# Anexo de códigos

En este anexo se adjuntaron los códigos más relevantes, tanto del lado del cliente como del servidor, que se desarrollaron para el funcionamiento del SAR.

## **Códigos del lado del servidor**

### Código StandarFirmata utilizado en el Arduino MEGA

/\*

Firmata is a generic protocol for communicating with microcontrollers

from software on a host computer. It is intended to work with

any host computer software package.

To download a host software package, please click on the following link

to open the list of Firmata client libraries in your default browser.

https://github.com/firmata/arduino#firmata-client-libraries

Copyright (C) 2006-2008 Hans-Christoph Steiner. All rights reserved.

Copyright (C) 2010-2011 Paul Stoffregen. All rights reserved.

Copyright (C) 2009 Shigeru Kobayashi. All rights reserved.

Copyright (C) 2009-2016 Jeff Hoefs. All rights reserved.

This library is free software; you can redistribute it and/or

modify it under the terms of the GNU Lesser General Public

License as published by the Free Software Foundation; either

version 2.1 of the License, or (at your option) any later version.

See file LICENSE.txt for further informations on licensing terms.

Last updated October 16th, 2016

\*/

#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 (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 ConfigurableFirmata utilizado en el Arduino NANO

/\*

Firmata is a generic protocol for communicating with microcontrollers

from software on a host computer. It is intended to work with

any host computer software package.

To download a host software package, please clink on the following link

to open the download page in your default browser.

https://github.com/firmata/ConfigurableFirmata#firmata-client-libraries

Copyright (C) 2006-2008 Hans-Christoph Steiner. All rights reserved.

Copyright (C) 2010-2011 Paul Stoffregen. All rights reserved.

Copyright (C) 2009 Shigeru Kobayashi. All rights reserved.

Copyright (C) 2013 Norbert Truchsess. All rights reserved.

Copyright (C) 2014 Nicolas Panel. All rights reserved.

Copyright (C) 2009-2017 Jeff Hoefs. All rights reserved.

This library is free software; you can redistribute it and/or

modify it under the terms of the GNU Lesser General Public

License as published by the Free Software Foundation; either

version 2.1 of the License, or (at your option) any later version.

See file LICENSE.txt for further informations on licensing terms.

Last updated: September 16th, 2017

\*/

/\*

README

This is an example use of ConfigurableFirmata. The easiest way to create a configuration is to

use http://firmatabuilder.com and select the communication transport and the firmata features

to include and an Arduino sketch (.ino) file will be generated and downloaded automatically.

To manually configure a sketch, copy this file and follow the instructions in the

ETHERNET CONFIGURATION OPTION (if you want to use Ethernet instead of Serial/USB) and

FIRMATA FEATURE CONFIGURATION sections in this file.

\*/

#include "ConfigurableFirmata.h"

/\*==============================================================================

\* ETHERNET CONFIGURATION OPTION

\*

\* By default Firmata uses the Serial-port (over USB) of the Arduino. ConfigurableFirmata may also

\* comunicate over ethernet using tcp/ip. To configure this sketch to use Ethernet instead of

\* Serial, uncomment the approprate includes for your particular hardware. See STEPS 1 - 5 below.

\* If you want to use Serial (over USB) then skip ahead to the FIRMATA FEATURE CONFIGURATION

\* section further down in this file.

\*

\* If you enable Ethernet, you will need a Firmata client library with a network transport that can

\* act as a server in order to establish a connection between ConfigurableFirmataEthernet and the

\* Firmata host application (your application).

\*

\* To use ConfigurableFirmata with Ethernet you will need to have one of the following

\* boards or shields:

\*

\* - Arduino Ethernet shield (or clone)

\* - Arduino Ethernet board (or clone)

\* - Arduino Yun

\*

\* If you are using an Arduino Ethernet shield you cannot use the following pins on

\* the following boards. Firmata will ignore any requests to use these pins:

\*

\* - Arduino Uno or other ATMega328 boards: (D4, D10, D11, D12, D13)

\* - Arduino Mega: (D4, D10, D50, D51, D52, D53)

\* - Arduino Leonardo: (D4, D10)

\* - Arduino Due: (D4, D10)

\* - Arduino Zero: (D4, D10)

\*

\* If you are using an ArduinoEthernet board, the following pins cannot be used (same as Uno):

\* - D4, D10, D11, D12, D13

\*============================================================================\*/

// STEP 1 [REQUIRED]

// Uncomment / comment the appropriate set of includes for your hardware (OPTION A, B or C)

/\*

\* OPTION A: Configure for Arduino Ethernet board or Arduino Ethernet shield (or clone)

\*

\* To configure ConfigurableFirmata to use the an Arduino Ethernet Shield or Arduino Ethernet

\* Board (both use the same WIZ5100-based Ethernet controller), uncomment the SPI and Ethernet

\* includes below.

\*/

//#include <SPI.h>

//#include <Ethernet.h>

/\*

\* OPTION B: Configure for a board or shield using an ENC28J60-based Ethernet controller,

\* uncomment out the UIPEthernet include below.

\*

\* The UIPEthernet-library can be downloaded

\* from: https://github.com/ntruchsess/arduino\_uip

\*/

//#include <UIPEthernet.h>

/\*

\* OPTION C: Configure for Arduino Yun

\*

\* The Ethernet port on the Arduino Yun board can be used with Firmata in this configuration.

\* To execute StandardFirmataEthernet on Yun uncomment the Bridge and YunClient includes below.

\*

\* NOTE: in order to compile for the Yun you will also need to comment out some of the includes

\* and declarations in the FIRMATA FEATURE CONFIGURATION section later in this file. Including all

\* features exceeds the RAM and Flash memory of the Yun. Comment out anything you don't need.

\*

\* On Yun there's no need to configure local\_ip and mac address as this is automatically

\* configured on the linux-side of Yun.

\*

\* Establishing a connection with the Yun may take several seconds.

\*/

//#include <Bridge.h>

//#include <YunClient.h>

#if defined ethernet\_h || defined UIPETHERNET\_H || defined \_YUN\_CLIENT\_H\_

#define NETWORK\_FIRMATA

// STEP 2 [REQUIRED for all boards and shields]

// replace with IP of the server you want to connect to, comment out if using 'remote\_host'

#define remote\_ip IPAddress(192, 168, 0, 1)

// OR replace with hostname of server you want to connect to, comment out if using 'remote\_ip'

// #define remote\_host "server.local"

// STEP 3 [REQUIRED unless using Arduino Yun]

// Replace with the port that your server is listening on

#define remote\_port 3030

// STEP 4 [REQUIRED unless using Arduino Yun OR if not using DHCP]

// Replace with your board or Ethernet shield's IP address

// Comment out if you want to use DHCP

#define local\_ip IPAddress(192, 168, 0, 6)

// STEP 5 [REQUIRED unless using Arduino Yun]

// replace with Ethernet shield mac. Must be unique for your network

const byte mac[] = {0x90, 0xA2, 0xDA, 0x0D, 0x07, 0x02};

#endif

/\*==============================================================================

\* FIRMATA FEATURE CONFIGURATION

\*

\* Comment out the include and declaration for any features that you do not need

\* below.

\*

\* WARNING: Including all of the following features (especially if also using

\* Ethernet) may exceed the Flash and/or RAM of lower memory boards such as the

\* Arduino Uno or Leonardo.

\*============================================================================\*/

#include <DigitalInputFirmata.h>

DigitalInputFirmata digitalInput;

#include <DigitalOutputFirmata.h>

DigitalOutputFirmata digitalOutput;

#include <AnalogInputFirmata.h>

AnalogInputFirmata analogInput;

#include <AnalogOutputFirmata.h>

AnalogOutputFirmata analogOutput;

#include <Servo.h>

#include <ServoFirmata.h>

ServoFirmata servo;

// ServoFirmata depends on AnalogOutputFirmata

#if defined ServoFirmata\_h && ! defined AnalogOutputFirmata\_h

#error AnalogOutputFirmata must be included to use ServoFirmata

#endif

#include <Wire.h>

#include <I2CFirmata.h>

I2CFirmata i2c;

#include <OneWireFirmata.h>

OneWireFirmata oneWire;

// StepperFirmata is deprecated as of ConfigurableFirmata v2.10.0. Please update your

// client implementation to use the new, more full featured and scalable AccelStepperFirmata.

#include <StepperFirmata.h>

StepperFirmata stepper;

#include <AccelStepperFirmata.h>

AccelStepperFirmata accelStepper;

#include <SerialFirmata.h>

SerialFirmata serial;

#include <FirmataExt.h>

FirmataExt firmataExt;

#include <FirmataScheduler.h>

FirmataScheduler scheduler;

// To add Encoder support you must first install the FirmataEncoder and Encoder libraries:

// https://github.com/firmata/FirmataEncoder

// https://www.pjrc.com/teensy/td\_libs\_Encoder.html

// #include <Encoder.h>

// #include <FirmataEncoder.h>

// FirmataEncoder encoder;

/\*===================================================================================

\* END FEATURE CONFIGURATION - you should not need to change anything below this line

\*==================================================================================\*/

// dependencies. Do not comment out the following lines

#if defined AnalogOutputFirmata\_h || defined ServoFirmata\_h

#include <AnalogWrite.h>

#endif

#if defined AnalogInputFirmata\_h || defined I2CFirmata\_h || defined FirmataEncoder\_h

#include <FirmataReporting.h>

FirmataReporting reporting;

#endif

// dependencies for Network Firmata. Do not comment out.

#ifdef NETWORK\_FIRMATA

#if defined remote\_ip && defined remote\_host

#error "cannot define both remote\_ip and remote\_host at the same time!"

#endif

#include <EthernetClientStream.h>

#ifdef \_YUN\_CLIENT\_H\_

YunClient client;

#else

EthernetClient client;

#endif

#if defined remote\_ip && !defined remote\_host

#ifdef local\_ip

EthernetClientStream stream(client, local\_ip, remote\_ip, NULL, remote\_port);

#else

EthernetClientStream stream(client, IPAddress(0, 0, 0, 0), remote\_ip, NULL, remote\_port);

#endif

#endif

#if !defined remote\_ip && defined remote\_host

#ifdef local\_ip

EthernetClientStream stream(client, local\_ip, IPAddress(0, 0, 0, 0), remote\_host, remote\_port);

#else

EthernetClientStream stream(client, IPAddress(0, 0, 0, 0), IPAddress(0, 0, 0, 0), remote\_host, remote\_port);

#endif

#endif

#endif

/\*==============================================================================

\* FUNCTIONS

\*============================================================================\*/

void systemResetCallback()

{

// initialize a default state

// pins with analog capability default to analog input

// otherwise, pins default to digital output

for (byte i = 0; i < TOTAL\_PINS; i++) {

if (IS\_PIN\_ANALOG(i)) {

#ifdef AnalogInputFirmata\_h

// turns off pull-up, configures everything

Firmata.setPinMode(i, PIN\_MODE\_ANALOG);

#endif

} else if (IS\_PIN\_DIGITAL(i)) {

#ifdef DigitalOutputFirmata\_h

// sets the output to 0, configures portConfigInputs

Firmata.setPinMode(i, OUTPUT);

#endif

}

}

#ifdef FirmataExt\_h

firmataExt.reset();

#endif

}

/\*==============================================================================

\* SETUP()

\*============================================================================\*/

void setup()

{

/\*

\* ETHERNET SETUP

\*/

#ifdef NETWORK\_FIRMATA

#ifdef \_YUN\_CLIENT\_H\_

Bridge.begin();

#else

#ifdef local\_ip

Ethernet.begin((uint8\_t \*)mac, local\_ip); //start Ethernet

#else

Ethernet.begin((uint8\_t \*)mac); //start Ethernet using dhcp

#endif

#endif

delay(1000);

#endif

/\*

\* FIRMATA SETUP

\*/

Firmata.setFirmwareVersion(FIRMATA\_FIRMWARE\_MAJOR\_VERSION, FIRMATA\_FIRMWARE\_MINOR\_VERSION);

#ifdef FirmataExt\_h

#ifdef DigitalInputFirmata\_h

firmataExt.addFeature(digitalInput);

#endif

#ifdef DigitalOutputFirmata\_h

firmataExt.addFeature(digitalOutput);

#endif

#ifdef AnalogInputFirmata\_h

firmataExt.addFeature(analogInput);

#endif

#ifdef AnalogOutputFirmata\_h

firmataExt.addFeature(analogOutput);

#endif

#ifdef ServoFirmata\_h

firmataExt.addFeature(servo);

#endif

#ifdef I2CFirmata\_h

firmataExt.addFeature(i2c);

#endif

#ifdef OneWireFirmata\_h

firmataExt.addFeature(oneWire);

#endif

#ifdef StepperFirmata\_h

firmataExt.addFeature(stepper);

#endif

#ifdef AccelStepperFirmata\_h

firmataExt.addFeature(accelStepper);

#endif

#ifdef SerialFirmata\_h

firmataExt.addFeature(serial);

#endif

#ifdef FirmataReporting\_h

firmataExt.addFeature(reporting);

#endif

#ifdef FirmataScheduler\_h

firmataExt.addFeature(scheduler);

#endif

#ifdef FirmataEncoder\_h

firmataExt.addFeature(encoder);

#endif

#endif

/\* systemResetCallback is declared here (in ConfigurableFirmata.ino) \*/

Firmata.attach(SYSTEM\_RESET, systemResetCallback);

// Network Firmata communicates with Ethernet-shields over SPI. Therefor all

// SPI-pins must be set to PIN\_MODE\_IGNORE. Otherwise Firmata would break SPI-communication.

// add Pin 10 and configure pin 53 as output if using a MEGA with Ethernetshield.

// No need to ignore pin 10 on MEGA with ENC28J60, as here pin 53 should be connected to SS:

#ifdef NETWORK\_FIRMATA

#ifndef \_YUN\_CLIENT\_H\_

// ignore SPI and pin 4 that is SS for SD-Card on Ethernet-shield

for (byte i = 0; i < TOTAL\_PINS; i++) {

if (IS\_PIN\_SPI(i)

|| 4 == i // SD Card on Ethernet shield uses pin 4 for SS

|| 10 == i // Ethernet-shield uses pin 10 for SS

) {

Firmata.setPinMode(i, PIN\_MODE\_IGNORE);

}

}

// pinMode(PIN\_TO\_DIGITAL(53), OUTPUT); configure hardware-SS as output on MEGA

pinMode(PIN\_TO\_DIGITAL(4), OUTPUT); // switch off SD-card bypassing Firmata

digitalWrite(PIN\_TO\_DIGITAL(4), HIGH); // SS is active low;

#endif

#if defined(\_\_AVR\_ATmega1280\_\_) || defined(\_\_AVR\_ATmega2560\_\_)

pinMode(PIN\_TO\_DIGITAL(53), OUTPUT); // configure hardware SS as output on MEGA

#endif

// start up Network Firmata:

Firmata.begin(stream);

#else

// Uncomment to save a couple of seconds by disabling the startup blink sequence.

// Firmata.disableBlinkVersion();

// start up the default Firmata using Serial interface:

Firmata.begin(57600);

#endif

systemResetCallback(); // reset to default config

}

/\*==============================================================================

\* LOOP()

\*============================================================================\*/

void loop()

{

#ifdef DigitalInputFirmata\_h

/\* DIGITALREAD - as fast as possible, check for changes and output them to the

\* stream buffer using Firmata.write() \*/

digitalInput.report();

#endif

/\* STREAMREAD - processing incoming message as soon as possible, while still

\* checking digital inputs. \*/

while (Firmata.available()) {

Firmata.processInput();

#ifdef FirmataScheduler\_h

if (!Firmata.isParsingMessage()) {

goto runtasks;

}

}

if (!Firmata.isParsingMessage()) {

runtasks: scheduler.runTasks();

#endif

}

/\* SEND STREAM WRITE BUFFER - TO DO: make sure that the stream buffer doesn't go over

\* 60 bytes. use a timer to sending an event character every 4 ms to

\* trigger the buffer to dump. \*/

#ifdef FirmataReporting\_h

if (reporting.elapsed()) {

#ifdef AnalogInputFirmata\_h

/\* ANALOGREAD - do all analogReads() at the configured sampling interval \*/

analogInput.report();

#endif

#ifdef I2CFirmata\_h

// report i2c data for all device with read continuous mode enabled

i2c.report();

#endif

#ifdef FirmataEncoder\_h

// report encoders positions if reporting enabled.

encoder.report();

#endif

}

#endif

#ifdef StepperFirmata\_h

stepper.update();

#endif

#ifdef AccelStepperFirmata\_h

accelStepper.update();

#endif

#ifdef SerialFirmata\_h

serial.update();

#endif

#if defined NETWORK\_FIRMATA && !defined local\_ip &&!defined \_YUN\_CLIENT\_H\_

// only necessary when using DHCP, ensures local IP is updated appropriately if it changes

if (Ethernet.maintain()) {

stream.maintain(Ethernet.localIP());

}

#endif

}

### Código Servidor Node (server.js)

const express = require('express');

const bodyParser = require('body-parser');

const path = require('path');

const http = require('http');

const app = express();

// API file for interacting with MongoDB

const api = require('./server/routes/api');

// Parsers

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: false}));

// Angular DIST output folder

app.use(express.static(path.join(\_\_dirname, 'dist')));

// CORS

app.use(function(req, res, next) {

    res.header("Access-Control-Allow-Origin", "\*");

    res.header("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");

    next();

});

// API location

app.use('/api', api.rutas);

app.use('/api', api.placas);

// Enviar todo lo otro a Angular

app.get('\*', (req, res) => {

    if (process.env.AMBIENTE == 'DESARROLLO'){

        //Estoy levantando angular en puerto 4200

     res.status(404).send('Estas en desarrollo');

    }else{

res.sendFile(path.join(\_\_dirname, 'dist/index.html'));

    }

});

//Set Port

const port = process.env.PORT || '3000';

app.set('port', port);

const server = http.createServer(app);

server.listen(port, () => console.log(`Running on localhost:${port}`));

# Código API Express (api.js)

const express = require('express');

const router = express.Router();

const MongoClient = require('mongodb').MongoClient;

const ObjectID = require('mongodb').ObjectID;

const MINIMODISTANCIA = 20;

/\*Esto es para apagar\*/

const control = require('./apagar');

console.log(control.saludar());

if (process.env.AMBIENTE == 'DESARROLLO') {

console.log('Iniciado desarrollo');

var hola = require('./hola');

module.exports = hola;

} else {

console.log('Iniciado Test');

var placas = require('./placas');

module.exports.placas = placas;

}

var cont = 0;

// Connect

const connection = (closure) => {

return MongoClient.connect('mongodb://localhost:27017/sar', (err, db) => {

if (err) return console.log(err);

closure(db);

});

};

// Error handling

const sendError = (err, res) => {

response.status = 501;

response.message = typeof err == 'object' ? err.message : err;

res.status(501).json(response);

};

// Response handling

let response = {

status: 200,

data: [],

message: null

};

router.get('/temperaturas', (req, res) => {

var lista = [];

connection((db) => {

db.collection('temperaturas')

.distinct("fecha").then((fechas) => {

fechas.forEach(f => {

db.collection("temperaturas")

.find({ "fecha": f }, { "valor": 1, "hora": 1, "\_id": 0 }).sort({ "fecha": 1, "hora": 1 }).batchSize(30000)

.toArray(function (err, result) {

if (err) throw err;

lista.push({ series: result, fecha: f });

if (lista.length === fechas.length) {

console.log('Mostrando ----\*\*\*\*');

console.log(lista[5].series.length);

console.log(lista[5].fecha);

res.json(lista);

db.close();

}

});

});

})

.catch((err) => {

sendError(err, res);

})

});

});

router.get('/monoxidos', (req, res) => {

connection((db) => {

db.collection('mq7')

.find()

.toArray()

.then((valores) => {

response = valores;

res.json(response);

db.close();

})

.catch((err) => {

sendError(err, res);

});

});

});

router.get('/monoxidosActual', (req, res) => {

var ahora = new Date();

var despues = new Date();

despues.setSeconds(ahora.getSeconds() + 5);

hora1 = ahora.getHours() + ':' + ahora.getMinutes() + ':' + ahora.getSeconds();

hora2 = despues.getHours() + ':' + despues.getMinutes() + ':' + despues.getSeconds();

// console.log('Buscando con '+hora1+' y '+hora2);

connection((db) => {

db.collection('monoxidos')

.findOne({ "hora": { $gte: hora1, $lte: hora2 } }, { "valor": 1, "hora": 1, "\_id": 0 },

function (err, result) {

if (err) throw err;

// console.log(result);

res.json(result);

db.close();

});

});

});

/\*

Se exportan las variables para que sean conocidas por Nodejs

\*/

router.get('/apagar', (req, res) => {

console.log('LLego solicitud de apagado...');

control.apagar();

});

router.get('/reiniciar', (req, res) => {

console.log('Llego solicitud de reinicio...');

control.reiniciar();

});

module.exports.rutas = router;

### Código Manejo de Arduino Mega y Arduino Nano (placas.js)

const express = require('express');

const router = express.Router();

const MongoClient = require('mongodb').MongoClient;

const ObjectID = require('mongodb').ObjectID;

const MINIMODISTANCIA = 20;

const connection = (closure) => {

return MongoClient.connect('mongodb://localhost:27017/sar', (err, db) => {

if (err) return console.log(err);

closure(db);

});

};

var ports = [{ id: "mega", port: "/dev/ttyACM0" },

{ id: "nano", port: "/dev/ttyUSB0" }

];

var five = require("johnny-five");

new five.Boards(ports).on("ready", function () {

//Se enciende la placa

var motor1;

var motor2;

var motor3;

var motor4;

// configuro el sensor de monoxido

//Se enciende la placa

var sensor = new five.Sensor({

pin: "A0",

board: this.byId("mega"),

freq: 1000 //Frecuencia en msg

});

//Configuro el sensor de temperatura

var thermometer = new five.Thermometer({

controller: "DS18B20",

pin: 2,

board: this.byId("nano")

});

var gps = new five.GPS({

pins: {

rx: 15,

tx: 14,

board: this.byId("mega")

}

});

// si latitud o longitud cambian

gps.on("ready", function () {

console.log("position");

console.log(" latitude : ", this.latitude);

console.log(" longitude : ", this.longitude);

console.log(" altitude : ", this.altitude);

connection((db) => {

db.collection('gps')

.insert({ "latitude": this.latitude, "longitude": this.longitude, "altitude": this.altitude, "fecha": new Date() })

.then((muestrasGPS) => {

console.log('Insertando muestra GPS');

})

.catch((err) => {

console.log('Error al insertar GPS');

});

});

});

// If speed, course change log it

gps.on("navigation", function () {

console.log("navigation");

console.log(" speed : ", this.speed);

console.log(" course : ", this.course);

connection((db) => {

db.collection('navegacion')

.insert({ "velocidad": this.speed, "curso": this.course, "fecha": new Date() })

.then((muestrasGPS) => {

console.log('Insertando muestra navegacion');

})

.catch((err) => {

console.log('Error al insertar GPS');

});

});

});

//Configuro el sensor de proximidad en el pin 22

var proximityAdelante = new five.Proximity({

controller: "HCSR04",

pin: 22,

board: this.byId("mega")

});

var proximityDerecho = new five.Proximity({

controller: "HCSR04",

pin: 24,

board: this.byId("mega")

});

var proximityIzquierdo = new five.Proximity({

controller: "HCSR04",

pin: 26,

board: this.byId("mega")

});

var distanciaAdelante = 0;

motor1 = new five.Motor({

pins: {

pwm: 10,

dir: 7,

cdir: 6,

board: this.byId("mega")

}

});

motor2 = new five.Motor({

pins: {

pwm: 11,

dir: 8,

cdir: 9,

board: this.byId("mega")

}

});

motor3 = new five.Motor({

pins: {

pwm: 13,

dir: 5,

cdir: 4,

board: this.byId("mega")

}

});

motor4 = new five.Motor({

pins: {

pwm: 12,

dir: 2,

cdir: 3,

board: this.byId("mega")

}

});

function stop() {

motor1.stop();

motor2.stop();

motor3.stop();

motor4.stop();

}

//Muestro la temperatura

thermometer.on("change", function () {

//console.log(this.celsius + "°C");

//Agregado para generar hora

var ahora = new Date();

var hora = ahora.getHours() + ':' + ahora.getMinutes() + ':' + ahora.getSeconds();

var fechaAlmacenar = ahora.getFullYear() + '/' + (ahora.getMonth() + 1) + '/' + ahora.getDate();

connection((db) => {

db.collection('temperaturas')

.insert({ "valor": this.celsius, "fecha": fechaAlmacenar, "hora": hora, "fechaSys": ahora })

.then((temperaturas) => {

console.log('Insertando temperatura:' + this.celsius);

db.close();

})

.catch((err) => {

console.log('Error al insertar');

});

});

});

sensor.on("change", function (value) {

connection((db) => {

var ahora = new Date();

hora = ahora.getHours() + ':' + ahora.getMinutes() + ':' + ahora.getSeconds();

db.collection('monoxidos')

.insert({ "valor": sensor.scaleTo([20, 2000]), "fecha": new Date(), "hora": hora })

.then((sensorMQ7) => {

console.log('Insertando MQ7');

console.log(sensor.scaleTo([20, 2000]) + 'ppm'); // float

db.close();

})

.catch((err) => {

console.log('Error al insertar valores de mq7');

});

});

});

// Si se generan modificaciones en la distancia de objetos, paran o avanzan los motores

proximityAdelante.on("change", function () {

if (proximityAdelante.cm <= MINIMODISTANCIA) {

stop();

}

});

router.get('/arriba', (req, res) => {

console.log('Accionando arriba');

if (proximityAdelante.cm > MINIMODISTANCIA) {

motor1.forward(255);

motor2.forward(255);

motor3.forward(255);

motor4.forward(255);

res.json("ok");

} else {

res.json("{'error':'objeto adelante'}");

}

});

router.get('/izquierda', (req, res) => {

console.log('Accionando izquierda');

res.json("ok");

motor1.forward(255);

motor2.reverse(255);

motor3.forward(255);

motor4.reverse(255);

});

router.get('/derecha', (req, res) => {

console.log('Accionando derecha');

motor1.reverse(255);

motor2.forward(255);

motor3.reverse(255);

motor4.forward(255);

res.json("ok");

});

router.get('/abajo', (req, res) => {

console.log('Accionando abajo');

motor1.reverse(255);

motor2.reverse(255);

motor3.reverse(255);

motor4.reverse(255);

res.json("ok");

});

router.get('/stop', (req, res) => {

console.log('deteniendo');

motor1.stop();

motor2.stop();

motor3.stop();

motor4.stop();

res.json("ok");

});

router.get('/ultrasonido', (req, res) => {

res.json([{ ultrasonidoAdelante: proximityAdelante.cm },

{ ultrasonidoDerecho: proximityDerecho.cm },

{ ultrasonidoIzquierdo: proximityIzquierdo.cm }

]);

});

router.get('/gps', (req, res) => {

res.json({

latitud: gps.latitude,

longitud: gps.longitude,

altitud: gps.altitude,

velocidad: gps.speed,

sat: gps.sat,

curso: gps.course,

tiempo: gps.time

});

});

router.get('/temperatura', (req, res) => {

res.json({

temperatura: thermometer.celsius, unidad: "celsius"

});

});

router.get('/monoxido', (req, res) => {

res.json({ monoxido: sensor.scaleTo([20, 2000]), unidad: "ppm" });

});

});

module.exports = router;

### Código de Apagar y reiniciar (apagar.js)

// Require child\_process

var exec = require('child\_process').exec;

// Create shutdown function

var os = require('os');

var apagar = function shutdown() {

if (os.type()[0] === 'W' || os.type()[0] === 'w') {

console.log('Es windows');

} else {

console.log('Es linux');

}

console.log(os.type());

exec('shutdown -h now', function (error, stdout, stderr) { console.log(stdout); });

}

var reiniciar = function reiniciar() {

exec('shutdown -r now', function (error, stdout, stderr) { console.log(stdout); });

}

var saludar = function saludar(callback) {

console.log('Hola desde apagar.js');

}

module.exports.reiniciar = reiniciar;

module.exports.apagar = apagar;

module.exports.saludar = saludar;

## **Códigos del lado del cliente**

### Código de Servicio Angular (servicio.ts)

import { Gps } from './Gps';

import { Injectable } from '@angular/core';

import { Http, Headers, RequestOptions } from '@angular/http';

import {HttpClient} from '@angular/common/http';

import 'rxjs/add/operator/map';

import { Observable } from 'rxjs/Observable';

import { AppComponent } from './app.component';

import { Temperatura } from './Temperatura';

import { Sensores } from './Sensores';

import { GraficaTemperatura } from './graficaTemperatura';

import { Monoxido } from './Monoxido';

@Injectable()

export class ServicioAplicacion {

public sensores: Sensores;

constructor(public http: Http, public http2: HttpClient, public app: AppComponent) {

this.sensores = new Sensores();

}

/\*\*

\* Evento que sirve para enviar desde el componente

\* @param url Url a enviar el evento [arriba, abajo, izquierda, derecha]

\*/

enviarEvento(accion: String): Observable <any> {

return this.http.get(this.app.rutaBasica + accion);

}

solicitarTemperaturaActual(): Observable <Temperatura> {

return this.http.get(this.app.rutaBasica + 'temperatura')

.map(res => res.json());

}

/\*\*

\* Devuelve ([{ultrasonidoAdelante:proximityAdelante.cm},

    {ultrasonidoDerecho: proximityDerecho.cm},

    {ultrasonidoIzquierdo: proximityIzquierdo.cm}

    ])

\*/

solicitarUltrasonidosActual(): Observable <any> {

return this.http.get(this.app.rutaBasica + 'ultrasonido')

.map(res => res.json());

}

/\*\*devuelve

\* {monoxido: sensor.scaleTo([20,2000]), unidad: "ppm"}

\*/

solicitarMonoxidoActual(): Observable <any> {

return this.http.get(this.app.rutaBasica + 'monoxido')

.map(res => res.json());

}

solicitarGpsActual(): Observable <Gps> {

return this.http.get(this.app.rutaBasica + 'gps')

.map(res => res.json());

}

pedirImagen(): Observable<boolean> {

return this.http.get(this.app.rutaWeb)

.map(res => res.json());

}

actualizarValores(muestraSensores: Sensores) {

this.sensores = muestraSensores;

}

apagar() {

return this.http.get(this.app.rutaBasica + 'apagar');

}

reiniciar() {

return this.http.get(this.app.rutaBasica + 'reiniciar');

}

solicitarTodasTemperaturas(): Observable <GraficaTemperatura[]> {

return this.http2.get<GraficaTemperatura[]>(this.app.rutaBasica + 'temperaturas');

}

solicitarMonoxidoActualBD(): Observable <Monoxido> {

return this.http2.get<Monoxido>(this.app.rutaBasica + 'monoxidosActual');

}

}

### Código de appComponent.ts Angular

import { Sensores } from './Sensores';

import { Component } from '@angular/core';

import { TemperaturaService } from './temperatura.service';

import { Temperatura } from './Temperatura';

import { DatePipe } from '@angular/common';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css'],

providers: [TemperaturaService],

})

export class AppComponent {

titulo = 'Sistema Autónomo Robótico';

public temperaturas: Temperatura[];

public sensoresActuales: Sensores;

public tiempoDelay;

public mensaje = '';

public rutaBasica = 'http://192.168.2.1:3000/api/';

// public rutaBasica = 'http://localhost:3000/api/';

// public rutaBasica = 'http://192.168.1.39:3000/api/';

public rutaWeb = 'http://192.168.2.1:8081';

constructor(public service: TemperaturaService) {

this.sensoresActuales = new Sensores();

this.tiempoDelay = 1000;

}

}

### Código de appModule (rutas) [Extracto]

const routes: Routes = [

{ path: '', component: PizarraComponent },

{ path: 'estadisticas', component: EstadisticasComponent },

{ path: 'control', component: PanelInferiorComponent }

];

@NgModule({

declarations: [

AppComponent,

CapturaVideoComponent,

TablaSensoresComponent,

TablaInfoComponent,

PanelInferiorComponent,

PanelControlComponent,

PizarraComponent,

EstadisticasComponent,

EncabezadoComponent,

],

imports: [

BrowserModule,

HttpModule,

RouterModule.forRoot(routes),

NgxChartsModule,

BrowserAnimationsModule,

FormsModule,

HttpClientModule

],

providers: [TemperaturaService],

bootstrap: [AppComponent]

})

export class AppModule { }