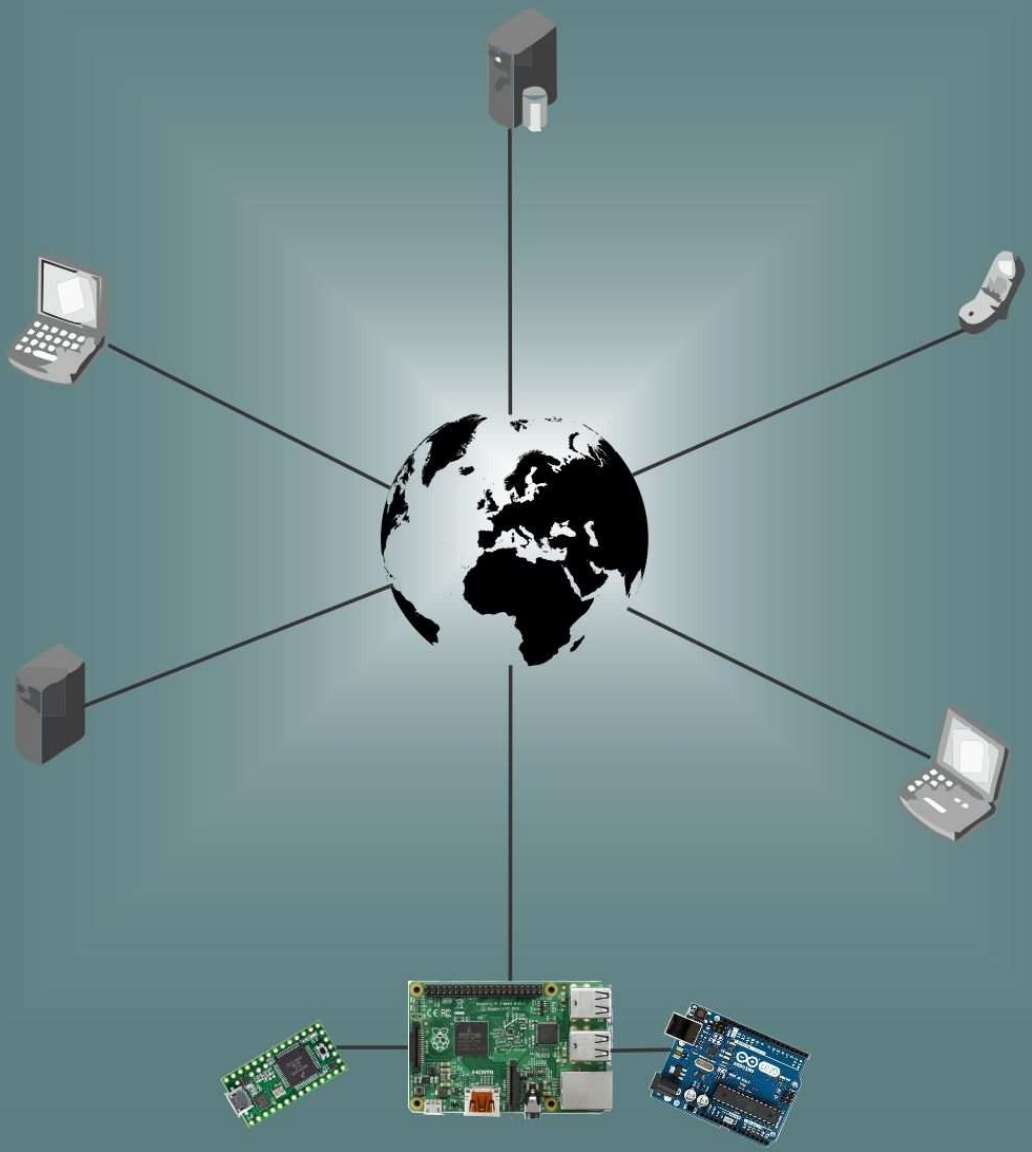
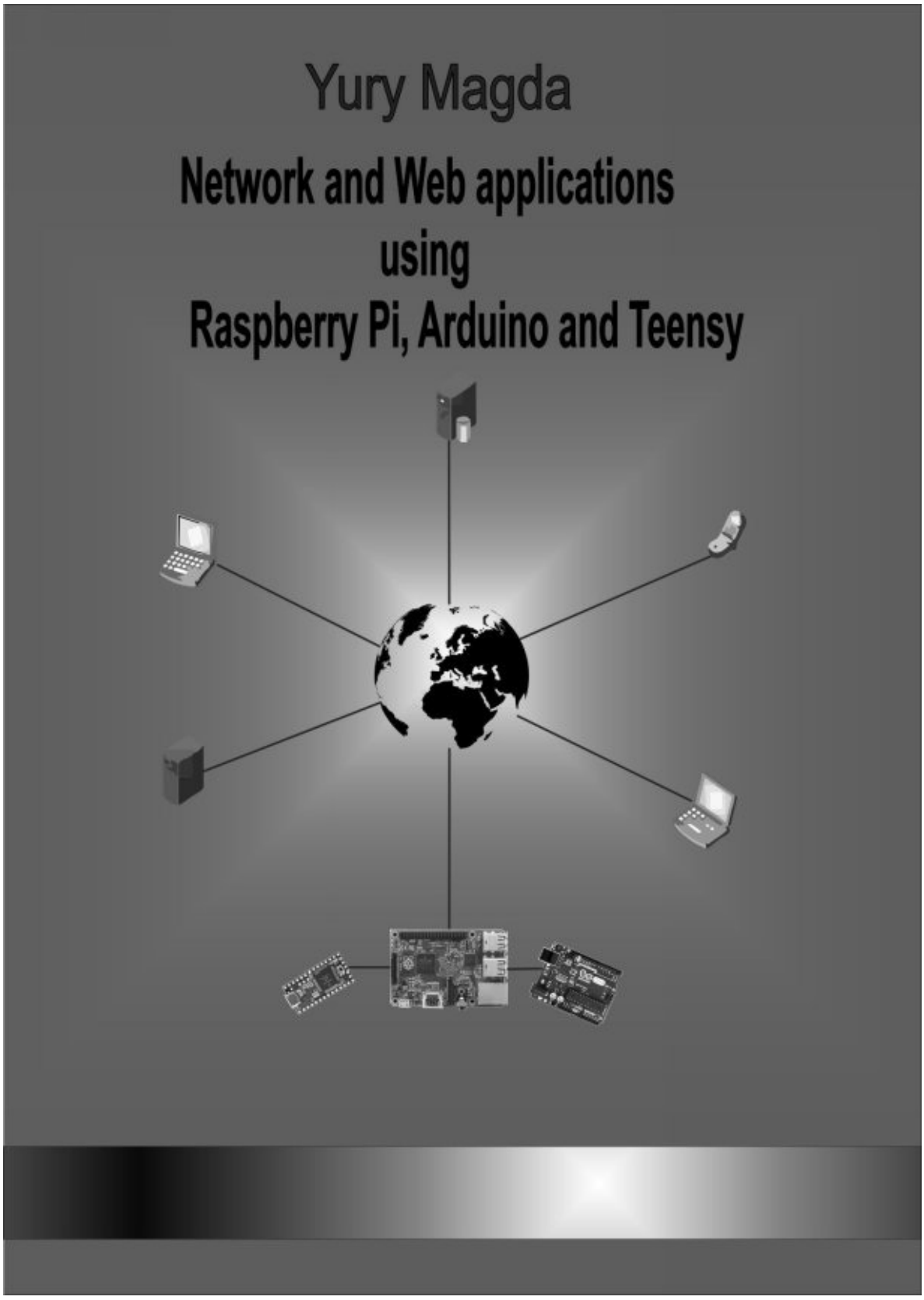
Yury Magda

Network and Web applications using Raspberry Pi, Arduino and Teensy





Network and Web applications
using
Raspberry Pi, Arduino and Teensy

By Yury Magda

The programs, examples, and applications presented in this book have been included for their instructional value. The author offer no warranty implied or express, including but not limited to implied warranties of fitness or merchantability for any particular purpose and do not accept any liability for any loss or damage arising from the use of any information in this book, or any error or omission in such information, or any incorrect use of these programs, procedures, and applications.

Contents

Introduction

The popular Raspberry Pi miniature computer and small development boards operating in real-time environment can be used together for designing complex and powerful measurement and control systems. In these systems, the high-level programming environment of the Raspbian OS running on the Raspberry Pi can be applied for high-level data processing, while real-time signal processing will be performed by some popular and low-cost development board such as Arduino or Teensy 3.1. A great benefit of such configurations is that a developer doesn’t need to build external circuitry that may take a long time.

This highly practical guide describes how to build measurement systems that may include the Raspberry Pi 2, Arduino Uno or Teensy 3.1 developments boards and may be controlled over a TCP/IP network or the Internet. Almost all applications from this guide use serial communications between Raspberry Pi, Arduino or Teensy 3.1, so a few projects at the beginning of the book illustrate the basics of serial communications when designing simple measurement systems.

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 designed using Raspberry Pi 2 Model B and Arduino Uno or Teensy 3.1 boards. The source code for applications running on Raspbian OS was developed in Python. The Arduino and Teensy 3.1 applications were developed using Processing and supplemental libraries. Most projects described in this guide can be easily improved or modified if necessary.

Disclaimer

The design techniques described in this book have been tested on the Raspberry Pi 2 Model B, Arduino Uno and Teensy 3.1 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.

Interfacing Raspberry Pi to Arduino Uno with UART

A few projects from this guide will use the Raspberry Pi and Arduino connected via a serial interface (UART). To make connection between those boards we will employ USB-to-Serial adapters with TTL-compatible output – this allows to avoid using 3.3V/5V level converters. In addition, there will be no need to configure an extra UART interface on the Raspberry Pi – the system will make this job for us.

The hardware interface between the Raspberry Pi and Arduino boards is shown in **Fig.1**.

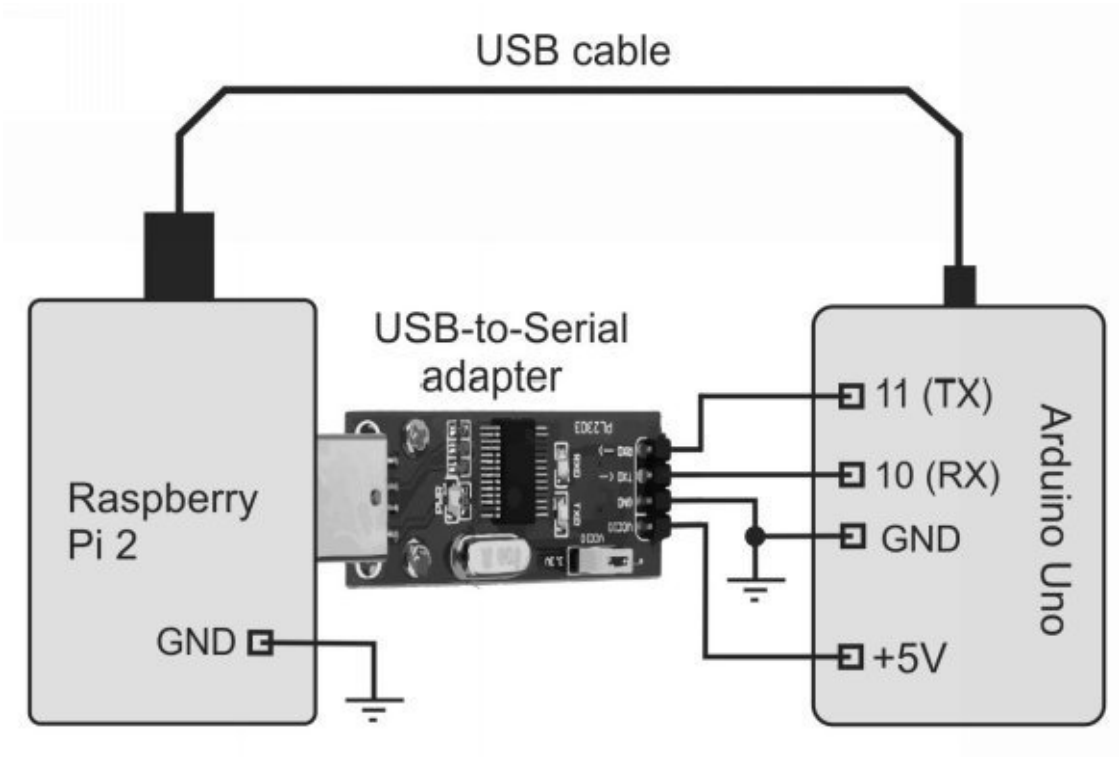


Fig.1

In this configuration, the Arduino Uno is powered by the Raspberry Pi 2 board through a USB cable. Note that the DC power supply of the Raspberry Pi 2 should provide about 2A that is enough for feeding a few external interfaces. Alternatively, we can feed the Arduino Uno from a separate +5V DC power supply.

The USB-to-Serial adapter is connected to one of the USB jacks on the Raspberry Pi 2 board. The UART side of the adapter is connected to the Arduino Uno through pin 11 (TX) and 10 (RX). Note that most USB-to-UART adapters usually have the following pins:

- TX, which allows to transfer data to the receiver;
- RX, through which data are received;

- +5V input, where the +5V voltage is fed;
- +3.3V input, where the +3.3V voltage is fed

The power supply applied to the adapter (5V or 3.3V) depends on the signal levels provided by external circuitry. Arduino Uno operates with +5V TTL levels, so we must apply +5V to the appropriate power pin on the USB-to-Serial adapter. In our projects, the +5V voltage is directly taken from the 5V pin on the Arduino Uno board (see **Fig.1**). The TX signal line of USB-to-Serial adapter goes to pin 10 of Arduino Uno which will be configured as RX in software; the RX signal line of the adapter is connected to pin 11 of Arduino Uno which should be configured as the TX signal line in software.

USB-to-Serial interface adapters

Almost all USB-to-Serial adapters can be used for interfacing the Raspberry Pi and Arduino. The projects from this guide use two adapters; one is built around the PL2303 chip while the other uses the FT232RL IC. The PL2303-based adapter manufactured by Profilic Tech. is shown in **Fig.2**.

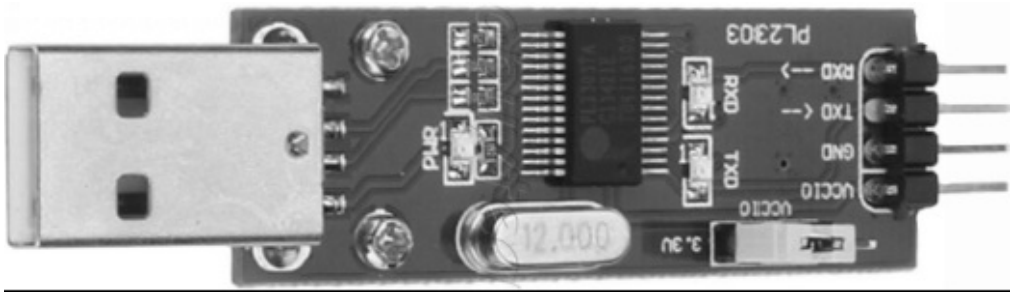


Fig.2

To view the configuration of the PL2303-based USB-to-Serial adapter in Raspbian OS we can enter the **lsusb** command:

pi@raspberrypi ~/Developer $ lsusb

Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp.

Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub

Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp.

Bus 001 Device 004: ID 067b:2303 Prolific Technology, Inc. PL2303 Serial Port

Bus 001 Device 005: ID 1a86:7523 QinHeng Electronics HL-340 USB-Serial adapter

Here the string containing PL2303 represents our USB-to-Serial adapter. In this configuration, the HL-340 chip represents to the Arduino Uno USB port connected to the Raspberry Pi board through the USB cable.

To dump the physical USB device hierarchy as a tree we can type the **lsusb** command followed by the **-t** option:

pi@raspberrypi ~/Developer $ lsusb -t

/: Bus 01.Port 1: Dev 1, Class=root_hub, Driver=dwc_otg/1p, 480M

|__ Port 1: Dev 2, If 0, Class=hub, Driver=hub/5p, 480M

|__ Port 1: Dev 3, If 0, Class=vend., Driver=smsc95xx, 480M

|__ Port 2: Dev 4, If 0, Class=vend., Driver=pl2303, 12M

|__ Port 4: Dev 5, If 0, Class=vend., Driver=ch341, 12M

One more USB-to-Serial adapter used in our projects and equipped with FT232RL chip is shown in **Fig.3**.

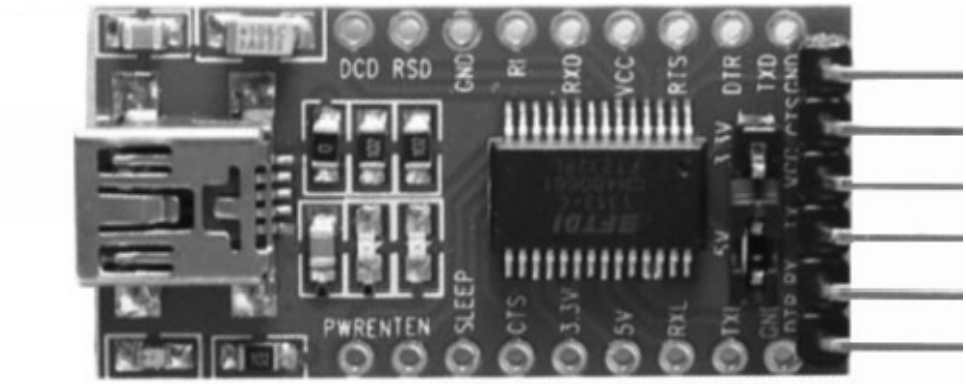


Fig.3

When connected to the Raspberry Pi this adapter appears as follows:

```
pi@raspberrypi ~/Developer $ lsusb
Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp.
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp.
Bus 001 Device 004: ID 0403:6001 Future Technology Devices International, Ltd FT232
USB-Serial (UART) IC
Bus 001 Device 005: ID 1a86:7523 QinHeng Electronics HL-340 USB-Serial adapter
```

The physical USB hierarchy in this case will look like the following:

```
pi@raspberrypi ~/Developer/UART $ lsusb -t
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=dwc_otg/1p, 480M
    |__ Port 1: Dev 2, If 0, Class=hub, Driver=hub/5p, 480M
        |__ Port 1: Dev 3, If 0, Class=vend., Driver=smsc95xx, 480M
        |__ Port 2: Dev 4, If 0, Class=vend., Driver=ftdi_sio, 12M
        |__ Port 4: Dev 5, If 0, Class=vend., Driver=ch341, 12M
```

To view the USB serial devices as they appear in the file system we can enter:

```
pi@raspberrypi ~/Developer $ ls -l /dev/ttyUSB*
crw-rw–T 1 root dialout 188, 0 Feb 27 15:13 /dev/ttyUSB0
crw-rw–T 1 root dialout 188, 1 Feb 27 15:35 /dev/ttyUSB1
```

Of course, your particular USB-to-Serial adapter may differ from those described above, so you must check the specs for your adapter before wiring circuitry. That’s the time to start off our first demo project.

Project 1: Switching Arduino digital outputs

This project illustrates how to drive the Arduino digital output ON/OFF from the Python application running on the Raspberry Pi.

The circuit diagram for our project is shown in **Fig.4.**

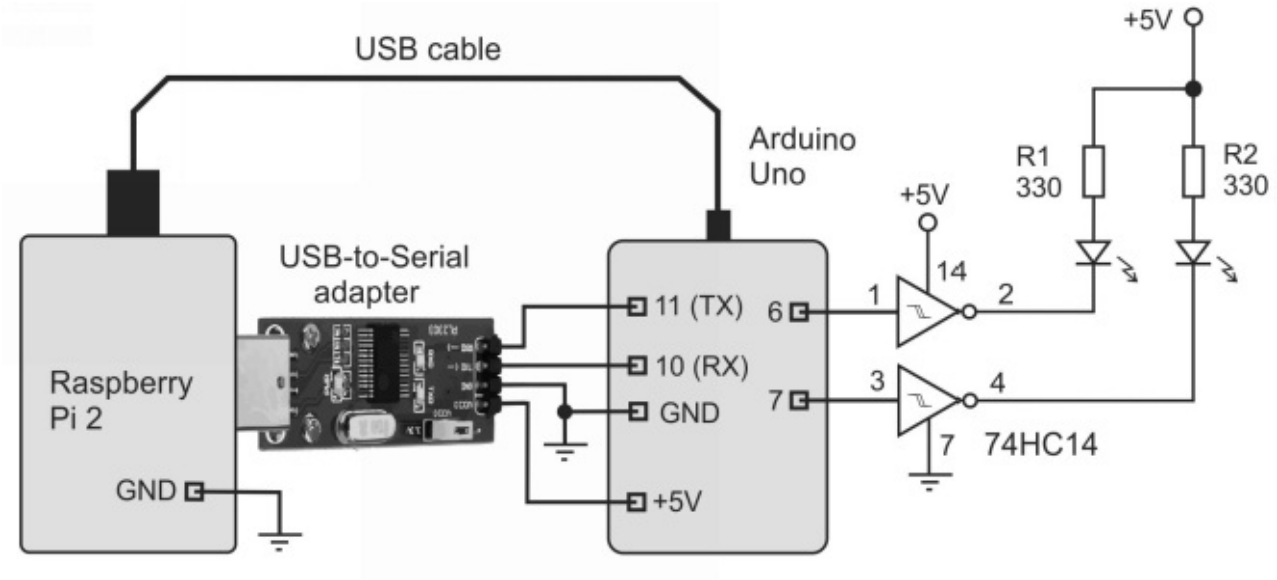


Fig.4

In this circuit, two LEDs are attached to digital pins 6 and 7 of Arduino Uno via the buffer 74HC14. The Arduino application receives the command from the Raspberry Pi 2 through the serial interface, parses the command and drives LEDs ON/OFF accordingly.

The source code of the Python application running on Raspbian OS is shown in **Listing 1.**

Listing 1.

```
import serial
import time
import argparse

parser = argparse.ArgumentParser(description='Driving output pins')
parser.add_argument('-c', '—command', help='Command', required=True)
parser.add_argument('-p', '—port', help='Port to be accessed', required=True)
parser.add_argument('-d', '—odata', help='Data to be written to', required=False)
```

```
args = parser.parse_args()
s1 = serial.Serial(port = “/dev/ttyUSB1”, baudrate=9600, timeout=5.0)
time.sleep(1)
str1 = args.command + ‘,’ + args.port + ‘,’ + args.odata
s1.write(str1)
```

The command line of the above Python application may look like the following:

<App_Path> -c <value1> -p <value2> -d <value3>

where **App_Path** is a full path to the executable file**, value1** determines the operation to be executed, **value2** is the Arduino port to be affected and **value3** can be assigned 0 or 1. In this project, **value1** should be assigned 0 (on the Arduino side this will be interpreted as the write command).

The Python application takes all parameters and forms the comma-separated sequence like the following:

<value1>,<value2>,<value3>

This is done by the statement:

```
str1 = args.command + ‘,’ + args.port + ‘,’ + args.odata
```

The string **str1** is then transferred to Arduino through the serial interface associated with the **s1** object. One of the parameters needed for creating the **s1** is **/dev/ttyUSB1** which is associated with my USB-to-Serial adapter (your adapter may be assigned a different value!).

The command string kept in **str1** is transferred through the serial interface by the statement:

```
s1.write(str1)
```

The Arduino application is represented by the following source code (**Listing 2**).

Listing 2.

```
/*
This application reads a serial ASCII-encoded string,
then parses the string using the Serial parseInt() function.
This function looks for an ASCII string of comma-separated values.
After the command has been decoded, the appropriate LED is affected.
*/
#include <SoftwareSerial.h>

SoftwareSerial mySerial(10, 11); // RX—>10, TX—>11

void setup() {
  // initialize serial:
  mySerial.begin(9600);
  // make the pins outputs:
  pinMode(6, OUTPUT);
  pinMode(7, OUTPUT);
  }

void loop() {
  // if there's any serial available, read it:
  while (mySerial.available() > 0)
  {

    // look for the next valid integer in the incoming serial stream:
    int cmd = mySerial.parseInt();
    if (cmd == 0) // 0 = write command
    {
    int port = mySerial.parseInt();
```

```
    int pVal = mySerial.parseInt();
    digitalWrite(port, pVal);
    }
  }
}
```

This application uses the **SoftwareSerial** library to configure the serial interface with the RX line assigned to pin 10 of Arduino Uno and TX line assigned to pin 11. That is done by the following statement:

```
SoftwareSerial mySerial(10, 11);
```

Both LEDS are driven by the digital pins 6 and 7 configured as outputs within the **setup()** function:

```
pinMode(6, OUTPUT);
pinMode(7, OUTPUT);
```

After running the **setup()** function, the program code enters the endless loop where the data from the serial interface will be processed. The availability of data received is checked by the **while()** loop:

```
while (mySerial.available() > 0)
```

After receiving the byte stream, the program code parses the byte sequence using the following statement:

```
int cmd = mySerial.parseInt();
```

Here the **mySerial.parseInt()** function parses the byte stream, retrieves the command (the first byte) and saves the result in the **cmd** variable.

Note that **parseInt()** returns the first valid (long) integer number from the serial buffer. Characters that are not integers (or the minus sign) are skipped. The function **parseInt()** is terminated by the first character that is not a digit.

The number of the Arduino digital pin being affected is assigned to the **port** variable:

int port = mySerial.parseInt();

Then the program code retrieves the value (0 or 1) to be written to the selected port:

int pVal = mySerial.parseInt();

This value is assigned to the **pVal** variable.

Finally, the statement

digitalWrite(port, pVal);

writes the data to the selected port.

The use of the Python application is illustrated below (the program code is saved in IOWrite_UART.py file).

pi@raspberrypi ~/Developer $ python IOWrite_UART.py -c 0 -p 7 -d 1

The above command writes 1 to the digital port 7 causing pin 7 of Arduino to go high.

pi@raspberrypi ~/Developer $ python IOWrite_UART.py -c 0 -p 6 -d 1

The above command writes 1 to the digital port 6 causing pin 6 of Arduino to go high.

pi@raspberrypi ~/Developer $ python IOWrite_UART.py -c 0 -p 13 -d 0

The above command writes 0 to the digital port 13 causing pin 13 of Arduino to go low.

pi@raspberrypi ~/Developer $ python IOWrite_UART.py -c 0 -p 7 -d 0

The above command writes 0 to the digital port 7 causing pin 7 of Arduino to go low.

Note that in this project the USB-to-UART adapter corresponds to the **/dev/ttyUSB1** device, while the Arduino Uno is controlled through the **/dev/ttyUSB0** device. **Don’t**

forget to check these assignments in your system and correct the source code if needed.

Project 2: Reading Arduino digital inputs

This project illustrates reading Arduino digital inputs via a serial interface. The control application running in the Raspberry Pi transfers a command to Arduino Uno and receives a result.

The circuit diagram for this project is shown in **Fig.5**.

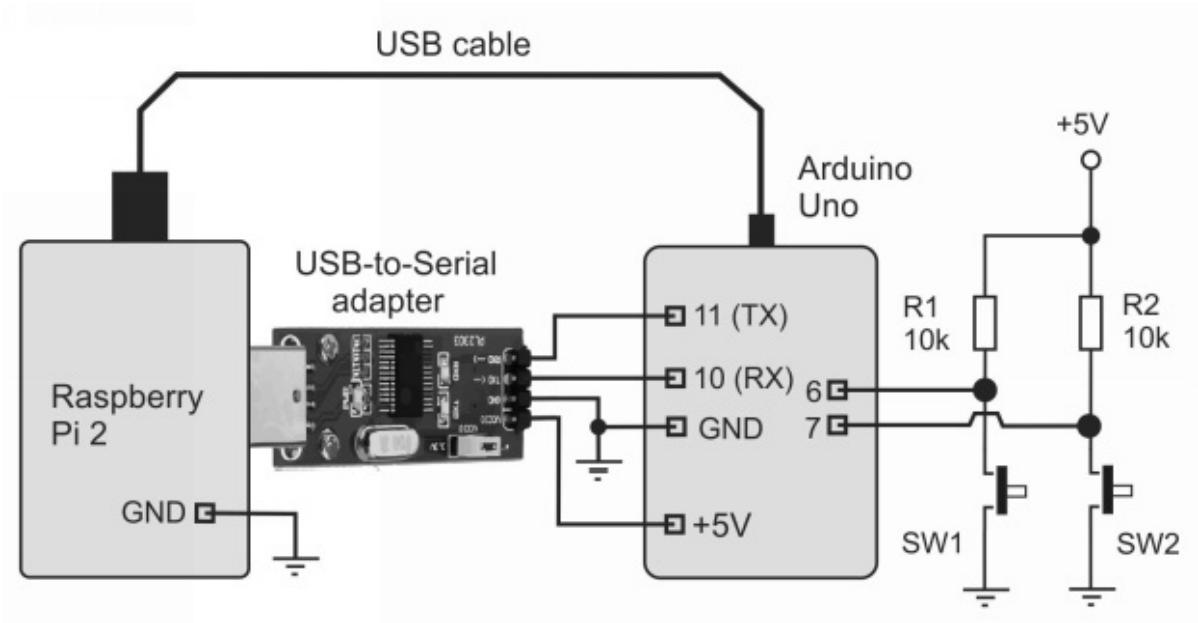


Fig.5

In this circuit, digital sensors are simulated by the switches SW1-SW2. These switches are attached to pins 6 and 7 of Arduino configured as inputs. The state of inputs is read by the Arduino application and then passed to the Raspberry Pi through the serial interface.

The Python application running on Raspbian OS is represented by the source code in **Listing 3**.

Listing 3.

```
# UART_ReadPort.py

# This script illustrates reading Arduino Uno digital pins
# through a serial interface. The command line
```

```
# arguments are as follows:
# -c determines the direction (1 = input)
# -p is the pin number to read

import serial
import time
import argparse

parser = argparse.ArgumentParser(description='Reading digital inputs')
parser.add_argument('-c', '—command', help='Command', required=True)
parser.add_argument('-p', '—port', help='Port to be accessed', required=True)
args = parser.parse_args()
str1 = args.command + ',' + args.port

s1 = serial.Serial(port = "/dev/ttyUSB0", baudrate=9600, timeout=5.0)
time.sleep(1)
s1.write(str1)
time.sleep(2)
readVal = s1.readline()
print "Pin " + args.port + " is " + readVal
```

The Python application takes two parameters and forms a comma-separated sequence like the following:

<value1>,<value2>

This is done by the statement:

```
str1 = args.command + ',' + args.port
```

The command string **str1** is then transferred to Arduino through the serial interface associated with the **s1** object:

s1.write(str1)

After some delay (2 s, in our case) the program code reads the serial interface:

readVal = s1.readline()

The **readVal** variable is assigned the value (0 or 1) depending on the state of the corresponding Arduino input.

The Arduino application has the following source code (**Listing 4)**.

Listing 4.

```
#include <SoftwareSerial.h>

SoftwareSerial mySerial(10, 11); // RX—>10, TX—>11

int cmd;
int port;
int pVal;

void setup() {
  // initialize serial port
  mySerial.begin(9600);

  pinMode(6, INPUT);
  pinMode(7, INPUT);
  }

void loop() {
  // if there’s any data available, read it:
```

```
  while (mySerial.available() > 0)
  {
    // look for the next valid integer in the incoming serial stream:
    cmd = mySerial.parseInt();
    if (cmd == 1) // 1 = read command
    {
    port = mySerial.parseInt();
    pVal = digitalRead(port);
    if (pVal == 1)
    mySerial.println("HIGH");
    else
    mySerial.println("LOW");
    }
  }
}
```

The Arduino application accomplishes the following tasks:

- receives the command string through the serial interface;
- parses the command just received;
- reads the digital input pin indicated by the command;
- sends the result back to the Raspberry Pi through the serial interface.

The command to read digital input has the code 1. That is checked by the **if ()** statement:

```
if (cmd == 1)
{
…
}
```

The statement

```
port = mySerial.parseInt();
```

parses the port to be read. This is followed by

pVal = digitalRead(port);

which stores the state of the selected digital input in the variable **pVal**.

The running application (file IORead_UART.py) produces the following output depending on the state of the **SW1 – SW2** switches:

pi@raspberrypi ~/Developer $ python IORead_UART.py -c 1 -p 6

Pin 6 is LOW

pi@raspberrypi ~/Developer $ python IORead_UART.py -c 1 -p 7

Pin 7 is LOW

pi@raspberrypi ~/Developer $ python IORead_UART.py -c 1 -p 7

Pin 7 is HIGH

Project 3: Reading Arduino analog inputs

This project illustrates reading the analog input voltage on pin A0 of Arduino Uno. The results of measurements are then passed back to the Raspberry Pi. The analog voltage on A0 is taken from the wiper of the potentiometer of 10k.

The circuit diagram of the project is shown in **Fig.6**.

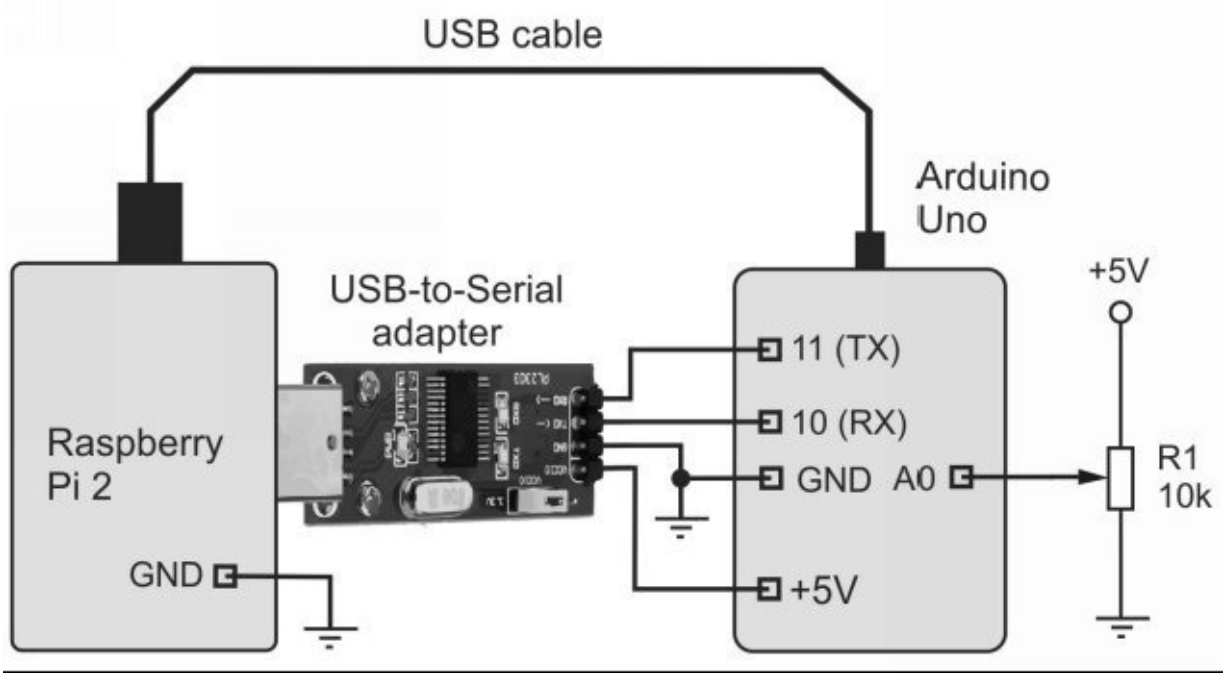


Fig.6

The serial interface connections between the Raspberry Pi 2 and Arduino Uno are the same as those in the previous projects.

The Arduino Uno source code is shown in **Listing 5**.

Listing 5.

```
#include <SoftwareSerial.h>

SoftwareSerial mySerial(10, 11); // RX—>10, TX—>11

int cmd; // command -> -c 2, where 2 means reading analog
```

```
int ch0 = A0;    // A0 is taken by default
int binVal = 0;  // binary code of analog is stored here
float Vref = 5.0; // Reference voltage to the A/D converter
float res;

void setup() {
    mySerial.begin(9600);
}

void loop() {

  while (mySerial.available() > 0)
  {

    // look for the next valid integer in the incoming serial stream:

    cmd = mySerial.parseInt();
    if (cmd == 2) // 2 = read analog input
    {
    binVal = analogRead(ch0);
    res = (Vref / 1024.0) * (float)binVal;
    mySerial.print(res);
    }
  }
}
```

The Arduino application receives the command string through the serial interface, reads the analog voltage level on pin A0 and sends the result in a float format back to the Raspberry Pi. The command should contain only a single parameter equal 2.

This condition is checked by the **if ()** statement:

```
if (cmd == 2)
```

```
{
…
}
```

The binary code corresponding to the analog voltage is read into the **binVal** variable by

```
binVal = analogRead(ch0);
```

Then the program code converts the binary data in **binVal** into the float value (variable **res**) and sends the data to the Raspberry Pi:

```
res = (Vref / 1024.0) * (float)binVal;
mySerial.print(res);
```

The Python source code is given in **Listing 6**.

Listing 6.

```
import serial
import time
import argparse
parser = argparse.ArgumentParser(description='Reading analog input')
parser.add_argument('-c', '—command', help='Command', required=True)
args = parser.parse_args()
str1 = args.command
s = serial.Serial(port = "/dev/ttyUSB0", baudrate=9600, timeout=5.0)
time.sleep(1)
s.write(str1)
time.sleep(2)
readVal = s.read(size=32)
print "Analog voltage on channel A0: " + readVal + "V"
```

The running application (file ARead_UART.py) produces the following output:

```
pi@raspberrypi ~/Developer $ python ARead_UART.py -c 2
Analog voltage on channel A0: 1.88V
```

```
pi@raspberrypi ~/Developer $ python ARead_UART.py -c 2
Analog voltage on channel A0: 2.02V
pi@raspberrypi ~/Developer $ python ARead_UART.py -c 2
Analog voltage on channel A0: 1.01V
pi@raspberrypi ~/Developer $ python ARead_UART.py -c 2
Analog voltage on channel A0: 0.76V
```

Interfacing Teensy 3.1 with Raspberry Pi

A Teensy 3.1 is a small but a powerful development board based upon the Cortex M4 MCU. This board has more computation power than a classic Arduino board. We can easily interface this board to the Raspberry Pi to design various measurement systems. The Teensy 3.1 can previously be programmed in either Windows or Linux x 32/x 64 OS. In our projects, Teensy 3.1 will be connected to a USB port of the Raspberry Pi 2.

The Teensy 3.1 board itself is shown in **Fig**.**7**.

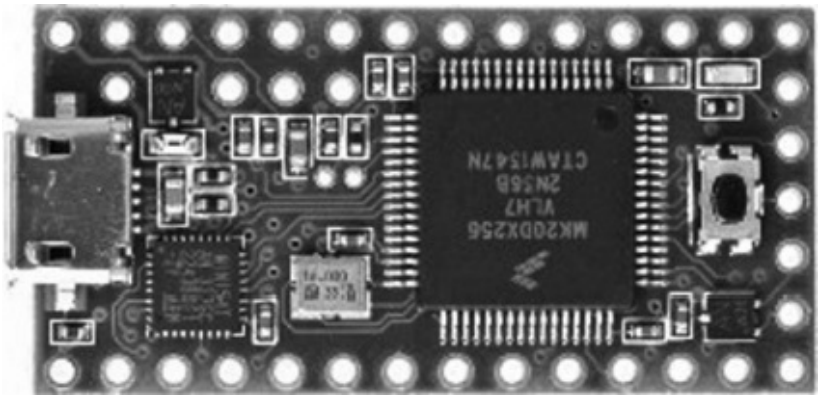


Fig.7

The pinout of the Teensy 3.1 is shown in **Fig**.**8**.

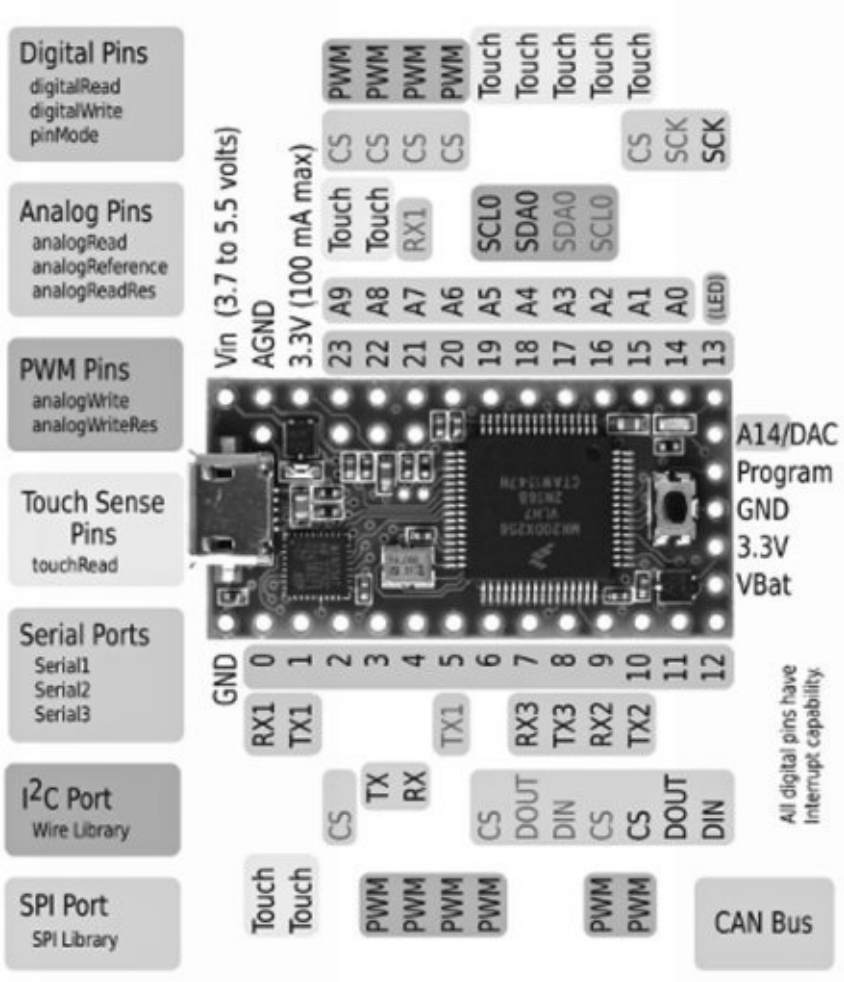


Fig.8

Before writing control applications, we first need to check if the Teensy 3.1 USB interface has been properly configured on the Raspbian OS. This is done by entering the **lsusb** commands:

pi@raspberrypi ~ $ lsusb

Bus 001 Device 002: ID 0424:9514 Standard Microsystems Corp.

Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub

Bus 001 Device 003: ID 0424:ec00 Standard Microsystems Corp.

Bus 001 Device 004: ID 16c0:0483 VOTI Teensyduino Serial

pi@raspberrypi ~ $ lsusb -t

/: Bus 01.Port 1: Dev 1, Class=root_hub, Driver=dwc_otg/1p, 480M

    |__ Port 1: Dev 2, If 0, Class=hub, Driver=hub/5p, 480M

    |__ Port 1: Dev 3, If 0, Class=vend., Driver=smsc95xx, 480M

    |__ Port 4: Dev 4, If 0, Class=comm., Driver=cdc_acm, 12M

|__ Port 4: Dev 4, If 1, Class=data, Driver=cdc_acm, 12M

Our control application will transfer command and data through the virtual serial interface, so we have to determine the device associated with the Teensy3.1 serial interface:

pi@raspberrypi ~ $ ls -l /dev/ttyA*

crw-rw–T 1 root dialout 166, 0 Mar 18 18:33 /dev/ttyACM0

crw-rw–- 1 root tty   204, 64 Mar 17 22:35 /dev/ttyAMA0

In Linux systems, the custom USB-to-Serial interface is usually assigned the device that appears as **/dev/ttyACMx**. In our case, the **/dev/ttyACM0** relates to Teensy 3.1.

The Teensy 3.1 features a digital-to-analog converter (D/A converter) which allows to generate various shape forms of signals. The following two projects describe using this D/A converter controlled by the Raspberry Pi 2.

Project 4: Designing a programmable DC voltage source with Teensy 3.1

This project illustrates the development of a programmable DC voltage source based upon the digital-to-analog converter (DAC) of the Teensy 3.1 board. Such a DC source will be controlled through the serial interface by the commands arriving from the Raspberry Pi.

The hardware configuration of our system is shown in **Fig.9**.

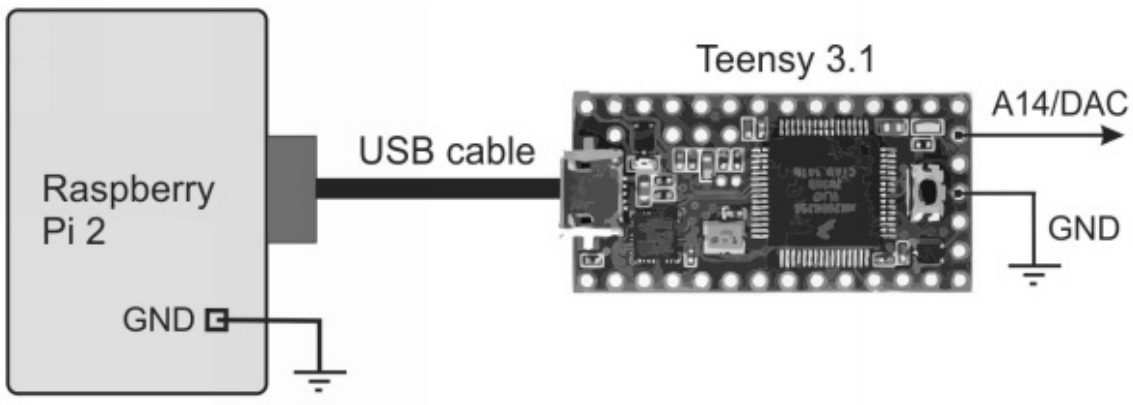


Fig.9

In this configuration, Teensy 3.1 is connected to the Raspberry Pi 2 through the USB cable, so the power and signals to Teensy 3.1 are fed from the USB port of the Raspberry Pi. The output DC signal is taken from the A14/DAC output of the Teensy 3.1 board.

The source code of the Teensy 3.1 application is shown in **Listing 7**.

Listing 7.

```
const float vref = 3.35; // ref.voltage is about 3.35V
const float res = 4096.0; // 12-bit DAc is used

void setup()
{
  Serial.begin(9600);
  analogWriteResolution(12);
}

void loop()
{
while (Serial.available() > 0)
{
  // look for the next valid integer in the incoming serial stream:
  int cmd = Serial.parseInt();
  if (cmd == 3) // 3 = write DAC command
  {
    float dacVal = Serial.parseFloat();
    int pVal = (int)(dacVal*res/vref);
    analogWrite(A14, pVal);
  }
}
}
```

The program code receives the command for writing the DAC (code = 3) and parses the value of the DAC output to be set. The float representation of the DAC output voltage is saved in the **dacVal** variable. The program code converts the value in **dacVal** into the corresponding binary code (variable **pVal**). Finally, this binary code is loaded into the

DAC by the command:

analogWrite(A14, pVal);

The source code of the Python application running on Raspbian OS is shown in **Listing 8**.

Listing 8.

```
#!/usr/bin/python
# UART_DAC.py
# This script illustrates configuring the DAC output on Teensy 3.1
# via a serial interface. The arguments are as follows:
# -c determines the type (3 = to write DAC)
# -d data (a float value) to be written to the Teensy DAC

import serial
import time
import argparse

parser = argparse.ArgumentParser(description='Demo parser script')
parser.add_argument('-c', '—command', help='Command', required=True)
parser.add_argument('-d', '—odata', help='Data to be written to', required=True)
args = parser.parse_args()

s1 = serial.Serial("/dev/ttyACM0", baudrate=9600, timeout=5.0)
time.sleep(1)
str1 = args.command + ',' + args.odata
s1.write(str1)
```

The Python application takes two parameters and forms a comma-separated sequence like the following:

<value1>,<value2>

This is done by the statement:

str1 = args.command + ‘,’ + args.odata

This command string (variable **str1**) is then transferred to Arduino through the serial interface associated with the **s1** object:

s1.write(str1)

Save this source code in the file UART_DAC.py and make this file executable by entering **chmod +x** command.

The command lines for setting various DAC outputs are shown below.

pi@raspberrypi ~/Developer $ ./UART_DAC.py -c 3 -d 1.23
pi@raspberrypi ~/Developer $ ./UART_DAC.py -c 3 -d 2.59
pi@raspberrypi ~/Developer $ ./UART_DAC.py -c 3 -d 0.82
pi@raspberrypi ~/Developer $ ./UART_DAC.py -c 3 -d 2.12
pi@raspberrypi ~/Developer $ ./UART_DAC.py -c 3 -d 1.99
pi@raspberrypi ~/Developer $ ./UART_DAC.py -c 3 -d 1.06
pi@raspberrypi ~/Developer $ ./UART_DAC.py -c 3 -d 0.77

Project 5: Generating sine waves with Teensy 3.1

The project describes developing the programmable sinewave oscillator using the Teensy 3.1 D/A converter. The hardware connections are shown in **Fig.10**.

In this configuration, the DAC output is driven by the commands received through the serial interface. The Python application running on the Raspberry Pi 2 determines the frequency of the sinewave signal. The Buffer/Amplifier block (optional) allows to isolate the DAC output from load.

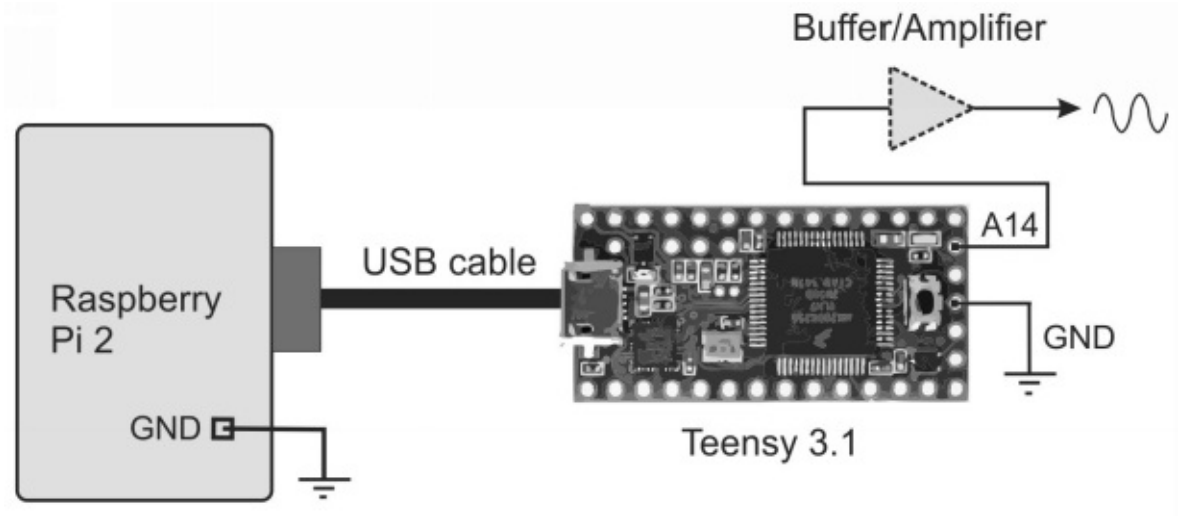


Fig.10

The Python source code for the Raspberry Pi is shown in **Listing 9**.

Listing 9.

```
#!/usr/bin/python
# UART_SineWave.py

# This script allows to adjust a sinewave signal on the DAC output on Teensy 3.1
# via a serial interface. The arguments are as follows:
# -c determines the type (4, in our case)
# -d determines the frequency (in Hz) of the DAC output signal

import serial
import time
import argparse

parser = argparse.ArgumentParser(description='Demo parser script')
parser.add_argument('-c', '—command', help='Command', required=True)
parser.add_argument('-d', '—odata', help='Data to be written to', required=True)
args = parser.parse_args()
```

```
port = serial.Serial(“/dev/ttyACM0”, baudrate=9600, timeout=5.0)
time.sleep(1)
str1 = args.command + ‘,’ + args.odata
port.write(str1)
```

The Teensy 3.1 application has the following source code (**Listing 10**).

Listing 10.

```
#include <TimerOne.h>
#define K 3906 //1000000/256
volatile int freq = 300; // an initial frequency is set to 300 Hz
volatile int tuS = K/freq;
const int NUM_SAMPLES = 256;
unsigned char SINE_TABLE[NUM_SAMPLES];
volatile int cnt = 0;
void setup()
{
Serial.begin(9600);
fillBuf();
analogWriteResolution(8); // 8-bit write resolution is set
Timer1.initialize(tuS); // interval in uS
Timer1.attachInterrupt(writeDAC);  // ISR to call when an interval expires
Timer1.start();
}
void loop()
{
while (Serial.available() > 0)
{
  // look for the next valid integer in the incoming serial stream:
  int cmd = Serial.parseInt();
  if (cmd == 4) // 4 = set DAC frequency
  {
   freq = Serial.parseInt();
   if (freq <= 500) // max frequency may be 500 Hz
   {
    tuS = K/freq;
    noInterrupts();
    Timer1.stop();
```

```
    Timer1.initialize(tuS);
    Timer1.start();
    interrupts();
   }
  }
}
}

void fillBuf(void)
{
//sine period is 2*PI
const float step = (2*M_PI)/(float)NUM_SAMPLES;
float s;
// calculations are taken in radians

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

void writeDAC(void)
{
  if(cnt == 256)cnt = 0;
  analogWrite(A14, SINE_TABLE[cnt]); // write DAC
  cnt++;
}
```

In this source code, the **fillBuf()** function forms a lookup table (LUT) associated with the **SINE_TABLE** array. Its size ($2^8 = 256$) is determined by the resolution of the DAC output signal. Each value from this table is written to the DAC by the **writeDAC()** interrupt service routine which is called when the **Timer1** interrupt occurs.

The frequency of the sinewave output signal is determined by the command received through the serial interface from the Raspberry Pi.

The **if()** structure

```
if (cmd == 4)

{

…

}
```

checks if the command code = 4. If true, the interval (in uS) for **Timer1** will be set by the statement:

tuS = K/freq;

Here the variable **tuS** is assigned the value proportional to the desired frequency (variable **freq**). When a new value of **tuS** is set, the **Timer1** will be restarted. This application allows to set the DAC output frequency <= 500 Hz. The higher values are also possible, but precision may drop down as a frequency increases.

The command line for application running on Raspbian OS is shown below (assume that the source code is saved in UART_SineWave.py):

pi@raspberrypi ~/Developer $ ./UART_SineWave.py -c 4 -d 250

pi@raspberrypi ~/Developer $ ./UART_SineWave.py -c 4 -d 200

pi@raspberrypi ~/Developer $ ./UART_SineWave.py -c 4 -d 150

Project 6: Driving Arduino digital outputs over a TCP/IP network

This project illustrates how to control the Arduino digital outputs through a TCP/IP network. The project uses the configuration shown below (**Fig.11**).

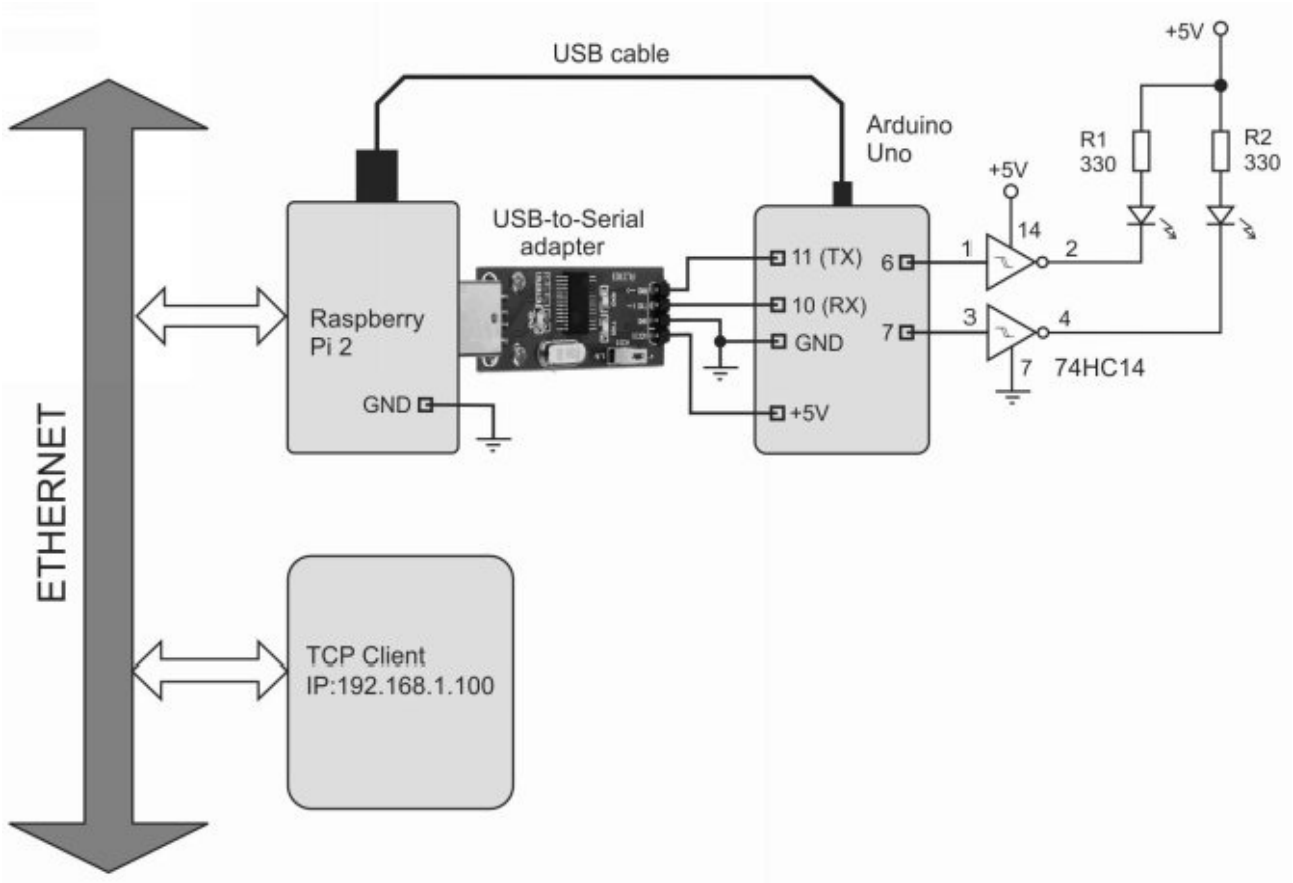


Fig.11

In this configuration, a TCP server application running in the Raspberry Pi 2 receives the commands from a TCP client application running anywhere on a TCP/IP network. The command just received is then redirected to Arduino through the serial interface.

Depending on the command received, Arduino drives appropriate LED ON/OFF (see **Fig.11**).

The Arduino Uno source code written in the Processing language is shown in **Listing 11**.

Listing 11.

```
#include <SoftwareSerial.h>

SoftwareSerial mySerial(10, 11); // RX—>10, TX—>11

void setup() {
  // initialize serial:
  mySerial.begin(9600);
  // make the pins outputs:
  pinMode(13, OUTPUT);  // there the LED is connected
  pinMode(6, OUTPUT);
  pinMode(7, OUTPUT);
  }

void loop() {
  // if there's any serial available, read it:
  while (mySerial.available() > 0)
  {

    // look for the next valid integer in the incoming serial stream:
    int cmd = mySerial.parseInt();
    if (cmd == 0) // 0 = write to a digital port
    {
    int port = mySerial.parseInt();
    int pVal = mySerial.parseInt();
    digitalWrite(port, pVal);
    }
  }
}
```

This source code parses the comma-separated sequence received from the serial port of the Arduino. The first value (command code) should be 0, the second indicates a digital port to be affected and the third value determines the value (0 or 1) to be written to the port.

The Raspberry Pi TCP server is represented by the following source code (**Listing 12)**.

Listing 12.

```
# TCPServer_UART.py
# This script receives the command string over a network and simply
# redirects the command to Arduino Uno.

import serial
import time
import socket               # import the socket module

uart1 = serial.Serial("/dev/ttyUSB1", baudrate=9600, timeout=5.0)
time.sleep(1)
srv = socket.socket()       # Create a socket object
host = socket.gethostname() # Get local machine name
port = 8089                 # Assign a port to receive incoming requests
addr = (host, port)
srv.bind(addr)
srv.listen(5)
print "TCP Server is waiting for incoming requests on port 8089…"

try:
    while True:
    client, caddr = srv.accept()
    cmd = client.recv(256)
    print "Received: " + cmd
    uart1.write(cmd)
    client.close()
except KeyboardInterrupt:
```

```
    print “TCP Server is shutting down…”
    srv.close()
```

To transfer data over the serial interface we should create the serial port object **uart1**:

```
uart1 = serial.Serial(“/dev/ttyUSB1”, baudrate=9600, timeout=5.0)
```

Note that your serial interface may be assigned a different device!

The sequence that follows initializes and starts the TCP server:

```
srv = socket.socket()            # Create a listening socket object srv
host = socket.gethostname() # Get a local machine name
port = 8089                      # Assign a port to receive incoming requests
addr = (host, port)              # Form an address structure
srv.bind(addr)                   # bind the address to the listening socket srv
srv.listen(5)
```

Our server uses port 8089 (you can select other suitable value, of course). When started, the TCP server enters the endless **while** loop and waits for incoming requests from TCP clients anywhere on a TCP/IP network.

When a request is accepted, the client (working) connection is established and a working socket is created by the statement:

```
client, caddr = srv.accept()
```

The command string sent by a TCP client is then received and stored in the **cmd** variable:

```
cmd = client.recv(256)
```

This command string is redirected to Arduino through the serial interface by the statement:

```
uart1.write(cmd)
```

When done, the client connection is closed and the **while** loop repeats:

client.close()

To terminate the TCP server application we should press Ctrl+C. We also need to close the server socket srv:

srv.close()

A TCP Client forms the comma-separated sequence to be delivered to the TCP Server running in the Raspberry Pi. The TCP client application is represented by the following source code (**Listing 13**).

Listing 13.

```
#TCPClient_UART.py

import socket
import argparse

parser = argparse.ArgumentParser(description='Demo parser script')
parser.add_argument('-c', '—command', help='Command', required=True)
parser.add_argument('-p', '—port', help='Port to be accessed', required=True)
parser.add_argument('-d', '—odata', help='Data to be written to', required=False)
args = parser.parse_args()
cmd = args.command + ',' + args.port + ',' + args.odata

port = 8089
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
```

```
s.connect((host, port))
s.send(cmd)
s.close()
```

In this source code, the comma-separated sequence is formed by the following statement:

```
cmd = args.command + ‘,’ + args.port + ‘,’ + args.odata
```

Then the program code creates the client socket and establishes the connection to the TCP server using the following fragment of code:

```
port = 8089
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
s.connect((host, port))
```

Note that the port associated with the client socket should be the same as that associated with the listening socket in the TCP server application.

Then the command string is sent to the TCP server and the client socket is closed:

```
s.send(cmd)
s.close()
```

Note that for the sake of brevity we don’t perform error checking, although you should do that in you code.

The output below shows the command lines for TCP client application (the TCP Server must be running before the TCP client is launched).

```
$ python TCPClient_UART.py -c 0 -p 13 -d 1
$ python TCPClient_UART.py -c 0 -p 13 -d 1
$ python TCPClient_UART.py -c 0 -p 6 -d 1
```

$ python TCPClient_UART.py -c 0 -p 7 -d 1

$ python TCPClient_UART.py -c 0 -p 13 -d 0

$ python TCPClient_UART.py -c 0 -p 6 -d 0

$ python TCPClient_UART.py -c 0 -p 7 -d 0

The running TCP Server application produces the following output:

pi@raspberrypi ~/Developer/UART $ python TCPServer_UART.py

TCP Server is waiting for incoming requests on port 8089…

The command received: 0,7,1

The command received: 0,6,1

The command received: 0,13,1

The command received: 0,6,0

The command received: 0,7,0

The command received: 0,13,0

The command received: 0,13,1

In this project, the TCP Server and TCP client were launched on the same Raspberry Pi board. Usually, network applications run on different hosts. For instance, the source code for the TCP Server running on the host with an IP-address **192.168.1.101** will look like the following (L**isting 14)**.

Listing 14.

```
import serial
import time
import socket               # import the socket module

uart1 = serial.Serial("/dev/ttyUSB1", baudrate=9600, timeout=5.0)
time.sleep(1)

srv = socket.socket()       # Create a socket object
port = 8089                  # Select a port to receive incoming requests
addr = ("192.168.1.101", port)
srv.bind(addr)
```

```
srv.listen(5)
print “TCP Server is waiting for incoming requests on port 8089…”

try:
    while True:
        client, caddr = srv.accept()
        cmd = client.recv(256)
        print “The command received: ” + cmd
        uart1.write(cmd)
        client.close()
except KeyboardInterrupt:
        print “TCP Server is shutting down…”
        srv.close()
```

The corresponding TCP Client source code may appear like the following (**Listing 15)**.

Listing 15.

```
#TCPClient_UART.py

import socket
import argparse

parser = argparse.ArgumentParser(description=‘Demo parser script’)
parser.add_argument(‘-c’, ‘—command’, help=‘Command’, required=True) # 0==output
parser.add_argument(‘-p’, ‘—port’, help=‘Port to be accessed’, required=True)
parser.add_argument(‘-d’, ‘—odata’, help=‘Data to be written to’, required=False)
args = parser.parse_args()
cmd = args.command + ‘,’ + args.port + ‘,’ + args.odata

port = 8089
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((“192.168.1.101”, port))
s.send(cmd)
s.close()
```

In this code, the IP-address (192.168.1.101) of the TCP server is explicitly passed to the **connect()** function.

Project 7: Reading digital inputs over a TCP/IP network

This project illustrates reading Arduino digital inputs over a TCP/IP network. In this configuration, the Arduino Uno application reads digital inputs each time a TCP server sends a command through a serial port. The data is then taken by a TCP Client located anywhere on the TCP/IP network.

The circuit diagram of the hardware is shown in **Fig.12**.

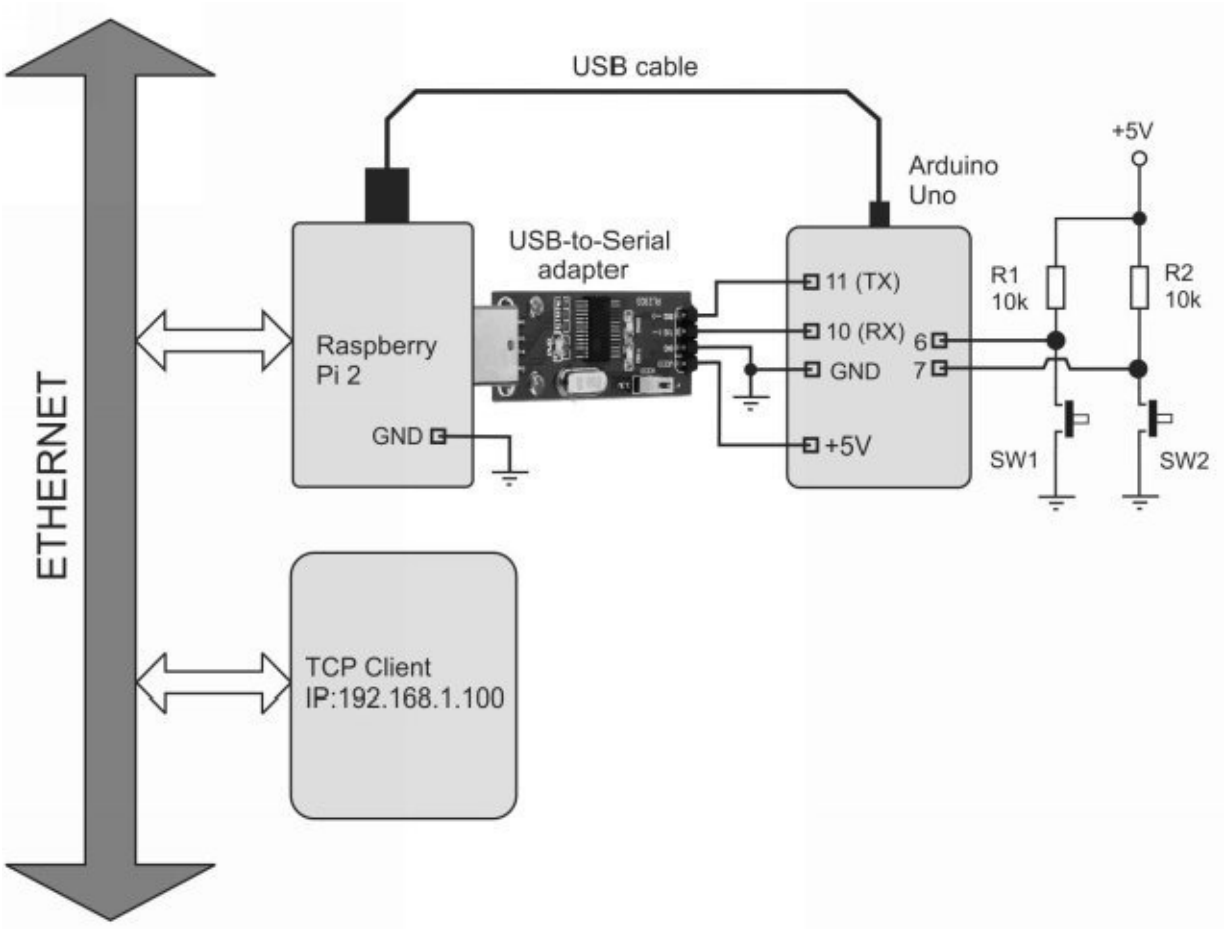


Fig.12

In this configuration, digital input sensors are simulated by the switches SW1-SW2. The input signals are taken from the Arduino pins 6-7 configured as inputs. The Arduino Uno source code is shown in **Listing 16**.

Listing 16.

```
/*  UART_ReadPin.py*/
```

```
#include <SoftwareSerial.h>

SoftwareSerial mySerial(10, 11); // RX—>10, TX—>11

int cmd;
int port;
int pVal;

void setup() {
  // initialize serial:
  mySerial.begin(9600);
    // make the pins outputs:

  pinMode(6, INPUT);
  pinMode(7, INPUT);
  }

void loop() {
  // if there’s any serial data available, read it:
  while (mySerial.available() > 0)
  {

    // look for the next valid integer in the incoming serial stream:

    cmd = mySerial.parseInt();
    if (cmd == 1) // 1 = read command
    {
    port = mySerial.parseInt();
    pVal = digitalRead(port);
    if (pVal == 1)
    mySerial.println(“HIGH”);
```

```
    else
    mySerial.println("LOW");
    }
  }
}
```

The above source code is easy to understand, so we will not discuss it.

The TCP Server source code is shown in **Listing 17**.

Listing 17.

```
import serial
import time
import socket              # import the socket module

uart0 = serial.Serial("/dev/ttyUSB0", baudrate=9600, timeout=5.0)
time.sleep(1)

srv = socket.socket()            # Create a socket object
host = socket.gethostname() # Get a local machine name
port = 8089                        # Assign a port to receive incoming requests
addr = (host, port)
srv.bind(addr)
srv.listen(5)
print "A TCP Server is waiting for incoming requests on port 8089…"

try:
   while True:
       client, caddr = srv.accept()
       cmd = client.recv(256)
       print "The command string to read data:" + cmd
       uart0.write(cmd)
       time.sleep(2)
       readVal = uart0.readline()
       client.send(readVal)
       client.close()
except KeyboardInterrupt:
       print "TCP Server is shutting down…"
       srv.close()
```

The TCP Client source code is shown in **Listing 18**.

Listing 18.

```
import socket
import argparse
import time

parser = argparse.ArgumentParser() #description='Demo parser script')
parser.add_argument('-c', '—command', help='Command', required=True) # 1 = digital read
parser.add_argument('-p', '—pin', help='Pin to be read', required=True)
args = parser.parse_args()
cmd = args.command + ',' + args.pin

port = 8089
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()

s.connect((host, port))
s.send(cmd)
time.sleep(3)
res = s.recv(128)
print "Pin " + args.pin + " is " + str(res)
s.close()
```

The command line to launch the TCP client appears as follows:

```
<path_to_executable> -c 1 –p <port_to_read>
```

Project 8: Reading analog inputs over a TCP/IP network

This project illustrates reading analog input signals by Arduino and passing the data through the TCP/IP network. The project uses the Raspberry Pi and Arduino Uno boards connected via a USB-to-Serial interface. Signal processing is accomplished by Arduino, then result is redirected to the TCP server running in the Raspberry Pi 2 board. The TCP server, in turn, sends the data to the TCP client located anywhere on a TCP/IP network.

For the sake of simplicity, both the TCP server and the TCP client are running on the same Raspberry Pi board. The circuit diagram of the hardware is shown in **Fig.13**.

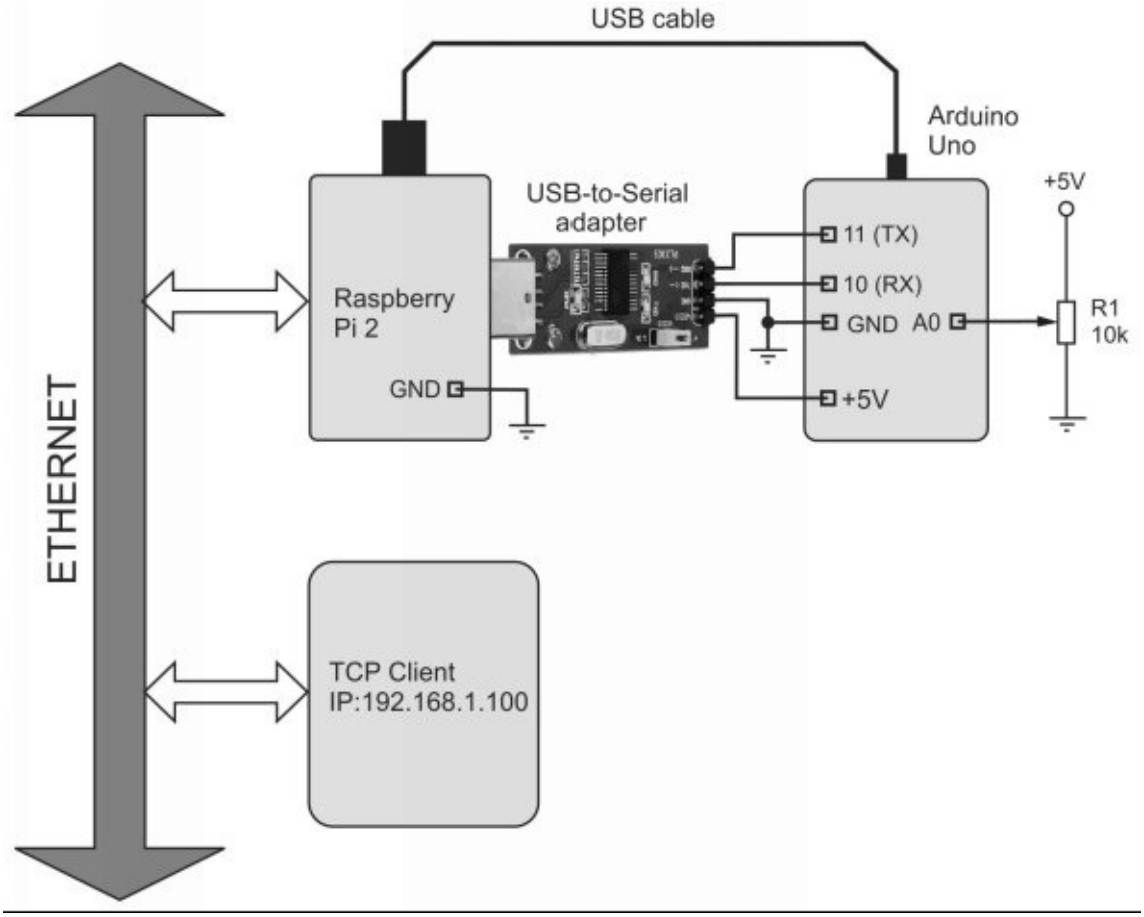


Fig.13

In this project, the analog input signal simulated by the potentiometer R1 goes to the analog channel A0 of the Arduino Uno board. To obtain the data from Arduino the TCP client sends a command to the TCP Server over a TCP/IP network; the TCP server, in turn, passes this command to the Arduino board through the serial port. After the A/D conversion is complete, the TCP server picks the result up and transfers the data back to the TCP client.

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

Listing 19.

```
/*
  UART_Analog Input
Demonstrates reading the analog sensor output on the analog pin 0
    */

#include <SoftwareSerial.h>

SoftwareSerial mySerial(10, 11); // RX—>10, TX—>11

int cmd; // command -> -c 2, where 2 means reading analog

int ch0 = A0;    // A0 is taken by default
int binVal = 0;  // binary code of analog is stored here
float Vref = 5.0; // Reference voltage to the A/D converter
float res;

void setup() {
    mySerial.begin(9600);
}

void loop() {

  while (mySerial.available() > 0)
  {

    // look for the next valid integer in the incoming serial stream:
```

```
    cmd = mySerial.parseInt();
    if (cmd == 2) // 2 = read analog input
    {
    binVal = analogRead(ch0);
    res = (Vref / 1024.0) * (float)binVal;
    mySerial.print(res);
    }
  }
}
```

The TCP Server source code written in Python is shown in **Listing 20**.

Listing 20.

```
import serial
import time
import socket               # import the socket module
uart0 = serial.Serial("/dev/ttyUSB0", baudrate=9600, timeout=5.0)
time.sleep(1)
srv = socket.socket()             # Create a socket object
host = socket.gethostname() # Get a local machine name
port = 8089               # Assign a port to receive incoming requests
addr = (host, port)
srv.bind(addr)
srv.listen(5)
print "A TCP Server is waiting for incoming requests on port 8089…"
try:
   while True:
       client, caddr = srv.accept()
       cmd = client.recv(256)
       print "The command to read an analog channel A0 has been received:" + cmd
       uart0.write(cmd)
       time.sleep(2)
       readVal = uart0.readline()
       client.send(readVal)
       client.close()
```

```
except KeyboardInterrupt:
    print “TCP Server is shutting down…”
    srv.close()
```

The TCP Client code in Python is shown in **Listing 21**.

Listing 21.

```
#TCPClient_UART_AnalogRead.py

import socket
import argparse
import time

parser = argparse.ArgumentParser(description=‘Demo parser script’)
parser.add_argument(‘-c’, ‘—command’, help=‘Command’, required=True) # 2 = analog read
args = parser.parse_args()
cmd = args.command

port = 8089
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host = socket.gethostname()
s.connect((host, port))
s.send(cmd)
time.sleep(3)
res = s.recv(128)
print “A0 input: ” + str(res) + “V”
s.close()
```

The TCP client application command line appears as follows:

```
<path_to_executable> -c 2
```

Raspberry Pi Web applications: a brief introduction

This section covers the basics of building Web-based measurement and control systems running on the Raspberry Pi 2. Before designing our first Web application, we need to become familiar with a few general concepts.

The next diagram (**Fig.14**) illustrates at the most basic level how a Web (HTML) page is brought to the screen of your PC, tablet, etc.

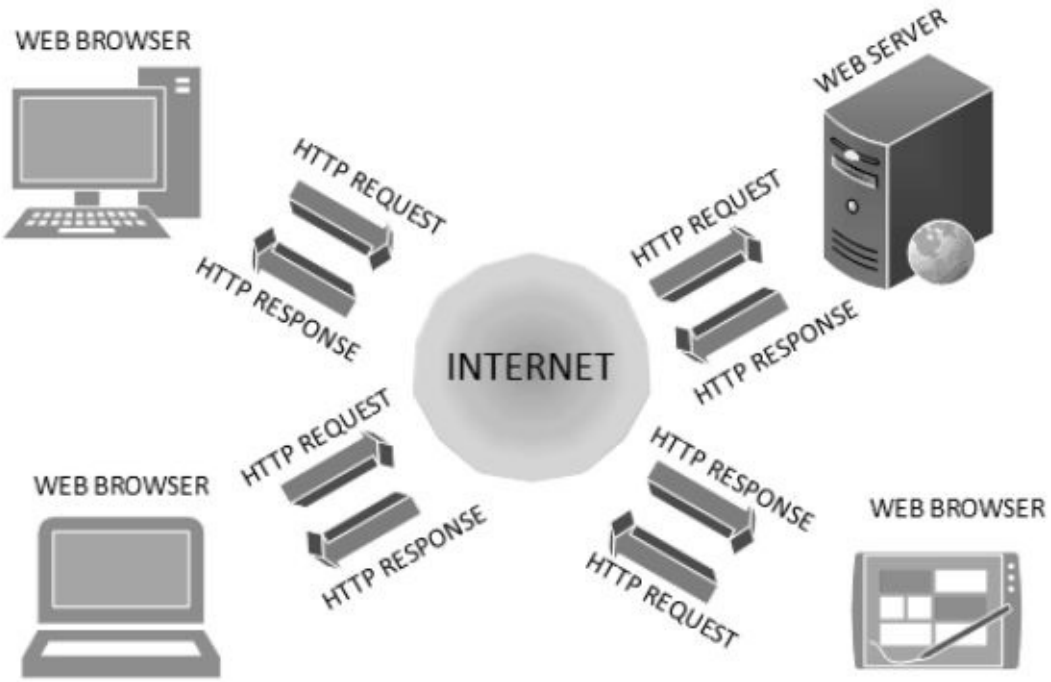


Fig.14

When you connect to a Web server from your Web browser, the following sequence is performed:

- Your **Web browser** (also called a "**Web Client"**) establishes the communication with the **Web Serve**r and demands for the URL (filename).
- **Web Server** will parse the URL and will look for the filename. If the filename has been found, the content of that file (**index.html**, for example) is sent back to the **Web browser**, otherwise the Web Server will send an error message indicating that you have requested a wrong file.
- **Web browser** takes response from the **Web Server** and displays either the content of the received file or error message.

The Web browser communicated with a name server to translate the server name, say, "www.webserver.com" into an IP-address, which is used to connect to the server machine. The browser then forms a connection to the server at that IP address on port 80.

Following the HTTP protocol, the browser sent a GET request to the server, asking for the file (Web page) with the .htm or .html extension, say, "http://www.webserver.com/page.htm."

The Web server then sends the HTML text of the Web page to the browser. The browser, in turn, reads the HTML tags and outputs the page onto your screen. This is a very simplified description of what happens when accessing a Web server.

In general, each computer on the Internet can be either a Web server or a Web client. Those machines that provide services like Web servers to other machines are servers. The machines used to connect to those services are clients. When you connect to Google at www.google.com to read a page, Google provides a machine running a Web server to service your request.

Your machine usually provides no services to anyone else on the Internet, so it is considered as a Web client. Nowadays, the same machine frequently runs both a Web server and Web client. In this guide, we will configure our Raspberry Pi to run a Web server application that will allow us to control GPIO and external circuitry over the Internet.

Raspberry Pi Web applications: installing the Apache Web server

There are various ways to develop Web applications, but we will discuss the method where the popular Web Server Apache is used.

Apache is a popular Web server application you can install on the Raspberry Pi to allow it to serve web pages. On its own, Apache can serve HTML files over HTTP, and with additional modules can serve dynamic web pages using scripting languages such as PHP, Python, Perl, etc.

First what we need is to install the apache2 package by typing the following command:

```
$ sudo apt-get install apache2 -y
```

By default, Apache puts a test HTML file in the web folder. This default web page is served when you browse to **http://localhost/** on the Pi itself, or **http://192.168.1.102** (that is my RPi IP address) from another computer on the network. To find the Pi's IP address, type

```
hostname -I
```

at the command line or simply enter the **ifconfig** command which will output a lot of network parameters including IP-addresses of available network interfaces.

Browse to the default web page either on the Pi or from another computer on the network and you should see the following (**Fig.15**).

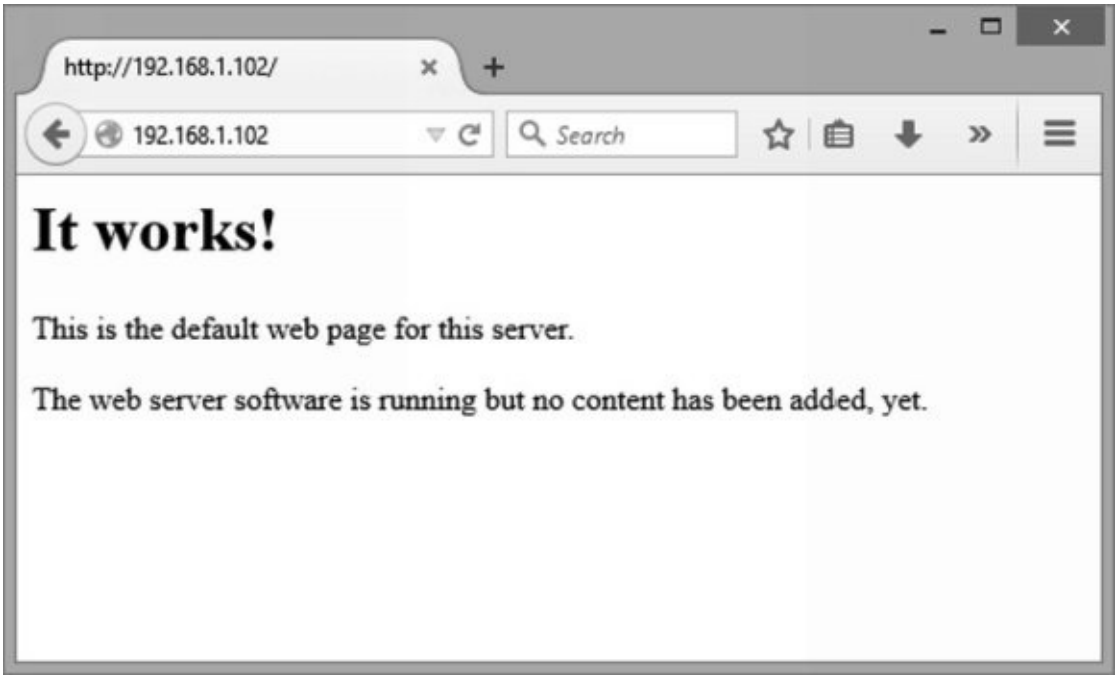


Fig.15

This means your Apache Web server operates properly.

What you see in **Fig.15** is a default web page – this is just a HTML file called **index.html** located in the directory **/var/www/**.

Initially, this directory contains nothing else but only index.html. You can see that by typing:

```
$ ls –l /var/www
total 12
-rw-r—r—  1 root root  177 Jan  8 01:29 index.html
```

By default, the **/var/ www** directory and **index.html** file are both owned by the **root** user. In order to edit this file and other HTML-pages being created, you must gain the **root** permissions. We can either change the owner to your own user by typing:

$ sudo chown pi index.html

before editing, or edit using

$ sudo nano index.html

Alternatively, you can get full access to the **/var/www** directory by adding the user **p**i to the **root** group using the **usermod** command and then become an owner of this directory using the **chown** command.

After installing the Apache Web server we will be capable to create HTML pages or simple Web sites.

Raspberry Pi Web applications: using server-side scripts

Using a simple HTML page is not enough to build Web applications capable of driving external circuitry connected to GPIO ports of the Raspberry Pi. We must apply another technique called a server-side scripting, when an HTTP (Web) server does not simply send back a content of a requested file; instead that file will be executed as a program, and whatever that program outputs is sent back for your browser to display.

The server-side scripting employs scripts on a Web server which produces a response customized for each user's (client's) request to the website. Scripts can be written in any of a number of server-side scripting languages (PHP, Java, Python, etc.).

The server-side scripting is implemented through what is called the Common Gateway Interface or CGI.

The Common Gateway Interface (CGI) is a set of standards that define how information is exchanged between a Web server and a custom script. The programs being executed on the Web server side are called CGI scripts. These CGI programs can be written in Python, Perl, Shell, C or C++ program, etc.

CGI scripts are usually called from an HTML form as is shown in **Fig.16**.

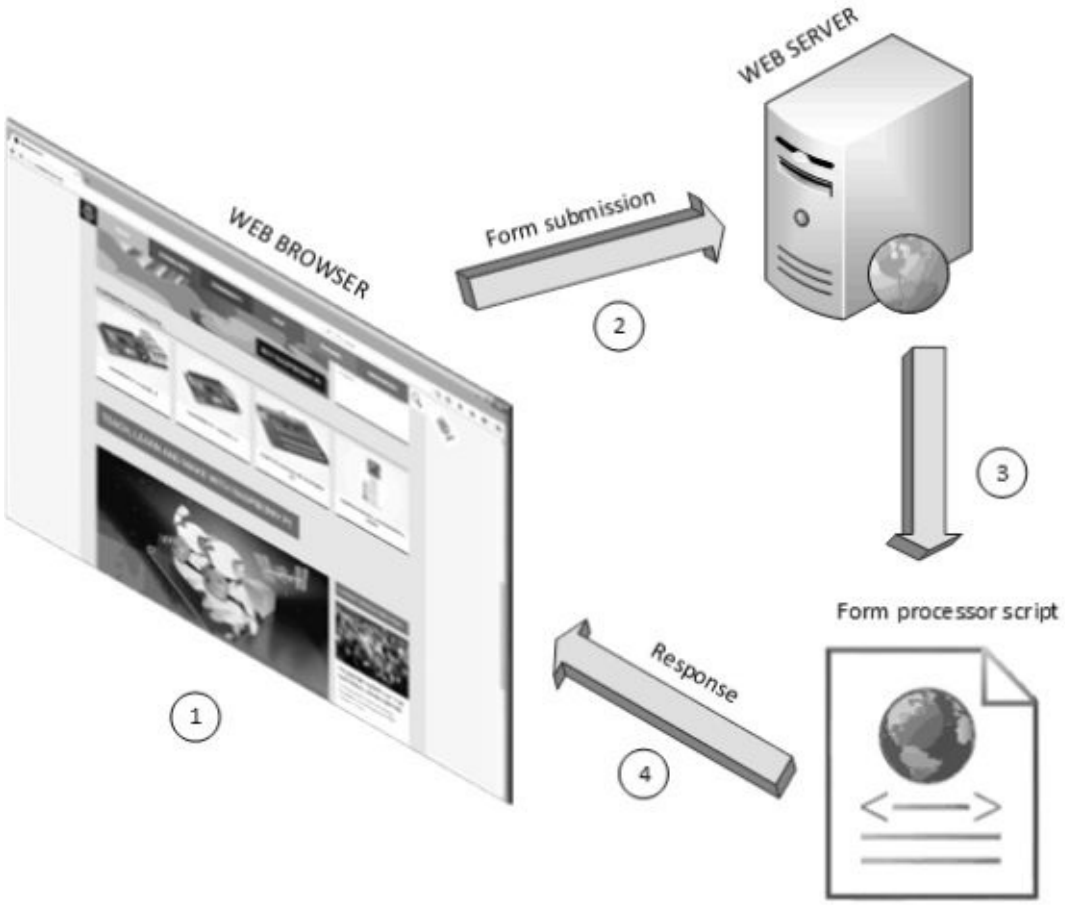


Fig.16

At the first step (**1** in **Fig.16**) a user loads the HTML form page in her web browser. The browser, in turn, sends a request to the web server. The web server returns the **HTML** page that contains the form. An HTML page returned by the web server can be either a HTML file stored on the server or an HTML page generated by a script running on the web server (like a Java, Python or PHP script).

Generally, the HTML page can contain references to Cascaded Style Sheets (CSS), JavaScript, PHP or Python files and various images. All that stuff is downloaded by the Web browser and the page is displayed. CSS files may also contain the visual elements (fonts, colors, etc.) of the page. The PHP, Python or JavaScript files will control the behavioral aspects (input validations, messages, etc.).

At the second step (**2** in **Fig.16**) the user fills the form and submits it to the Web server. Well written forms will have a bit of the code that gets triggered when the form is submitted. The script code checks the form submission values and informs the user if there is any error. If there is no error, the form is submitted.

Once the user has submitted the form, the form data is sent to the Web server. The form has to refer to the URL of some "**action**" processing the form submission data. The "**action**" pointed by the URL determine the PHP, Java or Python script that will process the data.

At the third step (**3** in **Fig.16**) the Web server passes the form submission data to the form processor script indicated by the "**action**".

Finally, a response is sent back to the browser (**4** in **Fig.16**). The form processor script sends a response indicating the success or failure of the form processing operations back to the server. The response could be to re-direct to another web page.

Raspberry Pi Web applications: running CGI scripts on the Apache Web server

Before writing the Python CGI scripts running on the server side, we need to configure our Apache Web server accordingly.

By default, Python is configured to only execute scripts with specific endings such as .cgi. In order to make Apache execute .py files, you need to edit a configuration file.

In a terminal window type this command:

sudo leafpad /etc/apache2/sites-enabled/000-default &

This file contains settings for the default site (you can create other sites on the same server). Find a section of code that may look like the following:

<Directory "/usr/lib/cgi-bin">
…
</Directory>

All CGI scripts have to be resided in the **/usr/lib/cgi-bin** directory. This entry determines the settings for scripts, so we must add the following line inside the above section of code:

AddHandler cgi-script .py

This line makes it possible for Apache to execute Python scripts. After saving changes, we need to reload Apache's configuration files by the command:

$ sudo service apache2 reload

Now we can test the Apache using the following Python source code:

```
#!/usr/bin/env python
print "Content-type: text/html\n\n"
print "<h3>Hello from the Python CGI script!</h3>"
```

Save the above source code in the in the **/usr/lib/cgi-bin/** directory as **hello.py** and make the file executable using the command:

$ sudo chmod +x /usr/lib/cgi-bin/hello.py

We can execute this script by accessing it through a Web browser. The IP address of my Raspberry Pi is 192.168.1.102, so I typed this address into my browser:

http://192.168.1.102/cgi-bin/hello.py

That's how to configure Apache Web server for using Python. Now we can develop our first Web application which will be capable of switching GPIO outputs via the Internet.

Project 9: Driving Raspberry Pi 2 GPIO outputs over the Internet

To access the Raspberry Pi 2 GPIO pins we need some special software. This and next projects will use the popular libraries called WiringPi and WiringPi2 developed by Gordon Henderson (you can visit his website at: http://wiringpi.com/). The WiringPi library implements most of the Arduino Wiring functions for the Raspberry Pi; WiringPi version 2 implements new functions for managing IO expanders and serial interfaces. Both libraries simplify a lot using the Raspberry Pi GPIO pins; additionally, the functions from those libraries allow to develop your own C applications.

To control GPIO pins we will use the **gpio** utility that comes with the **WiringP**i library. It allows you to control the GPIOs in a Python script. **Note** that the **gpio** command is designed to be installed as a setuid program, therefore **gpio** can be invoked by a normal

user without using the **sudo** command or the root privileges. This means that we can access a Web server running on Raspbian OS from the Internet and perform a read/write operation over some GPIO pin using the **gpio** command.

First, we need to install WiringPi and WiringPi2 modules. The installation procedure for WiringPi is described in detail on http://wiringpi.com/download-and-install/, so we just need to download it from GIT and build it using the ./**build** command.
Once those modules have been installed, we should be able to use the **gpio** utility. For example, to turn on and off the pin 0 we first need to configure this pin as output by entering the command

```
gpio mode 0 out
```

Here “0” is the Wiring pin number and “out” simply stands for output. The command

```
gpio write 0 1
```

pulls pin 0 high. Again, “0” is the pin number and “1” is the state (1 for HIGH and 0 for LOW). To drive pin 0 low, we should enter

```
gpio write 0 0
```

If you want to use the actual pin number (**GPIO17**) corresponding to the Wiring Pi number 0, you should use the **-g** flag in the command. For example, you can type

```
gpio -g write 17 1
```

instead of

```
gpio write 0 1
```

To read the state of the pin configured as input we can use the “gpio read” command. For example, to read the state of pin 0 we should enter:

```
gpio read 0
```

where “0” is the wiring pin number. The command returns the pin’s status (1 for HIGH and 0 for LOW).

Let’s develop a Web application capable of driving pins **GPIO18**, **GPIO23** and **GPIO24** HIGH/LOW. The circuit diagram for this project is shown in **Fig.17.**

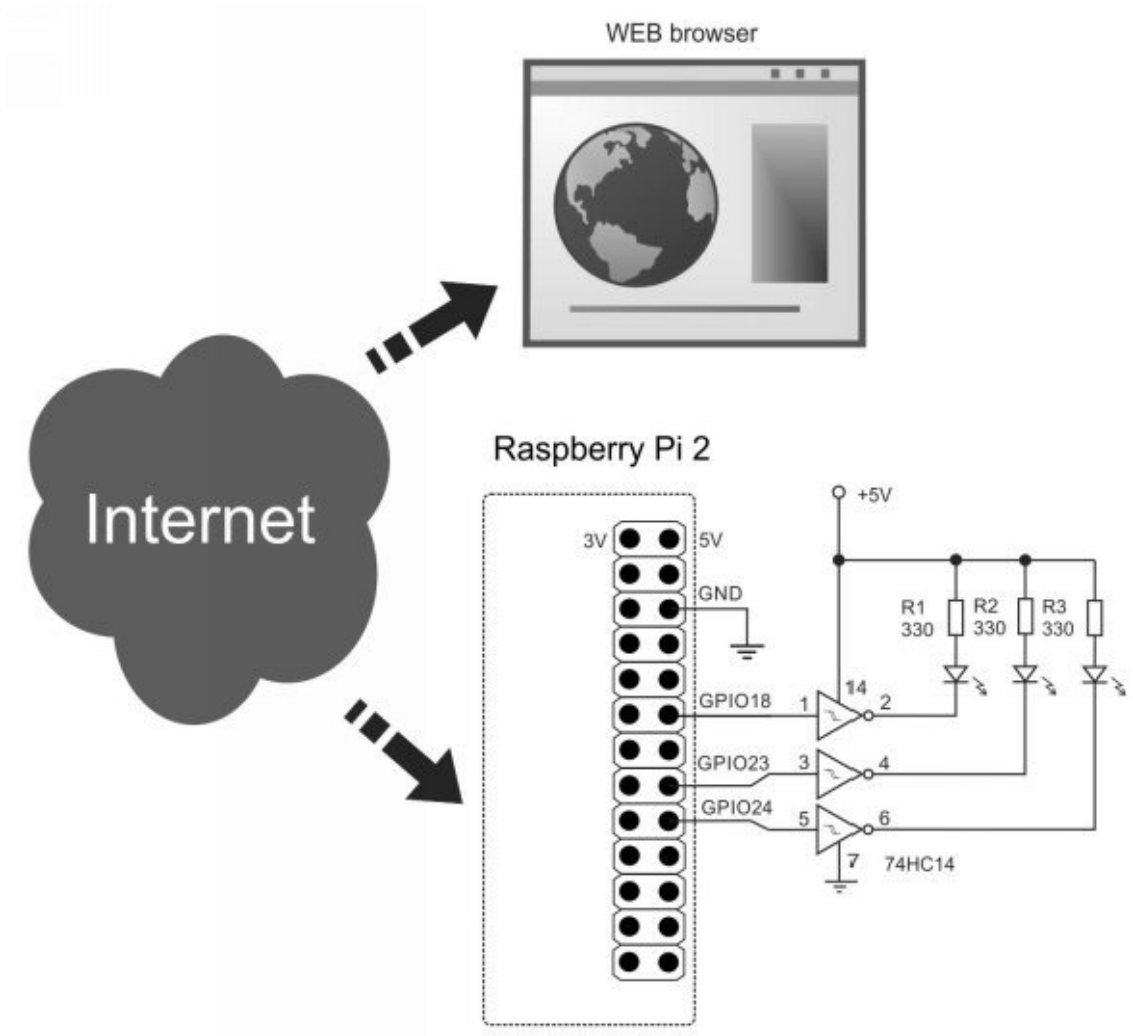


Fig.17

In this diagram, the **GPIO18**, **GPIO23** and **GPIO24** pins drive the LEDS through the corresponding buffers of the 74HC14 chip. The HIGH level on any GPIO pin causes the corresponding LED to switch ON. Otherwise, the LOW level on some pin drives the corresponding LED OFF.

The software part of our project will include a HTML web form and a server-side script written in Python. In general, HTML web forms are a composition of form elements such as buttons, checkboxes, and text input fields embedded inside of HTML documents. Those elements are needed to capture user input and execute some action depending on what form element has been selected.

The HTML form alone is unable to process form data, so we should develop some server-side script using Python. This script will be invoked by the HTML form code to process data captured by the HTML form elements.

The HTML form code for our project is shown in **Listing 22**. It must be saved in the **gpio_toggle.html** in the **/var/www** directory.

Listing 22.

```
<form action="/cgi-bin/gpio_toggle.cgi" method="POST">
  <input type="checkbox" name="port1" value="on" /> GPIO 18 <br>
  <input type="checkbox" name="port2" value="on" /> GPIO 23 <br>
  <input type="checkbox" name="port3" value="on" /> GPIO 24 <br>
  <input type="submit" name="Submit" value="Set output" />
</form>
```

Here the **form** tag contains two attributes, the "**action**" attribute which specifies the script to be executed when the "**submit"** button is clicked, and the "**method**" used to pass the form to the server, GET or POST. All our Web projects will use the POST method.

Our HTML form contains three "**checkbox**" form elements relating to **GPIO18**, **GPIO23** and **GPIO24** pins. When checked, each "**checkbox**" form element causes the corresponding GPIO pin to be pulled HIGH. Otherwise, when unchecked, the "**checkbox**" causes the corresponding GPIO pin to go LOW.

When a user press the button **Submit**, the **gpio_toggle.cgi** script is called. Upon termination, the Python application returns the result to the browser. **Note** that we should reside the HTML page in the default directory that is **var/www**. The Python script **gpio_toggle.cgi** should be saved in the **/usr/lib/cgi-bin** directory.

We also need to make the **gpio_toggle.cgi** file executable by typing the following command:

```
sudo chmod +x /usr/lib/cgi-bin/gpio_toggle.cgi
```

The Python CGI script is represented by the source code from **Listing 23**. It should be saved as **gpio_toggle.cgi.**

Listing 23.

```
#!/usr/bin/python

import cgi
import cgitb
import subprocess

subprocess.call([“gpio”,”-g”,“mode”,“18”,“out”])
subprocess.call([“gpio”,”-g”,“mode”,“23”,“out”])
subprocess.call([“gpio”,”-g”,“mode”,“24”,“out”])

form = cgi.FieldStorage()

if form.getvalue(‘port1’):
    p1Flag = “ON”
    subprocess.call([“gpio”,”-g”,“write”,“18”,“1”])
else:
    p1Flag = “OFF”
    subprocess.call([“gpio”,”-g”,“write”,“18”,“0”])

if form.getvalue(‘port2’):
    p2Flag = “ON”
    subprocess.call([“gpio”,”-g”,“write”,“23”,“1”])
else:
    p2Flag = “OFF”
    subprocess.call([“gpio”,”-g”,“write”,“23”,“0”])
if form.getvalue(‘port3’):
    p3Flag = “ON”
    subprocess.call([“gpio”,”-g”,“write”,“24”,“1”])
else:
    p3Flag = “OFF”
```

```
    subprocess.call([“gpio”,”-g”,“write”,“24”,“0”])

print “Content-type: text/html\n\n”
print “<html>”

print “<body”
print “<p>”
print “<h3> GPIO 18 is %s</h3>” % p1Flag
print “<h3> GPIO 23 is %s</h3>” % p2Flag
print “<h3> GPIO 24 is %s</h3>” % p3Flag

print “</body>”
print “</html>”
```

To launch our Web application we should enter the URL in the browser:

http://192.168.1.102/gpio_toggle.html

The running Web application produces the following outputs (**Fig.18-Fig.19)**.

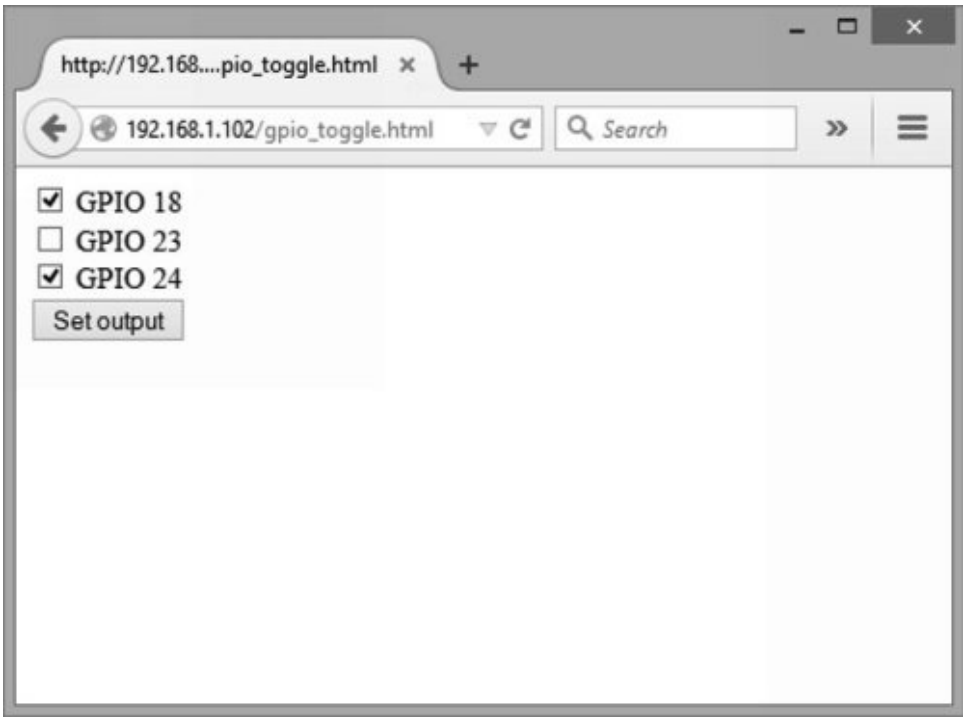


Fig.18

In this particular example, we drive pins **GPIO18** and **GPIO24** ON while leaving **GPIO23** OFF. After pressing the submit button “Set output“ the LEDs corresponding to pins **GPIO18** and **GPIO2**4 will be ON, while the LED driven by **GPIO23** will be OFF. The states of all pins will also be reflected in the browser window shown below.

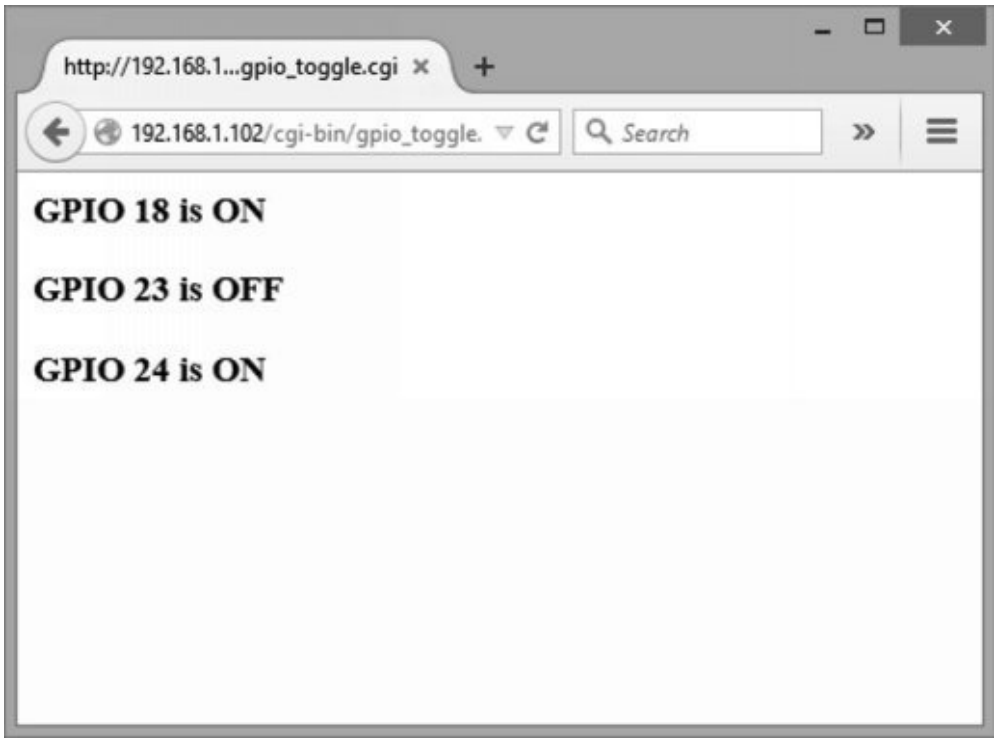


Fig.19

Project 10: Reading Raspberry Pi 2 GPIO inputs over the Internet

This project illustrates how to read the state of GPIO pins configured as inputs through your Web browser. The circuit diagram of the project is shown in **Fig.20**.

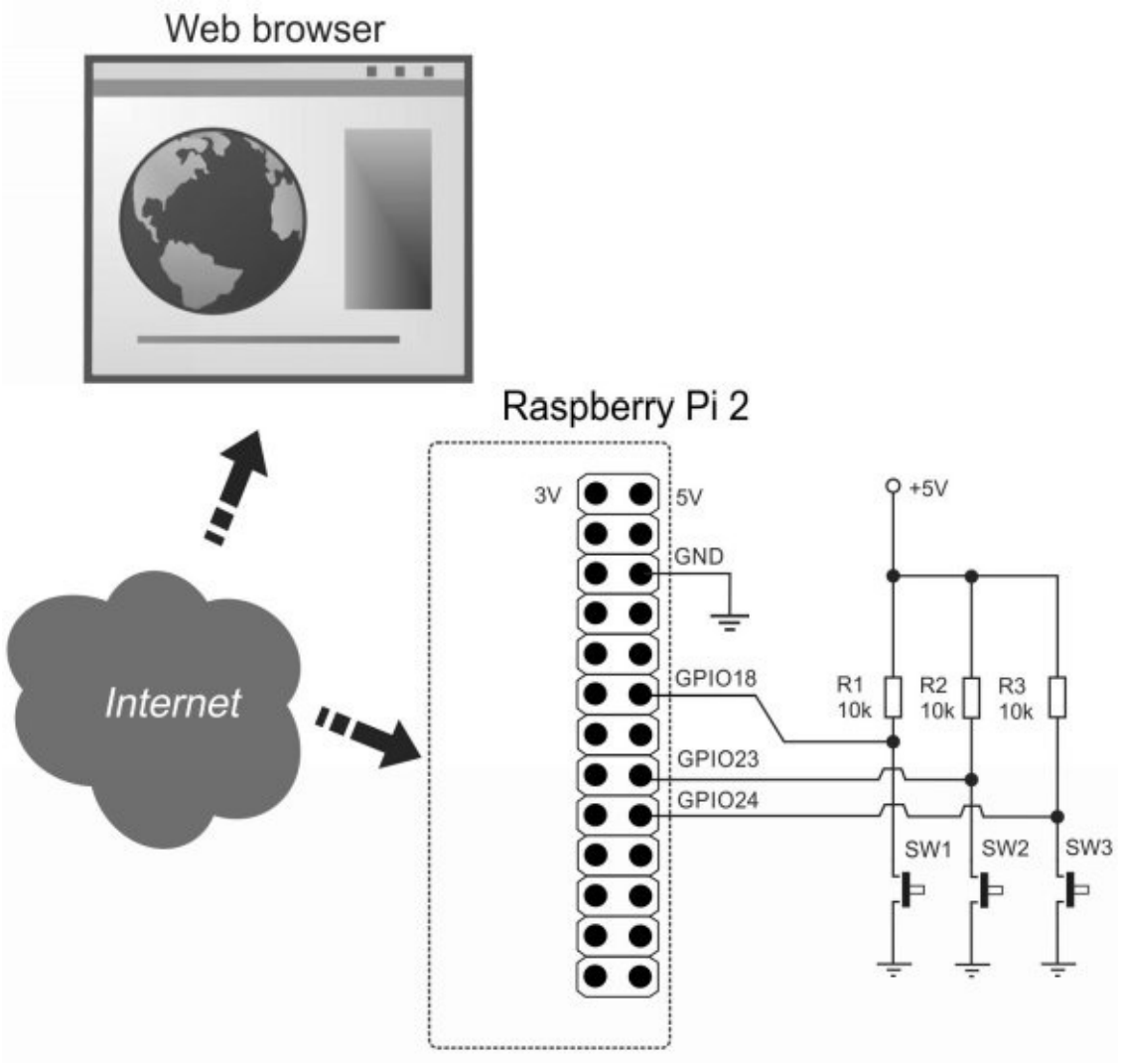


Fig.20

In this circuit, pins **GPIO18**, **GPIO23** and **GPIO24** configured as inputs pick up the signals from the digital sensors simulated by mechanical switches **SW1** – **SW3**.

The HTML form for this project is represented by the source code from **Listing 24**. This code should be saved in the **gpio_read.html** file in the **/var/www** directory.

Listing 24.

```
<form action=”/cgi-bin/gpio_read.cgi” method=“POST”>
    Select GPIO to read
    <select name=“gpio”>
    <option value=“18”> GPIO 18 </option>
    <option value=“23”> GPIO 23 </option>
    <option value=“24”> GPIO 24 </option>
    </select>
<br>
```

```
    <input type="submit" value="Submit">
</form>
```

The "**action**" attribute of the **form** tag points to the **gpio_read.cgi** script to be executed when the "**submit"** button is clicked.

Our HTML form also contains the drop down menu associated with the "**select**" element called "**gpio**".

The HTML "**select**" field provides essentially the same functionality as a "**checkbox**" form element. In this HTML web form, a user can select one of GPIO pins from a pre-determined series of options. Any of the **GPIO18**, **GPIO23** and **GPIO24** pins will be read when the corresponding option from the drop down menu has been selected.

When a user presses the button **Submit**, the **gpio_read.cgi** script is called and the Python application returns the result to the browser.

The Python script is represented by the source code from **Listing 25**. You should save the code in the **/usr/lib/cgi-bin** directory in the file **gpio_read.cgi**.

Listing 25.

```
#!/usr/bin/python

import cgi
import cgitb
import subprocess
from subprocess import Popen, PIPE

form = cgi.FieldStorage()

gpio = form['gpio'].value

subprocess.call(["gpio","-g","mode",gpio,"in"])
p1 = Popen(["gpio","-g","read",gpio], stdout=PIPE)
state = p1.communicate()[0]
```

```
print “Content-type: text/html\n\n”
print “<html>”

print “<body”
print “<p>”
print “<h3> GPIO %s input: %s </h3>” % (gpio, state)

print “</body>”
print “</html>”
```

The **gpio_toggle.cgi** file should be made executable by entering the command:

```
sudo chmod +x /usr/lib/cgi-bin/gpio_read.cgi
```

To launch our Web application we must enter the following URL in the browser window:

http://192.168.1.102/gpio_read.html

The running Web application produces the following outputs (**Fig.21** – **Fig.22)**.

Fig.21

Fig.22

Fig.21 illustrates the case when the **GPIO18** pin has been tested.

Project 11: A digital-to-analog converter driven over the Internet

This section represents the Web application which allows to set the DC voltage at the output of an digital-to-analog converter. The circuit diagram of the project is shown in **Fig.23**.

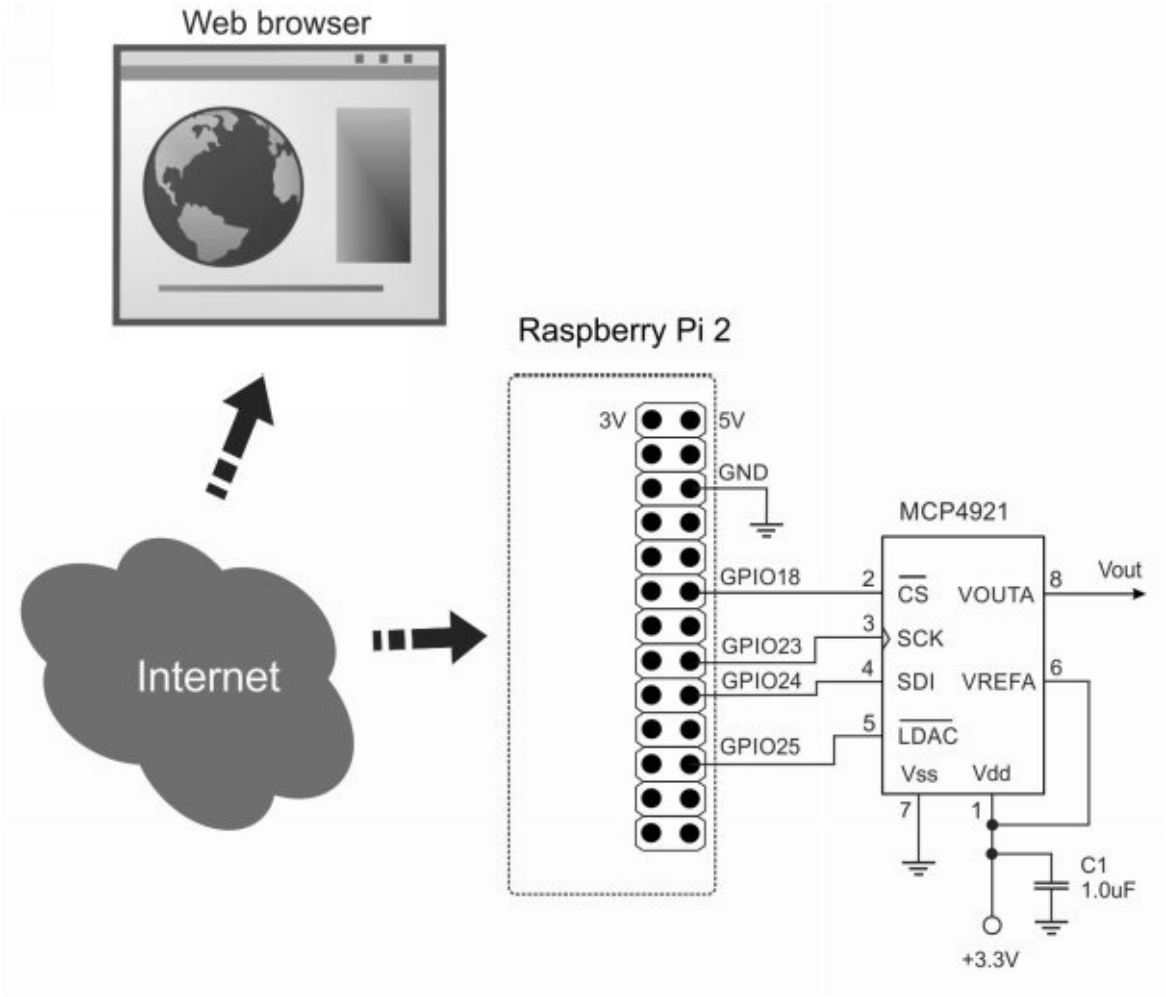


Fig.23

Our precision DC voltage source is based upon a digital-to-analog converter (DAC) MCP4921 driven through the SPI-compatible interface by the Raspberry Pi 2. The **CS** signal line of the DAC is connected to pin **GPIO18**. The clock signal **SCK** to the DAC is fed through the pin **GPIO23**. The bit stream to the **SDI** input of the DAC comes from pin **GPIO24**. The **LDAC** signal that updates the DAC output comes from pin **GPIO25**. All GPIO pins are configured as outputs.

The HTML form code is shown in **Listing 26**. This code should be saved in **/var/www/dac.html** file.

Listing 26.

```
<form action="cgi-bin/dac.py" method="POST"
    <h3>Enter DAC voltage in volts(0-3.3)</h3>
    <input type="text" size="20" name="dacinput" value""><br>
```

```
    <input type="submit" value="Set DAC output">
</form>
```

Our HTML web form comprises two form elements, the "**text**" field called "**dacinput**" and the button "**submit**". The "**text**" field captures the text string indicating the desired output level (in volts) of the DAC. When the "**submit**" button has been pressed, the Python script (**dac.py**) will process the data in the "**text**" field.

The Python source code (**Listing 27**) should be saved in the **/usr/lib/cgi-bin/dac.py** file.

Listing 27.

```
#!/usr/bin/python

import cgi
import cgitb
import subprocess

CS = "18"
SCK = "23"
SDI = "24"
LDAC = "25"

cmd = 0x7000  # the highest 4 bits compose the command
vref = 3.3    # the ref. voltage to MCP4921 dac
res = 4096.0

subprocess.call(["gpio","-g","mode",CS,"out"])
subprocess.call(["gpio","-g","mode",SCK,"out"])
subprocess.call(["gpio","-g","mode",SDI,"out"])
subprocess.call(["gpio","-g","mode",LDAC,"out"])
```

```
form = cgi.FieldStorage()

vdac = form[“dacinput”].value
vout = float(vdac)
bincode = int(vout/vref * res)  # binary code for the output voltage
fword = cmd | bincode

subprocess.call([“gpio”,”-g”,“write”,CS,“1”])
subprocess.call([“gpio”,”-g”,“write”,CS,“0”])
subprocess.call([“gpio”,”-g”, “write”, LDAC, “1”])

for i in range(0, 16):
    subprocess.call([“gpio”,”-g”, “write”, SCK, “0”]) # SCK goes LOW
    tmp = fword & 0x8000
    subprocess.call([“gpio”,”-g”, “write”, SDI, “0”])
    if (tmp):
    subprocess.call([“gpio”,”-g”, “write”, SDI, “1”])
    subprocess.call([“gpio”,”-g”, “write”, SCK, “1”]) # SCK goes HIGH
    fword = fword << 1

subprocess.call([“gpio”,”-g”,“write”,CS,“1”]) # the end of conversion
subprocess.call([“gpio”,”-g”, “write”, LDAC, “0”])



print “Content-type: text/html\n\n”
print “<html>”

print “<body”
print “<p>”
print “<h3> DAC output is set to: %s V</h3>” % vdac
```

print “</body>”

print “</html>”

The **dac.py** file should be made executable, so we need to enter the command:

sudo chmod +x /usr/lib/cgi-bin/dac.py

The Web application produces the output shown in **Fig.24** – **Fig.25**.

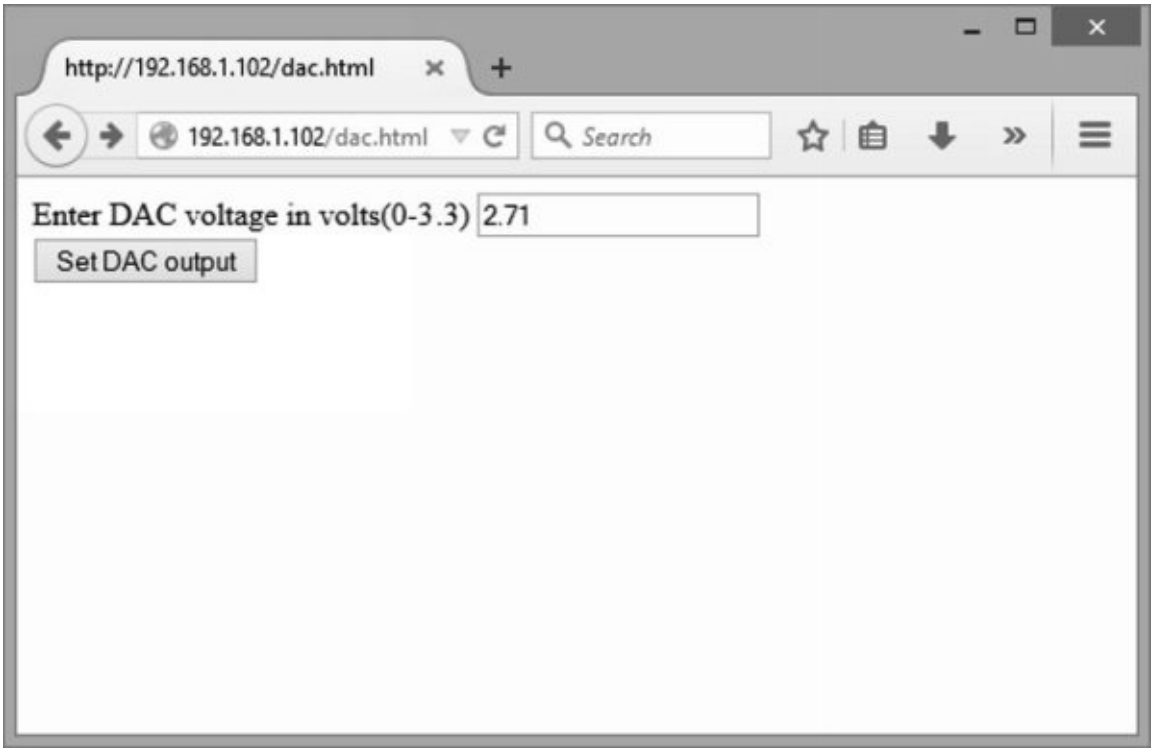


Fig.24

In this particular example, we set the DAC output to 2.71V.

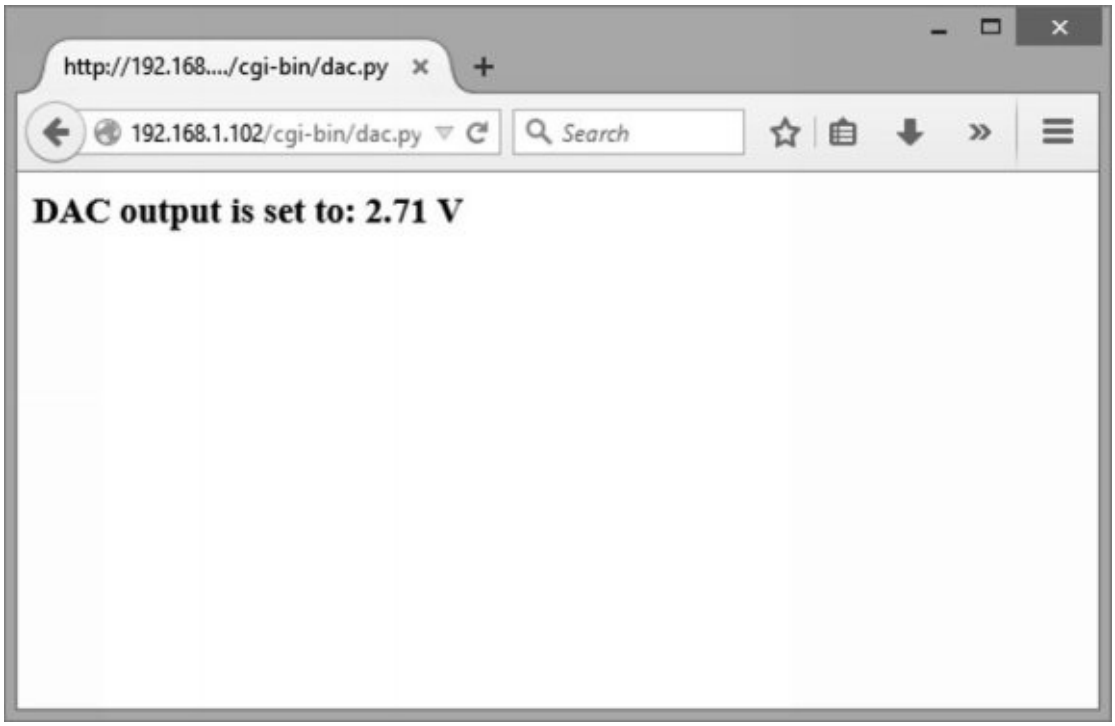


Fig.25

Project 12: A Web-controlled pulse width modulator

A Web application represented by this project allows to control the parameters of a PWM signal over the Internet. The circuit diagram of the project is shown in **Fig.26**.

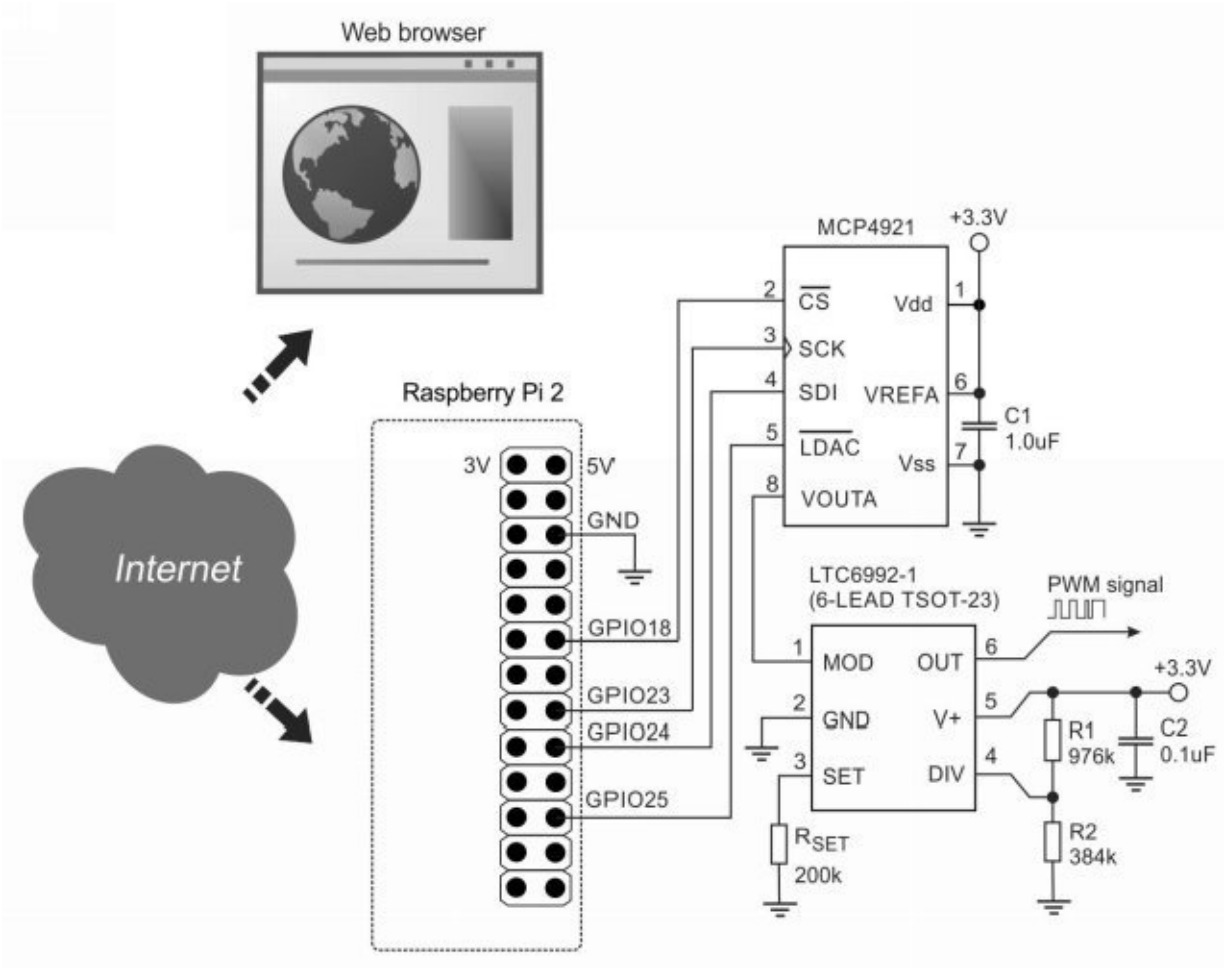


Fig.26

This circuit is based upon the LTC6992 device which is a silicon oscillator with an easy-to-use analog voltage-controlled pulse width modulation (PWM) capability. A single resistor, RSET, programs the LTC6992's internal master oscillator frequency. The output frequency is determined by this master oscillator and an internal frequency divider, programmable to eight settings from 1 to 16384. Applying a voltage between 0V and 1V on the MOD pin sets the duty cycle of the PWM signal.

The control voltage is fed to pin 1 (MOD) of LTC6992 from the DAC MCP4921. By altering the DAC output voltage, we can affect the duty cycle of the PWM signal on pin 6 (OUT) of LTC6992.

The software part of the project comprises the HTML Web form and the Python server-side script. The HTML source code is shown in **Listing 28**. This code should be saved as **pwm.html** in the **/var/www** directory.

Listing 28.

<form action="cgi-bin/pwm.py" method="POST"

```
    <h3>Enter a duty cycle value for PWM (0.1-0.9): </h3>
    <input type=“text” size=“20” name=“pwm” value””><br>
    <input type=“submit” value=“Set Duty Cycle”>
</form>
```

The HTML web form comprises two form elements, the “**text**” field called “**pwm**” and the button “**submit**”. The “**text**” field captures the text string indicating the desired duty cycle of the PWM signal on the LTC6992’s output. When the “**submit**” button has been pressed, the Python script (**pwm.py**) will process this string.

The Python source code is shown in **Listing 29**.

Listing 29.

```
#!/usr/bin/python

import cgi
import cgitb
import subprocess

CS = “18”
SCK = “23”
SDI = “24”
LDAC = “25”

cmd = 0x7000  # the highest 4 bits compose the command
vref = 3.3    # the ref.voltage to MCP4921 dac
res = 4096.0

subprocess.call([“gpio”,”-g”,“mode”,CS,“out”])
subprocess.call([“gpio”,”-g”,“mode”,SCK,“out”])
subprocess.call([“gpio”,”-g”,“mode”,SDI,“out”])
subprocess.call([“gpio”,”-g”,“mode”,LDAC,“out”])
```

```
form = cgi.FieldStorage()

duty = form[“pwm”].value
fduty = float(duty)

vdac = fduty*0.8 + 0.1 #using the approx. .formula: duty = (vdac-0.1)/0.8

bincode = int(vdac/vref * res)  # binary code for the output voltage
fword = cmd | bincode

subprocess.call([“gpio”,”-g”,“write”,CS,“1”])
subprocess.call([“gpio”,”-g”,“write”,CS,“0”])
subprocess.call([“gpio”,”-g”, “write”, LDAC, “1”])

for i in range(0, 16):
    subprocess.call([“gpio”,”-g”, “write”, SCK, “0”]) # SCK goes LOW
    tmp = fword & 0x8000
    subprocess.call([“gpio”,”-g”, “write”, SDI, “0”])
    if (tmp):
    subprocess.call([“gpio”,”-g”, “write”, SDI, “1”])
    subprocess.call([“gpio”,”-g”, “write”, SCK, “1”]) # SCK goes HIGH
    fword = fword << 1

subprocess.call([“gpio”,”-g”,“write”,CS,“1”]) # the end of conversion
subprocess.call([“gpio”,”-g”, “write”, LDAC, “0”])

print “Content-type: text/html\n\n”
print “<html>”
```

```
print “<body”
print “<p>”
print “<h3> Duty cycle is set to %s </h3>” % duty

print “</body>”
print “</html>”
```

The **pwm.py** should be made executable by entering the following command:

```
sudo chmod +x /usr/lib/cgi-bin/pwm.py
```

The following (**Fig.27** – **Fig.28**) illustrates working the Web application.

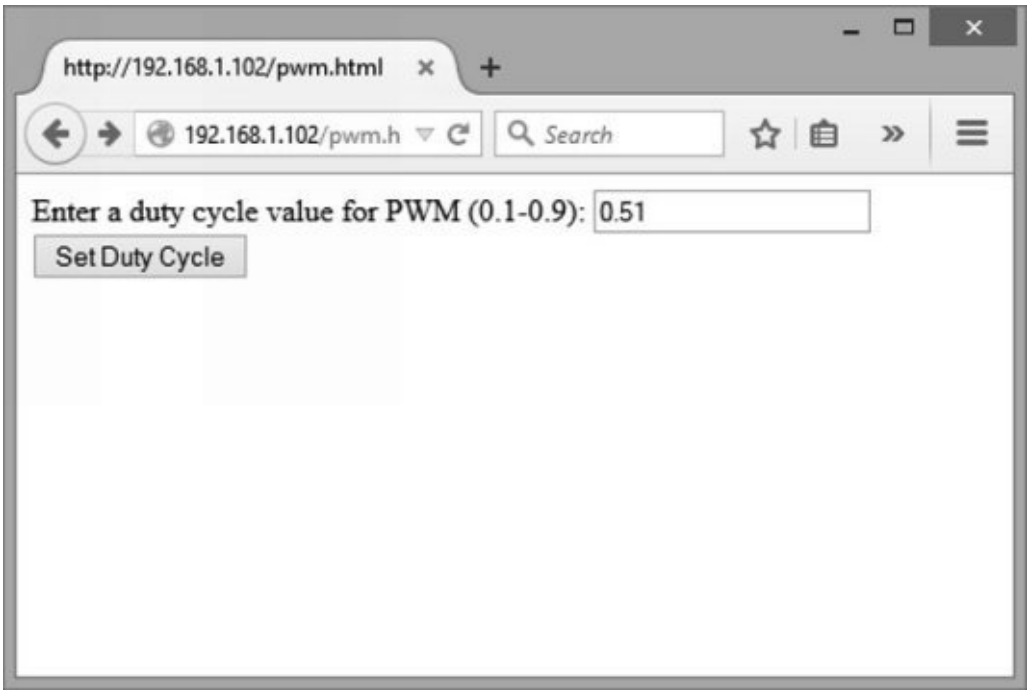


Fig.27

In this particular case, the duty cycle is set to 0.51 that is reflected in the browser window (**Fig.28**).

Fig.28

Project 13: Configuring trapezoidal signals from the Teensy DAC over the Internet

The project illustrates configuring a trapezoidal signal on the DAC output of the Teensy 3.1 over the Internet. The hardware circuit diagram of the project is shown in **Fig.29**.

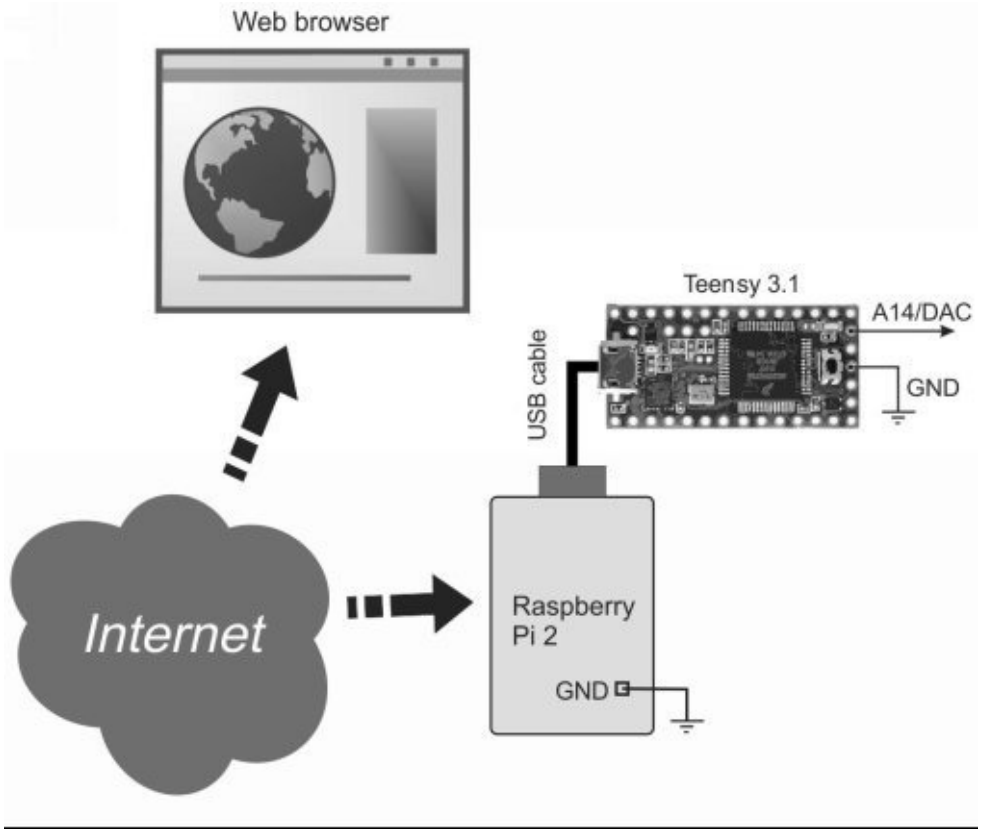


Fig.29

To access the hardware resources on the Teensy 3.1 board from the Internet we will use the approach illustrated in **Fig.30**.

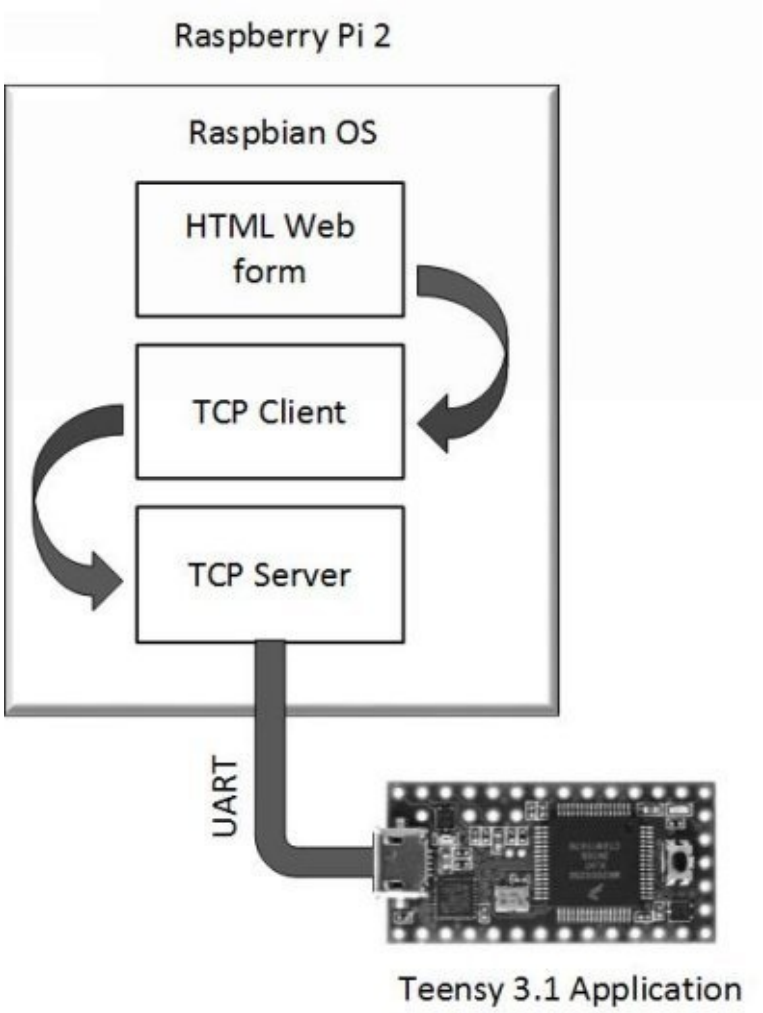


Fig.30

The Web application running on Raspbian OS will comprise three program modules: the HTML form, the server-side script (the TCP client) and the TCP server. When launched, the HTML form calls the server-side script which, in turn, passes the user's data via a local network to the TCP server. Finally, the TCP server transfers the data to Teensy 3.1 over a serial interface.

The Teensy 3.1 application parses the data received and adjusts the frequency on its DAC output accordingly.

Before launching the Web application, we should check if the Teensy 3.1 USB-to-Serial interface is properly configured on Raspbian OS. To observe serial interfaces type the following:

```
pi@raspberrypi /usr/lib/cgi-bin $ ls -l /dev/ttyA*
crw-rw–T 1 root dialout 166,  0 Apr 11 12:33 /dev/ttyACM0
crw-rw–- 1 root tty     204, 64 Apr 10 16:46 /dev/ttyAMA0
```

In this particular case, the /dev/ttyACM0 device is associated with the Teensy USB-to-

serial interface.

The HTML Web form is represented by the following code (**Listing 30**).

Listing 30.

```
<form action="cgi-bin/trapz_cl.py" method="POST"
    <h3>Enter the frequency of a trapezoidal waveform in Hz: </h3>
    <input type="text" size="20" name="trapz" value""><br>
    <input type="submit" value="Set Frequency">
</form>
```

It is seen that our Web form includes the "**text**" element called "**trapz**" and the "**submit**" button. The value of the desired frequency should be typed in the "**text**" field. The data in this field will be processed by the script **trapz_cl.py** after pressing the "**submit**" button. The HTML form code should be saved in the **/var/www/trapz.html** file.

The server-side script **trapz_cl.py** code is shown in **Listing 31**. You should place this file in the **/usr/lib/cgi-bin** directory.

Listing 31.

```
#!/usr/bin/python

import cgi
import cgitb
import socket


form = cgi.FieldStorage()
freq = form["trapz"].value
```

```
cl = socket.socket()
host = socket.gethostname()
port = 7999
cl.connect((host, port))
cl.send(freq)
cl.close()

print “Content-type: text/html\n\n”
print “<html>”
print “<body”
print “<p>”
print “<h3> The Frequency is set to %s Hz</h3>” % freq
print “</body>”
print “</html>”
```

The TCP server application written in Python (file **/usr/lib/cgi-bin/trapz_srv.py**) has the following source code (**Listing 32**). **Note** that the TCP server should be started before the Web application is launched.

Listing 32.

```
#!/usr/bin/python

import socket
import wiringpi2

srv = socket.socket()
host = socket.gethostname()
port = 7999
srv.bind((host, port))
srv.listen(5)
print “TCP Server is waiting for incoming requests on port 7999…”
```

```
try:
    while True:
    cl, addr = srv.accept()
    freq = cl.recv(256)
    cmd = “4,” + freq
    print cmd
    cl.close()
    s1 = wiringpi2.serialOpen(‘/dev/ttyACM0’, 9600)
    wiringpi2.serialPuts(s1, cmd)
    wiringpi2.serialClose(s1)
except KeyboardInterrupt:
    print “TCP Server is shutting down…”
    srv.close()
```

When running, the TCP server receives the data used for setting the desired. This data is then sent to the Teensy 3.1 board via the serial interface **/dev/ttyACM0**. In this code, the serial communication is controlled by the wiringPi2 library functions:

```
s1 = wiringpi2.serialOpen(‘/dev/ttyACM0’, 9600)
wiringpi2.serialPuts(s1, cmd)
wiringpi2.serialClose(s1
```

The TCP server also outputs the command string sent to Teensy 3.1:

```
pi@raspberrypi /usr/lib/cgi-bin $ ./trapz_srv.py
TCP Server is waiting for incoming requests on port 7999…
4,200
4,150
4,370
4,240
4,410
```

Here 4 determines the command which is followed by the value of the desired frequency.

Both **trapz_cl.py** (TCP client) and **trapz_srv.py** (TCP server) should be made executables by entering the commands:

sudo chmod +x /usr/lib/cgi-bin/trapz_cl.py

sudo chmod +x /usr/lib/cgi-bin/trapz_srv.py

The source code of the Teensy 3.1 application is shown in **Listing 33**.

Listing 33.

```
#include <TimerOne.h>
#define K 3906 //1000000/256

volatile int freq = 300; // an initial frequency freq is set to 300 Hz
volatile int tuS = K/freq;

const int NUM_SAMPLES = 256;
unsigned char TRAP_TABLE[NUM_SAMPLES];
volatile int cnt = 0;

void setup()
{
Serial.begin(9600);
fillBuf();
analogWriteResolution(8); // 8-bit write resolution is set
Timer1.initialize(tuS);          // interval in uS
Timer1.attachInterrupt(writeDAC);  // ISR to call when an interval expires
Timer1.start();
}
void loop()
{
```

```
while (Serial.available() > 0)
{
  // look for the next valid integer in the incoming serial stream:
  int cmd = Serial.parseInt();
  if (cmd == 4) // 4 = set DAC frequency
  {
    freq = Serial.parseInt();
    if (freq <= 500) // max frequency can not exceed 500 Hz
    {
    tuS = K/freq;
    noInterrupts();
    Timer1.stop();
    Timer1.initialize(tuS);
    Timer1.start();
    interrupts();
    }
  }
}
}

void fillBuf(void)
{
for(int i = 0;i < NUM_SAMPLES;i++)
{
  if (i >= 180)
    TRAP_TABLE[i] = 180;
  else
    TRAP_TABLE[i] = i;
}
}
```

```
void writeDAC(void)
{
  if(cnt == 256) cnt = 0;
  analogWrite(A14, TRAP_TABLE[cnt++]); // write DAC
}
```

The running Web application produces the following outputs (**Fig.31** – **Fig.32**).

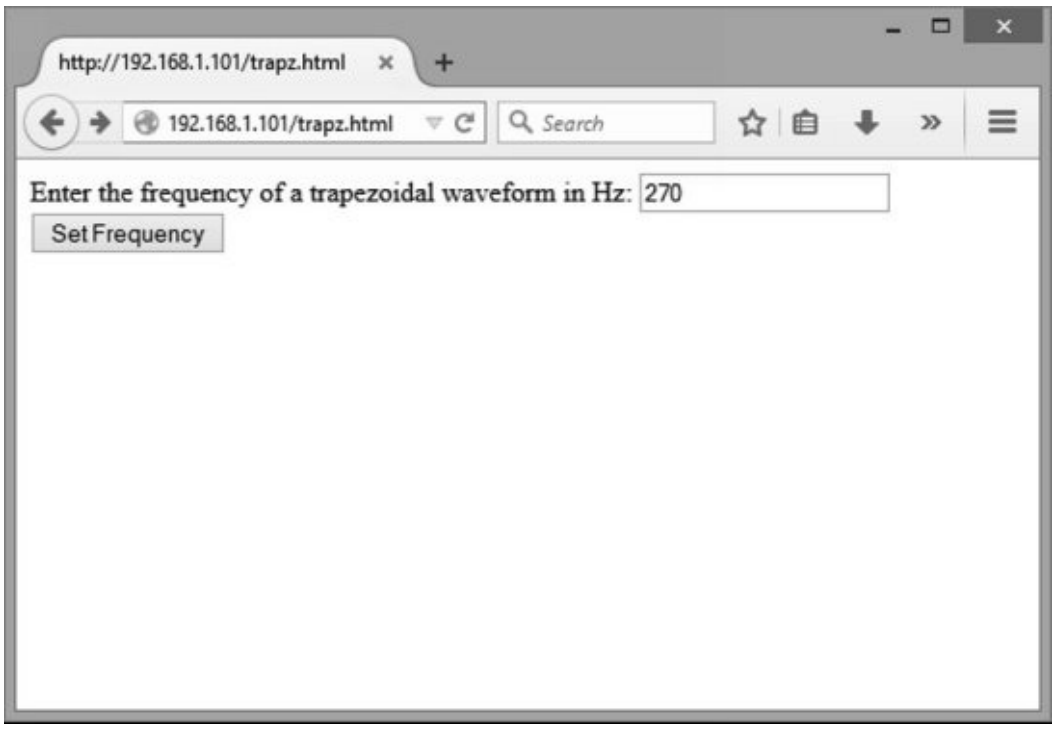


Fig.31

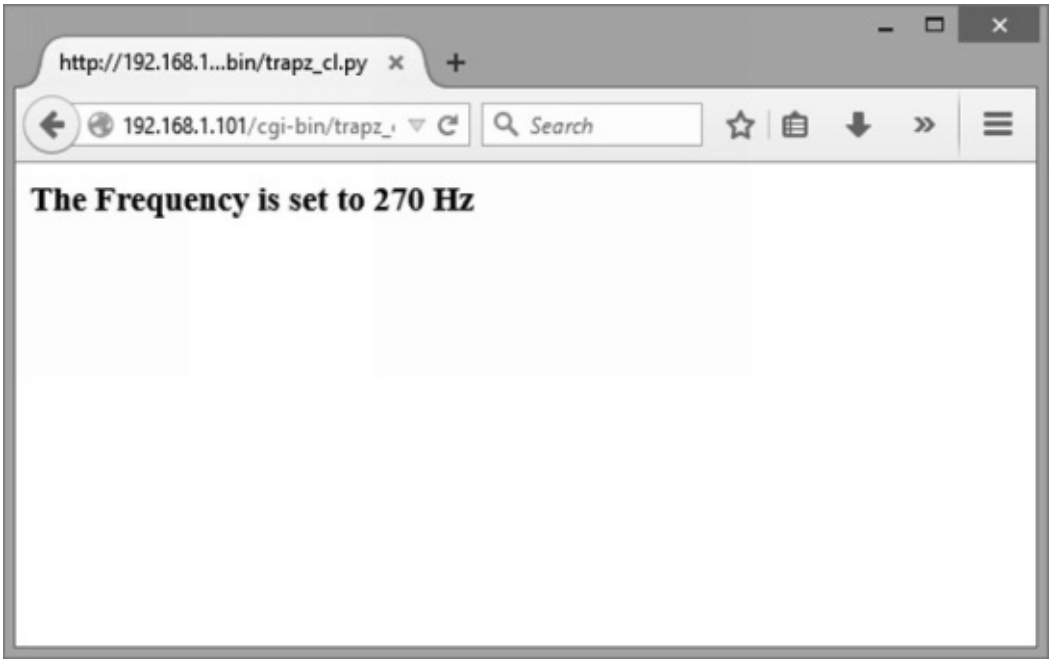


Fig.32

Project 14: Reading the analog channels of Teensy 3.1 over the Internet

This project will illustrate how to pick up analog input voltage from channels 0 – 2 of the Teensy 3.1 board. The basic hardware configuration for this project is shown in **Fig.33**.

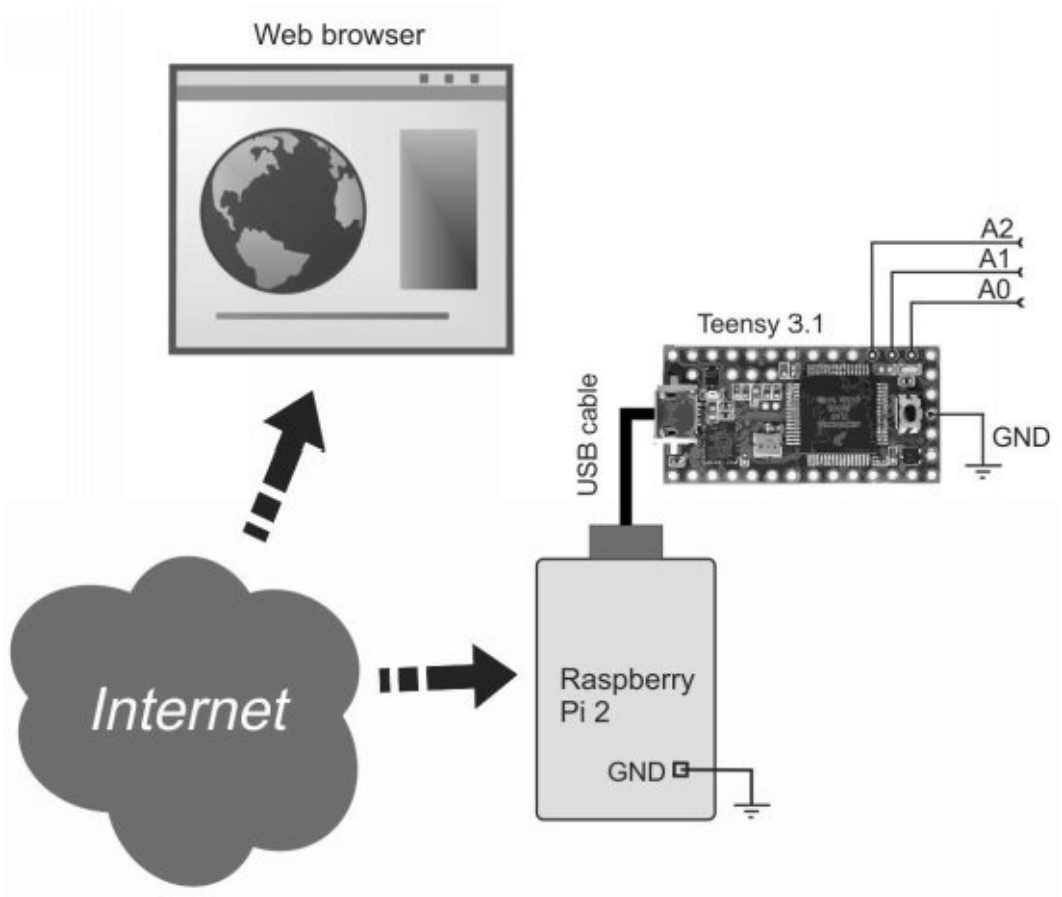


Fig.33

The software part of the Web application includes the HTML Web form, the TCP client and the TCP server. The Web application is running on Raspbian OS of the Raspberry Pi 2 board.

Before launching the Web application, check if Teensy 3.1 USB-to-Serial interface is

properly configured in Raspberry Pi 2 by entering:

```
pi@raspberrypi /usr/lib/cgi-bin $ ls -l /dev/ttyA*
crw-rw–T 1 root dialout 166,  0 Apr 11 12:33 /dev/ttyACM0
crw-rw–- 1 root tty     204, 64 Apr 10 16:46 /dev/ttyAMA0
```

In this particular case, the **/dev/ttyACM0** device is associated with the Teensy 3.1 USB-to-Serial interface.

The HTML Web form is represented by the following source code (**Listing 34**). The code should be saved in the **ain.html** file located in the **/var/www** directory.

Listing 34.

```
<form action=”/cgi-bin/ain_cl.py” method=“POST”>
  Select Teensy 3.1 analog channel to read
  <select name=“ain”>
    <option value=“0”> A0 </option>
    <option value=“1”> A1 </option>
    <option value=“2”> A2 </option>
  </select>
<br>
  <input type=“submit” value=“Submit”>
</form>
```

The “**action**” attribute of the **form** tag points to the **ain_cl.py** script to be executed when the “**submit”** button is clicked. The HTML form also contains the drop down menu associated with the “**select**” element called “**ain**”.

With this HTML Web form, a user can select one of the **A0** – **A2** analog inputs from a pre-determined series of options. Any of the **A0**, **A1** and **A2** pins will be read when the corresponding option from the drop down menu has been selected.

When a user presses the button **Submit**, the **ain_cl.py** server-side script located in the **/usr/lib/cgi-bin/** directory is called. The script runs the TCP client passing the data to the TCP server. The TCP client source code is shown in **Listing 35**.

Listing 35.

```
#!/usr/bin/python

import cgi
import cgitb
import socket
import time

form = cgi.FieldStorage()
ch = form[“ain”].value
cl = socket.socket()
host = socket.gethostname()
port = 7999
cl.connect((host, port))
cl.send(ch)

time.sleep(1)
inpV = cl.recv(256)
cl.close()

print “Content-type: text/html\n\n”
print “<html>”
print “<body”
print “<p>”
print “<h3> %s </h3>” % inpV
print “</body>”
print “</html>”
```

The TCP server source code is shown in **Listing 36**. It should be saved in the **/usr/lib/cgi-bin/ain_srv.py** file.

Listing 36.

```
#!/usr/bin/python

import socket
import time
import serial

srv = socket.socket()
host = socket.gethostname()
port = 7999
srv.bind((host, port))
srv.listen(5)
print “TCP Server is waiting for incoming requests on port 7999…”

try:
   while True:
```

```
        cl, addr = srv.accept()
        ch = cl.recv(32)
        cmd = "2," + ch
        print cmd
        s1 = serial.Serial(port = "/dev/ttyACM0", baudrate=9600, timeout=5.0)
    s1.write(cmd)
        time.sleep(1)
    readVal = s1.readline()
        s1.close()
        cl.send(readVal)
        cl.close()
except KeyboardInterrupt:
    print "TCP Server is shutting down…"
    srv.close()
```

Both **ain_cl.py** (TCP client) and **ain_srv.py** (TCP server) should be made executables by entering the commands:

```
sudo chmod +x /usr/lib/cgi-bin/ain_cl.py
sudo chmod +x /usr/lib/cgi-bin/ain_srv.py
```

The Teensy 3.1 application source code is shown in **Listing 37**.

Listing 37.

```
float vref = 3.3; // the reference to ADC
float res = 4096.0;

float vin;
float LSB;
int bincode;
int cmd, ch;

void setup()
{
```

```
  Serial.begin(9600);
  analogReadResolution(12);
  LSB = vref/res;
}

void loop()
{
  while (Serial.available() > 0)
  {
    // look for the next valid integer in the incoming serial stream:

    cmd = Serial.parseInt();
    if (cmd == 2) // 2 = read analog input
    {
    ch = Serial.parseInt(); // channel 0 through 2
    if ((ch >= 0) && (ch < 3))
    {
    bincode = analogRead(ch);
    vin = (float)bincode * LSB;
    Serial.print("Input voltage on channel ");
    Serial.print(ch);
    Serial.print(", V: ");
    Serial.println(vin,3);
    }
    }
  }
}
```

The running Web application produces the following output (**Fig.34** – **Fig.35**).

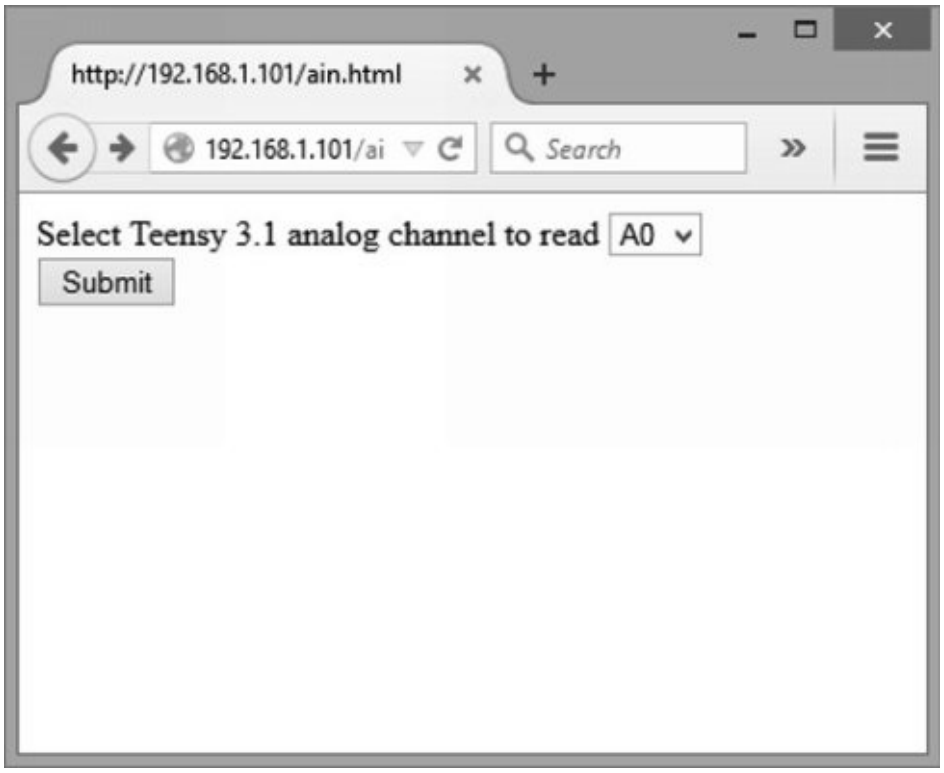


Fig.34

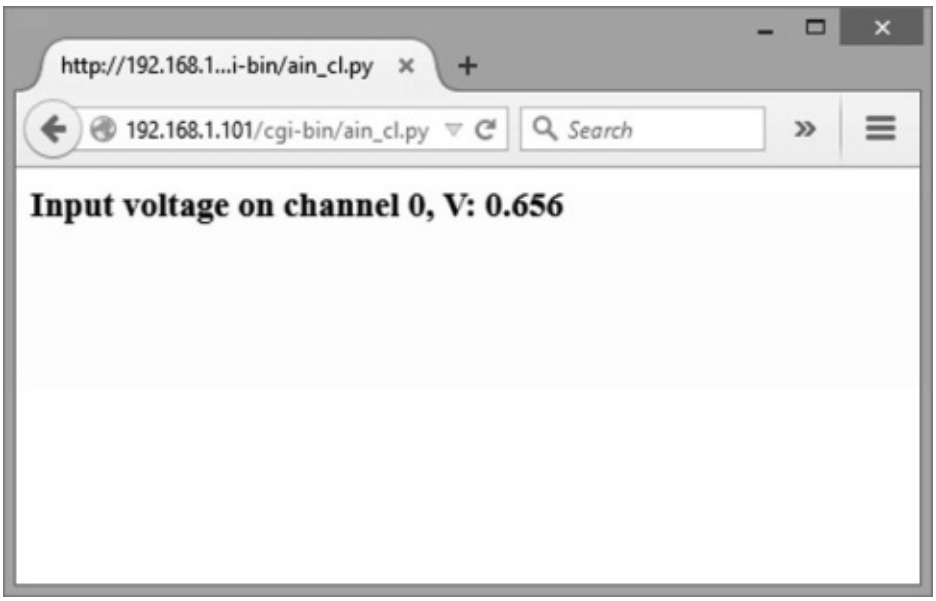


Fig.35

In this particular case, we read the input voltage fed to analog channel 0 (A0).