New Arduino Opcodes to Simplify the Streaming of Sensor and Controller Data to Csound

John ffitch[1] and Richard Boulanger[2]

[1] Alta Sounds
[2] Berklee College of Music
jpff@codemist.co.uk rboulanger@berklee.edu

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

Keywords: Csound, Arduino, UNO R3, sensors, controllers

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 that simplifies the Csound programming at the expense of a small protocol on the Arduino. We introduce three new opcodes and show their use via simple examples.[3] We explain the internal working of the current and are open to suggestions for additional features, 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 synchronizing 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 listener. 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 **serialBegin** needing a serial device name and a baud rate. As well as

[3] All our testing has been with a 2020 Arduino UNO R3.

opening the device for reading, it starts a thread that first gets in sync with the input stream and then reads the data in a loop that runs continuously.

Data is read by Csound with the k-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 1023. 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, particularly 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.

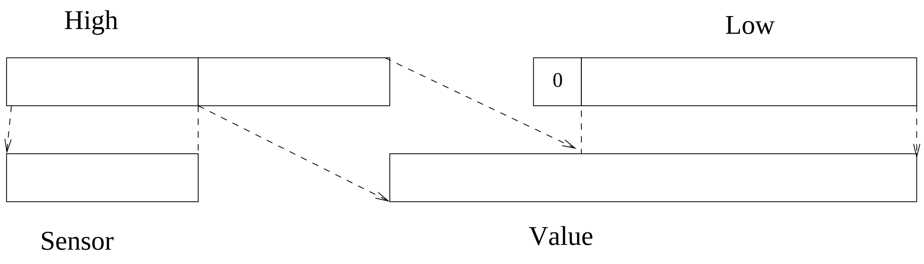

Fig. 1. Allocation of bits in two bytes

We solve the problem by choosing a particular byte that cannot be part of any valid data, top or bottom, so reading it indicates the parity of the stream. This byte is `0xf8`. 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 31 sensors numbered 0 through 30.

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 senChan, int senVal)
{
  int low = senVal&0x7F;
  int hi = ((senVal>>7)&0x07)|((senChan&0x1F)<<3);
  Serial.write(low); Serial.write(hi);
}
```

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

```
void loop()
{
            // Any calculations
  Serial.write(0xf8);
            // Write sensor data here
  delay(10);  /* User can fine-tune here */
}
```

The Arduino programmer can assist performance by only sending data if it changed. A delay at the end of the loop is to adjust 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 Arduino Sketch

On the Arduino the sketch program is given below.[4] It is fairly straightforward, using the `put_val` function and utilizing communication streams 1, 2 and 3 for Y values, X values and the button. Here is the sketch for the Joystick Example:

```
const int SW_pin = 2;// Joystick switch connected to digitalPin 2
const int Y_pin = 0; // Joystick Y out connected to analogPin A0
const int X_pin = 1; // Joystick X out connected to analogPin A1
     int lastState_SW_pin = 1;
     int currentState_SW_pin;
     int lastState_SW_pin = 1;
     int currentState_SW_pin;
```

[4] Breadboard and Arduino Setup for the JOYSTICK EXAMPLE: 1. Insert the Joystick Controller into the breadboard [NOTE: Joystick Pins (in order from left to right) are: GND, 5V, vrX, vrY, SW]; 2. Connect the power rails on breadboard to 5V and GND on Arduino; 3. Connect GND and 5V from Joystick to + and - power rails on the breadboard; 4. Connect Joystick vrX on breadboard to Analog In A0 on the Arduino; 5. Connect Joystick vrY on breadboard to Analog In A1 on the Arduino; 6. Connect Joystick SW on breadboard to Digital Pin 2 on Arduino.

```
     void setup()
     {
// NOTE: Digital pins can be either inputs or outputs.
// Digital pin set to input & using pullup resistor to reduce noise.
       pinMode(SW_pin, INPUT_PULLUP);
       Serial.begin(9600);
     }
// put_val( ) - send data values to the "arduinoRead" opcode
// The first argument, "int senChan" sets the software channel
// 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, set, and mapped elsewhere in sketch
// Set the Csound receive channel "senChan", and read from
// the sensor data stream "senVal"
// The packing of the data is sssssvvv 0vvvvvvv where s is a
// senChan bit, v a senVal bit, and 0 is a zero bit
     void put_val(int senChan, int senVal)
     {
       int low = senVal&0x7f;
       int hi = ((senVal>>7)&0x07)| ((senChan&0x1F)<<3);
       Serial.write(low); Serial.write(hi);
     }
     void loop()
     {
       Serial.write(0xf8);
       int currentState_SW_pin = digitalRead(SW_pin);
// reading digital input 2 and assigning it to "currrentState..."
       if (currentState_SW_pin != lastState_SW_pin)
       {
// checking if the value has changed
         if (currentState_SW_pin == 1)
         {
// In this sketch, the Joystick button, in Arduino digital pin 2,
// is sending a 0 or 1 to Csound arduinoRead channel 3
           put_val(3,0);
         }
         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, analog pin A1,
// is sending 0-1023 to Csound arduinoRead channel 2
// reading the Joystick vrY data (0-1023) and asigning to Y
       int Y = analogRead(Y_pin);
       put_val(1,Y);
// In this sketch, the Joystick vrY, analog pin A0,
// is sending a 0-1023 to Csound arduinoRead channel 1
       delay(10);
     }
```

3.5 The Csound Instrument

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.

```
giport init 0
giport arduinoStart "/dev/cu.usbmodem1414301",9600
gisin ftgen 1, 0, 16384, 10, 1

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 kX stream
  kYraw = kY
  kY port kYraw, .02  // smoothed kY stream
  kYscaled scale kY, 400, 100, 1023, 0 // scaling raw sensor
  kXscaled scale kX, 40, 0, 1023, 0  // scaling the raw sensor
    if(kSW == 1) then
        kAmp = .333
    elseif(kSW == 0) then
        kAmp = 0
    endif
  aOut foscil 1, kFreq + kYscaled, 1, 1, kIndx + kXscaled, 1
       outall aOut * kAmp
endin
```

4 Support for Floating-Point Sensors and Controls

The protocol described in this paper transfers integer values from the Arduino to Csound. Another need is to transfer floating-point numbers similarly. Single precision floating point on the Arduino should be sufficient and, with a small loss of precision, three 10-bit integers are sufficient to carry a floating-point Arduino sensor value to Csound. Support for this is provided with the **arduinoReadF** opcode which delivers the packing/unpacking using the integer protocol detailed in this paper.

5 Constraints and Possible Improvements

Restrictions to the current scheme are twofold. First, there can only be 31 sensor channels in the scheme. This may be a limitation when using larger Arduino hardware. A 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.

Also, 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 deal with this, but it would probably be better to add an optional filter to **arduinoRead**, making the smoothing with **port** internal and easy to use. This has been implemented in the latest version.

Similarly, the data is in a fixed range [0,1023]. This could be scaled to a user requirement, and implemented in the arduinoRead opcode, but the scale opcode is arguably sufficient, especially in its new revised *scale2* state[5].

6 Conclusions

The scheme has been tested successfully on GNU/Linux and Macintosh. The code is written so that it compiles on Windows, but it has not yet been tested on that platform. This scheme offers significant improvement over simply using **serialRead** with the associated difficulties of decoding the data, ignoring cases of no data, and possible blocking. Comments and suggestions are welcome.

References

1. ffitch, J.: Introduction to Program Design. In: R. Boulanger, V. Lazzarini (eds.) The Audio Programming Book, pp. 383–430. MIT Press, Cambridge (2010)
2. Vercoe, B.: Real-Time Csound, Software Synthesis with Sensing and Control. In: Proceedings of the International Computer Music Conference, pp. 209–211. Glasgow (1990)
3. Csound GitHub Page – `https://github.com/csound/csound`

[5] As of 12 June 2020 in GitHub.