Dr. E.Huckert 05-2021

Audio sampling and data transfer with Arduino Due

Inhaltsverzeichnis

Overview

- Samples audio data on Arduino Due
- Transfers the audio samples on a PC (Windows or Linux)
- Uses a serial line (or USB/Serial) to transfer the data
- Performance is quite good but not good enough to replace a sound card
- Two solutions with different protocols are offered

Design

Why Arduno Due?

The Arduino Due has a 32 Bit CPU running at 84 Mhz. Compared to a standard Arduino (Uno, Mega, Nano etc.) it is a **factor 2.3 faster** . The interesting feature for audio projects: it has 12 ADC with up to 12-bit precision.

Arduino Due offers the fastest sampling rates - up to 220000 samples/sec in my tests. I have also tested other microcontrollers (standard Arduino, Espressif ESP32): the Arduino Due was just the fastest.

Many microcontrollers do not have ADC pins. Others (like BeagleBone or Raspberyy Pi) run Linux as operating system. Linux is normally not suited for near-real-time operations as a multitasking / multiuser operating system cannot guarantee exact reaction times.

Why serial transfer?

Using the serial line (or rather the native USB port used as a serial line) is the only way to get the samples in near-real-time. There is **no network** (LAN or WLAN) directly available on Arduino Due. We use here 115200 Baud. This results in a theoretical transfer throughput of approx. 11520 bytes per second.

The problem with serial lines at high baud rates: the cables cannot be very long. When using 115200 Baud the cable length should not exceed some meters.

Why only 8-bit samples?

Memory is rare on Arduino Due. The Arduino Due has 96 KB static RAM (divided in two banks) and 512 KB Flash memory.

For a standard CD quality (stereo, 16 bit samples, 44100 samples/sec) 176400 bytes are required per second. This requires more memory than the SRAM on the Arduino offers - not taken into account that this SRAM is also used for other things (gloabl variables, heap).

Using **8-Bit samples** instead of the usual 16-bit samples reduces memory by 50 %. The same argument holds for the use of approx. **10000 samples/sec** instead of 44100 samples/sec in CD quality audio. Using mono (1 channel) **instead of stereo** (2 channels) reduces also memory usage on the Arduino side.

Using smaller samples, smaller sample rates and less channels reduces also the transfer time considerably.

Why a special receiver program?

We need a small protocol (see the comments in the code) to start the sampling on the Arduino and to transfer packets without overrunning the receiver. Standard programs like *TeraTerm* (Windows) or *minicom* (Linux) do not not include this protocol.

Our special receiver program rxArdDue4.cpp (1st version) performs these tasks:

- communicate with the Arduino Due via RS232 or USB/Serial
- sends a simple start signal (a '+' char) to tell the Arduino that it should start the sampling process and the download of the sample data
- receive the sampled audio data and writes a WAVE file

The Arduino Due sketch (first version ADCRead.ino) performs these tasks:

- waits for a start signal vias USB/Serial on the "native port"(a simple '+' char)
- samples input data for approx. 1 second
- sends the sampled 8-bit data (1 byte per sample) via USB/Serial. The data aer send in packets. Each packet has a 4-ASCII bytes length at the beginning.

The first approach uses two clearly separate steps for sampling and for download. The **protocol** used for the download of the sampled data is very simple. It includes no data integrity mechanisms like checksums etc. The only meta data used is a 4-bytes, ASCII encoded length of the packets. The sender (Arduino Due) may overrun the receiver. Including a more sophisticated protocol (with handshake etc.) will probably result in a poorer download performance.

The **2nd version** (sketch *ADCRead2.ino* and program *rxArdDue5.cpp*) uses a much simpler approach: the PC program sends a '+' character to start the sampling and download processes. Compared to the 1st approach sampling and data transfer are not separate steps: as soon as a sample (a byte) has been aquired it will be sent out - no length information is used. As the baud rate is high and as the ADC sampling is fast enough this second approach may even be more efficient and precise than the first approach. Here also the sender (Arduino Due) may overrun the receiver.

Performance

My tests with approx. 10000 samples/sec (exactly 9532 Samples/sec, but this may vary) resulted (sum over both operating sytems) in ca. **1150 ms** (1st approach) for sampling and transfer (115200 Baud). The second approach is even faster : approx. **1050 ms** for 10000 samples including sampling and transfer.

Note that for reasons explained earlier (small SRAM in Arduino Due) I used the **flash memory** to store the samples in the Arduino Due (keyword PROGMEM).

Why not a better performance?

Compared to the sound card in your PC this is not overwhelming and therefor cannot replace the sound card. The reasons:

- a sound card is a specialized **internal** device
- it can use **much faster communication** (probably DMA)

Compilation

Windows

with GNU g++ (here for the 1st approach):

```
g++ -o rxArdDue.exe -DWIN32 -static rxArdDue2.cpp hu_rs232u.cpp
      sound\WaveClass10.cpp
```

Linux (Ubuntu)

```
g++ -o rxArdDue2 -static rxArdDue2.cpp hu_rs232u.cpp
      sound/WaveClass10.cpp
```

Note that the support class file *WaveClass10.cpp* is expected in a subdirectory (can be changed).

The ZIP files contains two very simple procedures to be run under Windows: **ma4.bat** (1st approach) and **ma5.bat** (2nd approach).

Other compilers (I used g++) may produce errors as my program uses a special method in ***STL/vector*** that seems not to be implemented in all STL library versions.

Arduino Due

I used the standard Arduino IDE to compile and install the sketch ADCRead.ino

Start the application

Not the difference in the calls on the command line: under Windows a com port number is needed, under Linux a driver name!

Windows (here for the 1st approach)

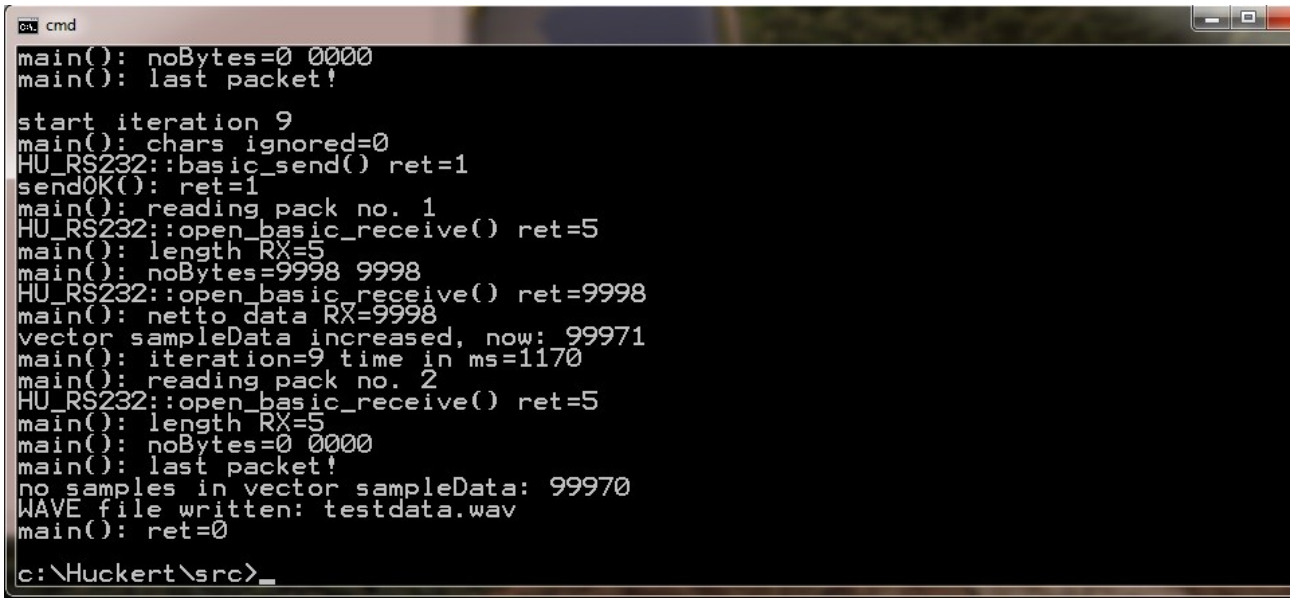
```
rxArdDue4 -comno 35 -baud 115200 -dtrrts -v
```

Note 1: on my Windows 7 machine the COM number was 35, on my Windows 10 machine it was 4! The respective COM number may be different from machine to machine. Use TeraTerm or a similar program to find it out.

Note 2: option -dtrrts (=enable DTR and RTS) seems to be necessary with Windows drivers

Note 3: for performance reasons the option -v should be omitted

This screen shot shows how the program is started under Windows. It also show the basic output (option -v used here):



Linux (here for the 2nd approach)

The application can be started on the command line as follows:

```
rxArdDue4 -dev /dev/ttyUSB0 -baud 115200 -v
```

Note 1: on Linux you have to use a driver name instead of a COM number - hence the option -dev /dev/ttyUSB0

Note 2: this driver name may vary. Use command ls -l /dev/tty* to find out which driver may be used. The driver name could also be */dev/ttyACM1* or */dev/ttyACM2* or ...

Note 3: option -dtrrts (sets DTR and RTS) was not needed on my Linux machines.

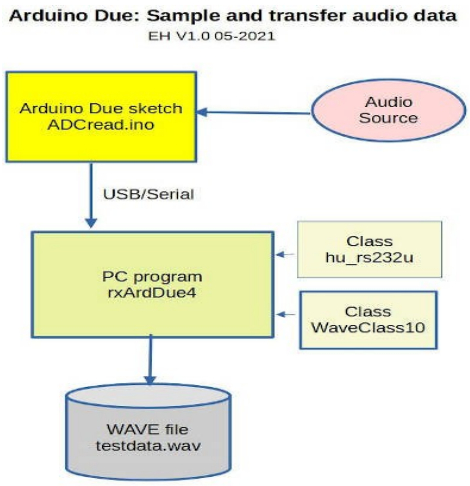
The source code

The code consists of 6 source files written in C or C++:

- an Arduino Due sketch called ADCRead.ino (1st approach)
- an Arduino Due sketch called ADCRead2.ino (2nd approach)
- a main PC application (no GUI!) called rxArdDue4.cpp (1st approach)
- a main PC application (no GUI!) called rxArdDue5.cpp (2nd approach)
- an auxiliary class for RS232 or USB/Serial called hu_rs232u.cpp and hu_rs232u.h (not commented here)
- an auxiliary class for WAVE generatiion called WaveClass10.cpp and waveclass10.h (not commented here)

The source code is commented on the Arduino Due side and on the PC side. You will find the complete sources in this zip file: www.huckert.com/ehuckert/programs/rxArdDue.zip.

This picture shows the software and process structure. The general structure is the same for both approaches:



Audio hints

The ADC inputs on the Arduino Due operate for voltages between 0 and 3.3V. They can be configured for 10-bit (=default) or 12-bit precision. We used the 10 bit default as our samples were written as 8-bit samples in the WAVE - so the maximum voltage can even not be used.

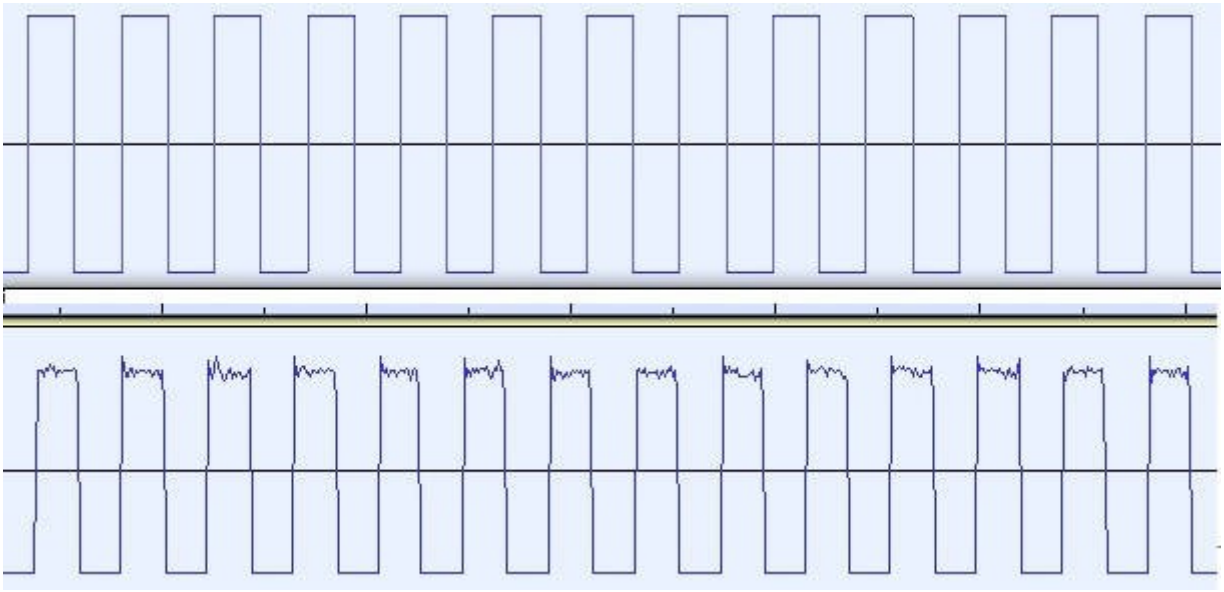
The input pin should be protected against bad input signals - a 100 Ohm resistor should be sufficient.

If you use a microphone as input source then you will get probably no usable inputs. If you use a microphone amplifier the take care that it doesn't produce more noise than usable input!

Do not use condenser microphone. They use up to 40V for their internal amplifier. If 40V are (accidentally) applied to an ADC input it may damage the microcontroller.

I have tested this with a second PC used as signal source. It is very easy to produce WAVEs in program **Audacity** (Windows or Linux) and to play them over an audio cable connecting the signal source PC and the ADC input of the Arduino Due.

The following picture shows the original signal (a 220 Hz square wave, top) and the sampled signal (bottom):



Hints for optimization

Instead of producing a WAVE file that can be played later on the program could also play the samples directly after receiving them. This works better with the first approach (where samples are sent in blocks) than with the second approach.

The Arduino Due sketch (1st approach) could be changed (I didn't test that) so that it samples the data in a timer based interrupt routine. The main routine ("loop()") could then concentrate of the sind the sample data. This can avoid the audible glitches in the WAVE files produced by the discontinuous sampling process.

As mentioned above the Arduino Due is fast enough to sample data and send the samples in one step - this is the second approach. But this is only true for the given sampling rate (approx. 10000 samples/sec) and the given baud rate (115200).

A microcotroller like the Espressif ESP32 is probably the better choice (but the ADC in standard mode is slower). It has faster communication devices (WLAN), has a faster CPU and even better: has two processors. At the same time one processor could do the sampling and the secod processor could do the data transfer time.