Glucose biosensor based on open-source wireless microfluidic potentiostat

Conan Mercer,[a]* Richard Bennett,[a] Peter Ó Conghaile,[b] James F. Rusling[acde] and Dónal Leech[a]

[a]School of Chemistry, National University of Ireland Galway, University Road, Galway, Ireland
[b]National Centre for Sensor Research, School of Chemical Sciences, Dublin City University, Dublin 9, Ireland
[c]Department of Chemistry, University of Connecticut, Storrs, CT 06269, USA
[d]Institute of Materials Science, University of Connecticut, Storrs, CT 06269, USA
[e]Department of Surgery, and Neag Cancer Center, UConn Health, Farmington, CT 06032, USA

* Author to whom correspondence should be addressed,
c.mercer1@nuigalway.ie

1 Hardware

1.1 ESP-32 Module Pin Definitions

The ESP-32 includes the ESP32 chip, built-in USB port, a hardware reset button, Wi-Fi antenna, Bluetooth, LED lights, and standard-sized GPIO (General Purpose Input Output) pins that are breadboard compatible. Figure 1 shows the ESP-32 module pin definitions.

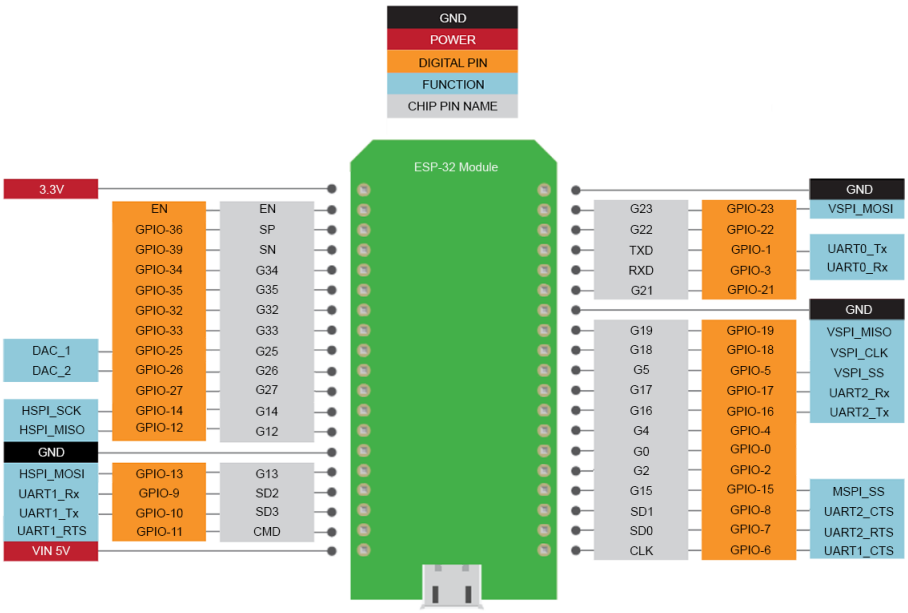


Figure 1: ESP-32 module pin definitions

1.2 ESP-32 Wireless Microcontroller Comparison

Specifications	ESP-32[1]	ESP-12E[2]	Arduino[3]	Teensy[4]	Raspberry Pi[5]	BeagleBone[6]	RFDUINO[7]
Core count	2	1	1	1	4	2	1
Architecture	32-Bit	32-Bit	8-Bit	32-Bit	64-Bit	32-Bit	32-Bit
CPU frequency	160 MHz	80 MHz	16 MHz	72 MHz	1.4 GHz	1 GHz	16 MHz
WiFi	Yes	Yes	No	No	Yes	No	No
Bluetooth	Yes	No	No	No	Yes	No	Yes
RAM	512KB	160KB	2KB	64KB	1GB	512MB	8KB
Flash memory	16MB	16MB	32KB	2MB	n/a	4GB	128KB
GPIO pins	34	17	14	24	40	64	19
ADC pins	18	1	6	2	0	1	1
DAC pins	2	0	0	1	0	0	0
Price (USD)	8.21	3.30	4.17	35.50	37.20	63.09	31.00
Part Number	32799954012	32779189539	32810825676	32776625006	32838484861	1894139445	32837897544

Table 1: Specification comparison of two ESP Wi-Fi enabled microcontroller variants, compared to the Arduino UNO, Teensy 3.0, Raspberry Pi 3 Model B+, BeagleBone Black and RFDUINO. The ESP32 reports both WiFi and bluetooth technology built in, has multiple core processors and is the most cost-effective, www.aliexpress.com, last accessed April 2018.

1.3 Pulse Width Modulation

Pulse width modulation (PWM) is used to control the direction of a servo valve. The width of the electrical pulse controls the angle of mechanical rotation. The pulse event occurs over a defined period of time, 50 Hz or every 20 ms [8]. The servo will stay in position for as long as the PWM is repeated. For example, if a 1 ms pulse of 5 V is applied to the servo, it will rotate to a position of 0°, conversely if the width is 2 ms a position of 180°, see Figure 2. In terms of precision, PWM is a digital signal that does not require analog conversion and is either 'on' or 'off', making it very precise compared to continuously variable analog signals. It is also resistant to noise because interference would need to be strong enough to change a logic 0 (LOW) to a logic 1 (HIGH) [1], or vice versa.

1.4 Library

It is good practice to implement libraries into code to save time and increase efficiency, in this context, efficient means that use of the library does not impose significant learning curves on the user compared with entirely hand written code. To control servomechanisms a 'servo' library is used. All of the library's used are stated in the header of the C program.

[1] Throughout the manuscript 5 V TTL Logic Levels are used. This is defined by the ESP-32 microcontoller. TTL is an acronym for Transistor-Transistor Logic. It relies on circuits built from bipolar transistors to achieve switching and maintain logic states. Transistors are essentially electrically controlled switches.

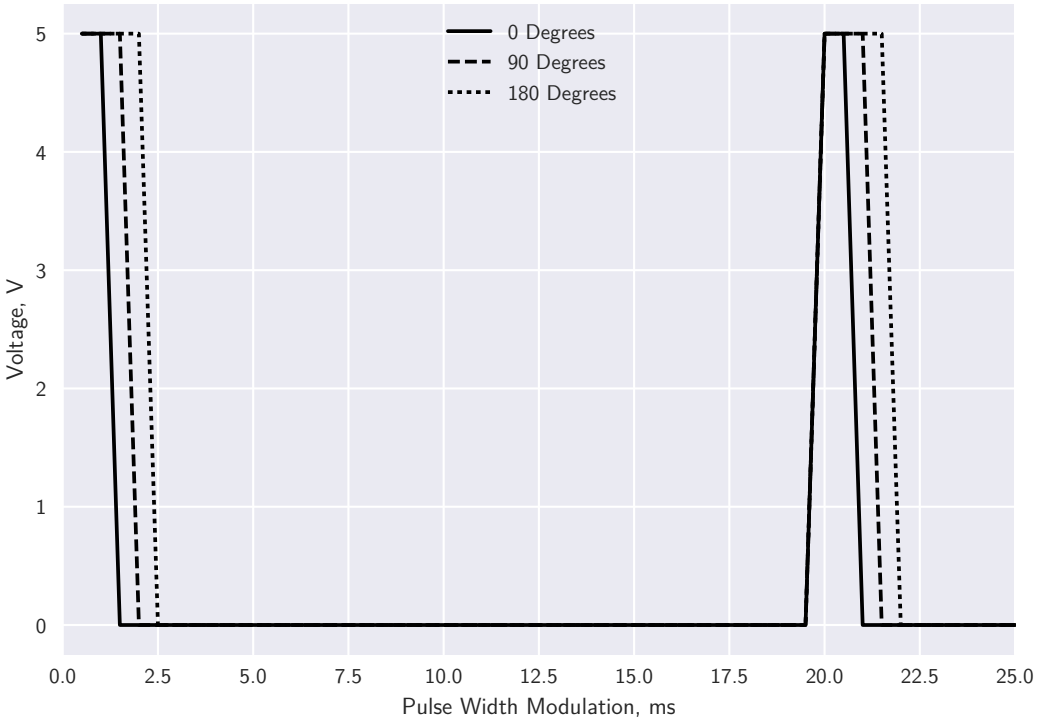


Figure 2: PWM graph for servo at frequency of 20 ms: 0° (——), 90° (- - -), 180° (·····).

1.5 Syringe Pump

Flow rate within the system is controlled by a syringe pump. Syringe pumps are commercially available and provide stable flow rates and greater precision over peristaltic pumps. Most syringe pumps have RS-232 connectivity, so the circuitry includes a MAX232 device for RS232 to TTL. Syringe pumps introduce slight fluctuations in flow rate and pressure in micro channels caused by a mechanical stepper motor which drives the syringe forward [9, 10]. Despite this, flow-rate control remains superior over pressure controlled methods when pumping in medical and microfluidic settings [11]. Communication with syringe pumps is established using serial protocols. Parity checks the state or condition of a received bit [2], baud rate is the unit determining how fast data is sent over serial line and is measured in bits-per-second. Start and stop bits marked the beginning and end of data packets [3]. The manuscript used an Aladdin syringe pump that operated at specific serial parameters, Table 2.

[2] A form of low-level error checking, it slows down communication but is not necessary and is usually turned off.

[3] While there is always one start bit the number of stops can be either one or two (commonly left at one).

Baud Rate	19200
Frame	10 bit data frame (8N1)
Start Bit	1
Data Bits	8
Stop Bits	1
Parity	None

Table 2: RS-232 Protocol for World Precision Instruments Syringe Pump Model: AL1000

1.5.1 Syringe Pump Serial Connection

The RJ-11 port is present on the rear of the Aladdin AL-1000 syringe pump which is connected to the MAX232 microchip in the detailed circuit diagram in section 3.

1.5.2 ASCII Table

ASCII stands for American Standard Code for Information Interchange. In order to control the syringe pump, the conversion Table 3 is used to obtain hexadecimal values converted from plain text. The simplified ASCII Table shows only the numbers 0 to 9, and lowercase character letters a to z.

Chr	Hex	Chr	Hex	Chr	Hex
0	30	c	63	o	6F
1	31	d	64	p	70
2	32	e	65	q	71
3	33	f	66	r	72
4	34	g	67	s	73
5	35	h	68	t	74
6	36	i	69	u	75
7	37	j	6A	v	76
8	38	k	6B	w	77
9	39	l	6C	x	78
a	61	m	6D	y	79
b	62	n	6E	z	7A

Table 3: ASCII table

1.6 Flushing Protocol

Step	Description	Flow Rate	Time
1	Flush detection chamber 1	$2000\,\mu l\,min^{-1}$	3min
2	Flush detection chamber 2	$2000\,\mu l\,min^{-1}$	3min
3	Flush detection chamber 3	$2000\,\mu l\,min^{-1}$	3min
4	Flush detection chamber 4	$2000\,\mu l\,min^{-1}$	3min

Table 4: Flushing protocol

2 Characterisation of $[Os(2,2'-bipyridine)_2(poly-vinylimidazole)_{10}Cl]^+$

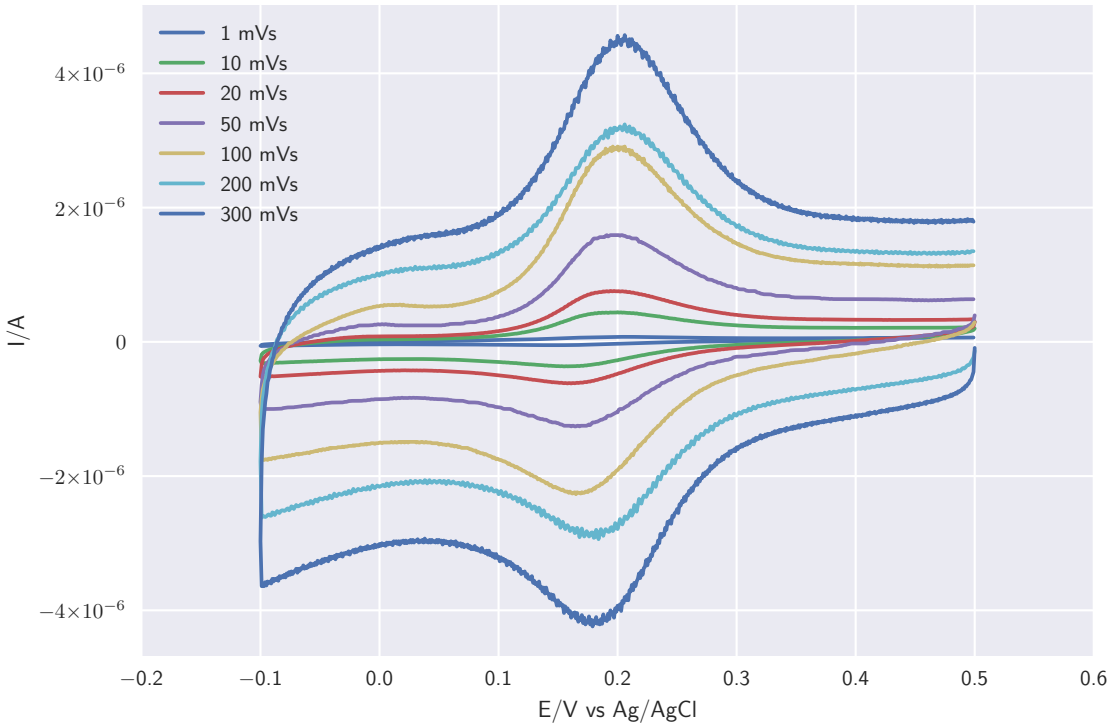


Figure 3: Various cyclic voltammograms of $[Os(2,2'-bipyridine)_2(poly-vinylimidazole)_{10}Cl]^+$ 5 mg ml^{-1} redox polymer aqueous solution immobilized on 3 mm graphite electrodes and recorded using the CHI 1030 potentiostat

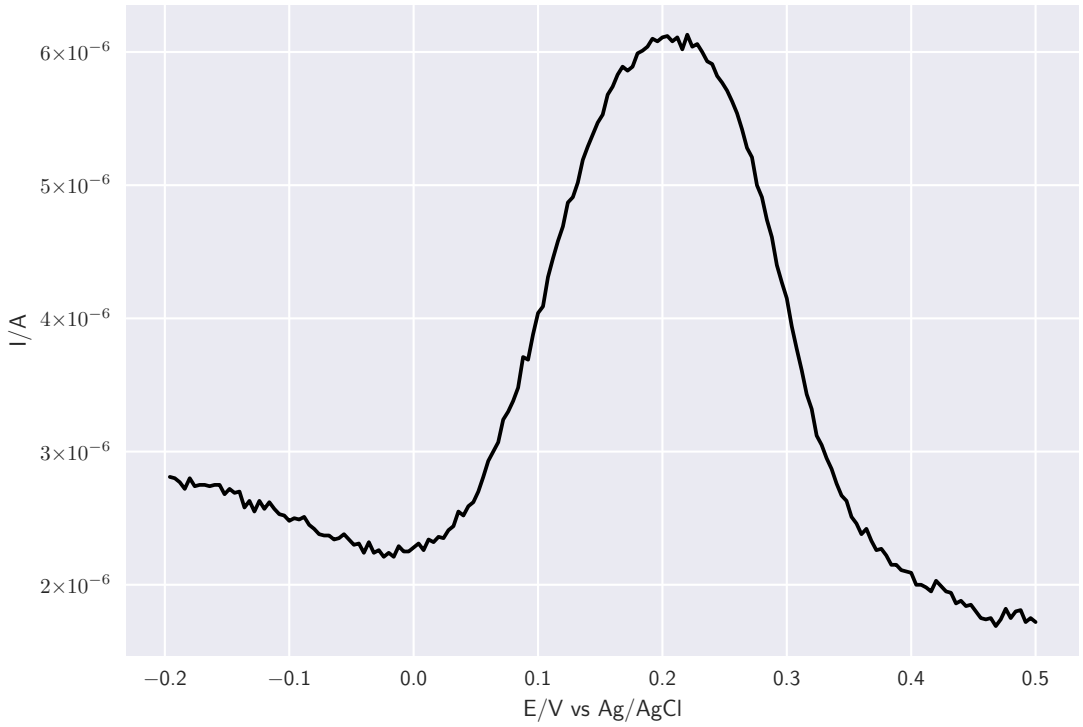


Figure 4: Differential pulse voltammogram of $[\mathrm{Os}(2,2'-\mathrm{bipyridine})_2(\mathrm{poly}-\mathrm{vinylimidazole})_{10}\mathrm{Cl}]^+$ $5\,\mathrm{mg\,ml^{-1}}$ redox polymer aqueous solution immobilized on 3 mm graphite electrodes and recorded using the CHI 1030 potentiostat

3 Operation in Artificial Plasma

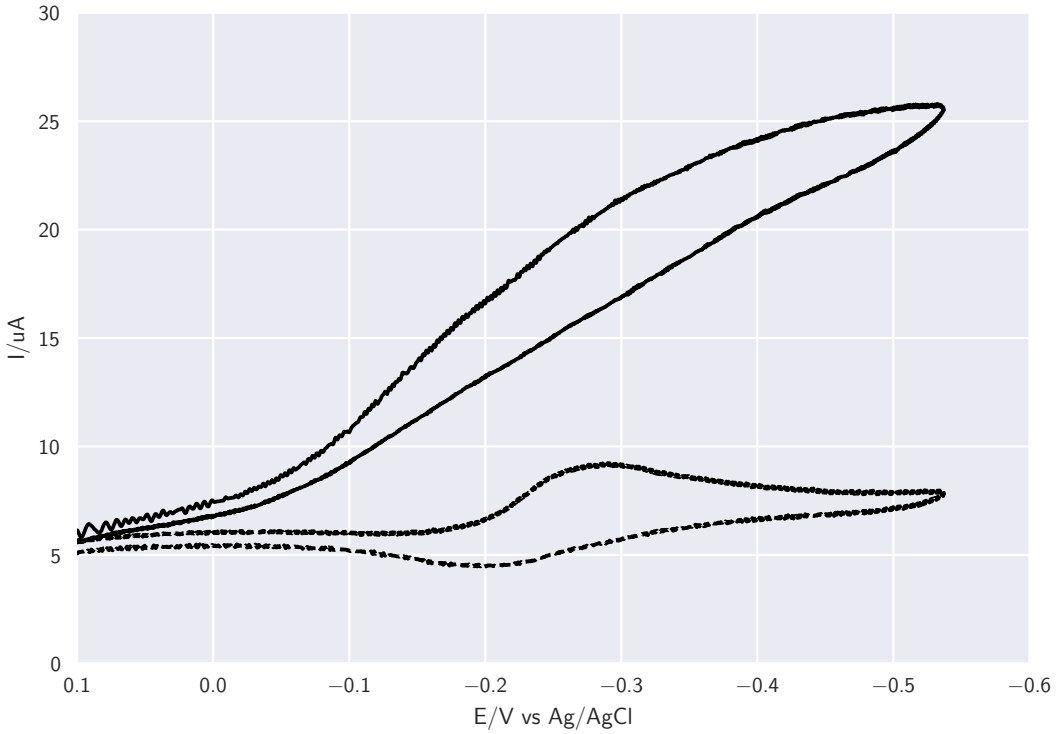


Figure 5: Slow scan ($1\,\mathrm{mV\,s^{-1}}$) CVs of films of $[\mathrm{Os}(2,2'-\mathrm{bipyridine})_2(\mathrm{poly}-\mathrm{vinylimidazole})_{10}\mathrm{Cl}]^+$, PEGDGE, MWCNTs and FADGDH on Kanichi electrodes in (- - -) absence of glucose in artificial plasma and (——) presence of 5mM glucose in artificial plasma. Recorded using the iMED

4 Electronics

4.1 Fixed Output Regulator

The LM79L05 voltage regulator used to provide a -5V potential to the circuitry was found to be unstable. To solve this a capacitor was added between the ground and Vout as detailed in the datasheet [12].

4.2 Digital-to-analog converter

The ESP-32 has a reported built in two channel 8-bit digital-to-analog converter (DAC) [1]. For true digital-to-analog conversion, important when applying a programmable potential via the potentiostat, we use an external MCP4725 DAC [13]. The key features of the MCP4725 DAC are; 12-bit resolution, fast settling time (0.6 µs), 1 external channel, single-supply operation: 2.7 - 5.5 V, and I^2C communication. The DAC is powered by 5 V to create a 5 V potential for the potentiostat circuit, this is necessary to increase the resolution of the potential window applied to the op-amps by an order of magnitude. This also overcomes the 3.3 V built in ESP-32 logic that would limit the potential window applied to the op-amp circuit for the potentiostat. The DAC requires a reference voltage (pin vcc), which must be stable, that is supplied via the reconfigured ATX power supply.

4.2.1 DAC Price Comparison

Component	Resolution	Channels	Part Number	Price (USD)	Reference
MCP4725	12-bit	1	32618620065	0.65	
MSP430F449	PWM	-	32614768918	2.58	[14],[15]
MAX5443BCUA	16-bit	1	32827261795	3.41	[16],[17]
C8051F020	12-bit	2	32822808164	3.01	[18],[19]
ATmega328P	PWM	-	32743536516	1.57	[20],[3]
ATXMEGA32A4-AU	12-bit	2	32600120380	2.43	[21],[22]

Table 5: Price comparison of various digital-to-analog converters used in lab-built potentiostats, www.aliexpress.com, last accessed April 2018. The MCP4725 is cost-effective with a 73% decrease in price from the ATXMEGA32A4-AU to the MCP4725. The ATXMEGA32A4-AU in this Table is used as an example because it is a true digital-to-analog converter closest in value to the MCP4725

4.3 Analog-to-digital converter

The ESP-32 has a reported 18 channel 12-bit analog-to-digital converter [1] (ADC). For increased resolution, important when measuring current using the potentiostat, we made obsolete the built in ADC and instead used an external ADS1115 ADC [23]. The key features of the ADS1115 ADC are; 16-bit resolution, between 8 to 860

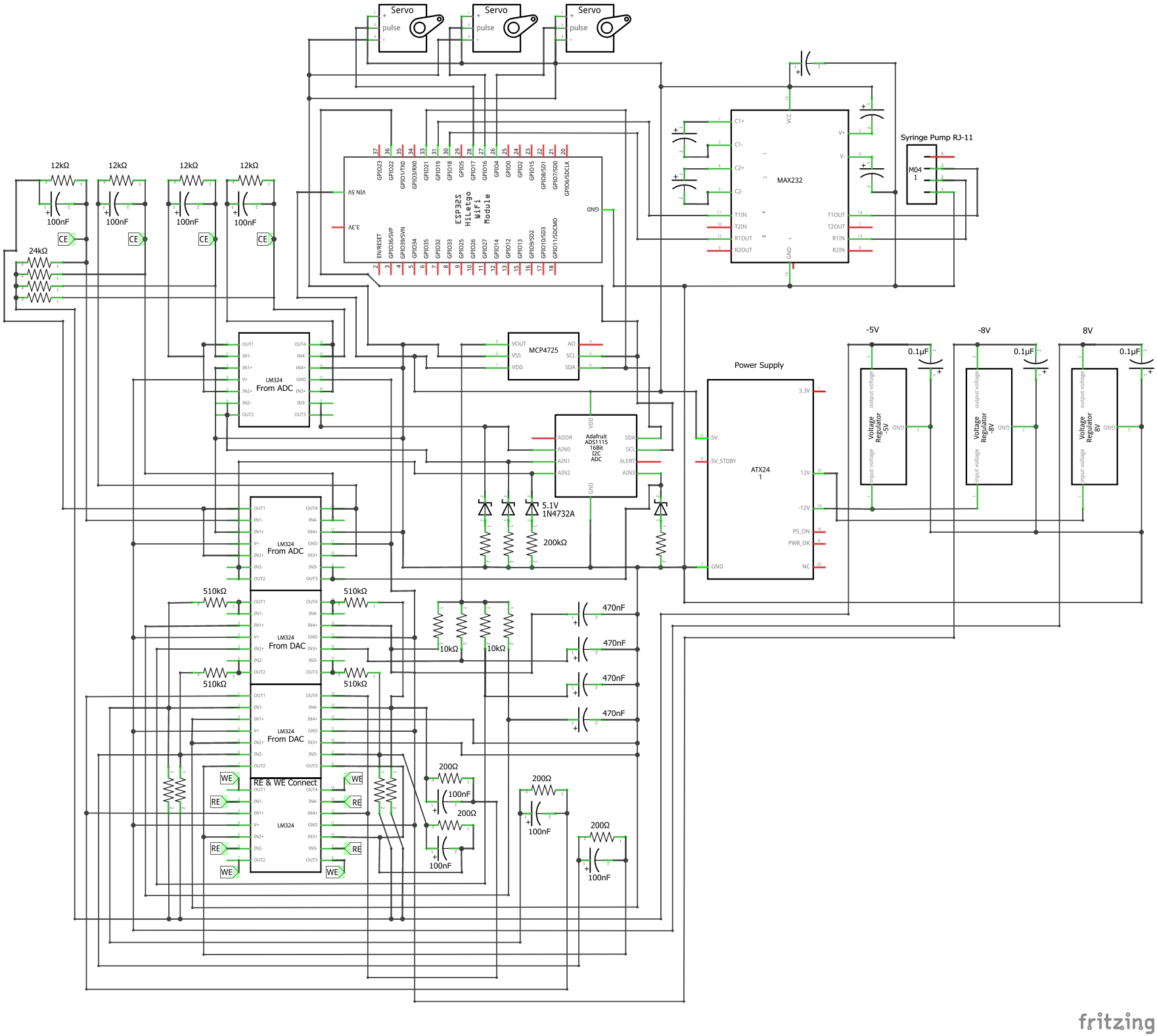
samples per second, 4 input channels, a programmable gain amplifier (PGA), and I^2C communication. The device has a number of programmable operating modes; samples per second (SPS) can be programmed within the range 8 to 860, important to note is this sets the amount of time a sample takes to read, and is not the frequency of sampling. For example, if the SPS is set to 128, each conversion will last for 1/128 seconds. This means that at lower sample rates the sample is read over a longer time period and the result of the conversion is the average of the input to the ADC over this period. This report uses 860 SPS because this is the fastest the ADC can read voltage, this is important when performing CV scan rate because a slower sampling speed may impact on the speed of the scan rate. The ADS1115 has 4 input channels, defined as A0-A3, that can be read simultaneously while operating in single ended or differential modes. Differential mode reads the voltage difference between two channels, and single ended reads the voltage difference between a single channel input and ground. This report uses four channels operating in single ended mode. The ADS1115 has a PGA. The voltage on the input pins passes through this amplifier before it enters the 16-bit ADC. The gain of this amplifier is programmable, therefore allows the measurement of small voltages with increased resolution, however, at the cost of a reduced measurement range. This report uses a 2/3x gain +/- 6.144V resulting in a 1 bit = 0.1875mV gain. This allows full use of the 5 V range of the op-amps.

4.3.1 ADC Price Comparison

Component	Resolution	Channels	Part Number	Price (USD)	Reference
ADS1115	16-bit	4	32817162654	1.99	
MSP430F449	12-bit	8	32614768918	2.58	[14],[15]
ADS1255IDBR	24-bit	2	32843087285	7.96	[16],[24]
C8051F020	12-bit	8	32822808164	3.01	[18],[19]
ATmega328P	10-bit	8	32743536516	1.57	[20],[3]
PIC16F877-04/PT	10-bit	8	32727063880	2.38	[25], [26]
ATXMEGA32A4-AU	12-bit	1	32600120380	2.43	[21],[22]
ATmega 644	10-bit	8	32703579081	2.02	[27], [28]

Table 6: Price comparison of various analog-to-digital converters used in lab-built potentiostats, www.aliexpress.com, last accessed April 2018. The ADS1115 is cost-effective with an 18% decrease in price from the ATXMEGA32A4-AU to the ADS1115. The ATXMEGA32A4-AU in this Table is used as an example because it is an analog-to-digital converter of $\leq$ resolution and closest in value to the ADS1115

4.4 Bill of Materials


Servo
pulse
Servo
pulse
Servo
pulse
12kΩ
12kΩ
12kΩ
12kΩ
100nF
100nF
100nF
100nF
CE
CE
CE
CE
24kΩ
ESP32S
HiLetgo
WiFi
Module
MAX232
Syringe Pump RJ-11
M04
1
LM324
From ADC
MCP4725
Adafruit
ADS1115
16Bit
I2C
ADC
Power Supply
ATX24
1
-5V
-8V
8V
0.1μF
Voltage
Regulator
-5V
Voltage
Regulator
-8V
Voltage
Regulator
8V
5.1V
1N4732A
200kΩ
510kΩ
510kΩ
510kΩ
510kΩ
LM324
From DAC
LM324
From DAC
10kΩ
10kΩ
470nF
470nF
470nF
470nF
200Ω
100nF
200Ω
100nF
200Ω
100nF
200Ω
100nF
RE & WE Connect
WE
RE
RE
WE
WE
RE
RE
WE

fritzing

Potentiostat			
Component	Vendor	Part Number	Price (USD)
ESP-32	AliExpress	32799954012	8.21
Solderless Breadboard	AliExpress	32801985959	0.75
Resistor Assortment (600 Pcs)	AliExpress	32816049069	2.25
Jumper Wire Assortment	Newark	99W1758	3.11
LM324N op-amp (20 Pcs)	AliExpress	32460833361	1.55
5V1 Zener Diode (100 Pcs)	AliExpress	32829392047	0.77
ADS1115	AliExpress	32817162654	1.99
MCP4725	AliExpress	32618620065	0.65
MAX232 (10 Pcs)	AliExpress	1702920370	1.50
Ceramic Capacitors Assortment (300 Pcs)	AliExpress	32659574417	3.98
LM79L05	AliExpress	32838411757	2.27
LM79L08	Mouser	511-L79L08ACZ-AP	1.53
LM78L08	Mouser	511-L78L08ABZ-AP	1.42
		Total:	29.98
Microfluidics			
Aladdin AL-1000	WPI	AL1000	660
MG996R Servo Motor (3 Pcs)	AliExpress	32823408130	11.73
PEEK tubing 1/16	Idex	1531	16.81
Fitting 1/16	Idex	6000-254BL	49.97
Ferrule 1/16	Idex	P-840	1.28
100 uL sample loop	Idex	9055-024	41.43
4-way switching valve (3 Pcs)	Idex	V-101D	112.78
SYLGARD 184 (PDMS kit)	Sigma	761036-5EA	161.50
		Total:	1393.84

Table 7: Bill of Materials list, www.aliexpress.com, www.newark.com, www.adafruit.com, www.idex-hs.com, www.wpi-europe.com, www.sigmaaldrich.com, www.mouser.com last accessed January 2018

5 C Programs

5.1 Online Repository

The C Programs contained within this SI have an MIT license. All program files are also uploaded to a Github repository[4] in an effort to improve the re-usability of this work.

5.2 CP210x USB to UART Bridge VCP Drivers

1. A driver needs to be installed onto any Windows/Macintosh OSX/Linux/Android platform in order to upload code to the ESP-32
2. Install the latest CP210x USB to UART Bridge VCP Drivers from: https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers
3. Once the CP210x driver is installed the computer platform can be used to communicate with ESP-32 via a micro USB cable

5.3 Installing Arduino ESP32

1. Install the latest Arduino integrated development environment (IDE) software version 1.8.5 or later [5] onto Windows
2. Install the latest Git software [6] onto Windows
3. Run the GIT GUI program
4. Select '*Clone Existing Repository*'
5. Enter into the field '*Source Location*' the following: https://github.com/espressif/arduino-esp32.git
6. Enter into the field '*Target Directory*' the following[7]: '*[ARDUINO_SKETCHBOOK_DIR]/hardware/espressif/esp32*'
7. Click '*Clone*' to start cloning the repository
8. Open Git Bash program and enter: '*[ARDUINO_SKETCHBOOK_DIR]/hardware/espressif/esp32*' and execute git submodule update –init –recursive
9. Open '*[ARDUINO_SKETCHBOOK_DIR]/hardware/espressif/esp32/tools*' and double-click get.exe
10. When get.exe finishes, plug in ESP32 board via USB and wait for Windows to install drivers

[4] https://github.com/ConanMercer/Wireless-Potentiostat
[5] https://www.arduino.cc/en/main/software
[6] https://git-scm.com/download/win
[7] Sketchbook Directory: Usually C:/Users/[YOUR_USER_NAME]/Documents/Arduino

11. Full documentation on how to use the Arduino ESP32 Core can be found on the Internet [8]

5.3.1 Setting Board Specifications

Select the following settings once the Arduino ESP32 is installed, settings found in tools drop down menu in the Arduino IDE program.

1. Board: '*ESP32 Dev module*'
2. Flash Mode: '*QIO*'
3. Flash Frequency: '80 MHz'
4. Flash Size: '*4MB (32Mb)*'
5. Upload Speed: '*921600*'
6. Core Debug Level: '*None*'

5.4 Setup WiFi Manager

Instead of hard coding the WiFi credentials into the ESP-32 that would require reprogramming the device every time a new WiFi network connection is desired, a WiFi manager library is used to allow connection to a WiFi network where the user can use a smart phone or other device. When the ESP-32 is booted it will attempt to connect to the last successfully connected WiFi network. If this fails the ESP-32 will start up an access point (AP) mode. Once in AP mode, the user can use a smart phone or any WiFi enabled device (computer, laptop, tablet) with a web browser to navigate to the default IP address which is `192.168.4.1`, once connected to the address a configuration window will display allowing the user to type the WiFi credentials of the network desired for connection to by the ESP-32, see Figure 6. Once the credentials have been entered, the ESP-32 will connect to the specified WiFi network.

Follow the settings below to include the WiFi manager library in the ESP-32 program.

1. Download and install this library found on a GitHub repository `https://github.com/zhouhan0126/WIFIMANAGER-ESP32`
2. Open Arduino IDE software
3. Verify that WiFi manger has been installed locating WiFi manager in Select Sketch > Include Library

[8] `https://github.com/espressif/arduino-esp32`

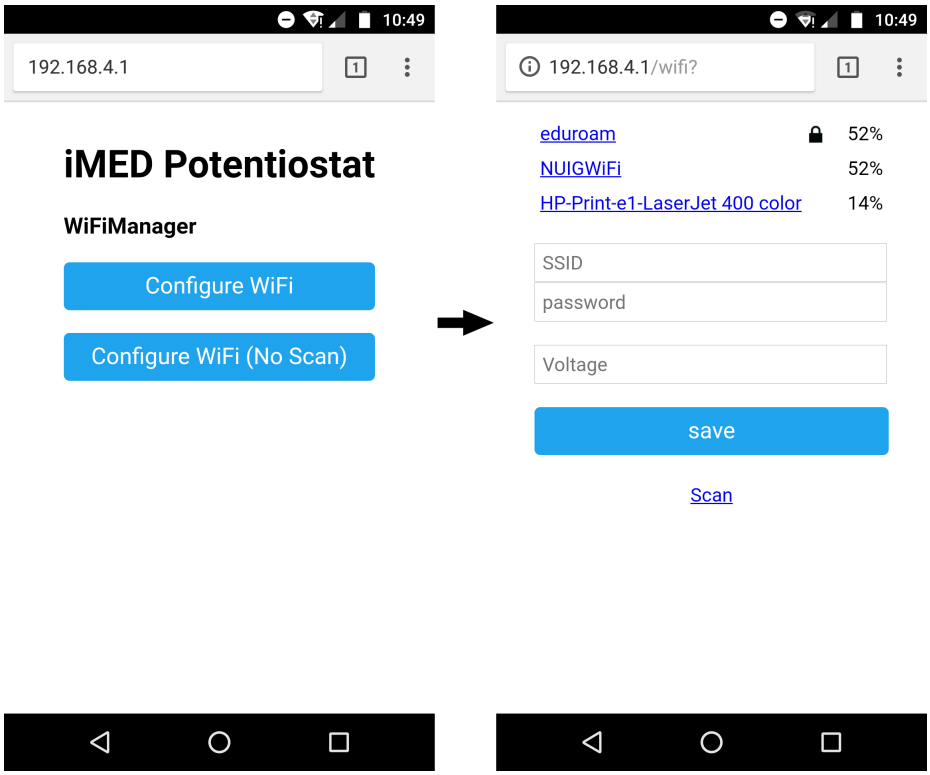


Figure 6: Screenshot showing the iMED which is connected to a smartphone via WiFi. The web browser viewed using the smartphone controls the input of the WiFi credentials and the voltage to be applied during amperometry.

5.5 Setup Over the Air updates

The Arduino OTA library is integrated into the ESP32 library, see section 5.3. This OTA library facilitates the iMED to update itself via WiFi based on data received while the normal firmware is running. Briefly, OTA works by configuring the partition table of the ESP32 with at least two 'OTA image' partitions and an 'OTA data partition'. The OTA operation function writes a new firmware image to whichever OTA image slot is not currently being used for booting. Once the image is verified, the OTA Data partition is updated to specify that this image should be used for the next boot, and then the device reboots itself, running the updated C program, detailed in Figure 7.

Follow the settings below to upload programs via OTA with the Arduino IDE.

1. Select Tools > Port > OTA ESP32 at 192.168.1.94 (ESP Dev Module)
2. Upload the Arduino program as normal

5.6 Setup ThingSpeak Channel

Follow the settings in the supporting information of our previously published work [29] for a detailed description of how to setup a ThinkSpeak channel.

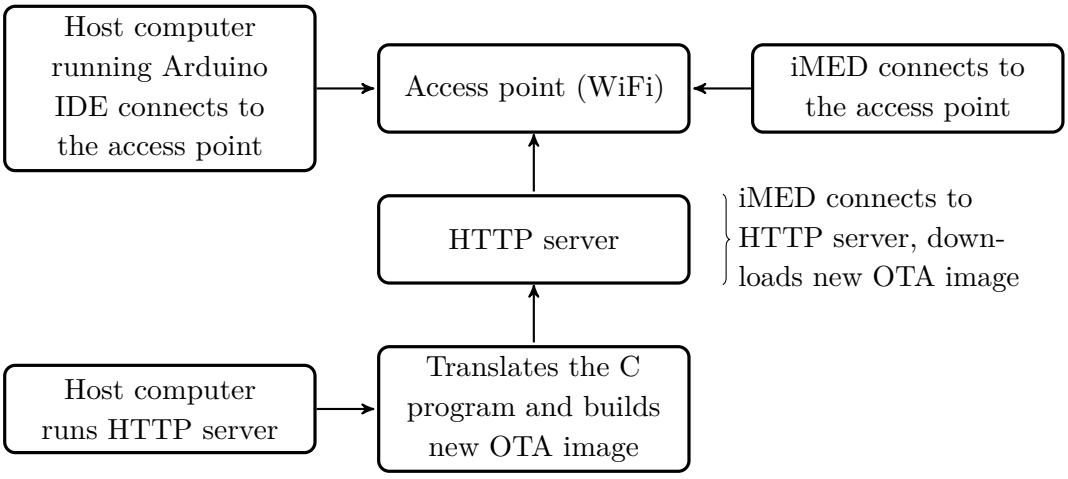


Figure 7: Diagram showing the overall work-flow of 'over the air' updates.

5.7 Syntax

The ESP-32 microcontroller is programmed in C/C++ language that is uploaded via the Arduino IDE. Libraries are used to simplify C programs. A library in C is a group of functions and declarations, exposed for use by other programs. The library therefore consists of an interface expressed in a .h file, and are stated in the header of the code that wishes to call a library (see lines 9 - 27 of C program in supporting information). The Arduino ESP32 board manager comes with libraries to communicate with wireless Wi-Fi using TCP protocols [9], and wired SPI[10] and I^2C[11] peripherals. Other library's used in his report include, adafruit_ADS1015.h, Adafruit_MCP4725.h, HardwareSerial.h, Servo.h. The C program requires two basic functions to run correctly, setup() and loop(). Setup is called first when the code is run and it is used to initiate pin modes and begin any communication such as serial. The setup must be included, regardless if statements are present. The loop is called to run after setup and is a powerful function that loops consecutively. The loop function responds to inputs and controls the ESP-32 board. By using FreeRTOS task (included in the Arduino ESP32 board manager) a program can assign tasks to specific cores on the ESP-32. The '*xTaskCreatePinnedToCore*' function is used to assign ADC and DAC modules to separate cores. Several attribute parameters can be defined, such as the task function, name, stack size, priority, handle, as well as the specific core to run on the task on.

[9] Transmission Control Protocol (TCP) is a Internet connection that once established, can send data in a bidirectional mode.

[10] Serial Peripheral Interface bus (SPI) is a synchronous serial communication interface specification used for short distance communication, primarily in embedded computer systems.

[11] I^2C is also used for short distance communication for lower-speed peripheral ICs attached to microcontrollers.

```
readChannelID = CHANNEL_ID; % TODO - replace CHANNEL_ID with your ThingSpeak channel ID
CurrentFieldID = 2;
readAPIKey = 'CHANNEL_READAPIKEY'; % TODO - replace CHANNEL_READAPIKEY with your ThingSpeak channel read API key

CurrentData = thingSpeakRead(readChannelID, 'Fields', CurrentFieldID, 'ReadKey', readAPIKey);

x = linspace(0,0.1,length(CurrentData)); % Generate linearly spaced time vector for x axis equal to length of y input data

thingSpeakScatter(x, CurrentData, 'xlabel', 'Time, s', 'ylabel', 'I, A '); % Graph amperometric data
```

Listing 1: MATLAB code

5.8 JavaScript Object Notation

For a detailed description of JavaScript Object Notation (JSON) and using JSON with ThingSpeak, follow the supporting information of our previously published work [29].

5.9 MATLAB Visualizations

MATLAB integration with ThinkSpeak is used to plot amperometric data from the iMED. MATLAB code is detailed in Listing 1. A screenshot of the MATLAB graphical visualization taken on a smartphone is detailed in Figure 8.

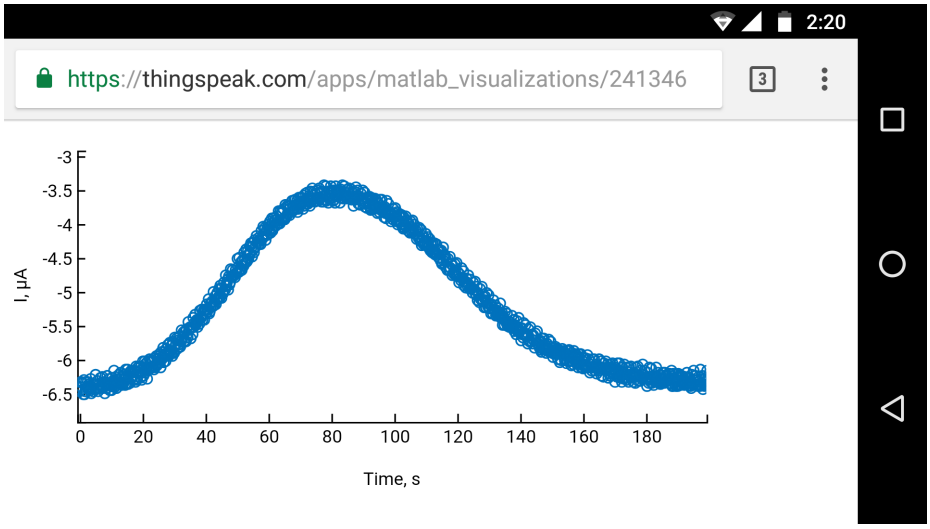


Figure 8: Screenshot showing the iMED which is connected to a smartphone via WiFi. The web browser viewed using the smartphone displays the MATLAB graphical visualization of amperometric data from the iMED.

5.10 C Programs

```
//-------------------------------------------------------------------
// --- iMED code written by Conan Mercer 2018. National University of Ireland Galway
// --- For over the air upload to ESP-32 microcontroller and Thingspeak upload
//-------------------------------------------------------------------

//-------------------------------------------------------------------
// --- Libraries
//-------------------------------------------------------------------
#include <Servo.h>               // Communicate with servo-valves
#include <HardwareSerial.h>      // To utilise all of ESP32 hardware UARTs
#include <Wire.h>                // Serial communicate with I2C / TWI devices
#include <Adafruit_ADS1015.h>    // Driver for ADS1115 16-bit ADC
#include <Adafruit_MCP4725.h>    // Driver for MCP4725 12-bit DAC
//For smart phone connection
#include "SPIFFS.h"
#include <WiFi.h>
#include <DNSServer.h>
#include <ESP8266WebServer.h>
#include <WiFiManager.h>
#include <ArduinoJson.h>
//For OTA updates
#include <ESPmDNS.h>
#include <WiFiUdp.h>
#include <ArduinoOTA.h>
#include "credentials.h"
#include <stdlib.h>
#include <stdio.h>

//-------------------------------------------------------------------
// --- Variables and settings
//-------------------------------------------------------------------

Adafruit_ADS1115 ads(0x49);      // Construct the ADS1115 at address 0x49
Adafruit_MCP4725 dac;            // Construct the MCP4725

WiFiClient client;               // Initialize WiFi client library
char server[] = "api.thingspeak.com"; // ThingSpeak Server address

Servo servo1;                    // Define servo-valves
Servo servo2;
Servo servo3;

HardwareSerial PumpSerial(1);    // Define hardware UARTs

/* Collect data once every 500 milliseconds and post data to ThingSpeak channel once every 16 seconds */
unsigned long lastConnectionTime = 0;              // Track the last connection time
unsigned long previousUpdate = 0;                  // Track the last update time
const unsigned long postingInterval = 16L * 1000L; // Post data every 16 seconds
```

```
49 const unsigned long updateInterval = 0.5L * 1000L;  // Update JSON once every
       500ms
50
51 /* Multi core tasks */
52 TaskHandle_t Task1, Task2;
53
54 int16_t adc0, adc1, adc2, adc3; // Signed integer type with width of 16 bits
       for ADC using four channels
55 uint32_t dac_value;             // Unsigned integer type with width of 32 bits
        for DAC values
56
57 //flag for saving data
58 bool shouldSaveConfig = false;
59
60 //callback notifying the need to save config
61 void saveConfigCallback () {
62   Serial.println("Should save config");
63   shouldSaveConfig = true;
64 }
65
66 char jsonBuffer[1500] = "[";  // Initialize the jsonBuffer to hold data
67 char Voltage[2890] = "";           // Variable for control of DAC output set
       to 0.4V, this can be changed in wifimanager via iMED access point
68 float c1 = 0;    // Variable for storage of current reading channel 1
69 float c2 = 0;    // Variable for storage of current reading channel 2
70 float c3 = 0;    // Variable for storage of current reading channel 3
71 float c4 = 0;    // Variable for storage of current reading channel 4
72 float v = 0;     // Variable for storage of voltage applied
73 float A = 0.01527;       // Variable for storage of slope current conversion
       200uA
74 float B = 200;           // Variable for storage of intersect current
       conversion 200uA
75 float M = 0.000504637;  // Variable for storage of voltage multiplier
76 float O = 1.01562452;   // Variable for storage of voltage offset
77 float E = 0;             // Variable for storage of voltage converter
78 byte pumpStop[] = {0x73, 0x74, 0x70, 0x0d, 0x0a}; //Pump stop
79 byte pump100ul[] = {0x72, 0x61, 0x74, 0x31, 0x30, 0x30, 0x75, 0x6d, 0x0d, 0x0a
       , 0x72, 0x75, 0x6e, 0x0d, 0x0a}; //Pump 100ul/min
80
81 //-------------------------------------------------------------------
82 // --- iMED initialisation
83 //-------------------------------------------------------------------
84
85 void setup() {
86   Serial.begin(115200);   // Initiate serial communication to ESP-12E at
       112500 baud rate
87   SPIFFS.begin (true);
88   //read configuration from FS json
89   Serial.println("mounting FS...");
90   if (SPIFFS.begin()) {
91     Serial.println("mounted file system");
92     if (SPIFFS.exists("/config.json")) {
93       //file exists, reading and loading
94       Serial.println("reading config file");
95       File configFile = SPIFFS.open("/config.json", "r");
96       if (configFile) {
```

```
97          Serial.println("opened config file");
98          size_t size = configFile.size();
99          // Allocate a buffer to store contents of the file.
100         std::unique_ptr<char[]> buf(new char[size]);
101
102         configFile.readBytes(buf.get(), size);
103         DynamicJsonBuffer jsonBuffer1;
104         JsonObject& json = jsonBuffer1.parseObject(buf.get());
105         json.printTo(Serial);
106         if (json.success()) {
107           Serial.println("\nparsed json");
108
109           strcpy(Voltage, json["Voltage"]);
110         } else {
111           Serial.println("failed to load json config");
112         }
113       }
114     }
115   } else {
116     Serial.println("failed to mount FS");
117   }
118   //end read
119   WiFiManagerParameter custom_Voltage("Voltage", "Voltage", Voltage, 40);
120   //Local intialization. Once its business is done, there is no need to keep
        it around
121   WiFiManager wifiManager;
122   //set config save notify callback
123   wifiManager.setSaveConfigCallback(saveConfigCallback);
124   //add all parameters here
125   wifiManager.addParameter(&custom_Voltage);
126
127   if (!wifiManager.autoConnect("iMED Potentiostat")) {
128     Serial.println("failed to connect and hit timeout");
129     delay(3000);
130     //reset
131     ESP.restart();
132     delay(5000);
133   }
134
135   //if you get here you have connected to the WiFi
136   Serial.println("connected...)");
137
138   //read updated parameters
139   strcpy(Voltage, custom_Voltage.getValue());
140
141   //save the custom parameters to FS
142   if (shouldSaveConfig) {
143     Serial.println("saving config");
144     DynamicJsonBuffer jsonBuffer1;
145     JsonObject& json = jsonBuffer1.createObject();
146     json["Voltage"] = Voltage;
147     File configFile = SPIFFS.open("/config.json", "w");
148     if (!configFile) {
149       Serial.println("failed to open config file for writing");
150     }
151     json.printTo(Serial);
```

```
152     json.printTo(configFile);
153     configFile.close();
154     Serial.println("Voltage");
155     //end save
156   }
157
158   //-----------------------------------------------------------------
159   // --- Setup needed for OTA
160   //-----------------------------------------------------------------
161   ArduinoOTA.onStart([]() {
162     String type;
163     if (ArduinoOTA.getCommand() == U_FLASH)
164       type = "sketch";
165     else // U_SPIFFS
166       type = "filesystem";
167     // NOTE: if updating SPIFFS this would be the place to unmount SPIFFS
      using SPIFFS.end()
168     Serial.println("Start updating " + type);
169   });
170   ArduinoOTA.onEnd([]() {
171     Serial.println("\nEnd");
172   });
173   ArduinoOTA.onProgress([](unsigned int progress, unsigned int total) {
174     Serial.printf("Progress: %u%%\r", (progress / (total / 100)));
175   });
176   ArduinoOTA.onError([](ota_error_t error) {
177     Serial.printf("Error[%u]: ", error);
178     if (error == OTA_AUTH_ERROR)          Serial.println("Auth Failed");
179     else if (error == OTA_BEGIN_ERROR)    Serial.println("Begin Failed");
180     else if (error == OTA_CONNECT_ERROR)  Serial.println("Connect Failed");
181     else if (error == OTA_RECEIVE_ERROR)  Serial.println("Receive Failed");
182     else if (error == OTA_END_ERROR)      Serial.println("End Failed");
183   });
184   ArduinoOTA.begin();
185   Serial.println("Ready");
186   Serial.print("IP address: ");
187   Serial.println(WiFi.localIP());
188
189   /* Core 0 task */
190   xTaskCreatePinnedToCore(
191     core0Task,                  /* task function. */
192     "core0",                    /* name of task. */
193     10000,                      /* stack size of task */
194     NULL,                       /* parameter of the task */
195     1,                          /* priority of the task */
196     &Task1,                     /* task handle to keep track of created task */
197     0);                         /* Core */
198
199   delay(500);                   /* needed to start-up Core 1 task */
200
201   /* Core 1 task */
202   xTaskCreatePinnedToCore(
203     core1Task,                  /* task function. */
204     "core1",                    /* name of task. */
205     10000,                      /* stack size of task */
206     NULL,                       /* parameter of the task */
```

```
207     1,                          /* priority of the task */
208     &Task2,                     /* task handle to keep track of created task */
209     1);                         /* Core */
210
211   //-------------------------------------------------------------------
212   // --- Servo-valve controls
213   //-------------------------------------------------------------------
214   chamber1();      // Change flow into detection chamber 1
215   //chamber2();    // Change flow into detection chamber 2
216   //chamber3();    // Change flow into detection chamber 3
217   //chamber4();    // Change flow into detection chamber 4
218
219   //-------------------------------------------------------------------
220   // --- Servo-valve controls
221   //-------------------------------------------------------------------
222   PumpSerial.begin(19200, SERIAL_8N1, 19, 18);      // RS-232 Protocol for
        Aladdin Syringe Pump Model: AL-1000
223   PumpSerial.write(pump100ul, sizeof(pump100ul));   // Initiate pumping at 100
        ul/min
224   Wire.begin(21, 22);               // I2C Pins on ESP32
225   ads.begin();                      // Initiate ADS1115 ADC
226   dac.begin(0x62);                  // Initiate MCP4725 DAC
227   Wire.setClock(400000L);           // Set i2c to 400 kHz
228   ads.setGain(GAIN_TWOTHIRDS);      // 2/3x gain +/- 6.144V  1 bit = 0.1875mV
229   //ads.setSPS(ADS1115_DR_860SPS);  // Set ADS1115 samples per second to 860 (
        effects scan rate speed)
230   E = atoi(Voltage);                // convert char to int for use with DAC
231   dac.setVoltage(E, false);         // Set potential, false = don't save to EEPROM
232 }
233
234 //-------------------------------------------------------------------
235 // --- This function will read the current using core 1
236 //-------------------------------------------------------------------
237 void core0Task( void * parameter )
238 {
239   /* loop forever */
240   for (;;) {
241
242     if (millis() - previousUpdate >=  updateInterval) {
243       updatesJson(jsonBuffer);
244     }
245   }
246   /* delete a task when finish,
247     this will never happen because this is infinity loop */
248   vTaskDelete( NULL );
249 }
250
251 //-------------------------------------------------------------------
252 // --- This function will read the voltage using core 2
253 //-------------------------------------------------------------------
254 void core1Task( void * parameter )
255 {
256   /* loop forever */
257   for (;;) {
258     voltageT();
259   }
```

```
260   /* delete a task when finish,
261      this will never happen because this is infinity loop */
262   vTaskDelete( NULL );
263 }
264
265 void loop() {
266 }
267
268 //---------------------------------------------------------------
269 // --- Checks for a wireless OTA update and computes Voltage
270 //---------------------------------------------------------------
271 void voltageT()
272 {
273   ArduinoOTA.handle();
274   yield();
275   delay(10);
276   v = ((E * M) - O);
277 }
278
279 //---------------------------------------------------------------
280 // --- Calls potential reading according to 'updateInterval'
281 //---------------------------------------------------------------
282 void potential()
283 {
284   if (millis() - previousUpdate >=  updateInterval) {
285     updatesJson(jsonBuffer);
286   }
287 }
288
289 //---------------------------------------------------------------
290 // --- Updates JSON with data
291 //---------------------------------------------------------------
292
293 void updatesJson(char* jsonBuffer) {
294   v = ((E * M) - O);
295   adc0 = ads.readADC_SingleEnded(0);
296   //adc1 = ads.readADC_SingleEnded(1);
297   //adc2 = ads.readADC_SingleEnded(2);
298   //adc3 = ads.readADC_SingleEnded(3);
299   c1 = (-((A * (adc0)) - B));    // Current reading output in  A  channel 1
300   //c2 = (-((A * (adc1)) - B)); // Current reading output in  A  channel 2
301   //c3 = (-((A * (adc2)) - B)); // Current reading output in  A  channel 3
302   //c4 = (-((A * (adc3)) - B)); // Current reading output in  A  channel 4
303   Serial.print(v, 4);
304   Serial.print(",");
305   Serial.println(c1, 3);
306   // Format the jsonBuffer as noted above
307   strcat(jsonBuffer, "{\"delta_t\":");
308   unsigned long deltaT = (millis() - previousUpdate) / 1000;
309   size_t lengthT = String(deltaT).length();
310   char temp[8];
311   String(deltaT).toCharArray(temp, lengthT + 1);
312   strcat(jsonBuffer, temp);
313   strcat(jsonBuffer, ",");
314   strcat(jsonBuffer, "\"field1\":");
315   lengthT = String(c1).length();
```

```
316   String(v).toCharArray(temp, lengthT + 1);      // Data uploaded to ThinkSpeak
        channel field 1
317   strcat(jsonBuffer, temp);
318   strcat(jsonBuffer, ",");
319   strcat(jsonBuffer, "\"field2\":");
320   String(c1).toCharArray(temp, lengthT + 1);     // Data uploaded to ThinkSpeak
        channel field 2
321   strcat(jsonBuffer, temp);
322   strcat(jsonBuffer, "},");
323   // If posting interval time has reached 16 seconds, update the ThingSpeak
        channel with data
324   if (millis() - lastConnectionTime >=  postingInterval) {
325     size_t len = strlen(jsonBuffer);
326     jsonBuffer[len - 1] = ']';
327     httpRequest(jsonBuffer);
328   }
329   previousUpdate = millis(); // Update the last update time
330 }
331
332 //-------------------------------------------------------------------
333 // --- Updates the ThingSpeak open-source server channel with data
334 //-------------------------------------------------------------------
335 void httpRequest(char* jsonBuffer) {
336   // Format the data buffer as noted above
337   char data[1500] = "{\"write_api_key\":\"YOUR-CHANNEL-WRITEAPIKEY\",\"updates
        \":"; // Replace YOUR-CHANNEL-WRITEAPIKEY with your ThingSpeak channel
        write API key
338   strcat(data, jsonBuffer);
339   strcat(data, "}");
340   // Close any connection before sending a new request
341   client.stop();
342   String data_length = String(strlen(data) + 1); //Compute the data buffer
        length
343   Serial.println(data);
344   // POST data to ThingSpeak
345   if (client.connect(server, 80)) {
346     client.println("POST /channels/YOUR-CHANNEL-ID/bulk_update.json HTTP/1.1")
      ; // Replace YOUR-CHANNEL-ID with your ThingSpeak channel ID
347     client.println("Host: api.thingspeak.com");
348     client.println("User-Agent: mw.doc.bulk-update (Arduino ESP8266)");
349     client.println("Connection: close");
350     client.println("Content-Type: application/json");
351     client.println("Content-Length: " + data_length);
352     client.println();
353     client.println(data);
354   }
355   jsonBuffer[0] = '[';                          // Reinitialize the jsonBuffer for
        next batch of data
356   jsonBuffer[1] = '\0';
357   lastConnectionTime = millis();                // Update the last connection time
358 }
359
360 //-------------------------------------------------------------------
361 // --- Servo-valve control subroutines
362 //-------------------------------------------------------------------
363 void chamber1() {
```

```
364   servo1.attach(17);
365   servo2.attach(16);
366   servo1.write(150);       // Turn 1st Servo right to 150 degrees (directs to
        chamber 1)
367   servo2.write(70);        // Turn 2nd Servo left to 60 degrees (directs to
        chamber 1)
368   delay(2000);
369   servo1.detach();
370   servo2.detach();
371 }
372 void chamber2() {
373   servo1.attach(17);
374   servo3.attach(4);
375   servo1.write(40);        // Turn 1st Servo left to 40 degrees (directs to
        chamber 2)
376   servo3.write(160);       // Turn 3rd Servo left to 160 degrees (directs to
        chamber 2)
377   delay(2000);
378   servo1.detach();
379   servo3.detach();
380 }
381 void chamber3() {
382   servo1.attach(17);
383   servo2.attach(16);
384   servo1.write(150);       // Turn 1st Servo right to 150 degrees (directs to
        chamber 3)
385   servo2.write(180);       // Turn 2nd Servo left to 60 degrees (directs to
        chamber 3)
386   delay(2000);
387   servo1.detach();
388   servo2.detach();
389 }
390 void chamber4() {
391   servo1.attach(17);
392   servo3.attach(4);
393   servo1.write(40);        // Turn 1st Servo left to 40 degrees (directs to
        chamber 4)
394   servo3.write(50);        // Turn 3rd Servo left to 50 degrees (directs to
        chamber 4)
395   delay(2000);
396   servo1.detach();
397   servo3.detach();
398 }
```

References

[1] Espressif, ESP32 Datasheet (2018).
URL https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

[2] Espressif, ESP8266EX Datasheet (2017).
URL https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf

[3] Atmel, ATmega328 / P Datasheet (2016). doi:10.1104/pp.108.130294.
URL ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Summary.pdf

[4] Freescale Semiconductor, K20P64M72SF1 Datasheet (2012). arXiv:arXiv:1011.1669v3, doi:10.1002/ejoc.201200111.

[5] Adafruit, Introducing the Raspberry Pi Model B+ Datasheet (2015).
URL https://cdn-shop.adafruit.com/datasheets/pi-specs.pdf

[6] Texas Instruments, AM335x ARM ® Cortex ™ -A8 Microprocessors (MPUs) Datasheet (2011).

[7] A. Ainla, M. P. S. Mousavi, M.-N. Tsaloglou, J. Redston, J. G. Bell, M. T. Fernández-Abedul, G. M. Whitesides, Open-Source Potentiostat for Wireless Electrochemical Detection with Smartphones, Anal. Chem. 90 (10) (2018) 6240–6246. doi:10.1021/acs.analchem.8b00850.
URL http://pubs.acs.org/doi/10.1021/acs.analchem.8b00850

[8] TowerPro, MG996R High Torque Metal Gear Dual Ball Bearing Servo Datasheet (2018).
URL https://www.electronicoscaldas.com/datasheet/MG996R_Tower-Pro.pdf

[9] Z. Li, S. Y. Mak, A. Sauret, H. C. Shum, Syringe-pump-induced fluctuation in all-aqueous microfluidic system implications for flow rate accuracy, Lab Chip 14 (4) (2014) 744. doi:10.1039/c3lc51176f.
URL http://www.ncbi.nlm.nih.gov/pubmed/24382584{%}5Cnhttp://xlink.rsc.org/?DOI=c3lc51176fhttp://xlink.rsc.org/?DOI=c3lc51176f

[10] W. Zeng, I. Jacobi, D. J. Beck, S. Li, H. A. Stone, Characterization of syringe-pump-driven induced pressure fluctuations in elastic microchannels, Lab Chip 15 (4) (2015) 1110–1115. doi:10.1039/C4LC01347F.
URL www.rsc.org/lochttp://xlink.rsc.org/?DOI=C4LC01347F

[11] T. Ward, M. Faivre, M. Abkarian, H. A. Stone, Microfluidic flow focusing: Drop size and scaling in pressureversus flow-rate-driven pumping, Electrophoresis 26 (19) (2005) 3716–3724. doi:10.1002/elps.200500173.
URL http://doi.wiley.com/10.1002/elps.200500173

[12] T. Instruments, LM79LXXAC Series 3-Terminal Negative Regulators (2013).
URL http://www.ti.com/lit/ds/symlink/lm79l.pdf

[13] Microchip Technology Inc, 12-Bit DAC with EEPROM Memory in SOT-23-6 (2009).

[14] S. Kwakye, A. Baeumner, An embedded system for portable electrochemical detection, Sensors Actuators B Chem. 123 (1) (2007) 336–343. doi:10.1016/j.snb.2006.08.032.
URL http://linkinghub.elsevier.com/retrieve/pii/S0925400506005764

[15] T. Instruments, MSP430x43x, MSP430x43x1, MSP430x44x Mixed Signal Microcontroller (2009).

[16] M. D. M. Dryden, A. R. Wheeler, DStat: A Versatile, Open-Source Potentiostat for Electroanalysis and Integration, PLoS One 10 (10) (2015) e0140349. doi:10.1371/journal.pone.0140349.
URL http://dx.plos.org/10.1371/journal.pone.0140349

[17] M. Integrated, +3V/+5V Serial-Input Voltage-Output 16-Bit DAC (2009).

[18] C.-Y. Huang, M.-J. Syu, Y.-S. Chang, C.-H. Chang, T.-C. Chou, B.-D. Liu, A portable potentiostat for the bilirubin-specific sensor prepared from molecular imprinting, Biosens. Bioelectron. 22 (8) (2007) 1694–1699. doi:10.1016/j.bios.2006.07.036.
URL http://linkinghub.elsevier.com/retrieve/pii/S0956566306003617

[19] S. Labs, C8051F020/1/2/3 (2003).

[20] G. N. Meloni, Building a Microcontroller Based Potentiostat: A Inexpensive and Versatile Platform for Teaching Electrochemistry and Instrumentation, J. Chem. Educ. 93 (7) (2016) 1320–1322. doi:10.1021/acs.jchemed.5b00961.
URL http://pubs.acs.org/doi/abs/10.1021/acs.jchemed.5b00961

[21] A. A. Rowe, A. J. Bonham, R. J. White, M. P. Zimmer, R. J. Yadgar, T. M. Hobza, J. W. Honea, I. Ben-Yaacov, K. W. Plaxco, CheapStat: An Open-Source, "Do-It-Yourself" Potentiostat for Analytical and Educational Applications, PLoS One 6 (9) (2011) e23783. doi:10.1371/journal.pone.0023783.
URL http://dx.plos.org/10.1371/journal.pone.0023783

[22] Atmel, 8/16-bit XMEGA A4 Microcontroller (2013).

[23] Texas Instruments, ADS111x Ultra-Small, Low-Power, I2C-Compatible, 860-SPS, 16-Bit ADCs With Internal Reference, Oscillator, and Programmable Comparator (2018).

[24] T. Instruments, Very Low Noise , 24-Bit Analog-to-Digital Converter D Data Output Rates to 30kSPS ADS1255 (2013).

[25] R. Beach, R. Conlan, M. Godwin, F. Moussy, Towards a Miniature Implantable In Vivo Telemetry Monitoring System Dynamically Configurable as a Potentiostat or Galvanostat for Two- and Three-Electrode Biosensors, IEEE Trans. Instrum. Meas. 54 (1) (2005) 61–72. doi:10.1109/TIM.2004.839757.
URL http://ieeexplore.ieee.org/document/1381799/

[26] Microchip Technology Inc, PIC16F87X (2013).

[27] E. S. Friedman, M. A. Rosenbaum, A. W. Lee, D. A. Lipson, B. R. Land, L. T. Angenent, A cost-effective and field-ready potentiostat that poises subsurface electrodes to monitor bacterial respiration, Biosens. Bioelectron. 32 (1) (2012) 309–313. doi:10.1016/j.bios.2011.12.013.
URL http://dx.doi.org/10.1016/j.bios.2011.12.013http://linkinghub.elsevier.com/retrieve/pii/S0956566311008086

[28] Atmel, ATmega164A/164PA/324A/324PA/644A/644PA/1284/1284P (2015).

[29] C. Mercer, D. Leech, Cost-Effective Wireless Microcontroller for Internet Connectivity of Open-Source Chemical Devices, J. Chem. Educ. 95 (7) (2018) 1221–1225. doi:10.1021/acs.jchemed.8b00200.
URL http://pubs.acs.org/doi/10.1021/acs.jchemed.8b00200