SVP 2022/23 Project report

Author: Jan Fiala, supervisor: doc. Ing. Jan Fischer, CSc

# USB oscilloscope with a web-based user interface

## Project goal and motivation

The goal of this project is to create a software defined oscilloscope using the Raspberry Pi Pico and a web-based user interface, with the WebUSB API acting as the interface between the two parts. A system overview of the proposed oscilloscope design is shown in image 1.

There are currently multiple software-defined instruments (oscilloscopes, function generators, logic analysers) created by students at the Department of Measurement, namely the LEO (Little Embedded Oscilloscope). The motivation behind creating a new oscilloscope with a web-based GUI is greater accessibility for its users: the web-based oscilloscope is multiplatform (it runs on Windows, Mac or Linux) and does not need to be installed.

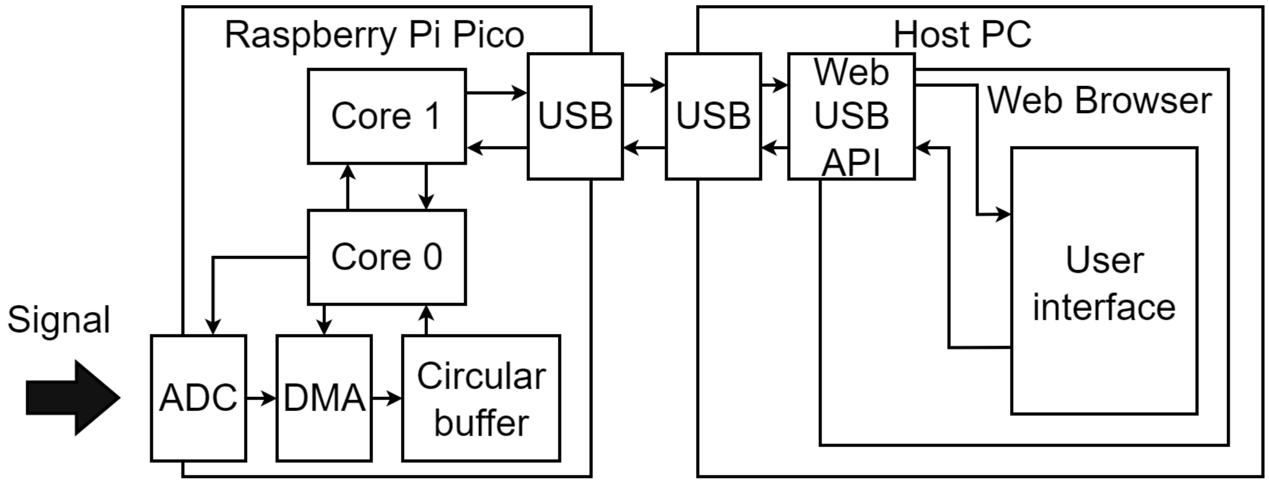


Image 1: A system overview of the oscilloscope.

# The WebUSB API

## Introduction

A crucial (and arguably the most novel) part of the proposed oscilloscope design is the communication between the hardware and the GUI. The WebUSB API serves exactly this purpose - communication between an USB device and software running in the browser.

Before describing the WebUSB API in more detail, I will cover a few terms and concepts relating to the USB interface itself. To accomodate devices with diverse communication requirements, USB is highly configurable. An USB device can use multiple (up to 32, althought  it is rare to use so many) virtual communication pipes, also called endpoints. There are three different types of endpoints: bulk, suitable for transferring large amounts of data, isochronous, which guarantees low latency, and control, mainly used for device configuration. When an USB device is connected to a host system, it sends the host a description of the required endpoints (this process is called enumeration), which the host system will then provide to the device.

On the host side of an USB connection is a device driver, which communicates with the device through endpoints exposed by the USB stack. Usually, an USB driver is written in C or C++ and runs inside the host operating system kernel. The WebUSB API exposes the USB device endpoints directly to the JavaScript runtime inside a web browser, so the device driver can be implemented directly using JavaScript.

Currently, WebUSB is in experimental phase and it is not a standard web API. Only web browsers based on the Chrome kernel (Chrome, Chromium and Microsoft Edge) support it at this time. There are officially no plans in the other two major browsers (Firefox and Safari) to implement this API, citing security concerns as the reason.

While these concerns are certainly valid, the way in which WebUSB exposes USB devices to the browser is more secure than it might sound. First, the user must explicitly select an USB device in a pop-up window for the web browser to access it. Second, it is not possible to access devices belonging to common USB classes such as HID (keyboards, mouses), Mass storage, Network interfaces and others. WebUSB can only access devices outside these classes, which are usually only a very specialised hardware.

## An example

The following JavaScript code snippet requests an USB device with the *vendor ID* equal to `0xCAFE`. The `requestDevice` function does not select an USB device by itself: when it is called, a pop-up window appears in the browser, prompting the user themselves to select a particular USB device. The `filters` object passed to the function can be used to filter the USB devices listed to the user. The device *vendor ID*, *product ID* or serial number can be set to only allow a particular device(s) to be listed. After a device is selected, a device *configuration* and *interface* are chosen. It is rare for an USB device to have multiple interfaces and even rarer to have multiple configurations. In this example, the first configuration and interface is selected.

```
let device = await navigator.usb.requestDevice({
   filters: [{ vendorId: 0xcafe }]
});
   await device.open();
   await device.selectConfiguration(1);
   await device.claimInterface(1);
```

After connecting, we can start communicating with the device. The `transferOut` function parameters are the number of the endpoint (3 in this case) to which a message will be sent to, and the message itself. The message must be an `ArrayBuffer` object, such as an `Uint8Array`, which is basically an array of bytes.

```
let message = new Uint8Array([1, 2, 3, 4]);
device.transferOut(3, message);
```

Similarly, the `transferIn` function can be used to receive data from a particular endpoint. The second parameter of the function sets the maximum number of bytes to be received. The function returns a `DataView` object, which can be parsed to an `Uint8Array` or other array types.

```
let result = await device.transferIn(3, 4);
let bytes = result.data.getUint8();
```

## Practical insights of using the WebUSB API

Because WebUSB is not widely adopted, there are only a few resources available to help with development. The first is an article called *Access USB Devices on the Web* [1], which contains a simple example of using WebUSB and also considers platform-specific issues with using the API, which will be described later. The second resource is the official draft [2] of the WebUSB API, which describes components of the API in more detail.

These resources only include very basic examples, so a large amount of experimentation was required to use the API in practice. These struggles were heightened by my limited knowledge of JavaScript and the browser runtime. JavaScript can only run in a single thread inside a browser window tab, and this thread also handles re-rendering of the web page. This means that any blocking JavaScript code will cause the whole web page to stop responding.

## Special USB descriptor on Windows

For the Windows operating system to allow the WebUSP API to communicate with an USB device, the device needs to include a special USB descriptor during its enumeration. This descriptor, called the *Microsoft OS 2.0 platform capability descriptor*, is fortunately included in the TinyUSB [3] open source USB stack, which was used for the oscilloscope firmware development.

## Udev rules on linux

Linux does not require a special USB descriptor, but it disables user space applications from accessing USB devices as a default behaviour. To enable access to an USB device, an *udev rule* file needs to be added to the `/etc/udev/rules.d/` folder. This file needs to have the `.rules` extension and contain the following text:

```
SUBSYSTEM=="usb", ATTR{idVendor}=="XXXX", MODE="0664", GROUP="plugdev"
```

The characters `XXXX` are to be replaced with the USB vendor ID number of the device (this number is a part of the device USB descriptor and can be configured in firmware).

## Debugging tools

Few tools proved to be useful for troubleshooting during the initial phase of this project. The first is an utility inside the Chrome web browser, called *USB Internals*. It can be accessed by typing `chrome://usb-internals/` into the search bar and lists all USB devices connected to the system. It also allows creating a virtual USB device (this device is only visible from inside the web browser, so it only has a limited utility, such as troubleshooting a WebUSB connection). Another utility, called Web USB tester [4], can be found on GitHub. It is a simple web page which can be used to test connection to an USB device with a specific USB *vendor ID* and *product ID*. Finally, the well-known WireShark packet analyzer can be used to capture and inspect “raw” USB communication on Linux, however this requires changing a few settings of the system. The required steps are described in a StackOverflow discussion [5].

## Hardware -The Raspberry Pi Pico

The Raspberry Pi Pico was selected as the hardware part of the oscilloscope. The main reasons were its availability, low price and the presence of an USB peripheral.

A serious drawback of the Pico (or more specifically, the RP2040 microcontroller) is its analog-to-digital converter (ADC): the maximum conversion speed is only 500 kS/s. This speed is the total maximum for all active analog channels, so when two or three channels are active, the speed is only 250 kS/s or 166 kS/s per channel, respectively. However, it was decided that the availability and low price of the Pico offset the underwhelming ADC performance and the sample rate is fast enough for the intended purpose of the oscilloscope.

A strong point of the Pico is the 264 kB SRAM memory. Considering a 200 kB buffer, up to 400ms of samples can be captured at the highest sample rate. Another advantage is the dual-core architecture, with two ARM Cortex-M0+ cores capable of clock speed up to 133 MHz. For the purpose of an oscilloscope, one core can be used for strict real-time tasks such as checking the analog samples for a trigger condition or stopping the signal capture, while the other core might handle the USB stack (this will be described in detail in the Firmware section).

The use of DMA (direct memory access) is necessary to maximally utilize given hardware resources. A digital oscilloscope saves the signal samples into a circular buffer, but unlike other microcontrollers such as the STMicroelectronics STM32 series, the DMA peripheral of the Pico does not have a circular mode. However, the Pico allows chaining DMA channels, which means that a finished memory transfer of one DMA channel can trigger the start of another channel. This feature can be used to implement a circular DMA transfer, as presented in the diploma thesis [5] of Ing. Vit Vanecek.

## Firmware

### The SDK

For the firmware development, the official *Raspberry Pi Pico SDK* [7] was used. The SDK contains illustrative examples of using the peripherals of the Pico and the documentation of the SDK was found to be very helpful. The part of the SDK I found to be lacking is the USB stack. The Pico SDK uses a port of the TinyUSB USB stack, but it only implements a few basic USB classes. The WebUSB device class, which is present in TinyUSB, is missing in the Pico SDK port. However, the only special requirement of a WebUSB device is an additional *Platform capability descriptor*, which I was able to add from the main TinyUSB branch.

## Dual-core architecture

Of the two cores of the Raspberry Pi Pico, one of them (further called the communication core) was dedicated solely to running the TinyUSB USB stack, while the other (further called the application core) does all other tasks.

When the application core wants to send data to the host PC, it calls the `write_USB` function, which copies the data into a buffer shared with the communication core and sets a shared variable to the number of bytes to be sent. The communication core is running in a super-loop, in which it (among other tasks) checks the shared variable. If the variable is found to be non-zero, the communication core then handles copying data to the USB peripheral internal buffer. From the peripheral buffer, the data is then sent over the USB bus by the peripheral itself. After the data is sent, the communication core resets the shared variable to zero, and the `write_USB` function returns (this means it is a blocking function).

When the application core wants to receive data from the host PC, it calls the `read_USB` function, with a buffer to copy the received data into as a parameter. When the communication core receives data from the USB peripheral, it sets a shared variable (which is initially set to zero) to the number of received bytes. The `read_USB` function waits in a busy loop until the variable is non-zero and then copies the received data into the buffer passed to it as a parameter. The `read_USB` function is blocking, so a non-blocking version, called `read_USB_NB` was implemented. This function returns immediately if there is no received data available and is used during signal capture to check if an abort message (a command to stop the capture) was received.

## Signal capture

The signal capture logic runs in a super-loop inside the application core. First, the Pico waits until it receives a configuration message from the host PC. The configuration message contains parameters such as the active channels, sample rate, capture depth and trigger conditions. The PWM generator settings are also included in the message. After the message is received, the DMA, ADC and PWM peripherals are configured accordingly and started. While the DMA is filling a buffer with new ADC samples, the application core is running in a loop, checking if a trigger condition occurred in any of the samples, or if an abort message was received. If a trigger condition is found, the Pico enters another loop, waiting until a configured number of samples is captured and stopping the ADC and DMA. After that, the captured samples are sent to the host PC and the Pico waits for another configuration message before starting the capture again. If an abort message is received during the capture, the Pico stops the capture immediately and waits until a configuration message is received again. A flow diagram of the signal capture logic is shown on image 2. below.

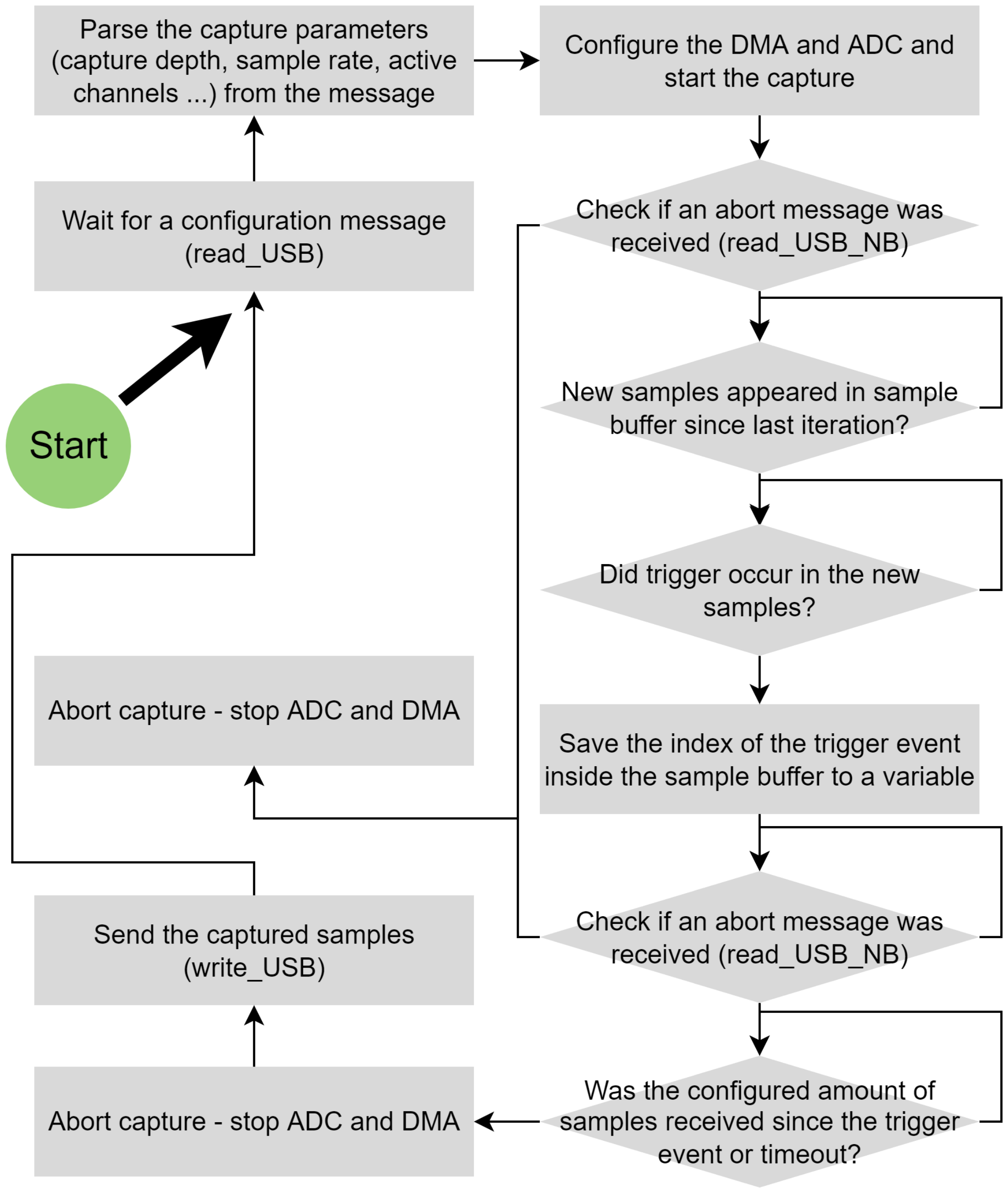

Parse the capture parameters (capture depth, sample rate, active channels ...) from the message
Configure the DMA and ADC and start the capture
Wait for a configuration message (read_USB)
Check if an abort message was received (read_USB_NB)
Start
New samples appeared in sample buffer since last iteration?
Did trigger occur in the new samples?
Abort capture - stop ADC and DMA
Save the index of the trigger event inside the sample buffer to a variable
Send the captured samples (write_USB)
Check if an abort message was received (read_USB_NB)
Abort capture - stop ADC and DMA
Was the configured amount of samples received since the trigger event or timeout?

Image 2.: Flow diagram of the singal capture logic.

## PWM generator

While the Pico does not have an ADC peripheral, it does have a PWM (pulse-width modulation) generator, which can be used to generate a clock signal with configurable frequency and duty cycle. This is a very practical feature for an oscilloscope to have, so it was added to the design. The generator can be enabled and its frequency and duty cycle can be set from the user interface.

# User Interface

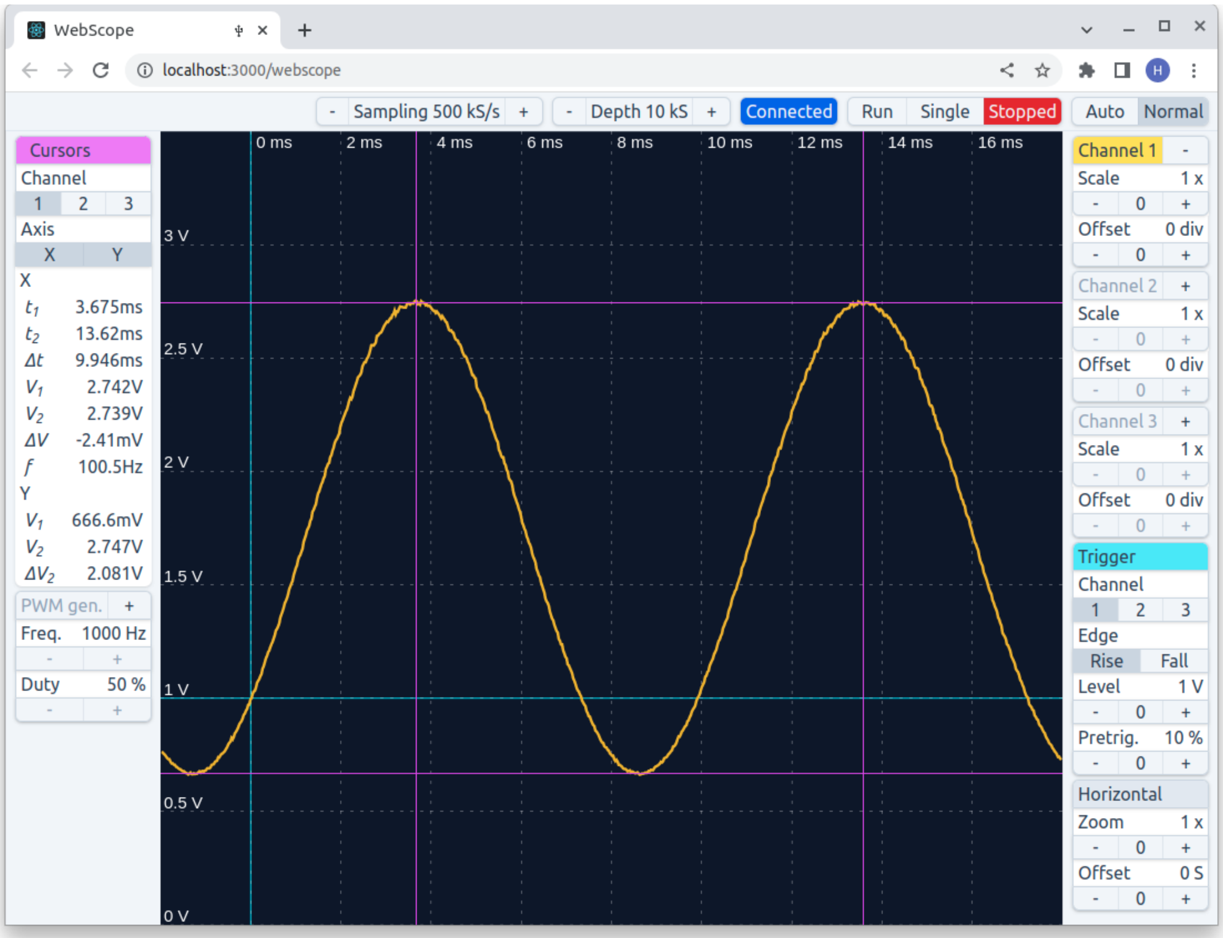
The graphical user interface of the oscilloscope is essentially a single-page web page. This kind of web page is often called the SPA (single page application). The GUI was not implemented using plain JavaScript, HTML and CSS - because the interface needs to be highly interactive, the React framework was used for its development. React is a JavaScript framework originated at Facebook and created precisely for the purpose of creating interactive SPAs. For styling of the interface, the TailWind CSS framework was used instead of raw CSS.

Communication with the hardware using the WebUSB API was covered in the WebUSB chapter. The other critical part of the GUI is the plotting of the signals. Multiple options were considered for data plotting: The first one was using an open source library called webgl-plot [8] which uses WebGL for high performance. This library was found difficult to integrate into the rest of the interface (in part due to using the React framework). The final solution is to use the *canvas* HTML element, which allows drawing on an area of a web page as if it was a bitmap through a JavaScript API. The performance concerns of using this solution were found to be unnecessary - after all, the signal needs to be redrawn no more than five times a second when the oscilloscope is running in continuous capture mode.

An oscilloscope is not complete without cursors, which enable measurement of voltage at an exact point of the capture or time and voltage difference between two points of the signal. Cursors were first implemented by rendering them inside the same canvas HTML element used for plotting the signal and using a HTML slider element to control them. However, this method of controlling the cursors was found to be too imprecise, so another solution was needed. In the final implementation, the cursors are rendered on top of the signal capture as floating HTML div elements. The cursors can be positioned by dragging them using the mouse pointer, allowing for more precise control. This behaviour was implemented by adding *onMouseDown, onMouseUp* and *onMouseMove* event handlers to the cursor HTML elements and changing their position programmatically.

The final user interface is shown on image 3. The interface is divided into four parts:

- Top bar, which contains sample rate and capture depth settings, capture mode selection (Normal/Auto) and the capture start and stop buttons.
- Right sidebar, which contains controls for each analog channel (enable/disable button, horizontal scale and offset), trigger settings (trigger channel, trigger threshold voltage, pretrigger ratio and rising/falling edge selection) and horizontal scale and offset settings.
- Left sidebar, which contains the cursor controls (horizontal and vertical cursors enable/disable button, channel selection), cursor measurements and the PWM generator controls.
- The plot, showing a capture of a sine wave signal as an example. The vertical cursors (purple horizontal lines in the plot) are positioned to measure the signal amplitude, and the horizontal cursors (purple vertical lines) are positioned to measure its frequency. The horizontal and vertical blue lines in the plot indicate the trigger voltage threshold and the pretrigger ratio, respectively.



Sampling 500 kS/s
Depth 10 kS

Cursors
Channel: 1 2 3
Axis: X Y

| | |
|---|---|
| X | |
| $t_1$ | 3.675ms |
| $t_2$ | 13.62ms |
| $\Delta t$ | 9.946ms |
| $V_1$ | 2.742V |
| $V_2$ | 2.739V |
| $\Delta V$ | -2.41mV |
| $f$ | 100.5Hz |
| Y | |
| $V_1$ | 666.6mV |
| $V_2$ | 2.747V |
| $\Delta V_2$ | 2.081V |

PWM gen.
Freq. 1000 Hz
Duty 50 %

Channel 1
Scale 1 x
Offset 0 div

Channel 2
Scale 1 x
Offset 0 div

Channel 3
Scale 1 x
Offset 0 div

Trigger
Channel: 1 2 3
Edge: Rise Fall
Level 1 V
Pretrig. 10 %

Horizontal
Zoom 1 x
Offset 0 S

Image 3.: The oscilloscope user interface

## Conclusion

The goal of this project was to create an oscilloscope based on the Raspberry Pi Pico and a web-based user interface. The WebUSB API was used for communication between the Pico and the user interface. The first part of the project was devoted to research of the API and to implementing basic communication between the hardware and the user interface. After this was achieved, the full firmware and user interface for the oscilloscope were implemented. All major requirements for the oscilloscope were satisfied, proving that the combination of minimal external hardware and a web-based control interface is a viable way of creating simple measurement instruments. The main features of the oscilloscope are the following:

- Three analog channels 8-bit sampling resolution, 0V to 3.3V voltage range
- Maximum total sampling frequency of 500kS/s
- Maximum total capture depth of 200kS
- Auto and normal trigger modes, configurable trigger threshold and trigger channel
- Vertical and horizontal cursors
- PWM generator with configurable frequency and duty
- Ability to vertically scale and move individual channels inside the capture plot.

## Sources

1. Access USB Devices on the Web
   URL:*https://developer.chrome.com/articles/usb/*
2. WebUSB API Draft
   URL:*https://wicg.github.io/webusb/*
3. TinyUSB
   URL:*https://github.com/hathach/tinyusb*
4. WebUSB Tester
   URL:*https://larsgk.github.io/webusb-tester/*
5. Capturing USB packets using WireShark
   URL:*https://stackoverflow.com/questions/31054437/how-to-install-wireshark-on-linux-and-capture-usb-traffic*
6. Vít Vaněček: Logic analyser based on the Raspberry Pi Pico
   URL:*https://dspace.cvut.cz/bitstream/handle/10467/101451/F3-DP-2022-Vanecek-Vit-Logicky-analyzator-s-RPI-Pico.pdf*
7. Raspberry Pi Pico SDK
   URL:*https://github.com/raspberrypi/pico-sdk*
8. Webgl-plot
   URL:*https://github.com/danchitnis/webgl-plot*