurred, if not done away with altogether. Rather than treat the organism as a Skinnerian black box and the environment as a

[1] Luckily a transcript is available here: `http://www.richardtrythall.com/Resources/22a.BartokPeasantMusic.pdf`

[2] McCarthy, John. *Programs with Common Sense* `http://www-formal.stanford.edu/jmc/mcc59/mcc59.html`

[3] Chomsky, Noam. *Three Models for the Description of Language* `https://www.princeton.edu/~wbialek/rome/refs/chomsky_3models.pdf`

[4] Abelson, Harold and Sussman, Gerald 1996. `https://mitpress.mit.edu/sicp/full-text/book/book.html`

source of inputs, ecological psychology attempts to view the two as components in a larger, complex system. For some researchers in the field, particularly Michael Turvey, this leads to conclusions that radically reinterpret what consciousness, knowledge, and meaning are and where they "reside." Turvey does away with conventional ideas of mind altogether and formulates cognitive processes that are traditionally labeled "mental" as products of the meeting of the physical configuration of an organism and the physical configuration of its environment; that is, as an emergent property of the interaction between the complex systems that are organisms and the complex systems that are environments.[5] This is to say that "intelligence" is both not unique to humans and not localized to any individual thing or being.

2.3 Music as place and geography

Henry Brant and John Luther Adams are two composers who have made their careers by composing large-scale, outdoor works that hover somewhere between installation and composition. A given piece might encompass miles of land with dozens of performers strewn throughout. It changes the perception of a musical piece from that of an object that is containable and observable, to that of a landscape and environment that one inhabits. These composers, then have relinquished control over a listeners linear, temporal experience of their music and instead exchanged it for control over how a listener might experience the piece in space. In a sense, they've performed an FFT on the idea of a musical piece.

2.4 Closing

SBATMS is an attempt to bring the above influences together in a single, embedded score. Each box represents a simple behavior and together they generate complexity. However, a performer does not have direct control over the system that is generated. Indeed, instead of controlling a parameter like pitch or rhythm, the performer must try to shape a general process, the specifics of which, she has no control over. Finally, as the sounds of the boxes are highly localized (and rather soft), the performer, now as an audience member, has control over the scope and scale of the sounds she is hearing: while she can sit and listen at a high level to the linear unfolding of a process, she also has the ability to create her own narrative alongside that process through up-close exploration of each individual box.

3 Design

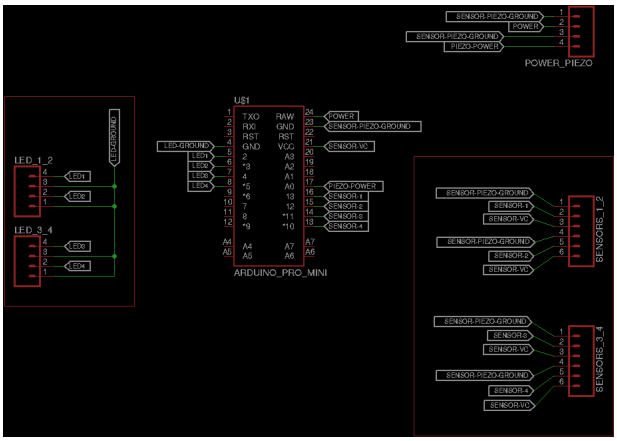
I designed SBATMS to be simplistic at every level. The materials for construction are simple and easy to find (in their first incarnation the boxes themselves are nothing more than decorative boxes found at a *Michael's* craft store); the circuits are simple; as a good old fashioned emergent system, the behaviors of the agents are simple.

SBATMS

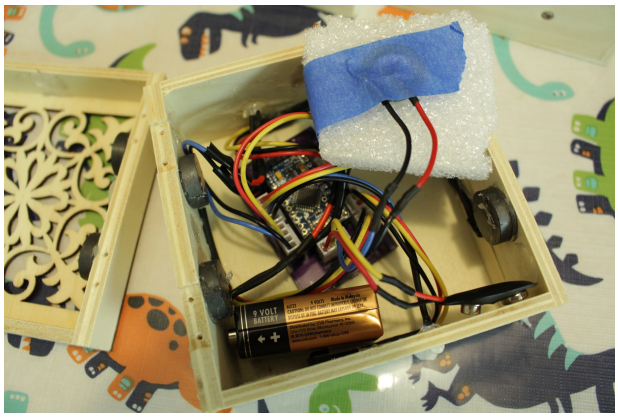
3.1 Circuit and Construction

The circuits for SBATMS are simple input-output devices: four infrared receivers, four infrared emitters and a single piezo.

[5] Turvey, Michael. *Nonrepresentational Perception and Action.* `https://www.youtube.com/watch?v=cWztQt_nlDU`



SBATMS board schematic



SBATMS innards

The emitters and receivers are set, in pairs, into the four sides of decorative boxes from *Michael's* and arranged in such a way that two-way communication can occur between two boxes no matter the specific sides that are placed next to each other.

SBATMS sensor/emitter pairs

The lid of each box is held down by ceramic magnets, which serve a secondary function as a loose coupling system between boxes. Hot glue abounds.

3.2 Interaction

There are two modes of interaction that I have discovered. The primary mode is moving boxes about in different configurations. The other is to move one's ears across the boxes to ear distinctly the different patterns each box is producing. These modes come together to produce an air of considered play: the performer/audience-member is able to let a system configuration remain static or to change for as long as she likes and she is able to experience these configurations at different levels of granularity. The restrictions intentionally encourage this kind of activity, which I find is distinctly composerly and heavily inspired by Andrew McLean's discussion of Paul Klee's *bricoleur*.[6]

4 Software

The software for SBATMS is written entirely in C++, using the Arduino headers. Each box is running the same simple program. That program is built around a "core message," a C array of unsigned 64-bit integers that is unique to each box. Each box uses this message in two capacities: it sonifies the contents of the message using bitwise operations and pulse-width modulation (PWM) and it broadcasts that message out to other boxes via its infrared LEDs. To facilitate these operations, the software has four main components: A reader, a parser, a sender, and a "bit player."

[6] McLean, Alex. *Bricolage Programming in the Creative Arts* `http://slab.org/writing/ppig.pdf`

4.1 Messages

Each box is initialized with two C arrays: one array that is hard-coded to have the values `[1,2,1]` and another containing a configurable number of values that are randomly initialized (randomness is seeded by reading analog pin 0 on the Arduino). The values in these arrays correspond to lengths of time, specifically the length of time that a box has its LEDs turned on. The messaging system is, in this way similar to Morse Code. The first array is used as "leader" for message broadcasting. If a box detects an incoming message it keeps track of the inputs it receives, but does not act on them until it receives a sequence of inputs that matches the values in the leader array:

```
// parser.cpp
case LEADER_LISTEN:
      // set and increment
      mLeader[mLeaderWritePos++] = word;

      if (mLeaderWritePos == mLeaderSize) {
        if (leaderMatch()) {
          mState = MESSAGE_LISTEN;
        }

        clearLeader();
        mLeaderWritePos = 0;
      }
```

```
// sketch.ino
if (parseOne.hasMessage()) {
    mutateCore(parseOne.getMessage());
    parseOne.listen();
}
```

Upon receipt of a message, the message is processed vís-a-vís the core message the box's software returns to watching for inputs. The processing of a message is carried out by the `mutateCore` method:

```
// sketch.ino
void mutateCore (const uint32_t * message) {
  uint16_t idx = random(0, msgLen);
  int32_t cW = coreMsg[idx], mW = message[idx];
  int32_t dif = cW - mW;
  if (dif > 0) {
     coreMsg[idx] = constrain(cW - 1, 1, 10);
  }
  else if (dif < 0){
     coreMsg[idx] = constrain(cW + 1, 1, 10);
  }
}
```

A random index is chosen and the values from the received message and the core message are compared. The value at the corresponding index in the core message is then either incremented or decremented in an attempt to move closer to the received message. This change is then reflected in the sound produced by the box and in the LED patterns broadcast by the box.

Through this process, boxes that are allowed to "communicate" for prolonged periods begin to converge in sound. It also means that a box that is communicating with multiple other boxes will tend to produce a noisier signal as it tries to match multiple different signals. Such patterns are the lynch pin of the piece: they provide consistent and repeatable patterns of overall behavior that are still specifically highly variable.

4.2 Sonification

Sonification of the core message is accomplished using a single equation of bitwise operations. The specific equation was arrived at through experimentation in a Supercollider program that can be found in Appendix F. The work relies heavily upon the work of Ville-Matias Heikkilä [7], known online as "viznut," a demoscene artist who was instrumental in the surge of interest in "byte beat" in the mid and late 2000s.

The operations are carried out by the `play` method of the `BitPulse` class, where `mWord` is a reference to the core message of the software

```
// bit_pulse.cpp
void BitPulse::play (uint32_t now) {
  uint32_t x,y;
  if (now >= mNextWrite) {
```

[7] This post and the 2 October post are crucial reading for anyone interested in byte beat: `http://countercomplex.blogspot.com/2011/10/some-deep-analysis-of-one-line-music.html`

```
    mVal = ((x=mTick>>mWords[0])
            | (y=mTick>>mWords[1]))
       ^ ((x^mWords[1])
          | (y^mWords[0]));
    analogWrite(mPin, (mTick * mVal));
    mNextWrite = now + SAMPLE;
    mTick+=mWords[0];
  }
}
```

5 Future

Experimentation with SBATMS so far has been more rewarding than I had anticipated. The effect of certain configurations is marked and, happily, reproducible. As a result it is easy to begin to build up a vocabulary, that is for different configurations to come to have meaning for a performer. The strength of this effect, points to a concert practice involving, ideally, hundreds of boxes with scores that are blueprints for rearrangement of the boxes in time by teams of people: collective action leading to the development of social meaning for the boxes.

6 Appendices

NB: Only implementation files are included

6.1 A. $\text{bit}_{\text{pulse.cpp}}$

```
#include "math.h"
#include "Arduino.h"
#include "bit_pulse.h"

#define SECOND 1000000.0 // # museconds in a second
#define SAMPLE 22.7      // # museconds in a sample @ 44.1kH

BitPulse::BitPulse (uint16_t pin, uint16_t mLen, const uint32_t * msg)
  : mNumWords(mLen)
  , mWords(msg)
  , mNextWrite(0)
  , mTick(0)
  , mPin(pin)
  , mVal(0)
  , mWriteVal(0)
{}

void
BitPulse::play (uint32_t now)
{
  uint32_t x,y;
  if (now >= mNextWrite)
  {
    mVal = ((x=mTick>>mWords[0]) | (y=mTick>>mWords[1])) ^ ((x^mWords[1]) | (y^mWords[0]));
    analogWrite(mPin, (mTick * mVal));
    mNextWrite = now + SAMPLE;
    mTick+=mWords[0];
  }
}
```

6.2 B. reader.cpp

```
#include "reader.h"
#include "Arduino.h"

Reader::Reader (uint32_t msgDelta, uint32_t timeOutInterval)
   : mMsgDelta(msgDelta)
   , mTimeOutInterval(timeOutInterval)
   , mCurrentVal(0)
   , mLastVal(0) {
   mState = UP_LISTENING;
}
```

```
/*
 * A change in the state of the pins can be detected while
 * the reader is one of two states, UP_LISTENING or DOWN_LISTENING.
 * If the pins change while UP_LISTENING, then we mark the start time
 * of an UP period and change state to DOWN_LISTENING.
 * If the pins while DOWN_LISTENING, then we mark the end time of an
 * UP period and, if the period was long enough, we change state to WORD_READ
 */
void Reader::read2 (uint32_t now, const uint8_t pinRegister, const uint8_t pins) {
   bool shift = false;

   switch (mState) {

      // Initial state: watching for a light to turn on.
      case UP_LISTENING:
         shift = detectShift2(pinRegister, pins);
         if (shift) {
            mStart = now;
            mState = DOWN_LISTENING;
            mTimeOut = now + mTimeOutInterval;
         }
         break;

         // Saw a light turn on, waiting for it to turn off.
      case DOWN_LISTENING:
         shift = detectShift2(pinRegister, pins);

         // record duration between light turning on and turning off
         // as a word.
         if (shift) {
            mEnd = now;
            mWord = (mEnd - mStart) / mMsgDelta;
            if (mWord > 0)
               mState = WORD_READ;
         }

         // shift back to UP_LISTENING on timeout
         else if (now > mTimeOut)
            mState = UP_LISTENING;
         break;

         // only move back to UP_LISTENING when the word is taken
      case WORD_TAKEN:
         mState = UP_LISTENING;
         break;

         // only other state is WORD_READ, do nothing in that state
```

```
      default:
         break;
   }
}

/*
 * Verify whether the reader has read a word
 * that has not been taken already
 */
bool Reader::hasWord () {
   switch(mState) {
      case WORD_READ:
         return true;
      default:
         return false;
   }
}

/*
 * Return the word and mark the reader as read
 */
uint32_t Reader::getWord () {
   mState = WORD_TAKEN;
   return mWord;
}

/*
 * Detects shifts on the pins in the pin register.e
 * "pinRegister" is the byte representing the current status of some
 * of the pins on an arduino (e.g. PIND or PINC).
 * "pins" is a filter to specify which pins on that register we want
 * to be checking.
 */
bool Reader::detectShift2 (const uint8_t pinRegister, const uint8_t pins) {
   // if the pins we care about are off,
   // then mCurrentVal will be 0
   // Otherwise mCurrentVal will be some number.
   mCurrentVal = pinRegister & pins;

   //   lo -> hi                  ||  hi -> lo
   if ((mCurrentVal && !mLastVal) || (!mCurrentVal && mLastVal)) {
      mLastVal = mCurrentVal;
      return true;
   }
   else {
      mLastVal = mCurrentVal;
      return false;
   }
```

```
}
```

6.3 C. parser.cpp

```
#include "parser.h"
#include "Arduino.h"

Parser::Parser (const uint32_t * leader,
                uint16_t leaderSize,
                uint16_t messageSize,
                uint32_t timeout)
  : mLeaderTemplate(leader)
  , mLeaderSize(leaderSize)
  , mMessageSize(messageSize)
  , mTimeout(timeout)
  , mState(LEADER_LISTEN) {
  // syntax with parens initializes to 0
  mLeader = new uint32_t[mLeaderSize]();
  mMessage = new uint32_t[mMessageSize]();
}

/*
 * Return true if the parser has completed parsing from a
 * Reader.
 */
bool Parser::hasMessage () {
  switch (mState) {
  case MESSAGE_READ:
    return true;
  default:
    return false;
  }
}

/*
 * Accepts "words" (durations a Reader spends in UP state)
 * and records them as part of the leader to a message, or as
 * the body to a message, depending on state.
 *
 * State will transition from LEADER_LISTEN to MESSAGE_LISTEN
 * when the Parser receives a number of words equal to the length
 * of a leader and matching the leader pattern that it is looking for.
 *
 * State will transition from MESSAGE_LISTEN to MESSAGE_READ when
 * the Parser receives a number of words equal to the length of a message.
 *
 * If the Parser is in a MESSAGE_READ state when this method is called,
 * the Parser will simply pass through.
```

```
 */
void Parser::parseMessage (uint32_t word) {
  // a kind of timeout: words > 10 are invalid
  if (word <= mTimeout) {
    switch (mState) {

      // record in the leader buffer
      // if we match on the leader template, start recording messages
    case LEADER_LISTEN:
      // set and increment
      mLeader[mLeaderWritePos++] = word;

      if (mLeaderWritePos == mLeaderSize) {
        if (leaderMatch()) {
          mState = MESSAGE_LISTEN;
        }

        clearLeader();
        mLeaderWritePos = 0;
      }
      break;

      // record into the message buffer,
      // when full, transition to MESSAGE_READ
    case MESSAGE_LISTEN:
      mMessage[mMessageWritePos++] = word;
      if (mMessageWritePos == mMessageSize) {
        mState = MESSAGE_READ;
        mMessageWritePos = 0;
      }
      break;

    default:
      break;
    }
  }
}

/*
 * Check if the Parser's current buffer of words matches the
 * leader pattern it is looking for.
 */
bool Parser::leaderMatch () {
  uint16_t score = mLeaderSize;

  for (uint16_t i = 0; i < mLeaderSize; i++) {
    if (mLeaderTemplate[i] == mLeader[i])
      score--;
```

```
  }

  // if score == 0, then we have a match (!0 == true), otherwise false
  return !score;
}

/*
 * Reset a the Parser's message buffer to 0s
 */
void Parser::clearMessage () {
  for (uint16_t i = 0; i < mMessageSize; i++)
    mMessage[i] = 0;
}

/*
 * Reset the Parser's leader buffer to 0s
 */
void Parser::clearLeader () {
  for (uint16_t i = 0; i < mLeaderSize; i++)
    mLeader[i] = 0;
}
```

6.4 D. sender.cpp

```
#include "sender.h"
#include "Arduino.h"

Sender::Sender (uint32_t delta
              , uint16_t leaderLen
              , const uint32_t * leader
              , uint16_t msgLen
              , const uint32_t * msg)
  : mMsgDelta(delta)
  , mLeader(leader)
  , mMsg(msg)
  , mLeaderLen(leaderLen)
  , mMsgLen(msgLen)
  , mNextWriteTime(0)
  , mMsgState(DOWN)
  , mSndState(LEADER)
  , mWritePointer(0)
{
}

void
Sender::send (uint32_t now, uint16_t numPins, const uint16_t * pins)
{
  if (now >= mNextWriteTime)
```

```
  {
    for (uint16_t i; i < numPins; i++)
      sendOut(pins[i]);
    scheduleNextWrite(now);
    changePhase();
  }
}

void
Sender::send2 (uint32_t now, volatile uint8_t * pinRegister, uint8_t pinTargets)
{
  if (now >= mNextWriteTime)
  {
    sendOut2(pinRegister, pinTargets);
    scheduleNextWrite(now);
    changePhase();
  }
}


void
Sender::sendOut (uint16_t pin)
{
  switch (mMsgState)
  {
    case DOWN:
      digitalWrite(pin, LOW);
      break;
    case UP:
      digitalWrite(pin, HIGH);
      break;
  }
}

void
Sender::sendOut2 (volatile uint8_t * pinRegister, uint8_t pinTargets)
{
  switch (mMsgState)
  {
    case DOWN:
      *pinRegister &= 0;
      break;
    case UP:
      *pinRegister |= pinTargets;
      break;
  }
}
```

```
void
Sender::scheduleNextWrite(uint32_t now)
{
  uint32_t interval = 0;

  if (mMsgState == DOWN)
    mNextWriteTime = now + mMsgDelta;
  else
  {
    switch (mSndState)
    {
      case LEADER:
        interval = mLeader[mWritePointer++];
        break;
      case MESSAGE:
        interval = mMsg[mWritePointer++];
        break;
    }
    mNextWriteTime = now + interval*mMsgDelta;
  }
}


void
Sender::changePhase ()
{
  if (mMsgState == DOWN)
    mMsgState = UP;
  else
    mMsgState = DOWN;
  switch (mSndState)
  {
    case LEADER:
      if (mWritePointer == mLeaderLen)
      {
        mWritePointer = 0;
        mSndState = MESSAGE;
        mMsgState = DOWN;
      }
      break;

    case MESSAGE:
      if (mWritePointer == mMsgLen)
      {
        mWritePointer = 0;
        mSndState = LEADER;
        mMsgState = DOWN;
      }
```

```
      break;
  }
}
```

6.5 E. sketch.ino

```
#include "bit_pulse.h"
#include "reader.h"
#include "sender.h"
#include "parser.h"

/* ------------- MESSAGES------------- */
// message outputs
const uint16_t leaderLen = 3;
const uint16_t msgLen = 2;

const uint32_t leader[leaderLen] = { 1, 2, 1 };
uint32_t coreMsg[msgLen];

// reader and sender config
uint32_t now = 0;

uint32_t msgTimeoutDelta = 11;
uint32_t msgDelta = 75000;
uint32_t timeout = msgTimeoutDelta * msgDelta;

Reader readOne = Reader(msgDelta, timeout);
Reader readTwo = Reader(msgDelta, timeout);
Reader readThree = Reader(msgDelta, timeout);
Reader readFour = Reader(msgDelta, timeout);

Parser parseOne = Parser(leader, leaderLen, msgLen, timeout);
Parser parseTwo = Parser(leader, leaderLen, msgLen, timeout);
Parser parseThree = Parser(leader, leaderLen, msgLen, timeout);
Parser parseFour = Parser(leader, leaderLen, msgLen, timeout);

Sender sendOne = Sender(msgDelta, leaderLen, leader, msgLen, coreMsg);
BitPulse pulse = BitPulse(A0, msgLen, coreMsg);

// reading buffers and core message handling
void printBuf (uint16_t bufLen, const uint32_t * buf);
void mutateCore (const uint32_t * message);
bool coreFlatlined ();
void randomizeCore ();

// ------------------ PROGRAM ----------------------------
void setup () {
  // start debug output
```

```
  Serial.begin(9600);
  Serial.println("Startup");

  // seed randomness
  randomSeed(analogRead(0));

  // set up IR
  // 2 - 6 are outputs (this is for variation in my perf-board circuits)
  for (int i = 2; i < 7; i++)
      pinMode(i, OUTPUT);

  // 10 - 13 are inputs
  PORTB = B11111111;
  PINB =  B00000000;

  // set up audio
  pinMode(A0, OUTPUT);

  // set up internal message
  Serial.print("Message: ");
  for (uint16_t i = 0; i < msgLen; i++) {
    coreMsg[i] = random(2, 10);
    Serial.print(coreMsg[i]);
    Serial.print(" ");
  }
  Serial.println();
}

void loop() {
  now = micros();
  cli();

  // PLAY AUDIO
  pulse.play(now);

  // MESSAGE HANDLING
  sendOne.send2(now,  &PORTD, B11111100); // set pin 2 to on
  readOne.read2(now,   PINB,  B00000100); // read pin 10
  readTwo.read2(now,   PINB,  B00001000); // read pin 11
  readThree.read2(now, PINB,  B00010000); // read pin 12
  readFour.read2(now,  PINB,  B00100000); // read pin 13

  // store a leader or buffer
  if (readOne.hasWord()) {
    parseOne.parseMessage(readOne.getWord());
  }
  if (readTwo.hasWord()) {
    parseTwo.parseMessage(readTwo.getWord());
```

```
  }
  if (readThree.hasWord()) {
    parseThree.parseMessage(readThree.getWord());
  }
  if (readFour.hasWord()) {
    parseFour.parseMessage(readFour.getWord());
  }

  // mutate core, maybe

  if (parseOne.hasMessage()) {
    mutateCore(parseOne.getMessage());
    Serial.println("message one!");
    printBuf(msgLen, coreMsg);
    parseOne.listen();
  }
  if (parseTwo.hasMessage()) {
    mutateCore(parseTwo.getMessage());
    Serial.println("message two!");
    printBuf(msgLen, coreMsg);
    parseTwo.listen();
  }
  if (parseThree.hasMessage()) {
    mutateCore(parseThree.getMessage());
    Serial.println("message three!");
    printBuf(msgLen, coreMsg);
    parseThree.listen();
  }
  if (parseFour.hasMessage()) {
    mutateCore(parseFour.getMessage());
    Serial.println("message four!");
    printBuf(msgLen, coreMsg);
    parseFour.listen();
  }
  if (coreFlatlined())
    randomizeCore();

  sei();
}

// mutate the values in the core message so that they
// converge on another message
void mutateCore (const uint32_t * message) {
  uint16_t idx = random(0, msgLen);
  int32_t cW = coreMsg[idx], mW = message[idx];
  int32_t dif = cW - mW;
  if (dif > 0) coreMsg[idx] = constrain(cW - 1, 1, 10);
  else if (dif < 0) coreMsg[idx] = constrain(cW + 1, 1, 10);
```

```
}

// print out a buffer of 32-bit unsigned int values
void printBuf (uint16_t bufLen, const uint32_t * buf) {
  Serial.print("[");
  for (uint16_t i = 0; i < bufLen; i++) {
    Serial.print(buf[i]);
    Serial.print(" ");
  }
  Serial.println("]");
}

// randomize the values in the core message
// IDEA: change "personality" - likelihood of change
void randomizeCore () {
  for (uint16_t i = 0; i < msgLen; i++)
    coreMsg[i] = random(1, 10);
}

// check whether the core message has become
// exclusively one value
bool coreFlatlined () {
  uint16_t i = 0;
  uint32_t lastVal = coreMsg[0];
  bool flat = true;

  while (i < msgLen && flat) {
    flat = flat && (lastVal == coreMsg[i]);
    lastVal = coreMsg[i++];
  }

  return flat;
}
```

6.6 F. mockups/audio.scd

```
(
Ndef(\bitter, { |pan = 0, gain = 0.1|
        var words = Array.fill(5, {10.rand});
        var t = PulseCount.ar(Impulse.ar(9e3));
        var x = 1, y = 1;
        words.postln;

        Out.ar(0,
                Pan2.ar(
                        HPF.ar((
                                ((x = x | (t & words[0])) & (y = y | (t | words[1])))
                                |
```

```
                                ((x + words[2]) * (y + words[3]))
                                - words[4] % 255
                        ) / 127-1,
                                20
                        ).tanh,
                        pan, gain)
                );
}).play;
)
```