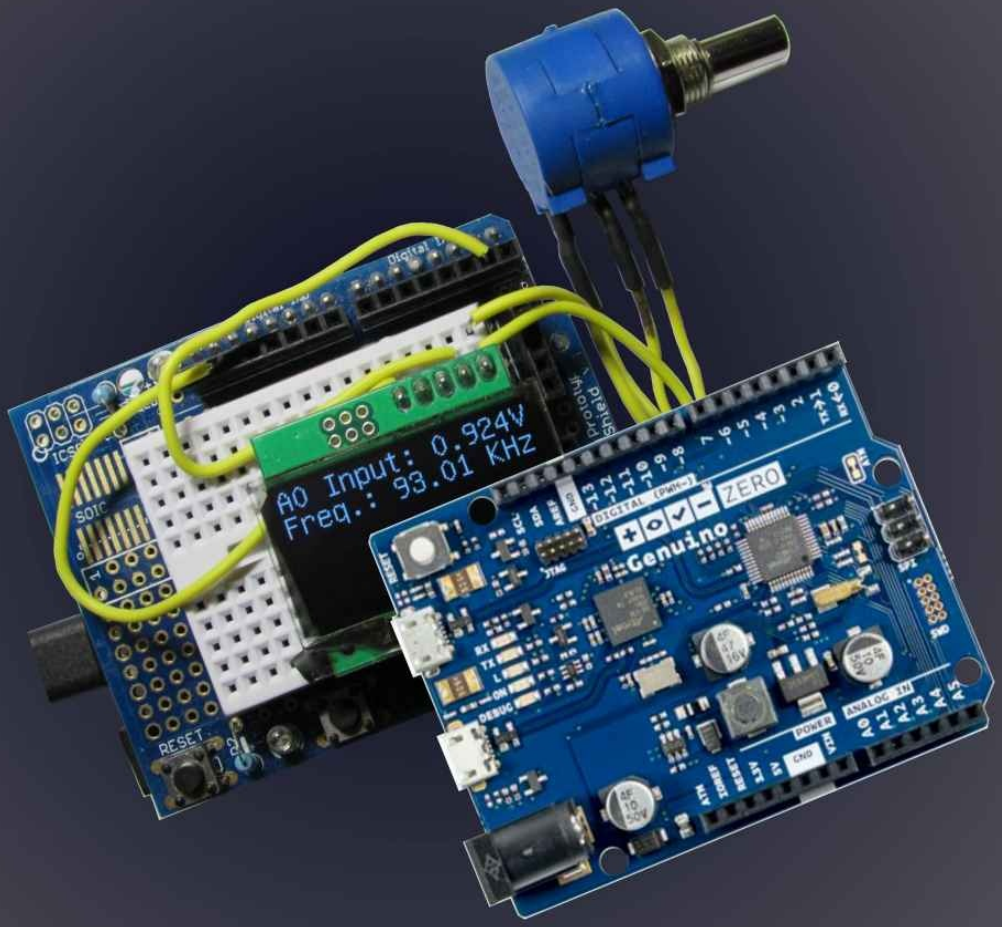
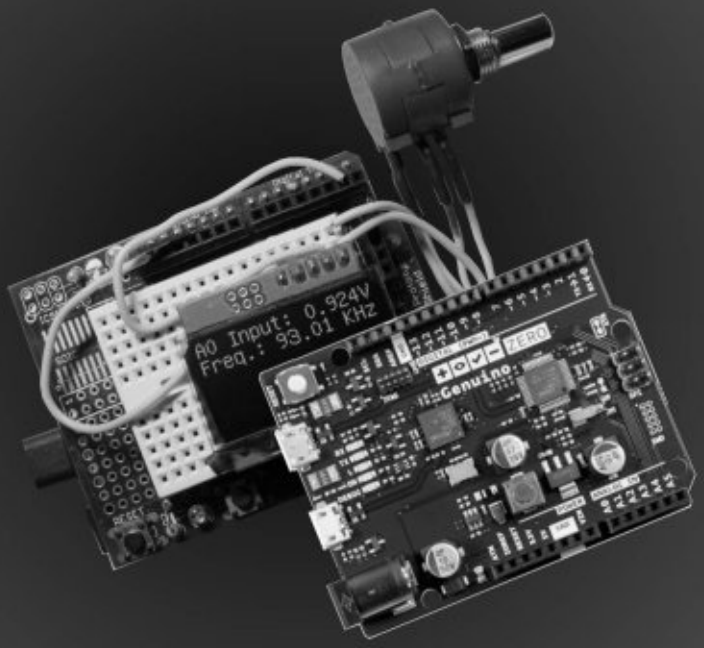
Yury Magda

Arduino Zero Projects Book



Yury Magda

Arduino Zero Projects Book



Arduino Zero

Projects Book

By Yury Magda

Contents

The simple programmable square-wave oscillator

Introduction

Nowadays developers and hobbyists designing various electronic devices and systems have a lot of opportunities to realize their ideas and thoughts. Arduino-compatible boards are undoubtedly the most popular development tools used by hobbyists and professional developers. Numerous measurement and control systems based upon Arduino have already been designed and will be designed in future. This book contains the collection of projects using the new Arduino/Genuino Zero board which represents a simple, yet powerful, 32-bit extension of the Arduino UNO platform. Arduino/Genuino Zero uses Atmel's SAMD21G18 microcontroller featuring a 32-bit ARM Cortex® M0 core that offers excellent potential to create most imaginative and new ideas for high tech automation, wearable technologies, IoT and robotics.

The book is thought as a highly practical guide which can help a developer to build various measurement and control systems based upon using the Arduino/Genuino Zero and Arduino M0 Pro boards. Each project allows to understand various theoretical and practical aspects of design using Arduino/Genuino Zero. The material of the book assumes that the readers are familiar, at least, with basics of designing and assembling electronic circuits. For most projects, having some basic skills in electronics will serve the readers well and allow them to understand what is going on behind the scenes. Each project is accompanied by a brief description which helps to make things clear.

All projects were assembled and tested on the Genuino Zero / Arduino M0 Pro boards. The accompanying source code is simple and explained in detail, so the readers can easily improve it if necessary. To build the demo applications the Arduino 1.6.7 IDE (available on https://www.arduino.cc/en/Main/Software) was used.

Disclaimer

The design techniques described in this book have been tested on both Genuino Zero and Arduino M0 Pro development boards without damage of the equipment. I will not accept any responsibility for damages of any kind due to actions taken by you after reading this book.

Arduino Zero features

In this section we will remeber a few key features that should be taken into account before designing applications with Arduino Zero.

Note that unlike most Arduino and Genuino boards, Arduino Zero runs at 3.3V. The maximum voltage that the I/O pins can tolerate is 3.3V. Applying voltages higher than 3.3V to any I/O pin could damage the board.

In contrast to some Arduino and Genuino boards (e.g. Uno), when you launch the **Serial Monitor** application and the board is connected through the **Programming Port,** the board does not automatically reset, so we need to reset the board manually for running a program code.

The Arduino Zero board can be powered via a USB connector or with an external power supply. The power source is selected automatically.

External (non-USB) power can come either from an AC-to-DC adapter (such as a wall-wart) or battery, and can be connected using a 2.1mm center-positive plug connected to the board's power jack, or directly to the **GND** and **VIN** pin headers of the **POWER** connector.

The board can operate on an external supply of 6 to 20 volts. The recommended range is 7 to 12 volts.

Several projects described in this guide use I^2C interface. Note that Arduino Zero has internal pull-up resistors connected to **SCL** and **SDA** pins, so you can directly connect I^2C-devices to Arduino Zero without inserting additional pull-up resistors.

Compared to Arduino Uno, Zero provides much more capabilities for developers. The Zero has a 32-bit ARM core that can outperform typical 8-bit microcontroller boards. The Central Processing Unit (CPU) is fed by 48 MHz clock source, so the peripheral devices (I/O ports, timers, interrupt controller, etc.) can operate with low-latency thus providing much better performance than 8-bit MCU. This allows to build precision application for measuring signals.

Arduino Zero can process analog signals with 12-bit resolution thus providing more precision measurements. The Arduino Zero board has also 32-bit Real Time Counter (RTC) with clock/calendar function. The digital-to-analog converter (DAC) of Arduino

Zero can provide a 10-bit voltage output on pin **A0**. DAC allows to design circuits generating various waveforms. The DAC output can also be used for designing voltage controlled external circuits (voltage and current sources, VCOs, etc.).

One of most advanced features of Arduino Zero is its capability to operate with a minimal power consumption due to selection of sleep mode. This allows to build low-power and battery-powered applications for autonomous control and measurement systems.

Interfacing Arduino Zero to the PC via an USB-to-UART adapter

Many Arduino Zero applications described in this guide receive commands from the PC and transfer results back to the PC via a serial interface (UART). This section describes how to connect Arduino Zero to the PC using a USB-to-UART adapters. All PCs are equipped with several USB ports, so we can apply one of numerous USB-to-Serial adapters to connect Arduino to the PC. As a USB-to-UART adapter for the projects in this book, we use a device with the FT232RL IC (**Fig.1)**.

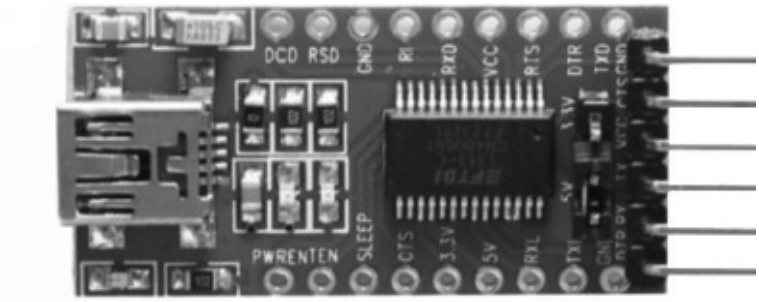


Fig.1

Connections between the Arduino Zero board and the USB-to-UART device are shown in **Fig.2**.

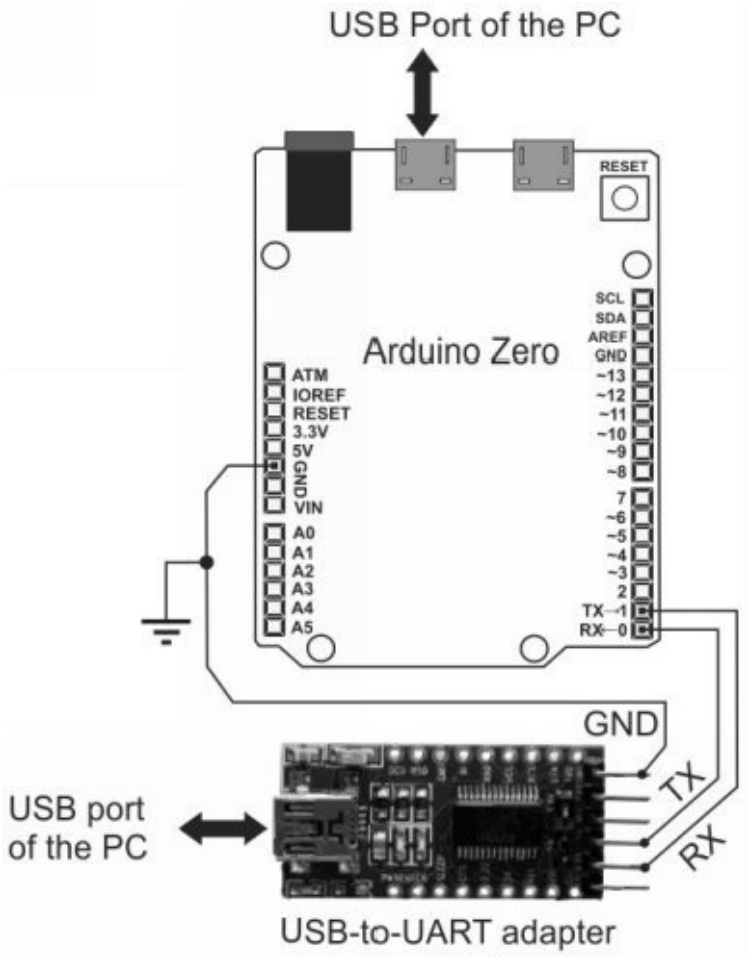


Fig.2

In this circuit, the **TX** and **RX** pins of the USB-to-UART adapter are wired to pins "**RX←0**" and "**TX→1**" respectively.

Note that the signal levels on the TX/RX pins of a USB-to-UART adapter should be the same as those on the corresponding pins of an Arduino Zero board. The digital pins of Arduino Zero operate with maximum voltage of +3.3V, so voltage levels on the USB-to-UART adapter pins must not exceed this value. Usually, USB-to-UART adapters have an on-board switch to select a suitable level (3.3V or 5V). Anyway, ensure that both an external serial interface and Arduino Zero operate with the same logical levels, otherwise you can damage your equipment!

To interact with an Arduino Zero application we can launch some terminal application on the PC. In this guide the popular terminal application **puTTY** will bu used; its configuration window is shown in **Fig.3**.

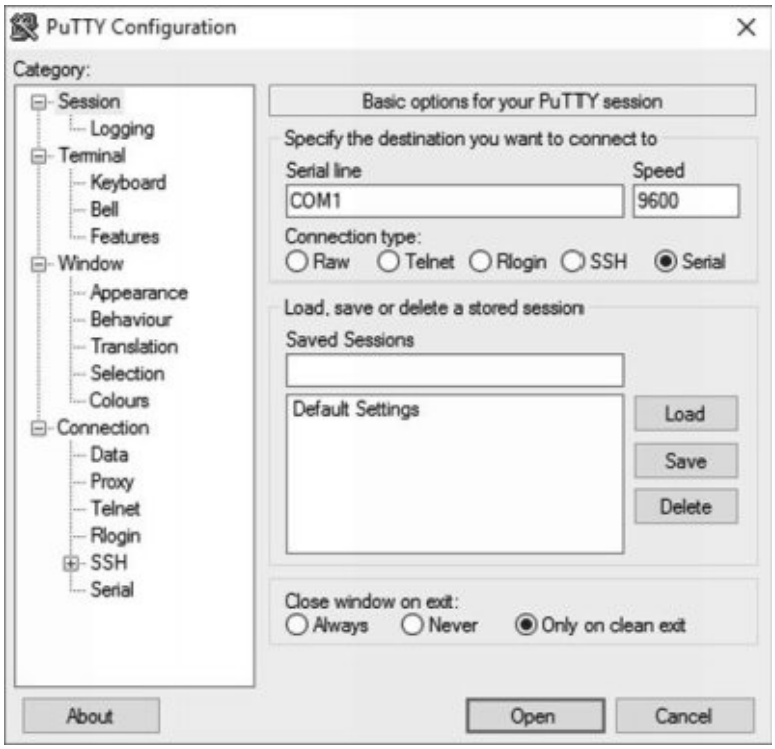


Fig.3

Usuall, we should only reconfigure the "**Serial line**" parameter assigned the "**COM1"** by default and "**Speed**" (**9600** bps by default).

Alternatively, we can develop a Windows application to communicate with Arduino Zero via a serial interface. The following section illustrates how to develop such application in C# .NET by using **Microsoft Visual Studio 2015 Common Edition** development environment.

Developing C# control applications for Arduino Zero

This section describes designing a Windows GUI application that can send commands to the Arduino Zero board and receive data back from this board via a serial interface. This application will be built in **Visual Studio 2015** running in Windows 10.

To build the application we can use the Project Wizard which will create the skeleton of a Windows Forms application in Visual C# .NET (**Fig.4)**.

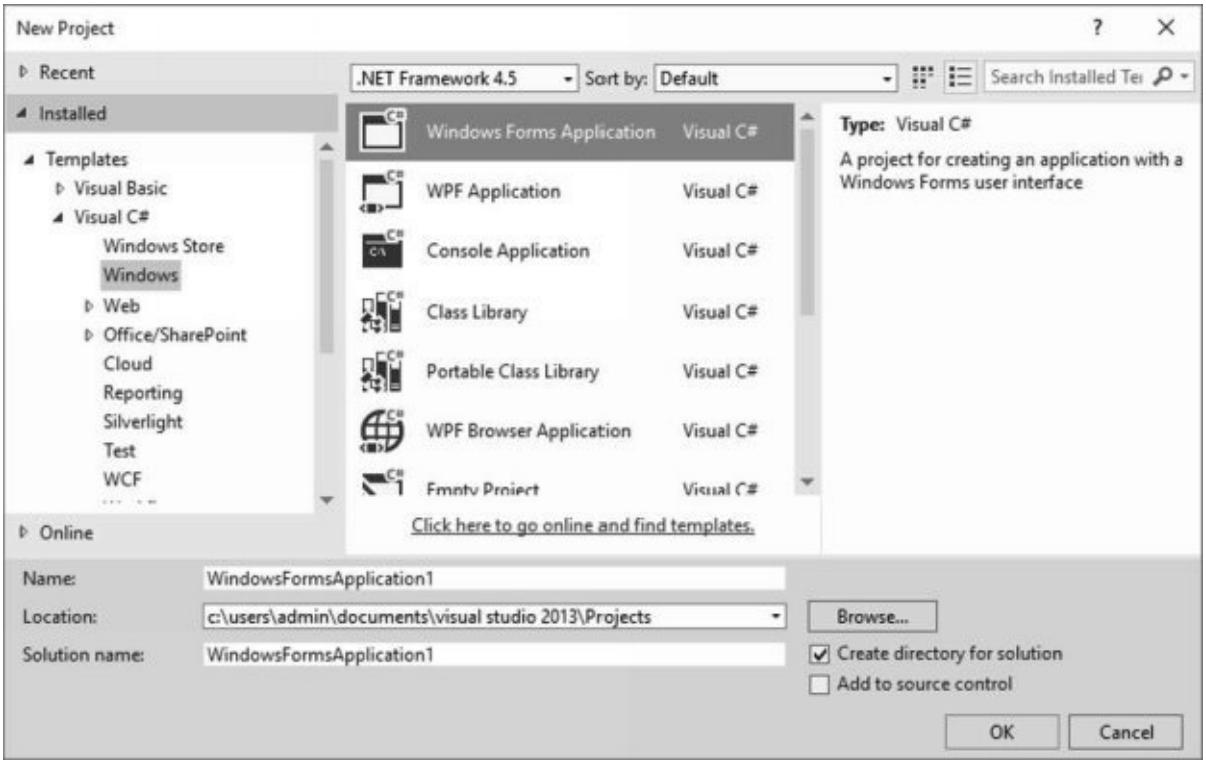


Fig.4

As a result, we get an empty project that needs to be modified. Place several components on the main form of our application (**Fig.5**).

Fig.5

Our application will use the following components from the **Toolbox** palette:

button1 instance of **Button;**

textBox1 instance of **TextBox;**

listBox1 instance of **ListBox;**

serialPort1 instance of **SerialPort**.

When launched, our application should receive the data stream from a serial interface (UART) and output the data received in the **listBox1** window. The data typed in the **textBox1** field can be transferred via UART after pressing **button1** ("**Send data via UART**").

The **serialPort1** component must be configured as is shown in **Fig.6**.

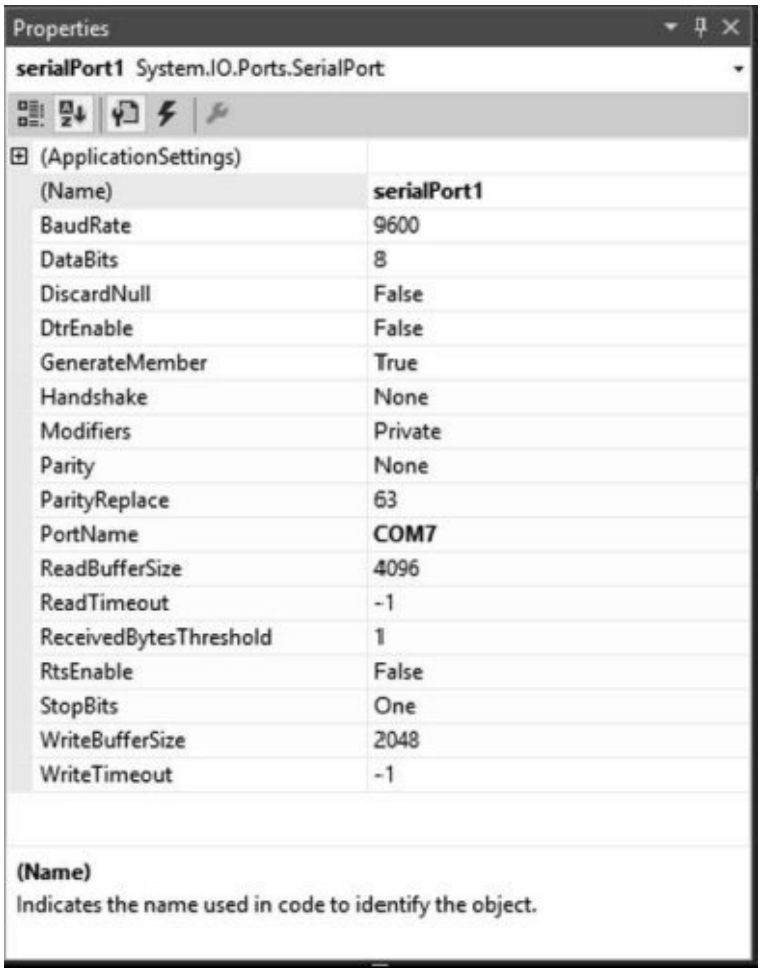


Fig.6

Although **serialPort1** has a lot of properties, we need to configure only several of these. The "**BaudRate"** parameter will be assigned the value of 9600 bps (a commonly used baud rate). The "**PortName"** parameter should be assigned the name of the serial interface being used for communication. In this particular application, there were used the USB-to-UART adapter associated with a virtual serial port **COM7**, so "**PortName**" was also assigned "**COM7**".

After **serialPort1** has been configured, we should write some source code for our application. The data received via UART can be processed by an event handler invoked when a **DataReceived** event occurs (**Fig.7**).

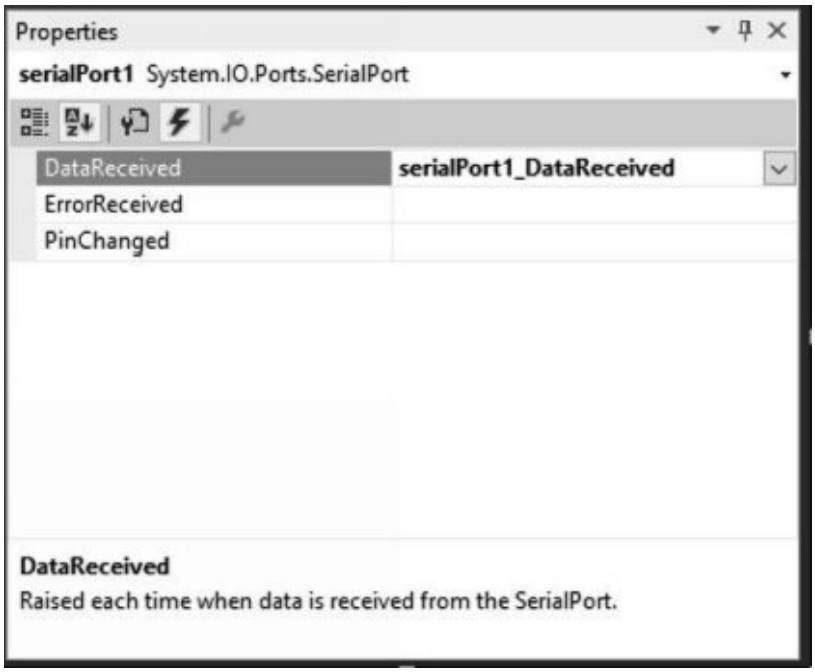


Fig.7

After double-clicking on the editing area of the **DataReceived** field we are brought to the editor window where we should type the source code (**Listing 1**) for the **DataReceived** event handler.

Listing 1.

```
private void serialPort1_DataReceived(object sender,
System.IO.Ports.SerialDataReceivedEventArgs e)
 {
   String s1 = serialPort1.ReadLine();
   listBox1.Items.Add(s1);
 }
```

The first statement within the **seriaPort1_DataReceived** event handler moves the data from the UART buffer to the string **s1**. The statement that follows displays this string in the **listBox1** window.

The serial interface should be opened when the application is launched and closed when the application terminates. The full source code of the application is shown in **Listing 2**.

Listing 2.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace WinForm_UART_TX
{
public partial class Form1 : Form
{
  public Form1()
  {
    InitializeComponent();
    }

private void button1_Click(object sender, EventArgs e)
{
  String s1 = textBox1.Text;
  serialPort1.WriteLine(s1);
}

private void Form1_Load(object sender, EventArgs e)
{
  if (serialPort1.IsOpen == false)
    serialPort1.Open();
}
```

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
  if (serialPort1.IsOpen == true)
    serialPort1.Close();
}


private void serialPort1_DataReceived(object sender,
System.IO.Ports.SerialDataReceivedEventArgs e)
{
  String s1 = serialPort1.ReadLine();
  listBox1.Items.Add(s1);
}
}
}
```

Here the **button1_Click()** event handler is invoked when button "**Send data via UART**" has been pressed. The string typed in the **textBox1** single-line editor is saved in the string **s1.** This string is then transferred via the serial interface by invoking the **WriteLine()** procedure.

The data stream received via the serial interface are processed by the **serialPort1_DataReceived** event handler. The data received is stored in string **s1**, then **s1** is displayed in the **listBox1** window.

We can modify this application by including the additional code for clearing the **listBox1** window (if needed). The Main Form of the modified application is shown in **Fig.8**.

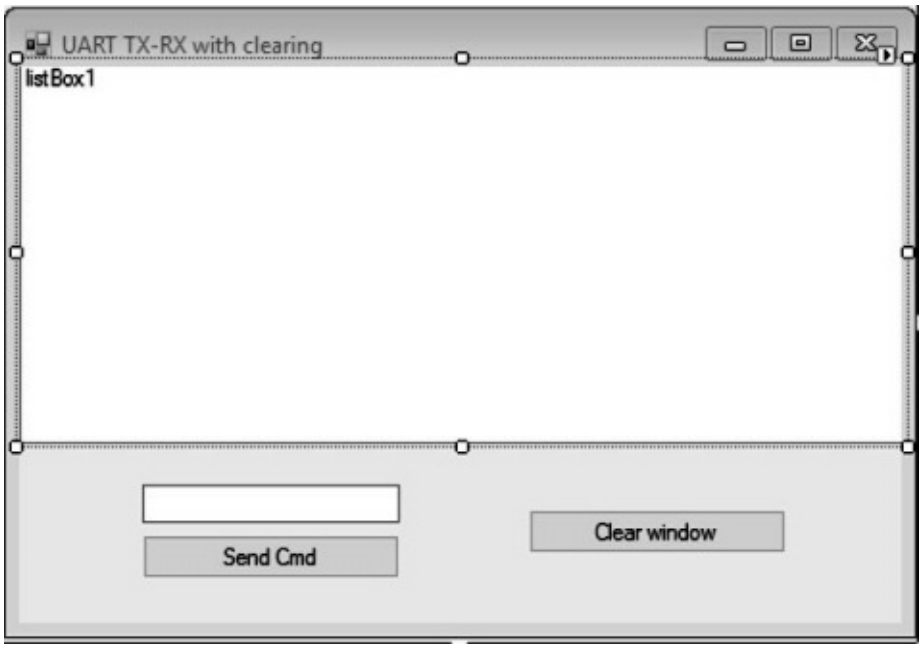


Fig.8

The source code of this application is shown in **Listing 3**.

Listing 3.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace WinApp_UART_RX
{
public partial class Form1 : Form
{
  public Form1()
```

```
  {
    InitializeComponent();
  }

private void Form1_Load(object sender, EventArgs e)
{
  if (serialPort1.IsOpen == false)
    serialPort1.Open();
  }

private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
  if (serialPort1.IsOpen == true)
    serialPort1.Close();
}

private void serialPort1_DataReceived(object sender,
System.IO.Ports.SerialDataReceivedEventArgs e)
{
  String s1 = serialPort1.ReadLine();
  listBox1.Items.Add(s1);
  }

private void button1_Click(object sender, EventArgs e)
{
  listBox1.Items.Clear();
}

private void button2_Click(object sender, EventArgs e)
{
  String s2 = textBox1.Text;
  serialPort1.WriteLine(s2);
```

```
}
}
}
```

Driving external circuitry

This section is dedicated to driving various AC/DC external loads connected to the Arduino Zero outputs.

The ATSAMD21 CPU pins configured as outputs can’t draw heavy loads since the output current through each pin is limited to a few milliamperes (about 7mA). Most external circuits, however, needs much more power, therefore at least a single buffer stage must be inserted after the Arduino Zero outputs. These buffers will prevent the CPU digital outputs from damaging by high currents flowing in high-power external circuitry.

Anyway, it’s bad practice to connect loads directly to CPU pins. Occasionally, even an ordinary LED if damaged may appear as a short-circuit load for an output pin. **Note** that even small currents flowing across pins may cause overheating the microprocessor thus leading to gradual degradation and destruction of the chip.

Buffer outputs themselves can draw relatively high currents, so you can use a suitable DC power supply for powering your circuitry.

Several buffer circuits are presented in the following sections.

DC loads

The simple buffer circuits for a digital output pin can be arranged as the power switches using a power bipolar or MOSFET transistor (**Fig.9 – Fig.10)**.

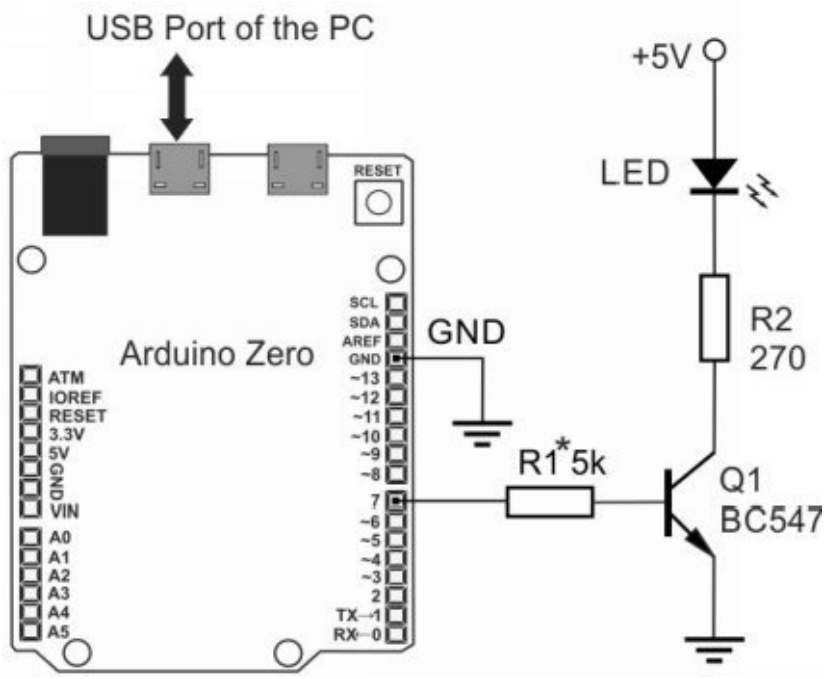


Fig.9

In the above circuit, the external load (LED) is driven by pin **7** via the small-signal bipolar transistor Q1. Q1 acts as a switch that opens when pin **7** is pulled HIGH (log.”1”, close to 3.3V) and closes when pin **7** is pulled LOW (log.”0”, about 0 V). This configuration is suitable for loads drawing the current from a few to tens milliamperes. Almost any small-signal general purpose bipolar transistor (2N2222, 2N4904, 2N4401, etc.) will fit this circuit. The value of R1 may be from 5k to 10k.

When a load draws a relatively high current, we should apply the power bipolar transistor with a large current gain (β). This may be some Darlington device, for example, TIP122, TIP142, etc.

The buffer stage can also be built around a power MOSFET device (**Fig.10**). This circuit uses the power MOSFET IRF510, although almost any of similar devices (IRF520, IRFZ24, etc.) can be taken as well. **Note** that general purpose power MOSFETs reliably operate at relatively low frequencies, up to a few KHz.

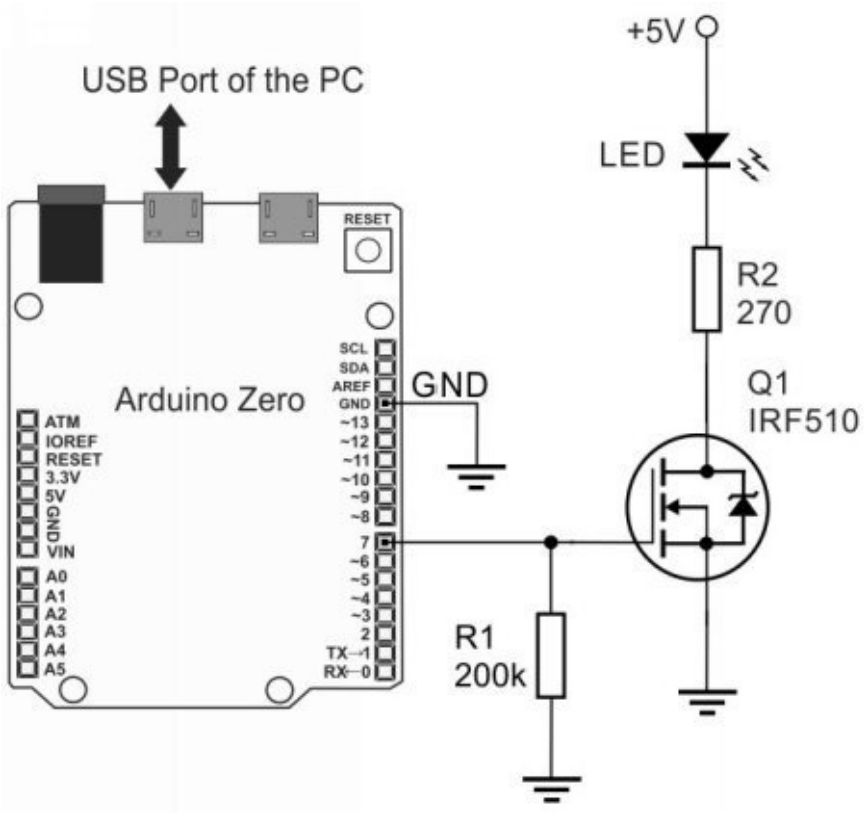


Fig.10

A special case is when a buffer circuit drives inductive load (relay, motor, etc.). In this circuit, a control device must be protected by the switching diode across the load as is shown in **Fig.11.**

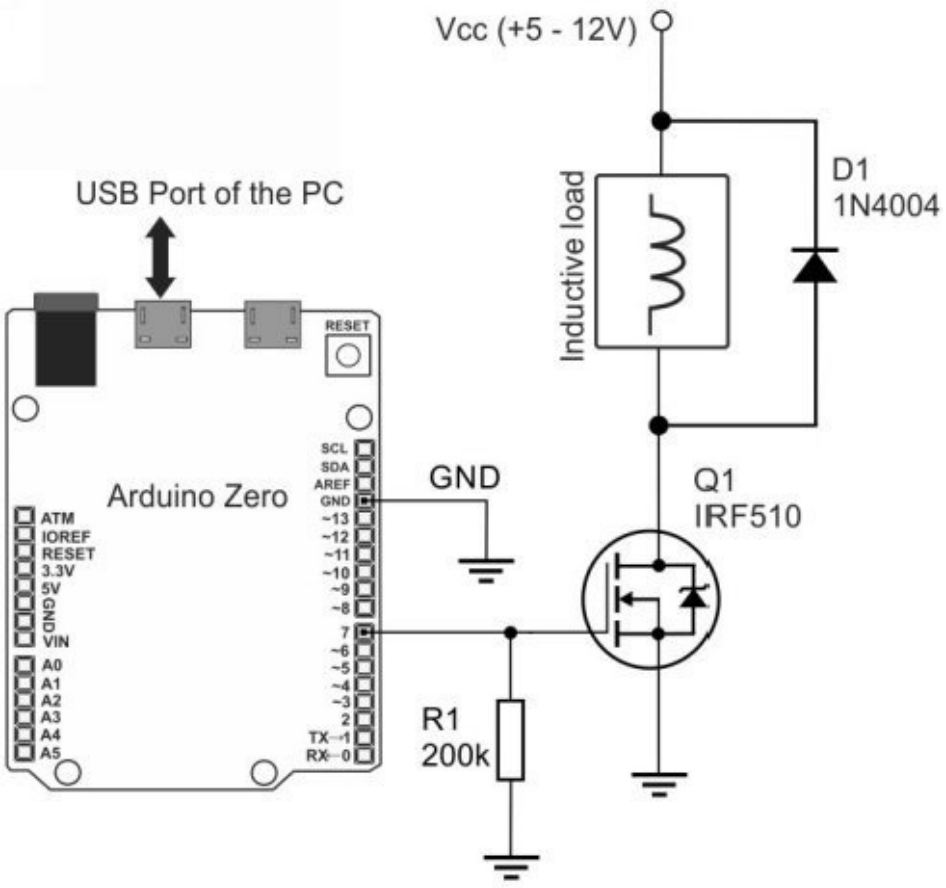


Fig.11

Alternatively, inductive loads can be driven by a power bipolar transistor **(Fig.12)**.

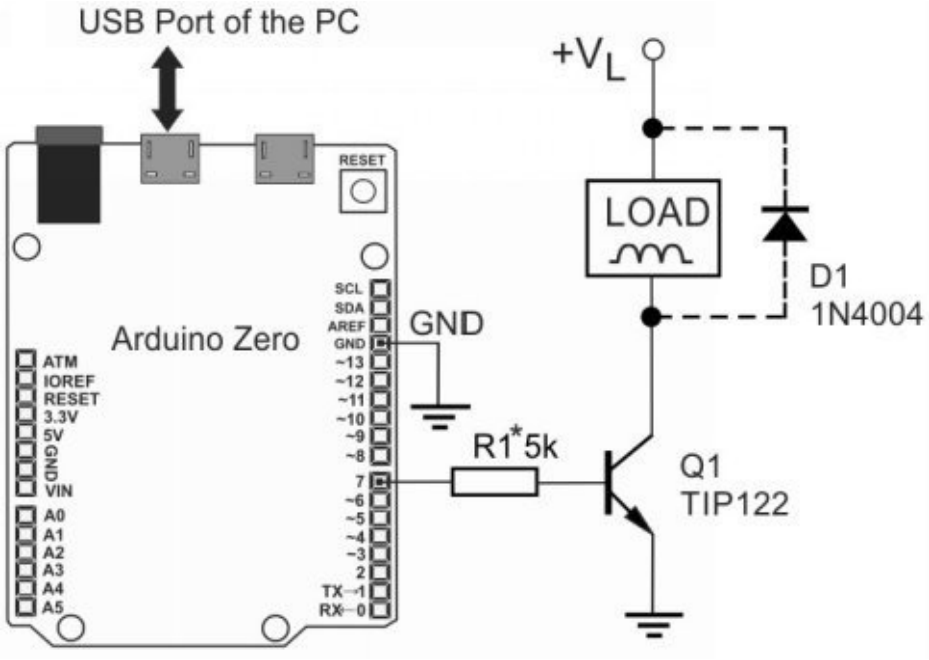


Fig.12

This circuit uses the power switch with a Darlington transistor (Q1). Transistor TIP122 is driven ON when the digital output **7** is pulled HIGH. In this case, the DC power supply $+V_L$ provides the current to the LOAD through the opened transistor Q1. Otherwise, when pin **7** goes LOW(log. ″ 0 ″), Q1 is cut off, thereby breaking the power chain to the LOAD. Diode D1 (noted in the dashed line) protects the power transistor Q1 during the ON/OFF transition.

With this circuit some degree of noise may appear on the ″ ground ″ wire while switching inductive loads ON/OFF. This can be ignored if there are no sensitive conditioning circuits such as analog-to-digital converters.

The best way to prevent flowing high currents through a common wire and, therefore, reduce noise is to isolate low-power circuits on the Arduino Zero side from high–power loads with optoisolated circuits.

One possible configuration shown in **Fig.13** uses an optoisolator 4N35 (4N36, 4N37 can be taken as well).

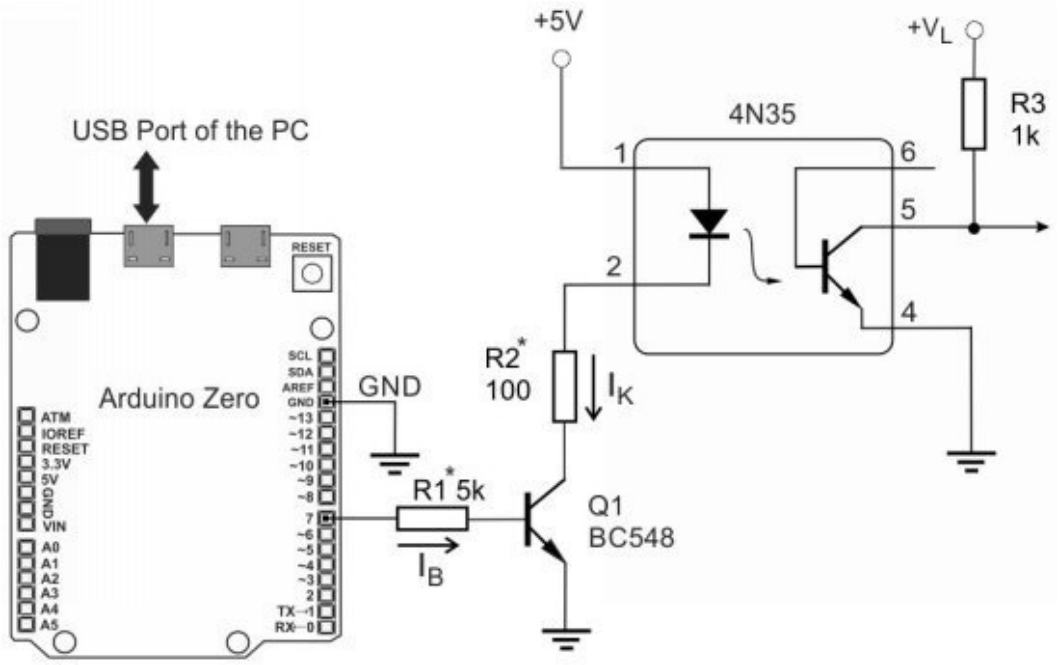


Fig.13

In this circuit, the IR LED inside optoisolator 4N35 is driven ON/OFF by the bipolar transistor Q1 (BC548). Q1 itself is controlled through the base by the voltage on the digital output **7**. The base current I_B (in mA) will approximately be calculated as follows:

(3.3 – 0.6) / R1

where R1 is in KOhms. The base current I_B can be very small, so the output pin pin will not be overloaded. Given the typical β of Q1, it is easily to calculate the collector current I_K flowing through the LED of the optoisolator using the following formula:

$I_K \approx \beta \times I_B$

Devices 4N35 – 4N37 are relatively slow and can't reliably operate at higher frequencies. At the frequencies of a few tens KHz we need to apply high speed optoisolators such as 6N137, HCPL2601, etc. The circuit diagram with 6N3137 (f = 10 MBd) is shown in **Fig.14**.

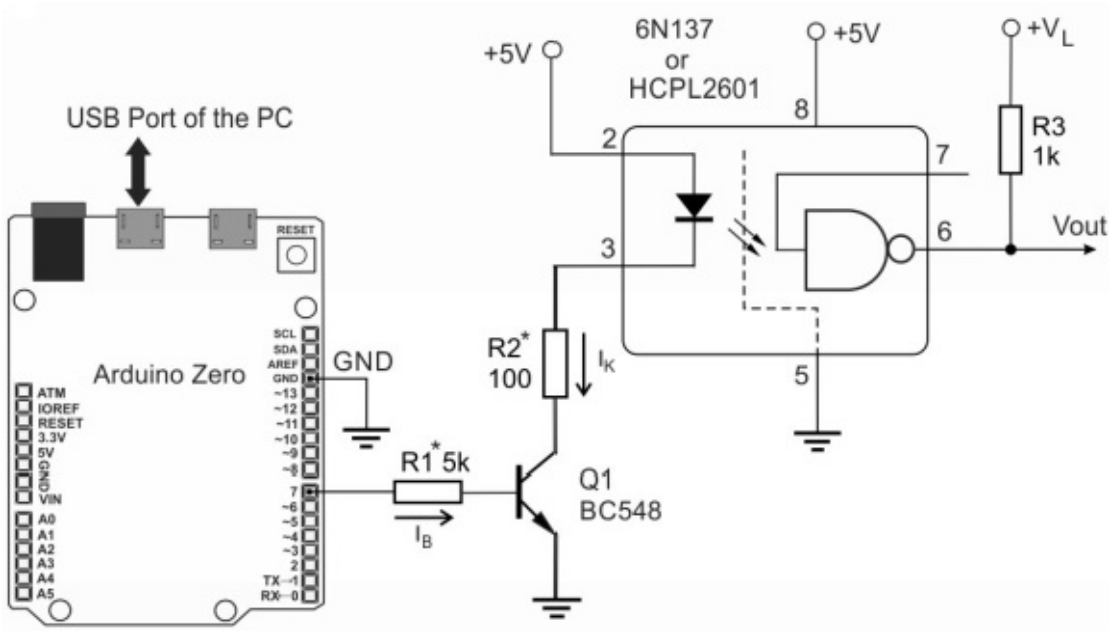


Fig.14

These optoisolators provide an open collector (OK) output that requires a pull-up resistor (R3) to be connected to pin **6**. For driving high-current DC loads we should apply an additional power switch driven by the optoisolator output pin. **Fig.15** illustrates the case when the power switch with MOSFET IRF510 is used.

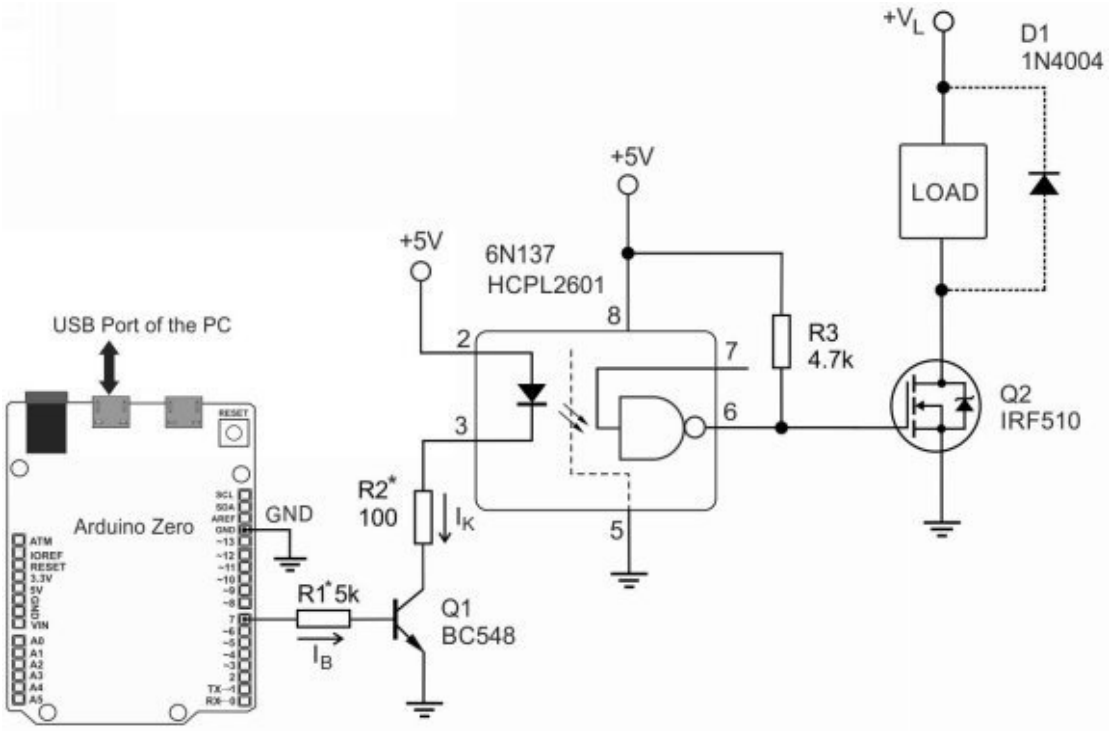


Fig.15

The above circuit can use a variety of components. Q1 (BC548) may be replaced by any general-purpose small-signal bipolar transistor (BC547, BC549, 2N3904, 2N4401, etc.). The power MOSFET IRF510 (Q2) can be replaced by almost any similar device (IRF530, IRFZ24, etc.) with suitable parameters (drain current, gate-source and drain-source voltages).

When choosing a power MOSFET, take into account the maximum power dissipation that is determined by load and the duty cycle of a signal on pin **7** of Arduino Zero.

AC loads

AC loads are usually driven by mechanical or solid state relays (SSR). Nowadays SSR devices are preferable because of their reliability and high speed. This section is dedicated to interfacing zero-crossing solid state relays to AC loads.

Such SSRs are also known as a “synchronous” solid state relays; this is the most common type of SSR found in the market today. As the name implies, the switching of such relay from a non – conducting to a conducting state occurs when the AC mains voltage reaches the zero-crossing point of the sine wave. This minimizes the surge current through a load during the first conduction cycle and helps reduce the level of conducted emissions placed on the AC mains.

An ordinary solid state relay usually includes an IR LED network optically coupled to the power switch. LED can be driven by the low-power DC control circuitry with output control voltage from 3 to 32V.

The power switch of a zero-crossing SSR is arranged around a triac with its control circuitry. The triac is driven by a zero-crossing detector circuit that ensures reliable operation as an AC wave crosses zero.

A typical SSR device has four terminals. Those marked with “ **–** “ and “+” must be wired to a low-power DC control circuit; the terminals marked “~” must be wired to AC high-power loads.

The next two circuit diagrams (**Fig.16** – **Fig.17**) show typical connections for SSR **Fotek SSR-25DA** driven by Arduino Zero; most SSRs are wired much the same way as is shown below.

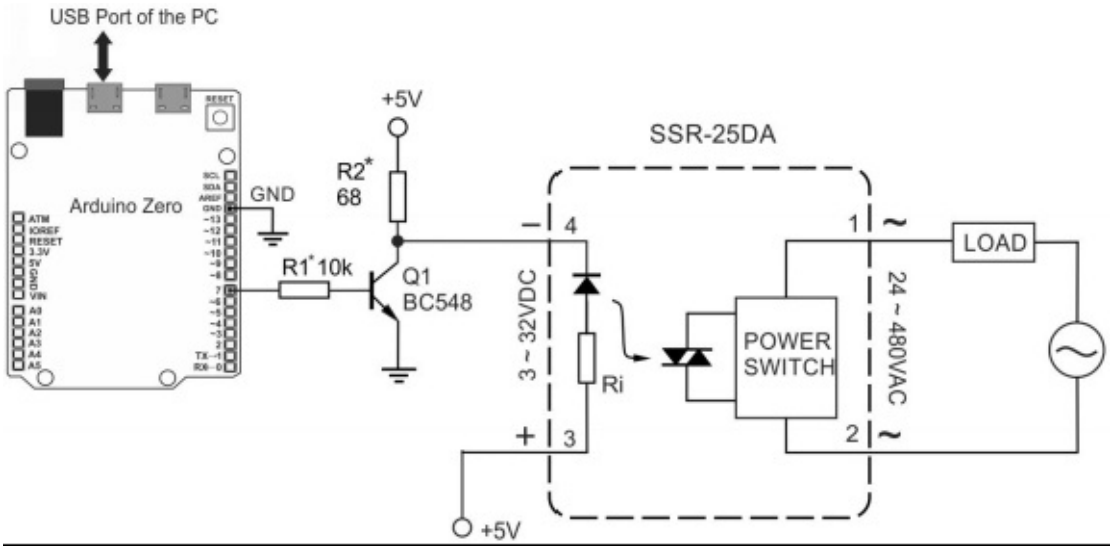


Fig.16

The above figure illustrates the case when SSR is driven ON by the high level on pin **7** of Arduino Zero. The high level applied to the left end of resistor R1 drives transistor Q1 ON and the control current begins to flow from the +5V DC source through the terminal “+” of SSR and the Ri – LED inner network. This causes LED to emit visible light or infrared radiation that, in turn, drives the power switch ON.

Conversely, when pin **7** of Arduino Zero is pulled LOW, transistor Q1 is cut off thus preventing the current flow through the Ri – LED network. As a result, the power switch is driven OFF at the next zero-crossing point of an AC sine wave.

The control circuit shown in **Fig.17** drives SSR ON by the low level on pin **7**. In this case, transistor Q1 is cut off and the control current flows through the LED – Ri network from +5V DC source to “ground”. Conversely, the high level on pin **7** opens transistor Q1, so that the LED – Ri network is shorted. As a result, the power switch breaks the LOAD.

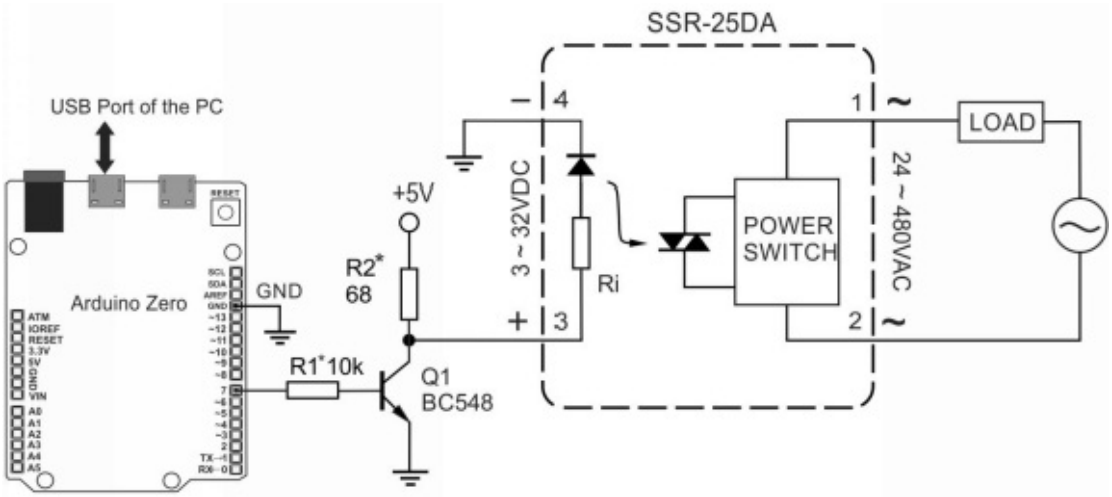


Fig.17

Almost any small-signal general purpose bipolar transistor with a maximum collector current of a few ten milliamperes will work in these circuits.

Interfacing digital input signals

There may be a variety of sources providing digital signals. The methods of interfacing these sources to Arduino Zero highly depends on the electrical parameters (output impedance, slew rate, frequency, etc.) of the circuits producing digital signals. When dealing with mechanical switches, we can apply the simplest possible circuit shown in **Fig.18**.

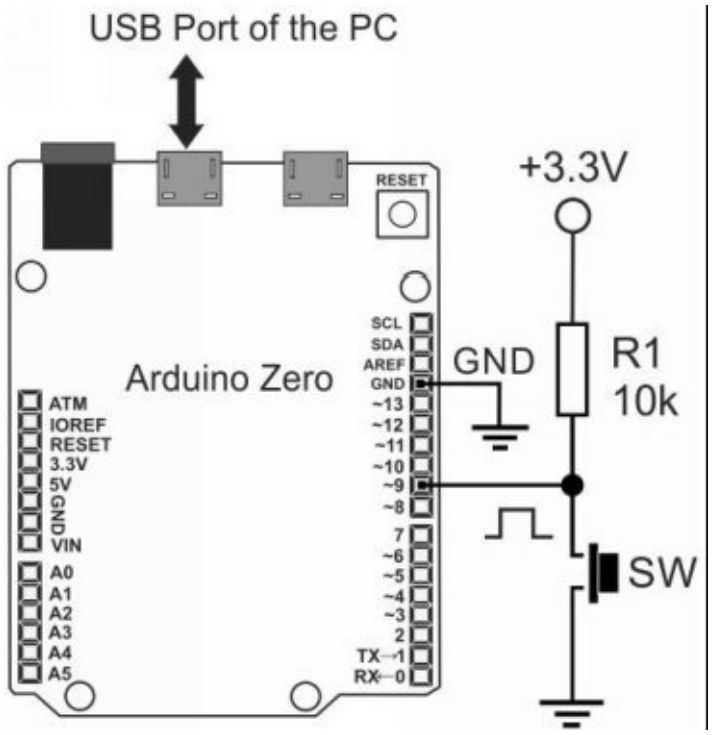


Fig.18

In this circuit pin **9** of Arduino Zero configured as input is connected to switch **SW**. When the switch is open, the voltage level on input **9** stays HIGH (log.”1”). Pressing **SW** causes the voltage level on pin **9** to drop down to 0 V (log.”0”). In the event of the voltage swing “0-1” or “1-0” the application will run the corresponding section of program code.

However, this simple ciruit poses a problem. Switch **SW** when toggled causes the contact bounce on an input pin. The reason for concern is that the time it takes for contacts to stop bouncing is measured in milliseconds, while digital circuits can respond in microseconds. There are several methods to solve the problem of contact bounce (that is, to ″ de-bounce ″ the input signals). One possible approach is shown in **Fig.19**.

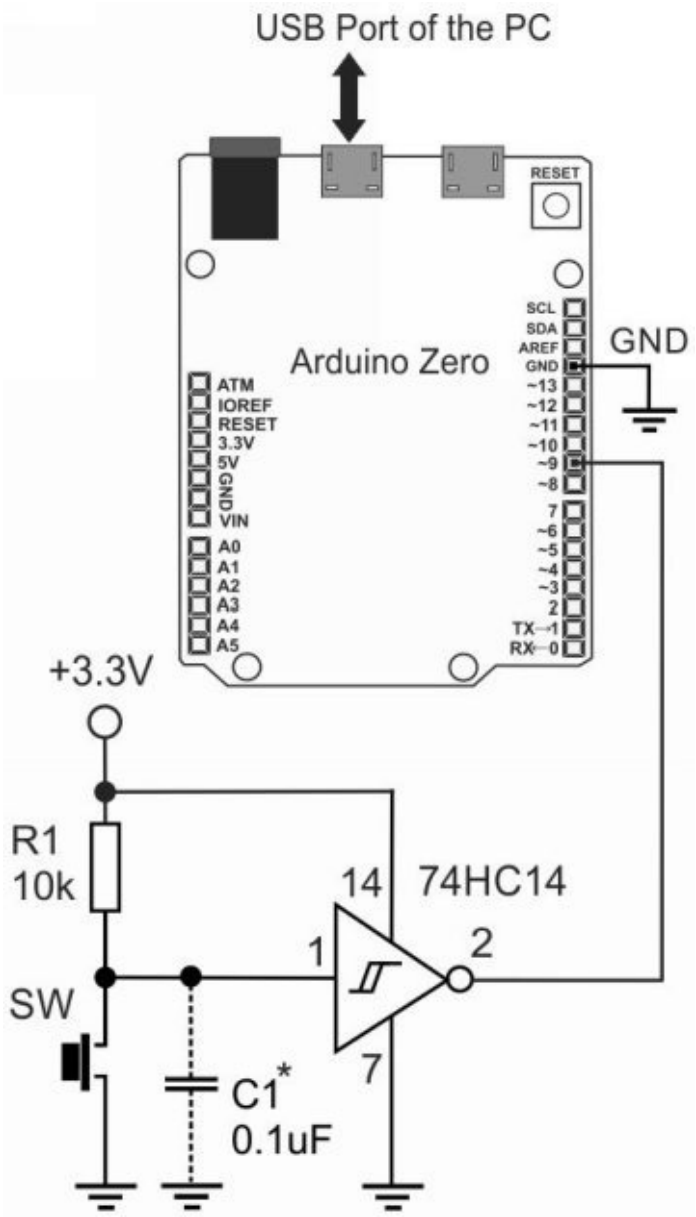


Fig.19

Here is a simple hardware debounce circuit for a momentary push-button switch **SW**. The circuit includes resistor R2 and capacitor C1 (dashed line in **Fig.19**). The R2×C1 time constant is approximately = 0.1 s (typical value) to swamp out the bounce. Usually, we can pick the values of R2 and C1 so that the time constant R2×C1 turns out to be longer than the expected bounce time. The Schmitt trigger of 74HC14 connected to the switch network provides sharp HIGH-to-LOW transition. With such a debounce circuit, we should wait before pressing the switch again. If the switch has been pressed too soon, it will not generate another signal.

Let’s consider one more case. Often we only need to know if any of several sensors is active. In this case, we can use a simple circuit shown in **Fig.20**.

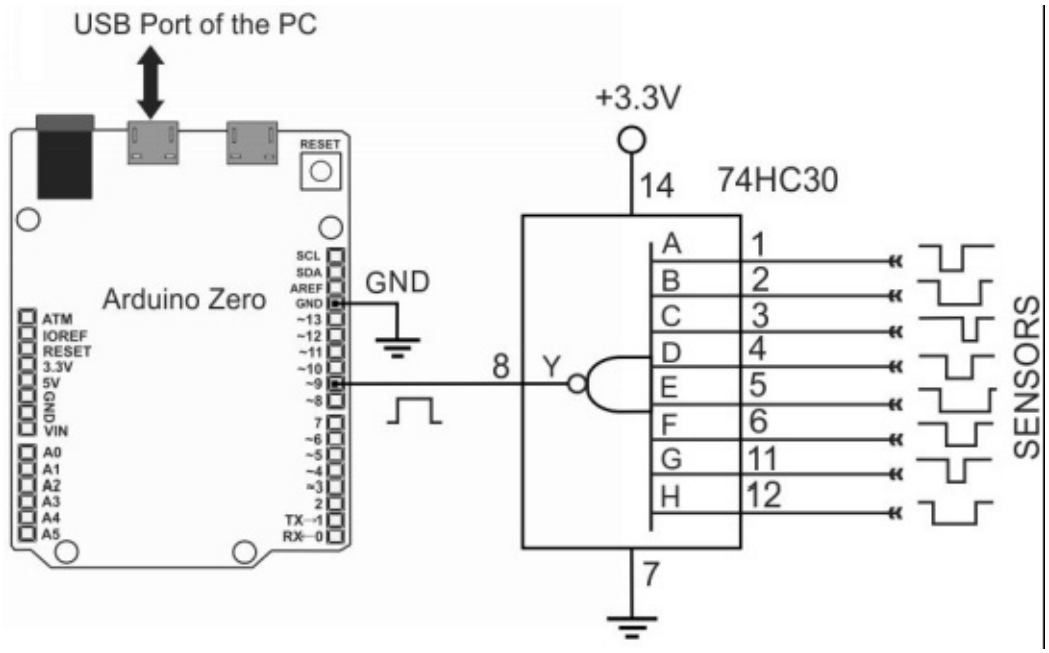


Fig.20

In this circuit, an 8-input NAND gate 74HC30 gatheres the signals from digital sensors. If any of inputs **A – H** is brought LOW, the output **Y** goes HIGH. Output **Y** stays LOW only when all inputs are HIGH.

To add more input channels to our application we can use a multiplexer IC 74HC4051. This device allows to route multiple input signals through its single output; the input channel to be processed is selected programmatically through 3 digital inputs.

The following circuit (**Fig.21**) illustrates the use of an 8-channel multiplexer 74HC4051 for processing digital input signals.

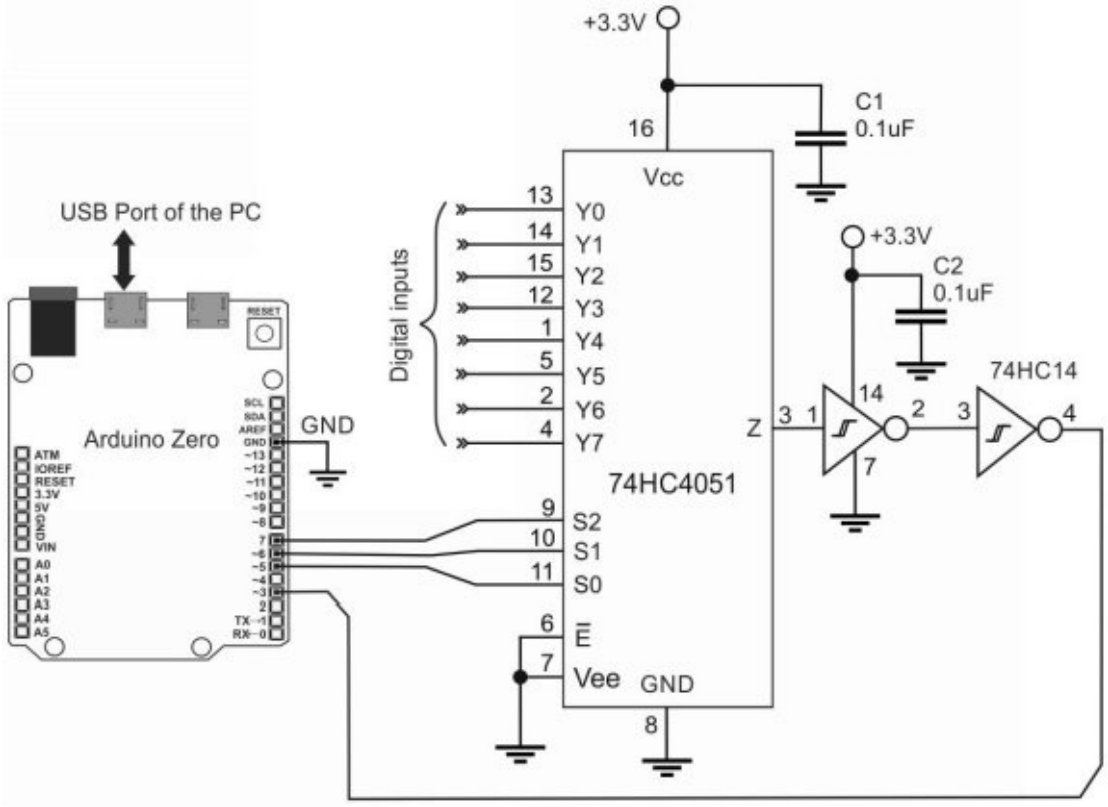


Fig.21

In this circuit, the digital signals are brought to inputs **Y0** – **Y7** of 74HC4051. The input channel to be read (**0** through **7**) is selected by applying the binary weighted code to pins **S0** – **S2** of the multiplexer. Signals to pins **S0** – **S2** lines come from digital outputs **5** – **7** of Arduino Zero.

The signal from the selected input line of the multiplexer passes through output **Z** (pin **3** of 4051) and buffer 74HC14 to pin **3** of Arduino Zero.

For example, to pick up the digital signal on the input line **Y2** program code should bring the binary code 0x2 on lines **S0** – **S2** prior to reading the **Z** output. In this case, pins **5** – **7** must be set as follows:

Pin 5 = LOW

Pin 6 = HIGH

Pin 7 = LOW

For reading the input **Y5**, for example, the following combination will be valid:

Pin 5 = HIGH

Pin 6 = LOW

Pin 7 = HIGH

The Schmitt triggers of 74HC14 transforms slowly changing input signals into sharply defined, jitter-free output signals.

The ″ ground ″ wire of the circuit should be tied to one of the ″ ground ″ pins of Arduino Zero; the bypass capacitors C1 and C2 should be tied closely to the power pins of the chips.

The table below specifies connections between the Arduino Zero board and the external circuitry.

Arduino Zero pin	**External line**
3	Pin 4 of 74HC14 (signal from the Z output of the multiplexer)
5	Pin 11 of 74HC4051 (S0)
6	Pin 10 of 74HC4051 (S1)
7	Pin 9 0f 74HC4051 (S2)

Processing isolated digital input signals

This section is dedicated to processing signals being passed through electrically isolated external circuitry. Optoisolator circuits are used when reading digital input signals from high-power and/or noisy circuits. We have already discussed such circuits, so let’s look at how to interface optoisolators to digital sensors.

Although most optoisolator devices are rated for operating at +5V, they can operate at +3.3V simplifies interfacing optoisolators to the Arduino Zero inputs.

The following circuit diagrams (**Fig.22** – **Fig.23**) illustrate several possible ways of interfacing digital sensors to Arduino Zero inputs via optoisolators.

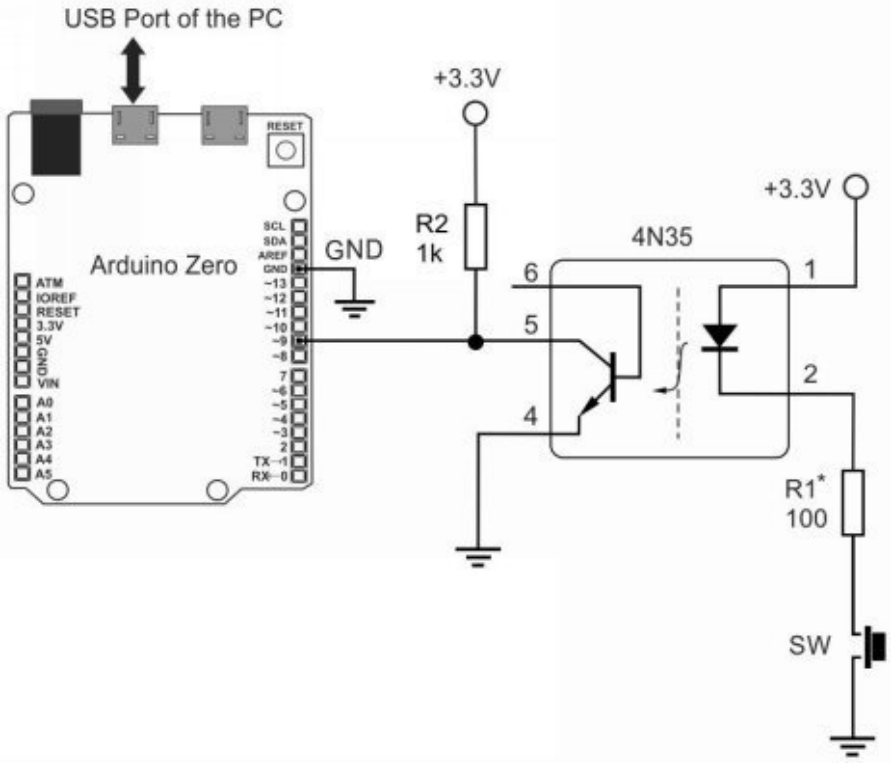


Fig.22

In this circuit, the digital sensor is simulated by the mechanical switch **SW**. When **SW** is pressed, the LED is ON and the signal on pin **5** of 4N35 optoisolator goes LOW. When **SW** is released, pin **5** goes high. The state of pin **5** of 4N35 optoisolator can be read on pin **9** of Arduino Zero.

Instead of the 4N35 device, we can also apply the high-speed optoisolator 6N137 or HCPL2601 as is shown in **Fig.23**.

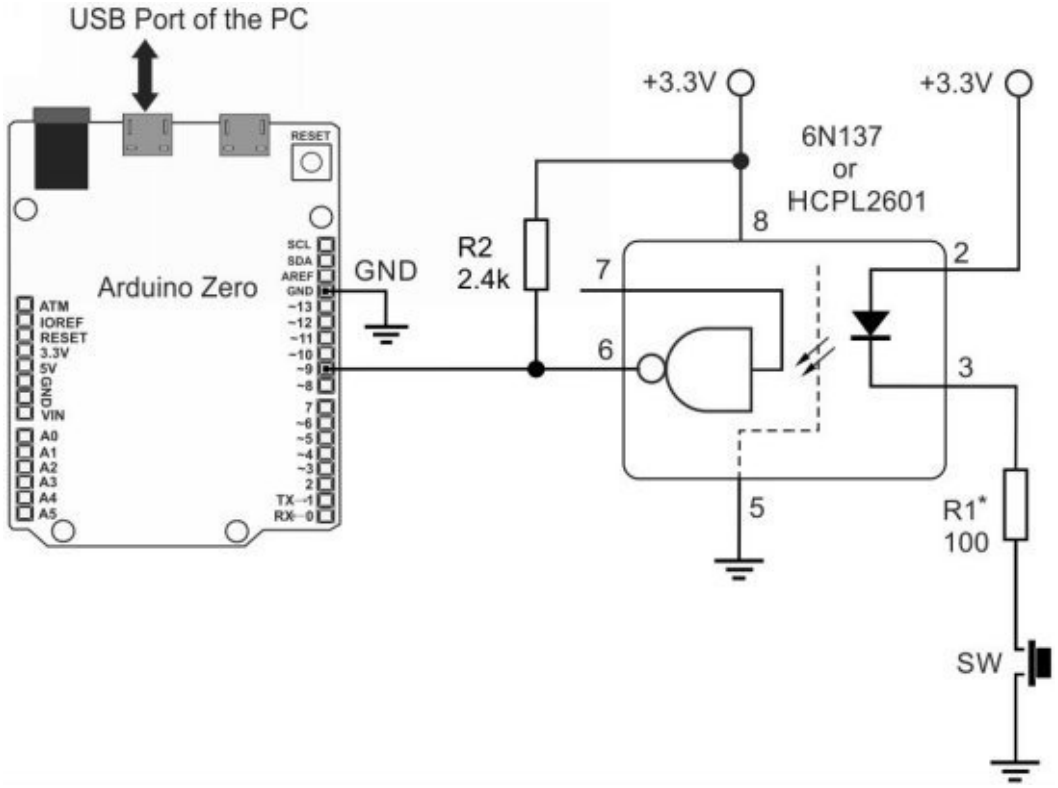

USB Port of the PC
Arduino Zero
RESET
SCL
SDA
AREF
GND
~13
~12
~11
~10
~9
~8
7
~6
~5
~4
~3
2
TX→1
RX←0
ATM
IOREF
RESET
3.3V
5V
GND
GND
VIN
A0
A1
A2
A3
A4
A5
GND
R2
2.4k
+3.3V
+3.3V
6N137
or
HCPL2601
8
7
6
5
2
3
R1*
100
SW

Fig.23

Processing distorted input signals

Often a system should process input pulses whose pulse width and/or slew rate randomly vary. When the pulse width of an input signal is very short (a few microseconds or less), the Arduino Zero application can’t reliably capture such a signal and process it in a timely manner. In those cases it would be worth to lengthen the pulse width of an input signal to a few ten or hundred microseconds so that a system could process the pulse (**Fig.24)**.

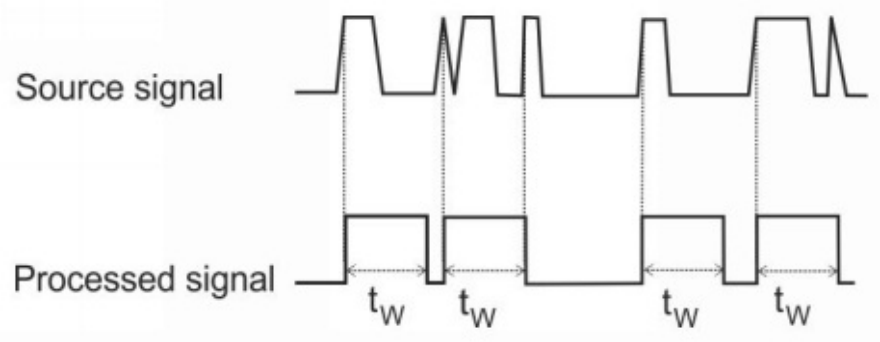


Fig.24

In the above diagram, the sequence of pulses with various widths and edges is converted into one with some predetermined pulse width $\mathbf{t_W}$ and sharp edges. The pulse width $\mathbf{t_W}$ should be selected long enough so that an application can reliably process an input signal.

Such conversion may be implemented using a simple circuit known as a “monostable multivibrator”. One possible circuit using a versatile chip 74HC4538 is shown in **Fig.25**.

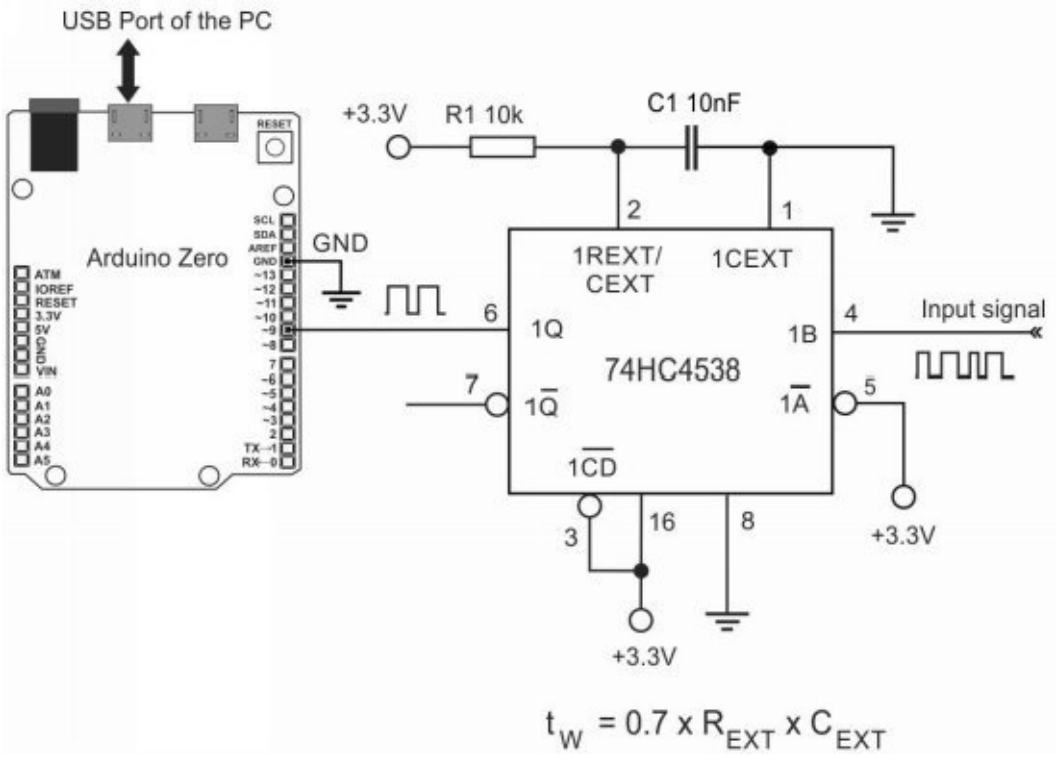


Fig.25

The 74HC4538 device is a dual retriggerable-resettable monostable multivibrator. Each multivibrator has an active LOW trigger/retrigger input (**nA**), an active HIGH trigger/retrigger input (**nB**), an overriding active LOW direct reset input (**nCD**), an output (**nQ**) and its complement, and two pins (**nREXT/CEXT** and **nCEXT**) for connecting the external timing components $\mathbf{C_{EXT}}$ and $\mathbf{R_{EXT}}$. Typical pulse width variation over the specified temperature range is ±0.2 %.

The multivibrator may be triggered by either the positive or negative edge of an input pulse. The duration and accuracy of an output pulse are determined by the external timing components $\mathbf{C_{EXT}}$ and $\mathbf{R_{EXT}}$.

The output pulse width $\mathbf{t_W}$ will be equal to

$0.7 \times R_{EXT} \times C_{EXT}$

Given $\mathbf{R_{EXT}}$ and $\mathbf{C_{EXT}}$, the pulse width $\mathbf{t_W}$ will be 70 microseconds (70 µS).

The linear design techniques guarantee precise control of the output pulse width. A LOW level on **nCD** terminates the output pulse immediately. Schmitt trigger action on pins **nA** and **nB** makes the circuit highly tolerant of slower rise and fall times.

Alternatively, we can build the circuit with the 74HC123 chip (**Fig.26**).

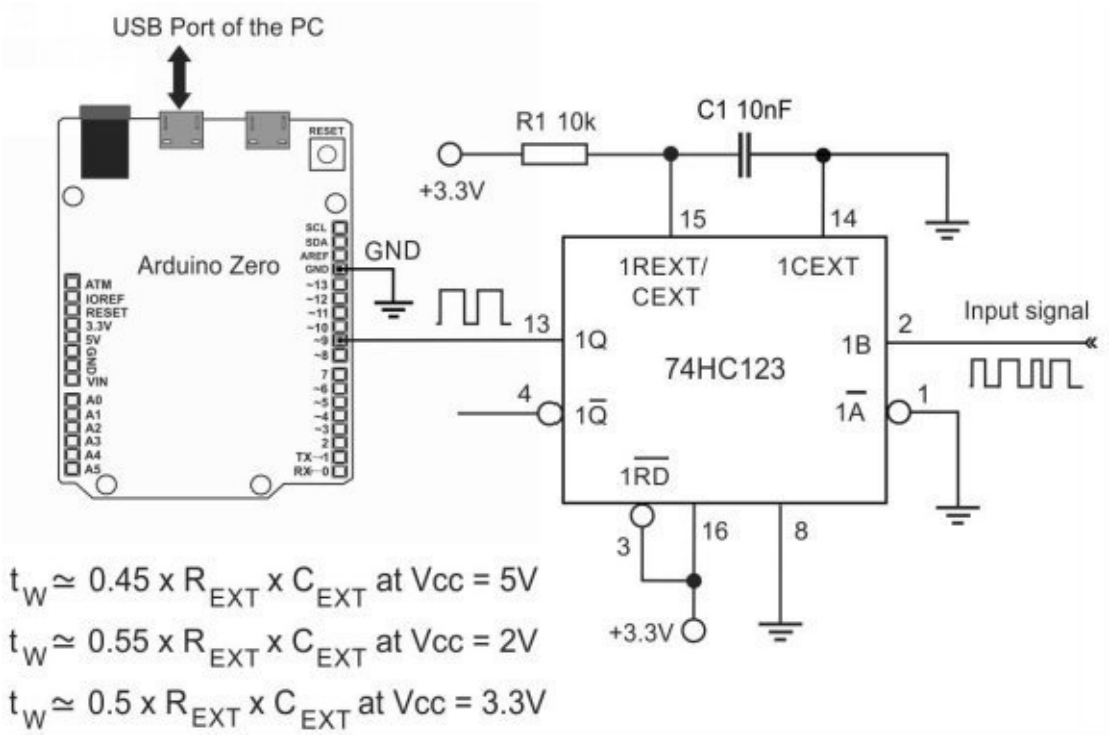


Fig.26

You can easily calculate the pulse width of an output signal on pin **13** of 74HC123 using the formulas on the left side of the above figure.

Processing non-digital input signals

For processing analog signals in digital domain we need to convert such signals into the sequence of TTL-compatible digital pulses. The simple circuit arranged around an analog comparator IC can help to do this (**Fig.27**).

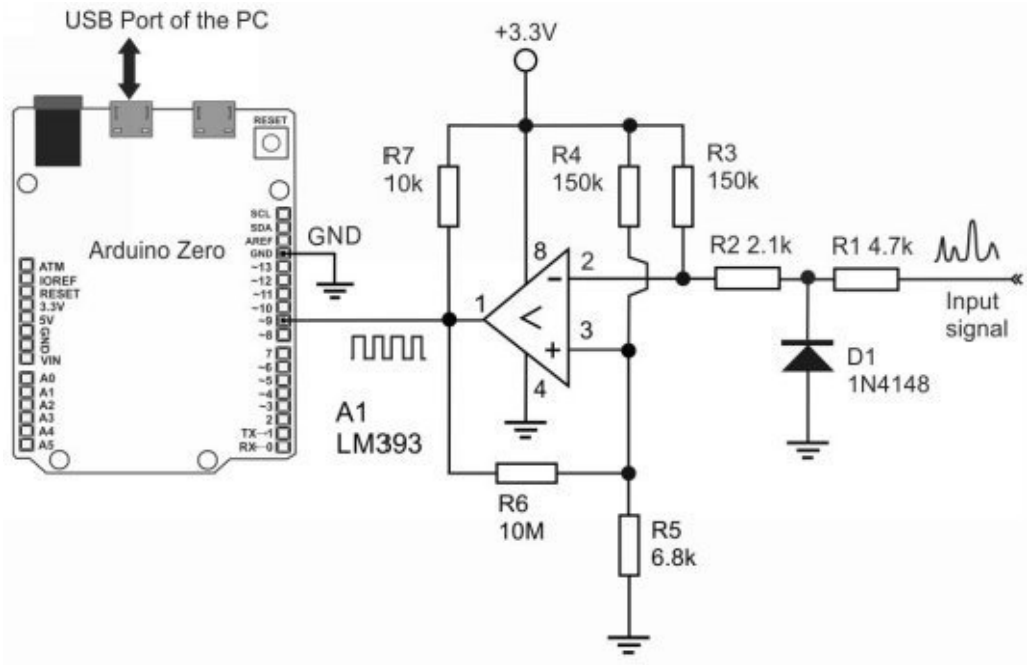


Fig.27

This circuit is arranged as the zero-crossing detector with a general-purpose comparator LM393. The threshold on the non-inverting input of LM393 is adjusted by the resistive divider R4 – R5; diode D1 clips the signal when it goes negative. The LM393 can be replaced by any general-purpose comparator (LMC7221, LMC7211, etc.).

Interfacing digital input devices: Capacitive Touch Keypad

This section is dedicated to use of the SparkFun Capacitive Touch Keypad with MPR121Q2 IC in Arduino Zero applications. Such a keypad comes in handy when an application needs to provide a user input.

The MPR121QR2 itself is a capacitive touch sensor controller that makes it very easy to integrate capacitive touch sensing into your project. It communicates via I^2C, and works by measuring the capacitance of twelve electrode points. When an object comes close to the electrode connector, the measured capacitance changes. This signals the MPR121 that something has touched a “button”. The IC is also capable of driving LEDs or basic GPIO functionality on electrode pins 4 through 11, giving you a lot of freedom for setting up your project. The sensor works from 1.6V to 3.3V. The sensor isn’t very current-hungry, drawing only around 29 µA when sampling every 16 milliseconds.

The following project illustrates interfacing and programming an MPR121 Capacitive Touch Keypad (**Touch Pot** for short). There are numerous low-cost boards with the MPR121 device being sold online – one of them purchased on www.ebay.com is used in the project.

The circuit diagram of the project (**Fig.28**) shows the connections between 12-key MPR121 Capacitive Touch Keypad and Arduino Zero (Arduino M0 Pro).

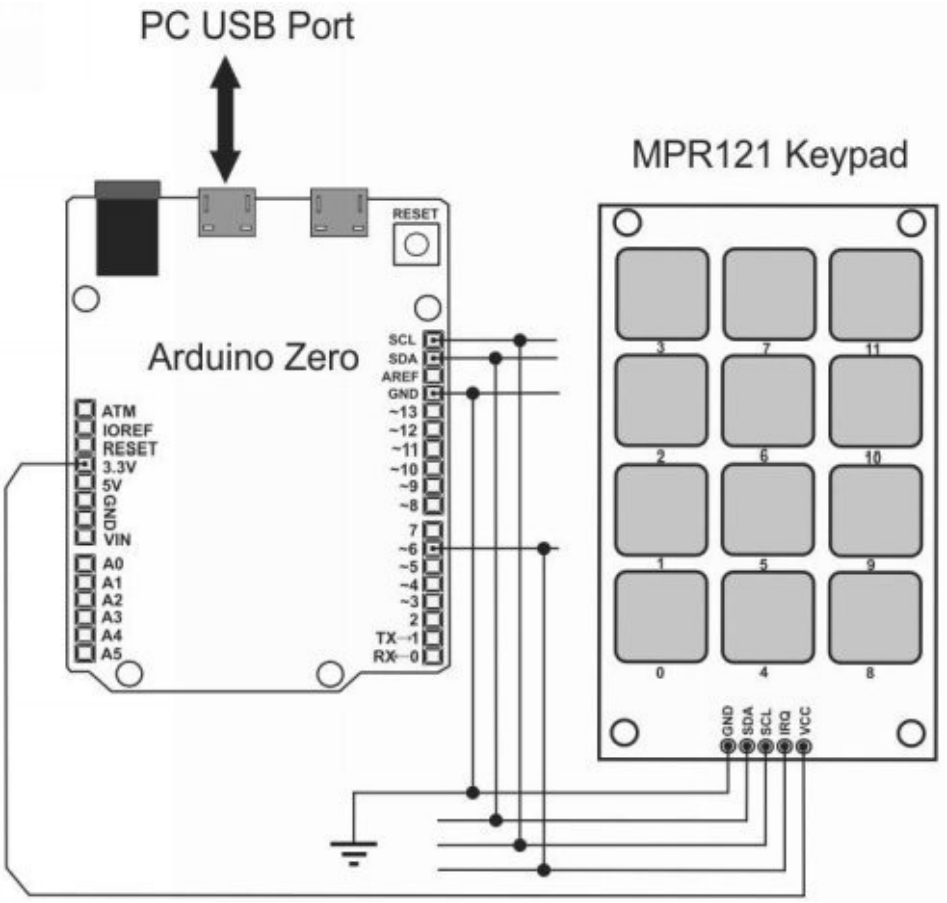


Fig.28

The connections between the boards are detailed in the table below.

Arduino Zero pin	**MPR121 board pin**
3.3V	VCC
GND	GND
SDA	SDA
SCL	SCL
~6	IRQ

To communicate with Capacitive Touch Keypad we can use the Arduino sketch available at https://cdn.sparkfun.com/assets/9/6/3/4/5/52a0dbb8757b7f21158b4567.zip.

To communicate with Capacitive Touch Keypad we can develop the Arduino sketch whose source code (**Listing 4**) is the modified version of that available at

https://cdn.sparkfun.com/assets/9/6/3/4/5/52a0dbb8757b7f21158b4567.zip.

Listing 4.

```
#include “mpr121.h”
#include <Wire.h>
#include <stdio.h>

int cnt[12] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
char buf[128];

int irqPin = 6;  // Digital input D6 is used for monitoring the IRQ line of MPR121
boolean touchStates[12]; //to keep track of the previous touch states

void setup(){
  pinMode(irqPin, INPUT_PULLUP);
  Serial.begin(9600);
  Wire.begin();
  mpr121_setup();
}

void loop(){
  readTouchInputs();
}

void readTouchInputs()
{
if(!checkInterrupt())
{
  //read the touch state from the MPR121
  Wire.requestFrom(0x5A,2);
```

```
byte LSB = Wire.read();
byte MSB = Wire.read();
uint16_t touched = ((MSB << 8) | LSB); //16bits that make up the touch states

for (int i=0; i < 12; i++)
{
  // Check what electrodes were pressed
  if(touched & (1<<i))
  {
  if(touchStates[i] == 0)
  {
  //pin i was touched
  memset(buf, 0, sizeof(buf));
  sprintf(buf,“Pin %d was touched %d times.”, i, ++cnt[i]);
  Serial.println(buf);
  }
  else if(touchStates[i] == 1)
  {
  //pin i is still being touched
  }
  touchStates[i] = 1;
}
else
  {
  if(touchStates[i] == 1)
  {
  memset(buf, 0, sizeof(buf));
  sprintf(buf,“Pin %d is released.”, i);
  Serial.println(buf);

  //pin i released
```

```
    }
    touchStates[i] = 0;
    }
  }
}
}

void mpr121_setup(void)
{
  set_register(0x5A, ELE_CFG, 0x00);

  // Section A - Controls filtering when data is > baseline.
  set_register(0x5A, MHD_R, 0x01);
  set_register(0x5A, NHD_R, 0x01);
  set_register(0x5A, NCL_R, 0x00);
  set_register(0x5A, FDL_R, 0x00);

  // Section B - Controls filtering when data is < baseline.
  set_register(0x5A, MHD_F, 0x01);
  set_register(0x5A, NHD_F, 0x01);
  set_register(0x5A, NCL_F, 0xFF);
  set_register(0x5A, FDL_F, 0x02);

  // Section C - Sets touch and release thresholds for each electrode
  set_register(0x5A, ELE0_T, TOU_THRESH);
  set_register(0x5A, ELE0_R, REL_THRESH);

  set_register(0x5A, ELE1_T, TOU_THRESH);
  set_register(0x5A, ELE1_R, REL_THRESH);

  set_register(0x5A, ELE2_T, TOU_THRESH);
```

```
set_register(0x5A, ELE2_R, REL_THRESH);

set_register(0x5A, ELE3_T, TOU_THRESH);
set_register(0x5A, ELE3_R, REL_THRESH);

set_register(0x5A, ELE4_T, TOU_THRESH);
set_register(0x5A, ELE4_R, REL_THRESH);

set_register(0x5A, ELE5_T, TOU_THRESH);
set_register(0x5A, ELE5_R, REL_THRESH);

set_register(0x5A, ELE6_T, TOU_THRESH);
set_register(0x5A, ELE6_R, REL_THRESH);

set_register(0x5A, ELE7_T, TOU_THRESH);
set_register(0x5A, ELE7_R, REL_THRESH);

set_register(0x5A, ELE8_T, TOU_THRESH);
set_register(0x5A, ELE8_R, REL_THRESH);

set_register(0x5A, ELE9_T, TOU_THRESH);
set_register(0x5A, ELE9_R, REL_THRESH);

set_register(0x5A, ELE10_T, TOU_THRESH);
set_register(0x5A, ELE10_R, REL_THRESH);

set_register(0x5A, ELE11_T, TOU_THRESH);
set_register(0x5A, ELE11_R, REL_THRESH);

// Set the Filter Configuration
// Set ESI2
```

```
  set_register(0x5A, FIL_CFG, 0x04);

  // Electrode Configuration
  // Set ELE_CFG to 0x00 to return to standby mode
  set_register(0x5A, ELE_CFG, 0x0C);  // Enables all 12 Electrodes

    // Enable Auto Config and auto Reconfig

  set_register(0x5A, ELE_CFG, 0x0C);
}

boolean checkInterrupt(void)
{
  return digitalRead(irqPin);
}

void set_register(int address, unsigned char r, unsigned char v)
{
Wire.beginTransmission(address);
Wire.write(r);
Wire.write(v);
Wire.endTransmission();
}
```

The contents of the header file **mpr121.h** is shown in **Listing 5**.

Listing 5.

```
// MPR121 Register Defines
#define MHD_R  0x2B
#define NHD_R   0x2C
```

```
#define NCL_R    0x2D
#define FDL_R    0x2E
#define MHD_F  0x2F
#define NHD_F   0x30
#define NCL_F    0x31
#define FDL_F    0x32
#define ELE0_T   0x41
#define ELE0_R   0x42
#define ELE1_T   0x43
#define ELE1_R   0x44
#define ELE2_T   0x45
#define ELE2_R   0x46
#define ELE3_T   0x47
#define ELE3_R   0x48
#define ELE4_T   0x49
#define ELE4_R   0x4A
#define ELE5_T   0x4B
#define ELE5_R   0x4C
#define ELE6_T   0x4D
#define ELE6_R   0x4E
#define ELE7_T   0x4F
#define ELE7_R   0x50
#define ELE8_T    0x51
#define ELE8_R    0x52
#define ELE9_T    0x53
#define ELE9_R    0x54
#define ELE10_T  0x55
#define ELE10_R  0x56
#define ELE11_T  0x57
#define ELE11_R  0x58
#define FIL_CFG  0x5D
```

```
#define ELE_CFG  0x5E
#define GPIO_CTRL0   0x73
#define GPIO_CTRL1   0x74
#define GPIO_DATA    0x75
#define GPIO_DIR       0x76
#define GPIO_EN        0x77
#define GPIO_SET       0x78
#define GPIO_CLEAR   0x79
#define GPIO_TOGGLE  0x7A
#define ATO_CFG0       0x7B
#define ATO_CFGU       0x7D
#define ATO_CFGL       0x7E
#define ATO_CFGT       0x7F



// Global Constants
#define TOU_THRESH  0x06
#define REL_THRESH    0x0A
```

Let's take a look and see exactly what the code in **Listing 4** is doing. This application uses the Wire library to provide the communication between capacitive sensor MPR121 and Arduino Zero. The following sequence defines digital pin **6** (**D6**) as the IRQ pin connection, and creates 12 instances of the boolean variable **touchStates**:

```
#include "mpr121.h"
#include <Wire.h>


int irqPin = 6;
boolean touchStates[12];
```

The section is followed by the **setup()** procedure which defines the **irqPin** as an INPUT with the internal pull-up resistor connected. The Arduino application will monitor the digital signal arriving on pin **6**. Serial, Wire and mpr121 interfaces are also started:

```
void setup(){
  pinMode(irqPin, INPUT_PULLUP);

  Serial.begin(9600);
  Wire.begin();

  mpr121_setup();
}
```

The **loop()** procedure of the code is simple – it repeatedly calls a single function **readTouchInputs()**:

```
void loop(){
  readTouchInputs();
}
```

The **readTouchInputs()** function reads the electrode states from the sensor, and the least significant bits and most significant bits are defined for the sensor. The Arduino code scans each electrode and sends a message over a serial interface if an electrode is triggered as being touched. The Arduino code then prints out a message as soon as the electrode is no longer being touched.

The source code also defines the threshold values for each electrode. Each electrode must have a touch threshold and a release threshold for the Arduino application to compare the current state of the electrode.

The last two functions in the example sketch simply check the status of the **irqPin** to determine if the IC is signaling that an electrode has been touched.

Interfacing digital input devices: Sparkfun Touch Potentiometer

This section is dedicated to use of the Sparkfun Touch Potentiometer which is an intelligent linear capacitive touch sensor that implements potentiometer functionality with 256 positions. It can operate as a peripheral to a computer or Arduino Zero or in a stand-alone capacity. The Touch Potentiometer provides both a dual-channel analog and PWM output for direct control of other circuitry. Configurable analog and PWM transfer functions support a wide variety of applications.

The device has the following features:

- Dual host interfaces: Logic-level serial and I^2C;
- Dual 8-bit 20 K 3-terminal digitally controlled variable resistor outputs;
- PWM output;
- 8 LED display with multiple display modes and intensity levels;
- Option for interpolated (soft) changes between touches;
- Configurable touch sensor parameters for a variety of PCB covers;
- Easily configurable I^2C address to allow multiple devices on one bus;
- Configurable linear or non-linear PWM transfer function;
- Configurable linear or simulated logarithmic variable resistor transfer function;
- Variable resistor supports single- or dual-supply operation;
- Simple register interface with jabber option;
- Programmable power-on default operation;
- Built-in calibration procedure;
- User-accessible EEPROM data storage;
- Built-in Low-dropout voltage regulator for operation up to DC 12 V.

The Touch Potentiometer board is shown in **Fig.29**.

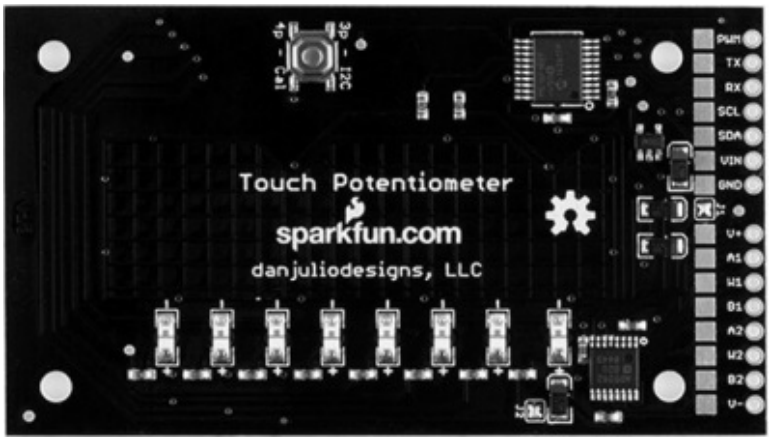


Fig.29

The block diagram of the Touch Potentiometer is shown in **Fig.30**.

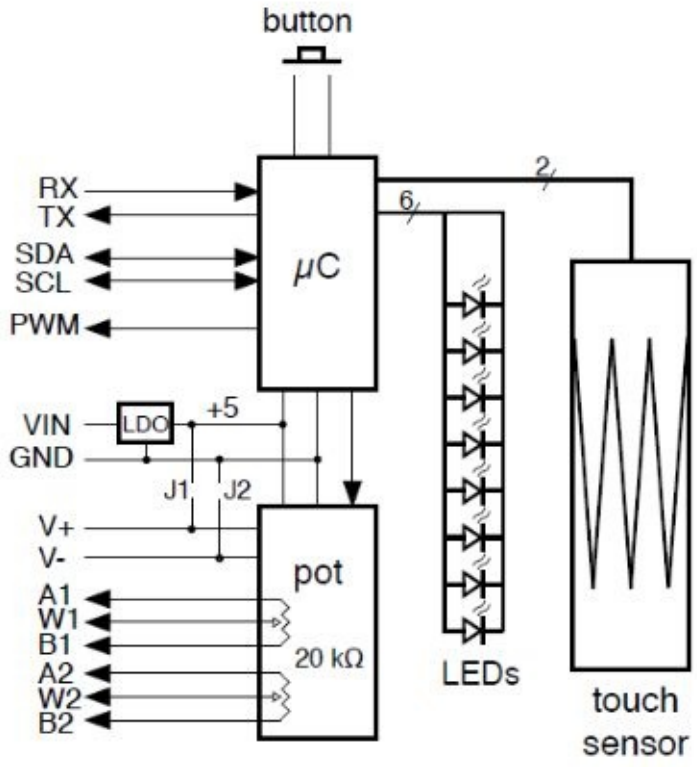


Fig.30

The Touch Potentiometer is controlled by a Microchip PIC16F1829 8-bit microcontroller that provides the host interface, LED control, capacitive sense and peripheral control functions. The digital potentiometer functions are provided by an Analog Devices AD5262 integrated circuit. Capacitive sensing is done by two, interleaved sensors on the top side of the PCB that control an oscillator inside the micro-controller. Touching the sensors changes the oscillator frequency which is read by the micro-controller. Position is determined by a ratio-metric calculation of the changes in oscillator frequency for both sensors. The LEDs are controlled using pulse-width-modulation and multiplexing. A push button allows changing the I2C address and device calibration without an external computer or micro-controller. A Low-Dropout voltage regulator (LDO) allows operation up to 12 VDC.

The next projects illustrate using the Touch Potentiometer with Arduino Zero.

Project 1

This project allows to build the application which generates the PWM signal with a duty cycle adjusted by Touch Potentiometer (**Touch Pot**). The circuit diagram is shown in **Fig.31**.

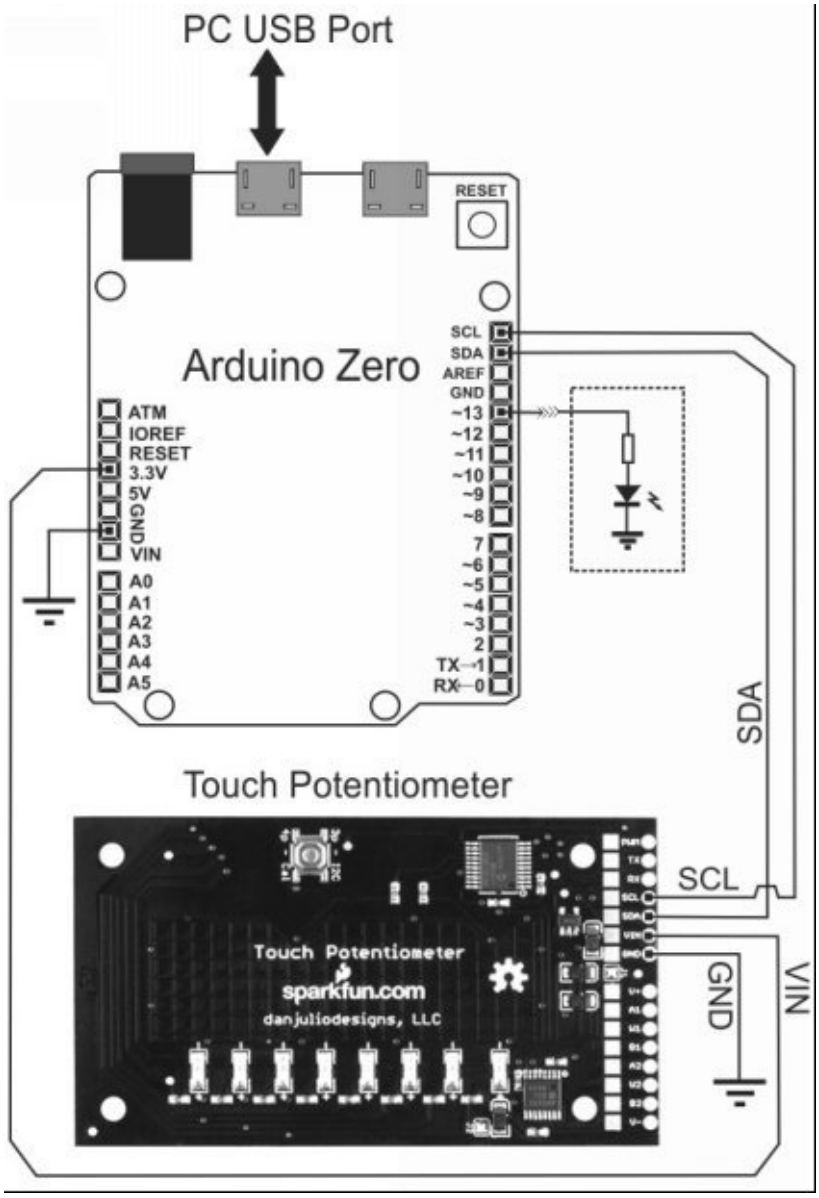


Fig.31

In this circuit, the PWM pulses feed the built-in LED network on pin **13** of Arduino Zero. As the duty cycle of PWM varies, the brightness of LED varies as well. According to the user manual on the device, the Touch Potentiometer board can be powered by the DC +5V source. Nevertheless, the board can operate with +3.3V, so we can take the power to Touch Potentiometer (pin **VIN**) from pin "**3.3V**" of Arduino Zero.

The source code of the demo application is shown in **Listing 6**.

Listing 6.

```
/*
The program code reads the touch.pot value and adjusts the brightness of the LED on pin 13. New values are output in the Serial Monitor window. Both the direct and indirect commands are used. Assume that Touch Pot is assigned the I2C address = 0x8 (by default).
*/

#include “Wire.h”

int i2cAddr = 8; // Direct access to i2cAddr = 8, indirect one uses the address i2cAddr+1
uint8_t prevValue;
uint8_t curValue;

void setup()
{
Serial.begin(9600);
Wire.begin();
pinMode(13, OUTPUT);

// demonstrates the access to Touch Potentiometer registers

WriteTpReg(1, 128); // configuring the wiper position to 50% by writing to register 1
curValue = ReadTpReg(1); // read back the value just set

// configuring the PWM duty cycle on pin 13
analogWrite(13, curValue);
prevValue = curValue;
}
```

```
void loop()
{
delay(50); // Reading registers every 20 mS
// demonstrates direct access to Touch Potentiometer value

curValue = ReadTpValue(); // faster I2C access to the register than ordinary reading
if (curValue != prevValue)
{
  analogWrite(13, curValue);
  Serial.println(curValue);
  prevValue = curValue;
}
}

// Writing a value to the Touch Potentiometer register

void WriteTpReg(uint8_t addr, uint8_t data)
{
Wire.beginTransmission(i2cAddr+1);
Wire.write('W');
Wire.write(addr);
Wire.write(data);
Wire.endTransmission();
}

// Reading the Touch Potentiometer value

uint8_t ReadTpValue()
{
Wire.requestFrom(i2cAddr, 1);
if (Wire.available())
```

```
{
  return Wire.read();
}
else
{
  return 0;
}
}

// Reading the Touch Potentiometer register

uint8_t ReadTpReg(uint8_t addr)
{
Wire.beginTransmission(i2cAddr+1);
Wire.write('R');
Wire.write(addr);
Wire.endTransmission();
Wire.requestFrom(i2cAddr+1, 1);
if (Wire.available())
{
  return Wire.read();
}
else
{
return 0;
}
}
```

Project 2

This project allows to build the application which generates a pulse train with a frequency adjusted by Touch Potentiometer. The circuit diagram for this project is the same as that used in the previous project. The output signal is taken from pin **11** of Arduino Zero.

The source code of the application is shown in **Listing 7**.

Listing 7.

```
/*
Assume that Touch Pot is accessible via I2C Address = 8
*/

#include “Wire.h”

int i2cAddr = 8;
uint8_t prevValue;
uint8_t curValue;

long initFreq = 500; // the value of an initial frequency = 500 Hz
long K = 10;
long Freq;

void setup()
{
Serial.begin(9600);
Wire.begin();

WriteTpReg(1, 128);
curValue = ReadTpReg(1);

// Set the initial frequency on pin 11
```

```
Freq = initFreq + K*curValue;
tone(11, Freq); // the frequency is set to 500 + 1280= 1780 Hz
prevValue = curValue;
}

void loop()
{
delay(50); // reading Touch Pot every 20mS

curValue = ReadTpValue();
if (curValue != prevValue)
{
  Freq = initFreq + K*curValue;
  tone(11, Freq);

  Serial.print("The Touch Pot value: ");
  Serial.print(curValue);
  Serial.print(" the freq. = ");
  Serial.print(Freq);
  Serial.println(" Hz");
  prevValue = curValue;
}
}

void WriteTpReg(uint8_t addr, uint8_t data)
{
Wire.beginTransmission(i2cAddr+1);
Wire.write('W');
Wire.write(addr);
Wire.write(data);
```

```
Wire.endTransmission();
}

uint8_t ReadTpValue()
{
Wire.requestFrom(i2cAddr, 1);
if (Wire.available())
{
  return Wire.read();
}
else
{
  return 0;
}
}

uint8_t ReadTpReg(uint8_t addr)
{
Wire.beginTransmission(i2cAddr+1);
Wire.write('R');
Wire.write(addr);
Wire.endTransmission();
Wire.requestFrom(i2cAddr+1, 1);
if (Wire.available())
{
  return Wire.read();
}
else
{
return 0;
}
```

}

Project 3

This project allows to build a simple digital-to analog converter (DAC) with the DC voltage taken from the wipers of digital potentiometers of Touch Potentiometer. The value of output is reflected in the **Serial Monitor** window. The circuit diagram of the project is shown in **Fig.32**.

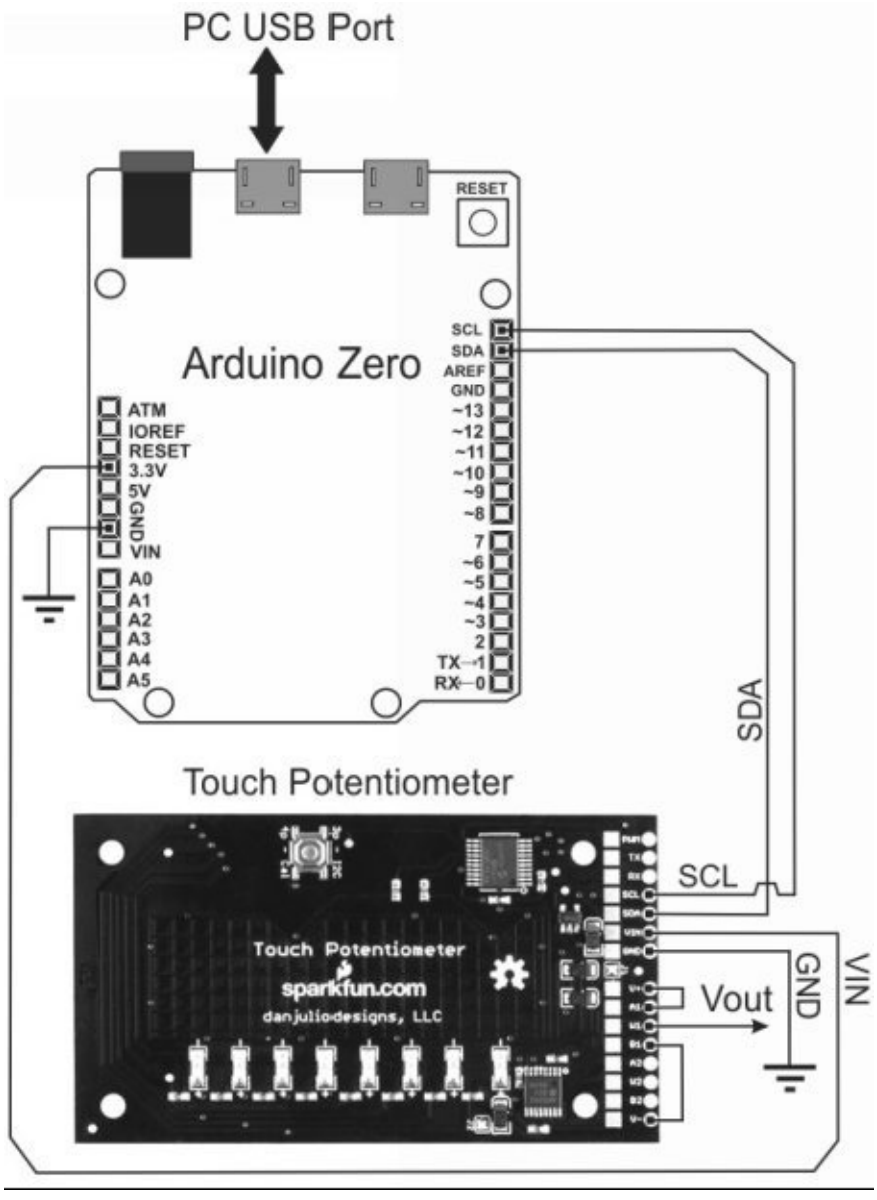


Fig.32

The source code of the Arduino Zero application is shown in **Listing 8**.

Listing 8.

```
/*
assume that Touch Pot is accessible via I2C Address = 8
*/

#include “Wire.h”

int i2cAddr = 8;
uint8_t prevValue;
uint8_t curValue;

double LSB = 3.3/256; // the LSB value for the reference voltage = Vdd = 3.3V
double Vout; // the output voltage on wiper W1

void setup()
{
Serial.begin(9600);
Wire.begin();

WriteTpReg(1, 128);
curValue = ReadTpReg(1);
prevValue = curValue;
}

void loop()
{
delay(200); // Reading Touch Pot every 20 mS

curValue = ReadTpValue();
if (curValue != prevValue)
{
  Vout = LSB * curValue - LSB;
```

```
  Serial.print(“Touch Pot value = “);
  Serial.print(curValue);
  Serial.print(“, wiper W1 output = “);
  Serial.print(Vout,3);
  Serial.println(” V”);
  prevValue = curValue;
}
}

void WriteTpReg(uint8_t addr, uint8_t data)
{
Wire.beginTransmission(i2cAddr+1);
Wire.write(‘W’);
Wire.write(addr);
Wire.write(data);
Wire.endTransmission();
}

uint8_t ReadTpValue()
{
Wire.requestFrom(i2cAddr, 1);
if (Wire.available())
{
  return Wire.read();
}
else
{
  return 0;
}
}
```

```
uint8_t ReadTpReg(uint8_t addr)
{
Wire.beginTransmission(i2cAddr+1);
Wire.write('R');
Wire.write(addr);
Wire.endTransmission();
Wire.requestFrom(i2cAddr+1, 1);
if (Wire.available())
{
  return Wire.read();
}
else
{
return 0;
}
}
```

The **Serial Monitor** window of the running application is shown in **Fig.33**.

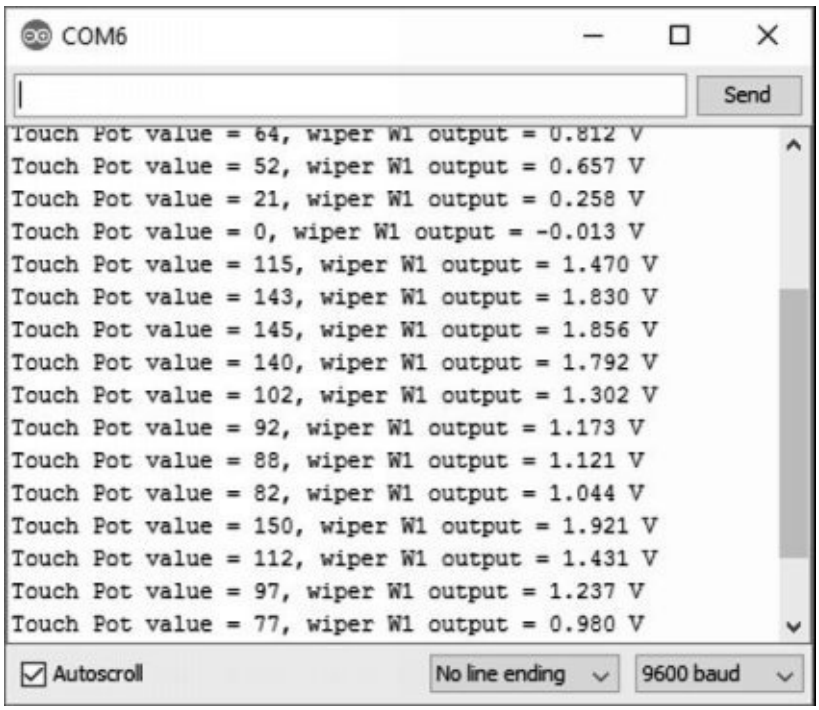


Fig.33

Data output using a serial OLED display

This project illustrates how to visualize data provided by an Arduino Zero application using popular OLED displays. The application will periodically measure analog voltage on **A0 – A2** inputs and output the result on the serial OLED display manufactured by Digole Digital Solutions.

Serial Digole OLED (**Fig.34**) can communicate with external circuitry via either of UART/I2C/SPI interfaces. The simplest of those interfaces is UART, so all projects from this guide will use UART for data transfer to OLED.

Fig.34

The circuit diagram of the project is shown in **Fig.35**.

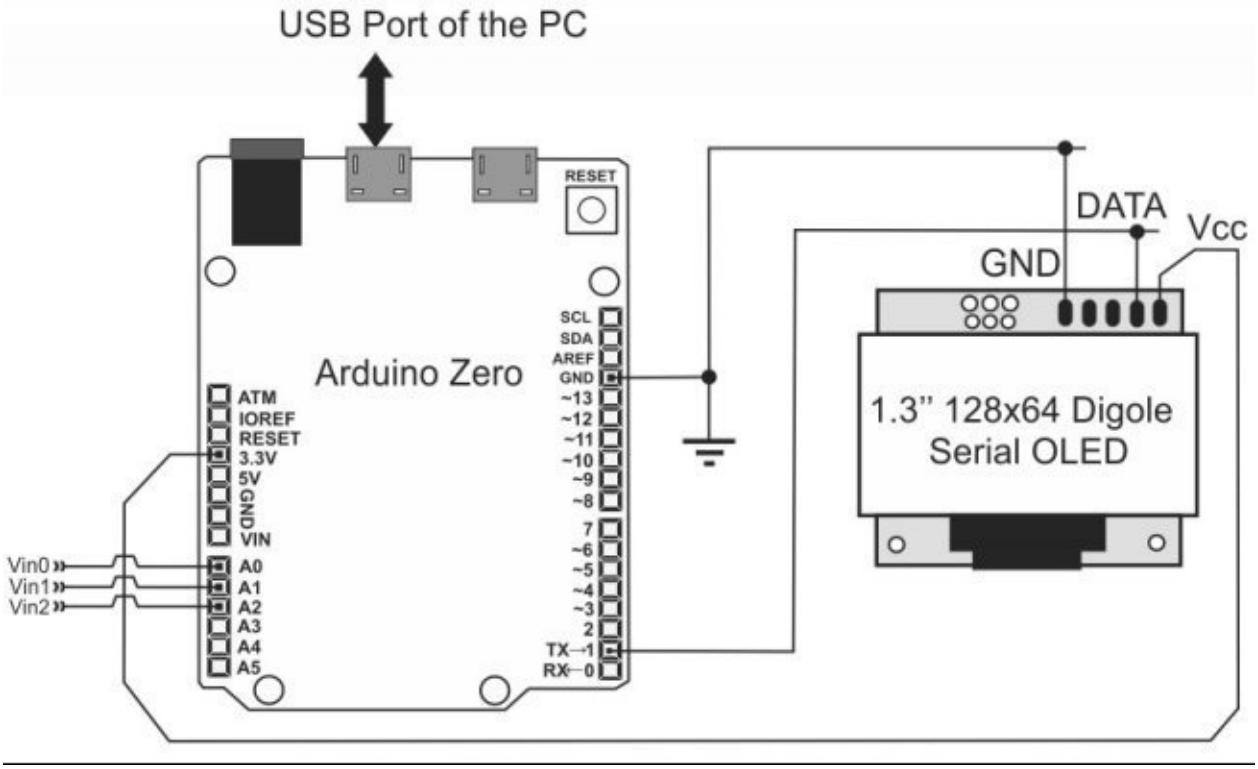


Fig.35

In this circuit, input **RX** of the OLED display (labeled "**DATA**" for this particular device) is wired to pin "**TX→1**" of Arduino Zero.

Before writing the source code we must include the **DigoleSerial** library in Arduino IDE. The Arduino source code of the demo application is shown in **Listing 9**.

Listing 9.

```
#define _Digole_Serial_UART_  // using UART for data transfer

#include <DigoleSerial.h>

//——UART setup————
// Pin 1 (TX) on Arduino is connected to RX line ("DATA") on OLED

DigoleSerialDisp mydisp(&Serial1, 9600);

int binADC;
```

```
const float LSB = 3.33 / 1024;
float vin[3];
int i, chs[3] = {0, 1,2};
const unsigned char fonts[] = {0, 6, 10};

void setup()
{
analogReadResolution(10);
Serial1.begin(9600);
mydisp.begin();
mydisp.clearScreen(); //clear screen
mydisp.disableCursor(); //disable cursor, to enable cursor use: enableCursor();

mydisp.setPrintPos(0, 0, _TEXT_);
mydisp.setFont(fonts[2]); // set font 6x10
}

void loop()
{
for (i = 0; i < 3; i++)
{
  binADC = analogRead(i);
  vin[i] = LSB * binADC;
  mydisp.setPrintPos(0, i, _TEXT_);

  mydisp.print("CH");
  mydisp.print(i);
  mydisp.print(" Input: ");
  mydisp.print(vin[i], 3);
  mydisp.print("V");
}
```

```
delay(5000);
}
```

In this source code, we set the resolution of the A/D converter to 10 bit and the reference voltage to 3.3V, therefore the **LSB** constant relects these settings.

The statement

```
DigoleSerialDisp mydisp(&Serial1, 9600);
```

assignes the Arduino Zero interface **Serial 1** to the **mydisp** object. To enable **mydisp** the following statement within the **setup()** function is used:

```
mydisp.begin();
```

Then we can apply various methods of the **mydisp** object to operate with the OLED display.

The sequence

```
mydisp.clearScreen();
mydisp.disableCursor();

mydisp.setPrintPos(0, 0, _TEXT_);
mydisp.setFont(fonts[2]);
```

clears the screen, disables the cursor, then sets the initial position for data output and sets the type of the font to 6x10.

Processing the signals is being performed within the **for (i = 0; i < 3; i++)** loop.

Arduino Zero multitasking

The several projects that follow are dedicated to designing multitasking applications. Multitasking allows to optimize performance of an application by executing several tasks in parallel instead of running a single super loop.

The **Scheduler** library enables Arduino Zero to run multiple functions at the same time. This allows tasks to happen without interrupting each other. This is a cooperative scheduler in that the CPU switches from one task to another. The library includes methods for passing control between tasks.

Project 1

This project allows to build a multitasking application measuring analog voltage levels on pins **A0** – **A2** of Arduino Zero. The circuit diagram of this application is the same as that shown in **Fig.35**.

The source code of this project is shown in **Listing 10**.

Listing 10.

```
#define _Digole_Serial_UART_  // using UART for data transfer
#include <Scheduler.h>
#include <DigoleSerial.h>

//——UART setup————
// Pin 1 (TX) on Arduino is connected to RX line (“DATA”) on OLED

DigoleSerialDisp mydisp(&Serial1, 9600);

int binADC;
const float LSB = 3.33 / 1024;
float vin[3];
int i,i1, chs[3] = {0, 1,2};
const unsigned char fonts[] = {0, 6, 10};
int done = 0;

void setup()
{
analogReadResolution(10);
Serial1.begin(9600);
mydisp.begin();
mydisp.clearScreen();
```

```
mydisp.disableCursor();

mydisp.setPrintPos(0, 0, _TEXT_);
mydisp.setFont(fonts[2]);

// Add “loop2” to scheduling.
  // “loop” is always started by default.

  Scheduler.startLoop(loop2);
}

void loop()
{
done = 0;
for (i = 0; i < 3; i++)
{
  binADC = analogRead(i);
  vin[i] = LSB * binADC;
}
done = 1;
delay(50);
// IMPORTANT:
  // We must call ‘yield’ at a regular basis to pass
  // control to other tasks.
yield();
}

// Task 2
void loop2()
{
if (done == 1)
```

```
{
for (i1 = 0; i1 < 3; i1++)
{
  mydisp.setPrintPos(0, i1, _TEXT_);
  mydisp.print(“Ch”);
  mydisp.print(i1);
  mydisp.print(” Input: “);
  mydisp.print(vin[i1], 3);
  mydisp.print(“V”);
}
  // IMPORTANT:
  // When multiple tasks are running ‘delay’ passes control to
  // other tasks while waiting and guarantees they get executed.
  delay(1000);
}
}
```

In this source code, the **loop()** function performs the measurements on channels **A0** – **A2**, while the second (function **loop2()**) outputs the data saved in array **vin[3]** on the OLED display. The variable **done** is used for synchronizing both threads with each other.

Project 2

This project represents one more multitasking application that running multiple threads. The mail thread (function **loop()**) reads the analog voltage level on pin **A0** and saves the result of the measurement in memory. The second (function **loop2()**) sends the result of the measurement to the **Serial Monitor**. The third thread (function **loop3()**) adjusts the PWM duty cycle (pin **9**) proportional to the voltage level on pin **A0**. The LED connected to pin **9** changes its brightness as the duty cycle varies. The input voltage should range from 0.05 to 0.95 V.

The circuit diagram of the project is shown in **Fig.36**.

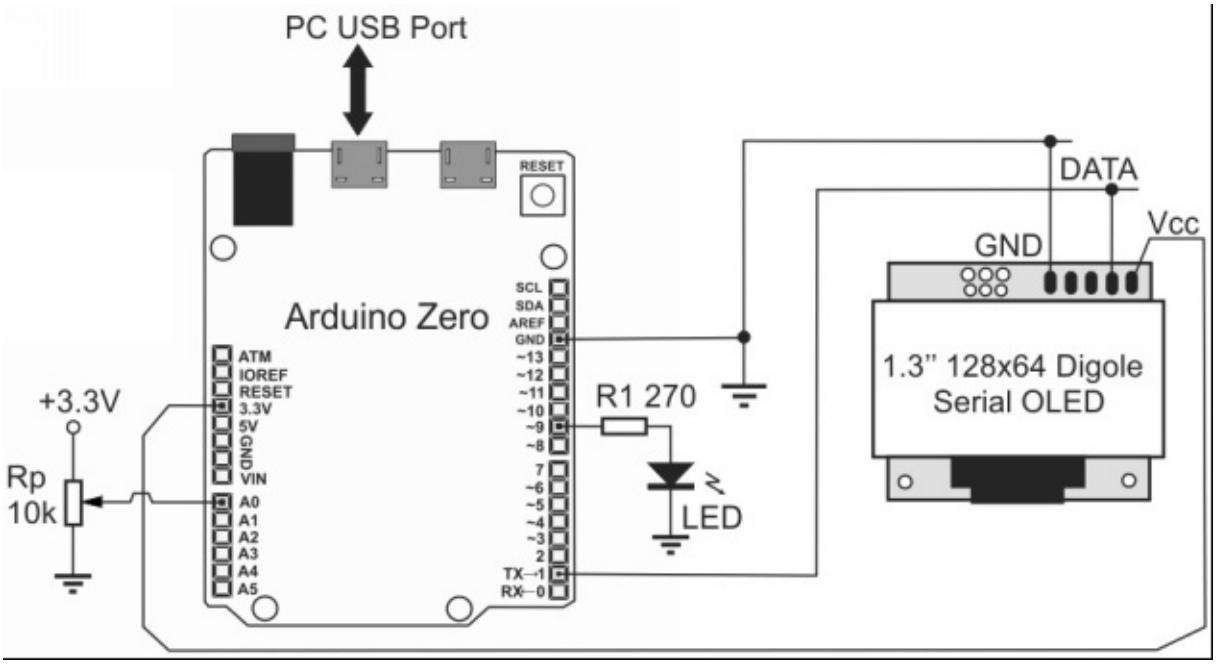


Fig.36

In this circuit, the control voltage to pin **A0** is taken from the wiper of the potentiometer **Rp** arranged as a voltage divider. The data to the OLED display (pin "**DATA**") is transferred from pin "**TX → 1**" of Arduino Zero. The network R1 – LED is connected to pin **9** of Arduino.

The source code of the application is shown in **Listing 11**.

Listing 11.

```
#include <Scheduler.h>

int ledPin = 9;
int binADC;
const float LSB = 3.33 / 1024;
float vin0;
int done = 0;

void setup()
{
analogReadResolution(10);
analogWriteResolution(10);
pinMode(9, OUTPUT);
analogWrite(ledPin, 100);
Serial.begin(9600);

// Add “loop2” to scheduling.
  // “loop” is always started by default.
  Scheduler.startLoop(loop2);
  Scheduler.startLoop(loop3);
}

void loop()
{
done = 0;
binADC = analogRead(0);
vin0 = LSB * binADC;
done = 1;
delay(50);
// IMPORTANT:
  // We must call ‘yield’ at a regular basis to pass
```

```
  // control to other tasks.
yield();
}

// Task 2
void loop2()
{
if (done == 1)
{
  Serial.print("Channel A0 Input, V: ");
  Serial.println(vin0, 3);
  // IMPORTANT:
  // When multiple tasks are running 'delay' passes control to
  // other tasks while waiting and guarantees they get executed.
  delay(3000);
}
}

// Task 3
void loop3()
{
if (done == 1)
{
  if ((vin0 <= 0.95) && (vin0 >= 0.05))
     analogWrite(ledPin, (int)(vin0 * 100));
  yield();
}
}
```

The programmable wide-band oscillator Si5351 in Arduino applications

This section is dedicated to designing various apllications using the wide-band oscillator Si5351 from Silicon Labs. There is a brief excerption from the datasheet on the device below.

The Si5351 is an I^2C configurable clock generator that is ideally suited for replacing crystals, crystal oscillators, VCXOs, phase-locked loops (PLLs), and fanout buffers in cost-sensitive applications. Based on a PLL/VCXO + high resolution MultiSynth fractional divider architecture, the Si5351 can generate any frequency up to 200 MHz on each of its outputs with 0 ppm error. Three versions of the Si5351 are available to meet a wide variety of applications. The Si5351A generates up to 8 free-running clocks using an internal oscillator for replacing crystals and crystal oscillators. The Si5351B adds an internal VCXO and provides the flexibility to replace both free-running clocks and synchronous clocks. It eliminates the need for higher cost, custom pullable crystals while providing reliable operation over a wide tuning range. The Si5351C offers the same flexibility but synchronizes to an external reference clock (CLKIN).

Our projects use the Adafruit Si5351 Clock Generator Breakout (**Fig.37**).

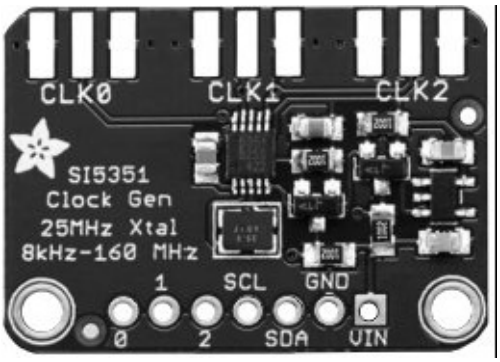


Fig.37

Project 1

This project allows to build an application that can adjust the frequency on output channel **0** of Si5351 proportional to the control voltage applied to channel **A0** of Arduino Zero. The base frequency of the oscillator is set to 100.0 KHz; this frequency can change from 100 to 110 KHz as the input voltage on **A0** changes from 0.1 V to 1V. The value of the control voltage and the output frequency are transferred to the **Serial Monitor**.

The circuit diagram of the project is shown in **Fig.38**.

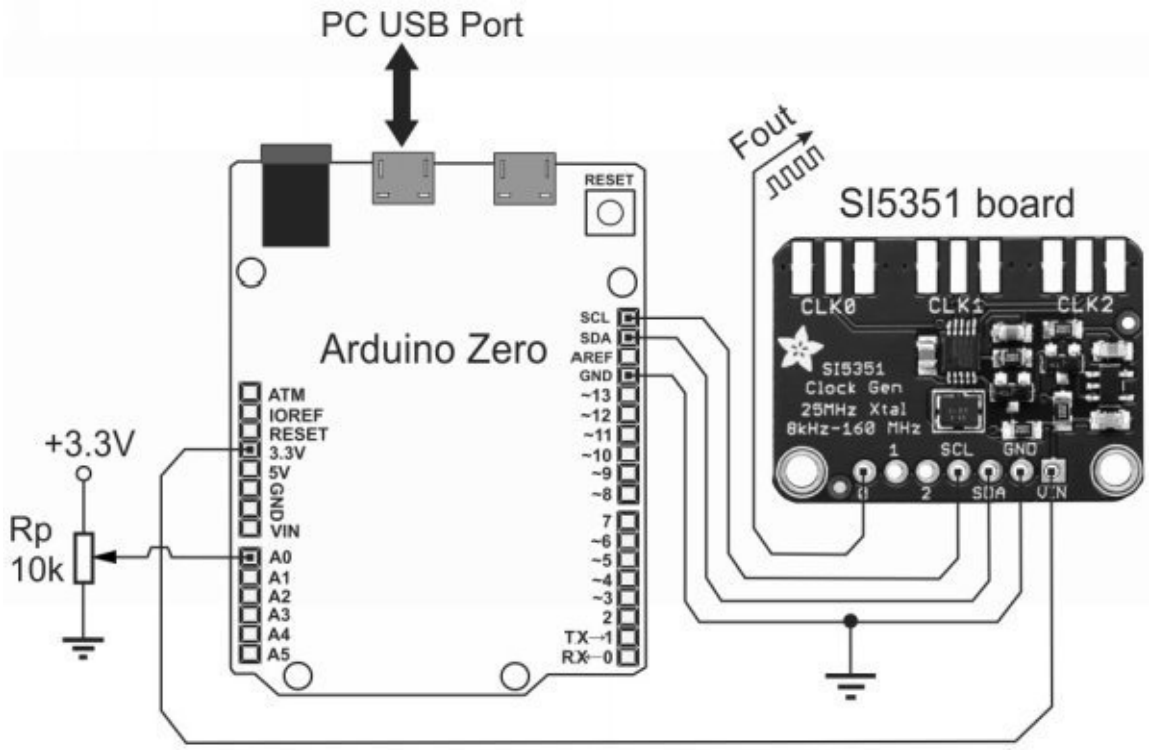


Fig.38

Here the control voltage to pin **A0** is taken from the wiper of the potentiometer **Rp**.

The connections between Arduino Zero and Si5351 are detailed in the table below.

Arduino Zero pin	**Si5351 board pin**
SCL	SCL
SDA	SDA
3.3V	VIN

GND	GND

The Arduino source uses the functions from the library written by Jason Milldrum (milldrum@gmail.com). The software is accompanied by the detailed description of the library functions. Much more information relating to the Si5351 library can be found on https://github.com/etherkit/Si5351Arduino.

The source code of the application is shown in **Listing 12**.

Listing 12.

```
#include “si5351.h”
#include “Wire.h”

unsigned long baseFreq = 10000000; // init. Freq. = 100 KHz
int i1;

const long KFreq = 1000000; // the coefficient for calculating a freq. in KHz
unsigned long newFreq, curFreq;
int binADC;
const float LSB = 3.33 / 1024;
float vin;
float fKHz;

Si5351 si5351;

void setup()
{
  // Start serial and initialize the Si5351
  analogReadResolution(10);
  Serial.begin(9600);
  si5351.init(SI5351_CRYSTAL_LOAD_8PF, 0);
```

```
  // 100.0KHz
  si5351.set_pll(SI5351_PLL_FIXED, SI5351_PLLA);
  si5351.set_freq(baseFreq, SI5351_PLL_FIXED, SI5351_CLK0);
  Serial.println("The default output frequency on channel 0 is 100.0 KHz");
  delay(10000);
  }

void loop()
{
  binADC = analogRead(0);
  vin = LSB * binADC;
  if ((vin >= 0.1) && (vin <= 1.0))
  {
    newFreq = (long)(vin * KFreq); // new freq. in KHz
    if (newFreq != curFreq)
    {
    curFreq = newFreq;
    si5351.set_freq(baseFreq + curFreq, SI5351_PLL_FIXED, SI5351_CLK0);
    }
  }

  fKHz = baseFreq + curFreq;
  Serial.print("A0 Input: ");
  Serial.print(vin, 3);
  Serial.println("V");

  Serial.print("Freq.: ");
  Serial.print(fKHz / 100000,3);
  Serial.println(" KHz");
```

```
  delay(7000);
}
```

The application provides the output that can be viewed in the **Serial Monitor** window (**Fig.39)**.

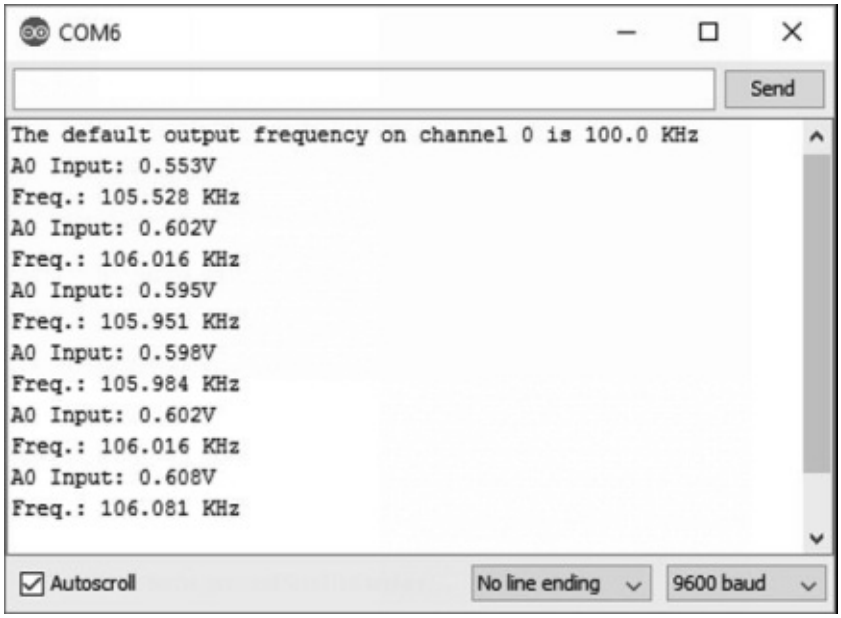


Fig.39

Project 2

The project is similar to the previous one, except that data are transferred to the OLED display instead of the **Serial Monitor** (**Fig.40**).

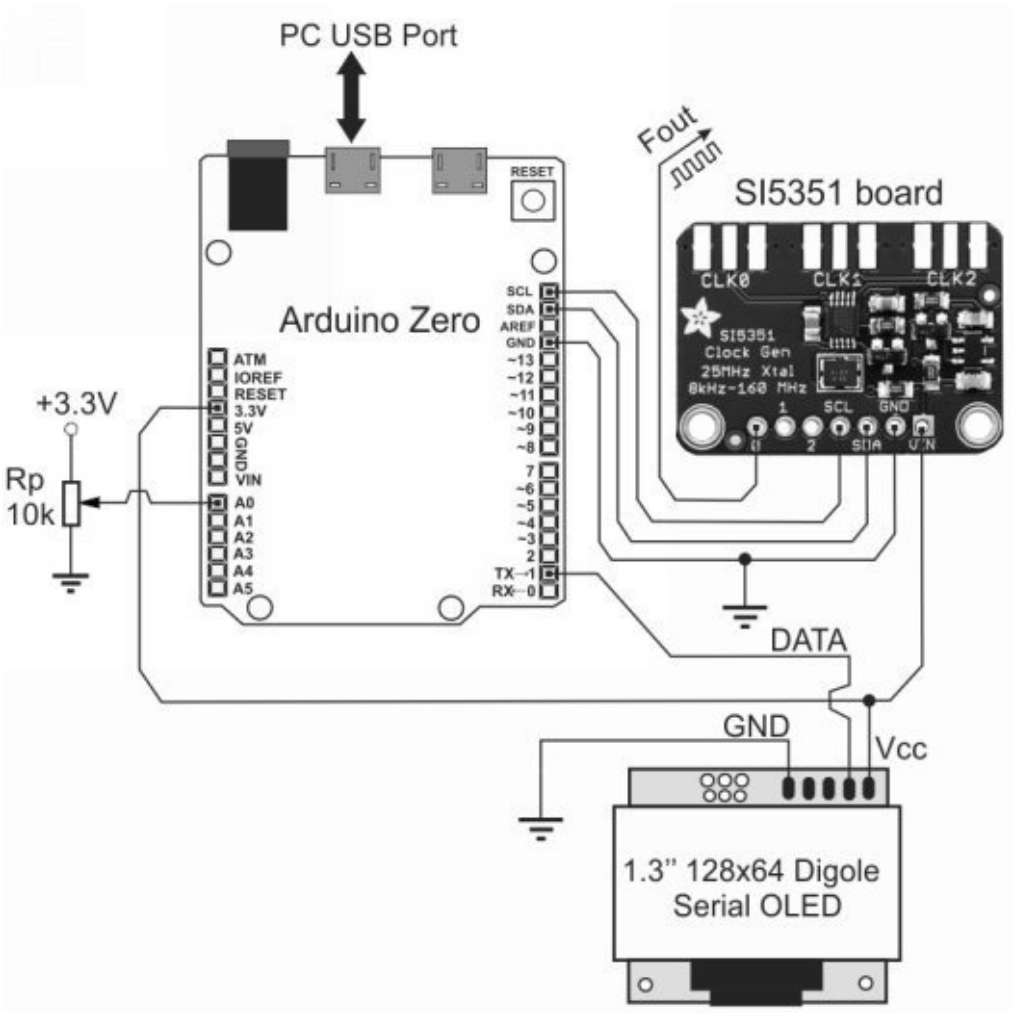


Fig.40

The connections between modules are detailed in the table below.

Arduino Zero pin	**Si5351 pin**	**Digole OLED pin**
SCL	SCL	–
SDA	SDA	–
3.3V	VIN	Vcc
GND	GND	GND

TX → 1	—	DATA

The Arduino source code for this application is shown in **Listing 13**.

Listing 13.

```
#define _Digole_Serial_UART_  // using UART for data transfer

#include <DigoleSerial.h>

//——UART setup————
// Pin 1 (TX) on Arduino is connected to RX line (“DATA”) on OLED

DigoleSerialDisp mydisp(&Serial1, 9600);

#include “si5351.h”
#include “Wire.h”

unsigned long baseFreq = 10000000; // initial frequency = 100 KHz
int i1;

const long KFreq = 1000000; // the coefficient for calculating a freq. in KHz
unsigned long newFreq, curFreq;
int binADC;
const float LSB = 3.33 / 1024;
float vin;
float fKHz;

const unsigned char fonts[] = {0, 6, 10};

Si5351 si5351;
```

```
void setup()
{
  // Start serial and initialize the Si5351
  analogReadResolution(10);

  Serial1.begin(9600);
  mydisp.begin();
  mydisp.clearScreen(); //CLear screen
  mydisp.disableCursor(); //disable cursor, enable cursore use: enableCursor();

  si5351.init(SI5351_CRYSTAL_LOAD_8PF, 0);
  si5351.set_pll(SI5351_PLL_FIXED, SI5351_PLLA);
  si5351.set_freq(baseFreq, SI5351_PLL_FIXED, SI5351_CLK0);

  delay(1000);
  }

void loop()
{
  binADC = analogRead(0);
  vin = LSB * binADC;
  if ((vin >= 0.1) && (vin <= 1.0))
  {
    newFreq = (long)(vin * KFreq); // new freq. in KHz
    if (newFreq != curFreq)
    {
    curFreq = newFreq;
    si5351.set_freq(baseFreq + curFreq, SI5351_PLL_FIXED, SI5351_CLK0);
    }
  }
```

```
  fKHz = baseFreq + curFreq;

  mydisp.clearScreen();
  mydisp.setPrintPos(0, 0, _TEXT_);
  mydisp.setFont(fonts[2]);
  mydisp.print("F0: 100.0 KHz");

  mydisp.setPrintPos(0, 2, _TEXT_);
  mydisp.print("A0 Input: ");
  mydisp.print(vin, 3);
  mydisp.print("V");

  mydisp.setPrintPos(0, 4, _TEXT_);
  mydisp.print("Freq.: ");
  mydisp.print(fKHz / 100000,3);
  mydisp.print(" KHz");

  delay(5000);
}
```

Project 3

This project uses the Si5351 library developed by Adafruit. While this book has been writing, there were the Adafruit library developed only for AVR devices. Nevertheless, the Adafruit Si5351 library can be adopted for Arduino Zero after modifications of the source code.

There will be several projects illustrating the use of the Adafruit Si5351 library.

The source code of the Si5351 library is contained in the **Adafruit_SI5351.cpp** and **Adafruit_SI5351.h** files. In the **Adafruit_SI5351.cpp** file we must comment a few lines and insert the reference to the **math** library. The modified section of **Adafruit_SI5351.cpp** will look like that shown in **Listing 14**.

Listing 14.

```
/**************************************************************************
/*#if defined(__AVR__)
#include <avr/pgmspace.h>
#include <util/delay.h>
#else
#include “pgmspace.h”
#endif
*/
#include <stdlib.h>
#include <math.h> // add this line for using with Arduino Zero

#include <Adafruit_SI5351.h>

/**************************************************************************
```

In the **Adafruit_SI5351.h** header file we must comment the lines in the section shown in **Listing 15**.

Listing 15.

```
.    .    .
/**************************************************************************
#ifndef _SI5351_H_
#define _SI5351_H_

/*#if ARDUINO >= 100
#include <Arduino.h>
#else
#include <WProgram.h>
#endif
#include <Adafruit_Sensor.h>
*/
#include <Wire.h>
. . .
```

After these changes have been done, we can build and launch the test application whose source code is shown in **Listing 16**.

Listing 16.

```
#include <Wire.h>
#include <Adafruit_SI5351.h>

Adafruit_SI5351 clockgen = Adafruit_SI5351();

void setup(void)
{
  Serial.begin(9600);
  Serial.println("Si5351 Clockgen Test"); Serial.println("");
```

```
  /* Initialise the sensor */
  if (clockgen.begin() != ERROR_NONE)
  {
    /* There was a problem detecting the IC … check your connections */
    Serial.print(“Ooops, no Si5351 detected … Check your wiring or I2C ADDR!”);
    while(1);
  }
  Serial.println(“OK!”);

  /* INTEGER ONLY MODE —> most accurate output */
  /* Setup PLLA to integer only mode @ 900MHz (must be 600..900MHz) */
  /* Set Multisynth 0 to 112.5MHz using integer only mode (div by 4/6/8) */
  /* 25MHz * 36 = 900 MHz, then 900 MHz / 8 = 112.5 MHz */
  Serial.println(“Set PLLA to 900MHz”);
  clockgen.setupPLLInt(SI5351_PLL_A, 36);
  Serial.println(“Set Output #0 to 112.5MHz”);
  clockgen.setupMultisynthInt(0, SI5351_PLL_A, SI5351_MULTISYNTH_DIV_8);

  /* FRACTIONAL MODE —> More flexible but introduce clock jitter */
  /* Setup PLLB to fractional mode @616.66667MHz (XTAL * 24 + 2/3) */
  /* Setup Multisynth 1 to 13.55311MHz (PLLB/45.5) */

clockgen.setupPLL(SI5351_PLL_B, 24, 2, 3);
  Serial.println(“Set Output #1 to 13.553115MHz”);
  clockgen.setupMultisynth(1, SI5351_PLL_B, 45, 1, 2);

  /* Multisynth 2 is not yet used and won’t be enabled, but can be */
  /* Use PLLB @ 616.66667MHz, then divide by 900 -> 685.185 KHz */
  /* then divide by 64 for 10.706 KHz */
  /* configured using either PLL in either integer or fractional mode */
```

```
// Serial.println("Set Output #2 to 10.706 KHz");
  //clockgen.setupMultisynth(2, SI5351_PLL_B, 900, 0, 1);
// clockgen.setupRdiv(2, SI5351_R_DIV_64);

  //Serial.println("Set Output #2 to 21.412 KHz");
  //clockgen.setupMultisynth(2, SI5351_PLL_B, 900, 0, 1);
  //clockgen.setupRdiv(2, SI5351_R_DIV_32);



  //Serial.println("Set Output #2 to 80.153 KHz");
  //clockgen.setupMultisynth(2, SI5351_PLL_B, 120, 7, 33);
//clockgen.setupRdiv(2, SI5351_R_DIV_64);

  Serial.println("Set Output #2 to 55.908 KHz");
  clockgen.setupMultisynth(2, SI5351_PLL_B, 172, 17, 50);
  clockgen.setupRdiv(2, SI5351_R_DIV_64);
  /* Enable the clocks */
  clockgen.enableOutputs(true);
}

void loop(void) { }
```

You can check various output frequencies on output **2** of Si5351 device by uncommenting corresponding sections of the source code. In our particular case, the application adjusts the frequency of 55.908 KHz on output **2** of the Si5351 board.

Project 4

This project illustrates how to build application capable of configuring the Si5351 frequency via a serial interface. As it is seen from the previous example, the output frequency of Si5351 can be adjusted by applying the **clockgen.setupMultisynth()** function. We can easily program the Si5351 frequency by passing 3 rightmost parameters. For example, the sequence <172, 17, 50> in the function

clockgen.setupMultisynth(2, SI5351_PLL_B, 172, 17, 50);

configures the divider 172.34 for the base PLL frequency of 616.66667 MHz.

In this case, the output frequency on channel **2** of Si5351 will be calculated as

616.66667 MHz / 172.34 / 64 = 55.908 KHz

Therefore, for a given output frequency **Fout** we can easily calculate the divider **Do** using the following formula:

Do = 616.66667MHz / Fout / 64 (1)

In our example, **Do** turns out to be 172.34 for **Fout** = 55.908 KHz.

The above formula (1) also includes the value 64 which is an additional frequency divider. This value appears as the rightmost parameter (**SI5351_R_DIV_64**) in the function

clockgen.setupRdiv(2, SI5351_R_DIV_64);

Of course, the expression (1) will change for different divider values (4, 8, 32, etc.) being used in function **clockgen.setupRdiv()**. For example, for the divider value = 32 in function

clockgen.setupRdiv(2, SI5351_R_DIV_32)

the formula (1) will change to

Do = 616.66667MHz / Fout / 32 (2)

The following source code (**Listing 17**) allows to adjust the frequency on output **0** of Si5351 by setting the value of **Do**.

Listing 17.

```
#include <Wire.h>
#include <Adafruit_SI5351.h>

const double fPLLB = 616.66667;  // PLLB freq. = 616.66667 MHz
double fDiv = 128; // divider
double updFreq;

double param, fractPart, intPart;
int intParam2, intParam3;
int cmd = 0;

Adafruit_SI5351 clockgen = Adafruit_SI5351();

void setup(void)
{
  Serial.begin(9600);
  Serial.println("Si5351 Clockgen Test"); Serial.println("");

  /* Initialise the sensor */
  if (clockgen.begin() != ERROR_NONE)
  {
    /* There was a problem detecting the IC … check your connections */
    Serial.print("No Si5351 detected … Check your wiring or I2C ADDR!");
    while(1);
```

```
  }
  Serial.println("OK!");

  /* FRACTIONAL MODE —> More flexible but introduce clock jitter */
  /* Setup PLLB to fractional mode @616.66667MHz (XTAL * 24 + 2/3) */
  /* Setup Multisynth 1 to 55.908 KHz (PLLB/172.34) */

  clockgen.setupPLL(SI5351_PLL_B, 24, 2, 3);
  Serial.println("Output #0 is set to 50.0 KHz");
  clockgen.setupMultisynth(0, SI5351_PLL_B, 192, 71, 100);
  clockgen.setupRdiv(0, SI5351_R_DIV_64);
  /* Enable the clocks */
  clockgen.enableOutputs(true);
  Serial.println("Enter a new frequency in KHz <0,###.###>:");
}

void loop(void)
{
while (Serial.available() > 0)
  {

    // look for the next valid integer in the incoming serial stream:
    cmd = Serial.parseInt();
    if (cmd == 0) // 0 = configure Si5351
    {
    updFreq = (double)(Serial.parseFloat());
    param = (fPLLB / fDiv / updFreq) * 1000; // value of the parameters

    fractPart = modf(param, &intPart);
    intParam2 = (int)intPart;
    intParam3 = (int)(fractPart * 100);
```

```
    clockgen.enableOutputs(false);
    clockgen.setupPLL(SI5351_PLL_B, 24, 2, 3);
    clockgen.setupMultisynth(0, SI5351_PLL_B, intParam2, intParam3, 100);
    clockgen.setupRdiv(0, SI5351_R_DIV_128);
    clockgen.enableOutputs(true);
    }
  }
}
```

The **Serial Monitor** window (**Fig.41**) shows the configuration steps for adjusting the output frequency 251.77 KHz on output **0** of Si5351.

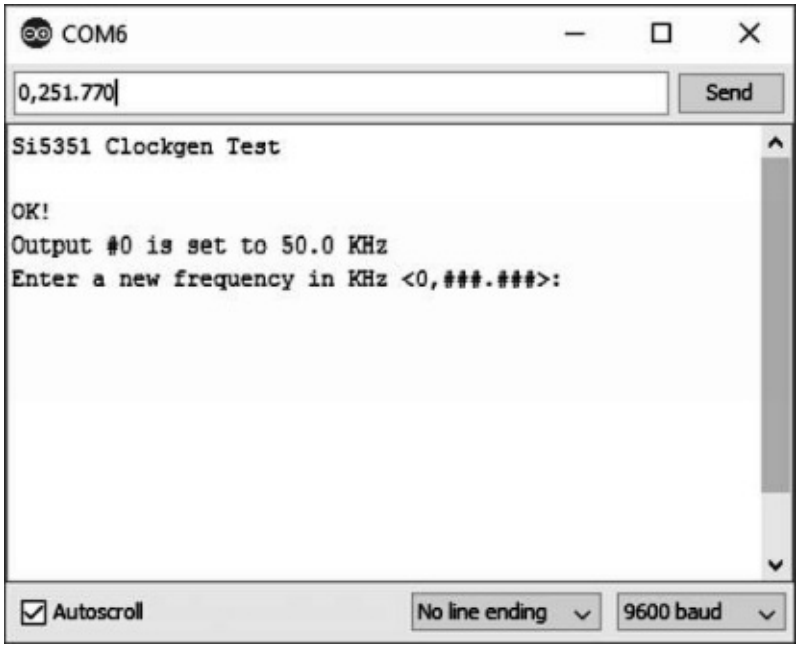


Fig.41

The output frequency of Si5351 was checked by the LabVIEW virtual frequency meter (**Fig.42**).

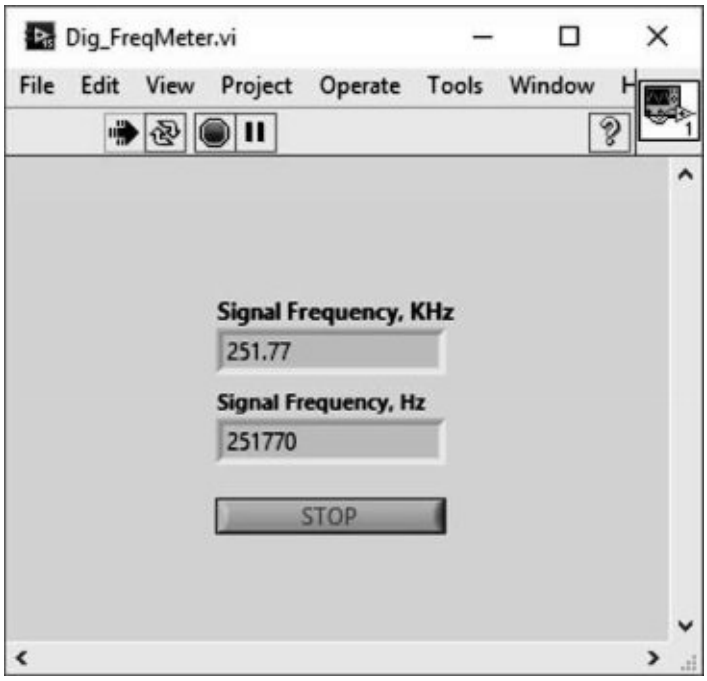

Dig_FreqMeter.vi
File Edit View Project Operate Tools Window H
Signal Frequency, KHz
251.77
Signal Frequency, Hz
251770
STOP

Fig.42

Project 5

This project illustrates how to configure the Si5351 output frequency from the C# .NET application running in Windows 10. The Windows application will send the commands to Arduino Zero via an USB-to-UART adapter.

The circuit diagram of the project is shown in **Fig.43**.

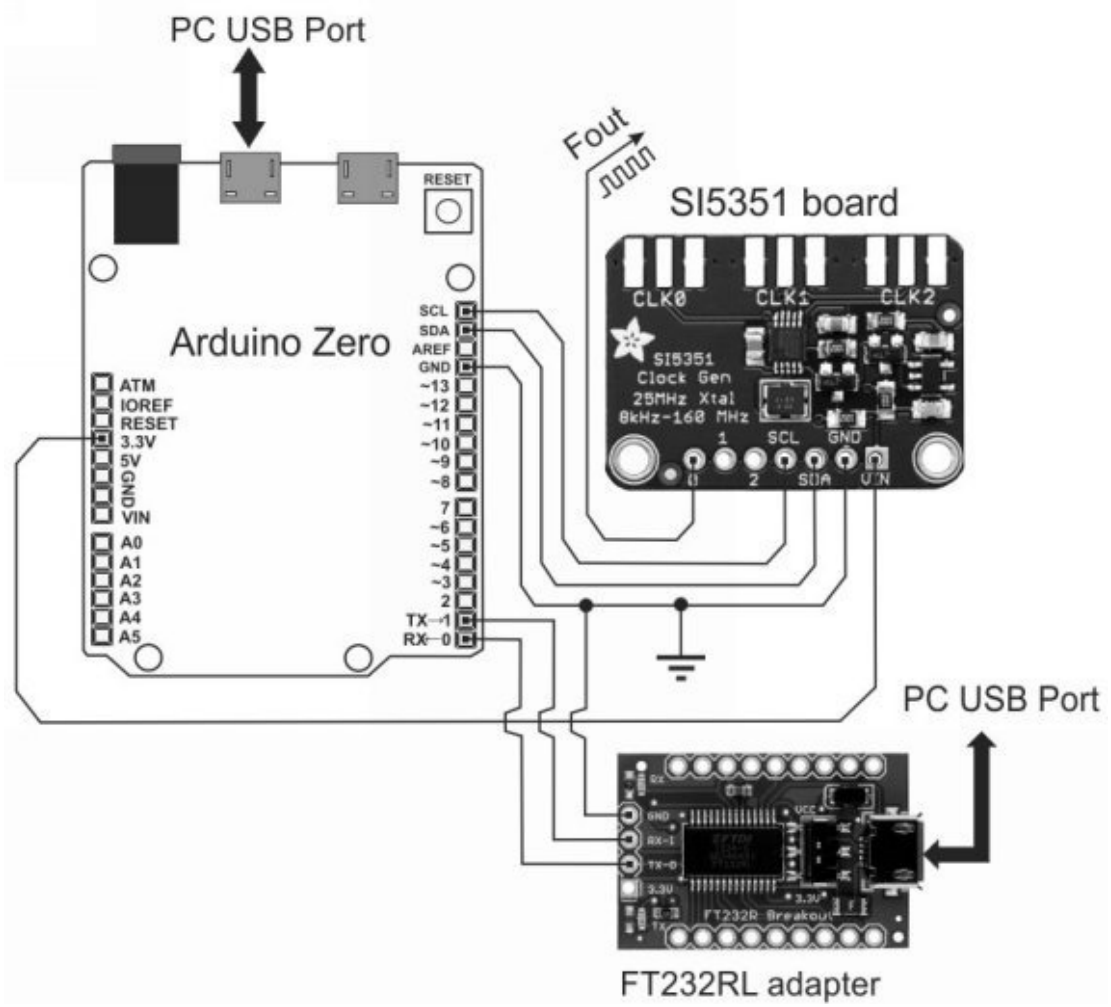


Fig.43

The connections between modules are detailed in the table below.

Arduino Zero pin	**Si5351 pin**	**FT232RL adapter**
SCL	SCL	—
SDA	SDA	—

3.3V	VIN	–
GND	GND	GND
TX→1	–	RX-I
RX←0	–	TX-O

The source code of the project will almost be the same as that shown in the previous section. The only thing to change is the serial interface which should be assigned "**Serial1**" instead of "**Serial**".

The source code of the Arduino Zero application is shown in **Listing 18**.

Listing 18.

```
#include <Wire.h>
#include <Adafruit_SI5351.h>

const double fPLLB = 616.66667;  // PLLB freq. = 616.66667 MHz
double fDiv = 128; // divider
double updFreq;

double param, fractPart, intPart;
int intParam2, intParam3;
int cmd = 0;

Adafruit_SI5351 clockgen = Adafruit_SI5351();

void setup(void)
{
  Serial1.begin(9600);
  Serial1.println("Si5351 Clockgen Test"); Serial1.println("");
```

```
  /* Initialise the sensor */
  if (clockgen.begin() != ERROR_NONE)
  {
    /* There was a problem detecting the IC … check your connections */
    Serial1.print("No Si5351 detected … Check your wiring or I2C ADDR!");
    while(1);
  }
  Serial1.println("OK!");

  /* FRACTIONAL MODE —> More flexible but introduce clock jitter */
  /* Setup PLLB to fractional mode @616.66667MHz (XTAL * 24 + 2/3) */
  /* Setup Multisynth 1 to 55.908 KHz (PLLB/172.34) */

  clockgen.setupPLL(SI5351_PLL_B, 24, 2, 3);
  Serial1.println("Output #0 is set to 50.0 KHz");
  clockgen.setupMultisynth(0, SI5351_PLL_B, 192, 71, 100);
  clockgen.setupRdiv(0, SI5351_R_DIV_128);
  /* Enable the clocks */
  clockgen.enableOutputs(true);
  Serial1.println("Enter a new frequency in KHz <0,###.###>:");
}

void loop(void)
{
while (Serial1.available() > 0)
  {

    // look for the next valid integer in the incoming serial stream:
    cmd = Serial1.parseInt();
    if (cmd == 0) // 0 = configure Si5351
```

```
    {
    updFreq = (double)(Serial1.parseFloat());
    param = (fPLLB / fDiv / updFreq) * 1000; // value of the parameters

    fractPart = modf(param, &intPart);
    intParam2 = (int)intPart;
    intParam3 = (int)(fractPart * 100);

    clockgen.enableOutputs(false);
    clockgen.setupPLL(SI5351_PLL_B, 24, 2, 3);
    clockgen.setupMultisynth(0, SI5351_PLL_B, intParam2, intParam3, 100);
    clockgen.setupRdiv(0, SI5351_R_DIV_64);
    clockgen.enableOutputs(true);
    }
  }
}
```

The C#. NET application running in Windows 10 produces the following output (**Fig.44**).

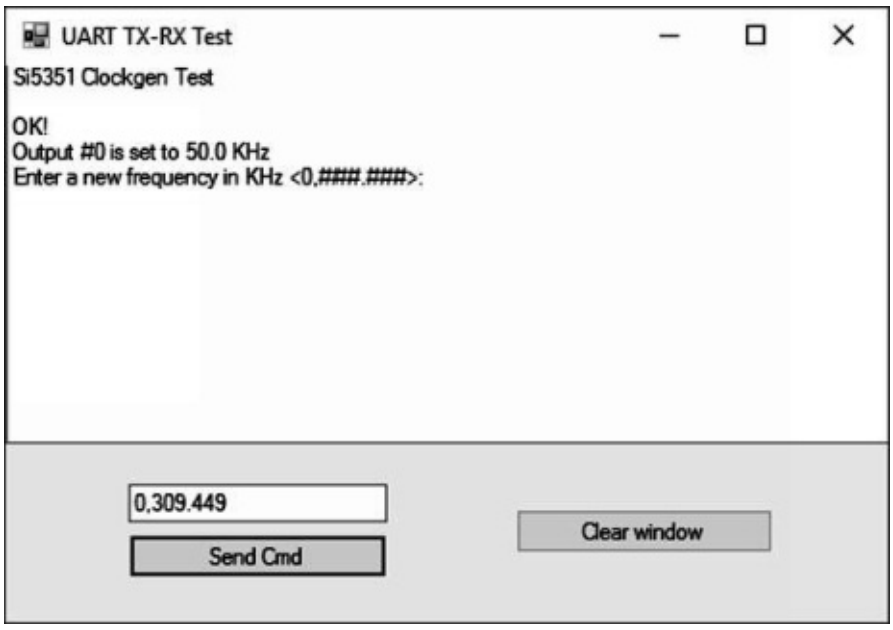


Fig.44

As you can see, the new frequency to be set is 309.449 KHz. After pressing the "**Send**

Cmd” button the new frequency of Si5351 will be set.

Project 6

This project illustrates designing the application which generates a sine wave signal on pin **A0** of Arduino Zero. The frequency of the sine wave signal is adjusted by the analog voltage (potentiometer **Rp**) applied to pin **A1**. In this application, the Si5351 device is used for clocking a DAC buffer.

The circuit diagram of the project is shown in **Fig.45**.

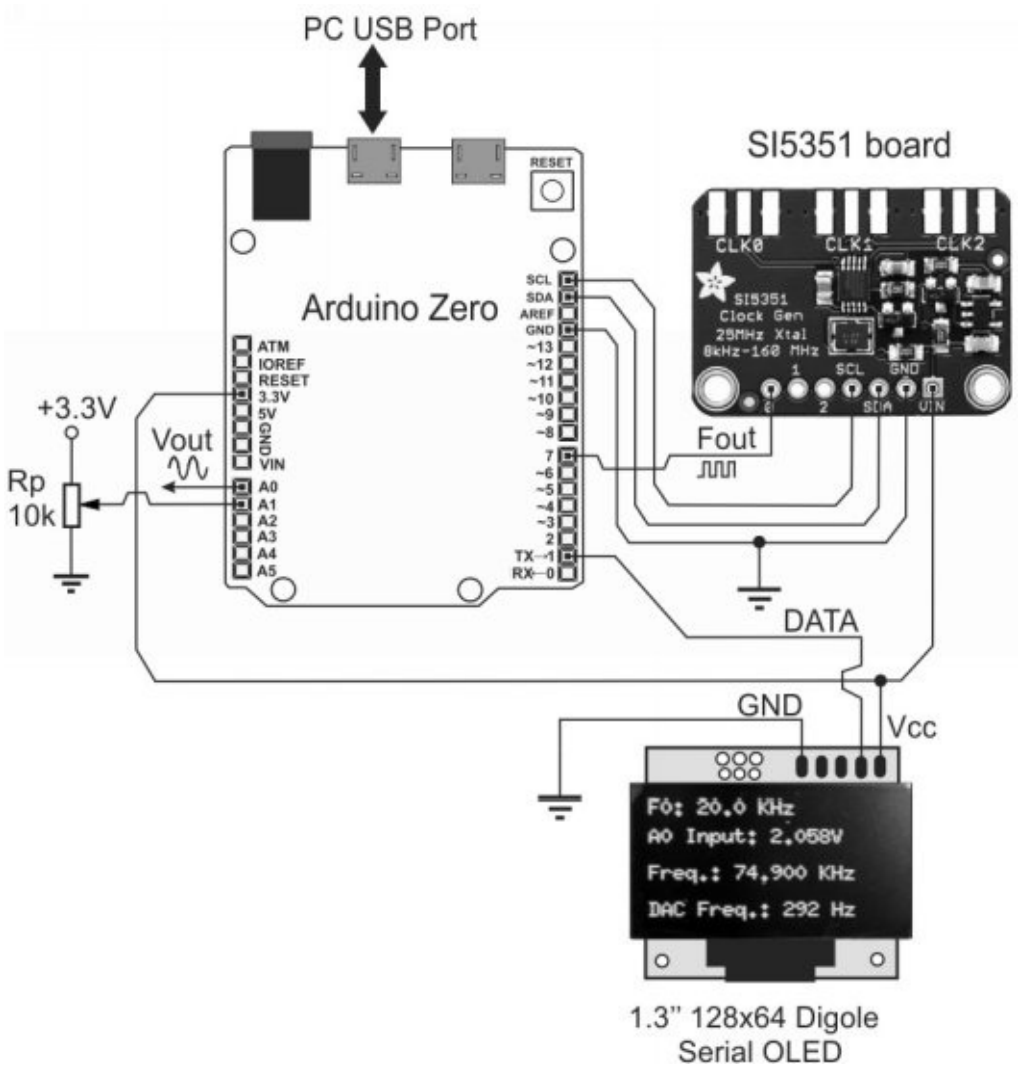


Fig.45

In this circuit, the analog input signal from the wiper of the potentiometer **Rp** is applied to the analog channel **A1** of Arduino Zero. The voltage level on **A1** adjusts the output frequency **Fout** (output **0**) of the Si5351 oscillator.

The rising edges of digital pulses on output **0** of Si5351 trigger the interrupt on pin **7** of Arduino Zero. The Interrupt Service Routine (ISR) associated with pin **7** writes the data from a DAC buffer to pin **A0** of Arduino Zero

The frequency of a sine wave signal on pin **A0** is linearly dependent on **Fout**, therefore the

changes of **Fout** will cause the changes of the output frequency on pin **A0**.

The connections between modules are detailed in the table below.

Arduino Zero pin	**Si5351 pin**	**Digole OLED pin**
SCL	SCL	—
SDA	SDA	—
3.3V	VIN	Vcc
GND	GND	GND
TX→1	—	DATA
“7”	“0” (output 0)	—

The Arduino source code is shown in **Listing 19**.

Listing 19.

```
#define _Digole_Serial_UART_  // using UART for data transfer

#include <DigoleSerial.h>

//——UART setup————
// Pin 1(TX)on Arduino is connected to RX line (“DATA”) on OLED

DigoleSerialDisp mydisp(&Serial1, 9600);

#include “si5351.h”
#include “Wire.h”
```

```
unsigned long baseFreq = 2000000; // init. Freq. = 20 KHz
int i1;

const long KFreq = 2667000; // the coefficient for calculating a freq. in KHz in the range 20 – 100 KHz
unsigned long newFreq, curFreq;
int binADC;
const float LSB = 3.33 / 1024;
float vin;
float fKHz;

const int NUM_SAMPLES = 256;
unsigned char SINE_TABLE[NUM_SAMPLES];
volatile int cnt = 0;
int pISR = 7;

const unsigned char fonts[] = {0, 6, 10};

Si5351 si5351;

void setup()
{
  // Start serial and initialize the Si5351
  analogReadResolution(10);
  analogWriteResolution(8); // 8-bit write resolution is set
  setLUT();

  pinMode(pISR, INPUT);
  attachInterrupt(pISR, writeDAC, RISING);

  Serial1.begin(9600);
  mydisp.begin();
```

```
  mydisp.clearScreen(); //CLear screen
  mydisp.disableCursor(); //disable cursor, enable cursore use: enableCursor();

  si5351.init(SI5351_CRYSTAL_LOAD_8PF, 0);

  // 20.0KHz
  si5351.set_pll(SI5351_PLL_FIXED, SI5351_PLLA);
  si5351.set_freq(baseFreq, SI5351_PLL_FIXED, SI5351_CLK0);

  delay(5000);
  }

void setLUT(void)
{
//sine period is 2*PI
const float step = (2*M_PI)/(float)NUM_SAMPLES;
float s;

// calculationa are in radians
for(int i = 0;i < NUM_SAMPLES;i++)
{
  s = sin(i * step );
  //s = s*s;
  SINE_TABLE[i] = (unsigned char) (128.0 + (s*127.0));
}
}

void writeDAC()
{
if (cnt >= 255)
    cnt = 0;
```

```
analogWrite(A0, SINE_TABLE[cnt]); // use DAC0
cnt++;
}

void loop()
{
  binADC = analogRead(1);
  vin = LSB * binADC;
  if ((vin >= 0.01) && (vin <= 3.0))
  {
    newFreq = (long)(vin * KFreq); // new freq. in KHz
    if (newFreq != curFreq)
    {
    curFreq = newFreq;
    si5351.set_freq(baseFreq + curFreq, SI5351_PLL_FIXED, SI5351_CLK0);
    }
  }

  fKHz = baseFreq + curFreq;

  mydisp.clearScreen();
  mydisp.setPrintPos(0, 0, _TEXT_);
  mydisp.setFont(fonts[2]);
  mydisp.print("F0: 20.0 KHz");

  mydisp.setPrintPos(0, 2, _TEXT_);
  mydisp.print("A0 Input: ");
  mydisp.print(vin, 3);
  mydisp.print("V");

  mydisp.setPrintPos(0, 4, _TEXT_);
```

```
  mydisp.print(“Freq.: “);
  mydisp.print(fKHz / 100000,3);
  mydisp.print(” KHz”);

  mydisp.setPrintPos(0, 6, _TEXT_);
  mydisp.print(“DAC Freq.: “);
  mydisp.print((long)(fKHz / 100 / 256));
  mydisp.print(” Hz”);
  delay(5000);
}
```

Direct Digital Synthesizer AD9850 in Arduino applications

Direct Digital Synthesis (DDS) is a method of producing an analog waveform (usually a sine wave) by generating a time-varying signal in digital form and then performing a digital-to-analog conversion. Because operations within a DDS device are primarily digital, this gives fast switching between output frequencies, fine frequency resolution, and operation over a broad spectrum of frequencies.

The ability to accurately produce and control waveforms of various frequencies and profiles has become a key requirement common to a number of industries. Many possibilities for frequency generation are open to a designer, but the DDS technique is rapidly gaining acceptance for solving frequency (or waveform) generation requirements in both communications and industrial applications because single-chip IC devices can generate programmable analog output waveforms simply and with high resolution and accuracy. By using modern DDS chips, we can obtain sine wave, triangular and rectangle signals with low-level total harmonic distortion (THD).

This section describes the programmable waveform oscillator using a popular DDS chip AD9850 which can produce the sine wave and square wave signals in a wide range of frequencies. The device output frequency is controlled through the SPI-compatible interface by the BeagleBone Black board. Before we start describing the hardware/software let’s see on how AD9850 works. The brief description that follows is taken from the datasheet on the device.

The AD9850 is a highly integrated device that uses advanced DDS technology coupled with an internal high speed, high performance D/A converter and comparator to form a complete, digitally programmable frequency synthesizer and clock generator function. When referenced to an accurate clock source, the AD9850 generates a spectrally pure, frequency/phase programmable, analog output sine wave. This sine wave can be used directly as a frequency source, or it can be converted to a square wave for agile-clock generator applications.

The AD9850’s innovative high speed DDS core provides a 32-bit frequency tuning word, which results in an output tuning resolution of 0.0291 Hz for a 125 MHz reference clock input. The AD9850’s circuit architecture allows the generation of output frequencies of up to one-half the reference clock frequency (or 62.5 MHz), and the output frequency can be digitally changed (asynchronously) at a rate of up to 23 million new frequencies per second. The device also provides five bits of digitally controlled phase modulation, which enables phase shifting of its output in increments of 180°, 90°, 45°, 22.5°, 11.25°, and any combination thereof. The AD9850 also contains a high speed comparator that can be

configured to accept the (externally) filtered output of the DAC to generate a low jitter square wave output. This facilitates the device's use as an agile clock generator function.

The frequency tuning, control, and phase modulation words are loaded into the AD9850 via a parallel byte or serial loading format. The parallel load format consists of five iterative loads of an 8-bit control word (byte). The first byte controls phase modulation, power-down enable, and loading format; Bytes 2 to 5 comprise the 32-bit frequency tuning word. Serial loading is accomplished via a 40-bit serial data stream on a single pin. The AD9850 Complete DDS uses advanced CMOS technology to provide this breakthrough level of functionality and performance on just 155 mW of power dissipation (3.3 V supply).

The following projects can use one of numerous low-cost ready-to-use modules with AD9850. One of these is shown in **Fig.46**.

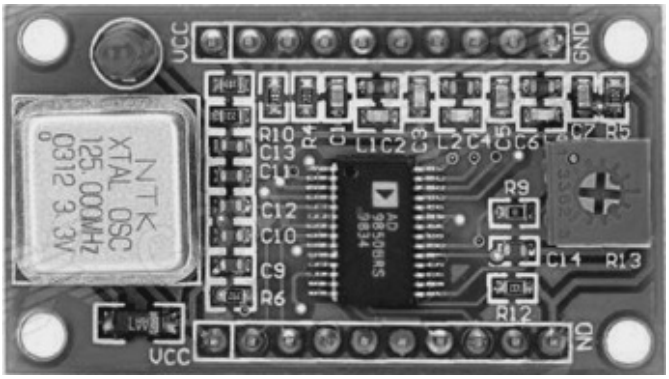


Fig.46

Most AD9850 modules use the reference clock of 125 MHz, therefore the output frequency can reach 62.5 MHz. As to this module, it is stated that the real output frequency can reach 40 MHz because of the low-pass filter placed at the DAC output of AD9850. However, the developers experimenting with this module can pull up the output to its maximal frequency.

The first project in this section demonstrates the basic programming techniques applied to the AD9850 device.

Project 1

The project uses the AD9850 module shown in **Fig.46**. The circuit diagram (**Fig.47)** shows connections between the AD9850 board and Arduino Zero.

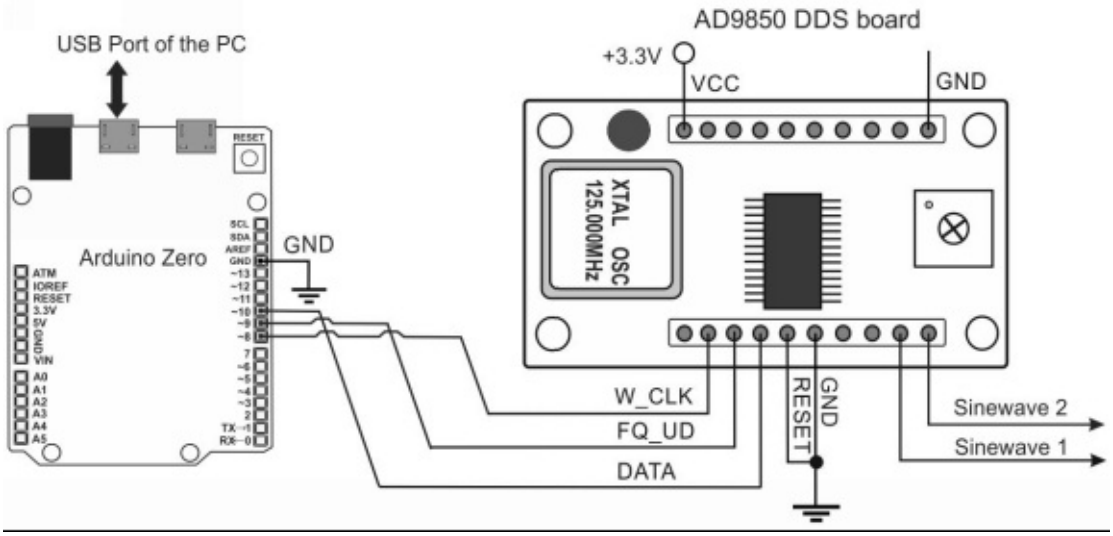


Fig.47

In this circuit, the **DATA** line (pin **D7** of AD9850 IC) is connected to pin **10** of Arduino Zero. The framing signal **FQ_UD** goes to pin **9** and the clock input **W_CLK** of AD9850 goes to pin **8** of the Arduino Zero board. The power to the AD9850 board (pin "**VCC**") can be taken from pin "**3.3V**" on the Arduino Zero board.

The sine wave output signal can be taken from either **Sinewave 1** or **Sinewave 2** pin of the AD9850 board.

Note that the connections in circuit shown in **Fig.47** are specific for my own AD9850 board purchased at www.ebay.com; the pin configuration on other AD9850 boards may be different, so you must carefully examine your own AD9850 board before connecting it to Arduino Zero!

To calculate the tuning word for configuring the AD9850 output frequency we need to know the relationship of the output frequency, reference clock, and tuning word of AD9850. This is determined by the formula taken from the datasheet on the device (**Fig.48)**.

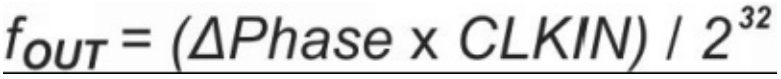


Fig.48

Here **ΔPhase** is the value of the 32-bit tuning word, **CLKIN** is the input reference clock frequency in MHz and $\mathbf{f_{OUT}}$ is the output frequency in MHz.

The following source code (**Listing 20**) allows to build the application that adjusta the output frequency according the above formula.

Listing 20.

```
const int W_CLK = 8;
const int FQ_UD = 9;
const int DAT = 10;

unsigned long freqWord;
unsigned long mask = 0x1;
unsigned long tmp;
int wcnt;

void setup() {
  // put your setup code here, to run once:
pinMode(W_CLK, OUTPUT);
pinMode(FQ_UD, OUTPUT);
pinMode(DAT, OUTPUT);

digitalWrite(W_CLK, HIGH);
digitalWrite(W_CLK, LOW);
digitalWrite(FQ_UD, HIGH);
digitalWrite(FQ_UD, LOW);
setFreq(1067000); // freq. in Hz
}
```

```
void loop() { }

void setFreq(unsigned long freq)
{
  freqWord = (long)(freq * pow(2,32) / 125000000); // ref. clock = 125 MHz
  for (wcnt = 0; wcnt < 32; wcnt++)
  {
    digitalWrite(W_CLK, LOW);
    digitalWrite(DAT, freqWord & mask);
    digitalWrite(W_CLK, HIGH);
    freqWord = freqWord >> 1;
  }
  for (wcnt = 32; wcnt < 40; wcnt++) //  # writing the last byte = 0x0, bits W32—>W39
  {
    digitalWrite(W_CLK, LOW);
    digitalWrite(DAT, LOW);
    digitalWrite(W_CLK, HIGH);
  }
  digitalWrite(FQ_UD, HIGH);
}
```

In this source code, the **setFreq()** function sets the frequency of the sine wave output of AD9850. The function takes a single parameter that is the desired frequency in Hz. When invoked, the function calculates the binary code corresponding to the output frequency and stores the value in the **freqWord** variable by executing the following statement:

```
freqWord = freq * pow(2, 32) / 125000000
```

Then the **freqWord** bits are written to the AD9850 control registers using two **for()** loops. The **W_CLK**, **FQ_UD** and **DAT** constants are assigned the pins being used in the SPI-compatible interface.

Project 2

This project allows to build a simple sine wave generator controlled by the **Touch Pot** board described earlier in this guide. The value of the frequency adjusted by **Touch Pot** is reflected in the **Serial Monitor** window. The circuit diagram of the project is shown in **Fig.49**.

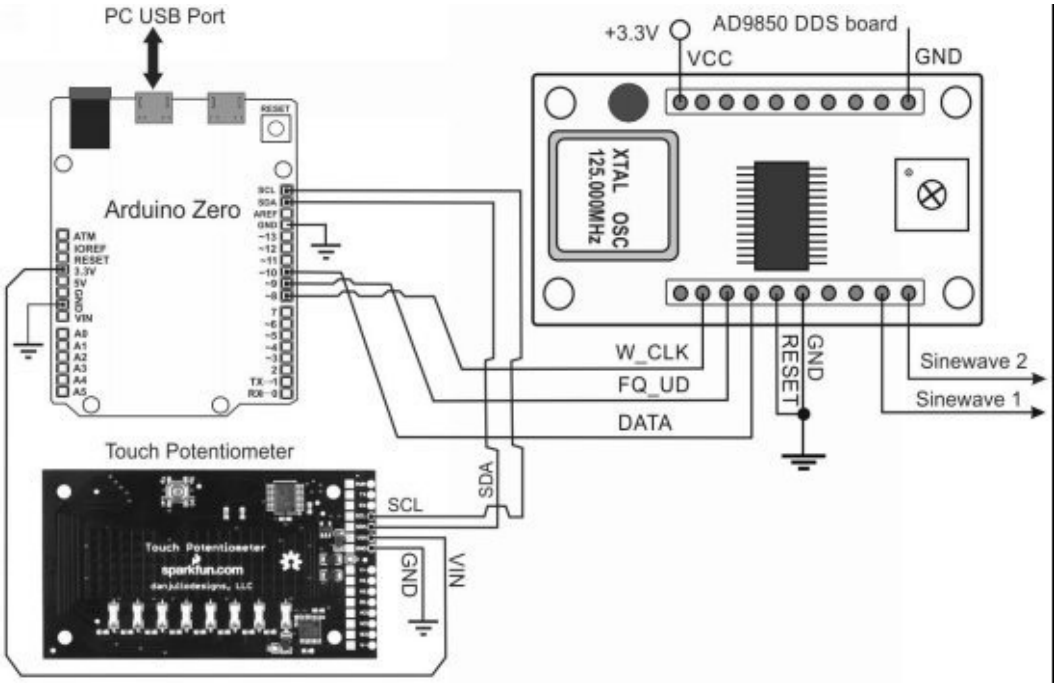


Fig.49

In this circuit, the **DATA** line (pin **D7** of AD9850 IC) is connected to pin **10** of Arduino Zero. The framing signal **FQ_UD** goes to pin **9** and the clock input **W_CLK** of AD9850 goes to pin **8** of the Arduino Zero board.

The **Touch Pot** pins **SCL** and **SDA** are wired to corresponding pins on the Arduino Zero board. The power to AD9850 and Touch Pot can be taken from pin "**3.3V**" on the Arduino Zero board.

The sine wave output signal can be taken from either **Sinewave 1** or **Sinewave 2** pin of the AD9850 board.

The source code of the application is shown in **Listing 21**.

Listing 21.

```
#include "Wire.h"
```

```
// Touch.Pot parameters

int i2cAddr = 8; // Direct access at i2cAddr, indirect registers at i2cAddr+1
uint8_t prevValue;
uint8_t curValue;

unsigned long initFreq = 500; // initial frequency of AD9850 = 500 Hz
const long K = 50;
unsigned long Freq;

// AD9850 parameters

const int W_CLK = 8;
const int FQ_UD = 9;
const int DAT = 10;

unsigned long freqWord;
unsigned long mask = 0x1;
unsigned long tmp;
int wcnt;

void setup() {
  // put your setup code here, to run once:

Serial.begin(9600);
Wire.begin();

// Demonstrate access to Touch Potentiometer registers
WriteTpReg(1, 10); // set to 50% by writing to register 1
curValue = ReadTpReg(1); // read back value just set
```

```
// Configuring AD9850 pins

pinMode(W_CLK, OUTPUT);
pinMode(FQ_UD, OUTPUT);
pinMode(DAT, OUTPUT);

// Set initial frequency of AD9850

Freq = initFreq + K*curValue;
setFreq(Freq); // freq. in Hz // the freq. will be set to 500 + 10*50 = 1000 Hz
prevValue = curValue;
}

void loop() {
  // put your main code here, to run repeatedly:
delay(1000); // Reading each 1 second
// Demonstrate direct access to Touch Potentiometer value

curValue = ReadTpValue(); // faster I2C access than register read
if (curValue != prevValue)
{
  Freq = initFreq + K*curValue;
  setFreq(Freq);
  delay(10);
  Serial.print("The Touch Pot value: ");
  Serial.print(curValue);
  Serial.print(", AD9850 freq. = ");
  Serial.print(Freq);
  Serial.println(" Hz");
  prevValue = curValue;
```

```
}
}

void setFreq(unsigned long freq)
{
  freqWord = (long)(freq * pow(2,32) / 125000000); // ref. clock = 125 MHz
  digitalWrite(W_CLK, HIGH);
  digitalWrite(W_CLK, LOW);
  digitalWrite(FQ_UD, HIGH);
  digitalWrite(FQ_UD, LOW);
  for (wcnt = 0; wcnt < 32; wcnt++)
  {
    digitalWrite(W_CLK, LOW);
    digitalWrite(DAT, freqWord & mask);
    digitalWrite(W_CLK, HIGH);
    freqWord = freqWord >> 1;
  }
  for (wcnt = 32; wcnt < 40; wcnt++) //  # writing the last byte = 0x0, bits W32—>W39
  {
    digitalWrite(W_CLK, LOW);
    digitalWrite(DAT, LOW);
    digitalWrite(W_CLK, HIGH);
  }
  digitalWrite(FQ_UD, HIGH);
}

// Write a Touch Potentiometer register

void WriteTpReg(uint8_t addr, uint8_t data)
{
Wire.beginTransmission(i2cAddr+1);
```

```
Wire.write('W');
Wire.write(addr);
Wire.write(data);
Wire.endTransmission();
}

// Get the Touch Potentiometer value

uint8_t ReadTpValue()
{
Wire.requestFrom(i2cAddr, 1);
if (Wire.available())
{
  return Wire.read();
}
else
{
  return 0;
}
}

// Read a Touch Potentiometer register

uint8_t ReadTpReg(uint8_t addr)
{
Wire.beginTransmission(i2cAddr+1);
Wire.write('R');
Wire.write(addr);
Wire.endTransmission();
Wire.requestFrom(i2cAddr+1, 1);
if (Wire.available())
```

```
{
  return Wire.read();
}
else
{
return 0;
}
}
```

Project 3

This project is the modified version of the previous one. The difference is that the value of an output frequency is transferred to the serial OLED display. The circuit diagram of the project is shown in **Fig.50**.

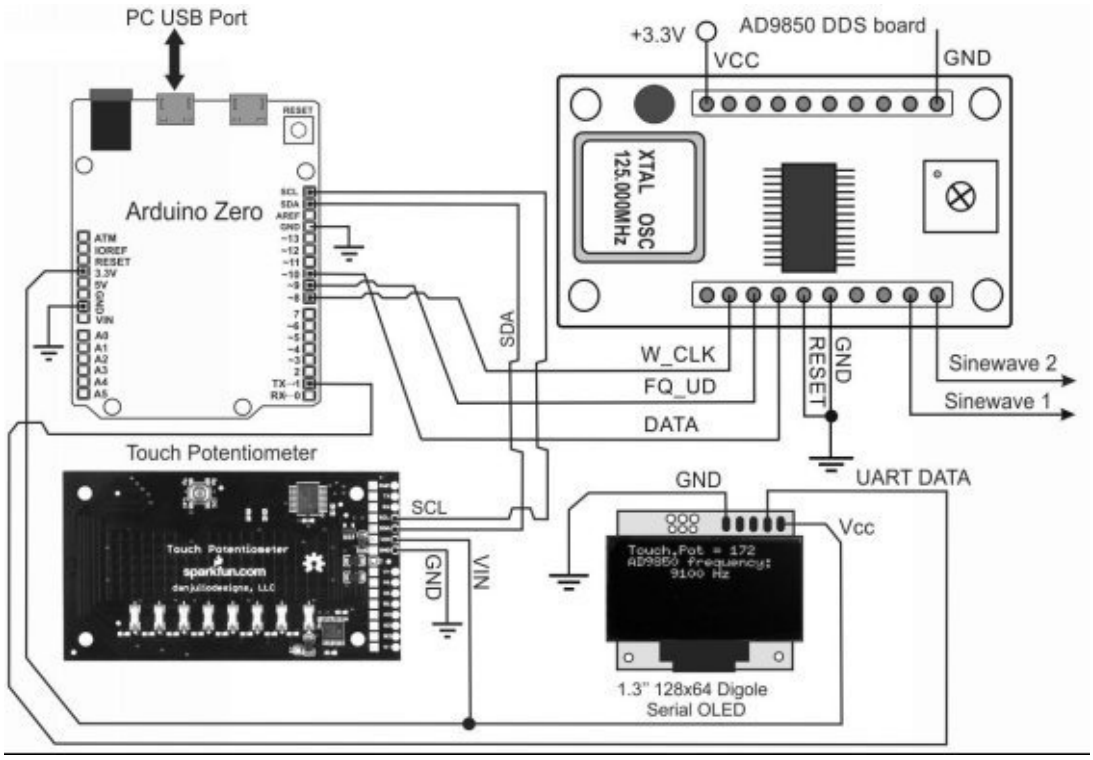


Fig.50

The connections between devices are the same as in the previous projects. The "**DATA**" pin on the OLED display is connected to pin "**TX→1**" on the Arduino Zero Board. The power to the OLED display (pin "**VCC**") is fed from pin "**3.3V**" on the Arduino Zero board.

The **Touch Pot** pins **SCL** and **SDA** are wired to corresponding pins on the Arduino Zero board. The power to AD9850 and Touch Pot can be taken from pin "**3.3V**" on the Arduino Zero board.

The sine wave output signal can be taken from either **Sinewave 1** or **Sinewave 2** pin of the AD9850 board.

The source code of the application is shown in **Listing 22**.

Listing 22.

```
#include "Wire.h"
```

```
#define _Digole_Serial_UART_  // using UART for data transfer

#include <DigoleSerial.h>

//——UART setup————
// Pin 1(TX)on Arduino is connected to RX line (“DATA”) on OLED

DigoleSerialDisp mydisp(&Serial1, 9600);

const unsigned char fonts[] = {0, 6, 10};

// Touch Pot parameters

int i2cAddr = 8; // Direct access at i2cAddr, indirect registers at i2cAddr+1
uint8_t prevValue;
uint8_t curValue;

unsigned long initFreq = 500; // initial frequency of AD9850 = 500 Hz
const long K = 50;
unsigned long Freq;

// AD9850 parameters

const int W_CLK = 8;
const int FQ_UD = 9;
const int DAT = 10;

unsigned long freqWord;
unsigned long mask = 0x1;
unsigned long tmp;
int wcnt;
```

```
void setup() {
  // put your setup code here, to run once:

Serial1.begin(9600);
mydisp.begin();
mydisp.disableCursor(); //disable a cursor, to enable a cursor use: enableCursor();

Wire.begin();

WriteTpReg(1, 10); // dig.pot initial value is set to 10/256
curValue = ReadTpReg(1);

// Configuring AD9850 pins

pinMode(W_CLK, OUTPUT);
pinMode(FQ_UD, OUTPUT);
pinMode(DAT, OUTPUT);

// Set initial frequency of AD9850

Freq = initFreq + K*curValue;
setFreq(Freq); // the freq. in Hz will be set to 500 + 10*50 = 1000 Hz

mydisp.clearScreen(); //CLear screen
mydisp.setFont(fonts[2]); // set font 6x10

mydisp.setPrintPos(0, 0, _TEXT_);
mydisp.print("Touch.Pot = ");
mydisp.print(curValue);
```

```
mydisp.setPrintPos(0, 1, _TEXT_);
mydisp.print(“Initial AD9850 freq.:”);

mydisp.setPrintPos(3, 2, _TEXT_);
mydisp.print(Freq);
mydisp.print(” Hz”);
prevValue = curValue;
}

void loop() {
  // put your main code here, to run repeatedly:
delay(1000); // Reading Touch Pot every 1 s

curValue = ReadTpValue();
if (curValue != prevValue)
{
  Freq = initFreq + K*curValue;
  setFreq(Freq);
  delay(10);

  mydisp.clearScreen(); //Clear screen
  mydisp.setFont(fonts[2]); // set font 6x10

  mydisp.setPrintPos(0, 0, _TEXT_);
  mydisp.print(” Touch.Pot = “);
  mydisp.print(curValue);

  mydisp.setPrintPos(0, 1, _TEXT_);
  mydisp.print(” AD9850 frequency:”);

  mydisp.setPrintPos(6, 2, _TEXT_);
```

```
  mydisp.print(Freq);
  mydisp.print(" Hz");

  prevValue = curValue;
}
}

void setFreq(unsigned long freq)
{
  freqWord = (long)(freq * pow(2,32) / 125000000); // ref. clock = 125 MHz
  digitalWrite(W_CLK, HIGH);
  digitalWrite(W_CLK, LOW);
  digitalWrite(FQ_UD, HIGH);
  digitalWrite(FQ_UD, LOW);
  for (wcnt = 0; wcnt < 32; wcnt++)
  {
    digitalWrite(W_CLK, LOW);
    digitalWrite(DAT, freqWord & mask);
    digitalWrite(W_CLK, HIGH);
    freqWord = freqWord >> 1;
  }
  for (wcnt = 32; wcnt < 40; wcnt++) //  writing the last byte = 0x0, bits W32—>W39
  {
    digitalWrite(W_CLK, LOW);
    digitalWrite(DAT, LOW);
    digitalWrite(W_CLK, HIGH);
  }
  digitalWrite(FQ_UD, HIGH);
}

void WriteTpReg(uint8_t addr, uint8_t data)
```

```
{
Wire.beginTransmission(i2cAddr+1);
Wire.write('W');
Wire.write(addr);
Wire.write(data);
Wire.endTransmission();
}

uint8_t ReadTpValue()
{
Wire.requestFrom(i2cAddr, 1);
if (Wire.available())
{
  return Wire.read();
}
else
{
  return 0;
}
}

uint8_t ReadTpReg(uint8_t addr)
{
Wire.beginTransmission(i2cAddr+1);
Wire.write('R');
Wire.write(addr);
Wire.endTransmission();
Wire.requestFrom(i2cAddr+1, 1);
if (Wire.available())
{
  return Wire.read();
```

```
}
else
{
return 0;
}
}
```

Processing DDS waveforms

The sine wave output of AD9850 can also be used for creating a square wave signal by introducing a simple converter circuit (**Fig.51)**.

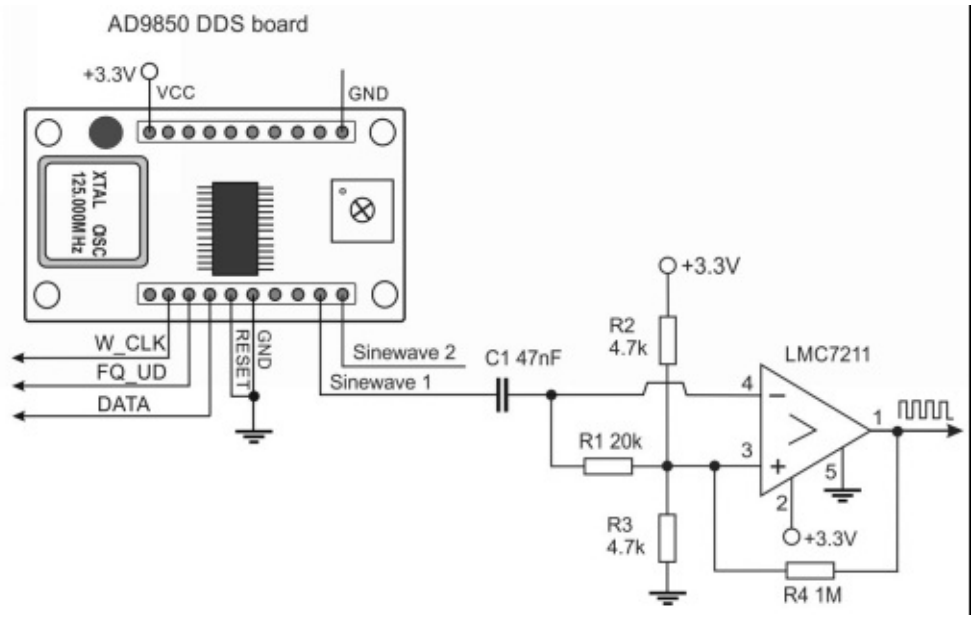


Fig.51

In this circuit, the sine wave signal (**Sinewave 1** output) is applied to the zero-crossing detector with comparator LMC7211 through the coupling capacitor C1. The TTL-compatible rectangular pulse train can be taken from pin **1** of the comparator.

Almost any general purpose comparator (TLC3702, LM2903, etc.) fits this circuit. **Note** that for high frequencies it would be better to use some high-speed comparator with low propagation delays (MAX961, LT1719, etc.).

The amplitude of an output signal of Ad9850 is about 0.6V that may be too low for many applications. A simple BJT transistor circuit (**Fig.52**) provide the signal with the amplitude larger than that of the input signal.

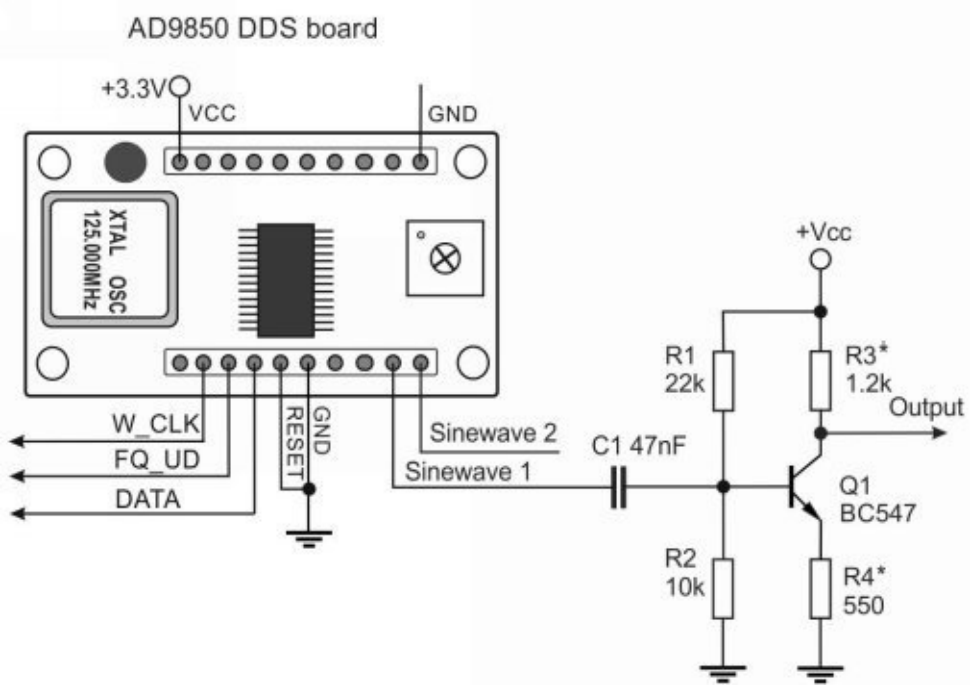


Fig.52

This amplifier circuit is built around the bipolar transistor BC547 (Q1) arranged as a classic amplifier with a common emitter. The R1-R2 network provides the bias to Q1. The gain **G** of this circuit is close to

G = R3/R4

In our case, G = 1.2/0.55 ≈ 2.2, therefore for the input signal of 0.6V applied to the base of Q1 the output signal on the collector of Q1 will be as large as 0.6 x 2.2 ≈ 1.32V .

Measuring parameters of signals

Due to availability of a high performance ATSAMD21G18A (Cortex-M0) CPU we can apply Arduino Zero in systems measuring parameters of signals. CPU and peripherals of Aruino Zero can process signals much faster than Arduino Uno, therefore, the measurements performed with Arduino Zero are much more precious. The following sections are dedicated to design of simple but effective measurement systems.

Measuring the period and frequency of digital signals

One of common tasks implemented by embedded systemes is the measurement of a period (frequency) of digital pulses. The following project illustrates one possible approach for implementing such a task. The idea behind this approach is extremely simple: the program code counts the number of pulses passing during 1 s interval. When counting is complete, this number gives us the frequency of an incoming signal. Knowing the frequency allows to calculate the period of a pulse train – this will be a value reciprocal of the frequency.

With Arduino Zero the frequency measurement can be accomplished by using interrupts. The rising (falling) edges of the pulses being measured trigger the interrupt on some dedicated pin. The Interrupt Service Routine (ISR) associated with a given interrupt simply increments a counter. After 1 s interval expires, counting is stopped. The results of measurement are then output on the OLED display.

The circuit diagram of the project is shown in **Fig.53**.

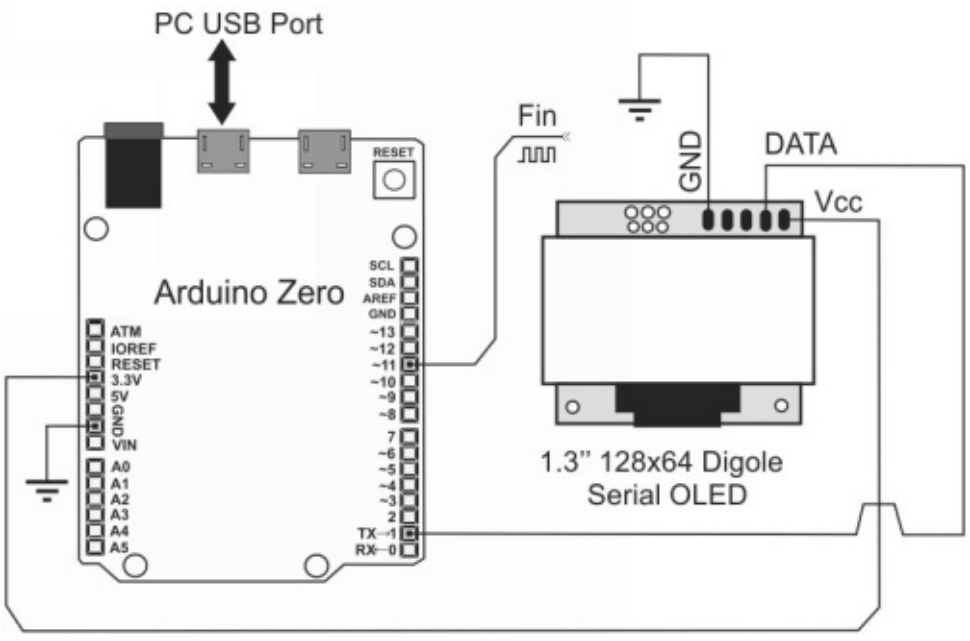


Fig.53

In this circuit, the input digital signal **Fin** goes to pin **11** configured as input. The results of measurements are transferred to the OLED display (pin "**DATA**") via a serial interface (pin "**TX→1**" of Arduino Zero). The OLED display is fed by 3.3V DC source (pin "**3.3V**" of Arduino Zero).

The source code of the application is shown in **Listing 23**.

Listing 23.

```
#define _Digole_Serial_UART_  // using UART for data transfer

#include <DigoleSerial.h>

//——UART setup————
// Pin 1 (TX) on Arduino is connected to RX line (“DATA”) on OLED

DigoleSerialDisp mydisp(&Serial1, 9600);

long int cnt = 0;
const int fPin = 11;
float freq;

const unsigned char fonts[] = {0, 6, 10};

void setup(void)
{
pinMode(11, INPUT);
Serial1.begin(9600);
mydisp.begin();
mydisp.clearScreen(); //Clear screen
mydisp.disableCursor(); //disable cursor, enable cursore use: enableCursor();
attachInterrupt(fPin, cntISR, RISING);
}

void cntISR()
{
cnt++;
```

```
}

void loop(void)
{
cnt = 0;
delay(1000);
freq = (float)cnt;

mydisp.clearScreen();
mydisp.setFont(fonts[0]);
mydisp.setPrintPos(1, 1, _TEXT_);
mydisp.print("Freq. on D11:");
mydisp.setPrintPos(3, 2, _TEXT_);
mydisp.print(freq / 1000, 3);
mydisp.println(" KHz");

delay(3000);
}
```

In this source code, we configure the interrupt triggered by the rising edge of an incoming pulse on pin **11:**

```
attachInterrupt(fPin, cntISR, RISING);
```

The ISR procedure **cntISR()** when invoked simply increments a counter (variable **cnt**).
The following sequence starts measuring during 1 s:

```
cnt = 0;
delay(1000);
freq = (float)cnt;
```

The result of a single measurement is saved in variable **freq**.

Measuring the pulse width and duty cycle of digital signals

This section is dedicated to the design of an application performing the high precision measurements of the period, pulse width and duty cycle of a digital pulse train. The application uses the LTC2644-L12 device from Linear Technology. LTC2644-L12 belongs to a family of dual 12-, 10-, and 8-bit PWM-to-Voltage output DACs with an integrated high accuracy, low drift, 10ppm/°C reference in a 12-lead MSOP package.

The LTC2644-L12 device measures the period and pulse width of the PWM input signals and updates the voltage output DACs after each corresponding PWM input rising edge. The DAC outputs update and settle to 12-bit accuracy within 8µs typically and are capable of sourcing and sinking up to 5mA (3V) or 10mA (5V), eliminating voltage ripple and replacing slow analog filters and buffer amplifiers. More details concerning the LTC2644 chip can be found in the datasheet on this device.

The circuit diagram of the project is shown in **Fig.54**.

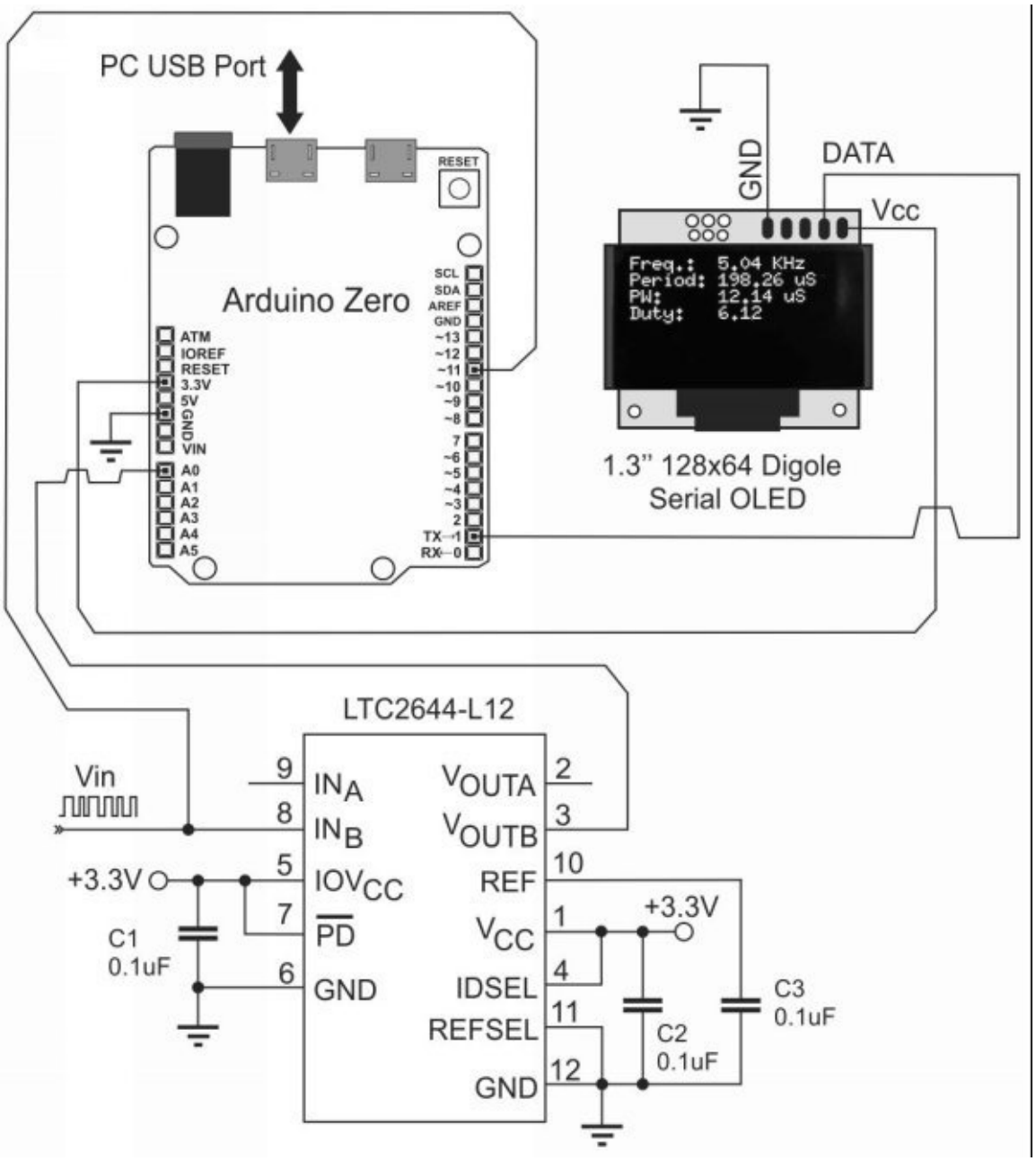


Fig.54

In this circuit, a pulse-width modulated input frequency between 30 Hz and 6.25 KHz (for devices with 12-bit resolution) is applied to input $\mathbf{IN_B}$ (pin **8**) of LTC2644-L12 chip. After each $\mathbf{IN_B}$ rising edge, the part calculates the duty cycle based upon the pulse width and period and updates DAC channel $\mathbf{V_{OUTB}}$. Logic levels are referenced to $\mathbf{IOV_{CC}}$ that is 3.3V for this circuit.

The DAC analog voltage is taken from the $\mathbf{V_{OUTB}}$ output (pin **3**) of LTC2644-L12. This voltage can be calculated using the following equation:

$$V_{OUTB} = V_{REF} \times (t_{PWHB} / T_B) \quad (1)$$

where $\mathbf{V_{REF}}$ is 2.5V for internal reference mode (our case), $\mathbf{t_{PWHB}}$ is the pulse width of the preceding $\mathbf{IN_B}$ period and $\mathbf{T_B}$ is the time between the two most recent $\mathbf{IN_B}$ rising edges (period). The DAC output voltage $\mathbf{V_{OUTB}}$ is then fed to the analog input channel **A0** of Arduino Zero for calculating the duty cycle that is equal to $\mathbf{t_{PWHB}/T_B}$ (see formula (1)).

The program code calculates the frequency of the pulse train arriving on pin **11** of Arduino Zero. Both measured values (frequency and duty cycle) are then used for calculating the pulse width of this sequence of pulses.

The results are transferred from of Arduino Zero (pin "**TX → 1**") to the OLED display (pin "**DATA**") via a serial interface.

The source code of the application is shown in **Listing 24**.

Listing 24.

```
#define _Digole_Serial_UART_  // using UART for data transfer
#include <DigoleSerial.h>

//——UART setup————
//  Pin 1 (TX) on Arduino is connected to RX line ("DATA") on OLED

DigoleSerialDisp mydisp(&Serial1, 9600);

long int cnt = 0;
const int fPin = 11;

double freq;
double period;
double pWidth;
double dutyCycle;

int binADC;
```

```
const double LSB = 3.3 / 4096.0;
double Vin;
const double Vref = 2.5; // an internal ref. of LTC2644 is used
const unsigned char fonts[] = {0, 6, 10};

void setup(void)
{
analogReadResolution(12);
pinMode(fPin, INPUT); // pin 11 of Arduino is configured as input

Serial1.begin(9600);
mydisp.begin();
mydisp.clearScreen(); //Clear screen
mydisp.disableCursor(); //disable cursor, enable cursore use: enableCursor();
interrupts();
attachInterrupt(fPin, cntISR, RISING);
delay(100);
}

void cntISR()
{
cnt++;
}

void loop(void)
{

// Measuring the frequency of a pulse train on pin 11 of Zero
cnt = 0;
delay(1000);
freq = (double)cnt;
```

```
delay(100);
period = 1000000 / freq; // period in uS

// Measuring a Duty Cycle and Pulse Width using LTC2644-12
// A pulse train goes to pin 8, pin 3 is wired to input A0 of Zero
// An input frequency should range from 30 Hz to 6.25 KHz

binADC = analogRead(0);
Vin = LSB * binADC;
dutyCycle = Vin / Vref;
pWidth = dutyCycle * period; // a pulse width in uS

mydisp.clearScreen();
mydisp.setFont(fonts[2]);

mydisp.setPrintPos(0, 0, _TEXT_);
mydisp.print("Freq.:  ");
mydisp.print(freq / 1000, 2);
mydisp.println(" KHz");

mydisp.setPrintPos(0, 1, _TEXT_);
mydisp.print("Period: ");
mydisp.print(period, 2);
mydisp.print(" uS");



mydisp.setPrintPos(0, 2, _TEXT_);
mydisp.print("PW:     ");
mydisp.print(pWidth,2);
mydisp.print(" uS");
```

```
mydisp.setPrintPos(0, 3, _TEXT_);
mydisp.print(“Duty:   “);
mydisp.print(dutyCycle * 100, 2);

delay(3000);
}
```

Measuring the frequency of analog signals

Often we need to measure the frequency (period) of periodical analog signals (sinewave, triangular, sawtooth, etc.) produced by analog sensors and/or external analog circuitry. Obviously, we can't directly process such signals in digital domain until we convert them into sequence of digital pulses. In addition, almost every signal contains unwanted component called ″noise″. Noise is a random fluctuation in an electrical signal when unpredictable variations in the measured signal from moment to moment or from measurement to measurement occur.

We can significantly reduce a noise component by applying a simple active low-pass or bandpass filter with appropriate parameters. The filter can be followed by a zero-crossing detector which converts a random analog signal into a TTL-compatible rectangular pulse train suitable for processing by the Arduino program code.

This approach is illustrated by the circuit shown in **Fig.55**.

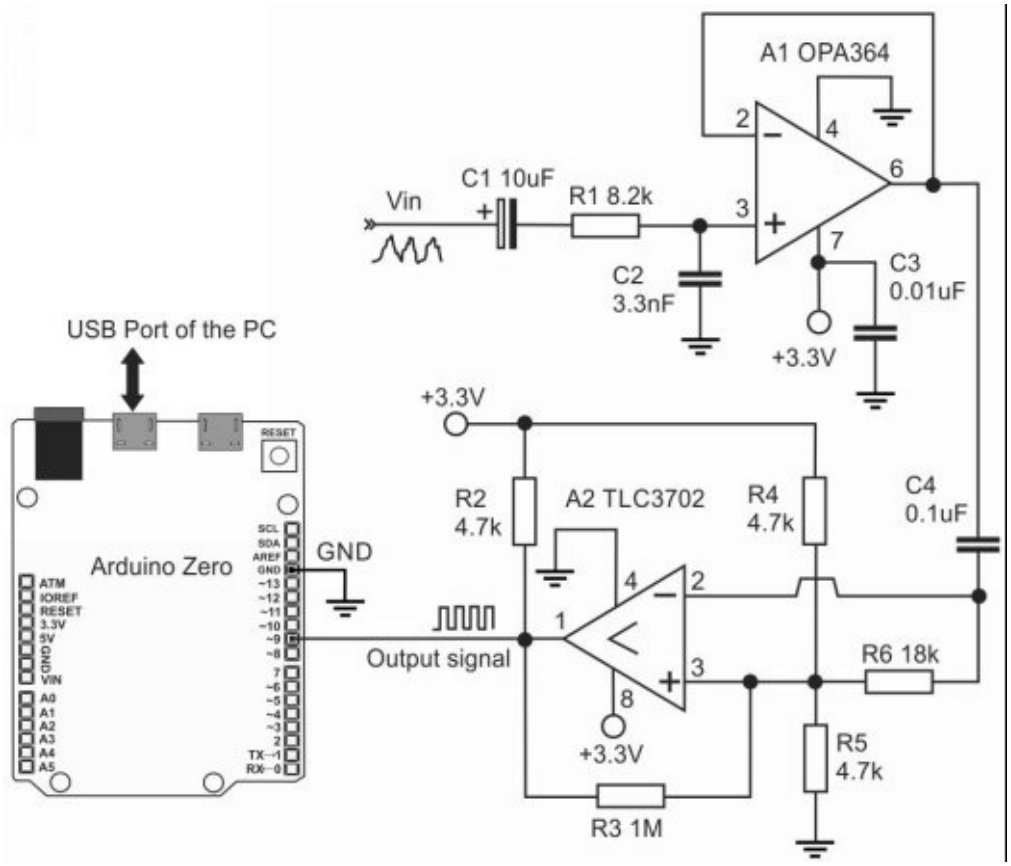


Fig.55

In this circuit, the analog input signal **Vin** goes to the low-pass active filter (R1-C2 network and op-amp OPA364) through the coupling capacitor C1. The cut-off frequency $\mathbf{f_{cut\text{-}off}}$ of this filter is determined by the expression

$f_{cut\text{-}off} = 1/2 \times \pi \times R1 \times C2$

that gives us the value close to 6 KHz. By altering the values of R1 and/or C2 we can extend or narrow cut-off frequency of the filter.

The filtered signal is then fed to the next stage arranged as a zero-crossing detector with the comparator TLC3702 (A2). The zero-crossing detector converts the input signal into a TTL-compatible pulse train picked up by the Arduino Zero board on pin **9**.

The test signals of various waveforms (sine wave, sawtooth, triangular and rectangular) applied to this circuit were taken from the LabVIEW virtual signal generator based upon NI PCI-6221 DAQ module. The frequencies of these signals changed from 100 Hz through 5000 Hz and amplitudes changed from 0.15V to 2V. The test signals were deliberately combined with low-level noise of white and 1/F types.

This circuit is capable of reliably measuring the frequency of analog signals (sine wave, triangle and sawtooth) with amplitudes ranging from about 0.15 V to 1.9V. The frequency of signals may range from several tens Hz to 5 KHz.

Real-Time Counter in Arduino applications

A microcontroller ATSAMD21G18 used in Arduino Zero has an internal Real-Time Counter (RTC) which together with the **RTCZero** library allows to implement various useful tasks.

A real-time counter keeps track of the current time and that can be used in order to program actions at a certain time. Most RTCs use a crystal oscillator (like in the Arduino Zero) whose frequency is 32.768 kHz (same frequency used in quartz clocks and watches). Namely this the frequency equal to 2^{15} cycles per second and so is a convenient rate to use with simple binary counter circuits. Furthermore the RTC can continue to operate in any sleep mode, so it can be used to wake up the device from sleep modes in a programmed way.

To use functions from the library we should install the **RTCZero** library first by choosing the "**Sketch**"→"**Include Library**"→"**Manage Libraries…**" option. If this library has been listed, click "**Install**". After installation has succeeded, the following screen appears (**Fig.56**).

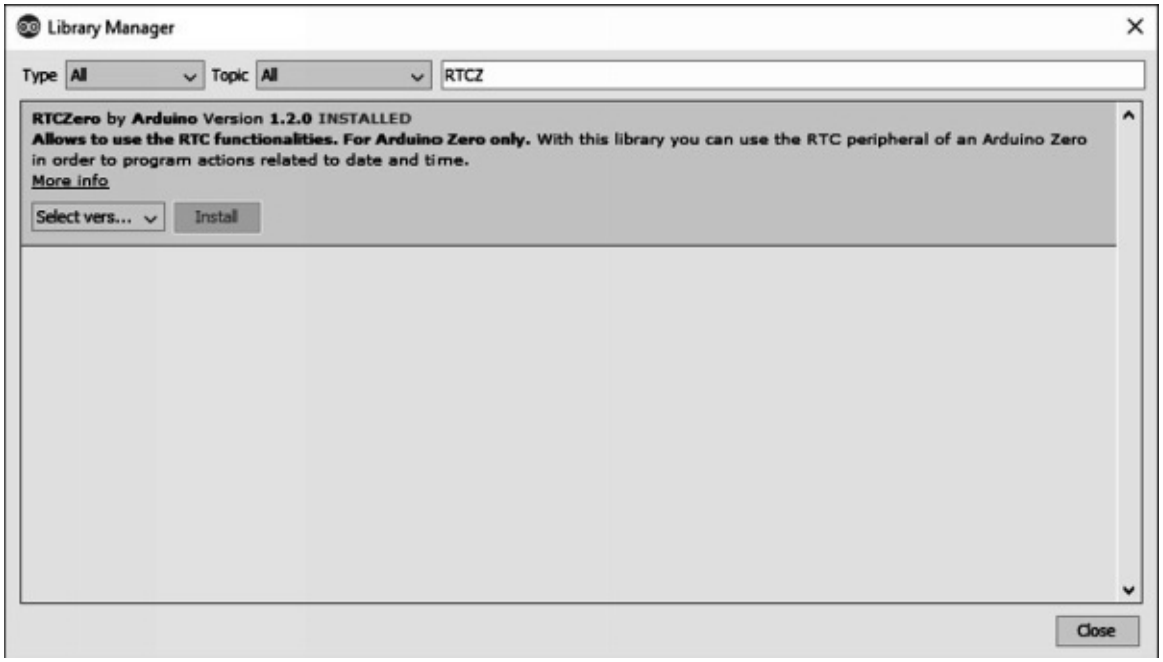


Fig.56

Note that this library is constantly updated so I recommend you to download the newest versions from http://github.com/arduino-libraries/RTCZero.

The following projects illustrate using the **RTCZero** library.

Project 1

This project allows to build a simple application will sets the date (HH:MM:SS), then activates an alarm and sends a message to the **Serial Monitor** each time when the alarm is activated (every 5 seconds, in our case). The source code of this application is shown in **Listing 25**.

Listing 25.

```
#include <RTCZero.h>

/* Create an rtc object */
RTCZero rtc;

/* Change these values to set the current initial time */
const uint8_t seconds = 0;
const uint8_t minutes = 58;
const uint8_t hours = 20;

/* Change these values to set the current initial date */
const uint8_t day = 25;
const uint8_t month = 12;
const uint8_t year = 15;

uint8_t tmpS, tmpMin, tmpH;

int done = 0;

void setup()
{
  Serial.begin(9600);
```

```
  rtc.begin(); // initialize RTC 24H format

  rtc.setTime(hours, minutes, seconds);
  rtc.setDate(day, month, year);

  tmpH = 20;
  tmpMin = 58;
  tmpS = 30;
  rtc.setAlarmTime(tmpH, tmpMin, tmpS);

  rtc.enableAlarm(rtc.MATCH_HHMMSS);

  rtc.attachInterrupt(alarmMatch);
}

void loop()
{
if (done == 1)
  {
    tmpS += 5;
    if (tmpS == 60)
    {
    tmpS = 0;
    tmpMin++;
    }
    if (tmpMin == 60)
    {
    tmpMin = 0;
    tmpH++;
    }
    if (tmpH == 24)
```

```
    tmpH = 0;
    rtc.setAlarmTime(tmpH, tmpMin, tmpS);
    done = 0;
}
}


void alarmMatch()
{
  Serial.print("Alarm is triggered on ");
  Serial.print(tmpH);
  Serial.print(" hour, ");
  Serial.print(tmpMin);
  Serial.print(" min, ");
  Serial.print(tmpS);
  Serial.println(" s.");
  done = 1;
}
```

The running application produces the following output in the **Serial Monitor** window (**Fig.57**).

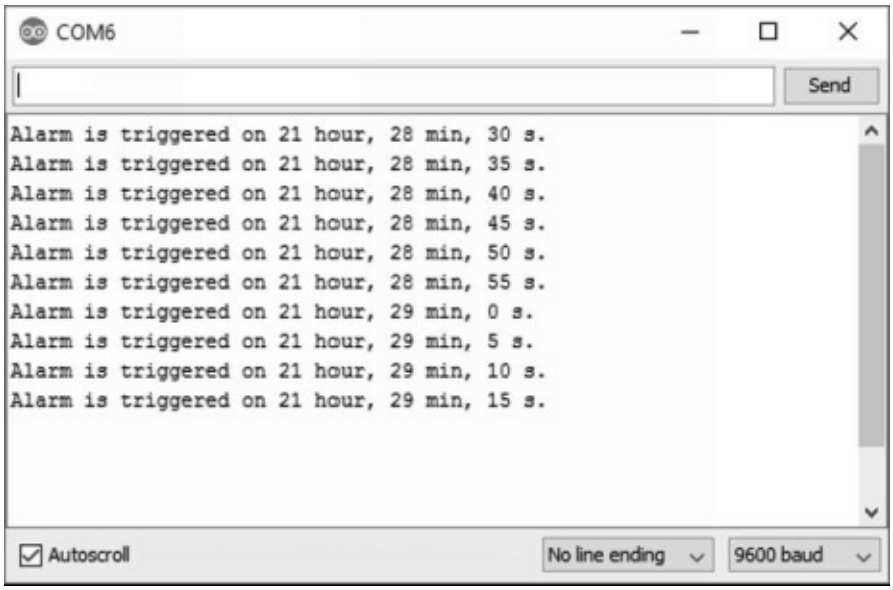


Fig.57

Instead of the **Serial Monito**r we can use an OLED display for output. The source code for this case is shown in **Listing 26**.

Listing 26.

```
#include <RTCZero.h>
#define _Digole_Serial_UART_  // using UART for data transfer

#include <DigoleSerial.h>

//——UART setup————
// Pin 1(TX)on Arduino is connected to RX line (“DATA”) on OLED

DigoleSerialDisp mydisp(&Serial1, 9600);

/* Create an rtc object */
RTCZero rtc;

/* Change these values to set the current initial time */
const uint8_t seconds = 0;
const uint8_t minutes = 58;
const uint8_t hours = 20;

/* Change these values to set the current initial date */
const uint8_t day = 25;
const uint8_t month = 12;
const uint8_t year = 15;

uint8_t tmpS, tmpMin, tmpH;
```

```
int done = 0;

const unsigned char fonts[] = {0, 6, 10};

void setup()
{
Serial1.begin(9600);
mydisp.begin();
mydisp.disableCursor(); //disable cursor, enable cursore use: enableCursor();
mydisp.clearScreen(); //Clear screen
mydisp.setFont(fonts[2]); // set font 6x10
mydisp.setPrintPos(0, 0, _TEXT_);
mydisp.print("Configuring RTC…");

rtc.begin(); // initialize RTC 24H format

rtc.setTime(hours, minutes, seconds);
rtc.setDate(day, month, year);

tmpH = 20;
tmpMin = 58;
tmpS = 30;
rtc.setAlarmTime(tmpH, tmpMin, tmpS);
rtc.enableAlarm(rtc.MATCH_HHMMSS);
rtc.attachInterrupt(alarmMatch);
mydisp.setPrintPos(0, 1, _TEXT_);
mydisp.print("Waiting for alarm…");
}

void loop()
{
```

```
if (done == 1)
  {
    tmpS += 5;
    if (tmpS == 60)
    {
    tmpS = 0;
    tmpMin++;
    }
    if (tmpMin == 60)
    {
    tmpMin = 0;
    tmpH++;
    }
    if (tmpH == 24)
    tmpH = 0;
    rtc.setAlarmTime(tmpH, tmpMin, tmpS);
    done = 0;
}
}

void alarmMatch()
{
  mydisp.clearScreen(); //Clear screen
  mydisp.setFont(fonts[2]); // set font 6x10

  mydisp.setPrintPos(0, 0, _TEXT_);
  mydisp.print("Alarm is triggered on");

  mydisp.setPrintPos(0, 1, _TEXT_);
  mydisp.print(tmpH);
  mydisp.print(" hour");
```

```
  mydisp.setPrintPos(0, 2, _TEXT_);
  mydisp.print(tmpMin);
  mydisp.print(" min");

  mydisp.setPrintPos(0, 3, _TEXT_);
  mydisp.print(tmpS);
  mydisp.print(" s");
  done = 1;
}
```

Project 2

This project illustrates using RTC in an application that periodically (every 10 s) measures the level of an analog signal on input **A0**. The circuit diagram for the project is shown in **Fig.58**.

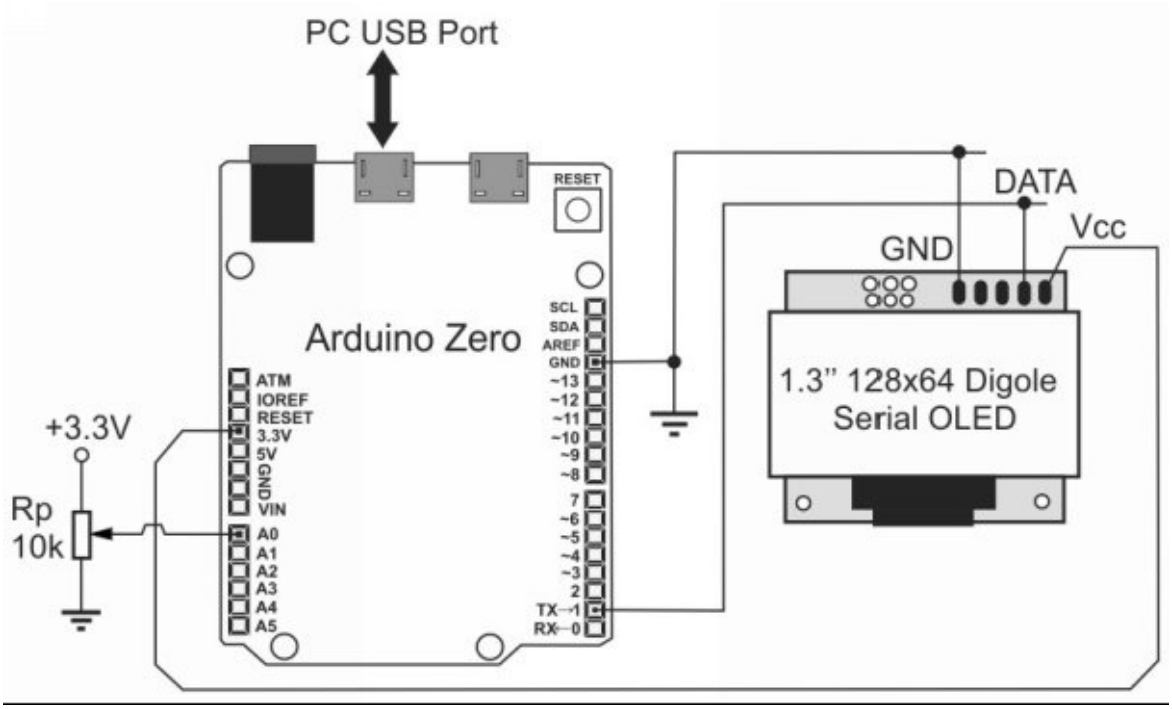


Fig.58

In this circuit, the analog signal on pin **A0** is simulated by the voltage divider with potentiometer **Rp**. The data are transferred via a serial interface from Arduino Zero (pin "**TX→1**") to the OLED display (pin "**DATA**").

The source code of the application is shown in **Listing 27**.

Listing 27.

```
#include <RTCZero.h>
#define _Digole_Serial_UART_  // using UART for data transfer

#include <DigoleSerial.h>
```

```
//——UART setup————
// Pin 1 (TX) on Arduino is connected to RX line (“DATA”) on OLED

DigoleSerialDisp mydisp(&Serial1, 9600);

/* Create an rtc object */
RTCZero rtc;

/* Change these values to set the current initial time */
const uint8_t seconds = 0;
const uint8_t minutes = 58;
const uint8_t hours = 20;

/* Change these values to set the current initial date */
const uint8_t day = 25;
const uint8_t month = 12;
const uint8_t year = 15;

uint8_t tmpS, tmpMin, tmpH;
int done = 0;

const unsigned char fonts[] = {0, 6, 10};

int binADC;
const double LSB = 3.33 / 4096; // 12-bit resolution
double Vin; // an input voltage on pin A0

void setup()
{
analogReadResolution(12);
```

```
Serial1.begin(9600);
mydisp.begin();
mydisp.disableCursor(); //disable cursor, enable cursore use: enableCursor();
mydisp.clearScreen(); //Clear screen
mydisp.setFont(fonts[2]); // set font 6x10
mydisp.setPrintPos(0, 0, _TEXT_);
mydisp.print("Configuring RTC…");

rtc.begin(); // initialize RTC 24H format

rtc.setTime(hours, minutes, seconds);
rtc.setDate(day, month, year);

tmpH = 20;
tmpMin = 58;
tmpS = 30;
rtc.setAlarmTime(tmpH, tmpMin, tmpS);
rtc.enableAlarm(rtc.MATCH_HHMMSS);
rtc.attachInterrupt(alarmMatch);
mydisp.setPrintPos(0, 1, _TEXT_);
mydisp.print("Waiting for alarm…");
}

void loop()
{
if (done == 1)
  {
    tmpS += 10;
    if (tmpS == 60)
    {
    tmpS = 0;
```

```
    tmpMin++;
    }
    if (tmpMin == 60)
    {
    tmpMin = 0;
    tmpH++;
    }
    if (tmpH == 24)
    tmpH = 0;
    rtc.setAlarmTime(tmpH, tmpMin, tmpS);
    done = 0;
}
}

void alarmMatch()
{
  binADC = analogRead(0);
  Vin = LSB * binADC;

  mydisp.clearScreen(); //Clear screen
  mydisp.setFont(fonts[2]); // set font 6x10

  mydisp.setPrintPos(0, 0, _TEXT_);
  mydisp.print("Time: ");
  mydisp.print(tmpH);
  mydisp.print("h ");
  mydisp.print(tmpMin);
  mydisp.print("min ");
  mydisp.print(tmpS);
  mydisp.print("s");
```

```
  mydisp.setPrintPos(0, 1, _TEXT_);
  mydisp.print(“A0 input: “);
  mydisp.print(Vin, 3);
  mydisp.print(“V”);
  done = 1;
}
```

Project 3

This project demonstrates one more programming technique for processing alarm events. Often we are interested in performing some particular task in the predetermined intervals without knowing the exact time (HH:MM:SS) when the task starts running.

In this case, we can simplify our source code as is shown in **Listing 28**. The application represented by this code simply toggles the on-board LED (pin **13**) every 1 s.

Listing 28.

```
#include <RTCZero.h>

/* Create an rtc object */
RTCZero rtc;

const int LED = 13;
/* Change these values to set the current initial time */
const byte seconds = 0;
const byte minutes = 26;
const byte hours = 18;

/* Change these values to set the current initial date */
const byte day = 29;
const byte month = 12;
const byte year = 15;

volatile byte oldSeconds, curSeconds;
volatile int bRead;

int done = 0;
int cnt = 0;
void setup()
```

```
{
  REG_PORT_DIRSET0 = PORT_PA17; // pin D13 is set as output
  REG_PORT_OUTSET0 = PORT_PA17; // LED is ON
  delay(500);
  REG_PORT_OUTCLR0 = PORT_PA17; // LED is OFF
  Serial.begin(9600);

  rtc.begin();

  rtc.setTime(hours, minutes, seconds);
  rtc.setDate(day, month, year);

  curSeconds = 7;
  rtc.setAlarmSeconds(curSeconds);
  rtc.enableAlarm(rtc.MATCH_SS);

  rtc.attachInterrupt(alarmMatch);
}

void loop()
{
if (done == 1)
{
  oldSeconds = curSeconds;
  curSeconds = (curSeconds + 1) % 60;
  rtc.setAlarmSeconds(curSeconds);
  Serial.print(++cnt);
  Serial.print(“: Alarm is triggered after “);
  Serial.print(oldSeconds);
  Serial.println(” s”);
  done = 0;
```

```
}
}

void alarmMatch()
{
REG_PORT_OUTTGL0 = PORT_PA17;
done = 1;
}
```

In this source code we use a few low-level I/O instructions that work faster than **pinMode()** and **digitalWrite()** statements.

The instruction

```
REG_PORT_DIRSET0 = PORT_PA17;
```

allows to configure pin **13** (defined as **PORT_PA17** in ATSAMd21G18 MCU header files) as output.

The instruction

```
REG_PORT_OUTSET0 = PORT_PA17;
```

pulls pin **13** HIGH thus driving the LED ON.

One more instruction

```
REG_PORT_OUTCLR0 = PORT_PA17;
```

pulls pin **13** LOW thus driving the LED OFF.

The statement

```
REG_PORT_OUTTGL0 = PORT_PA17;
```

toggles pin **13**.

The **Serial Monitor** window for this application looks like the following (**Fig.59**).

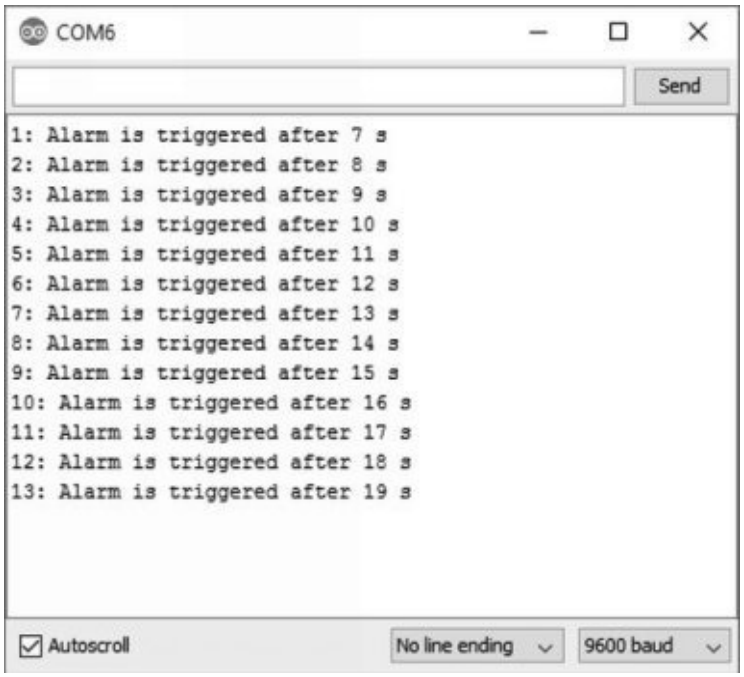


Fig.59

Arduino Zero low-powered applications

Modern applications should often be powered from batteries, therefore a developer must think about extending battery life when designing such applications. Microprocessor ATSAMD21G18 used in Arduino Zero belongs to the Cortex-M0 family capable of operating in sleep mode.

The Cortex-M0 processor sleep modes reduce power consumption. The sleep modes your device implements are implementation-defined, but they might be one or both of the following:

- a sleep mode that stops the processor clock;
- a deep sleep mode that stops the system clock and switches off the PLL and flash

memory.

If your device implements two sleep modes providing different levels of power saving, the **SLEEPDEEP** bit of the **System Control Register** (**SCR**) selects which sleep mode is used.

This section describes the mechanisms for entering sleep mode, and the conditions for waking up from sleep mode.

To enter to sleep mode we can use one of the CPU instructions, **WFI** or **WFE**.

The Wait For Interrupt instruction, **WFI**, causes immediate entry to sleep mode. When the processor executes a **WFI** instruction it stops executing instructions and enters sleep mode.

The Wait For Event instruction, **WFE**, causes entry to sleep mode conditional on the value of a one-bit event register. When the processor executes a **WFE** instruction, it checks the value of the event register:

- 0 – The processor stops executing instructions and enters sleep mode
- 1 – The processor sets the register to zero and continues executing instructions without entering sleep mode.

If the **SLEEPONEXIT** bit of **SCR** is set to 1, when the processor completes the execution of an exception handler and returns to Thread mode it immediately enters sleep mode. We can use this mechanism in applications that only require the processor to run when an interrupt occurs.

ISO/IEC C cannot directly generate the **WFI** and **WFE** instructions. The CMSIS provides the following intrinsic functions for these instructions:

void __WFE(void) // Wait for Event

void __WFI(void) // Wait for Interrupt

Our demo applications in this section will use the **__WFI()** function. This function when called refers to the parameters written in the **System Control Register** (**SCR**) belonging to **System Control Block** (**SCB**). The **SCB** provides system implementation information, and system control. This includes configuration, control, and reporting of the system exceptions.

The **SCR** controls features of entry to and exit from low power state. The bit assignments for this register are shown in **Fig.60**.

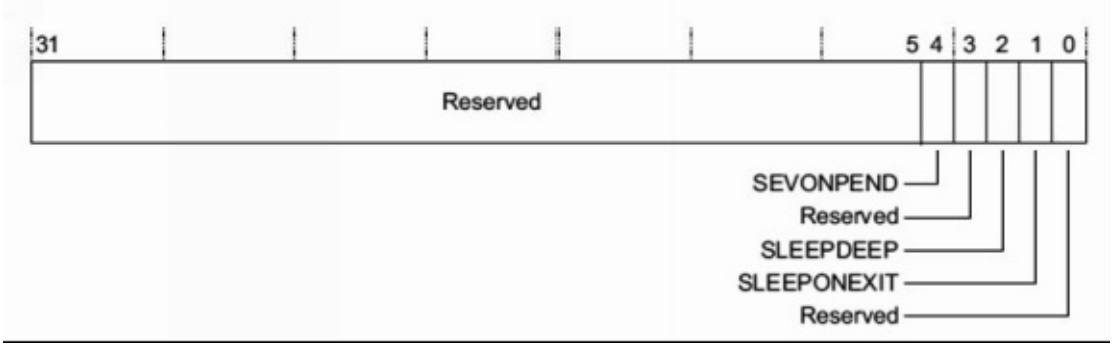


Fig.60

We are especially interested in SLEEPDEEP and SLEEPONEXIT bits. The SLEEPDEEP bit controls whether the processor uses sleep or deep sleep as its low power mode:

- 0 = sleep
- 1 = deep sleep.

If your device does not support two sleep modes, the effect of changing the value of this bit is implementation-defined.

The **SLEEPONEXIT** bit Indicates sleep-on-exit when returning from Handler mode to Thread mode:

- 0 = do not sleep when returning to Thread mode.
- 1 = enter sleep, or deep sleep, on return from an ISR to Thread mode.

Setting this bit to 1 enables an interrupt driven application to avoid returning to an empty main application.

The following projects shows how to build low-power applications using sleep mode.

Project 1

This project allows to build the application which spends most of the time in sleep mode. The application will run a useful task only after an external interrupt is triggered on pin **9** of Arduino Zero. To periodically implement the task the interrupt should be triggered by some external clock source, for example, oscillator circuit with TTL-compatible output.

The source code of the application is shown in **Listing 29**.

Listing 29.

```
const int LED = 13;

int pinState;
int done = 0;
int cnt = 0;

void setup()
{
  pinMode(9, INPUT_PULLUP);
  REG_PORT_DIRSET0 = PORT_PA17; // pin D13 is set as output
  REG_PORT_OUTSET0 = PORT_PA17; // LED is ON
  delay(1000);
  REG_PORT_OUTCLR0 = PORT_PA17; // LED is OFF
  Serial.begin(9600);
  attachInterrupt(9, pinISR, RISING);
  delay(5);
  nosleep_ONEXIT();
}

void loop()
{
```

```
if (done == 1)
{
    done = 0;
    Serial.print(++cnt);
    Serial.println(“: Main loop is running…”);
    sleep_ONEXIT();
  }
}

void pinISR()
{
  REG_PORT_OUTTGL0 = PORT_PA17;
  Serial.print(++cnt);
  Serial.println(“: LED is toggled.”);
  done = 1;
}

void nosleep_ONEXIT()
{
  SCB->SCR |= 0x0; // sleep mode WITHOUT entering to sleep mode ON EXIT when returning from an interrupt
  __WFI();
}

void sleep_ONEXIT()
{
    SCB->SCR |= 0x2; // sleep mode WITH entering to sleep mode ON EXIT when returning from an interrupt
  __WFI();
}
```

There are two functions, **nosleep_ONEXIT()** and **sleep_ONEXIT()**, that determine the

behavior of our application. The **nosleep_ONEXIT()** function within the **setup()** function puts CPU in sleep mode with very low power consumption. When CPU stays in this mode, the **loop()** function will not be executed. CPU exits sleep mode when an interrupt is triggered on pin **9**. After the Interrupt Service Routine **pinISR()** has been executed, CPU returns to Thread mode and the application enters the **loop()** procedure.

The statements within the **loop()** function will be executed only once, before the **sleep_ONEXIT()** function is reached. This function puts CPU in sleep mode all the time, except the interval when the ISR **pinISR()** is been executed. Therefore, the instructions within the **loop()** function will never be executed one more.

The output of the running application is shown in **Fig.61**.

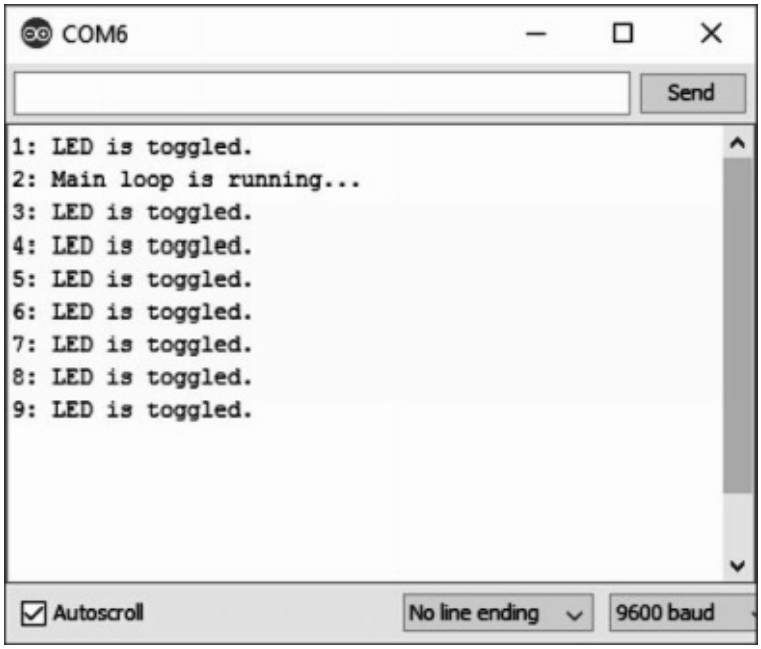


Fig.61

Project 2

This is one more project that illustrates operating Arduino Zero in sleep mode. The application puts CPU in sleep mode by invoking function **sleep_ONEXIT()** within the **setup()** procedure. No instructions will be executed until an external interrupt is triggered on pin **9**. CPU wakes up only when an interrupt is triggered. When ISR returns the control to the main thread, CPU immediately enters sleep mode again. As a consequence, the program code within the **loop()** function will never be executed.

The source code of the application is shown in **Listing 30**.

Listing 30.

```
const int LED = 13;


int pinState;
int done = 0;
int cnt = 0;


void setup()
{
  pinMode(9, INPUT_PULLUP);
  REG_PORT_DIRSET0 = PORT_PA17; // pin D13 is set as output
  REG_PORT_OUTSET0 = PORT_PA17; // LED is ON
  delay(1000);
  REG_PORT_OUTCLR0 = PORT_PA17; // LED is OFF
  Serial.begin(9600);
  attachInterrupt(9, pinISR, RISING);
  delay(5);


    // Entering standby mode when connected
  // via the native USB port causes issues.
  sleep_ONEXIT();
}
```

```
void loop()
{
if (done == 1)
{
    done = 0;
    Serial.print(++cnt);
    Serial.println(“: Main loop is running…”);
    sleep_ONEXIT();
  }
}

void pinISR()
{
  REG_PORT_OUTTGL0 = PORT_PA17;
  Serial.print(++cnt);
  Serial.println(“: LED is toggled.”);
  done = 1;
}

void nosleep_ONEXIT()
{
  SCB->SCR |= 0x0; // sleep mode WITHOUT entering to sleep mode ON EXIT
    // when returning from an interrupt
  __WFI();
}

void sleep_ONEXIT()
{
    SCB->SCR |= 0x2; // sleep mode WITH entering to sleep mode ON EXIT
    // when returning from an interrupt
```

```
  __WFI();
}
```

The application produces the following output in the **Serial Monitor** window (**Fig.62)**.

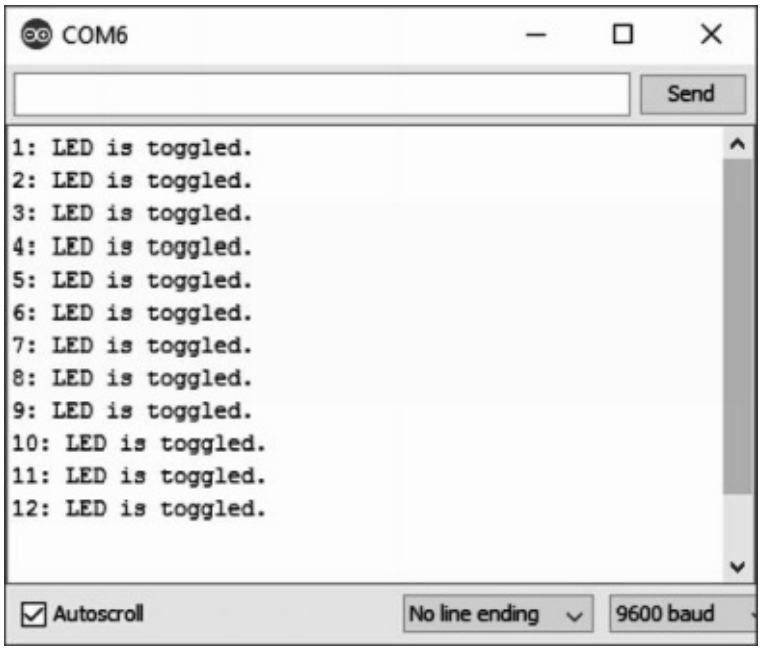


Fig.62

It is seen that only the program code within ISR is executed.

Note that applications represented in both projects will stay in sleep mode infinitely until an interrupt on pin **9** is triggered.

Project 3

This project illustrates the case when the program code within the main loop (function **loop()**) is executed each time after an interrupt has been processed. In this application, the sleep mode is initiated by the function **nosleep_ONEXIT()**. The source code of the application is shown in **Listing 31**.

Listing 31.

```
const int LED = 13;

int done = 0;
int cnt = 0;

void setup()
{
  pinMode(9, INPUT_PULLUP);
  REG_PORT_DIRSET0 = PORT_PA17; // pin D13 is set as output
  REG_PORT_OUTSET0 = PORT_PA17; // LED is ON
  delay(1000);
  REG_PORT_OUTCLR0 = PORT_PA17; // LED is OFF
  Serial.begin(9600);
  attachInterrupt(9, pinISR, RISING);
  delay(5);

    // Entering standby mode when connected
  // via the native USB port causes issues.
  nosleep_ONEXIT();
}

void loop()
{
```

```
if (done == 1)
{
    done = 0;
    Serial.print(++cnt);
    Serial.println(“: Main loop is running…”);
    nosleep_ONEXIT();
  }
}

void pinISR()
{
  REG_PORT_OUTTGL0 = PORT_PA17;
  Serial.print(++cnt);
  Serial.println(“: LED is toggled.”);
  done = 1;
}

void nosleep_ONEXIT()
{
  SCB->SCR |= 0x0; // sleep mode WITHOUT entering to sleep mode ON EXIT
    // when returning from an interrupt
  __WFI();
}

void sleep_ONEXIT()
{
    SCB->SCR |= 0x2; // sleep mode WITH entering to sleep mode ON EXIT
    // when returning from an interrupt
  __WFI();
}
```

In this source code, the first **nosleep_ONEXIT()** function is called within the **setup()** procedure. Therefore, CPU will be brought to Thread mode only after the interrupt triggered on pin 9 has been processed. The second **nosleep_ONEXIT()** function within the **loop()** procedure allows the rest of the code to be executed after Interrupt Service Routine terminates.

The running application produces the following output (**Fig.63)** in the **Serial Monitor** window.

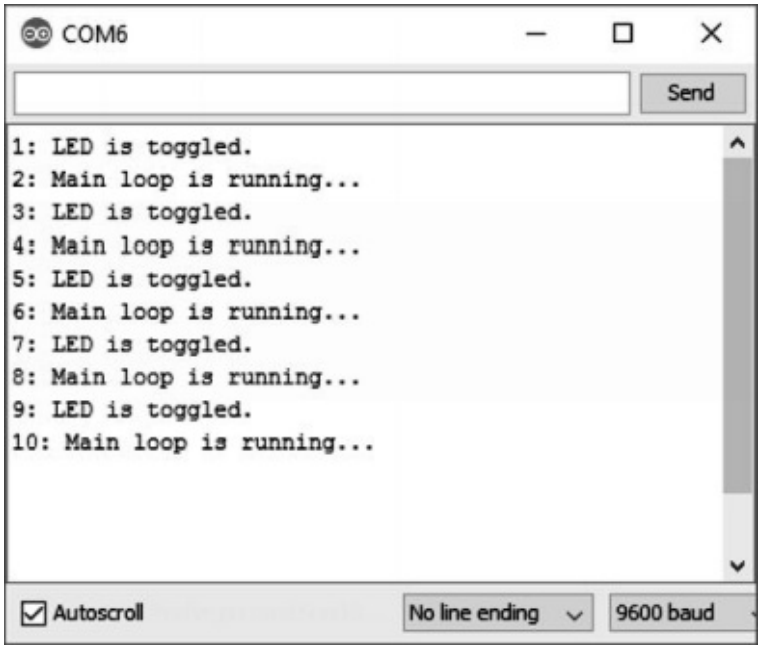


Fig.63

Project 4

This project illustrates the design of a simple temperature control system which generates a signal when the temperature of environment exceeds a given threshold. The application when started puts CPU in sleep mode.

The circuit diagram of such system is shown in **Fig.64**.

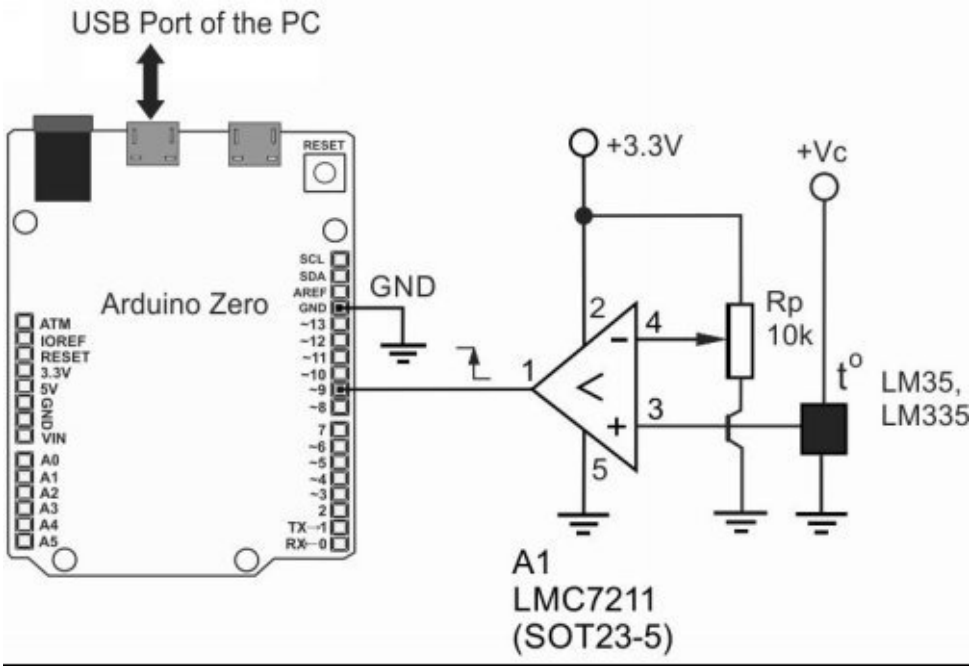


Fig.64

Here the circuit built around comparator LMC7211 with the TTL-compatible output is used for triggering an interrupt on pin **9** of Arduino Zero when the temperature of environment exceeds a threshold. The threshold is determined by the voltage level applied to the inverting input of LMC7211 from the wiper of potentiometer R_P, while the output voltage of the temperature sensor is applied to the non-inverting input of the comparator.

The source code of the application is shown in **Listing 32**.

Listing 32.

```
const int LED = 13;

volatile int pinState;
```

```
volatile int done = 0;
volatile int cnt = 0;

void setup()
{
  pinMode(9, INPUT_PULLUP);
  pinMode(LED, OUTPUT);
  Serial.begin(9600);
  attachInterrupt(9, pinISR, CHANGE);
  delay(5);

    // Entering standby mode when connected
  // via the native USB port causes issues.

  Serial.println("Temperature control is active…");
  pinState = digitalRead(9);
  if (pinState != 0x0)   // non-inverting input of the comp. is greater than an inverting one
  {
    digitalWrite(LED, HIGH);
    Serial.println("Too high temperature in the room.");
  }
  delay(5);
  sleep_ONEXIT();
}

void loop(){}

void pinISR()
{
  pinState = digitalRead(9);
  if (pinState != 0x0)   // non-inverting input of the comp. is greater than an inverting one
```

```
  {
    digitalWrite(LED, HIGH);
    Serial.print(++cnt);
    Serial.println(“: Too high temperature in the room.”);
  }
  else
    digitalWrite(LED, LOW);
}

void sleep_ONEXIT()
{
    SCB->SCR |= 0x2; // reentering sleep mode ON EXIT from an interrupt
  __WFI();
}
```

In this source code, an application starts operating in Thread mode until **sleep_ONEXIT()** function is reached. This function puts CPU in sleep mode so the signal from the temperature sensor will be processed only by an Interrupt Service Routine **pinISR()**. When ISR returns control to the main thread, CPU is put in sleep mode again.

As you can see, almost all the time CPU stays in sleep mode with very low power consumption.

The application produces the following output in the **Serial Monitor** window (**Fig.65)**.

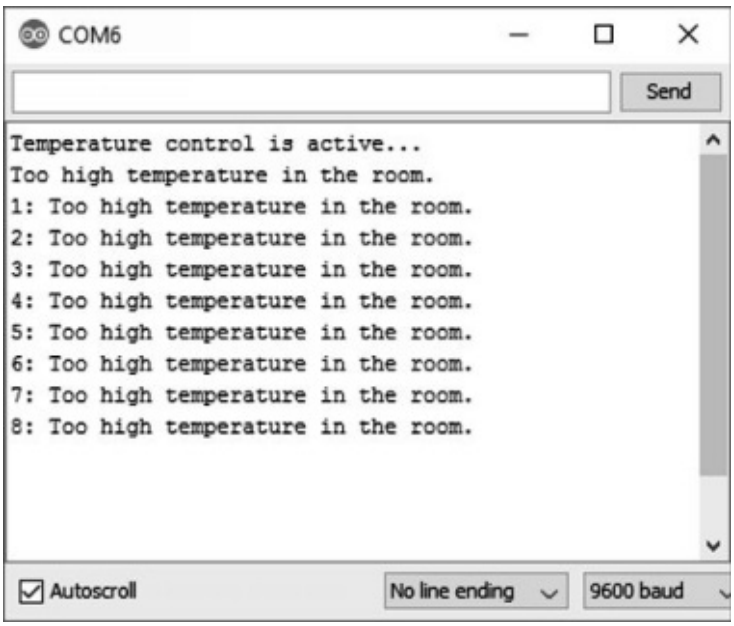


Fig.65

Real-Time Counter in low-powered applications

The built-in Real-Time Counter (RTC) block of the ATSAMD21G18 microcontroller allows to design very low-power Arduino Zero applications where CPU is held in sleep (power-save) mode waking periodically up for performing particular tasks. To use RTC in low-power applications we can apply an excellent **RTCZero** library accessible at https://github.com/arduino-libraries/RTCZero. The library includes functions that allow to develop applications operating in low-power mode. This section describes a few projects allowing to build low-power applications using RTC.

Project 1

This demo project illustrates how to build a simple low-power application driven by Real-Time Counter. In such application, most of the time Arduino Zero stays in sleep mode consuming very low power. Each 10 seconds RTC triggers the alarm that wakes up the CPU and peripherals. When the alarm is activated, the program code of the Interrupt Service Routine toggles the on-board LED (pin **13)**. The program code within a **loop()** procedure also sends a message to the **Serial Monitor**.

The source code for this project is shown in **Listing 33**.

Listing 33.

```
#include <RTCZero.h>

/* Create an rtc object */
RTCZero rtc;

const int LED = 13;
/* Change these values to set the current initial time */
const byte seconds = 0;
const byte minutes = 26;
const byte hours = 18;

/* Change these values to set the current initial date */
const byte day = 29;
const byte month = 12;
const byte year = 15;

volatile byte oldSeconds, curSeconds;
volatile int bRead;

int done = 0;
int cnt = 0;
```

```
void setup()
{
  REG_PORT_DIRSET0 = PORT_PA17; // pin D13 is set as output
  REG_PORT_OUTSET0 = PORT_PA17; // LED is ON
  delay(1000);
  REG_PORT_OUTCLR0 = PORT_PA17; // LED is OFF
  Serial.begin(9600);

  rtc.begin();

  rtc.setTime(hours, minutes, seconds);
  rtc.setDate(day, month, year);

  curSeconds = 8;
  rtc.setAlarmSeconds(curSeconds);
  rtc.enableAlarm(rtc.MATCH_SS);
  rtc.attachInterrupt(alarmMatch);
  rtc.standbyMode();
}

void loop()
{
if (done == 1)
{
  oldSeconds = curSeconds;
  curSeconds = (curSeconds + 10) % 60;
  rtc.setAlarmSeconds(curSeconds);
  Serial.print(++cnt);
  Serial.print(“: CPU wakes up after “);
  Serial.print(oldSeconds);
  Serial.println(” s”);
```

```
  done = 0;
  rtc.standbyMode(); // Sleep until next alarm match
}
}

void alarmMatch()
{
REG_PORT_OUTTGL0 = PORT_PA17;
done = 1;
}
```

In this source code, the sequence

```
  rtc.begin();
  rtc.setTime(hours, minutes, seconds);
  rtc.setDate(day, month, year);
```

initializes RTC and configures the current time and date. To periodically toggle the LED on pin **13** we will trigger the RTC interrupt every 10 s. Configuring the alarm and the corresponding interrupt is performed by the following sequence:

```
rtc.setAlarmSeconds(curSeconds);
rtc.enableAlarm(rtc.MATCH_SS);
rtc.attachInterrupt(alarmMatch);
```

The Interrupt Service Routine **alarmMatch()** will simply toggle pin **13** every 10 s. Additionally, the variable **done** will be assigned 1 – this enables the program code within the **loop()** procedure to be executed. Note that the RTC interrupt wakes up CPU so that the program code in the **loop()** procedure can be executed.

The last statement in the **loop()** procedure

```
rtc.standbyMode();
```

puts CPU in sleep mode again. CPU exits this mode only when the next alarm occurs.

Project 2

This project illustrates use of various programming techniques. The program code of the application configures RTC and puts CPU and peripherals in sleep mode. Once alarm is triggered, the corresponding ISR **alarmMatch()** is invoked and two tasks begin running. The source code of the application is shown in **Listing 34**.

Listing 34.

```
/*
  Demonstrates the use of an alarm to wake up CPU from Standby mode in
  the multitasking application
*/

#include <Scheduler.h>
#include <RTCZero.h>

/* Create an rtc object */
RTCZero rtc;

const double LSB = 3.3/4096; // LSB value for 12-bit resolution
int binADC;  // binary code for analog input
double Vin; // analog input value

const int LED = 13;
/* Change these values to set the current initial time */
const byte seconds = 0;
const byte minutes = 26;
const byte hours = 18;

/* Change these values to set the current initial date */
const byte day = 29;
```

```
const byte month = 12;
const byte year = 15;

volatile byte oldSeconds, curSeconds;
volatile int bRead;

int done = 0;
int cnt = 0;
void setup()
{
  analogReadResolution(12);
  REG_PORT_DIRSET0 = PORT_PA17; // pin D13 is set as output
  REG_PORT_OUTSET0 = PORT_PA17; // LED is ON
  delay(1000);
  REG_PORT_OUTCLR0 = PORT_PA17; // LED is OFF
  Serial.begin(9600);

  rtc.begin();

  rtc.setTime(hours, minutes, seconds);
  rtc.setDate(day, month, year);

  curSeconds = 8;
  rtc.setAlarmSeconds(curSeconds);
  rtc.enableAlarm(rtc.MATCH_SS);
  rtc.attachInterrupt(alarmMatch);
  delay(100);
  Scheduler.startLoop(loop2);
  rtc.standbyMode();
}
```

```
void loop()
{
if (done == 1)
{
  oldSeconds = curSeconds;
  curSeconds = (curSeconds + 10) % 60;
  rtc.setAlarmSeconds(curSeconds);
  Serial.print(++cnt);
  Serial.print(“: CPU wakes up after “);
  Serial.print(oldSeconds);
  Serial.println(” s”);
  done = 0;
  delay(250);
  rtc.standbyMode(); // Sleep until next alarm match
}
}

void loop2()
{
  if (done == 0)
  {
    binADC = analogRead(0);
    Vin = (double)binADC * LSB;
    Serial.print(“A0 input: “);
    Serial.print(Vin,3);
    Serial.println(” V”);
    delay(50);
  }
  // IMPORTANT:
  // We must call ‘yield’ at a regular basis to pass
  // control to other tasks.
```

```
  yield();
}


void alarmMatch()
{
REG_PORT_OUTTGL0 = PORT_PA17;
done = 1;
}
```

One main thread associated with the **loop()** procedure sends a message to the **Serial Monitor**, reconfigures the RTC alarm and puts CPU in sleep mode.

The **loop2()** procedure implents the additional task which performs the measurement of the analog signal on pin **A0** of Arduino Zero and transfers the result obtained to the **Serial Monitor**.

The **alarmMatch()** ISR simply toggles pin **13** (LED) and allows the main thread to start running.

The **done** variable is used for synchronizing threads.

The application produces the following output in the **Serial Monitor** window (**Fig.66**).

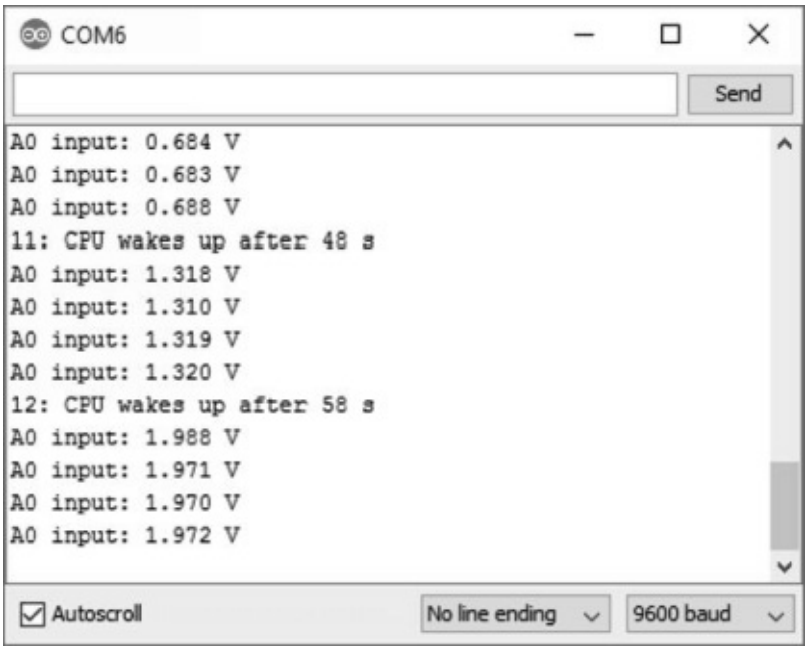


Fig.66

Project 3

This project allows to build the application which measures the analog input voltage on channel **A0** during the interval determined by the RTC alarm. The data are transferred via the USB-to-UART adapter to the PC. The circuit diagram of the project is shown **Fig.67**.

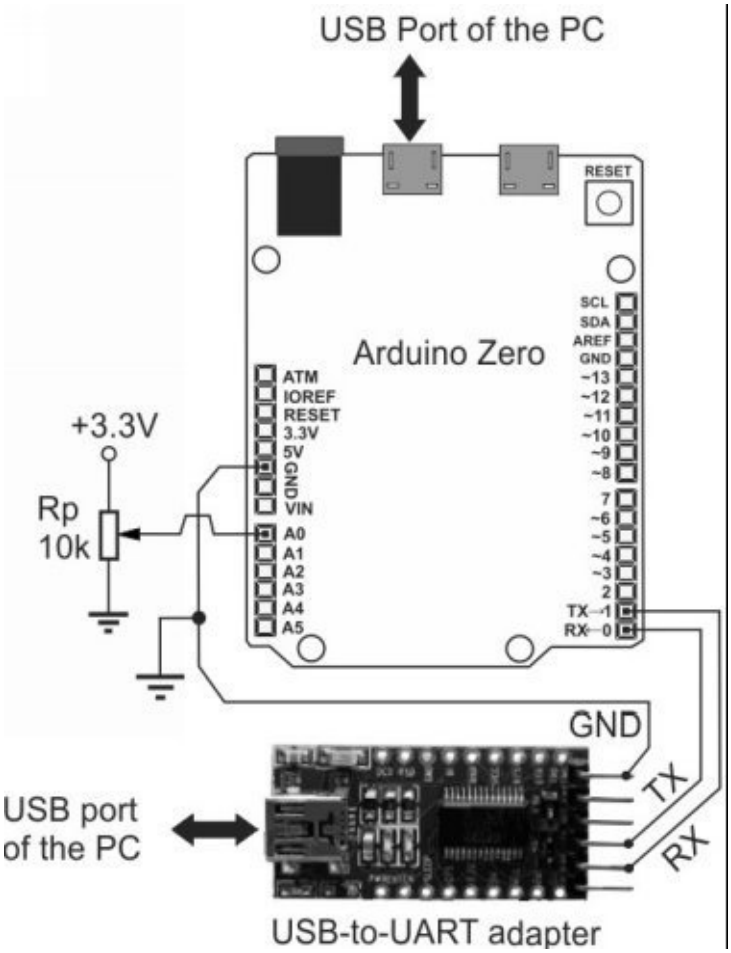


Fig.67

In this circuit, the analog signal is fed to pin **A0** of Arduino Zero from the voltage divider using potentiometer **Rp**. The data are then transferred to the PC via an USB-to-UART adapter.

The source code of the application is almost the same as that from the previous project. The only difference is that in this application we use the **Serial1** interface (associated with data lines "**TX → 1**" and "**RX ← 0**") instead of **Serial**. The source code of the application is shown in **Listing 35**.

Listing 35.

```
#include <Scheduler.h>
#include <RTCZero.h>

/* Create an rtc object */
RTCZero rtc;

const double LSB = 2.0/4096; // LSB value for 12-bit resolution when battery of 2.7V minus 1N4004 diode is used
int binADC;  // binary code for analog input
double Vin; // analog input value

const int LED = 13;
/* Change these values to set the current initial time */
const byte seconds = 0;
const byte minutes = 26;
const byte hours = 18;

/* Change these values to set the current initial date */
const byte day = 29;
const byte month = 12;
const byte year = 15;

volatile byte oldSeconds, curSeconds;
volatile int bRead;

int done = 0;
int cnt = 0;
void setup()
{
  analogReadResolution(12);
  REG_PORT_DIRSET0 = PORT_PA17; // pin D13 is set as output
```

```
  REG_PORT_OUTSET0 = PORT_PA17; // LED is ON
  delay(1000);
  REG_PORT_OUTCLR0 = PORT_PA17; // LED is OFF
  Serial1.begin(9600);

  rtc.begin();

  rtc.setTime(hours, minutes, seconds);
  rtc.setDate(day, month, year);

  curSeconds = 8;
  rtc.setAlarmSeconds(curSeconds);
  rtc.enableAlarm(rtc.MATCH_SS);
  rtc.attachInterrupt(alarmMatch);
  delay(100);
  Scheduler.startLoop(loop2);
  rtc.standbyMode();
}

void loop()
{
if (done == 1)
{
  oldSeconds = curSeconds;
  curSeconds = (curSeconds + 10) % 60;
  rtc.setAlarmSeconds(curSeconds);
  Serial1.print(++cnt);
  Serial1.print(“: CPU awoke after “);
  Serial1.print(oldSeconds);
  Serial1.println(” s”);
  done = 0;
```

```
  delay(250);
  rtc.standbyMode(); // Sleep until next alarm match
}
}

void loop2()
{
  if (done == 0)
  {
    binADC = analogRead(0);
    Vin = (double)binADC * LSB;
    Serial1.print("A0 input: ");
    Serial1.print(Vin,3);
    Serial1.println(" V");
    delay(50);
  }
  // IMPORTANT:
  // We must call 'yield' at a regular basis to pass
  // control to other tasks.
  yield();
}

void alarmMatch()
{
REG_PORT_OUTTGL0 = PORT_PA17; // toggle D13
done = 1;
}
```

Arduino Zero battery-powered applications

This section is dedicated to design of battery-powered application with Arduino Zero. As it was illustrated in the previous section, Arduino Zero can operate with very low power consumption when CPU spends most of the time in sleep mode. We also know how to build low-power applications using Real-Time Counter (RTC) of the ATSAMD21 processor.

With these programming techniques we can easily design Arduino Zero applications powered from nonrechargeable or rechargeable batteries. Let's consider several power schemes that can be applied to Arduino Zero when using batteries.

There can be two cases:

1. Arduino Zero is powered from a stand-alone DC power source 7 – 12 V or via a USB port. A battery can be connected to pin "**3.3V**" to provide power to Arduino when a main power source fails or is OFF for some reason;
2. Arduino Zero is powered by a stand-alone battery all the time.

The power scheme in case 1 may look like the following (**Fig.68)**.

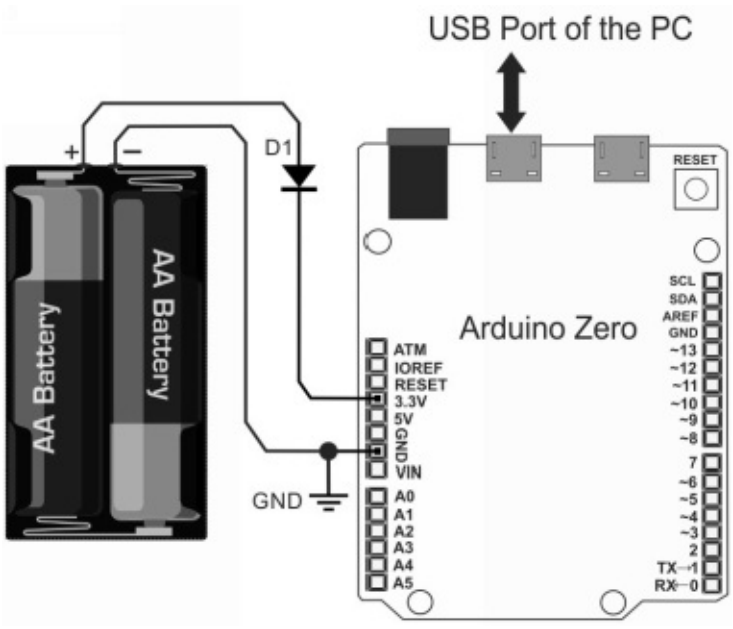


Fig.68

In this circuit, two Ni-Mh rechargeable batteries connected in series are used. Diode D1 prevents current flow to pin "**3.3V**" when Arduino Zero is powered from USB or a stand-

alone DC power supply. When an external power supply fails, the current begins to flow from the batteries to pin "**3.3V**" through the opened diode D1.

Note that the actual voltage **Vin** on pin "**3.3V**" of Arduino Zero will be less than the voltage **Vb** provided by the battery because of the voltage drop **Vd** on the rectifier (diode D1). Therefore, the formula for calculating **Vin** appears as follows:

$$Vin = Vb - Vd$$

The voltage drop **Vd** across diode 1N4004 for the forward current < 1 A is about 680-710 mV (0.68-0.71V). For a Schottky diode such as 1N5817 the voltage drop at 1A will be about 0.55V. Two full-charged batteries usually provide the voltage of 2.6 – 2.75 V, so the **Vin** voltage can vary in the range of about 2.0 – 2.2V.

Note that this power scheme can't provide the high stable DC voltage on pin "**3.3V**" of Arduino Zero because batteries will discharge with time. Nevertheless, the ATSAMD21G18A CPU and most peripherals will operate until the voltage on pin "**3.3V**" drops below 1.8 V. When an application should operate with analog-to-digital or digital-to-analog converter, we have to provide a stand-alone precision reference voltage to these peripherals to ensure proper operations.

The advanced power scheme shown in **Fig.69** allows to keep the high stable DC voltage on pin "**3.3V**" by using a Low DropOut voltage regulator (LDO). As in the previous case, diode D1 prevents the current flow to pin "**3.3V**" when Arduino Zero is powered from USB or an external DC power supply. When an external power source fails or is OFF, the current begins to flow from the LDO output to pin "**3.3V**" through the opened diode D1.

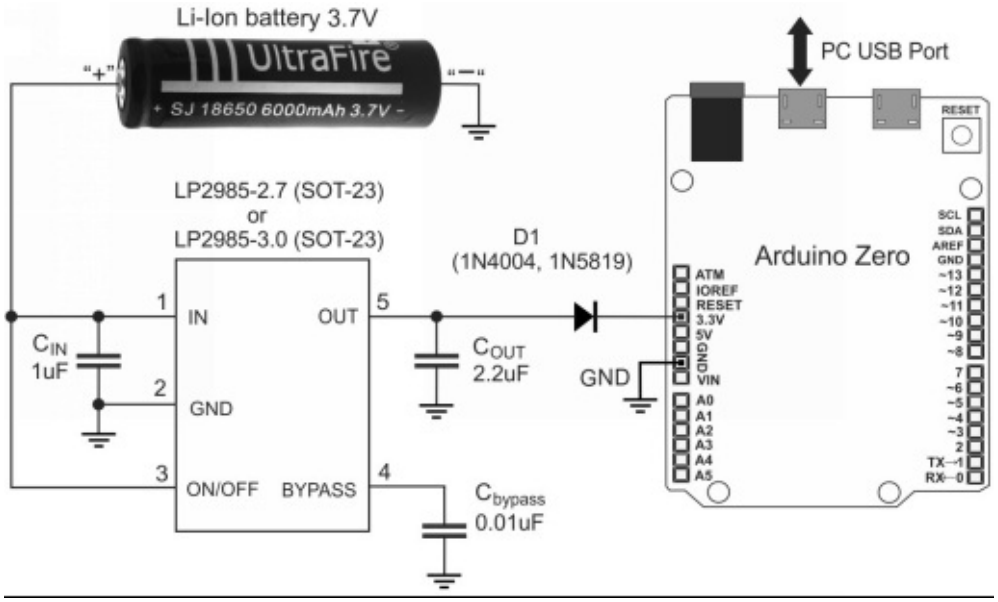


Fig.69

In this circuit, we use the Lithium ion (Li-Ion) rechargeable battery which usually provides 3.7 V at its output. The battery voltage is converted into a high stable voltage +2.7 or +3.0 V by using the LDO device LP2985-2.7 or LP2985-3.0, respectively.

Some words about LD2985-N LDO devices.

These devices belongs to the LP2985-N family of fixed-output, ultra-low-dropout and low-noise regulators offers exceptional, cost-effective performance for battery-powered applications. The family has an output tolerance of 1% for the A version (1.5% for the non-A version) and is capable of delivering 150-mA continuous load current. Standard regulator features, such as overcurrent and overtemperature protection, are also included. The LP2985 devices provide very low dropout voltage (typical dropout is 300 mV at 150 mA load current and 7 mV at 1 mA load) and low quiescent current and low ground current, typically 850-µA at 150 mA load, and 75-µA at 1-mA load. Such features save the battery power and extends the time of operating because the circuit can operate properly even if the battery is significantly discharged.

The voltage applied to pin "**3.3V**" of Arduino Zero will be $\mathbf{V_{LDO} - V_{D1}}$, where $\mathbf{V_{LDO}}$ is the LDO output voltage (2.7 or 3.0V) and $\mathbf{V_{D1}}$ is about 0.7V for the 1N4004 rectifier or about 0.3 – 0.5V for the Schottky diode 1N5817 – 1N5819. You can apply other rectifiers with similar parameters as well.

Additionally, the LP2985 devices provide a shutdown feature allowing the regulator to consume only 0.01-uA typically when the ON/OFF pin is pulled low. In our case, the LDO is always ON (pins 1-3 are connected together). A BYPASS pin allows for low-noise operation, with a typical output noise of 30 μV_{RMS}, with the use of a 10 nF bypass capacitor.

When Arduino Zero is powered from a battery, we can apply the power scheme shown in **Fig.69** but without diode D1 (**Fig.70**).

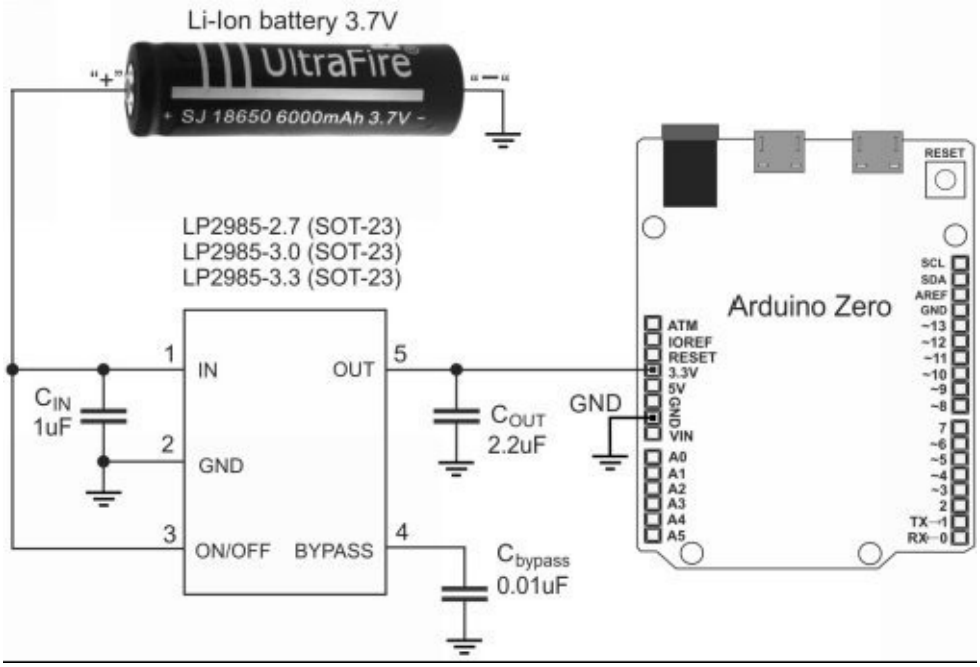


Fig.70

Any of LP2985 devices with an output in the range from 2.7 to 3.3 V can be used in the above circuit.

Note that this circuit must be used only when your Arduino Zero board is powered from a battery. Don't connect any external power sources to Arduino Zero (USB or an external DC power source)!

One more power scheme for use with Arduino Zero is shown in **Fig.71**.

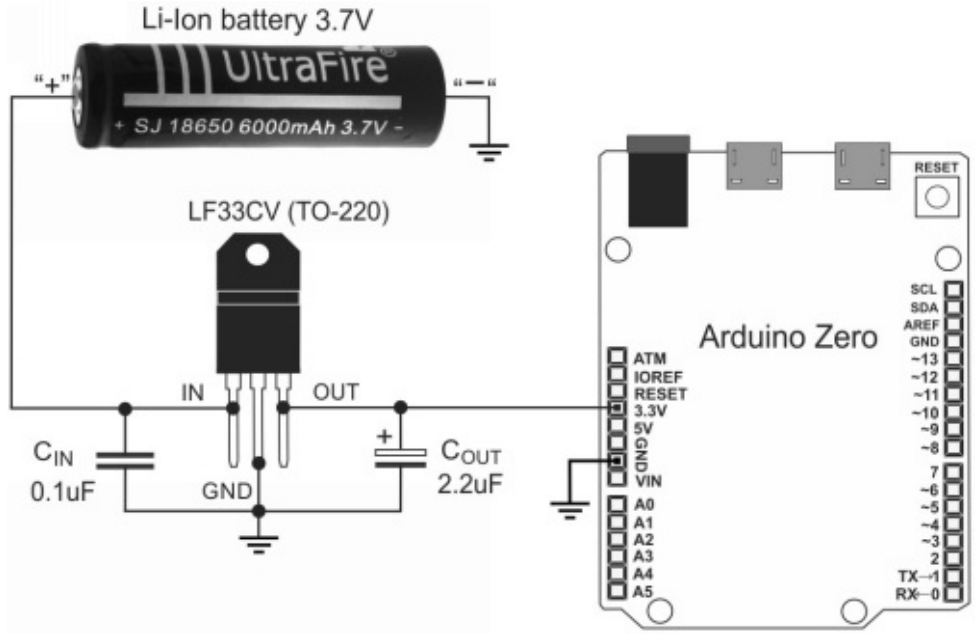


Fig.71

In this circuit we use the LF33CV LDO device of LFXX family produced by STMicro. LF33CV provides the stable DC output of 3.3V (LF25CV LDO with 2.5V output also fits this circuit).

The LFXX device is a very low drop regulator available in a wide range of output voltages. The low drop voltage (0.45 V) and low quiescent current make it particularly suitable for low-noise, low-power applications and especially in battery-powered systems. When used in standard configuration the device requires a capacitor of only 2.2 µF for stability. The LFXX devices consume very low quiescent current (typically 500 µA) and provide an output current up to 500 mA.

We must take into account a few key aspects of designing battery-powered applications. To save battery life we have to apply LDO devices with a low-drop voltage and very low quiescent current. Note that levels of digital signals of Arduino Zero depend on the supply voltage. When the supply voltage reduces, the signal levels reduce as well. This may have a major influence on external circuitry, especially with BJT and MOSFET switches directly driven by Arduino pins, so we must take this fact into account when calculating bias circuits.

To optimize power consuming in battery-powered applications we can use the capability of ATSAMD21G18 CPU to operate in sleep mode with minimal power consuming. This feature allows to extend battery life. One approach to put CPU in sleep mode is to use the Real-Time Counter (RTC). Usually, most Arduino applications perform tasks periodically spending most of the time in loops. We can optimize such applications by using alarm functions from the **RTCZero** library – this allows to perform useful tasks during short intervals after CPU exits sleep mode.

The following projects illustrate this approach.

Project 1

This project allows to build the battery-powered Arduino Zero application running a simple weather station with the BME280 pressure/temperature/humidity sensor. The project uses a breakout board with BME280 (taken from www.ebay.com, **Fig.72**). You can take the similar BME280 board from Adfruit Industries (www.adafruit.com) as well.

Fig.72

The BME280 sensor is controlled via I^2C interface. The data being measured are transferred via a serial interface to the PC. The circuit diagram is shown in **Fig.73**.

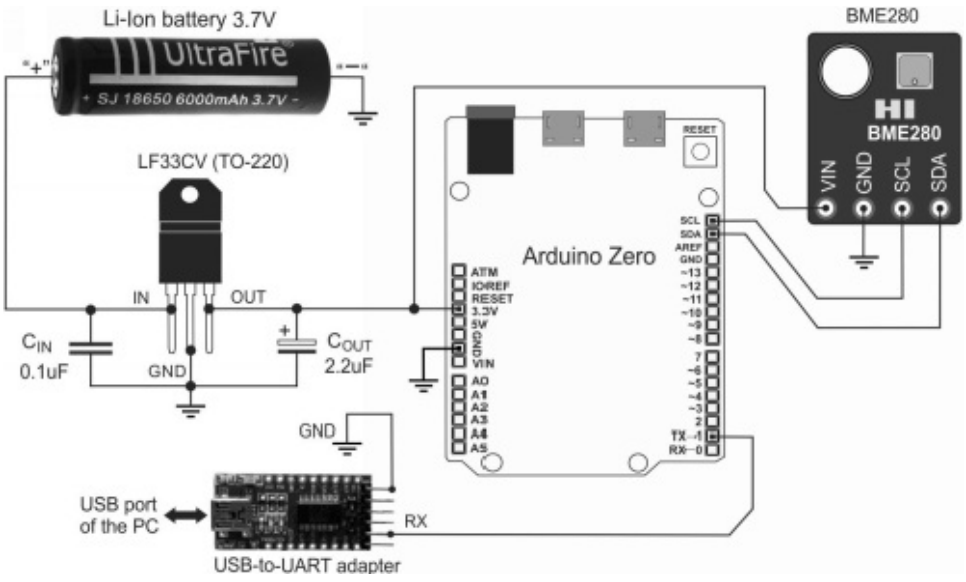


Fig.73

In this circuit, the power circuit includes a Li-Ion battery 3.7V connected to the LF33CV

LDO device which produces the high stable DC voltage +3.3V. The LDO output is wired to pin "**3.3V**" of Arduino Zero and pin "**VIN**" of BME280. **Note** that the ATSAMD21G18 CPU and BME280 can operate at lower voltage, so LDO devices with the output voltage of 3.0/2.7 V can fit this circuit.

The BME280 sensor is connected to Arduino Zero via I^2C interface using **SCL** and **SDA** lines.

The source code of the application (**Listing 36**) uses the Adafruit library developed for BME280.

Listing 36.

```
/***************************************************************************
  This is a library for the BME280 humidity, temperature & pressure sensor

  Designed specifically to work with the Adafruit BME280 Breakout
  ----> http://www.adafruit.com/products/2650

  These sensors use I2C or SPI to communicate with Arduino.

  Adafruit invests time and resources providing this open source code,
  please support Adafruit and open-source hardware by purchasing products
  from Adafruit!

  Written by Limor Fried & Kevin Townsend for Adafruit Industries.
  BSD license, all text above must be included in any redistribution
 ***************************************************************************

#include <Scheduler.h>
#include <RTCZero.h>
#include <Wire.h>
#include <SPI.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>
```

```
/* Create an rtc object */
RTCZero rtc;

/* Change these values to set the current initial time */
const byte seconds = 0;
const byte minutes = 26;
const byte hours = 18;

/* Change these values to set the current initial date */
const byte day = 30;
const byte month = 12;
const byte year = 15;

volatile byte oldSeconds, curSeconds;
volatile int bRead;

int done = 0;
int cnt = 0;

#define BME_SCK 13
#define BME_MISO 12
#define BME_MOSI 11
#define BME_CS 10

#define SEALEVELPRESSURE_HPA (1013.25)

Adafruit_BME280 bme; // I2C is used
//Adafruit_BME280 bme(BME_CS); // hardware SPI
//Adafruit_BME280 bme(BME_CS, BME_MOSI, BME_MISO,  BME_SCK);
```

```
void setup() {
  Serial1.begin(9600);
  Serial1.println(“BME280 test”);

  if (!bme.begin()) {
    Serial1.println(“Could not find a valid BME280 sensor, check wiring!”);
    while (1);
  }
  rtc.begin();

  rtc.setTime(hours, minutes, seconds);
  rtc.setDate(day, month, year);

  curSeconds = 8;
  rtc.setAlarmSeconds(curSeconds);
  rtc.enableAlarm(rtc.MATCH_SS);
  rtc.attachInterrupt(alarmMatch);
  delay(100);
  Scheduler.startLoop(loop2);
  rtc.standbyMode();
}

void loop()
{
if (done == 1)
{
  oldSeconds = curSeconds;
  curSeconds = (curSeconds + 10) % 60;
  rtc.setAlarmSeconds(curSeconds);
  Serial1.print(++cnt);
  Serial1.print(“: CPU awoke after “);
```

```
  Serial1.print(oldSeconds);
  Serial1.println(" s");
  done = 0;
  delay(300);
  rtc.standbyMode(); // Sleep until next alarm match
}
}

void loop2()
{
if (done == 0)
  {
    Serial1.print("Temperature = ");
    Serial1.print(bme.readTemperature());
    Serial1.println(" *C");

    Serial1.print("Pressure = ");

    Serial1.print(bme.readPressure() / 100.0F);
    Serial1.println(" hPa");

    Serial1.print("Approx. Altitude = ");
    Serial1.print(bme.readAltitude(SEALEVELPRESSURE_HPA));
    Serial1.println(" m");

    Serial1.print("Humidity = ");
    Serial1.print(bme.readHumidity());
    Serial1.println(" %");

    Serial1.println("————————");
    delay(150);
```

```
  }
  // IMPORTANT:
  // We must call 'yield' at a regular basis to pass
  // control to other tasks.
  yield();
}

void alarmMatch()
{
done = 1;
}
```

By default, the BME280 device is assigned address 0x77. In this particular case, the address set by manufacturer was 0x76 so the appropriate changes in **Adafruit_BME280.h** header file were needed. Therefore, the line

```
#define BME280_ADDRESS                (0x77)
```

was changed to

```
#define BME280_ADDRESS                (0x76)
```

This is reflected in **Listing 37.**

Listing 37.

```
/**************************************************************************
  This is a library for the BME280 humidity, temperature & pressure sensor

  Designed specifically to work with the Adafruit BME280 Breakout
  ----> http://www.adafruit.com/products/2650

  These sensors use I2C or SPI to communicate with Arduino.
```

Adafruit invests time and resources providing this open source code, please support Adafruit andopen-source hardware by purchasing products from Adafruit!

Written by Limor Fried & Kevin Townsend for Adafruit Industries.

BSD license, all text above must be included in any redistribution

****

```
#ifndef __BME280_H__
#define __BME280_H__

#if (ARDUINO >= 100)
#include “Arduino.h”
#else
#include “WProgram.h”
#endif

#include <Adafruit_Sensor.h>

#ifdef __AVR_ATtiny85__
#include “TinyWireM.h”
#define Wire TinyWireM
#else
#include <Wire.h>
#endif

/*=========================================================
    I2C ADDRESS/BITS
    ———————————*/
#define BME280_ADDRESS                (0x76)
/*=========================================================
```

The **puTTY** terminal application which receives the data from the Arduino application

provides the following output (**Fig.74**).

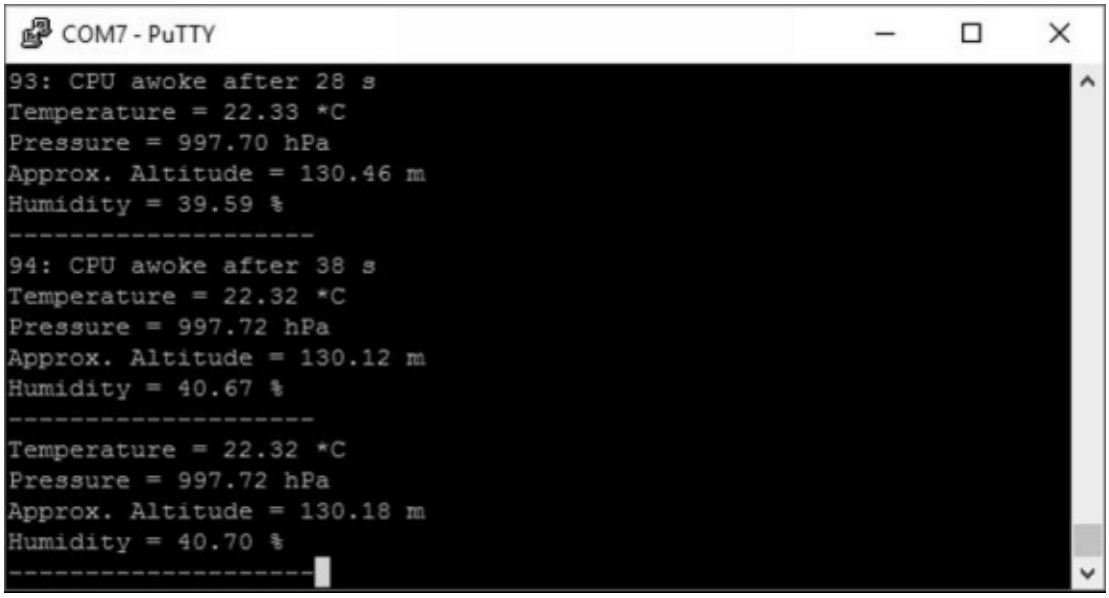


Fig.74

Project 2

This project is the modified version of that discussed earlier. Our application acts as a simple stand-alone weather station based upon the BME280 sensor, but the data being measured are moved to the serial OLED display. The BME280 sensor is controlled via I^2C interface.

The application is powered by a Li-Ion battery with a rated voltage of 3.7V. The LDO circuit (LF33CV chip) provides the steady DC voltage of +3.3V fed to the BME280 breakout board, the OLED display and Arduino Zero. The circuit diagram of the project is shown in **Fig.75**.

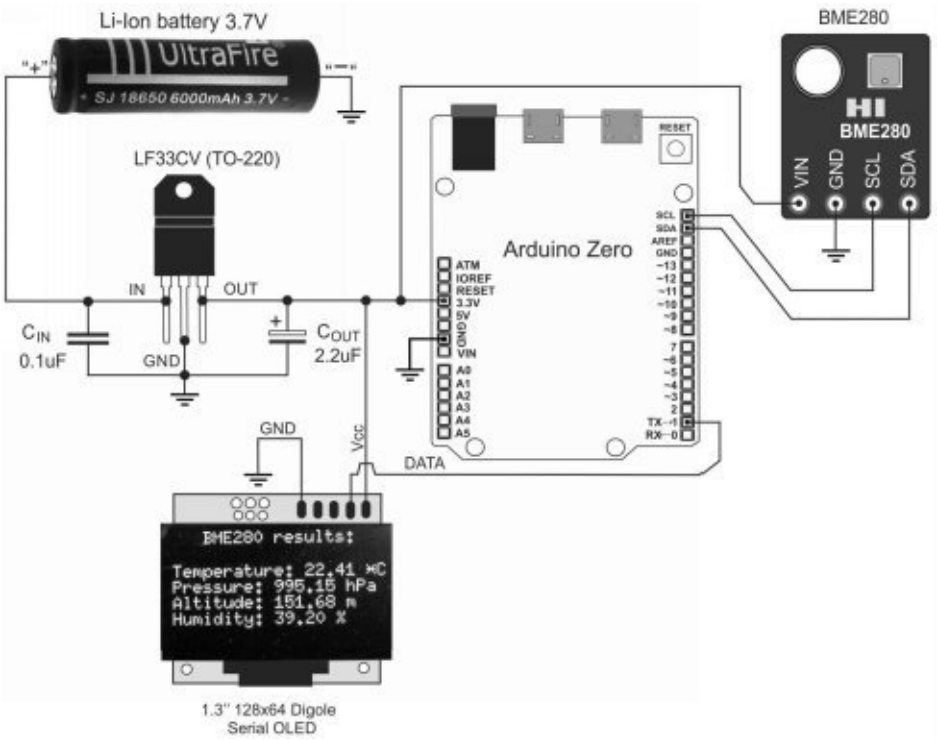


Fig.75

The source code of the application is shown in **Listing 38**.

Listing 38.

```
/***************************************************************************
  This is a library for the BME280 humidity, temperature & pressure sensor
```

Designed specifically to work with the Adafruit BME280 Breakout
–-> http://www.adafruit.com/products/2650

These sensors use I2C or SPI to communicate with Arduino.

Adafruit invests time and resources providing this open source code,
please support Adafruit andopen-source hardware by purchasing products
from Adafruit!

Written by Limor Fried & Kevin Townsend for Adafruit Industries.
BSD license, all text above must be included in any redistribution
****

```
#define _Digole_Serial_UART_  // using UART for data transfer
#include <Scheduler.h>
#include <RTCZero.h>
#include <Wire.h>
#include <SPI.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BME280.h>

#include <DigoleSerial.h>

//——UART setup————
// Pin 1 (TX) on Arduino is connected to RX line (“DATA”) on OLED

DigoleSerialDisp mydisp(&Serial1, 9600);

const unsigned char fonts[] = {0, 6, 10};
```

```
/* Create an rtc object */
RTCZero rtc;

/* Change these values to set the current initial time */
const byte seconds = 0;
const byte minutes = 26;
const byte hours = 18;

/* Change these values to set the current initial date */
const byte day = 30;
const byte month = 12;
const byte year = 15;

volatile byte oldSeconds, curSeconds;
volatile int bRead;

int done = 0;
int cnt = 0;

#define BME_SCK 13
#define BME_MISO 12
#define BME_MOSI 11
#define BME_CS 10

#define SEALEVELPRESSURE_HPA (1013.25)

Adafruit_BME280 bme; // I2C is used
//Adafruit_BME280 bme(BME_CS); // hardware SPI
//Adafruit_BME280 bme(BME_CS, BME_MOSI, BME_MISO,  BME_SCK);

void setup() {
```

```
  Serial1.begin(9600);
  mydisp.begin();
  mydisp.clearScreen(); //Clear screen
  mydisp.disableCursor(); //disable cursor, to enable cursor use: enableCursor();

  mydisp.setPrintPos(0, 0, _TEXT_);
  mydisp.setFont(fonts[2]); // set font 6x10

  mydisp.print("BME280 test");

  if (!bme.begin()) {
    mydisp.setPrintPos(1, 0, _TEXT_);
    mydisp.print("No BME280 sensor is found.");
    while (1);
  }
  rtc.begin();

  rtc.setTime(hours, minutes, seconds);
  rtc.setDate(day, month, year);

  curSeconds = 8;
  rtc.setAlarmSeconds(curSeconds);
  rtc.enableAlarm(rtc.MATCH_SS);
  rtc.attachInterrupt(alarmMatch);
  delay(100);
  Scheduler.startLoop(loop2);
  rtc.standbyMode();
}

void loop()
{
```

```
if (done == 1)
{
  oldSeconds = curSeconds;
  curSeconds = (curSeconds + 10) % 60;
  rtc.setAlarmSeconds(curSeconds);

  mydisp.clearScreen(); //Clear screen
  mydisp.disableCursor(); //disable cursor, to enable cursor use: enableCursor();
  mydisp.setFont(fonts[2]); // set font 6x10
  done = 0;
  delay(300);
  rtc.standbyMode(); // Sleep until next alarm match
}
}

void loop2()
{
if (done == 0)
  {
    mydisp.setPrintPos(0, 0, _TEXT_);
    mydisp.print("   BME280 results:");

    mydisp.setPrintPos(0, 2, _TEXT_);
    mydisp.print("Temperature: ");
    mydisp.print(bme.readTemperature());
    mydisp.print(" *C");

    mydisp.setPrintPos(0, 3, _TEXT_);
    mydisp.print("Pressure: ");
    mydisp.print(bme.readPressure() / 100.0F);
    mydisp.print(" hPa");
```

```
    mydisp.setPrintPos(0, 4, _TEXT_);
    mydisp.print("Altitude: ");
    mydisp.print(bme.readAltitude(SEALEVELPRESSURE_HPA));
    mydisp.print(" m");

    mydisp.setPrintPos(0, 5, _TEXT_);
    mydisp.print("Humidity: ");
    mydisp.print(bme.readHumidity());
    mydisp.print(" %");

    delay(150);
  }
  // IMPORTANT:
  // We must call 'yield' at a regular basis to pass
  // control to other tasks.
  yield();
}

void alarmMatch()
{
done = 1;
}
```

USB in Arduino Zero applications

Due to an USB Native Port Arduino Zero can interact with a host PC appearing as a native Mouse and/or Keyboard. These features are supported by the core libraries allowing Arduino Zero board to appear as a native Mouse and/or Keyboard to a connected computer.

Note that if the Mouse or Keyboard library is constantly running, it will be difficult to program your board. Functions such as Mouse.move() and Keyboard.print() will move your cursor or send keystrokes to a connected computer and should only be called when you are ready to handle them. It is recommended to use a control system to turn this functionality on, like a physical switch or only responding to specific input you can control.

The following two sections are dedicated to designing simple applications using the Keyboard library.

Project 1

This is a simple demo application which illustrates the capabilities of Arduino Zero acting as a USB keyboard. The Arduino Zero board should be connected to the PC via USB Native Port. To test the application connect the mechanical switch to pin **9.** The circuit diagram of the project is shown in **Fig.76**.

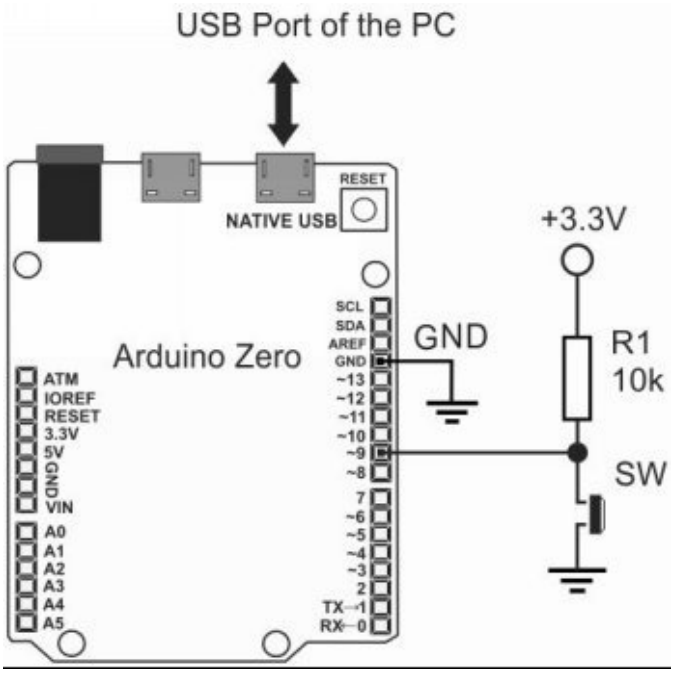


Fig.76

In this circuit, switch **SW** simulates a digital sensor. When the state of **SW** has been changed, the interrupt is triggered on pin **9**. The ISR **pinISR()** simply outputs the state (ON/OFF) of switch **SW** in the **Serial Monitor** window.

The Arduino application puts CPU in sleep mode. When the interrupt on pin **9** is activated, the CPU wakes up and data are transferred to some application such as Notepad running on the host PC.

The application is represented by the following source code (**Listing 39**).

Listing 39.

```
#include “Keyboard.h”

const int LED = 13;

int pinState;
```

```
int done = 0;

void setup()
{
  pinMode(9, INPUT_PULLUP);
  attachInterrupt(9, pinISR, CHANGE);
  delay(5);

    // Entering standby mode when connected
  // via the native USB port causes issues.
  nosleep_ONEXIT();
}

void loop()
{
if (done == 1)
{
    done = 0;

    Keyboard.begin();
    delay(1000);
    Keyboard.print(“The main thread is reached only once.\n”);
    Keyboard.end();
    delay(1000);
    sleep_ONEXIT();
  }
}

void pinISR()
{
  pinState = digitalRead(9);
```

```
  Keyboard.begin();
  if (pinState != 0x0)
    Keyboard.print("The Sensor on pin 9 is HIGH\n");
  else
    Keyboard.print("The Sensor on pin 9 is LOW\n");
  Keyboard.end();
  done = 1;
}

void nosleep_ONEXIT()
{
  SCB->SCR |= 0x0; // sleep mode WITHOUT reentering to sleep mode ON EXIT
    // when returning from an interrupt
  __WFI();
}

void sleep_ONEXIT()
{
    SCB->SCR |= 0x2; // sleep mode WITH reentering to sleep mode ON EXIT
    // when returning from an interrupt
  __WFI();
}
```

Note that Arduino Zero application takes a control over a keyboard only a while, so this doesn't lock your main USB keyboard.

Before building/uploading the binary code we should also configure Arduino IDE properly.

Once code has been uploaded, open the "**Computer Management**" application and observe the list of new devices. An additional HID Keyboard should appear in the list of devices. This is illustrated in **Fig.77** (for Windows 10).

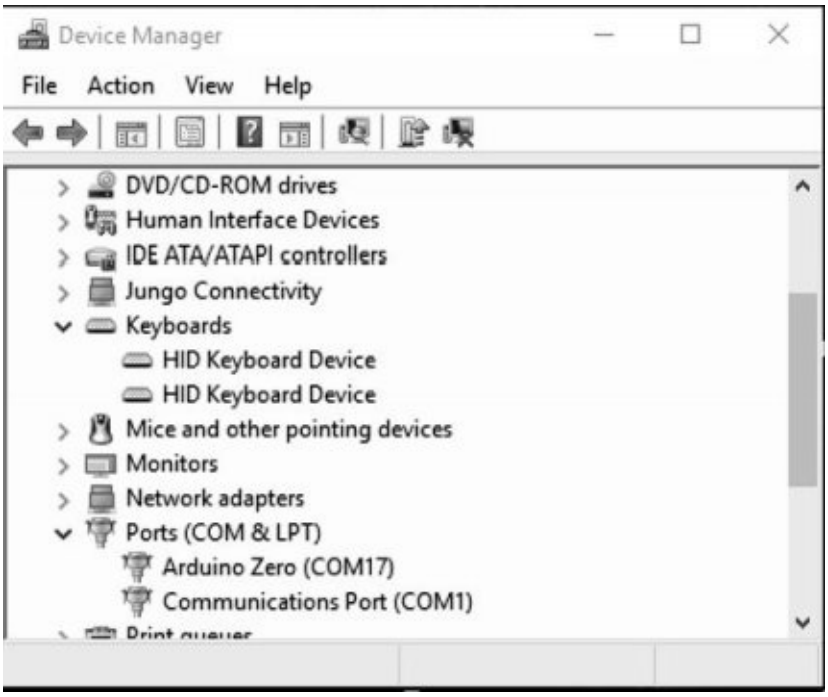


Fig.77

Open some text editor (**Notepad**, for example) and pass control to it. Then you can press/release the switch **SW** and observe the text appearing in the editor window (**Fig.78)**.

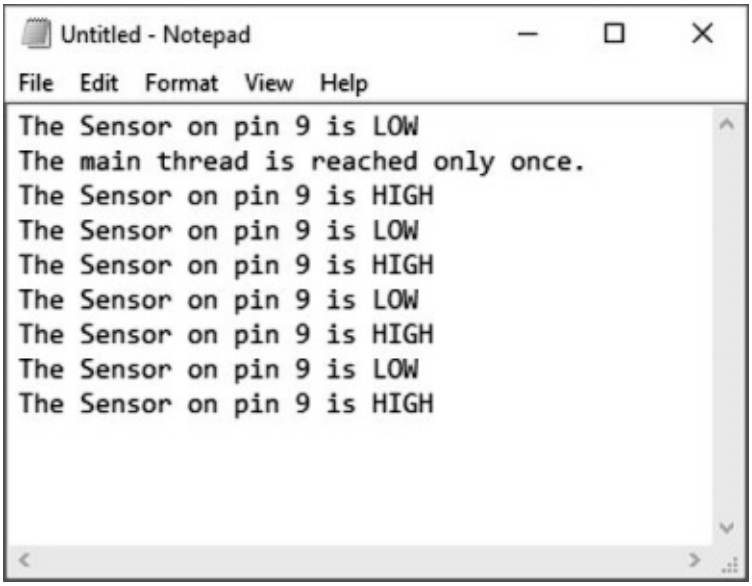


Fig.78

Note that you can save the text in the editor area in some text file if needed. You can also develop a Windows application and process data from Arduino Zero in your own way.

The following project illustrate how to process data from the “Arduino keyboard” in application written in C#. NET.

Project 2

This project illustrates interaction between a Windows application written in C#. NET and Arduino Zero. The Arduino Zero program code measures analog voltage on pin **A0** and transfers the result to the C#. NET application via **USB Native Port**.

The source code of the Arduino Zero application is shown in **Listing 40**.

Listing 40.

```
#include <Keyboard.h>

const double LSB = 3.3/1024; // LSB value for 10-bit resolution with int.ref = 1.0V
int binADC;  // binary code for analog input
double Vin; // analog input value

int cnt = 0;

void setup()
{
  analogReadResolution(10);
}

void loop()
{
    Keyboard.begin();
    delay(500);
    while (cnt <= 10)
    {
    binADC = analogRead(0);
    Vin = (double)binADC * LSB;
    Keyboard.print(++cnt);
    Keyboard.print(“: A0 input: “);
```

```
    Keyboard.print(Vin,3);
    Keyboard.print(" V\n");
    delay(500);
  }
  Keyboard.print("Done.");
  delay(500);
  Keyboard.end();
  while (1);
}
```

The C#. NET application has been written in Microsoft Visual Studio 2015 Community Edition. The main form of the application with components is shown in **Fig.79**.

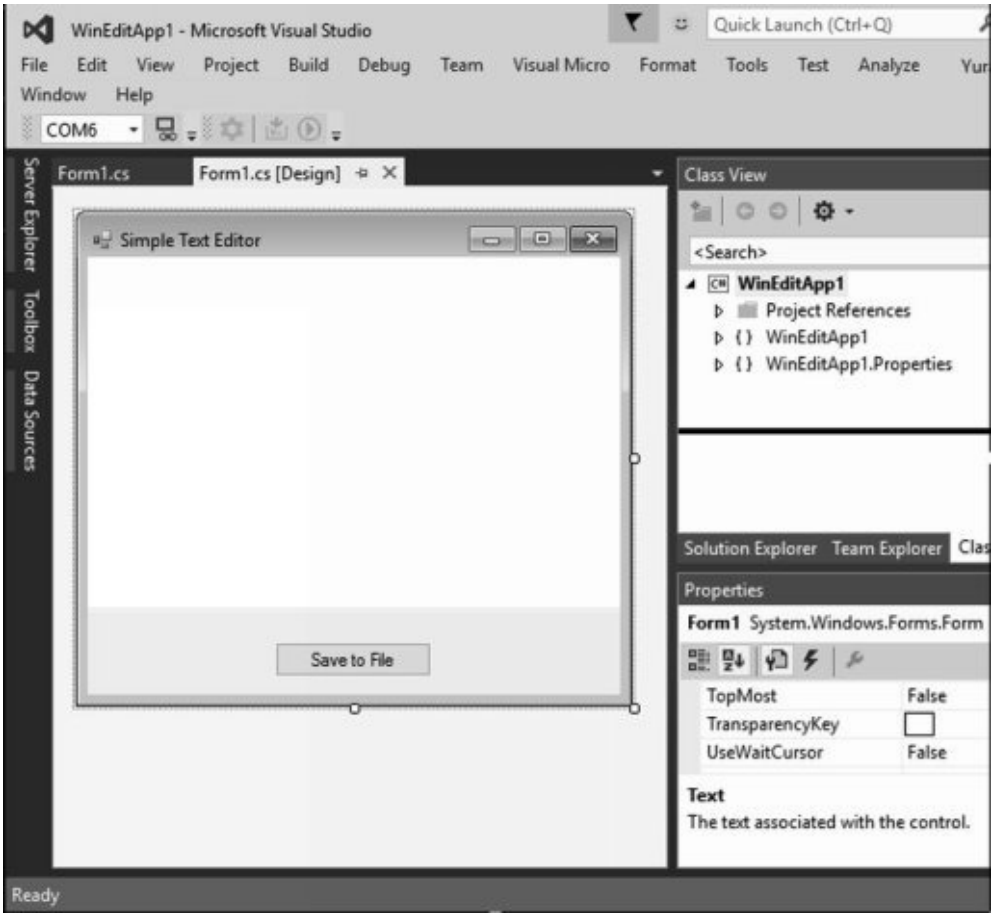


Fig.79

Our C# application includes two components taken from **Toolbox**: **RichTextBox** and **Button**. The **richTextBox1** instance of **RichTextBox** takes the text data from the USB HID device when the cursor is in the edit area. We can edit the text obtained or save it in

the file by pressing the “**Save to File**” button named **button1.**
The source code of the C#. NET application is shown in **Listing 41**.

Listing 41.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.IO;

namespace WinEditApp1
{
  public partial class Form1 : Form
  {
    public Form1()
    {
    InitializeComponent();
    }

  private void button1_Click(object sender, EventArgs e)
  {
    // Declare a new memory stream.
    MemoryStream userInput = new MemoryStream();

    // Save the content of RichTextBox1 to the memory stream
```

```
richTextBox1.SaveFile(userInput, RichTextBoxStreamType.PlainText);
userInput.WriteByte(13); // add CRLF to the end of a text

// Use a SaveFileDialog component to save a MemoryStream to a file.
// Declare a SaveDialogBox
SaveFileDialog saveFileDialog1 = new SaveFileDialog();

// Set the properties on SaveFileDialog1 so the user is
// prompted to create the file if it doesn't exist
// or overwrite the file if it does exist.
saveFileDialog1.Filter = "txt files (*.txt)|*.txt|All files (*.*)|*.*";
saveFileDialog1.FilterIndex = 2;
saveFileDialog1.RestoreDirectory = true;

// Call ShowDialog and check for a return value of DialogResult.OK,
// which indicates that the file was saved.
DialogResult result = saveFileDialog1.ShowDialog();
Stream fileStream;

if (result == DialogResult.OK)
{
// Open the file, copy the contents of memoryStream to fileStream,
  // and close fileStream. Set the memoryStream.Position value to 0
// to copy the entire stream.
fileStream = saveFileDialog1.OpenFile();
userInput.Position = 0;
userInput.WriteTo(fileStream);
fileStream.Close();
}
}
}
```

}

In this code, the **button1_Click** event handler create two objects, a **MemoryStream** (named **userInput**) and **SaveFileDialog** object (named **saveFileDialog1**). The **userInput** stream moves data from **richTextBox1** to a memory buffer, and **saveFileDialog1** allows to save the content of this memory buffer in a text file.

The window of the running Windows C#. NET application is shown in **Fig.80**.

Fig.80

As you can see, the output data appears in the **RichTextBox** window. You can edit the text and/or save this text in a text file by pressing the "**Save to File**" button (**Fig.81**).

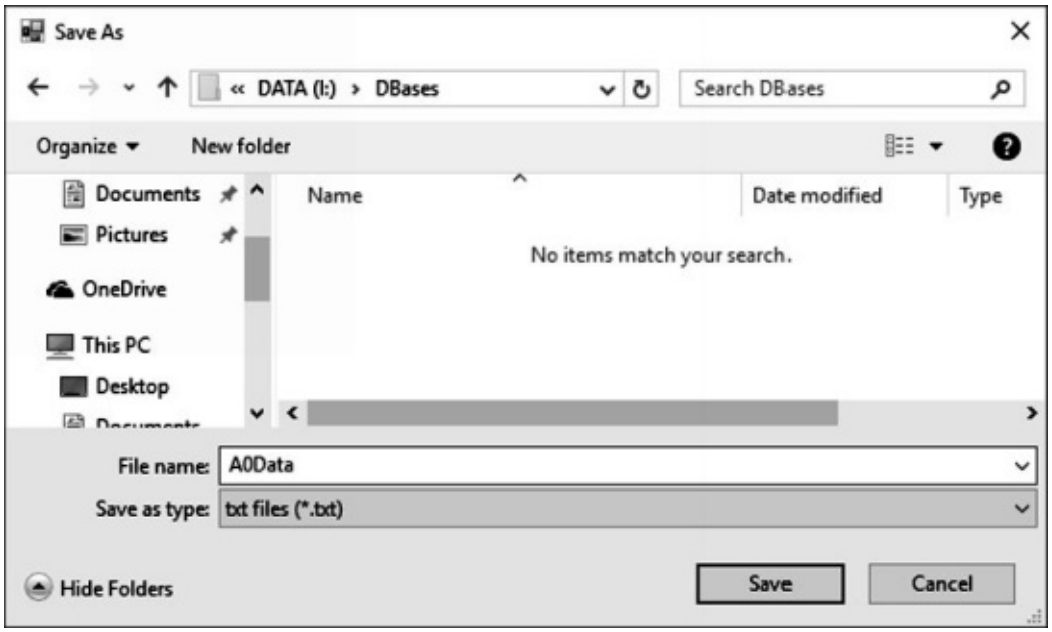

Save As
« DATA (I:) › DBases
Search DBases
Organize
New folder
Documents
Pictures
OneDrive
This PC
Desktop
Name
Date modified
Type
No items match your search.
File name: A0Data
Save as type: txt files (*.txt)
Hide Folders
Save
Cancel

Fig.81

Miscellaneous Arduino Zero projects

This section represents various projects using Arduino Zero. The following two projects show how to operate with data storages such as SD (microSD) cards.

Using SD cards: project 1

This project uses a low-cost (about 1$ on www.ebay.com) shield arranged to operate with an microSD card (**Fig.82**).

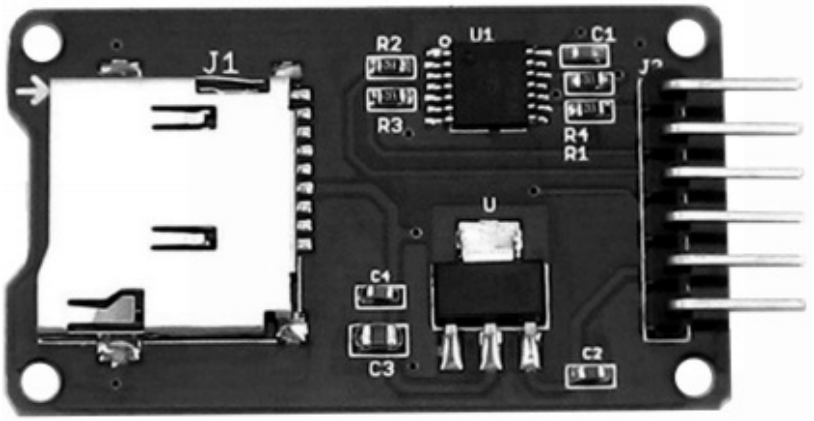


Fig.82

Below is a brief description of this module given by a seller.

The module (MicroSD Card Adapter) is a Micro SD card reader module with an SPI interface implementing the file system driver for reading/writing files. The MicroSD Card adapter provides the level conversion circuit to operate either with +5V or +3.3V. The power supply to this board may be in the range of 4.5V – 5.5V. The on-board LDO voltage regulator provides +3.3V for inner circuits.

The next project illustrates basic operations with data on a microSD card.

The circuit diagram of the project is shown in **Fig.83**.

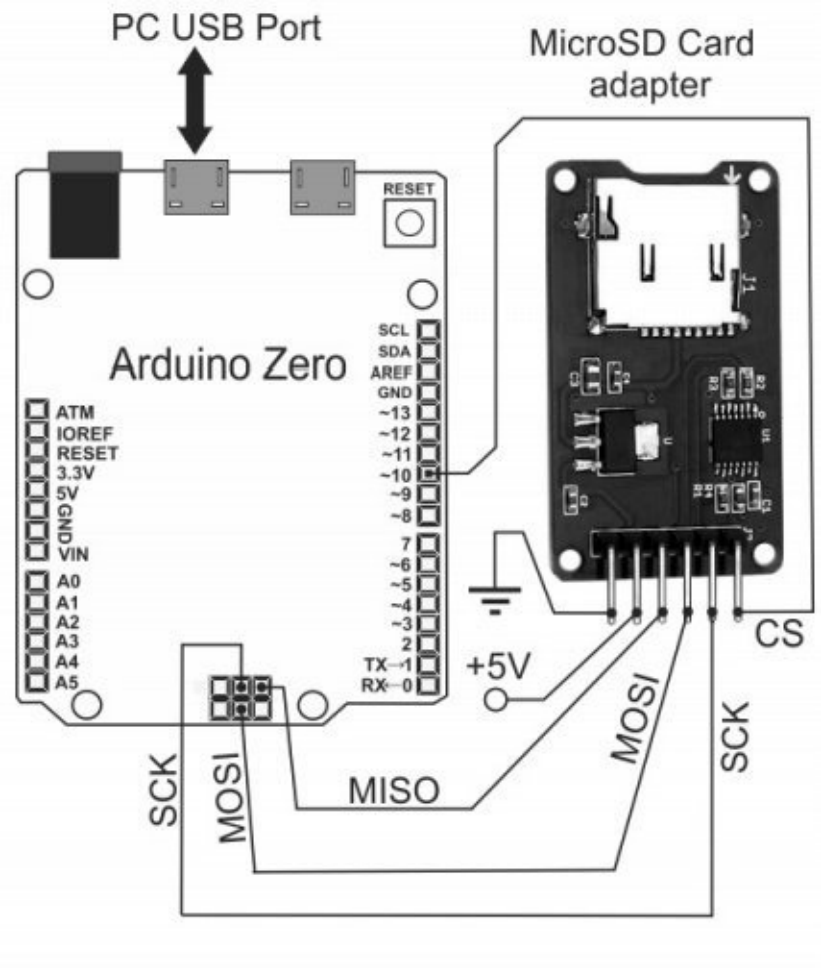


Fig.83

In this circuit, the MicroSD Card Adapter is connected to Arduino Zero via the SPI-compatible interface. The SPI signal lines on the Arduino Zero board are taken from the SPI header, the **CS** signal is taken from pin **10**. The power to MicroSD Card Adapter is taken from pin "**5V**" of Arduino Zero.

The source code of the application illustrating basic operations with files is shown in **Listing 42**.

Listing 42.

```
#include <SPI.h>
#include <SD.h>

File myFile;
```

```
void setup() {
  // Open serial communications and wait for port to open:
  Serial.begin(9600);
  Serial.print("Initializing SD card…");

  if (!SD.begin(10)) {       // specifies the pin where (SS) CS goes to
    Serial.println("initialization failed!");
    return;
  }
  Serial.println("initialization done.");

  // open the file. Note that only one file can be open at a time,
  // so you have to close this one before opening another.
  if (SD.exists("testsd.txt"))
  {
    SD.remove("testsd.txt");
    delay(1000);
  }

  myFile = SD.open("testsd.txt", FILE_WRITE);

  // if the file opened OK, write to it:
  if (myFile)
  {
    Serial.print("Writing to testsd.txt…");
    myFile.println("THE STRING TO BE WRITTEN TO THE FILE.");
    // close the file:
    myFile.close();
    Serial.println("done.");
  } else {
    // if the file didn't open, print an error:
```

```
    Serial.println("error opening testsd.txt");
  }

  // re-open the file for reading:
  myFile = SD.open("testsd.txt");
  if (myFile)
  {
    Serial.println("Contents of testsd.txt:");

    // read from the file until there's nothing else in it:
    while (myFile.available())
    {
    Serial.write(myFile.read());
    }
    // close the file:
    myFile.close();
  } else {
    // if the file didn't open, print an error:
    Serial.println("error opening test.txt");
  }
}

void loop() { }
```

The running application provides the following output in the **Serial Monitor** window (**Fig.84**).

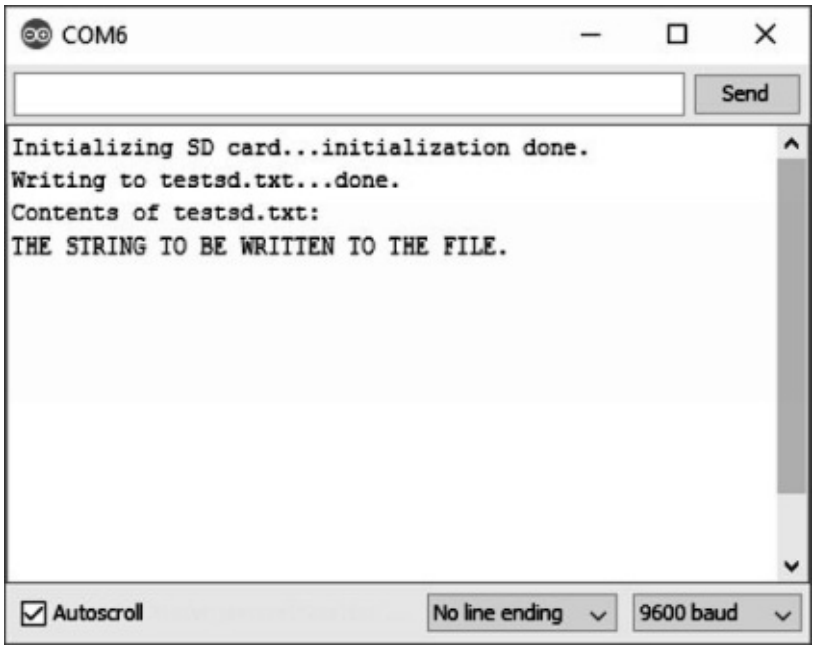

COM6
Send
Initializing SD card...initialization done.
Writing to testsd.txt...done.
Contents of testsd.txt:
THE STRING TO BE WRITTEN TO THE FILE.
Autoscroll
No line ending
9600 baud

Fig.84

Using SD cards: project 2

This project illustrates how to create an application that measures analog signals on input **A0** of Arduin Zero and writes the data obtained in the text file on a MicroSD card. After 10 measurements the file is closed and data saved in this file are then sent to **Serial Monitor**.

The circuit diagram of the project (**Fig.85**) is almost the same as that for the previous project. The only addition is the potentiometer **Rp** providing the test analog signal **Vin** to pin **A0**.

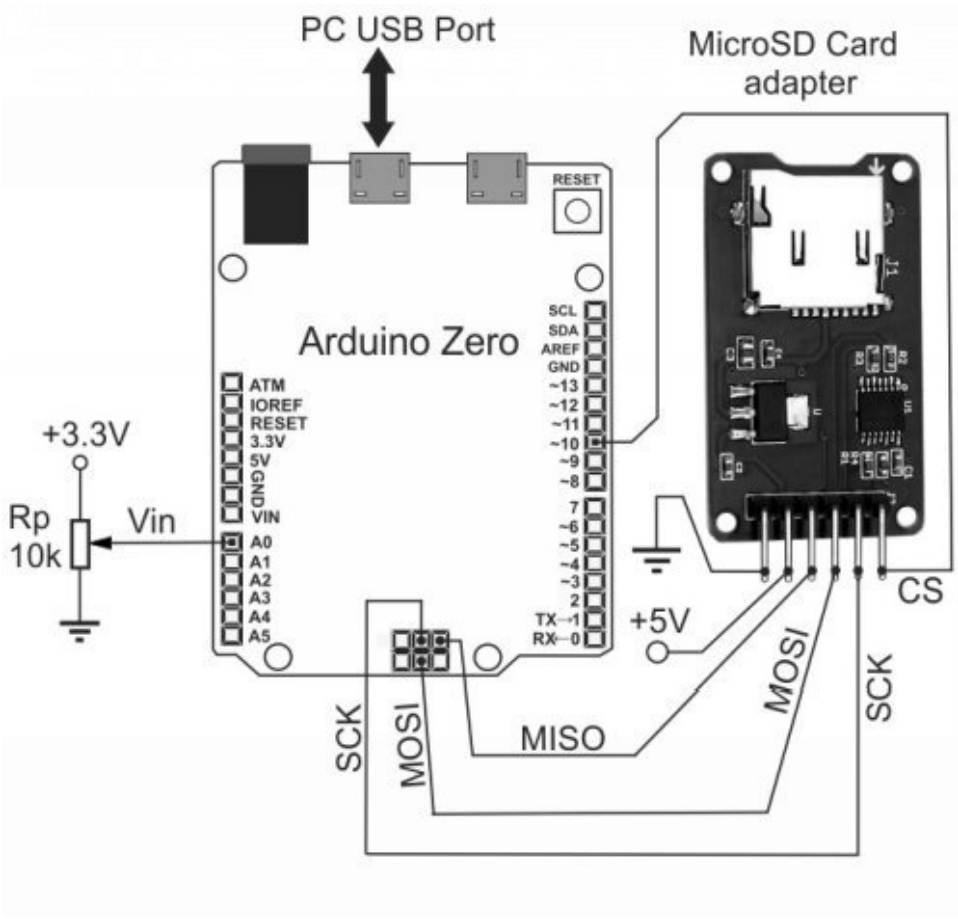


Fig.85

The source code of the application is shown in **Listing 43**.

Listing 43.

```
#include <SPI.h>
#include <SD.h>
```

```
int binADC;
const double LSB = 3.33 / 1024;
double vin0;

int cnt = 0;
File myFile;

void setup()
{
  analogReadResolution(10);
  // Open serial communications and wait for port to open:
  Serial.begin(9600);
  Serial.print("Initializing SD card…");

  if (!SD.begin(10))
  {
    // specifies the pin where (SS) CS goes to
    Serial.println("initialization failed!");
    return;
  }
  Serial.println("initialization done.");

  // removing old adcdat.txt file
  if (SD.exists("adcdat.txt"))
  {
    SD.remove("adcdat.txt");
  }
  delay(250);
  myFile = SD.open("adcdat.txt", FILE_WRITE);
  if (!myFile)
```

```
  {
    // if the file didn't open, print an error:
    Serial.println("error opening adcdat.txt");
    return;
  }
}

void loop()
{
while (1)
{
  binADC = analogRead(0);
  vin0 = LSB * binADC;

  myFile.print(++cnt);
  myFile.print(": A0 channel: ");
  myFile.print(vin0, 3);
  myFile.println(" V");
  if (cnt > 9)
  {
    myFile.close();
    Serial.println(" completed.");
    break;
  }
  else
  {
    delay(3000);
    Serial.print(".  ");
    continue;
  }
}
```

```
delay(50);
// re-open the file for reading:
myFile = SD.open("adcdat.txt");
if (myFile)
{
  Serial.println("——adcdat.txt——");
  // read from the file until there's nothing else in it:
  while (myFile.available())
    Serial.write(myFile.read());
    // close the file:
  myFile.close();
  }
  else
  {
    // if the file didn't open, print an error:
    Serial.println("error opening adcdat.txt");
  }
while (1);
}
```

The **Serial Monitor** window (**Fig.86**) reflects the output provided by the application.

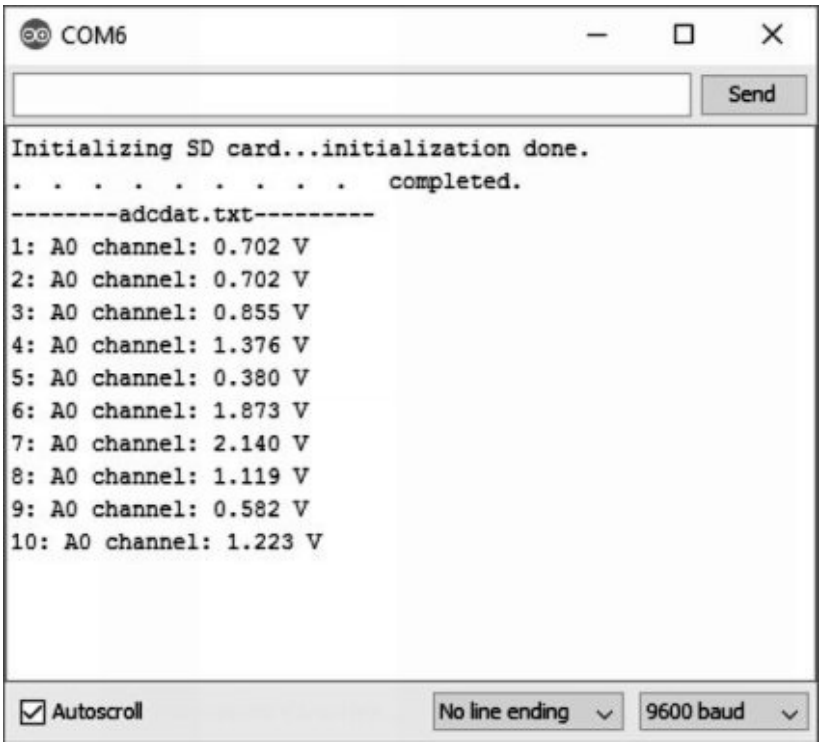

COM6
Send
Initializing SD card...initialization done.
. . . . . completed.
--------adcdat.txt---------
1: A0 channel: 0.702 V
2: A0 channel: 0.702 V
3: A0 channel: 0.855 V
4: A0 channel: 1.376 V
5: A0 channel: 0.380 V
6: A0 channel: 1.873 V
7: A0 channel: 2.140 V
8: A0 channel: 1.119 V
9: A0 channel: 0.582 V
10: A0 channel: 1.223 V
Autoscroll
No line ending
9600 baud

Fig.86

The high-precision D/A converter controlled by PWM

This project illustrates the design of a programmable high-precision 10-bit Digital-to-Analog (DAC) converter using the LTC2644 device from Linear Tech. The DAC is very simple – it requires only the PWM signal on LTC2644 input for adjusting the analog output voltage. This design allows to add an additional DAC channel to your application. The LTC2644 provides 2 PWM-to-DAC channels, so you can add two additional DACs to your project if needed.

The circuit diagram of the project is shown in **Fig.87**.

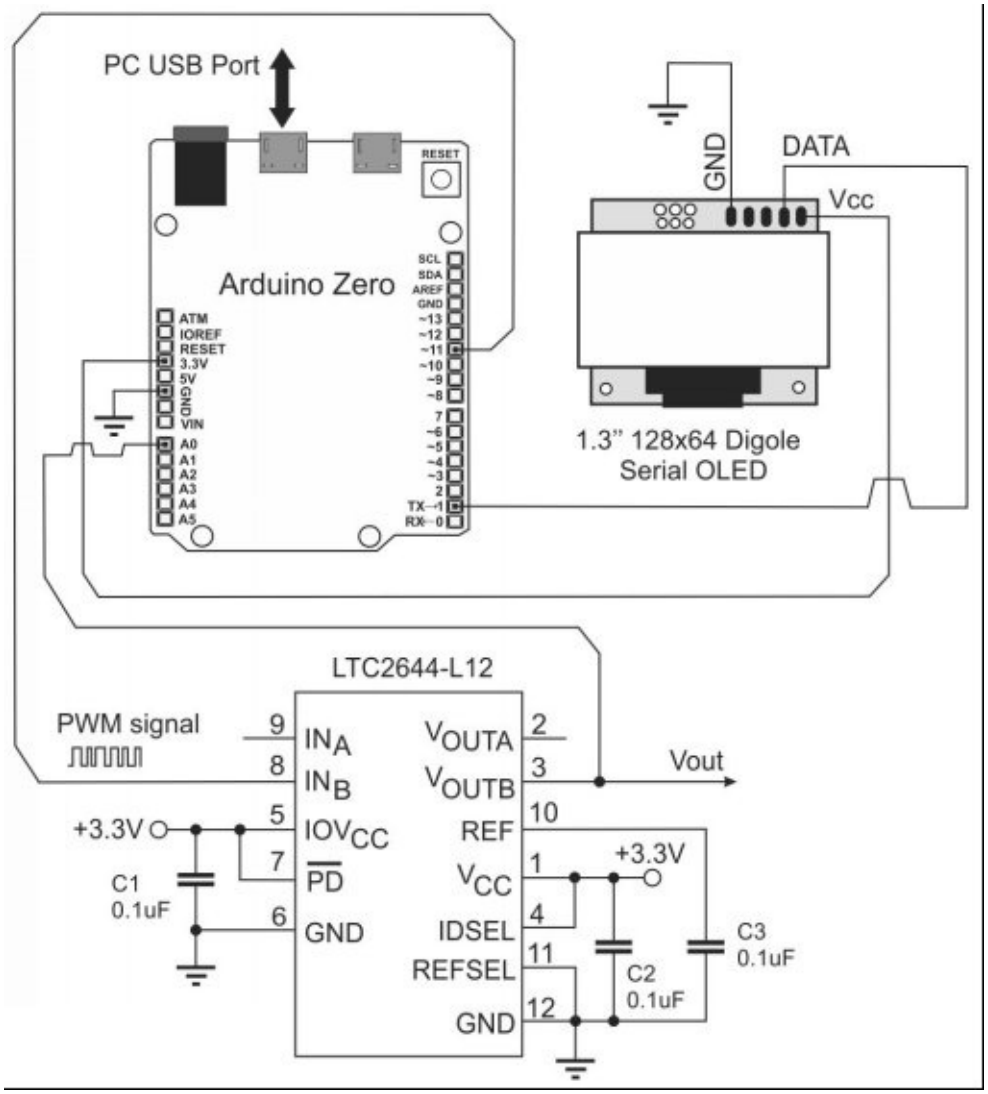


Fig.87

In this circuit, the PWM signal provided by Arduino Zero (pin **11**) goes to input $\mathbf{IN_B}$ (pin **8**) of LTC2644-L12. The DC analog voltage is taken from output $\mathbf{V_{OUTB}}$ (pin **3**) of the LTC2644-L12 device. The value on pin $\mathbf{V_{OUTB}}$ is also measured by Arduino Zero application (pin **A0**) and sent to the OLED display. The power to LTC2644-L12 is taken from pin "**3.3V**" on the Arduino Zero board.

To transfer data to the OLED display the **Serial1** interface is used (pin "**TX → 1**" of

Arduino Zero).
The source code for the application is shown in **Listing 44**.

Listing 44.

```
#define _Digole_Serial_UART_  // using UART for data transfer

#include <DigoleSerial.h>

//——UART setup————
// Pin 1(TX)on Arduino is connected to RX line (“DATA”) on OLED

DigoleSerialDisp mydisp(&Serial1, 9600);

const int pwPin = 11;
int binADC;
const double LSB = 3.28 / 4096.0;
double Vin;
const unsigned char fonts[] = {0, 6, 10};

void setup(void)
{
analogReadResolution(12);
analogWriteResolution(10);
pinMode(pwPin, OUTPUT);

Serial1.begin(9600);
mydisp.begin();
mydisp.clearScreen(); //Clear screen
mydisp.disableCursor(); //disable cursor, enable cursore use: enableCursor();
analogWrite(pwPin, 593); // a test PWM signal
```

```
delay(100);
}

void loop(void)
{

binADC = analogRead(0);
Vin = LSB * binADC;

mydisp.clearScreen();
mydisp.setFont(fonts[2]);

mydisp.setPrintPos(0, 0, _TEXT_);
mydisp.print(“DAC output: “);
mydisp.print(Vin, 3);
mydisp.println(“V”);
delay(3000);
}
```

The simple programmable square-wave oscillator

This project illustrates how to develop a simple square-wave generator with the output frequency linearly proportional to analog input voltage. This application reads the analog input voltage that varies from 0.1 to 1.0 V. The input control signal arrives on pin **A0**, the output digital pulse train is taken from pin **11**. The value of the output frequency is reflected on the OLED display.

The output frequency **Fou**t is dependent on the analog input voltage **Vin** as

Fout = K x Vin,

where **K** – coefficient for calculating a frequency in KHz, **Vin** – input voltage on pin **A0**. In this project, the analog input signal is taken from the wiper of a potentiometer **Rp** connected as a voltage divider.

The circuit diagram for the project is shown in **Fig.88**.

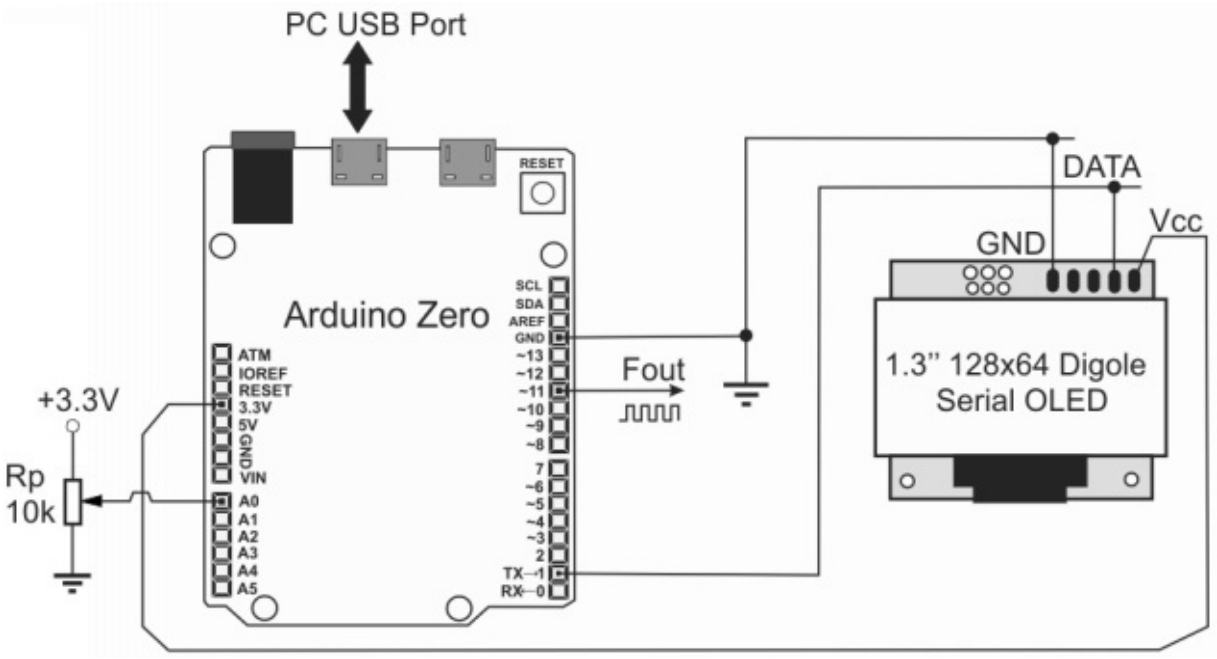


Fig.88

The source code for this application is shown in **Listing 45**.

Listing 45.

```
#define _Digole_Serial_UART_  // using UART for data transfer

#include <DigoleSerial.h>

//——UART setup————
// Pin 1(TX)on Arduino is connected to RX line (“DATA”) on OLED

DigoleSerialDisp mydisp(&Serial1, 9600);

const float KFreq = 100.0; // the coefficient for calculating a freq. in KHz
const int tonePin = 11;
float newFreq, curFreq;
int binADC;
const float LSB = 3.33 / 1024;
float vin;
const unsigned char fonts[] = {0, 6, 10};

void setup()
{
analogReadResolution(10);
Serial1.begin(9600);
mydisp.begin();
mydisp.clearScreen(); //CLear screen
mydisp.disableCursor(); //disable cursor, enable cursore use: enableCursor();

mydisp.setPrintPos(0, 0, _TEXT_);
mydisp.setFont(fonts[0]); // set def. font (6x10)
curFreq = 10; // initial freq = 10 KHz
tone(tonePin, curFreq); // an initial freq. on pin 11 = 5000 Hz
}
```

```
void loop()
{
  binADC = analogRead(0);
  vin = LSB * binADC;

  mydisp.clearScreen();
  mydisp.setPrintPos(0, 0, _TEXT_);
  mydisp.print("A0 Input: ");
  mydisp.print(vin, 3);
  mydisp.print("V");

  if ((vin >= 0.1) && (vin <= 1.0))
  {
    newFreq = vin * KFreq; // new freq. in KHz
    if (newFreq != curFreq)
    {
    curFreq = newFreq;
    tone(tonePin, curFreq * 1000);
    }
  }
  mydisp.setPrintPos(0, 1, _TEXT_);
  mydisp.print("Freq.: ");
  mydisp.print(curFreq,2);
  mydisp.print(" KHz");
  delay(3000);
}
```

The assembled circuit is shown in **Fig.89**.

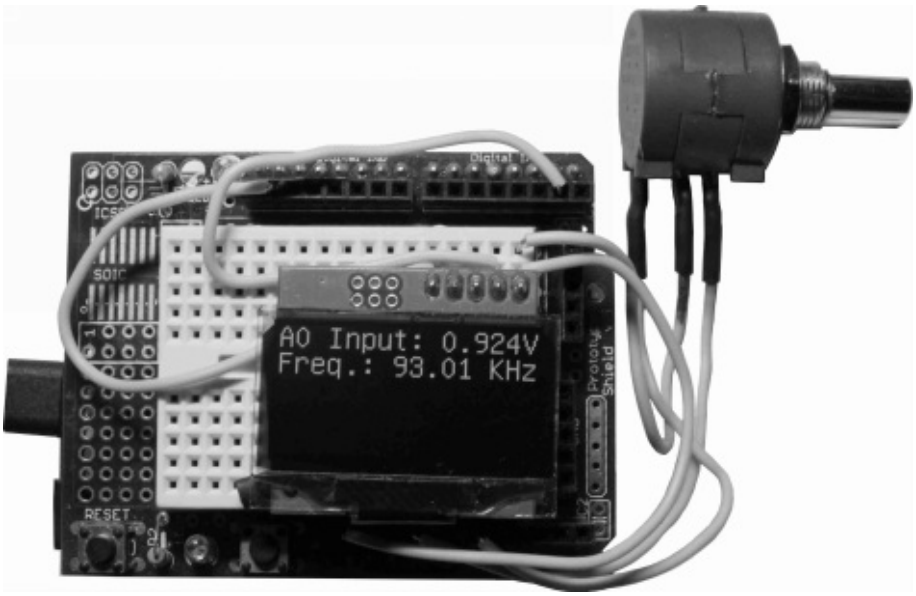


Fig.89