Using Arduino as sensor bank with Csound

John ffitch* Richard Boulanger†

June 12, 2020

Abstract

An alternative communication mechanism between an Arduino and Csound is proposed and described in detail, with simple examples. Comments on this design and possible developments are sought.

1 Introduction

For some time Csound has incorporated code to read from a serial line, written explicitly to connect an Arduino to Csound via USB. However the connection was direct and low level such that its use was problematic. In this document we describe an alternative mechanism which simplifies the Csound programming at the expense of a small protocol on the Arduino. We also explain the internal working of this to explain the design.

In addition we introduce three new opcodes and show their use via simple examples. All our testing has been with a 2020 Arduino Uno R3. There may be more features needed to satisfy user wishes but we believe this is an improvement on the serial opcodes.

2 The Large picture

At the heart of the design is the separation of the reading of the serial line from the presentation of signals to Csound, using a new thread to listen to the incoming serial data. The listener thread is responsible for synchronising the reading with the writing (so either end can be started first) and for parsing the data from consecutive input bytes. The sensor values are stored within the listen. When Csound wants to read a sensor value it gets the latest value from the listener and returns it to the Csound instrument. Of course this has to be done taking care of read/write conflicts.

Csound has three new opcodes to control this. In order to start reading from an attached Arduino there is the init-time opcode **ardiunoStart**. This looks similar to

*Maynooth University, Ireland
†Berklee College of Music, Boston

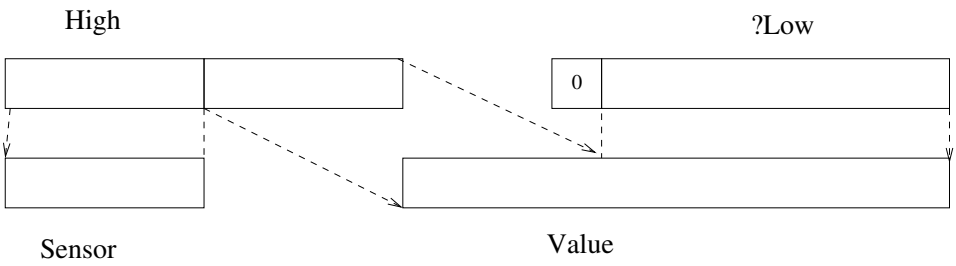

Figure 1: Allocation of bits in two bytes

serialBegin needing a serial device name and a baud rate. But as well as opening the device for reading it starts a thread that first gets in synchronisation with the input stream and then reads the data in a loop that runs continuously.

Data is read by Csound with the kontrol-rate opcode **arduinoRead**. It needs to be told which sensor it is reading and returns a value. Thread safety is achieved with a mutex.

The last Arduino opcode is **arduinoStop** which stops the listener thread. This can be omitted in which case the thread stops at the end of the program.

3 Details and Restrictions

3.1 The Data Protocol

Sensors on the Arduino generate either on/off data or an integer in the range 0 to 1024. In order to transmit 10 bits of data we need two bytes. It is a design decision that any scaling will happen on the Csound end rather than the Arduino. This gives an opportunity to provide multiple sensor data streams if we use the other bits to say to which sensor the data belongs. It also introduces a problem; when the reader starts listening to the serial line the next byte might be the least significant part or the most significant part of the two byte item.

We solve the problem by choosing a articular byte that cannot be part of any valid data, top or bottom, so reading it indicates the parity of the stream. This byte is `0xf0`. If this is to be invalid elsewhere we need to ensure the the top bit of the least significant byte is always zero, so in effect claiming one bit there and to ensure it does not happen in the most significant byte we have to make one possible sensor number invalid. This gives a data layout as shown in Figure 1.

This limits the scheme to 15 sensors numbered 0 through 14.

3.2 Restrictions in the Arduino sketch

In the sketch the construction of the two-byte data item is controlled by the function `put_val`.

```
void put_val(int sen, int val)
{
  int low = val&0x7F;
  int hi = ((val>>7)&0x0F)| ((sen&0x0F)<<4);
  Serial.write(low); Serial.write(hi);
}
```

The loop must also send the synchronisation byte at the start of each loop.

```
void loop() {
  // Any calculations
  Serial.write(0xf0);

  // Write sensor data here

  delay(10);                    /* This need tuning */
}
```

The Arduino programmer can assist performance by only sending data if it changed. A delay at the end of the loop is to tune for any data overload.

3.3 A complete example

This example has an x-y joystick generating two position values and a single digital button. We present the Arduino sketch and Csound csd file; there is a hardware description in the comments in the sketch.

3.4 The sketch

On the Arduino the sketch program is given below. It is fairly straight forward using the `put_val` function and utilising communication streams 1, 2 and 3 for Y values, X values and the button.

```
// Joystick Example

// John ffitch & Richard Boulanger
// June 11, 2020

// BreadBoard & Arduino Setup

// Insert the Joystick Controller into the breadboard
// NOTE:  Joystick Pins (in order from left to right) are:
          GND, 5V, vrX, vrY, SW
// Connect the power rails on breadboard to 5V and GND on Arduino
// Connect GND and 5V from Joystick to + and - power rails on
```

```
//         the breadboard
// Connect Joystick vrX on breadboard to Analog In A0 on the Arduino
// Connect Joystick vrY on breadboard to Analog In A1 on the Arduino
// Connect Joystick SW on breadboard to Digital Pin 2 on Arduino

// Arduino pin numbers
// Note: Because of the way that the Joystick stands up in a
//       breadboard, in this example the vrX and vrY have been swapped.

const int SW_pin = 2; // Joystick switch output connected to
                      //   Arduino digital pin 2
const int Y_pin = 0;  // Joystick Y output connected to Arduino
                      //   analog pin A0
const int X_pin = 1;  // Joystick X output connected to Arduino
                      //   analog pin A1

int lastState_SW_pin = 1;
int currentState_SW_pin;

void setup() {
              // NOTE: Digital pins can be either inputs or outputs.
  pinMode(SW_pin, INPUT_PULLUP); // Setting digital pin to input mode
                                 // and using onboard pullup resistor
                                 // to reduce noise.

  Serial.begin(9600);
}

// put_val( ) - a function to send data values to the Csound
//              "arduinoRead" opcode
// The first argument of the put_val function "int senChan" sets
// the software channel number that Csound reads
// NOTE: "senChan" does "not" define the input pin that is used on
// the Arduino for a specific sensor
// The specific Arduino input pin used by any sensor is assigned
// and set elsewhere in the Arduino sketch and mapped to a
// user-defined put_val "senChan" channel

void put_val(int senChan, int senVal)
        // Set the Csound receive channel "senChan", and read from
        // the sensor data stream "senVal"
{       // The packing of the data is ssssvvvv 0vvvvvvv where s is a
        // senChan bit, v a senVal bit and 0 is zero` bit
  int low = senVal&0x7f;
```

```
  int hi = ((senVal>>7)&0x0f) | ((senChan&0x0F)<<4);
  Serial.write(low); Serial.write(hi);
}

void loop() {

   Serial.write(0xf0);

   int currentState_SW_pin = digitalRead(SW_pin); // reading digital
                                                  // input 2 and
                                                  // assigning it to
                                                  // "currentState..."

  if (currentState_SW_pin != lastState_SW_pin)
  {                            // checking if the value has changed
    if (currentState_SW_pin == 1)
    {
      put_val(3,0); // In this sketch, the Joystick button,
                    //in Arduino digital pin 2, is sending a 0 or 1
                    // to Csound arduinoRead channel 3
    }
    else
    {
      put_val(3,1);
    }
  }

  lastState_SW_pin = currentState_SW_pin;

  int X = analogRead(X_pin); // reading the Joystick vrX data
                             //  (0-1023) and asigning to X
    put_val(2,X);            // In this sketch, the Joystick vrX,
                             // in Arduino analog pin A1, is sending
                             // a 0-1023 to Csound arduinoRead channel 2

  int Y = analogRead(Y_pin); // reading the Joystick vrY data
                             // (0-1023) and asigning to Y
    put_val(1,Y);            // In this sketch, the Joystick vrY,
                             // in Arduino analog pin A0, is sending
                             // a 0-1023 to Csound arduinoRead channel 1

  delay(10);

}
```

3.5 Csound CSD file

The Csound orchestra receives the data for the X and Y values and uses the Y value to modify the frequency of an oscillator. The button is used to turn the sound on and off.

```
Arduino-Joystick2Csound1g-fm1-port (version with smoothing)

- Push Joystick button to turn on note
- Use JoystickX to offset and control the modulation index of
  the foscil opcode from 0-40 via the scale opcode
- Use JoystickY to offset and control the Frequency of the foscil
  opcode up/down two octaves via the scale opcode

<CsoundSynthesizer>

<CsInstruments>

sr = 44100
ksmps = 441
nchnls = 2
0dbfs = 1

giport init 0

// NOTE: change USB port "/dev/cu.usbmodem1414301" to correspond
//       with USB port used by Arduino on your system
giport arduinoStart  "/dev/cu.usbmodem1414301", 9600

instr 1

 kY  arduinoRead giport, 1 ; Joystick Y
 kX  arduinoRead giport, 2 ; Joystick X
 kSW arduinoRead giport, 3 ; Joystick Button/Switch

kAmp  init 0
kFreq init 0
kIndx init 1

kXraw = kX
kX port kXraw, .02  // smoothed kY stream
kYraw = kY
```

```
kY port kYraw, .02  // smoothed kY stream

kYscaled scale kY, 400, 100, 1023, 0 ; scaling the raw sensor
                                     ; data to a user-defined
                                     ; range of (100-400)
kXscaled scale kX, 40, 0, 1023, 0    ; scaling the raw sensor
                                     ; data to a user-defined
                                     ; range of (0-40)

if(kSW == 1) then
kAmp = .333
elseif(kSW == 0) then
kAmp = 0
endif

 aOut foscil 1, kFreq + kYscaled, 1, 1, kIndx + kXscaled, 1

  outs aOut * kAmp, aOut * kAmp

printks "Button=%d, RawX=%d, ScaleX=%d, RawY=%d, ScaleY=%d \\n", \
            .5, kSW, kX, kXscaled, kY, kYscaled

 endin

</CsInstruments>

<CsScore>

f 1 0 16384 10 1

i 1 0 z

e

</CsScore>

</CsoundSynthesizer>
```

The sound from a simpler version of this example showed a small but noticeable zipper effect. In order to limit this the input values are passed through a lowpass filter **port**. See section 4 for further comments on this.

4 Constraints and Possible Improvements

The restrictions to the current scheme are twofold. First there can only be 15 sensor channels in the scheme. This may be a significant restriction for larger Arduino hardware. The second restriction is that only one Arduino can be attached to a Csound program.

It would be possible to allow multiple serial lines but it would need some re-coding. **Question**: Is it necessary?

People with careful counting of bits may notice that the value field in the data stream is actually in the range 0 to 2047. Originally this was in case it was useful, but it could be re-engineered to allow 31 sensor streams.

As seen from the simple example above the data transferred can be noisy and subject to jitter. The additional use of a **port** opcode is one way to do is but probably better is to add an optional filter to **arduinoRead**, making the **port** internal and easy to use. At the time of writing this is not operational but should be added soon.

Similarly the data is in a fixed range [0,2047]. This could be scaled to a user requirement, implemented in‘ arduinoRead but he scale opcode is arguably sufficient, especially in its new revised state[1]. **Question**: is this important?

5 Conclusions

The scheme has been tested successfully on GNU/Linux and Macintosh. The code is written so it compiles on Windows but has not been tested at all. We think the scheme is a significant improvement of just using **serialRead** with the difficulties of decoding the data, ignoring cases of no data and possible blocking.

Comments and suggestions welcome.

[1] As of 12 June 2020 in github.