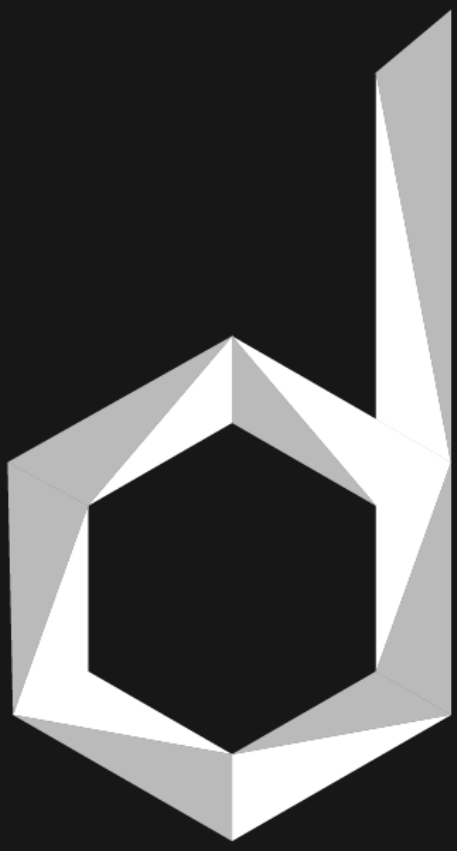
INGENIERÍA MECATRÓNICA



DI_CERO

DIEGO CERVANTES RODRÍGUEZ

INSTRUMENTACIÓN VIRTUAL

PYTHON 3.9.7, C# & LABVIEW

Instrumentación Virtual con Arduino - Recibir y Mandar Datos

Contenido

Teoría – **Instrumentación Virtual**:

La instrumentación virtual es el uso de una computadora como instrumento de medición, en vez de utilizar herramientas físicas como osciloscopios. Esto se logra haciendo uso de instrumentos que no son tangibles para desarrollar aplicaciones con interfaz de usuario que realicen adquisición de datos a través de lenguajes de programación, protocolos de comunicación, etc. Su principal uso es bajar costos de operación en algún proceso.

Instrucciones – **Instrumentación Virtual Arduino**:

Escriba un programa que realice una operación de instrumentación virtual con Arduino que pueda recibir y mandar datos a la tarjeta de desarrollo.

En este caso lo que se busca hacer es leer los datos de tensión de un pin analógico del Arduino (A0) y escribir en un pin digital (13) para encender y apagar un led, estos datos se almacenarán en una variable de Python, se graficarán actualizando sus datos en tiempo real y finalmente serán guardados en un archivo de Excel.

Para ello primero se debe haber ejecutado un programa en el IDE de Arduino que suba la librería Standard Firmata a la tarjeta de desarrollo, además de indicar cuál es el puerto de conexión serial que se debe utilizar.

Pseudocódigo.

1. Ejecutar un código en el IDE de Arduino que suba a la tarjeta de desarrollo la librería Standard Firmata de Arduino, para que de esa forma se pueda controlar y/o monitorear el estado de los pines digitales y analógicos del Arduino, luego se recaben datos de tensión que vayan de 0 a 5V en el pin A0 y al mismo tiempo se haga parpadear un led en el pin 13, además de indicar cual es el puerto de conexión serial entre la placa de desarrollo y la computadora.
2. Iniciar la comunicación serial con el puerto de conexión previamente utilizado en el IDE del Arduino.
3. Indicar el número de datos a recabar del Arduino.
4. Leer los datos del pin analógico A0 del Arduino y realizar su conversión de número digital a valor de tensión, esto se realiza tomando en cuenta el rango de valores de tensión (que va de 0 a 5V) y el rango de valores digitales que se conforma de 10 bits, cuando este número binario se convierte a decimal se considera que podrá estar en el rango de 0 a $2^{10} - 1 = 1023$, algo importante a mencionar del método read() de la librería pyfirmata es que indica los datos binarios de tensión en un rango de valores de 0 a 1, por lo cual la conversión entonces se realiza a través de la siguiente ecuación:

$$Tensión = Tensión_{Binaria} * Tensión_{Max} = Tensión_{Binaria} * 5\ [V]$$

5. Almacenar los datos de tensión en una lista, tupla o diccionario.
6. Graficar los datos recabados de tiempo vs. tensión.

7. Actualizar dinámicamente la gráfica para que se actualicen sus valores y se muestren en tiempo real.
    a. Esto se realiza utilizando la librería **drawnow** de Python.
8. Hacer que solo se muestren dos decimales de los valores de tensión recabados.
9. Imprimir en consola el vector que haya almacenado todos los valores de tensión recabados.

Código IDE Arduino – Librería Standard Firmata:

Es importante mencionar que los archivos de Arduino a fuerza deben encontrarse dentro de una carpeta que tenga el mismo nombre que el nombre del archivo con extensión .ino que almacena el programa escrito en lenguaje Arduino, este nombre tanto de la carpeta no puede contener espacios.

```
/*La librería de Arduino que estás viendo es una biblioteca llamada "Firmata". Firmata es un protocolo
de comunicación que permite controlar un Arduino desde un software en una computadora o dispositivo
compatible. La biblioteca implementa el protocolo Firmata en Arduino, lo que permite establecer una
comunicación bidireccional entre el Arduino y el software de control, permitiendo controlar los pines
de Arduino, incluidos los pines analógicos y digitales, los pines PWM, los servos y la comunicación I2C.
En este caso ese software de control estará hecho con Python en Visual Studio Code.*/

#include <Servo.h>
#include <Wire.h>
#include <Firmata.h>

#define I2C_WRITE                   B00000000
#define I2C_READ                    B00001000
#define I2C_READ_CONTINUOUSLY       B00010000
#define I2C_STOP_READING            B00011000
#define I2C_READ_WRITE_MODE_MASK    B00011000
#define I2C_10BIT_ADDRESS_MODE_MASK B00100000
#define I2C_END_TX_MASK             B01000000
#define I2C_STOP_TX                 1
#define I2C_RESTART_TX              0
#define I2C_MAX_QUERIES             8
#define I2C_REGISTER_NOT_SPECIFIED  -1

// the minimum interval for sampling analog input
#define MINIMUM_SAMPLING_INTERVAL   1


/*==============================================================================
 * GLOBAL VARIABLES
 *============================================================================*/

#ifdef FIRMATA_SERIAL_FEATURE
SerialFirmata serialFeature;
#endif

/* analog inputs */
int analogInputsToReport = 0; // bitwise array to store pin reporting

/* digital input ports */
byte reportPINs[TOTAL_PORTS];       // 1 = report this port, 0 = silence
byte previousPINs[TOTAL_PORTS];     // previous 8 bits sent

/* pins configuration */
byte portConfigInputs[TOTAL_PORTS]; // each bit: 1 = pin in INPUT, 0 = anything else

/* timer variables */
unsigned long currentMillis;        // store the current value from millis()
unsigned long previousMillis;       // for comparison with currentMillis
unsigned int samplingInterval = 19; // how often to run the main loop (in ms)

/* i2c data */
struct i2c_device_info {
  byte addr;
  int reg;
  byte bytes;
  byte stopTX;
};
```

```
/* for i2c read continuous more */
i2c_device_info query[I2C_MAX_QUERIES];

byte i2cRxData[64];
boolean isI2CEnabled = false;
signed char queryIndex = -1;
// default delay time between i2c read request and Wire.requestFrom()
unsigned int i2cReadDelayTime = 0;

Servo servos[MAX_SERVOS];
byte servoPinMap[TOTAL_PINS];
byte detachedServos[MAX_SERVOS];
byte detachedServoCount = 0;
byte servoCount = 0;

boolean isResetting = false;

// Forward declare a few functions to avoid compiler errors with older versions
// of the Arduino IDE.
void setPinModeCallback(byte, int);
void reportAnalogCallback(byte analogPin, int value);
void sysexCallback(byte, byte, byte*);

/* utility functions */
void wireWrite(byte data)
{
#if ARDUINO >= 100
  Wire.write((byte)data);
#else
  Wire.send(data);
#endif
}

byte wireRead(void)
{
#if ARDUINO >= 100
  return Wire.read();
#else
  return Wire.receive();
#endif
}

/*==============================================================================
 * FUNCTIONS
 *============================================================================*/

void attachServo(byte pin, int minPulse, int maxPulse)
{
  if (servoCount < MAX_SERVOS) {
    // reuse indexes of detached servos until all have been reallocated
    if (detachedServoCount > 0) {
      servoPinMap[pin] = detachedServos[detachedServoCount - 1];
      if (detachedServoCount > 0) detachedServoCount--;
    } else {
      servoPinMap[pin] = servoCount;
      servoCount++;
    }
    if (minPulse > 0 && maxPulse > 0) {
      servos[servoPinMap[pin]].attach(PIN_TO_DIGITAL(pin), minPulse, maxPulse);
    } else {
      servos[servoPinMap[pin]].attach(PIN_TO_DIGITAL(pin));
    }
  } else {
    Firmata.sendString("Max servos attached");
  }
}

void detachServo(byte pin)
{
  servos[servoPinMap[pin]].detach();
  // if we're detaching the last servo, decrement the count
  // otherwise store the index of the detached servo
  if (servoPinMap[pin] == servoCount && servoCount > 0) {
    servoCount--;
  } else if (servoCount > 0) {
    // keep track of detached servos because we want to reuse their indexes
    // before incrementing the count of attached servos
    detachedServoCount++;
    detachedServos[detachedServoCount - 1] = servoPinMap[pin];
  }
```

```
  servoPinMap[pin] = 255;
}

void enableI2CPins()
{
  byte i;
  // is there a faster way to do this? would probaby require importing
  // Arduino.h to get SCL and SDA pins
  for (i = 0; i < TOTAL_PINS; i++) {
    if (IS_PIN_I2C(i)) {
      // mark pins as i2c so they are ignore in non i2c data requests
      setPinModeCallback(i, PIN_MODE_I2C);
    }
  }

  isI2CEnabled = true;

  Wire.begin();
}

/* disable the i2c pins so they can be used for other functions */
void disableI2CPins() {
  isI2CEnabled = false;
  // disable read continuous mode for all devices
  queryIndex = -1;
}

void readAndReportData(byte address, int theRegister, byte numBytes, byte stopTX) {
  // allow I2C requests that don't require a register read
  // for example, some devices using an interrupt pin to signify new data available
  // do not always require the register read so upon interrupt you call Wire.requestFrom()
  if (theRegister != I2C_REGISTER_NOT_SPECIFIED) {
    Wire.beginTransmission(address);
    wireWrite((byte)theRegister);
    Wire.endTransmission(stopTX); // default = true
    // do not set a value of 0
    if (i2cReadDelayTime > 0) {
      // delay is necessary for some devices such as WiiNunchuck
      delayMicroseconds(i2cReadDelayTime);
    }
  } else {
    theRegister = 0;  // fill the register with a dummy value
  }

  Wire.requestFrom(address, numBytes);  // all bytes are returned in requestFrom

  // check to be sure correct number of bytes were returned by slave
  if (numBytes < Wire.available()) {
    Firmata.sendString("I2C: Too many bytes received");
  } else if (numBytes > Wire.available()) {
    Firmata.sendString("I2C: Too few bytes received");
  }

  i2cRxData[0] = address;
  i2cRxData[1] = theRegister;

  for (int i = 0; i < numBytes && Wire.available(); i++) {
    i2cRxData[2 + i] = wireRead();
  }

  // send slave address, register and received bytes
  Firmata.sendSysex(SYSEX_I2C_REPLY, numBytes + 2, i2cRxData);
}

void outputPort(byte portNumber, byte portValue, byte forceSend)
{
  // pins not configured as INPUT are cleared to zeros
  portValue = portValue & portConfigInputs[portNumber];
  // only send if the value is different than previously sent
  if (forceSend || previousPINs[portNumber] != portValue) {
    Firmata.sendDigitalPort(portNumber, portValue);
    previousPINs[portNumber] = portValue;
  }
}

/* -----------------------------------------------------------------------------
 * check all the active digital inputs for change of state, then add any events
 * to the Serial output queue using Serial.print() */
void checkDigitalInputs(void)
{
```

```
  /* Using non-looping code allows constants to be given to readPort().
   * The compiler will apply substantial optimizations if the inputs
   * to readPort() are compile-time constants. */
  if (TOTAL_PORTS > 0 && reportPINs[0]) outputPort(0, readPort(0, portConfigInputs[0]), false);
  if (TOTAL_PORTS > 1 && reportPINs[1]) outputPort(1, readPort(1, portConfigInputs[1]), false);
  if (TOTAL_PORTS > 2 && reportPINs[2]) outputPort(2, readPort(2, portConfigInputs[2]), false);
  if (TOTAL_PORTS > 3 && reportPINs[3]) outputPort(3, readPort(3, portConfigInputs[3]), false);
  if (TOTAL_PORTS > 4 && reportPINs[4]) outputPort(4, readPort(4, portConfigInputs[4]), false);
  if (TOTAL_PORTS > 5 && reportPINs[5]) outputPort(5, readPort(5, portConfigInputs[5]), false);
  if (TOTAL_PORTS > 6 && reportPINs[6]) outputPort(6, readPort(6, portConfigInputs[6]), false);
  if (TOTAL_PORTS > 7 && reportPINs[7]) outputPort(7, readPort(7, portConfigInputs[7]), false);
  if (TOTAL_PORTS > 8 && reportPINs[8]) outputPort(8, readPort(8, portConfigInputs[8]), false);
  if (TOTAL_PORTS > 9 && reportPINs[9]) outputPort(9, readPort(9, portConfigInputs[9]), false);
  if (TOTAL_PORTS > 10 && reportPINs[10]) outputPort(10, readPort(10, portConfigInputs[10]), false);
  if (TOTAL_PORTS > 11 && reportPINs[11]) outputPort(11, readPort(11, portConfigInputs[11]), false);
  if (TOTAL_PORTS > 12 && reportPINs[12]) outputPort(12, readPort(12, portConfigInputs[12]), false);
  if (TOTAL_PORTS > 13 && reportPINs[13]) outputPort(13, readPort(13, portConfigInputs[13]), false);
  if (TOTAL_PORTS > 14 && reportPINs[14]) outputPort(14, readPort(14, portConfigInputs[14]), false);
  if (TOTAL_PORTS > 15 && reportPINs[15]) outputPort(15, readPort(15, portConfigInputs[15]), false);
}

// -----------------------------------------------------------------------------
/* sets the pin mode to the correct state and sets the relevant bits in the
 * two bit-arrays that track Digital I/O and PWM status
 */
void setPinModeCallback(byte pin, int mode)
{
  if (Firmata.getPinMode(pin) == PIN_MODE_IGNORE)
    return;

  if (Firmata.getPinMode(pin) == PIN_MODE_I2C && isI2CEnabled && mode != PIN_MODE_I2C) {
    // disable i2c so pins can be used for other functions
    // the following if statements should reconfigure the pins properly
    disableI2CPins();
  }
  if (IS_PIN_DIGITAL(pin) && mode != PIN_MODE_SERVO) {
    if (servoPinMap[pin] < MAX_SERVOS && servos[servoPinMap[pin]].attached()) {
      detachServo(pin);
    }
  }
  if (IS_PIN_ANALOG(pin)) {
    reportAnalogCallback(PIN_TO_ANALOG(pin), mode == PIN_MODE_ANALOG ? 1 : 0); // turn on/off reporting
  }
  if (IS_PIN_DIGITAL(pin)) {
    if (mode == INPUT || mode == PIN_MODE_PULLUP) {
      portConfigInputs[pin / 8] |= (1 << (pin & 7));
    } else {
      portConfigInputs[pin / 8] &= ~(1 << (pin & 7));
    }
  }
  Firmata.setPinState(pin, 0);
  switch (mode) {
    case PIN_MODE_ANALOG:
      if (IS_PIN_ANALOG(pin)) {
        if (IS_PIN_DIGITAL(pin)) {
          pinMode(PIN_TO_DIGITAL(pin), INPUT);    // disable output driver
#if ARDUINO <= 100
          // deprecated since Arduino 1.0.1 - TODO: drop support in Firmata 2.6
          digitalWrite(PIN_TO_DIGITAL(pin), LOW); // disable internal pull-ups
#endif
        }
        Firmata.setPinMode(pin, PIN_MODE_ANALOG);
      }
      break;
    case INPUT:
      if (IS_PIN_DIGITAL(pin)) {
        pinMode(PIN_TO_DIGITAL(pin), INPUT);    // disable output driver
#if ARDUINO <= 100
        // deprecated since Arduino 1.0.1 - TODO: drop support in Firmata 2.6
        digitalWrite(PIN_TO_DIGITAL(pin), LOW); // disable internal pull-ups
#endif
        Firmata.setPinMode(pin, INPUT);
      }
      break;
    case PIN_MODE_PULLUP:
      if (IS_PIN_DIGITAL(pin)) {
        pinMode(PIN_TO_DIGITAL(pin), INPUT_PULLUP);
        Firmata.setPinMode(pin, PIN_MODE_PULLUP);
        Firmata.setPinState(pin, 1);
      }
      break;
```

```
    case OUTPUT:
      if (IS_PIN_DIGITAL(pin)) {
        if (Firmata.getPinMode(pin) == PIN_MODE_PWM) {
          // Disable PWM if pin mode was previously set to PWM.
          digitalWrite(PIN_TO_DIGITAL(pin), LOW);
        }
        pinMode(PIN_TO_DIGITAL(pin), OUTPUT);
        Firmata.setPinMode(pin, OUTPUT);
      }
      break;
    case PIN_MODE_PWM:
      if (IS_PIN_PWM(pin)) {
        pinMode(PIN_TO_PWM(pin), OUTPUT);
        analogWrite(PIN_TO_PWM(pin), 0);
        Firmata.setPinMode(pin, PIN_MODE_PWM);
      }
      break;
    case PIN_MODE_SERVO:
      if (IS_PIN_DIGITAL(pin)) {
        Firmata.setPinMode(pin, PIN_MODE_SERVO);
        if (servoPinMap[pin] == 255 || !servos[servoPinMap[pin]].attached()) {
          // pass -1 for min and max pulse values to use default values set
          // by Servo library
          attachServo(pin, -1, -1);
        }
      }
      break;
    case PIN_MODE_I2C:
      if (IS_PIN_I2C(pin)) {
        // mark the pin as i2c
        // the user must call I2C_CONFIG to enable I2C for a device
        Firmata.setPinMode(pin, PIN_MODE_I2C);
      }
      break;
    case PIN_MODE_SERIAL:
#ifdef FIRMATA_SERIAL_FEATURE
      serialFeature.handlePinMode(pin, PIN_MODE_SERIAL);
#endif
      break;
    default:
      Firmata.sendString("Unknown pin mode"); // TODO: put error msgs in EEPROM
  }
  // TODO: save status to EEPROM here, if changed
}

/*
 * Sets the value of an individual pin. Useful if you want to set a pin value but
 * are not tracking the digital port state.
 * Can only be used on pins configured as OUTPUT.
 * Cannot be used to enable pull-ups on Digital INPUT pins.
 */
void setPinValueCallback(byte pin, int value)
{
  if (pin < TOTAL_PINS && IS_PIN_DIGITAL(pin)) {
    if (Firmata.getPinMode(pin) == OUTPUT) {
      Firmata.setPinState(pin, value);
      digitalWrite(PIN_TO_DIGITAL(pin), value);
    }
  }
}

void analogWriteCallback(byte pin, int value)
{
  if (pin < TOTAL_PINS) {
    switch (Firmata.getPinMode(pin)) {
      case PIN_MODE_SERVO:
        if (IS_PIN_DIGITAL(pin))
          servos[servoPinMap[pin]].write(value);
        Firmata.setPinState(pin, value);
        break;
      case PIN_MODE_PWM:
        if (IS_PIN_PWM(pin))
          analogWrite(PIN_TO_PWM(pin), value);
        Firmata.setPinState(pin, value);
        break;
    }
  }
}

void digitalWriteCallback(byte port, int value)
{
```

```
  byte pin, lastPin, pinValue, mask = 1, pinWriteMask = 0;

  if (port < TOTAL_PORTS) {
    // create a mask of the pins on this port that are writable.
    lastPin = port * 8 + 8;
    if (lastPin > TOTAL_PINS) lastPin = TOTAL_PINS;
    for (pin = port * 8; pin < lastPin; pin++) {
      // do not disturb non-digital pins (eg, Rx & Tx)
      if (IS_PIN_DIGITAL(pin)) {
        // do not touch pins in PWM, ANALOG, SERVO or other modes
        if (Firmata.getPinMode(pin) == OUTPUT || Firmata.getPinMode(pin) == INPUT) {
          pinValue = ((byte)value & mask) ? 1 : 0;
          if (Firmata.getPinMode(pin) == OUTPUT) {
            pinWriteMask |= mask;
          } else if (Firmata.getPinMode(pin) == INPUT && pinValue == 1 && Firmata.getPinState(pin) != 1) {
            // only handle INPUT here for backwards compatibility
#if ARDUINO > 100
            pinMode(pin, INPUT_PULLUP);
#else
            // only write to the INPUT pin to enable pullups if Arduino v1.0.0 or earlier
            pinWriteMask |= mask;
#endif
          }
          Firmata.setPinState(pin, pinValue);
        }
      }
      mask = mask << 1;
    }
    writePort(port, (byte)value, pinWriteMask);
  }
}


// -----------------------------------------------------------------------------
/* sets bits in a bit array (int) to toggle the reporting of the analogIns
 */
//void FirmataClass::setAnalogPinReporting(byte pin, byte state) {
//}
void reportAnalogCallback(byte analogPin, int value)
{
  if (analogPin < TOTAL_ANALOG_PINS) {
    if (value == 0) {
      analogInputsToReport = analogInputsToReport & ~ (1 << analogPin);
    } else {
      analogInputsToReport = analogInputsToReport | (1 << analogPin);
      // prevent during system reset or all analog pin values will be reported
      // which may report noise for unconnected analog pins
      if (!isResetting) {
        // Send pin value immediately. This is helpful when connected via
        // ethernet, wi-fi or bluetooth so pin states can be known upon
        // reconnecting.
        Firmata.sendAnalog(analogPin, analogRead(analogPin));
      }
    }
  }
  // TODO: save status to EEPROM here, if changed
}

void reportDigitalCallback(byte port, int value)
{
  if (port < TOTAL_PORTS) {
    reportPINs[port] = (byte)value;
    // Send port value immediately. This is helpful when connected via
    // ethernet, wi-fi or bluetooth so pin states can be known upon
    // reconnecting.
    if (value) outputPort(port, readPort(port, portConfigInputs[port]), true);
  }
  // do not disable analog reporting on these 8 pins, to allow some
  // pins used for digital, others analog.  Instead, allow both types
  // of reporting to be enabled, but check if the pin is configured
  // as analog when sampling the analog inputs.  Likewise, while
  // scanning digital pins, portConfigInputs will mask off values from any
  // pins configured as analog
}

/*==============================================================================
 * SYSEX-BASED commands
 *============================================================================*/

void sysexCallback(byte command, byte argc, byte *argv)
{
```

```
byte mode;
byte stopTX;
byte slaveAddress;
byte data;
int slaveRegister;
unsigned int delayTime;

switch (command) {
  case I2C_REQUEST:
    mode = argv[1] & I2C_READ_WRITE_MODE_MASK;
    if (argv[1] & I2C_10BIT_ADDRESS_MODE_MASK) {
      Firmata.sendString("10-bit addressing not supported");
      return;
    }
    else {
      slaveAddress = argv[0];
    }

    // need to invert the logic here since 0 will be default for client
    // libraries that have not updated to add support for restart tx
    if (argv[1] & I2C_END_TX_MASK) {
      stopTX = I2C_RESTART_TX;
    }
    else {
      stopTX = I2C_STOP_TX; // default
    }

    switch (mode) {
      case I2C_WRITE:
        Wire.beginTransmission(slaveAddress);
        for (byte i = 2; i < argc; i += 2) {
          data = argv[i] + (argv[i + 1] << 7);
          wireWrite(data);
        }
        Wire.endTransmission();
        delayMicroseconds(70);
        break;
      case I2C_READ:
        if (argc == 6) {
          // a slave register is specified
          slaveRegister = argv[2] + (argv[3] << 7);
          data = argv[4] + (argv[5] << 7);  // bytes to read
        }
        else {
          // a slave register is NOT specified
          slaveRegister = I2C_REGISTER_NOT_SPECIFIED;
          data = argv[2] + (argv[3] << 7);  // bytes to read
        }
        readAndReportData(slaveAddress, (int)slaveRegister, data, stopTX);
        break;
      case I2C_READ_CONTINUOUSLY:
        if ((queryIndex + 1) >= I2C_MAX_QUERIES) {
          // too many queries, just ignore
          Firmata.sendString("too many queries");
          break;
        }
        if (argc == 6) {
          // a slave register is specified
          slaveRegister = argv[2] + (argv[3] << 7);
          data = argv[4] + (argv[5] << 7);  // bytes to read
        }
        else {
          // a slave register is NOT specified
          slaveRegister = (int)I2C_REGISTER_NOT_SPECIFIED;
          data = argv[2] + (argv[3] << 7);  // bytes to read
        }
        queryIndex++;
        query[queryIndex].addr = slaveAddress;
        query[queryIndex].reg = slaveRegister;
        query[queryIndex].bytes = data;
        query[queryIndex].stopTX = stopTX;
        break;
      case I2C_STOP_READING:
        byte queryIndexToSkip;
        // if read continuous mode is enabled for only 1 i2c device, disable
        // read continuous reporting for that device
        if (queryIndex <= 0) {
          queryIndex = -1;
        } else {
          queryIndexToSkip = 0;
          // if read continuous mode is enabled for multiple devices,
```

```
        // determine which device to stop reading and remove it's data from
        // the array, shifiting other array data to fill the space
        for (byte i = 0; i < queryIndex + 1; i++) {
          if (query[i].addr == slaveAddress) {
            queryIndexToSkip = i;
            break;
          }
        }

        for (byte i = queryIndexToSkip; i < queryIndex + 1; i++) {
          if (i < I2C_MAX_QUERIES) {
            query[i].addr = query[i + 1].addr;
            query[i].reg = query[i + 1].reg;
            query[i].bytes = query[i + 1].bytes;
            query[i].stopTX = query[i + 1].stopTX;
          }
        }
        queryIndex--;
      }
      break;
    default:
      break;
  }
  break;
case I2C_CONFIG:
  delayTime = (argv[0] + (argv[1] << 7));

  if (argc > 1 && delayTime > 0) {
    i2cReadDelayTime = delayTime;
  }

  if (!isI2CEnabled) {
    enableI2CPins();
  }

  break;
case SERVO_CONFIG:
  if (argc > 4) {
    // these vars are here for clarity, they'll optimized away by the compiler
    byte pin = argv[0];
    int minPulse = argv[1] + (argv[2] << 7);
    int maxPulse = argv[3] + (argv[4] << 7);

    if (IS_PIN_DIGITAL(pin)) {
      if (servoPinMap[pin] < MAX_SERVOS && servos[servoPinMap[pin]].attached()) {
        detachServo(pin);
      }
      attachServo(pin, minPulse, maxPulse);
      setPinModeCallback(pin, PIN_MODE_SERVO);
    }
  }
  break;
case SAMPLING_INTERVAL:
  if (argc > 1) {
    samplingInterval = argv[0] + (argv[1] << 7);
    if (samplingInterval < MINIMUM_SAMPLING_INTERVAL) {
      samplingInterval = MINIMUM_SAMPLING_INTERVAL;
    }
  } else {
    //Firmata.sendString("Not enough data");
  }
  break;
case EXTENDED_ANALOG:
  if (argc > 1) {
    int val = argv[1];
    if (argc > 2) val |= (argv[2] << 7);
    if (argc > 3) val |= (argv[3] << 14);
    analogWriteCallback(argv[0], val);
  }
  break;
case CAPABILITY_QUERY:
  Firmata.write(START_SYSEX);
  Firmata.write(CAPABILITY_RESPONSE);
  for (byte pin = 0; pin < TOTAL_PINS; pin++) {
    if (IS_PIN_DIGITAL(pin)) {
      Firmata.write((byte)INPUT);
      Firmata.write(1);
      Firmata.write((byte)PIN_MODE_PULLUP);
      Firmata.write(1);
      Firmata.write((byte)OUTPUT);
      Firmata.write(1);
```

```
        }
        if (IS_PIN_ANALOG(pin)) {
          Firmata.write(PIN_MODE_ANALOG);
          Firmata.write(10); // 10 = 10-bit resolution
        }
        if (IS_PIN_PWM(pin)) {
          Firmata.write(PIN_MODE_PWM);
          Firmata.write(DEFAULT_PWM_RESOLUTION);
        }
        if (IS_PIN_DIGITAL(pin)) {
          Firmata.write(PIN_MODE_SERVO);
          Firmata.write(14);
        }
        if (IS_PIN_I2C(pin)) {
          Firmata.write(PIN_MODE_I2C);
          Firmata.write(1);  // TODO: could assign a number to map to SCL or SDA
        }
#ifdef FIRMATA_SERIAL_FEATURE
        serialFeature.handleCapability(pin);
#endif
        Firmata.write(127);
      }
      Firmata.write(END_SYSEX);
      break;
    case PIN_STATE_QUERY:
      if (argc > 0) {
        byte pin = argv[0];
        Firmata.write(START_SYSEX);
        Firmata.write(PIN_STATE_RESPONSE);
        Firmata.write(pin);
        if (pin < TOTAL_PINS) {
          Firmata.write(Firmata.getPinMode(pin));
          Firmata.write((byte)Firmata.getPinState(pin) & 0x7F);
          if (Firmata.getPinState(pin) & 0xFF80) Firmata.write((byte)(Firmata.getPinState(pin) >> 7) & 0x7F);
          if (Firmata.getPinState(pin) & 0xC000) Firmata.write((byte)(Firmata.getPinState(pin) >> 14) & 0x7F);
        }
        Firmata.write(END_SYSEX);
      }
      break;
    case ANALOG_MAPPING_QUERY:
      Firmata.write(START_SYSEX);
      Firmata.write(ANALOG_MAPPING_RESPONSE);
      for (byte pin = 0; pin < TOTAL_PINS; pin++) {
        Firmata.write(IS_PIN_ANALOG(pin) ? PIN_TO_ANALOG(pin) : 127);
      }
      Firmata.write(END_SYSEX);
      break;

    case SERIAL_MESSAGE:
#ifdef FIRMATA_SERIAL_FEATURE
      serialFeature.handleSysex(command, argc, argv);
#endif
      break;
  }
}

/*==============================================================================
 * SETUP()
 *============================================================================*/

void systemResetCallback()
{
  isResetting = true;

  // initialize a defalt state
  // TODO: option to load config from EEPROM instead of default

#ifdef FIRMATA_SERIAL_FEATURE
  serialFeature.reset();
#endif

  if (isI2CEnabled) {
    disableI2CPins();
  }

  for (byte i = 0; i < TOTAL_PORTS; i++) {
    reportPINs[i] = false;    // by default, reporting off
    portConfigInputs[i] = 0;  // until activated
    previousPINs[i] = 0;
  }
```

```
  for (byte i = 0; i < TOTAL_PINS; i++) {
    // pins with analog capability default to analog input
    // otherwise, pins default to digital output
    if (IS_PIN_ANALOG(i)) {
      // turns off pullup, configures everything
      setPinModeCallback(i, PIN_MODE_ANALOG);
    } else if (IS_PIN_DIGITAL(i)) {
      // sets the output to 0, configures portConfigInputs
      setPinModeCallback(i, OUTPUT);
    }

    servoPinMap[i] = 255;
  }
  // by default, do not report any analog inputs
  analogInputsToReport = 0;

  detachedServoCount = 0;
  servoCount = 0;

  /* send digital inputs to set the initial state on the host computer,
   * since once in the loop(), this firmware will only send on change */
  /*
  TODO: this can never execute, since no pins default to digital input
        but it will be needed when/if we support EEPROM stored config
  for (byte i=0; i < TOTAL_PORTS; i++) {
    outputPort(i, readPort(i, portConfigInputs[i]), true);
  }
  */
  isResetting = false;
}

void setup()
{
  Firmata.setFirmwareVersion(FIRMATA_FIRMWARE_MAJOR_VERSION, FIRMATA_FIRMWARE_MINOR_VERSION);

  Firmata.attach(ANALOG_MESSAGE, analogWriteCallback);
  Firmata.attach(DIGITAL_MESSAGE, digitalWriteCallback);
  Firmata.attach(REPORT_ANALOG, reportAnalogCallback);
  Firmata.attach(REPORT_DIGITAL, reportDigitalCallback);
  Firmata.attach(SET_PIN_MODE, setPinModeCallback);
  Firmata.attach(SET_DIGITAL_PIN_VALUE, setPinValueCallback);
  Firmata.attach(START_SYSEX, sysexCallback);
  Firmata.attach(SYSTEM_RESET, systemResetCallback);

  // to use a port other than Serial, such as Serial1 on an Arduino Leonardo or Mega,
  // Call begin(baud) on the alternate serial port and pass it to Firmata to begin like this:
  // Serial1.begin(57600);
  // Firmata.begin(Serial1);
  // However do not do this if you are using SERIAL_MESSAGE

  Firmata.begin(57600);
  while (!Serial) {
    ; // wait for serial port to connect. Needed for ATmega32u4-based boards and Arduino 101
  }

  systemResetCallback();  // reset to default config
}

/*==============================================================================
 * LOOP()
 *============================================================================*/
void loop()
{
  byte pin, analogPin;

  /* DIGITALREAD - as fast as possible, check for changes and output them to the
   * FTDI buffer using Serial.print()  */
  checkDigitalInputs();

  /* STREAMREAD - processing incoming messagse as soon as possible, while still
   * checking digital inputs.  */
  while (Firmata.available())
    Firmata.processInput();

  // TODO - ensure that Stream buffer doesn't go over 60 bytes

  currentMillis = millis();
  if (currentMillis - previousMillis > samplingInterval) {
    previousMillis += samplingInterval;
    /* ANALOGREAD - do all analogReads() at the configured sampling interval */
    for (pin = 0; pin < TOTAL_PINS; pin++) {
```

```
      if (IS_PIN_ANALOG(pin) && Firmata.getPinMode(pin) == PIN_MODE_ANALOG) {
        analogPin = PIN_TO_ANALOG(pin);
        if (analogInputsToReport & (1 << analogPin)) {
          Firmata.sendAnalog(analogPin, analogRead(analogPin));
        }
      }
    }
    // report i2c data for all device with read continuous mode enabled
    if (queryIndex > -1) {
      for (byte i = 0; i < queryIndex + 1; i++) {
        readAndReportData(query[i].addr, query[i].reg, query[i].bytes, query[i].stopTX);
      }
    }
  }

#ifdef FIRMATA_SERIAL_FEATURE
  serialFeature.update();
#endif
}
```

Código Python – Visual Studio Code (Logo Azul):

```
# -*- coding: utf-8 -*-


#En Python se introducen comentarios de una sola linea con el simbolo #.
#La primera línea de código incluida en este programa se conoce como declaración de codificación o codificación
#de caracteres. Al especificar utf-8 (caracteres Unicode) como la codificación, nos aseguramos de que el archivo
#pueda contener caracteres especiales, letras acentuadas y otros caracteres no ASCII sin problemas, garantizando
#que Python interprete correctamente esos caracteres y evite posibles errores de codificación.
#Se puede detener una ejecución con el comando [CTRL] + C puesto en consola, con el comando "cls" se borra su
#historial y en Visual Studio Code con el botón superior izquierdo de Play se corre el programa.
#Para comentar en Visual Studio Code varias líneas de código se debe pulsar:
#[CTRL] + K (VSCode queda a la espera). Después pulsa [CTRL] + C para comentar y [CTRL] + U para descomentar.

#INSTRUMENTACIÓN VIRTUAL CON PYFIRMATA Y ARDUINO:
#Para que se pueda realizar la instrumentación virtual con una tarjeta de desarrollo Arduino, primero se debe
#correr en su IDE (editor de código) el programa 21.-Instrumentacion_Virtual_Mandar_Recibir_Datos_Arduino.ino,
#donde se encuentra la librería Standard Firmata, además se indica de esta manera cuál es el puerto de conexión
#serial entre la placa de desarrollo Arduino y el ordenador.
#La función de la biblioteca Standard Firmata de Arduino y la librería pyfirmata es permitir el intercambio
#bidireccional entre la placa de desarrollo y el ordenador, permitiendo así su control y monitoreo. Para ello,
#el pin de salida donde se hará parpadear un led y el pin de entrada analógico con el cual se reciben y grafican
#sus niveles de tensión son los siguientes:
#   - Puerto de salida digital donde parpadea un led = 13.
#   - Puerto de entrada analógico que recibe niveles de tensión = A0.
#Si la tarjeta de desarrollo Arduino no se encuentra conectada al ordenador, el programa arrojará un error.
#El código completo de Arduino se incluye comentado en la parte de abajo de este programa.

import matplotlib.pyplot as plt #matplotlib: Librería de graficación matemática.
import drawnow #drawnow: Librería para generar gráficos que se actualizan continuamente en tiempo real.
import time #time: Librería del manejo de tiempos, como retardos, contadores, etc.
#pyfirmata: Librería que permite la comunicación bidireccional entre Python y Arduino, brindando control y
```

```
#monitoreo de sus pines digitales y analógicos, envío de señales PWM, lectura y escritura de datos y establecer
#una comunicación a través de los protocolos I2C y Serial. La comunicación entre Arduino y Python se realiza
#utilizando el protocolo Firmata, el cual permite controlar y monitorear dispositivos conectados a una placa
#Arduino desde un software de computadora, para habilitarlo, primero se debe subir el programa:
#21.-Stardard_Firmata_Mandar_Datos_Arduino.ino a la placa de desarrollo a través del IDE de Arduino.
import pyfirmata

#VARIABLES:
t = []                #Lista que almacena los datos del eje horizontal (x = tiempo [s]) de la gráfica.
data = []             #Lista que almacena los datos del eje vertical (y = tensión [V]) de la gráfica.
temp_t = 0            #Variable que cuenta cada 0.5 segundos el tiempo de ejecución del programa.
samplingTime = 0.5    #Intervalo del conteo del objeto Time indicado en segundos.

serial_port = "COM7"  #Variable tipo string con el mismo puerto al que está conectado el Arduino en su IDE.

#PYFIRMATA: CONEXIÓN SERIAL, CONTROL Y MONITOREO DE PINES DE UNA TARJETA DE DESARROLLO
#Instancia de la clase Arduino, que pertenece a la librería pyfirmata, dicho objeto proporciona métodos para
#comunicarse con placas Arduino utilizando el protocolo Firmata, permitiendo así el control y monitoreo de sus
#pines digitales y analógicos, realizar el envío de señales PWM, lectura y/o escritura de datos y establecer una
#comunicación a través de los protocolos I2C y Serial, la conexión serial con los microcontroladores se realiza
#utilizando la clase pyfirmata.Arduino(), independientemente del tipo de microcontrolador que se esté
#utilizando, ya que la librería pyfirmata proporciona una interfaz común para comunicarse con diferentes placas
#de desarrollo, incluyendo Arduino, Raspberry Pi, Intel Galileo, PyCARD, etc. Para ello su constructor recibe
#los siguientes parámetros:
# - port (obligatorio): Especifica el puerto de comunicación a través del cual se conectará la placa Arduino.
#   Puede ser una cadena de texto que representa el nombre del puerto, como por ejemplo 'COM3' en Windows
#   o '/dev/ttyACM0' en Linux. El nombre del parámetro no se indica explícitamente.
# - timeout (opcional): Especifica el tiempo máximo de espera (en segundos) para establecer la conexión con la
#   placa Arduino. Si no se especifica, se utilizará un valor predeterminado.
# - baudrate: Este parámetro establece la velocidad de comunicación en baudios para la comunicación entre la
#   computadora y el microcontrolador. Los baudios representan la cantidad de bits que se pueden transmitir por
#   segundo.
#       - El valor que utiliza la librería Standard Firmata por default es de 57600, pero también se puede usar
#         otros valores como 9600, 115200, etc. Pero esto debería ser cambiado igual en el código Arduino de la
#         librería que se sube al Arduino.
# - bytesize, parity y stopbits (opcionales): Estos parámetros permiten configurar la transmisión serial y se
#   utilizan en conjunto para establecer cómo se transmiten los datos entre la computadora y la placa de
#   desarrollo.
board = pyfirmata.Arduino(serial_port, baudrate = 57600)   #Conexión serial.
#Instancia de la clase Iterator, que hereda de la clase util y ambas pertenecen a la librería pyfirmata, dicho
#objeto permite al programa percibir, capturar y procesar los cambios que ocurran en los pines de entrada de la
#placa, para ello su constructor recibe como parámetro un objeto pyfirmata.Board que ya haya realizado una
#conexión serial entre la computadora y la tarjeta de desarrollo.
it = pyfirmata.util.Iterator(board)
#pyfirmata.util.Iterator.start(): Método utilizado para iniciar el proceso de lectura de datos entrantes desde
```

```
#la placa de desarrollo previamente conectada al ordenador de forma serial con el constructor:
#pyfirmata.Arduino().
it.start()
#SELECCIÓN DE LOS PINES QUE SE QUIERE CONTROLAR Y/O MONITOREAR:
#Opción 1:
#pyfirmata.Arduino.digital[númeroPin].mode = pin_de_entrada_o_salida: Método utilizado para acceder a un pin
#digital específico, perteneciente a la placa de desarrollo con la que ya se ha realizado una conexión serial.
#Indicando después del signo igual si este es utilizado como entrada o salida por medio de las constantes OUTPUT
#o INPUT de la librería pyfirmata.
#pyfirmata.Arduino.analog[númeroPin].mode = pin_de_entrada_o_salida: Método utilizado para acceder a un pin
#analógico específico, perteneciente a la placa de desarrollo con la que ya se ha realizado una conexión serial.
#Indicando después del signo igual si este es utilizado como entrada o salida por medio de las constantes OUTPUT
#o INPUT de la librería pyfirmata.
#Opción 2:
#pyfirmata.Arduino.get_pin(): Método utilizado para acceder a un pin específico de la placa de desarrollo con la
#que ya se ha realizado una conexión serial. Indicando si este es analógico o digital, su número y si será
#utilizado como entrada o salida siguiendo la sintaxis descrita a continuación:
# - 'analogico_o_digital:   numero_pin:     entrada_o_salida'
#       - 'analogico:       numero_pin:     entrada'            = 'a: numero_pin: i'
#       - 'analogico:       numero_pin:     salida'             = 'a: numero_pin: o'
#       - 'digital:         numero_pin:     entrada'            = 'd: numero_pin: i'
#       - 'digital:         numero_pin:     salida'             = 'd: numero_pin: o'
#El número y asignación de pin analógico o digital varía dependiendo de la tarjeta de desarrollo.
board.digital[13].mode = pyfirmata.OUTPUT
board.analog[0].mode = pyfirmata.INPUT

#time.sleep(): Método que se utiliza para suspender la ejecución de un programa durante un intervalo de tiempo
#específico dado en segundos.
time.sleep(2)         #Delay de 2 segundos

#pyfirmata.Arduino.analog[númeroPin].enable_reporting(): Método utilizado para para habilitar el monitoreo y
#lectura de valores analógicos en un pin analógico específico de la placa Arduino.
board.analog[0].enable_reporting()


#GRÁFICA ÚNICA TIPO MATPLOTLIB:
#make_figure(): Función para graficar los datos recabados del Arduino, se declara dentro de una función propia
#la graficación de los datos ya que para que estos puedan ser actualizados en tiempo real, se utiliza el método
#drawnow(), perteneciente a la librería drawnow, que debe recibir esta función como su parámetro.
def make_figure():
    #matplotlib.title(): Método que indica indica el título de la ventana que muestra la gráfica dinámica.
    plt.title("Instrumentación Virtual Arduino - Pyfirmata")
    plt.ylim(-0.1, 6) #matplotlib.ylim(): Método que indica el rango del eje vertical en la gráfica.
    #matplotlib.grid(): Método que recibe un valor booleano para indicar si aparece una rejilla o no en la
    #gráfica, por default está en valor False.
```

```
    plt.grid(True)
    plt.xlabel("Tiempo (s)") #matplotlib.xlabel(): Método que indica el texto que aparece en el eje horizontal.
    plt.ylabel("Tensión (V))") #matplotlib.ylabel(): Método que indica el texto que aparece en el eje vertical.
    #matplotlib.plot(): Método usado para crear la gráfica, indicando como primer parámetro su eje horizontal,
    #luego su eje vertical y finalmente el estilo de la gráfica.
    # - Colores:             C1: color naranja, r: color rojo, b: color azul, g: verde, c: cyan, m: morado,
    #   y: amarillo, k: negro, w: blanco.
    # - Tipo de marcadores:  o: círculos, +: símbolos de más, .; puntos, v: Triángulo hacia abajo, h: Hexágono,
    #   s: cuadrados, etc.
    # - Tipo de Líneas:      -: sólida, --: punteada (líneas), :: punteada (puntos), -.: línea y punto,
    #   'or': Nada.
    plt.plot(t,data,'ch--')
#matplotlib.show(): Método para mostrar la gráfica creada.
plt.show()


#EJECUCIÓN DE LA GRÁFICA:
N = 50        #Variable que indica el límite de datos recopilados, el límite de Excel es de 32,000 datos.
print("El programa recopilará ", N, " datos.")
for i in range(N):
    #La operación % significa módulo y lo que hace esta es dividir un número y ver el resultado de su residuo.
    #En la operación 3%2 == 0, lo que está haciendo es dividir 3/2 y ver si su residuo es cero, que en este caso
    #no lo sería, ya que residuo = 1.
    # - n%2 == 0: La operación verifica si el valor de n es divisible por 2 sin dejar residuo. En otras
    #   palabras, verifica si n es un número par.
    if (i%2 == 0):
        #pyfirmata.Arduino.digital[númeroPin].write(): Método que se utiliza para escribir un valor en algún pin
        #digital de la placa, controlando si su estado está en alto (1 o True) o bajo (0 o False).
        # - Encender LED:   write(True)  o write(1).
        # - Apagar LED:     write(False) o write(0).
        #El método write() solamente se puede usar con pines de salida digital, si se quiere controlar la salida
        #de un pin analógico se debe usa el método analog_write().
        board.digital[13].write(1)
    else:
        board.digital[13].write(0)
    if (i == 0):
        print("Tiempo [s]\tTensión [V]")
    #MANEJO DE EXCEPCIONES: Es una parte de código que se conforma de dos partes, try y except:
    # - Primero se ejecuta el código que haya dentro del try y si es que llegara a ocurrir una excepción durante
    #   su ejecución, el programa brinca al código del except
    # - En la parte de código donde se encuentra la palabra reservada except, se ejecuta cierta acción cuando
    #   ocurra el error.
    #Se utiliza esta arquitectura de código cuando se quiera efectuar una acción donde se espera que pueda
    #ocurrir un error durante su ejecución.
    try:
```

```
#pyfirmata.Arduino.analog[númeroPin].read(): Método que se utiliza para leer el valor actual de un pin
#en la placa. Permite obtener el estado del pin, que puede ser un valor analógico o digital, dependiendo
#de cómo esté configurado.
# - Pin digital de entrada: Devolverá un valor booleano (True o False), que indica si el pin está en
#   alto (encendido) o en bajo (apagado) respectivamente.
# - Pin analógico de entrada: Devolverá un valor numérico que representa la lectura analógica en el
#   pin.
#       - Placa Arduino: Este valor suele estar en un rango de 0 a 1, donde 0 coresponde al valor
#         binario 0 y 1 al valor binario 1023, que hace referencia a la resolución de 10 bits del
#         conversor analógico a digital (ADC) en la placa Arduino:
#               - Resolución de 10 bits del ADC del Arduino: (2^10)-1 = 1023
#Si se utiliza el método read() en un pin que está configurado como salida, el resultado puede ser
#impredecible o no tener sentido.
tensionBinariaA0 = board.analog[0].read()
print("Valor decimal obtenido del ADC de 10 bits del Arduino:       ", tensionBinariaA0)

#CONVERSIÓN DE NUMEROS BINARIOS NUMÉRICOS DE TENSIÓN A VALORES DE TENSIÓN REALES:
#float(): Método que convierte un tipo de dato cualquiera en númerico decimal.
#Se realiza esta operación porque como el ADC del arduino lee de 0 a 5V y como tiene una resolución de
#10 bits permitiendo que en el ADC los valores de tensión se interpreten como valores numéricos enteros
#que valen de 0 a (2^10)-1 = 1023, se hace una regla de 3 para que se imprima el valor de la tensión en
#consola en vez del valor decimal binario. En esta operación no es necesario dividir el resultado entre
#la resolución del ADC porque el valor que retorna el método read ya está normalizado en el rango de 0 a
#1 (correspondiente a los 1024 niveles).
#Tensión = Tensión_decimal_read*(ValorMáximoTensión) = Tensión_decimal_read*(5)
highValueBoard = 5.0            #Valor de tensión Máxima = 5V.
VoltsA0 = float(tensionBinariaA0)*highValueBoard
print(i, "-. Valor real de tensión:                                ", VoltsA0, "[V]")
#append(): Método que sirve para agregar valores a una lista, tupla, numpy array o diccionario.
data.append(VoltsA0)            #Creación del vector tensión (voltaje).

#VECTOR TIEMPO:
#Se usa una variable intermedia que va contando el tiempo transcurrido desde que se empezó a recopilar
#los valores de tensión del puerto analógico A0 del Arduino hasta que acaba. El intervalo de tiempo con
#el que cuenta el temporizador y el tiempo que se detiene delay que se declarará después del except debe
#ser el mismo.
temp_t = temp_t + samplingTime  #Variable intermedia que cuenta el tiempo de ejecución del programa.
#append(): Método que sirve para agregar valores a una lista, tupla, numpy array o diccionario.
t.append(temp_t)                #Creación del vector tiempo.
print("%.1f \t %g"%(temp_t, VoltsA0))

#drawnow.drawnow(): Método que permite actualizar y redibujar una figura o gráfico en tiempo real. Se
#utiliza comúnmente junto con la librería matplotlib para crear visualizaciones interactivas que se
#actualizan dinámicamente y se utiliza en conjunto con una función de graficación personalizada.
#Esta función de trazado define cómo se representan los datos en el gráfico.
```

```
        drawnow.drawnow(make_figure)    #Actualización dinámica de la gráfica.
    except:
        #print(): Método para imprimir un mensaje en consola y después dar un salto de línea (Enter).
        print("Chafeó la medida")

    #time.sleep(): Método que se utiliza para suspender la ejecución de un programa durante un intervalo de
    #tiempo específico dado en segundos.
    time.sleep(samplingTime)            #Delay de 0.5 segundos

#pyfirmata.Arduino.analog[númeroPin].disable_reporting(): Método utilizado para para desactivar el monitoreo y
#lectura de valores analógicos en un pin analógico específico de la placa Arduino.
board.analog[0].disable_reporting()
#pyfirmata.Arduino.digital[númeroPin].write(): Método que se utiliza para escribir un valor en algún pin digital
#de la placa, controlando si su estado está en alto (1 o True) o bajo (0 o False).
# - Encender LED:   write(True)  o write(1).
# - Apagar LED:     write(False) o write(0).
#El método write() solamente se puede usar con pines de salida digital, si se quiere controlar la salida de un
#pin analógico se debe usa el método analog_write().
board.digital[13].write(0)
#pyserial.Arduino.exit(): Método que cierra la comunicación serial. Es muy importante mencionar que si no se
#ejecuta este método, el puerto serial se va a quedar bloqueado y no se podrá usar.
board.exit()
#matplotlib.close(): Método que cierra la gráfica previamente abierta con el método matplotlib.show().
plt.close()
print("Se recopilaron correctamente ", N, " datos.\n")
print(data)


#GUARDAR LOS DATOS RECOPILADOS EN UN ARCHIVO:
filename =
"0.-Archivos_Ejercicios_Python/21.-Instrumentacion_Virtual_Mandar_Recibir_Datos_Arduino/DatosArduinoPyfirmata.csv"
#open(): Método que sirve para abrir un archivo cualquiera, para ello es necesario indicar dos parámetros, el
#primero se refiere a la ruta relativa o absoluta del archivo previamente creado y la segunda indica qué es lo
#que se va a realizar con él, el contenido del archivo se asigna a una variable.
# - w: Sirve para escribir en un archivo, pero borrará la información que previamente contenía el archivo.
# - a: Sirve para escribir en un archivo sin que se borre la info anterior del archivo, se llama append.
new_file = open(filename,'w')
#file.write(): Método que sirve para escribir una palabra o string en un archivo.
new_file.write("Library, Pyfirmata" + "\n")
new_file.write("Time [s], Voltage Pin A0 [V]" + "\n")
for i in range(len(data)):
    #Lista que guarda los valores de tiempo [s] y tensión [V] recopilados.
    new_file.write(str(t[i]) + "," + str(data[i]) + "\n")
#file.close(): Método para cerrar un archivo previamente abrierto con el método open(), es peligroso olvidar
#colocar este método, ya que la computadora lo considerará como si nunca hubiera sido cerrado, por lo cual no
```

```
#podré volver a abrirlo al dar clic sobre él.

new_file.close()
```

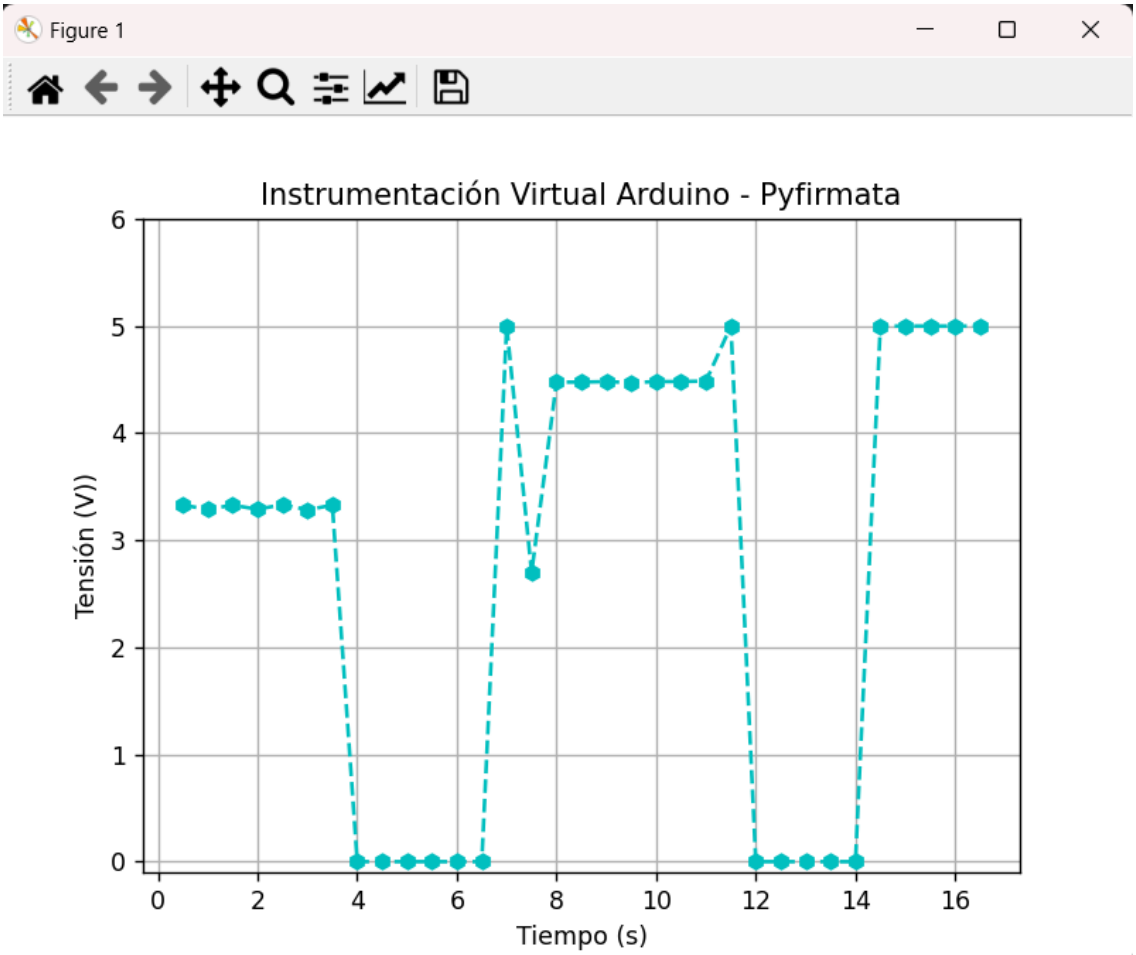
Resultado del Código Python

```
PROBLEMAS   SALIDA   TERMINAL   CONSOLA DE DEPURACIÓN

Se recopilaron correctamente  50  datos.
  File "D:\Users\diego\AppData\Local\Programs\Python\Python39\lib\threading.py", line 973, in _bootstrap_inner

[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 2.6635, 4.477, 4.482, 4.477, 4.482, 4.477, 4.355, 5.0, 0.0, 0.0, 0.0, 0.0, 0.0, 5.0, 5.0, 5.0, 5.0, 5.
0, 3.5925000000000002, 0.0, 0.0, 0.0, 0.0, 4.7215, 3.2845000000000004, 3.3285, 3.2845000000000004, 3.3285, 3.2845000000000004, 3.3285
, 3.2845000000000004, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

Python
Python
Python
Python
Python
Lín. 110, col. 1   Espacios: 4   UTF-8   CRLF   Python   3.9.7 64-bit
```




```
Valor decimal obtenido del ADC de 10 bits del Arduino:        1.0
30 -. Valor real de tensión:                                  5.0 [V]
15.0     5
Valor decimal obtenido del ADC de 10 bits del Arduino:        1.0
31 -. Valor real de tensión:                                  5.0 [V]
15.5     5
Valor decimal obtenido del ADC de 10 bits del Arduino:        1.0
32 -. Valor real de tensión:                                  5.0 [V]
16.0     5
Valor decimal obtenido del ADC de 10 bits del Arduino:        1.0
33 -. Valor real de tensión:                                  5.0 [V]
16.5     5
```