Arduino Analyzer

Luís Manuel Ribeiro Silva
luis.m.ribeiro.silva@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2020

Abstract

While developing and debugging circuits or digital systems it is useful to visualize, capture and generate electric signals. This is needed in some courses in the area of Embedded Systems or Cyber-Physical Systems at IST, where students may work with sensors and digital communication during laboratory classes and in their projects for these courses. These tasks require specialized and expensive equipment that is not usually available in Computer Science laboratories nor at students' homes. Arduino Starter Kits are already available in some laboratory rooms at IST.

This document describes the system that was developed to allow an Arduino Uno and a computer application to be used as an oscilloscope, logic analyzer and signal generator. It is able to sample up to 6 analog channels and up to 8 digital channels simultaneously. Captured signals can be stored for later analysis. Using an external DAC, the system is able to generate analog signals with various shapes. The logic analyzer functionality allows samples on multiple digital channels to be interpreted in order to decode and display I2C communication. The system evaluation shows that it is able to continuously sample a single analog channel at 76.9 kSps or 6 digital channels at 50 kHz, although with some constraints. Sample rates can be higher if not continuous, reaching 5.33 MHz for 6 digital channels.

A few laboratory exercises are described to demonstrate some of the functionality. The system and these exercises were improved based on user feedback.

Keywords: Arduino, Oscilloscope, Logic Analyzer, Signal Generator, I2C, Laboratory Classes

1. Introduction

While developing, prototyping, testing and debugging circuits and digital systems it is useful and may be necessary to measure, visualize and capture signals, generate electronic waves and, for digital circuits, to visualize and capture multiple signals simultaneously. This is needed in courses in the area of Embedded Systems or Cyber-Physical Systems, where students work with sensors and Inter-Integrated Circuit (I^2C) communication in laboratory classes or projects.

This requires specialized tools such as oscilloscopes, signal generators and logic analyzers, which are expensive, heavy, and large [1]. These are not available on laboratories for Computer Science students and can be difficult to use. Also, most Computer Science students do not have experience with such devices or Electronics nor have this equipment while developing projects at home.

An Arduino Uno [1] is included in the Arduino Starter Kit [2], which is already available on some laboratory rooms at Instituto Superior Técnico (IST). It can read digital inputs and includes an analog-to-digital converter (ADC) to sample analog signals. Samples can be sent over Universal Serial Bus (USB) to a Personal Computer (PC) application to plot waveforms, as a digital oscilloscope does. It includes a breadboard, jumper cables, multiple Light-emitting diodes (LEDs), resistors and other components. These resistors are enough to build at least a 4-bit R-2R digital-to-analog converter (DAC) to output analog signals.

A device with some functionality of these tools would improve laboratory classes for digital communication and sensor interfacing without requiring expensive equipment and hours of training. It may also be useful for students to debug and test their projects, even at home.

This work is carried in the scope of the Cyber-

[1] https://store.arduino.cc/arduino-uno-rev3, last accessed on 18th May 2018

[2] https://store.arduino.cc/genuino-starter-kit, last accessed on 2nd May 2018

Physical Laboratories proposal in Pedagogic Innovation Projects (PIP) 2018 at IST.

1.1. Objectives

This project aims at implementing some functionality of an oscilloscope, logic analyzer with I²C decoding, and signal generator using only components from the Arduino Starter Kit, as it is available on laboratory rooms at IST, and a PC. The system [3] is to be used by Master of Science program in Computer Science and Engineering (MEIC) students at IST. A single Arduino board should support all functionality to optimize board allocation to multiple groups of students in the same room.

These are the objectives:

- Sample analog signals on the Arduino;
- Capture digital signals on the Arduino;
- Send sampled data to a PC;
- Display this data on a PC;
- Display I²C protocol analysis on a PC;
- Provide a user-friendly interface on the PC;
- Write laboratory guides that will also be used to evaluate this work.

2. Related Work

This section explores how the Arduino platform is used in some educational contexts and projects with similar or related functionality. Since this project is to be used in laboratory classes for some courses at IST, this institution is explored first.

2.1. Applications in Higher Education

The Arduino platform has been used in teaching in many ways, in multiple projects and institutions.

IST At IST, the Arduino platform has been used in various courses where this project could be useful. In some MEIC courses, students must use Arduino boards in their projects, while in others they may choose to Arduino boards.

In the Software for Embedded Systems course [2], now named Applications and Computation for the Internet of Things (ACIC)[4], students use an Arduino Starter Kit in laboratory classes and to develop their projects, during which they can take it home. This course focuses on cyber-physical systems for Internet of Things (IoT). Students are introduced to the Arduino environment by programming one to control LEDs depending on temperature, a potentiometer, and light intensity; I²C communication is also studied. Students then develop an automatic traffic lights project with some fault tolerance and multiple Arduino Uno boards communicating over I²C. The same protocol is used by all students and implementations must be interoperable, therefore it must be correctly implemented.

The Ambient Intelligence (AI) course [5] presents this concept and the use of Information Technology (IT) systems for monitoring, control and interaction with environments frequented by people. Students must develop a project that monitors an environment and reacts to changes in it; some students choose to use Arduino boards.

The Internet of Things Interaction Design (DIIC) course [6] deals with IoT, but focuses on design and interaction. The use of Arduino and other platforms for interactive objects in IoT prototypes is studied. Students develop an IoT project to tackle a choosen challenge. Available hardware for projects includes the Arduino Starter Kit, a Raspberry Pi, and various sensors.

Other Institutions Many other institutions use Arduino boards in laboratory exercises, including using RGB LEDs and various types of displays to study and build a display module controlled by an Arduino [3], increase student engagement [4], motivate them [5], and in Embedded Systems course projects [6]. These can even be used as a Digital Signal Processor (DSP) [7], with one Arduino outputing a random analog signal (with an 8-bit R-2R DAC) that is sampled, filtered and output by another Arduino. It is also used to control LEDs and actuators [8]–[11], read data from various sensors, and as a Data Acquisition (DAQ) device.

2.2. Oscilloscope Functionality

Girino Girino [7] constantly samples to a circular buffer. It uses an Interrupt Service Routine (ISR) to detect a value on a signal and capture an amount of samples to a buffer, then sent to a PC to be displayed. It is missing an interface for visualizing data. Using assembly instructions and 8-bit analog samples increases speed and saves RAM.

Arduino Oscilloscope (6-Channel) There are instructions and code available to sample up to 6 digital channels with triggering support on an Arduino board [8], with support for continuous sampling, a pause switch, and trigger with voltage on a fixed

[3] https://web.tecnico.ulisboa.pt/ist177944/Arduino-Analyzer/, last accessed on 19th October 2020

[4] https://fenix.tecnico.ulisboa.pt/disciplinas/ACPIC/2018-2019/1-semestre/, last accessed on 19th July 2020

[5] https://fenix.tecnico.ulisboa.pt/disciplinas/AI514/2019-2020/2-semestre/, last accessed on 19th July 2020

[6] https://fenix.tecnico.ulisboa.pt/disciplinas/DIIC-2/2018-2019/1-semestre/, last accessed on 19th July 2020

[7] http://www.instructables.com/id/Girino-Fast-Arduino-Oscilloscope/, last accessed on 19th April 2018

[8] https://create.arduino.cc/projecthub/Meeker6751/arduino-oscilloscope-6-channel-674166, last accessed on 18th May 2018

channel and rising or falling edge.

xoscillo xoscillo[9] can acquire data in various hardware platforms, including Arduino. It supports viewing multiple waveforms simultaneously, real-time Fast Fourier Transform (FFT), zooming and other functions. It can sample 1 analog channel up to a sample rate of 7 kHz and 4 channels at $7/4$ kHz, with 8-bit samples.

Instrumentino Instrumentino [9] allows creating Graphical User Interface (GUI) programs to control an Arduino's Input/Output (I/O) pins, and monitor sensors and actuators from a PC. It can produce logs detailing when a sequences of commands is run and signal-logs in Comma Separated Values (CSV). The Controlino sketch parses and acts depending on commands from the PC.

Many projects use an Arduino to act as an oscilloscope [12]–[14]. Intrumentino is too slow for this, as explicitly requesting each sample certainly limits its sample rate. Girino uses assembly for low-level control of the ADC and Analog Comparator, but only supports 1 channel to maximize sample rate. Lower-level control may be interesting to avoid library overhead. Xoscillo, Arduino Oscilloscope (6-Channel) and Arduinolyzer.js[10] , support multiple channels. Arduino Oscilloscope (6-Channel) displays waveforms on the Serial Plotter, while Arduinolyzer.js uses an HTML5/JavaScript interface. Girino has no data visualization software. Other projects use external displays [1], [14], which are not flexible as a PC interface. A voltage divider to sample up to 200 V [13] is interesting, but can be dangerous and is not need considering work carried in the laboratories. An external ADC [8], [15], provides more precision and higher sample rate but requires additional software and hardware.

2.3. Logic Analyzer Functionality

Reviewed projects simply show multiple waveforms in parallel [16], but none decodes protocols. This project aims to also do protocol decoding.

2.4. Signal Generation Functionality

"Smart Electronic Kit" includes an oscilloscope that uses analog inputs on an Arduino Mega and an Liquid-Crystal Display (LCD) for oscilloscope functionality, and a signal generator that uses an Integrated Circuit (IC) [14]. Another project uses an 8-bit DAC IC [17]. An Arduino can generate square waves on digital pins [12] by toggling pin output in software or using Pulse Width Modulation (PWM) functionality.

Instead of external ICs, an R-2R DAC circuit [7], allows outputing software-generated shapes. The Arduino Starter Kit has enough resistors to build at least a 4-bit R-2R DAC, eliminating the need for additional ICs to generate analog signals.

3. Architecture

This section describes the system requirements, expected limitations, and an overview of its architecture. Implementation is detailed in section 4.

3.1. Requirements

Since the system is to be used in laboratory classes at IST, it must only use material available on the laboratory rooms, notably the Arduino Starter Kit [11] , which includes an Arduino Uno, and a PC (or student laptops). Based on work and projects that students develop, the system should:

- Sample more than one analog channel — to allow viewing multiple signals simultaneously;
- Perform basic analysis such as showing minimum and maximum signal values, and FFT;
- Sample more than 2 digital channels (for I^2C) plus additional channels at the same time;
- Decode I^2C protocol;
- Generate electric signals;
- Store samples for later analysis.

3.2. Functional Architecture

The system uses software — *sketch* — running on an Arduino Uno to sample and generate signals and a PC Application — *App* — to display, decode, and store data, and to control sketch parameters or output pin state. The sketch and App were developed for this purpose. This sketch must be uploaded once to the Arduino with, for instance, the Arduino IDE. Data is transferred using a custom protocol over the USB Universal Synchronous and Asynchronous Serial Receiver-Transmitter (USART). The sketch sends samples and parameters when these change. Depending on the sampling mode, samples may be stored on the Arduino before transmission, or alternately sampled and transmitted. An overview of the system architecture is shown in Figure 1.

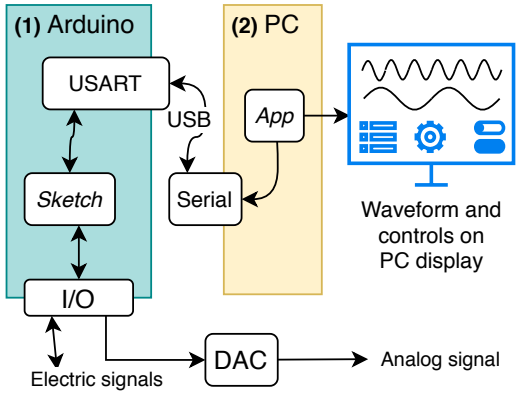
The sketch uses the Arduino libraries to, in particular, interface the USART. It sends samples to and reads commands from the App. Commands execute actions, such as toggling pin output, or control parameters such as sampling mode and number of input channels. The sketch supports various sampling modes to increase flexibility and maximize sample rate. The sketch main loop executes code depending on the current mode. Like

[9] https://code.google.com/archive/p/xoscillo/, last accessed on 18th May 2018

[10] http://www.instructables.com/id/Arduinolyzerjs-Turn-your-Arduino-into-a-Logic-Anal/, last accessed on 23rd May 2018

[11] https://store.arduino.cc/genuino-starter-kit, last accessed on 2nd May 2018

Figure 1: System Architecture Overview



one pass of the electron beam on an analog oscilloscope, one iteration of the main loop, or the set of samples it produces, shall be called a *sweep*.

The sketch generates samples for a few shapes or accepts samples for custom shapes from the App. Samples for one period of the shape are stored in the Arduino's Static Random-Access Memory (SRAM). An external DAC is needed to output analog signals.

The App is built using the Electron [12] framework, chosen since it allows developing desktop applications with web technologies, namely JavaScript, HTML5, and CSS, and because it allows targeting Linux, Mac, and Windows, using the same codebase. The following libraries are used by the App:

- *Node SerialPort* [13] — to interface the Serial Port on the PC.
- *Papa Parse* [14] — to read and parse CSV files containing shapes for the Wave Generator.
- *node-fft* [15] — to compute FFT on samples for each input channel.

The App has a GUI to display and control sketch parameters such as sampling mode, sample rate, number of input channels, output pin values and Wave Generator parameters. It receives sweep samples from the sketch to plot waveforms for each channel using HTML5 Canvas. Channels can use custom colors and title labels. Minimum, maximum, and FFT are computed on the App for each channel. Sweeps can be exported as PNG, CSV or saved as a session file in JSON, which can be loaded later for visualization and analysis without an Arduino. The App also provides logic analyzer functionality to decode and display I^2C communication. An XY mode is included.

[12] https://www.electronjs.org/, last accessed on 19th July 2020
[13] https://serialport.io/, last accessed on 19th July 2020
[14] https://www.papaparse.com/, last accessed on 10th August 2020
[15] https://github.com/vail-systems/node-fft, last accessed on 10th August 2020

3.3. Expected Limitations

Considering processing power on current PCs and that the Central Processing Unit (CPU) on the Microcontroller Unit (MCU) used on the Arduino Uno — the ATmega328P — is runs at 16 MHz, it is clear that the Arduino will limit the system. The USART will also limit continuous sampling. The system should continuously sample and display signals in real-time. When sample rate is higher than the USART can transmit, samples may be stored in SRAM on the Arduino before being transmitted. Not all 2 KBytes of SRAM can be used, to leave space for the stack and other variables.

Input signals must be between 0 V and 5 V due to I/O pin limitations and analog samples may be less accurate if USB voltage is not exactly 5 V. Only 12 of the 14 digital I/O pins can be used as input or output since the USART uses 2 pins. The ADC samples up to 6 analog inputs in round-robin; there will be a negligible time offset between these. There is also an offset between analog and digital samples. The App considers these while plotting.

4. Implementation

This section details part of the implementation of the architecture described in section 3, which is composed of an Arduino sketch and a PC App.

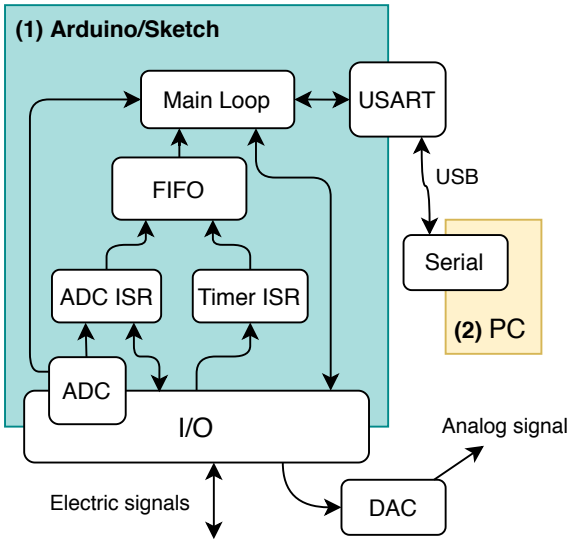
4.1. Sketch

The sketch has initialization code and a main loop, which samples inputs and checks for incoming commands from the App. To increase flexibility and sample rates, multiple sampling modes were implemented. The main loop runs code depending on the sampling mode. Some modes store samples on a First In, First Out (FIFO) before these are sent to the App; others can fill a buffer before transmission. After each loop iteration, the sketch checks for commands to process. An overview of the sketch is shown in Figure 2. Since some parts use AVR libraries or assembly to avoid overhead or control the MCU, the sketch is only guaranteed to run correctly on the Arduino Uno or same MCU.

4.1.1. Sampling

The ADC outputs 8 bits per sample instead of 10 to reduce code complexity and improve sample rate. Only up to 8 digital channels (bits) are sampled so these fit in the same byte to simplify transmission of analog and digital samples. Digital channels in the same hardware PORT are sampled at the same time in a single instruction.

Figure 2: Sketch Overview



Samples have no timing information. Sample rate is known or computed based on the time the sketch spends in a sweep. Some modes use ISRs at a known rate. *Loop* modes either write samples to a buffer (*fill*) before transmitting them all, or transmit each sample after acquiring it (no buffering). The *Loop (analog)* mode measures sweep duration to compute sample rate, as does the *Loop (digital)* mode when not filling a buffer. When the *Loop (digital)* mode is filling a buffer, it uses routines that take a known number of CPU cycles per sample; sample rate is computed based on that.

4.1.2. FIFO Queue

Modes using ISRs produce samples to a FIFO, implemented as a circular buffer on the Arduino's SRAM. Using the FIFO allows modes to sample continuously if the USART transmits fast enough. Other modes may use its internal buffer directly, which can hold 1024 samples (bytes). Only 1023 are allowed on the FIFO as to not keep a variable with sample count. The modulo operation for 1024 is implemented with an AND instead of an expensive procedure. This is half the SRAM on the Uno, so space is left for the stack and other variables.

Depending on mode, an ISR produces samples to the FIFO. These are consumed one-by-one by the main loop, which writes them to the USART. After each iteration, commands from the App are processed, during which samples are not transmitted but can be continuously produced to the FIFO until it becomes full. If it becomes full, samples are ignored between each sweep to avoid jitter; the App will advise the user that sweeps are *spaced*. Sweeps may be "truncated" if this is before the requested number of samples is acquired.

4.1.3. Sampling Modes

Disabled Only processes commands from the App, such as toggling the state of output pins.

Free Running ADC The ADC is in Free Running Mode. ADC clock depends on the $prescaler$; the ADC ISR is runs after each sample is ready. Each sample takes 13 ADC clock cycles. With one channel, the sample rate in this mode is $F\_CPU/prescaler/13$ Hz; F_CPU is the CPU clock (16 MHz). The App computes the sample rate based on this. When sampling multiple channels the ADC sample rate is divided by the number of analog channels; when the ISR for channel n is executed, it is already sampling channel $n+1$ and will be configured for $n+2$. Sampling wraps to the first channel after the last channel. Digital channels may be sampled in the ISR for the last analog channel. Advantages of this mode include known sample rate for 0 to 6 analog and 0 to 8 digital inputs, plus continuous sampling if the baud allows.

Timer (digital) Timer 1 (16-bit) is runs an ISR at the desired sample rate and 6 digital inputs pins are read from a PORT. Timer 1 is used since it allows a larger number of frequencies. Advantages include known sample rate, wide selection of sample rates, and continuous sampling if the baud allows. The ISR adds overhead that limits sample rate. *Loop* modes do not use ISRs to avoid this.

Loop (analog) Captures multiple analog channels busy-waiting for the ADC to finish each conversion. Samples are either sent to the App before a new conversions (no buffering; sample rate limited by USART) or put in a buffer which is filled before the sweep is transmitted. There is no ISR overhead, but sample rate is not easily predicted.

Loop (digital) Supports sampling 6 digital inputs in busy-wait. Either each sample is immediately sent to the App (no buffering; sample rate limited by USART) or special sampling routines fill a buffer with 1024 samples before the sweep is transmitted. There is no ISR overhead and ISRs are disabled to avoid losing samples at high sample rates.

The routines repeat assembly instructions in program to read and store samples in SRAM, trading flexibility and larger program size for higher sample rate. These are generated by C macros to avoid repetitions in source code. AVR assembly is used to avoid compiler optimizations that would affect the number of cycles for the routines. The App chooses the best routine and how many cycles to waste per sample to get the desired sample rate.

The fastest routine takes 3 CPU cycles per sample, which with a CPU clock of 16 MHz results in a sample rate of 5.33 MHz, for 6 digital channels. Instructions are repeated 1024 times in the program to fill a buffer with 1024 samples. Using `NOP` instructions or inner loops, routines waste an arbitrary number of cycles per sample to get various sample rates.

The sampling routines give a large selection of predictable sample rates, but increase program size, always capture a very small intervals, and sampling is not continuous.

4.1.4. Trigger

Trigger allows starting to acquire on a condition, which allows sweeps to start at the same phase of a periodic signal, resulting in a steady plot, or on I^{2}C start condition. If enabled, samples are tested and discarded in the sketch until the chosen condition is met; it is tested in ISRs if these are used. Supported trigger modes are **Rising edge on A0**, which detects in software when a rising signal on input A0 goes higher than a specified voltage level, **I2C Start on D2, D3**, which detects an I^{2}C start condition (falling egde on SDA while SCL is HIGH) in software, using input D2 as SDA and D3 as SCL, **Condition on D2 or D3**, which uses the External Interrupt functionality to detect a change, falling or rising edge in hardware for D2 or D3, **Condition on D2**, and **Condition on D3**.

4.1.5. Wave Generator

The system can generate and output periodic signals (such as sine, square, triangular) or receive samples for user-supplied shapes from the App, using an external DAC, such as an R-2R resistor ladder connected to up-to 8 digital outputs. The sketch generates samples for one period of the shape, chosen on the App. One period can only use up to 128 samples due to SRAM size.

It works in the *Free Running ADC* mode. One Wave Generator iteration runs on each ADC ISR, at a known rate. A divisor, div, lowers the output sample rate to generate lower frequencies without more samples. Samples are output at SR_ADC/div Sps, independently of input count; SR_ADC is the ADC sample rate, which depends on ADC prescaler. The App computes div and number of samples for one period based on prescaler and desired output frequency.

It can, in theory, generate signals from 0.29 Hz ($div = 255$; 128 samples; prescale 128) to 307 kHz ($div = 1$; 2 samples; prescale 2). Output frequencies from 4.70 Hz to 9.61 kHz were tested during system evaluation and shown to be accurately generated.

4.2. Communication

The sketch and the App communicate over the USART (serial port on the PC) using 1 start bit, 8 data bits (1 byte), 1 even parity bit and 1 stop bit, resulting in 11 bits per frame. So, at 115200 baud it can transmit at most $115200/11 = 10.47k$ (8-bit) Samples per second (Sps), excluding protocol overhead. Hardware parity checks discard some frames with errors; not all are detected. More reliable error detection, such as Cyclic Redundancy Check (CRC), would require additional software, processing time, and transmission overhead.

When sampling multiple analog channels, losing just one byte in transmission of a sweep results in misaligned samples between channels. This is detected based on packet length, and the number of serial errors is shown to the user. The App detects some errors in packets with pin state updates or sketch parameters by checking that headers end with `\r\n`. These will be re-transmitted since all sweeps from there on are affected. The App may detect errors in sweep packets, but will try to process them; re-transmission of these would require samples to be stored on the Arduino, which is not always the case. If the App detects too many errors or timeouts, it will disconnect the serial port and alert the user to prevent it from continuing to acquire wrong data.

4.2.1. Protocol

When the App opens the serial port, the sketch runs from the start. The USART always starts at 115200 baud and an handshake is performed; the sketch sends a packet indicating it is ready and the App replies with the sampling mode and baud to use. The sketch acknowledges this with a *mode* packet, completing the handshake, and sampling starts. These and other parameters can be changed after the handshake. Higher baud allows transmitting more samples, but may increase errors, leading to some samples being lost.

Data is sent from the sketch to the App in *packets*. A packet starts with $BLOCK\_START$, 3 bytes, `=:%` in ASCII, followed by 2-byte packet *type* (2 characters). Depending on the type, a *header* may follow, which ends in `\r\n`. Some packets of some types have a binary *payload* with samples after the header.

The App sends commands to the sketch to do actions like changing output pin values, sketch parameters (such as sampling mode and number of input channels), or requesting state. A command is a string with space-separated fields and ends with a line-end. The first field is a number for the action. The following fields depend it. Some commands include binary data, such as samples for a

custom wave shape. The App sends data in blocks of up to 16 bytes with a small delay for each block to avoid filling the receive buffer, causing errors.

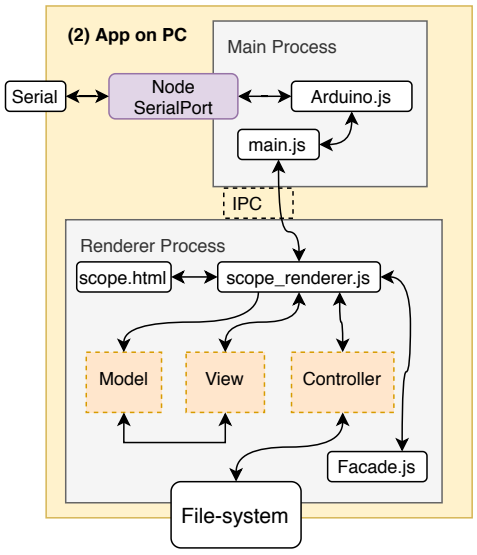
4.3. App

The App uses the Electron framework. The main process runs JavaScript code and manages the creation of renderer processes, which show (local) web pages that use JavaScript, HTML5 and CSS. Processes communicate over Inter-Process Communication (IPC) using Electron Application Programming Interfaces (APIs).

The main process communicates with the sketch and creates a renderer process. The App asks the user to choose a serial port, but a session file can be loaded instead. Then, the *scope.html* page is shown, allowing the user to view and change sketch parameters. The user can view *live* samples as these arrive from the sketch, or view samples that were previously stored, either from the current session or from a session file. Stored sweeps can be viewed with the same timing as captured. Sweeps are plotted on a canvas with 10 divisions in the X axis (time). The user can specify a time base (duration) for future sweeps, in milliseconds per division, ms/div; the App will compute the required amount of samples for the current sample rate.

App code is split in files; code for the renderer is organized resembling the Model–View–Controller (MVC) pattern. Some of the files and interactions between parts are shown in Figure 3.

Figure 3: App architecture overview and code interactions; not all files are shown. *Node SerialPort* is an external library. *Serial* refers to the serial port to which an Arduino running the sketch is connected. *File-system* refers to Electron File-system APIs



main.js *main.js* runs on the main process. It is the entry point for the App and communicates with Arduino.js and the renderer process. When the user chooses a serial port, the page will tell the main process which port was chosen and *main.js* will tell *Arduino.js*. *Arduino.js* confirms that a correct port was chosen, or, if the user chooses to not open a port, *main.js* tells the renderer to load the *scope.html* page and ask for a file.

Arduino.js *Arduino.js* receives *sweeps* from the sketch, processes, adds timestamps and sends these to the page, which displays and optionally stores them. Transmissions from the sketch are parsed in *Arduino.js*. It uses a Delimiter Parser (from Node SerialPort) to split data into packets. *Facade.js* provides functions for the page to control the sketch; these functions generate data, sent to *Arduino.js*. These two files abstract the sketch, communication, and low-level parameters.

scope_renderer.js The *scope_renderer.js* script, loaded by *scope.html*, handles GUI events, instantiates classes and registers callbacks. If live input is active, *scope_renderer.js* sends received sweeps to *BufferManager.js*, which takes care of demultiplexing samples into an instance of *ChannelBuffer* for each channel. Sweeps are also sent to *SweepController.js*, which, if the user enables that option, stores them to be viewed later. These are stored in memory and can optionally be saved to a session file in JSON to be loaded later. Saving and loading *sessions* — sets of sweeps and attributes such as names given to channels — is taken care of in *ExportController.js*. Most of the classes are instantiated in *scope_renderer.js*. The user can choose to export a PNG image showing what is drawn on the canvas; in this case, *scope_renderer.js* will ask the canvas to generate a PNG image and use *SessionDialogs* to prompt where to save it. *SweepStorageView* allows the user to select a stored sweep to be viewed with original timing; the sweep will be feed to the *BufferManager*.

ChannelUI (view) manages a list of input channels that allows choosing color, name and other options for each input, such as if the channel is drawn over the previous one, whether to fill the background on HIGH values and whether to show FFT using *FFTView*. Options are stored by *SessionAttributes*.

OutputUI (view) manages the GUI for the state of digital pins. The user can choose, for each pin, whether it is an input (with high impedance or pull-up) or an output (LOW or HIGH).

Sweeps are plotted on an HTML5 Canvas. An instance of *MyCanvas* draws a grid. The mouse and scroll wheel allow pan and zoom around waveforms. The instance of *MyCanvas* has instances of

the *Division* class, each of which draws itself depending on its purpose. Subclasses of *Division* can be a waveform, bar graph showing FFT, an XY plot or show I²C data. Additional subclasses could easily extend functionality. Existing ones include *ChannelGraph*, *XYGraph*, *FFTView* and *DecoderView*. Each instance of *ChannelGraph* plots the sweep for one of the channels, data for which is stored in an associated instance of *ChannelBuffer*. The *DecoderView* class draws a representation of I²C transitions that are decoded.

I²C decoding in *I2CDecoder* uses samples from two *ChannelBuffer* instances: one for SDA (data channel) and another for SCL (clock). On each sweep, *I2CDecoder* uses an *EdgeDetector* to detect rising edges on samples for SCL. Start and stop conditions are also detected. When a rising edge is detected, a data channel bit is read at the same offset in the sweep. Instances of *Label* are created for each bit, frame, data direction bit, and acknowledge bit. An instance of *Packet* contains all *Label* instances for a single I²C transition, destination address and data direction. The *Packet* class is not to be confused with I²C packets, referred to as frames. An instance of the *DecoderView* draws a representation of all instances of *Label* for a sweep on the canvas, aligned in time with the corresponding sample; each address uses a different color; read and write transactions are represented in different lines.

5. Evaluation

The system was evaluated in two perspectives. First, the technical characteristics were determined. Then, since the system is to be used to aid MEIC students, it is important to assess if it can perform basic tasks. For that, the system was tested with exercises the can be used in laboratory classes and only use material from the Arduino Starter Kit to demonstrate some functionality. Two MEIC students used the system and performed the exercises to verify results and provide feedback. The exercises and the system's GUI where improved with their feedback.

5.1. Technical Characteristics

Each sampling mode was tested to determine sampling limits, namely input frequency range and sampling accuracy. The Wave Generator was also tested.

5.1.1. Methodology

Except where noted, tests use 115200 baud to decrease the likelihood of transmission errors. As seen in some tests, high sample rate or baud rate might make the USART unreliable.

Sampling Duration In the Loop (analog) mode, and when not filling a buffer in Loop (digital) mode, sample rate depends on USART baud and possibly other factors, as is detailed for each mode.

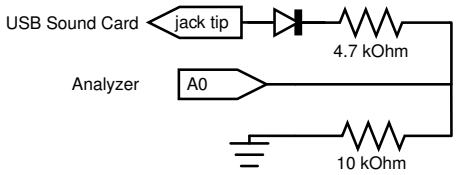
Sample Count and Frequency Range To sample a signal of f Hz, at least one period must be sampled. At a sample rate of SR Hz, at least SR/f samples are needed. The maximum number of samples in a sweep, $samples\_max$, limits the minimum frequency that can be sampled: $f_{min} = SR/samples\_max$. The maximum frequency that can be sampled is less than the Nyquist frequency, which is half the sample rate.

For each mode, various combinations of baud and sample rate were tested to determine the input frequency range. The total number of captured samples is $samples = requested - missing$, where $missing$ is the number of samples missing from the sweep, shown in the App, and $requested$ is the total number of samples requested for each sweep.

Sampling Accuracy Sine waves were generated with *ffplay* [16] *3.4.8-0ubuntu0.2* and a USB Sound Card at a sample rate of 44.1 kHz, to not depend on the same system.

The Card was connected to the studied input channel using a voltage divider and a diode, shown in Figure 4. The Card and Arduino have a common ground through the computer.

Figure 4: Sound Card output connected to A0



A single frequency was generated at a time and compared to the measured based frequency (frequency with largest amplitude on FFT, in the App, ignoring frequency 0: DC offset).

The error is $error = \frac{|measured - generated|}{generated}$, where $measured$ is the measured base frequency in Hz and $generated$ is the generated sine wave frequency in Hz. Error values are averaged to obtain the average error in frequency for a given combination of parameters.

Channel Crosstalk or Sample Misalignment There can be crosstalk between analog channels; losing just one sample in transmission, or if sampling ISRs take too long, will result in swapped samples between channels. To test this, a sine

[16] https://ffmpeg.org/, last accessed on 18th August 2020

wave was generated to input A0, while input A1 was fixed 3.3 V; sampled signals where then visually compared.

5.1.2. Results

In general, high sample rates or baud increase the likelihood of transmission errors.

Results for each mode are summarized below.

Free Running ADC Sample rate depends on the prescale value and can be continuous depending on baud. These combinations are continuous (1 analog channel): prescale 128 (9,615.38 Sps) at 115200 baud, prescale 64 (19,230.77 Sps) at 230400 baud, and prescale 32 (38,461.54 Sps) at 500000 baud. Sampling is, based on average error in frequency, accurate with prescale 128, 64, and 32, even sampling multiple channels. With prescale 16, results are acceptable, giving a maximum sample rate of 76,923.08 Sps, unless the Wave Generator, digital sampling or multiple analog channels are used. Prescale 8, 4, and 2 should not be used in this mode.

Timer (digital) This mode can sample 6 digital channels continuously at 50 kHz, using 1 M baud, though this will result in errors in commands received by the sketch. Sampling is accurate up to 128 kSps; higher sample rates are unreliable.

Loop (analog) With *fill*, sample rate depends on the ADC prescale and was observed up to 137,833.32 Sps (prescale 2) with accurate results for a single channel; it is however limited to 1024 samples per sweep. With *no fill*, the sample rate depends on baud in addition to prescale; it can go up to 78,048.78 Sps with prescale 4 and 4 M baud, but results in some transmission errors. Accuracy was tested up to 45,454.12 Sps (prescale 2; 500000 baud), which produces accurate results. However, when sampling multiple channels, prescale 4 and 2 are not reliable.

Loop (digital) When filling a buffer using sampling routines, sampling 6 digital channels shows to be accurate up to a sample rate of 5.3 MHz; it is however limited to 1024 samples per sweep. When not filling a buffer, sample rate ranges from 1.75 kSps at 19200 baud to 90.91 kSps at 1 M baud; however, since at 1 M baud many samples are lost during transmission, baud should not go above 500000 baud, which results in 45.64 kSps.

Summary Test results and some of the technical characteristics are summarized in Table 1 for each sampling mode. Based on these, *Loop (analog)* with *no fill* is not useful since its sample rate is only slightly higher than in *Free Running ADC*, at the cost of varying the sample rate by changing baud, as is *Loop (digital)* with *no fill* when compared to *Timer (digital)*. These should be hidden to simplify the interface. Settings which do not produce accurate results should also not be allowed in the interface, such as prescale 8, 4, and 2 while in *Free Running ADC* mode.

The Wave Generator was tested to output frequencies from 4.70 Hz to 9.61 kHz.

5.2. Laboratory Exercises

The *Arduino-analyzer* sketch must be uploaded to an Arduino — referred as *analyzer*. It will measure voltage levels relative to its ground, so a common ground is needed. Arduino boards connected to the same computer have a common ground through the USB port. Most exercises use the *Free Running ADC* mode and default settings.

5.2.1. Visualizing AC noise

A loose wire on A0 picks up Alternating Current (AC) noise, which appears on the sampled signal with an amplitude dependent on the proximity to a nearby power line. **Warning:** do not touch the power line or insert things into the electrical socket. Results may vary depending on the electric noise present in the surrounding environment.

5.2.2. Reading the angle on a potentiometer

A 10 kOhm potentiometer produces different voltages in a voltage divider circuit. Voltage is read using one analog channel on the analyzer. Varying the angle on the potentiometer makes measured voltage vary between 0 V and 5 V.

5.2.3. Visualizing a PWM signal that is dimming an LED

The "Fade" sketch [17] uses a 488 Hz PWM signal to dim an LED. The user can see the LED brightness changing depending on the PWM duty cycle and its average voltage.

5.2.4. Visualizing more than one analog channel

Adding a Light Dependent Resistor (LDR) and a resistor to the previous exercise, the user can see on A1 a signal proportional to LED brightness.

5.2.5. Decoding I²C Communication

Using the *master_writer* and *slave_receiver* sketches [18], two Arduino boards send data over I²C bus with an I²C clock of 100 kHz; a sample rate higher than 200 kHz is needed. With a sample rate of 500 kHz in the *Loop (digital)* mode with fill, the user can view this data on the I²C bus.

[17] https://www.arduino.cc/en/tutorial/fade, last accessed on 16th August 2020

[18] https://www.arduino.cc/en/Tutorial/MasterWriter, last accessed on 22nd August 2020

Table 1: Summary of technical characteristics for each sampling mode. *Continuous* refers to the whether the mode can sample continuously without losing samples between sweeps, if the USART can transmit fast enough

Mode	Buffer	# Channels		Samplerate		
		Analog	Digital	Sps	Predictable	Continuous
Free Running ADC	FIFO	0 - 6	0 - 8	9.6k - 76.9k	Yes	Yes
Timer (digital)	FIFO	-	1 - 6	0.24 - 128k	Yes	Yes
Loop (analog)	No fill	1 - 6	-	10k - 78k	No	No
	Fill			8.92k - 137k	No	No
Loop (digital)	No fill	-	1 - 6	10.69k - 45.64k	No	No
	Fill; routines			15.48 - 5.3M	Yes	No

6. Conclusion

The developed system [19] allows using an Arduino Uno and a PC as an oscilloscope, logic analyzer and signal generator. It can sample up to 6 analog and up to 8 digital channels. Sampled signals can be exported to CSV or PNG and stored in session files for later analysis without an Arduino. Using an external DAC, such an R-2R DAC, signals with various shapes can be generated. Logic analyzer functionality decodes and displays I^2C communication. Evaluation shows that, in the best case, the system can sample 1 analog channel at 137 kSps or 6 digital channels at a sample rate of 5.33 MHz, with some limitations; also, the system can continuously sample 1 analog channel at 76.9 kSps or 6 digital channels at 50 kHz.

Laboratory exercises were described to demonstrate some of the system functionality; both were improved based on user feedback.

The system can be used in some MEIC classes to help students understand signals and digital communication with existing Arduino Starter Kits instead of expensive equipment. Students can also use it to debug their projects, even at home.

6.1. Future Work

Although it works as intended, more features could be added such as continuously streaming samples from the App to the sketch, support for trigger pre-capture, decoding more protocols, and plotting math operations for channels.

References

[1] D. Ashok Kadge and D. Nandgaonkar, "Portable Oscilloscope using Arduino and GLCD", *International Journal of Engineering Technology Science and Research (IJETSR)*, vol. 4, no. 6, pp. 797–801, 2017, ISSN: 2394-3386.

[2] A. R. Cunha, "Software for Embedded Systems Laboratory Guide", Instituto Superior Técnico, Lisbon, November 2017.

[3] J. Sarik and I. Kymissis, "Lab kits using the arduino prototyping platform", *Proceedings - Frontiers in Education Conference, FIE*, pp. 1–5, 2010, ISSN: 15394565. DOI: 10.1109/FIE.2010.5673417.

19 https://web.tecnico.ulisboa.pt/ist177944/Arduino-Analyzer/, last accessed on 19th October 2020

[4] W. L. Tan, S. Venema, and R. Gonzalez, *Using Arduino to Teach Programming to First-Year Computer Science Students*, 2017.

[5] W. H. Kuan, C. H. Tseng, S. Chen, and C. C. Wong, "Development of a Computer-Assisted Instrumentation Curriculum for Physics Students: Using LabVIEW and Arduino Platform", *Journal of Science Education and Technology*, vol. 25, no. 3, pp. 427–438, 2016, ISSN: 15731839. DOI: 10.1007/s10956-016-9603-y.

[6] P. Jamieson, "Arduino for teaching embedded systems. are computer scientists and engineering educators missing the boat?", *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)*, pp. 289–294, 2011.

[7] A. Vostrukhin and E. Vakhtina, "Studying digital signal processing on arduino based platform", *Engineering for Rural Development*, pp. 236–241, 2016, ISSN: 1691-5976.

[8] J. P. Grinias, J. T. Whitfield, E. D. Guetschow, and R. T. Kennedy, "An inexpensive, open-source USB Arduino data acquisition device for chemical instrumentation", *Journal of Chemical Education*, vol. 93, no. 7, pp. 1316–1319, 2016, ISSN: 19381328. DOI: 10.1021/acs.jchemed.6b00262.

[9] I. J. Koenka, J. Sáiz, and P. C. Hauser, "Instrumentino: An open-source modular Python framework for controlling Arduino based experimental instruments", *Computer Physics Communications*, vol. 185, no. 10, pp. 2724–2729, 2014, ISSN: 00104655. DOI: 10.1016/j.cpc.2014.06.007.

[10] D. Calinoiu, R. Ionel, M. Lascu, and A. Cioabla, "Arduino and LabVIEW in educational remote monitoring applications", in *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, 2014, pp. 1–5, ISBN: 9781479939220. DOI: 10.1109/FIE.2014.7044027.

[11] A. D'Ausilio, "Arduino: A low-cost multipurpose lab equipment", *Behavior Research Methods*, vol. 44, no. 2, pp. 305–313, 2012, ISSN: 1554-3528. DOI: 10.3758/s13428-011-0163-z.

[12] N. S. A. Pereira, "Erratum: Measuring the RC time constant with Arduino (2016 Phys. Educ. 51 065007)", *Physics Education*, vol. 52, no. 1, p. 019501, 2017, ISSN: 0031-9120. DOI: 10.1088/1361-6552/52/1/019501.

[13] G Prashanthi, K; Swetha Reddy, N; Kavya, N; Alekhya, "Oscilloscope / Logic Analyzer using Arduino", Gokaraju Rangaraju Institute of Engineering and Technology, Hyderabad, Andhra Pradesh, Tech. Rep., 2013.

[14] D. Ba, C Karthick, A Lohith, and A. Ayub, "Smart Electronic Kit", *Imperial Journal of Interdisciplinary Research (IJIR)*, vol. 3, no. 1, pp. 748–752, 2017, ISSN: 2454-1362.

[15] D. Hercog and B. Gergič, "A flexible microcontroller-based data acquisition device.", *Sensors (Basel, Switzerland)*, vol. 14, no. 6, pp. 9755–9775, 2014, ISSN: 1424-8220. DOI: 10.3390/s140609755.

[16] L. Gomes, "Desenvolvimento de um analisador lógico simples", *Revista do DETUA*, vol. 4, no. 5, 2005.

[17] R. Katona and D. Fodor, "Texas instruments MSP430 microcontroller based portable multi-purpose instrument for android platforms", *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, pp. 1–5, 2014. DOI: 10.1109/EDERC.2014.6924347.