



Laboratorio de desarrollo Hardware

Memoria de prácticas

Ramón Chávez García
Curso 2016-2017

ÍNDICE

| | |
|--|-----------|
| OBJETIVOS | 5 |
| PLATAFORMA ARDUINO | 6 |
| INTRODUCCIÓN: PLATAFORMA, COMPONENTES Y ENTORNO DE TRABAJO | 6 |
| DESARROLLO PRÁCTICO | 10 |
| ➤ Blink | 10 |
| ➤ Fade | 10 |
| ➤ Blink (con switch) | 11 |
| ➤ Encendido/Apagado de un LED mediante el puerto serie | 11 |
| ➤ Controlar encendido-apagado de una bombilla mediante un relé | 12 |
| ➤ Encendido de un LED mediante pulsador (uso de interrupciones) | 13 |
| ➤ Control de la posición de un servo con un potenciómetro | 14 |
| ➤ Control de la posición de un servo mediante puerto serie | 15 |
| ➤ Control de la velocidad de giro de un motor DC doble sentido con potenciómetro | 16 |
| ➤ Impresión en una pantalla LCD | 17 |
| ➤ Impresión en una pantalla LCD usando el puerto serie | 19 |
| ➤ Impresión de la temperatura en una pantalla LCD con el sensor TMP36GZ | 20 |
| ➤ Contador de 00 a 59 en display 7 segmentos | 21 |
| PLATAFORMA ZPUINO | 24 |
| INTRODUCCIÓN: PLATAFORMA Y ENTORNO DE TRABAJO | 24 |
| DESARROLLO PRÁCTICO | 28 |
| ➤ Diseñando circuitos básicos sobre la FPGA | 28 |
| ➤ Cargar SoC ZPUino y desarrollar sketches | 30 |
| ➤ Convirtiendo la placa Papilio en un analizador lógico | 31 |
| ➤ Rediseñando el SoC. Añadiendo periféricos | 33 |
| PLATAFORMA RASPBERRY PI | 36 |
| INTRODUCCIÓN: PLATAFORMA Y ENTORNO DE TRABAJO | 36 |
| DESARROLLO PRÁCTICO | 41 |
| ➤ Manejo de GPIOs desde línea de comandos | 41 |
| ➤ Ejemplos sencillos con PYTHON | 42 |
| ➤ Conexión por SSH | 42 |
| ➤ Servidor Web (LEDs) | 43 |
| ➤ Servidor Web (Temperatura) | 47 |
| ➤ Servidor Web (Relé) | 48 |
| CONCLUSIONES | 50 |
| ANEXO AUDIOVISUAL | 50 |

ÍNDICE DE ILUSTRACIONES

| | |
|--|----|
| Ilustración 1: Plataformas de desarrollo utilizadas | 5 |
| Ilustración 2: Placas proyecto Arduino.cc | 6 |
| Ilustración 3: Kit desarrollo laboratorio | 7 |
| Ilustración 4: Fototransistores | 7 |
| Ilustración 5: Potenciómetro..... | 7 |
| Ilustración 6: Pulsador..... | 7 |
| Ilustración 7: Sensor TMP36..... | 7 |
| Ilustración 8: Sensor de inclinación..... | 7 |
| Ilustración 9: LCM1602C | 7 |
| Ilustración 10: LEDs | 7 |
| Ilustración 11: Motor DC..... | 7 |
| Ilustración 12: Servomotor..... | 8 |
| Ilustración 13: Sensor piezoeléctrico | 8 |
| Ilustración 14: L293D..... | 8 |
| Ilustración 15: 4N35 | 8 |
| Ilustración 16: IRF520..... | 8 |
| Ilustración 17: Condensador | 8 |
| Ilustración 18: 1N4007 | 8 |
| Ilustración 19: Resistencias | 8 |
| Ilustración 20: Pines macho | 8 |
| Ilustración 21: Switch | 8 |
| Ilustración 22: Doble relé | 8 |
| Ilustración 23: Cables de interconexión..... | 8 |
| Ilustración 24: Shield 7 segmentos compatible con Arduino..... | 9 |
| Ilustración 25: Configuración de la placa Arduino UNO..... | 9 |
| Ilustración 26: Estructura básica Arduino | 10 |
| Ilustración 27: Código Blink..... | 10 |
| Ilustración 28: Código Fade..... | 10 |
| Ilustración 29: Código Blink con switch..... | 11 |
| Ilustración 30: Montaje LED puerto serie | 11 |
| Ilustración 31: Código Arduino ON/OFF LED puerto serie | 11 |
| Ilustración 32: Código Python ON/OFF LED puerto serie..... | 12 |
| Ilustración 33: Ejecución código Python ON/OFF LED puerto Serie | 12 |
| Ilustración 34: Encendido y apagado del LED respectivamente. | 12 |
| Ilustración 35: Montaje LED mediante el uso de interrupciones..... | 13 |
| Ilustración 36: Código Arduino LED mediante el uso de interrupciones | 13 |
| Ilustración 37: Ejecución código Arduino LED mediante el uso de interrupciones | 13 |
| Ilustración 38: Montaje servo con potenciómetro | 14 |
| Ilustración 39: Código Arduino control servo mediante potenciómetro..... | 15 |
| Ilustración 40: Código Arduino control servo mediante puerto serie. | 15 |
| Ilustración 41: Código Python control servo mediante puerto serie..... | 15 |
| Ilustración 42: Ejecución códigos para el control servo mediante puerto serie..... | 15 |
| Ilustración 43: Partes de un motor DC..... | 16 |
| Ilustración 44: Motor brushless | 16 |

| | |
|---|----|
| Ilustración 45: Puente en H..... | 16 |
| Ilustración 46: Puente en H L293DNE | 17 |
| Ilustración 47: Montaje motor DC controlado por potenciómetro | 17 |
| Ilustración 48: Código Arduino motor con puente en H | 17 |
| Ilustración 49: Partes LCD | 17 |
| Ilustración 50: Montaje LCD | 19 |
| Ilustración 51: Código Arduino LCD sin usar puerto serie..... | 19 |
| Ilustración 52: Montaje y ejecución de la impresión en pantalla LCD sin usar puerto serie | 19 |
| Ilustración 53: Código Arduino LCD usando puerto serie | 20 |
| Ilustración 54: Código Python LCD usando puerto serie..... | 20 |
| Ilustración 55: Ejecución en consola de LCD usando puerto serie..... | 20 |
| Ilustración 56: Montaje y ejecución LCD usando puerto serie | 20 |
| Ilustración 57: Código Arduino temperatura por LCD..... | 21 |
| Ilustración 58: Números y letras en display 7 segmentos..... | 21 |
| Ilustración 59: Identificación de las partes del 7 segmentos | 21 |
| Ilustración 60: Código Arduino write7seg (...)..... | 22 |
| Ilustración 61: Código Arduino refresh (... , ...)..... | 22 |
| Ilustración 62: Código Arduino write_data (...)..... | 22 |
| Ilustración 63: Código Arduino loop() del 7 segmentos..... | 23 |
| Ilustración 64: Código Arduino variables globales y setup() del 7 segmentos..... | 23 |
| Ilustración 65: Wings vs. MegaWings | 24 |
| Ilustración 66: Papilio DUO | 25 |
| Ilustración 67: Papilio One | 25 |
| Ilustración 68: Papilio Pro | 25 |
| Ilustración 69: Xilinx Spartan 3E..... | 25 |
| Ilustración 70: Logo ZPUino..... | 25 |
| Ilustración 71: Diagrama SoC ZPUino..... | 26 |
| Ilustración 72: Descargar ISE | 26 |
| Ilustración 73: IDE ZPUino | 27 |
| Ilustración 74: Selección de la placa en plataforma ZPUino | 27 |
| Ilustración 75: Selección del puerto en la plataforma ZPUino..... | 27 |
| Ilustración 76: Creación proyecto FPGA..... | 28 |
| Ilustración 77: Editar circuito ZPUino..... | 28 |
| Ilustración 78: Abrir el editor esquemático | 28 |
| Ilustración 79: Arrastrando el inversor | 28 |
| Ilustración 80: Ver los pines disponibles | 28 |
| Ilustración 81: Conexiones Papilio | 29 |
| Ilustración 82: Renombrando I/O Papilio..... | 29 |
| Ilustración 83: Diseño inversor..... | 29 |
| Ilustración 84: Generación de programa DesignLab | 29 |
| Ilustración 85: Carga del circuito en la placa..... | 29 |
| Ilustración 86: Ejecución de la carga del circuito en la placa | 30 |
| Ilustración 87: Carga del sketch en ZPUino | 30 |
| Ilustración 88: Monitor serie Papilio_Quickstart | 30 |
| Ilustración 89: Modificación para mostrar estado LED ZPUino..... | 30 |
| Ilustración 90: Código Python LED ON/OFF puerto serie Papilio | 31 |

| | |
|--|----|
| Ilustración 91: Código ZPUino LED ON/OFF puerto serie | 31 |
| Ilustración 92: Ejecución LED ON/OFF por puerto serie Papilio | 31 |
| Ilustración 93: Icono analizador lógico..... | 31 |
| Ilustración 94: Analizador lógico | 32 |
| Ilustración 95: Configuración conexión analizador lógico | 32 |
| Ilustración 96: Código Fade analizador lógico..... | 32 |
| Ilustración 97: Ejecución del analizador lógico en varios canales..... | 33 |
| Ilustración 98: Montaje del analizador lógico | 33 |
| Ilustración 99: Creación del proyecto SoC en ZPUino | 33 |
| Ilustración 100: Añadiendo la UART..... | 34 |
| Ilustración 101: Supresión de pines para renombrar en la UART | 34 |
| Ilustración 102: Conexión de la UART con el Wishbone | 34 |
| Ilustración 103: Código SoC ZPUino | 34 |
| Ilustración 104: Código SoC ZPUino puerto serie | 35 |
| Ilustración 105: Montaje de la UART | 35 |
| Ilustración 106: Modelos RPi..... | 36 |
| Ilustración 107: Escritura de Ubuntu MATE en µSD..... | 37 |
| Ilustración 108: Partición montada de Ubuntu MATE | 37 |
| Ilustración 109: Desmontando partición de Ubuntu MATE..... | 38 |
| Ilustración 110: Resolución error RPi | 39 |
| Ilustración 111: Inicialización de Ubuntu MATE | 39 |
| Ilustración 112: Conexión RPi casera | 39 |
| Ilustración 113: Instalación de Ubuntu MATE..... | 39 |
| Ilustración 114: Resize de RPi..... | 40 |
| Ilustración 115: Botón de resize..... | 40 |
| Ilustración 116: GPIO RPi 3 B..... | 41 |
| Ilustración 117: Ejecución manejo de GPIO LED ON/OFF | 41 |
| Ilustración 118: Código control LEDs Python RPi | 42 |
| Ilustración 119: Conexión a RPi vía SSH | 43 |
| Ilustración 120: Código Python para probar el Flask | 43 |
| Ilustración 121: Ejecución hello-flask.py en terminal | 44 |
| Ilustración 122: Acceso a la ruta raíz mostrando información con la modificación del idioma. 44 | |
| Ilustración 123: Código Python hello-template.py | 44 |
| Ilustración 124: Código main.html del ejemplo Python hello-template.py..... | 44 |
| Ilustración 125: Acceso a la ruta raíz mostrando información con la modificación del idioma (fecha y hora) | 45 |
| Ilustración 126: BOARD Vs. BCM..... | 45 |
| Ilustración 127: Código Python servidor web (LEDs) | 46 |
| Ilustración 128: Ejecución sobre navegador de servidor web LEDs (I) | 46 |
| Ilustración 129: Ejecución sobre navegador de servidor web LEDs (II) | 46 |
| Ilustración 130: Ejecución y montaje servidor web LEDs..... | 47 |
| Ilustración 131: Código Arduino servidor web (temperatura y relé) | 48 |
| Ilustración 132: Código Python servidor web (completo) | 49 |
| Ilustración 133: Código HTML (y CSS) servidor web (completo)..... | 50 |

Objetivos

Durante estas sesiones hemos visto tres plataformas. De la primera de ellas, Arduino, hemos conocido la plataforma, sus características, variantes y modos de programación. También hemos conocido componentes hardware básicos que hemos empleado en las siguientes plataformas. Hemos configurado el PC para ejecutar su entorno de desarrollo para hacer ejemplos básicos y más complejos.

La segunda que hemos empleado ha sido ZPUino. Hemos conocido la plataforma Papilio, hemos instalado y configurado el entorno Design Lab para diseñar circuitos a nivel de captura de esquemáticos y su implementación en la FPGA, cargar el SoC ZPUino, desarrollar sketches para ZPUino, modificarlo añadiendo algún nuevo periférico y desarrollar algún sketch para su uso y configurar la placa Papilio para que funcione como un analizador lógico.

Por último, hemos trabajado con RPi 3 Model B. Hemos aprendido a preparar la plataforma para cargar sistemas operativos, comprobar su funcionamiento, desarrollar ejemplos que emplearon el uso de los pines de expansión GPIOs y a instalar un servidor web que pueda ejecutar código Python para interactuar con sensores y actuadores.

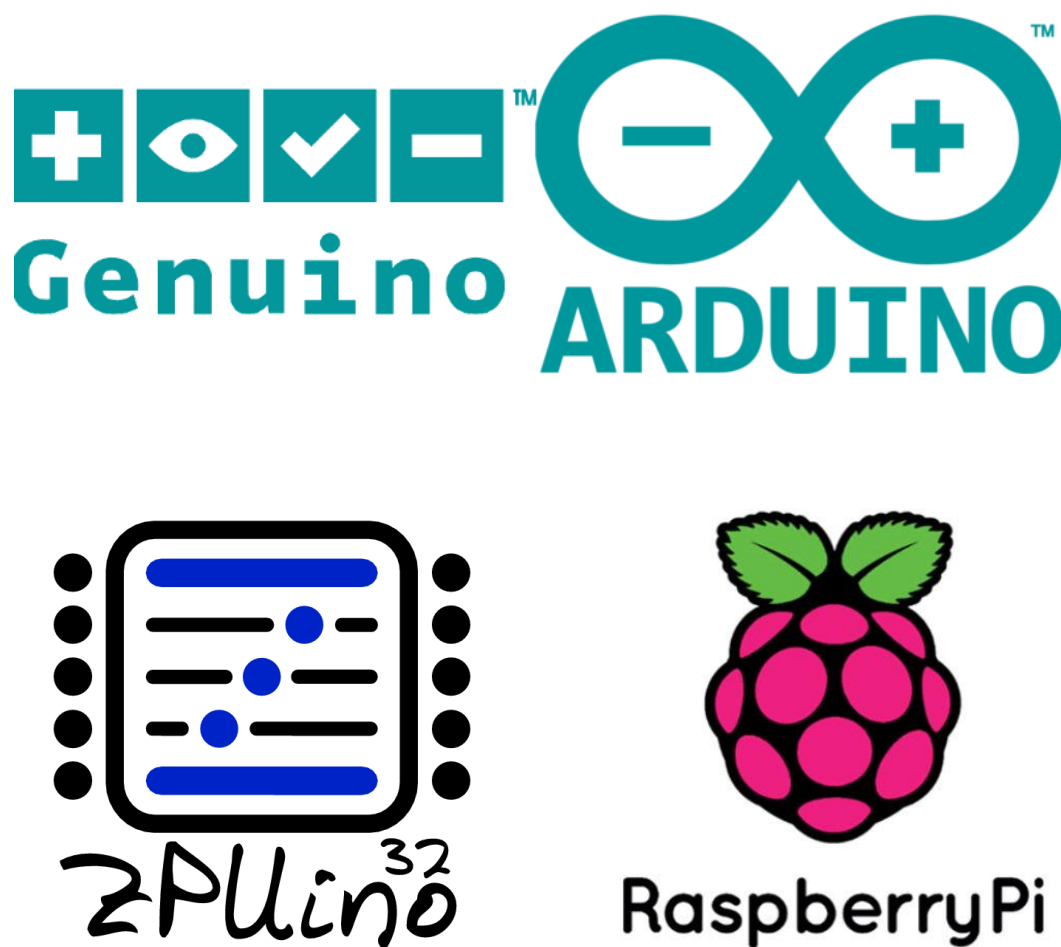


Ilustración 1: Plataformas de desarrollo utilizadas

Plataforma Arduino

INTRODUCCIÓN: PLATAFORMA, COMPONENTES Y ENTORNO DE TRABAJO

Arduino es una plataforma de desarrollo de hardware libre (**open hardware**) que está basada en software y hardware flexibles y fáciles de usar. Arduino toma información del entorno mediante sus pines de entrada y toda una gama de sensores, consiguiendo controlar luces, motores y otros tipos de actuadores.

Utilizaba, inicialmente, microcontroladores de 8 bits AVR (ATmega8, 128, 328, 1280) aunque también cuenta con alguna versión basada en ARM de 32 bits (Arduino Due). Entre las ventajas de Arduino destacan un entorno de desarrollo manejable, basado en **Processing**, un conjunto de librerías para el manejo de periféricos muy extenso y una comunidad grande de desarrolladores.

En 2015, uno de los cinco fundadores iniciales se separó del proyecto original y fundó www.arduino.org. El resto mantuvo el proyecto inicial www.arduino.cc. Como consecuencia de la escisión, estos últimos cambiaron la referencia al proyecto fuera de E.E.U.U. por **Genuino**, ya que Arduino como tal se encontraba registrada en Italia por el antiguo integrante. Además, ninguna de las dos corrientes se ha preocupado por hacer el IDE compatible para nuevos productos, diferentes para cada rama. Estas son algunas de las placas que oferta el proyecto inicial:

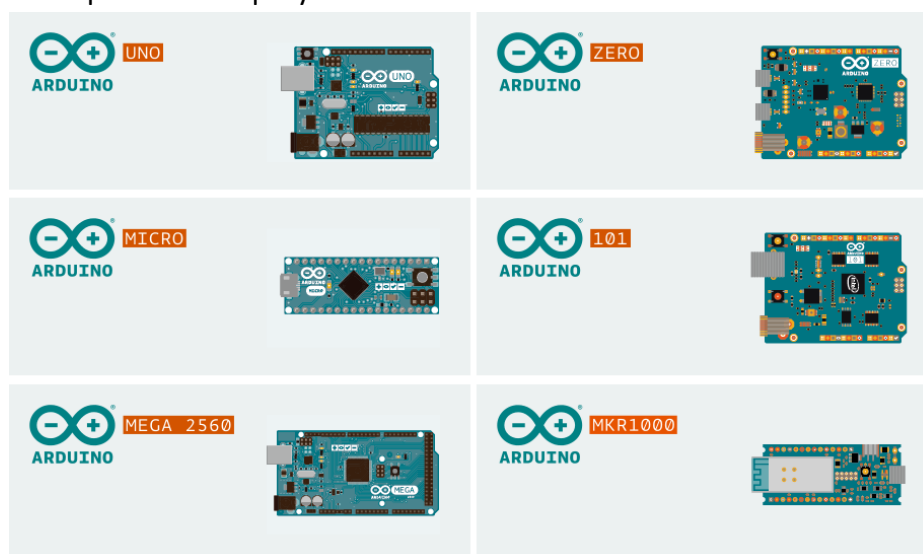


Ilustración 2: Placas proyecto Arduino.cc

En el laboratorio empleamos el pack **Starter Kit**, que trae un Arduino UNO con una protoboard entre otros componentes. Esta placa tiene un ATMEGA328P, una memoria flash de 32K, 14 GPIO (de los cuales 6 pueden ser PWM), 6 pines analógicos y un reloj de 16MHz:



Ilustración 3: Kit desarrollo laboratorio

Cabe mencionar también la interesante característica de Arduino que es el hecho de poder emplear shields. Un **shield** es una placa de expansión que permite a Arduino utilizar las características de la misma, por ejemplo, de un display 7 segmentos. El montaje es tan sencillo como hacer coincidir los pines de Arduino con la shield.

La propia página de Arduino nos proporciona también los **esquemáticos** de las placas y los archivos en **Eagle**. En cuanto a componentes electrónicos hardware destacamos algunos incluidos en el Starter Kit como:

Fototransistores: en estos componentes se sustituye la base por un cristal fotosensible que al recibir luz, sensible o infrarroja, genera corriente, desbloqueándolo.

Potenciómetros: es una resistencia variable que se compone de tres terminales. Los extremos se conectan a la VDD y GND de manera que, gracias a la resistencia variable, el tercer terminal cambia de forma mecánica entre ellos.

Pulsadores: una lámina conductora establece contacto con dos terminales al oprimir el botón, entonces, un muelle recobra la lámina a su posición primitiva al cesar la presión sobre este elemento.

Sensor de temperatura TMP36: la tensión varía en función de la temperatura.

Sensor de inclinación: cuatro terminales; una bola de metal interconecta dos de los terminales en función de la inclinación del componente.

Pantalla LCD LCM1602C: basada en controlador Hitachi HD44780.

LEDs: componente optoelectrónico pasivo que, más concretamente, es un diodo que emite luz.

Motor DC pequeño (6/9V): una tensión continua activa el giro del motor, que da vueltas en dos sentidos, según la polaridad de la corriente. Un μ controlador puede cambiar el sentido empleando hardware como un puente en H.



Ilustración 4: Fototransistores



Ilustración 5: Potenciómetro



Ilustración 6: Pulsador



Ilustración 7: Sensor TMP36



Ilustración 8: Sensor de inclinación



Ilustración 9: LCM1602C



Ilustración 10:



Ilustración 11: Motor DC

Servomotor: es un motor DC con circuitería de control incluida. La función del mismo es girar hasta colocarse en un ángulo definido y mantenerse ahí el tiempo que se desee. El ángulo de giro va de 0 a 180 grados y depende del tamaño del pulso que se envíe por la señal de control (**PWM** normalmente).



Ilustración 12: Servomotor

Sensor piezoeléctrico: dispositivo que utiliza el efecto piezoeléctrico para medir presión, aceleración, tensión o fuerza, transformando las lecturas en señales eléctricas.



Ilustración 13: Sensor piezoeléctrico

Driver de motores (Puente en H) L293D: circuito que puede intercambiar la polaridad en las señales de salida con señales de control de entrada.

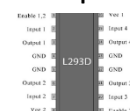


Ilustración 14: L293D

Optoacoplador 4N35: permite activar/desactivar un switch de un circuito desde otro circuito separado.



Ilustración 15: 4N35

Transistores MOSFET IRF520: Componentes de tres terminales que se emplean como interruptores, activando/desactivando a través del terminal de **gate**. Existen transistores de potencia para actuar de interruptores en sistemas de mas potencia controlados por señales digitales.



Ilustración 16: IRF520

Condensadores: dispositivo pasivo formado por un par de láminas conductoras contrapuestas y separadas, por un dieléctrico, generalmente, que se ven sometidas a una diferencia de potencial, adquiriendo carga eléctrica.



Ilustración 17: Condensador

Diodos 1N4007: componente electrónico de dos terminales que permite la circulación de la corriente eléctrica a través de él en un solo sentido.



Ilustración 18: 1N4007

40 pines macho y resistencias:



Ilustración 20: Pines macho



Ilustración 19: Resistencias

Además, en el laboratorio también hemos hecho uso de **cables de interconexión**, **relés**, **switches** o **display 7 segmentos**:



Ilustración 23: Cables de interconexión



Ilustración 22: Doble relé

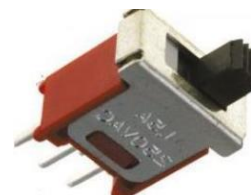


Ilustración 21: Switch



Ilustración 24: Shield 7 segmentos compatible con Arduino

Por otra parte, en lo referente al software, hemos empleado el sistema operativo Ubuntu, el IDE de Arduino como entorno de desarrollo y el propio bootloader, que es el cargador de arranque. El μ controlador de la placa se programa usando la conexión RS-232 a TTL serial.

La descarga del mismo la podemos realizar desde su página web: <https://www.arduino.cc/en/Main/Software>. Siguiendo las pantallas, elegimos el paquete a descargar según el S.O. que usemos (32 o 64 bits) y obtendremos un archivo **.tar.xz**. Este archivo lo descomprimos y situamos un terminal de Ubuntu en la carpeta que hayamos elegido.

Si queremos crear un acceso directo en el escritorio, escribimos entonces **“./install.sh”**. En caso de querer solo ejecutar, podemos optar por dos métodos: el primero es usar **“sudo arduino”** en un terminal desde cualquier directorio. Esto arrancará un **sketch** de Arduino vacío. Si queremos que tenga una pequeña estructura, desde el terminal donde creamos el acceso directo, habría que escribir **“./arduino”**.

Una vez tenemos abierto el entorno de trabajo, podemos recurrir a este link para familiarizarnos con las diferentes opciones que este presenta: <https://www.arduino.cc/en/Guide/Environment>.

La configuración de nuestra placa es muy sencilla de hacer:

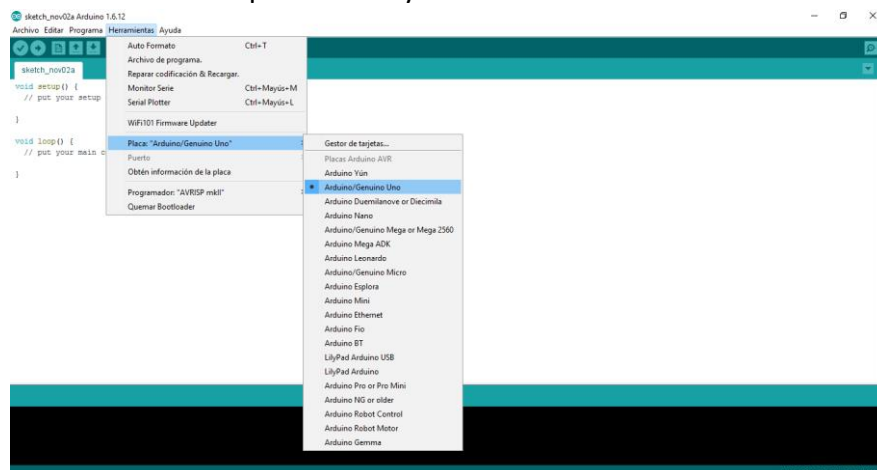


Ilustración 25: Configuración de la placa Arduino UNO

El lenguaje de programación es muy similar al C. Está basado en Wiring, aunque si no lo tenemos muy claro o queremos entrar en más detalle es bueno tener este enlace a mano:

<https://www.arduino.cc/en/Reference/HomePage>. El programa inicial presenta la siguiente estructura, propia de un sistema stand-alone, es decir, que siempre va a ejecutar la misma tarea:

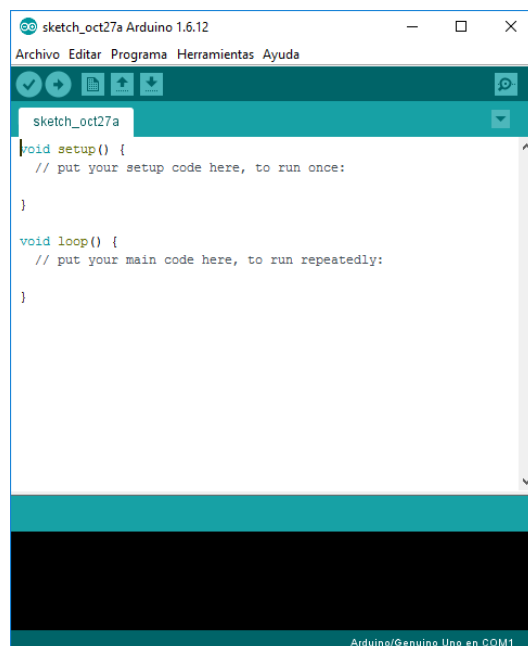


Ilustración 26: Estructura básica Arduino

DESARROLLO PRÁCTICO

Como toma de contacto, iniciamos el desarrollo práctico ejecutando un par de programas de ejemplo como son el Blink y el Fade.

- **Blink:** El programa apaga durante un segundo y enciende durante un segundo, repetidamente, el LED que incorpora la placa para este ejemplo específico:

```
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(1000);                     // wait for a second
  digitalWrite(LED_BUILTIN, LOW);  // turn the LED off by making the voltage LOW
  delay(1000);                     // wait for a second
}
```

Ilustración 27: Código Blink

- **Fade:** El programa se encarga de utilizar la función “analogWrite (... , ...)” para iluminar, paulatinamente, un LED desde su estado OFF a su estado ON y viceversa:

```
int led = 9;           // the pin that the LED is attached to
int brightness = 0;    // how bright the LED is
int fadeAmount = 5;    // how many points to fade the LED by

// the setup routine runs once when you press reset:
void setup() {
  // declare pin 9 to be an output:
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  // set the brightness of pin 9:
  analogWrite(led, brightness);

  // change the brightness for next time through the loop:
  brightness = brightness + fadeAmount;

  // reverse the direction of the fading at the ends of the fade:
  if (brightness == 0 || brightness == 255) {
    fadeAmount = -fadeAmount ;
  }
  // wait for 30 milliseconds to see the dimming effect
  delay(30);
}
```

Ilustración 28: Código Fade

Sometemos el programa Blink a una serie de modificaciones:

- **Blink (con switch):** Alteramos el comportamiento de Blink mediante un switch que ejerce la función de habilitar (como un pulsador lo haría):

```
int habilitador;
// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  habilitador = digitalRead(8);
  if(habilitador){
    digitalWrite(LED_BUILTIN, HIGH); // turn the LED on (HIGH is the voltage level)
    delay(1000);                      // wait for a second
    digitalWrite(LED_BUILTIN, LOW);   // turn the LED off by making the voltage LOW
    delay(1000);                      // wait for a second
  }else{
    digitalWrite(LED_BUILTIN, LOW);
  }
}
```

Ilustración 29: Código Blink con switch

Tras esto, comenzaremos con la plataforma Arduino controlando algunos componentes.

- **Encendido/Apagado de un LED mediante el puerto serie:** El montaje del circuito es muy sencillo:

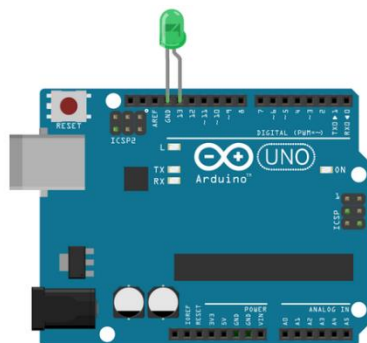


Ilustración 30: Montaje LED puerto serie

En cuanto al programa, hay dos partes. La primera de ellas la comprende la plataforma Arduino y es el siguiente código:

```
int led = 13;
void setup () {
  pinMode(led, OUTPUT); //LED 13 como salida
  Serial.begin(9600); //Inicializo el puerto serial a 9600 baudios
}

void loop () {
  if (Serial.available()) { //Si está disponible
    char c = Serial.read(); //Guardamos la lectura en una variable char
    if (c == 'H') { //Si es una 'H', enciendo el LED
      digitalWrite(led, HIGH);
    } else if (c == 'L') { //Si es una 'L', apago el LED
      digitalWrite(led, LOW);
    }
  }
}
```

Ilustración 31: Código Arduino ON/OFF LED puerto serie

Hacemos uso de la librería 'Serial' para la lectura del puerto serie (**Serial.read**), para inicializar el mismo (**Serial.begin**) y para comprobar si este está disponible (**Serial.available**).

Arduino inicializa la variable 'led' con 13, lo pone en modo **OUTPUT** e inicializa el **puerto serie**. El **loop** se encarga de leer los valores que recibe por consola de la segunda parte del código: el archivo '.py' de Python.

```
import serial

arduino = serial.Serial('/dev/ttyACM0', 9600)

print("Starting!")

while True:
    comando = raw_input('Introduce un comando: ') #Input
    arduino.write(comando) #Mandar un comando hacia Arduino
    if comando == 'H':
        print('LED ENCENDIDO')
    elif comando == 'L':
        print('LED APAGADO')

arduino.close() #Finalizamos la comunicacion
```

Ilustración 32: Código Python ON/OFF LED puerto serie

Este código se ha ejecutado desde el PC del aula de prácticas. Instalamos Python si no estaba instalado mediante el comando: **"sudo apt-get install python-serial"**. El programa en Python importa las funciones del puerto serie, establece conexión con Arduino por este puerto y actúa en bucle pidiendo al usuario por consola que introduzca 'H' o 'L' para encender o apagar el LED. Ejecutamos el archivo desde consola mediante **"python nombreArchivo.py"**. El resultado es el siguiente:

```
practicas@MCR-92:~$ cd Escritorio
practicas@MCR-92:~/Escritorio$ python pyth.py
Starting!
Introduce un comando: H
LED ENCENDIDO
Introduce un comando: L
LED APAGADO
```

Ilustración 33: Ejecución código Python ON/OFF LED puerto Serie

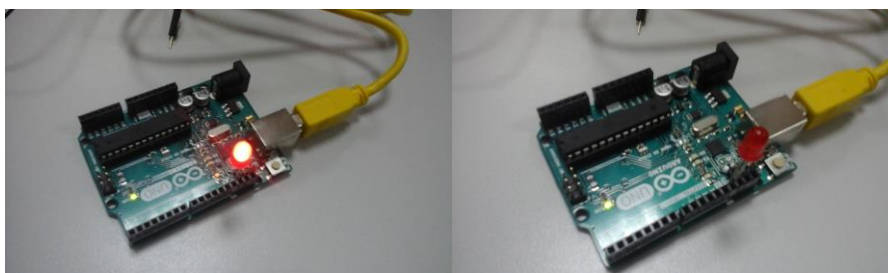


Ilustración 34: Encendido y apagado del LED respectivamente.

- **Controlar encendido-apagado de una bombilla mediante un relé:** El código empleado es el mismo que el anterior. En este caso cambia el conexionado de la parte hardware: donde antes teníamos un LED, ahora tenemos una entrada a los relés conectándose a 'IN1' o 'IN2', según escojamos uno u otro relé (ver Anexo audiovisual para resultados).

Un relé (ver **Ilustración 22**) es un dispositivo electromagnético. Este funciona como un interruptor controlado por un circuito eléctrico que acciona, mediante bobina y electroimán, uno o varios contactos que abren o cierran otros circuitos eléctricos independientes. Se le considera amplificador eléctrico en tanto que es capaz de controlar un circuito de salida de potencia mayor al que tiene como entrada.

Entrando en un aspecto más técnico, el electroimán hace girar la “armadura” verticalmente al ser alimentada, de esta forma se cierran los contactos (si es normalmente abierto). El hecho de aplicar un voltaje a la bobina genera un campo magnético que provoca la conexión de los contactos, considerados como interruptores, permitiendo fluir la corriente entre ambos.

- **Encendido de un LED mediante pulsador (uso de interrupciones):** Se ha usado un cable a modo de pulsador. El montaje queda así:

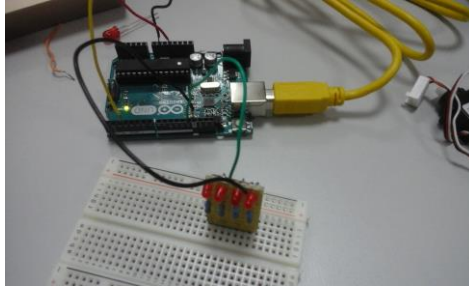


Ilustración 35: Montaje LED mediante el uso de interrupciones

El código en Arduino es el siguiente y su funcionalidad es declarar el pin que usamos como LED, el que usamos como interrupción y declarar una variable que controle el estado del LED. La interrupción se producirá cada vez que conectemos el cable a VDD.

```
const byte ledPin = 13;
const byte interruptPin = 2;
volatile byte state = LOW;

void setup() {
  pinMode(ledPin, OUTPUT);
  pinMode(interruptPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(interruptPin), blink, CHANGE);
}

void loop() {
  digitalWrite(ledPin, state);
}

void blink(){
  state = !state;
}
```

Ilustración 36: Código Arduino LED mediante el uso de interrupciones

El LED conmuta entre el estado mostrado anteriormente como montaje y este otro:

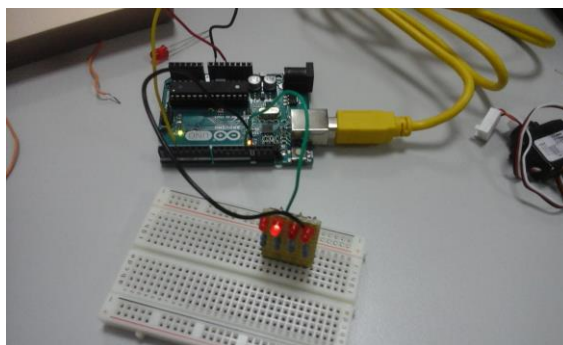


Ilustración 37: Ejecución código Arduino LED mediante el uso de interrupciones

El hecho de usar un pulsador tenía un inconveniente y es que, no siempre funciona bien a causa de rebotes, es decir, puede que se encienda o apague en función del último valor que haya tomado. Una posible solución a este problema es leer la entrada y activar “manualmente” el pin desde Arduino.

➤ **Control de la posición de un servo con un potenciómetro:**

El desarrollo de esta parte consta del control del servo mediante un potenciómetro. Esto es, según la posición del potenciómetro el servo actuará en consecuencia.

El montaje del mismo se muestra en la siguiente imagen:

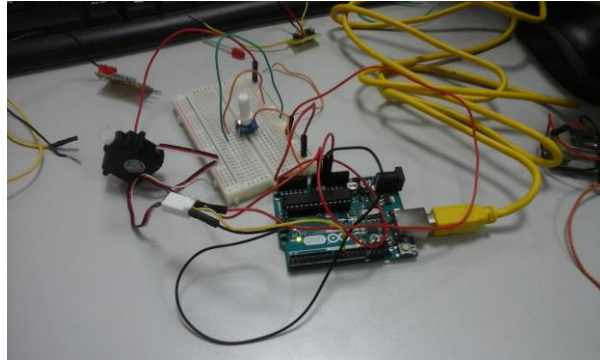


Ilustración 38: Montaje servo con potenciómetro

Un servo (servomotor de modelismo) es un dispositivo actuador que tiene es capaz de ubicarse en una posición dentro de su rango de operación, por lo general inferior a una vuelta completa (en nuestro caso lo hicimos girar hasta un máximo de 180 grados). Además se mantiene en esta de forma estable. La composición del mismo comprende a un motor de corriente continua, que recibe un voltaje entre sus dos terminales permitiendo el giro a “alta velocidad”; una caja reductora que aumenta el bajo par que produce el giro del motor mencionado antes y transforma parte de la velocidad en torsión; y un circuito de control, que ubica el motor en un punto, consistente en un controlador proporcional.

La señal cuadrada empleada para el control del mismo es una señal PWM, o de modulación por ancho de pulso, que indica el ángulo de posición: mientras más ancho sea el pulso, mayor es el ángulo donde se ubicará el motor.

El funcionamiento del servomotor de modelismo se inicia con un amplificador de error que calcula el error de posición (diferencia entre la referencia y la posición del motor). De este modo, el motor deberá rotar más rápido para alcanzarlo si es grande; uno menor, significa que la posición del motor está cerca de la deseada: el motor rotará más lentamente. No rotará si está en la posición deseada. La PWM se convierte en un valor analógico de voltaje, mediante un convertidor de ancho de pulso a voltaje, cuando resta dos valores de voltaje analógicos para calcular el error de posición. Una vez obtenido el error de posición, este se amplifica con una ganancia y se aplica a los terminales del motor.

En lo que respecta a software, el código en Arduino es este:

```
#include <Servo.h>

Servo myServo; // create servo object to control a servo

int potpin = 0; // analog pin used to connect the potentiometer
int val; // variable to read the value from the analog pin

void setup(){
  myServo.attach(9); // attach the servo on pin 9 to the servo object
}

void loop(){
  val = analogRead(potpin); // reads the value of the potentiometer (value between 0 and 1023)
  val = map(val,0,1023,0,180); //scale it to use with the servo (value between 0 and 180)
  myServo.write(val); // sets the servo position according to the scaled value
  delay(15); // waits for the servo to get there
}
```

Ilustración 39: Código Arduino control servo mediante potenciómetro

Empleamos la función ‘map’ para transformar de un rango (salida del potenciómetro digitalizada de 0 a 1023) a otro (rango de giro de 0 a 180 grados). También hacemos uso de la librería <Servo.h>, en concreto de las funciones “write”, para escribir en nuestro servo, y “attach”, para asignar el pin 9 de Arduino al servo.

➤ Control de la posición de un servo mediante puerto serie:

Ahora prescindimos del potenciómetro empleado en el apartado anterior: usamos el puerto serie para enviar el valor de giro deseado al servo.

El montaje es casi idéntico al anterior. Solo necesitamos desconectar el potenciómetro. El código en Arduino queda de la siguiente forma:

```
#include <Servo.h>

Servo myServo;

int potpin = 0;
int val;

void setup(){
  myServo.attach(9);
  Serial.begin(9600);
}

void loop(){
  if(Serial.available()){
    val = Serial.parseInt();
    myServo.write(val);
    delay(15);
  }
}
```

Ilustración 40: Código Arduino control servo mediante puerto serie.

Con el uso de `Serial.parseInt()` pasamos a enteros lo que llega por serie

En Python, el código queda de la siguiente manera:

```
import serial

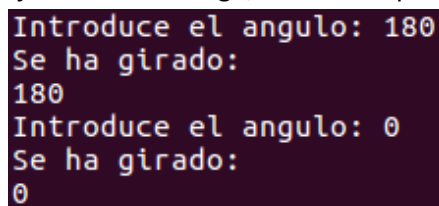
arduino = serial.Serial('/dev/ttyACM1', 9600)

print('Starting!')

while True:
    angulo = raw_input('Introduce el angulo: ') #input
    arduino.write(angulo) #Mandar un comando hacia Arduino
    print('Se ha girado: ')
    print(angulo)
arduino.close() #Finalizamos la comunicación
```

Ilustración 41: Código Python control servo mediante puerto serie

La ejecución del código, resulta tal que así:



```
Introduce el angulo: 180
Se ha girado:
180
Introduce el angulo: 0
Se ha girado:
0
```

Ilustración 42: Ejecución códigos para el control servo mediante puerto serie

➤ **Control de la velocidad de giro de un motor DC doble sentido con potenciómetro:**

En esta sección de la plataforma Arduino, introducimos dos nuevos componentes: un motor DC y un puente en H.

El primero de ellos, el motor de corriente continua (CC) o de corriente directa (DC: Direct Current del inglés) es el que convierte la energía eléctrica en mecánica, provocando un movimiento rotatorio, gracias a la acción que se genera del campo magnético.

Esta máquina se compone principalmente de dos partes: el **estator**, que da soporte mecánico al aparato y contiene los devanados principales (polos) de la máquina, compuestos por imanes permanentes o devanados con hilo de cobre sobre un núcleo de hierro; y el **rotor**, que normalmente es cilíndrico, devanado y con núcleo, es alimentado con corriente directa por escobillas fijas (carbones). El principal inconveniente es el mantenimiento: muy caro y laborioso por el desgaste que sufren las escobillas al entrar en contacto con las delgas.

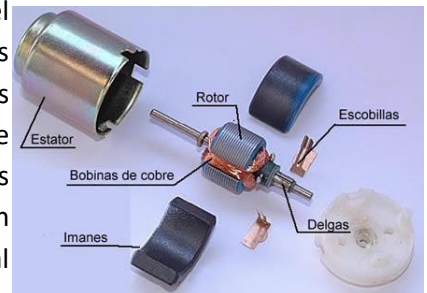


Ilustración 43: Partes de un motor DC

Los motores de corriente continua (CC) también se utilizan en la construcción de **servomotores** y **motores paso a paso**. Además existen motores de DC sin escobillas llamados **brushless** utilizados en el aeromodelismo por su bajo torque y su gran velocidad:



Ilustración 44: Motor brushless

Controlar la velocidad y el par de estos motores mediante técnicas de control de motores (CD) es posible. Por otra parte tenemos lo que se denomina un **puente en H**. Es un circuito electrónico que permite a un motor eléctrico DC girar en ambos sentidos: avance y retroceso. Muy usados en robótica y como convertidores de potencia. Se pueden encontrar como circuitos integrados o nos podemos inclinar por construir uno partiendo de componentes discretos.

El hecho de que se le denomine así es por la forma de la típica representación gráfica del circuito. Constituido por 4 interruptores (mecánicos o mediante transistores), si los interruptores S1 y S4 están cerrados y, por tanto, S2 y S3 abiertos, se aplica una tensión positiva en el motor, haciéndolo girar en un sentido. De forma contraria, el voltaje se invierte, permitiendo el giro en sentido inverso del motor. La característica que lo hace tan útil es que S1 y S2 no podrán estar cerrados al mismo tiempo, porque esto cortocircuitaría la fuente de tensión. Lo mismo sucede con S3 y S4.

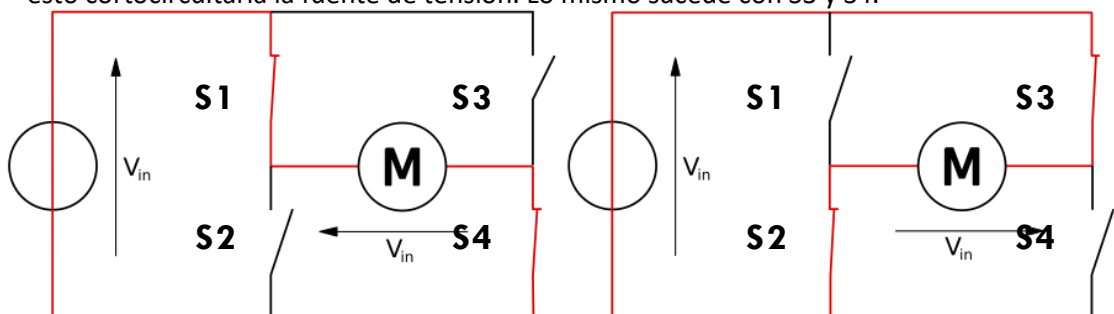


Ilustración 45: Puente en H

En nuestro caso hemos utilizado el chip L293DNE, el cual contiene dos puentes en H: podemos alternar dos motores independientes. Las señales M1 y M2 conectan con el motor DC; pwm1 y pwm2 son las salidas que controla el potenciómetro.

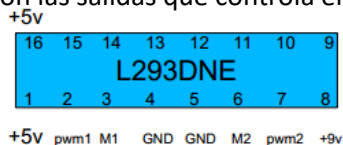


Ilustración 46: Puente en H L293DNE

El objetivo consiste en que el motor DC que vea controlada su velocidad y sentido por la pareja potenciómetro y L293DNE. Mientras más giramos el potenciómetro en un sentido más aumenta la velocidad a la que el motor DC gira en dicho sentido. El montaje queda de la siguiente manera:

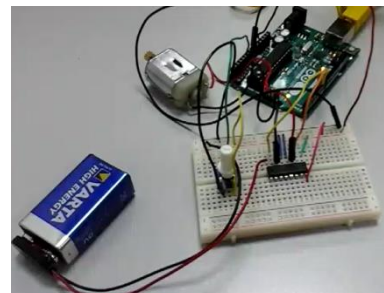


Ilustración 47: Montaje motor DC controlado por potenciómetro

En cuanto al código, utilizamos el siguiente en el entorno de Arduino:

```
int pin2=5; //Entrada 2 del L293D
int pin7=6; //Entrada 7 del L293D
int pot=A0; //Potenciometro
int valorpot; //Variable que recoge el valor del potenciómetro
int pwm1; //Variable del PWM 1
int pwm2; //Variable del PWM 2

void setup(){
  //Inicializamos los pins de salida
  pinMode(pin2,OUTPUT);
  pinMode(pin7, OUTPUT);
}

void loop(){
  //Almacenamos el valor del potenciómetro en la variable
  valorpot=analogRead(pot);

  //Como la entrada analógica del Arduino es de 10 bits, el rango va de 0 a 1023.
  //En cambio, la salidas del Arduino son de 8 bits, quiere decir, rango entre 0 a 255.
  //Por esta razón tenemos que mapear el número de un rango a otro usando este código.
  pwm1 = map(valorpot, 0, 1023, 0, 255);
  pwm2 = map(valorpot, 0, 1023, 255, 0); //El PWM 2 esta invertido respecto al PWM 1

  //Sacamos el PWM de las dos salidas usando analogWrite(pin,valor)
  analogWrite(pin2,pwm1);
  analogWrite(pin7,pwm2);
}
```

Ilustración 48: Código Arduino motor con puente en H

La ejecución del mismo se puede observar en el anexo audiovisual.

➤ Impresión en una pantalla LCD:

Seguimos conectando el Arduino con “el mundo exterior”. Introducimos el componente que nos permite visualizar texto: la pantalla LCD (Liquid Crystal Display). En concreto la pantalla LCM1602C, la cual está basada en un controlador Hitachi (HD44780). Las LCD son pantallas delgadas y planas formadas por píxeles colocados delante de una fuente de luz o reflectora que utilizan cantidades muy pequeñas de energía eléctrica.

Su estructura consta de seis partes:

1. Filtro vertical que polariza la luz que entra.
2. Sustrato de vidrio con electrodos de Óxido de Indio (ITO) que forman siluetas negras cuando la pantalla se enciende y apaga.
3. Cristales líquidos.

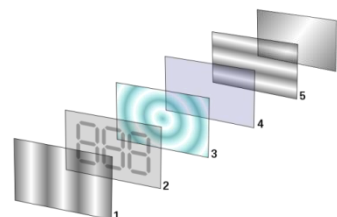


Ilustración 49: Partes LCD

4. Sustrato de vidrio con ITO para alinearse con el filtro horizontal.
5. Filtro horizontal que bloquea/permite el paso de luz.
6. Superficie reflectante que devuelve luz al espectador o fuente luminosa (LCD retroiluminado).

Existen varios tipos de LCD:

- Segmentados: los denominamos como “antiguos”. Evolucionaron del 7 segmentos a otros que logran hacer todas las letras.
- Alfanuméricos: matriz de puntos.
- Matrix: se dividen a su vez en los que llevan controladores, puertos y comandos (como el display NOKIA, sin color); y en los que no llevan controladores, y por ello necesitan de una interfaz más compleja, requieren de más memoria y son más costosos (con color).

Nuestro LCD en esta práctica será de tipo alfanumérico. Este cuenta con 3 líneas de control y 8 de datos y posee 16 posiciones horizontales y 2 verticales. En cuanto a cómo se organiza esto en memoria, este LCD tiene dos partes:

- Gráfica (Display Data RAM): cada posición de memoria representa un carácter (visualización). Esta memoria tiene autoincremento.
- Definición de caracteres (Character Generation ROM): Es una ROM en la que se guarda la definición del carácter, pero con la que no se está visualizando (gráficos). También cuenta con la CGRAM, que permite definir caracteres al usuario que ocupan las posiciones 0-9. Sería necesario cargarlos cada vez que se usen. La posición a la que apunta es el carácter en ASCII.

Este controlador Hitachi puede controlar displays de mayor dimensión, por eso es posible escribir más allá de la posición 15 (solamente que no lo visualizaremos en el que usamos). Una alternativa posible es utilizar **scroll horizontal** (como se utilizaba en la Game Boy). En otras palabras: la memoria gráfica es mayor que el display.

Los pines de la pantalla LCD tienen el siguiente significado (de izquierda a derecha):

- Pin 1: GND, pin conectado a tierra.
- Pin 2: VDD, pin de alimentación de tensión (5V normalmente).
- Pin 3: V_o, pin de contraste del display.
- Pin 4: RS (Register Select), controla la memoria del LCD: indica en qué registro se escribe o se lee: se obtienen los datos a mostrar y las instrucciones que el controlador del LCD necesita para actuar.
- Pin 5: RW (Read/Write), indica si se lee o escribe (de) en memoria.
- Pin 6: E (Enable), habilita los registros.
- Pines 7-14: Son los pines de datos, designados como DB0-DB7.
- Pines 15 y 16: BL1 y BL2 (Backlights), son los pines de retroiluminación. Controlan la luminosidad del LCD.

La conexión con los pines de Arduino es la siguiente:

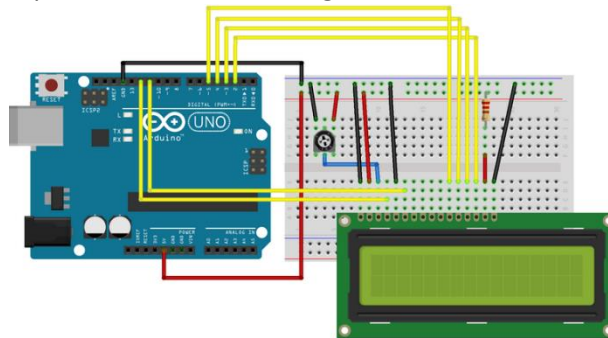


Ilustración 50: Montaje LCD

En este ejercicio hacemos uso de la librería “<LiquidCrystal.h>” de Arduino para el manejo del LCD. En concreto, durante el mismo, obtenemos la habilidad para movernos por el display LCD entre las dos filas y las dieciséis columnas que posee y para imprimir directamente en el LCD. El código en Arduino es el siguiente:

```
#include <LiquidCrystal.h> //Biblioteca necesaria para LCD

//Iniciamos los pines a utilizar
LiquidCrystal lcd(12,11,5,4,3,2);

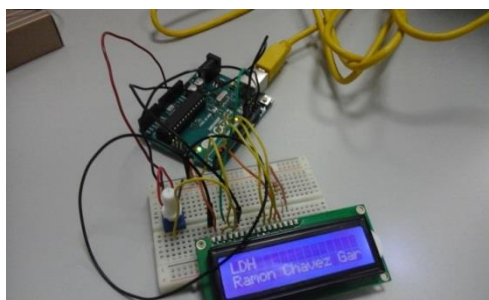
void setup() {
  //Expresamos el numero de columnas y filas de nuestro LCD
  lcd.begin(16, 2);
  //Imprimimos el mensaje deseado en el LCD
  lcd.print("LDH");
  lcd.setCursor(0,2);
  lcd.print("Ramon Chavez Garcia");
}

void loop() {
  //Preparamos el cursor en la columna 0 de la linea 1
  //Esto indica donde comenzara a imprimirse el texto
  lcd.setCursor(0,1);
}
```

Ilustración 51: Código Arduino LCD sin usar puerto serie

La librería mencionada al comienzo de este apartado nos proporciona la función “**.begin (nColumnas, nFilas)**”, que nos permite inicializar el LCD, dimensionándolo. También contamos con la función “**.setCursor (nColumna, nFila)**”, que nos permite situar el cursor en una posición estricta para comenzar a escribir.

La ejecución del mismo sería esta:



Podemos apreciar también el montaje del mismo, donde los pines quedaron aclarados anteriormente.

Ilustración 52: Montaje y ejecución de la impresión en pantalla LCD sin usar puerto serie

➤ **Impresión en una pantalla LCD usando el puerto serie:**

Introducimos una pequeña variación, que afecta sobre todo al código, pues el montaje no se ve afectado y la ejecución no cambia esencialmente. La variación presenta el uso del lenguaje Python para imprimir mediante el puerto serie.

El código en Arduino es este:

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(12,11,5,4,3,2);
void setup() {
  lcd.begin(16,2);
  Serial.begin(9600);
}

void loop() {
  if(Serial.available()) {
    delay(100);
    lcd.clear();
    while(Serial.available() > 0) {
      lcd.write(Serial.read());
    }
  }
}
```

Ilustración 53: Código Arduino LCD usando puerto serie

Para poder tener éxito con este código, es necesario recurrir a Python mediante:

```
import serial
arduino = serial.Serial('/dev/ttyACM1', 9600)
print("Starting!")
while True:
    comando = raw_input('Introduce palabra: ') #Input
    arduino.write(comando) #Mandar un comando hacia Arduino
    print(comando);
arduino.close()
```

Ilustración 54: Código Python LCD usando puerto serie

La ejecución del mismo queda así:

```
practicas@MCR-92:~/Escritorio$ python lcd.py
Starting!
Introduce palabra: Hola LDH!
Hola LDH!
Introduce palabra: █
```

Ilustración 55: Ejecución en consola de LCD usando puerto serie

Un código prácticamente idéntico al utilizado para el control de la posición del servo con puerto serie. Durante la ejecución de los mismos solo podemos constatar la diferencia de texto, ya que el display recibe la información de forma distinta, pero la muestra igualmente:

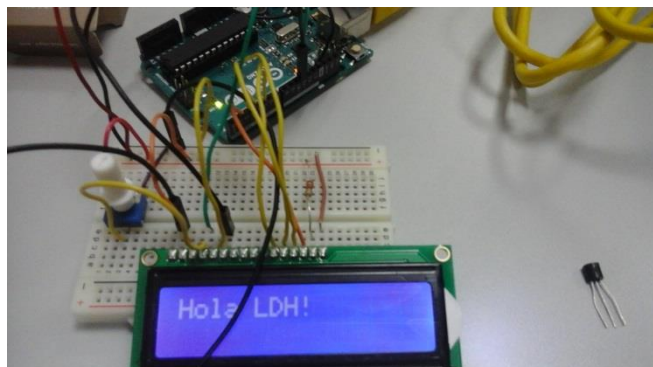


Ilustración 56: Montaje y ejecución LCD usando puerto serie

➤ Impresión de la temperatura en una pantalla LCD con el sensor TMP36GZ:

En la última imagen se dejaba entrever un elemento que hemos utilizado durante esta sección: el TMP36GZ, un sensor de temperatura. Con él, medimos la temperatura ambiente (relativamente bien: no requiere calibración externa para dar medidas de $\pm 1^\circ\text{C}$ a $+25^\circ\text{C}$ y $\pm 2^\circ\text{C}$ pasado el rango -40° hasta $+125^\circ\text{C}$) y la mostramos por pantalla.

El montaje del mismo, así como su ejecución, lo podemos encontrar en el anexo audiovisual. El código en Arduino es el siguiente:

```

#include <LiquidCrystal.h>
LiquidCrystal lcd(12,11,5,4,3,2);
int sensorPin = 0;

void setup() {
  lcd.begin(16,2);
}

void loop() {
  float sensorVal;
  float temperatura;
  lcd.setCursor(0,0);
  sensorVal = analogRead(sensorPin);
  lcd.setCursor(0,0);
  lcd.write("Temperatura: ");
  temperatura = ((sensorVal/1024)*5 - .5)*100;
  lcd.setCursor(0,1);
  lcd.print(temperatura);
  lcd.setCursor(5,1);
  lcd.write((char)223);
  lcd.setCursor(6,1);
  lcd.write("C");
  delay(2000);
}

```

Ilustración 57: Código Arduino temperatura por LCD

➤ **Contador de 00 a 59 en display 7 segmentos:**

Es la última parte de la plataforma Arduino, así que, para darle un gran final, empleamos uno de los mencionados shields de Arduino: un shield de un display 7 segmentos. La particularidad de este shield es que incluye el display 7 segmentos doble.

Consta de 16 segmentos (7 segmentos de cada uno y el 'punto' de ambos) independientes, cada uno, que permiten formar números e incluso, con un poco de ingenio, letras:

| Números | | | | | | | | | |
|---------|-----|-----|------|--------|-------|------|-------|------|-------|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Cero | Uno | Dos | Tres | Cuatro | Cinco | Seis | Siete | Ocho | Nueve |

| Alfabeto latino | | | | | | | | | |
|-----------------|---------|------|------|------|------|------|------|------|--|
| A a, @ | B, b | C, c | D, d | E, e | F, f | G, g | H, h | I, i | |
| J, j | K, k | L, l | M, m | N, n | Ñ, ñ | O, o | P, p | Q, q | |
| R, r | S, s, f | T, t | U, u | V, v | W, w | X, x | Y, y | Z, z | |

Ilustración 58: Números y letras en display 7 segmentos

Internamente, están constituidos por una serie de leds con conexiones internas, estratégicamente ubicados formando un número '8'. En muchos casos aparece un octavo segmento: DP (Decimal Point). La identificación de los segmentos es la siguiente:

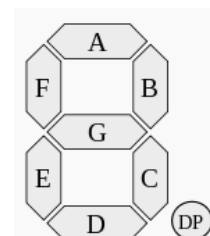


Ilustración 59: Identificación de las partes del 7 segmentos

Los leds trabajan a baja tensión y con pequeña potencia, por tanto, podrán excitarse directamente con puertas lógicas. Normalmente se utiliza un codificador BCD que activando una sola pata de la entrada del codificador, activa las salidas correspondientes mostrando el número deseado. También que existen pantallas alfanuméricas de 16 segmentos e incluso de una matriz de 7*5 (35 bits).

Existen de dos tipos:

- **Ánodo común:** todos los ánodos de los segmentos están unidos internamente y conectados a un potencial positivo (uno lógico). El encendido de cada uno se lleva a cabo aplicando un potencial negativo (cero lógico) a la patilla correspondiente mediante una resistencia que limite el paso de la corriente.
- **Cátodo común:** todos los cátodos de los segmentos están unidos internamente y conectados a un potencial negativo (cero lógico). El encendido de cada uno se lleva a cabo aplicando un potencial positivo (uno lógico) a la patilla correspondiente mediante una resistencia que limite el paso de la corriente.

Los segmentos pueden ser de diversos colores, aunque normalmente son de color rojo, por su facilidad de visualización. El visualizador de 14 segmentos tuvo menor éxito y existe de forma marginal a causa de la competencia de la matriz 5 x 7 puntos. Aunque parezcan antiguos u obsoletos, son una excelente opción en situaciones que requieran mayor poder lumínico y trabajo en áreas donde pantallas de píxeles podrían verse afectadas por condiciones ambientales adversas.

En nuestro caso lo hemos empleado para simular un contador en el que la cuenta comienza en 00 y va avanzando de uno en uno hasta 59, al ritmo de un cambio por segundo. Para ello, necesitamos en primer lugar, una función capaz de transformar los valores digitales en hexadecimales: **void write_data (int arg)**. Una vez tenemos esta función, necesitamos emplear otra para escribirlo en el display 7 segmentos: **void write7seg (unsigned char arg)**. Esta función, recorre el argumento recibido (en hexadecimal) bit a bit, de forma que el bit a '0' deja el LED correspondiente apagado, y el bit a '1' lo enciende. A estas dos funciones le tenemos que sumar la función **void refresh (int data1, int data0)**. Esta función cumple el cometido de seleccionar el display sobre el que vamos a escribir, alternándolos, y de refrescar la información enviada a estos de forma que podamos controlar el parpadeo. Las funciones son las siguientes:

```
void write_data(int arg){
  switch(arg){
    case 0:
      write7seg(0x7e);
      break;
    case 1:
      write7seg(0x30);
      break;
    case 2:
      write7seg(0x6d);
      break;
    case 3:
      write7seg(0x79);
      break;
    case 4:
      write7seg(0x33);
      break;
    case 5:
      write7seg(0x5b);
      break;
    case 6:
      write7seg(0x1f);
      break;
    case 7:
      write7seg(0x70);
      break;
    case 8:
      write7seg(0x7f);
      break;
    case 9:
      write7seg(0x73);
      break;
  }
}
```

Ilustración 62: Código Arduino write_data (...)

```
void write7seg (unsigned char arg) {
  unsigned char segmen = 0x01;
  unsigned char display1;
  display1 = arg;
  for (int i = 0; i < 8; i++) {
    if ((display1 & segmen) == 0x00)
      digitalWrite(segPins[i], LOW);
    else
      digitalWrite(segPins[i], HIGH);
    segmen <<= 1;
  }
}
```

Ilustración 60: Código Arduino write7seg (...)

```
void refresh (int data1, int data0){
  int i;
  int j=40;
  int tiempo_refresco = tiempototal/(2*j);
  for(i=0; i<j;i++){
    delay(tiempo_refresco);
    digitalWrite(displ1, 1);
    digitalWrite(displ2, 0);
    write_data(data1);
    delay(tiempo_refresco);
    digitalWrite(displ1, 0);
    digitalWrite(displ2, 1);
    write_data(data0);
  }
}
```

Ilustración 61: Código Arduino refresh (..., ...)

A esto hay que añadirles las variables globales y las funciones **void setup()**, **void loop()**:

```
int segPins[]={0,1,2,3,4,5,6,7};
int tiempoTotal=1000;
int disp1=8;
int disp2=9;
int dat1=0;
int dat0=0;
int tiempoMax=60;

void setup(){
  for (int seg = 0; seg < 8; seg++) {
    pinMode(segPins[seg], OUTPUT);
  }
  pinMode(disp1, OUTPUT);
  pinMode(disp2, OUTPUT);
}

void loop(){
  int valor;
  while(1){
    for (valor=0; valor<tiempoMax; valor++) {
      dat1=valor/10;
      dat0=valor%10;
      refresh(dat1,dat0);
    }
  }
}
```

Ilustración 63: Código Arduino loop() del 7 segmentos

Ilustración 64: Código Arduino variables globales y setup() del 7 segmentos

Con todo este código, lo cargamos en nuestro Arduino y vemos el resultado del anexo audiovisual. La variable '**tiempoMax**' indica el valor máximo al que llegará el display a contar, '**dat1**' y '**dat0**' corresponden, respectivamente, a los valores que alcanzan las decenas y unidades de los segundos. Durante el transcurso del ejercicio se pedía que el incremento del contador respetase un segundo entre un cambio y otro (variable '**tiempoTotal**' y el arreglo en la función '**refresh (...,...)**': hay dos delays de 12,5 milisegundos ('**tiempo_refresco**') y el bucle se repite 40 veces: $12,5 * 2 * 40 = 1000 \text{ ms} = 1 \text{ segundo}$).

Además, se pidieron modificaciones como observar el parpadeo de refresco o que la cuenta empezara y acabase en otros números.

- El primero de los cambios se consigue cambiando el número de iteraciones del bucle, es decir, tenemos que conseguir que el bucle se ejecute menos veces pero que mantenga los incrementos en un segundo: para esto, la '**j**' pasa a tener el valor de 20: '**tiempo_refresco**' tiene el valor de 25ms y el bucle se repite 20 veces. Esto es: $25 * 2 * 20 = 1000 \text{ ms} = 1 \text{ segundo}$).
- El segundo de ellos basta con reemplazar '**valor**' y '**tiempoMax**' por los valores deseados en la función '**loop ()**'.
- Otro cambio puede ser el de hacer que los números no respeten los segundos y avancen más rápido o más lento. Esto lo conseguimos cambiando el factor que multiplica a '**j**'. Si ponemos un '**4**', entonces el bucle en '**refresh (...,...)**' se completará cada 500ms. Por el contrario, si eliminamos el factor multiplicador, lo completará cada 2 segundos.

El montaje y la ejecución de los ejercicios del display 7 segmentos pueden verse en el anexo audiovisual.

Plataforma ZPUino

INTRODUCCIÓN: PLATAFORMA Y ENTORNO DE TRABAJO

Papilio es una plataforma de desarrollo hardware y software libre que pone el asombroso poder de una **FPGA** (Field Programmable Gate Array) en nuestras manos. Permite dibujar los circuitos que queremos usar mediante herramientas software y hacer uso de los mismos. Es decir, la plataforma abstrae la FPGA para que cualquiera pueda usar las ventajas de la misma en sus proyectos.

Entre ellas, están las bibliotecas “drag and drop”, o “arrastra y suelta”, que permiten escoger el componente y ponerlo directamente en nuestro circuito (permite visualizarlo fácilmente).

Al igual que Arduino, la plataforma Papilio permite usar sus pines como expansión. En lugar de llamarse ‘shields’, estos se llaman ‘MegaWings’ y proveen a la placa Papilio del hardware con el que no cuenta: conector VGA, DAC, ratón, teclado, Joystick, botones, LEDs, displays 7 segmentos... Además, Papilio cuenta con ‘mini-shields’, o lo que ellos denominan como ‘Wings’, que son unas placas de expansión de menor tamaño. Podemos decir, en este sentido, que la plataforma Papilio es modulable. No obstante, al igual que Arduino, puede tomar y enviar información con el entorno mediante sus pines de entrada. Aquí podemos observar la diferencia entre las **Wings** (a la izquierda) y las **MegaWings** (a la derecha):

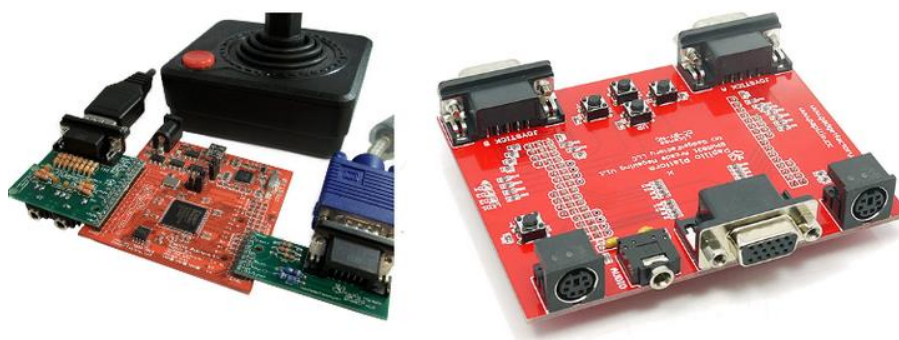


Ilustración 65: Wings vs. MegaWings

Papilio emplea un **Soft Core** porque se puede manejar con código C, evitando el uso de VHDL, que es más complejo para aprender y usar. Un Soft Core es un microprocesador que puede ser enteramente implementado usando síntesis lógica, en nuestro caso bajo la lógica programable de una FPGA. Se decidió a usar un AVR Core (AVR8) porque el sintetizado era fácil, estaba bien organizado y era sencillo de entender. Además, contaba con todos los periféricos de AVR (UART, Timers, SPI, GPIO) y era muy parecido al utilizado por el Arduino original. Es más, lo que lo diferencia del Arduino es la habilidad de desplazar físicamente los pines de SPI, UART o PWM mediante programación y crear tu funcionalidad personalizada en el espacio de memoria del AVR8.

El proyecto oferta una serie de placas:

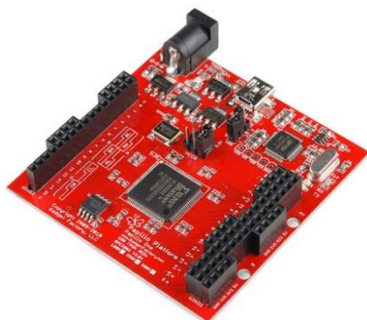


Ilustración 67: Papilio One



Ilustración 66: Papilio DUO



Ilustración 68: Papilio Pro

También ofertan Wings y MegaWings: <http://papilio.cc/index.php?n=Papilio.Hardware>

En el laboratorio empleamos la placa Papilio One 500K (hay otra versión de 250K). Su core tiene un chip de la FPGA Xilinx Spartan 3E. Posee 4Mb de memoria flash SPI, dos canales de conexión USB, para JTAG y comunicaciones serie, cuatro raíles independientes de tensión a 5V, 3.3V, 2.5V, and 1.2V, alimentación por USB o externa y 48 líneas de I/O.



Ilustración 69: Xilinx Spartan 3E

Podemos generar cualquier señal de reloj entre 5 y 300MHz usando el DCM (Digital Clock Manager). Son cuatro los DCM que permiten generar multitud de señales de reloj provenientes del oscilador externo de 32 MHz incluido en nuestra placa Papilio One. Más detalles sobre el hardware de la placa en:

<http://papilio.cc/index.php?n=Papilio.PapilioOne>

En ese mismo enlace podemos encontrar los esquemáticos, como también ocurría en el caso de Arduino. Los componentes electrónicos que hemos empleado en el desarrollo de las prácticas de ZPUino son algunos de los ya utilizados con Arduino, por lo que no será necesaria la presentación de muchos de ellos.

La plataforma ZPUino es un SoC (System on Chip) basado en el μ procesador ZPU de Zylins. Cuenta con un Soft Core de 32 bits que funciona a 96 MHz. El entorno que utiliza es una versión modificada del IDE de Arduino con el que cargamos los sketches. Posee un bus Wishbone, un bus de interconexión open source que permite conectar de forma estándar periféricos a un Soft Core dentro de una FPGA. Hemos trabajado esta plataforma desarrollando periféricos personalizados para la conexión con este Wishbone en lugar del AVR8. La plataforma ZPUino fue desarrollada por Alvaro Lopes basándose en el entorno de Arduino, el proyecto de CPU ZPU y las placas Papilio.



Ilustración 70: Logo ZPUino

Al igual que utilizamos en el despliegue de las prácticas de Arduino, usamos Ubuntu para trabajar con el entorno de desarrollo.

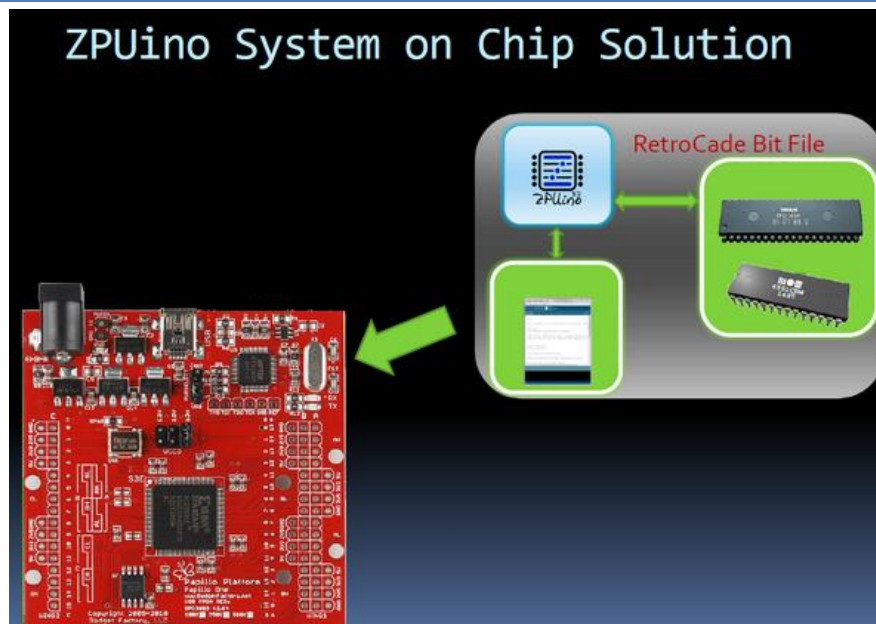


Ilustración 71: Diagrama SoC ZPUino

ZPUino cuenta con DesignLab, entorno para rediseñar circuitos que puede usarse bajo Linux o Windows. La instalación del mismo la llevamos a cabo descargando el archivo "**DesignLab-1.0.7-linux64.tgz**" desde <http://10.1.15.78/~bellido> (en nuestro caso, aunque también tenemos la opción de poderlo descargar de la página oficial: <http://forum.gadgetfactory.net/index.php?/files/file/236-papilio-designlab-ide>). Tras descargar el archivo, situamos un terminal de Ubuntu sobre la carpeta donde se encuentra y lo descomprimos con el comando **"tar -xvzf DesignLab-1.0.7-linux64.tgz"**. Cambiamos a la carpeta que se nos ha creado con **"cd DesignLab-1.0.7"** y ejecutamos el comando **"sudo ./ubuntu-setup.sh"**.

El siguiente paso es instalar el JRE (Java Runtime Environment), necesario para el funcionamiento de DesignLab. El comando a ejecutar es **"sudo apt-get install default-jre"**.

Durante las prácticas no ha sido necesario instalar el ISE de Xilinx porque ya estaba instalado en los PC del laboratorio, no obstante, su procedimiento comenzaría con un registro en la página de Xilinx <https://www.xilinx.com/registration/create-account.html>. Una vez hecho esto, nos descargamos el WebPack desde <https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools.html>:

ISE Design Suite - 14.7 Full Product Installation

Last Updated October 2013

As of October 2013, ISE has moved into the sustaining phase of its product life cycle, and there are no more planned ISE releases.

ISE supports the following devices families and their previous generations: Spartan-6, Virtex-6, and Coolrunner. For more information, please visit the ISE Design Suite.

Xilinx recommends Vivado Design Suite for new design starts with Virtex-7, Kintex-7, Artix-7, and Zynq-7000.

| Download Type | Full Product Installation |
|--|---------------------------|
| Last Updated | Oct 23, 2013 |
| Full DVD Single File Download Image (TAR/GZIP - 7.78 GB) | |
| MD5 SUM Value: bfe4e9c3cd8d2d7024163ca140113d25 | |
| Full Installer for Linux (TAR/GZIP - 6.09 GB) | |
| MD5 SUM Value: e8065b2ffb411bb74ae32efa475f9817 | |
| Full Installer for Windows (TAR/GZIP - 6.18 GB) | |
| MD5 SUM Value: 94f40553a93dfbeca642503e2721b270 | |

Ilustración 72: Descargar ISE

Descomprimos el archivo de forma similar a como hemos hecho anteriormente y situamos un terminal en la carpeta descomprimida. Ejecutamos el comando **"sudo ./xsetup"**. Aceptamos los términos de licencia y escogemos la opción **"ISE WebPack"** y pulsamos en **"Next"** hasta que se complete la instalación. Una vez terminado este proceso, se nos abre una ventana para adquirir la licencia. En nuestro caso pulsamos sobre **"Get Free ISE WebPack License"**. Nos dirigimos a la página <http://www.xilinx.com/getlicense>, seleccionamos **"ISE Design Suite: WebPACK License"** y hacemos click sobre **"Generate Node-Locked License"**. Tras pulsar **"Next"** un par de veces, tendremos la licencia en nuestro correo, así que abrimos nuestro correo y descargamos el archivo **"Xilinx.lic"**. Este archivo lo ponemos en la ventana que se nos abrió anteriormente y tendremos configurado completamente el ISE.

Una vez instalado todo, ejecutamos el programa desde la carpeta donde extrajimos DesignLab y ejecutamos desde terminal en dicha ruta con **"./DesignLab"**.

El entorno de trabajo que se nos presenta es muy similar a Arduino:

Comprobamos que en Archivo -> Preferencias, la ruta del ISE es esta:
/opt/Xilinx/14.7/ISE_DS/ISE/bin/lin64/

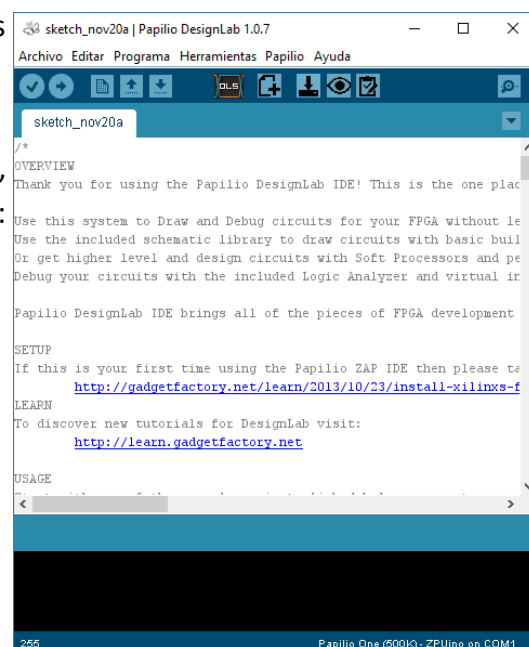


Ilustración 73: IDE ZPUino

La configuración de nuestra placa es muy sencilla de hacer:

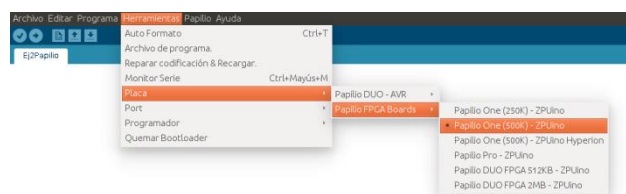


Ilustración 74: Selección de la placa en plataforma ZPUino

El puerto lo seleccionamos de igual forma que lo hacíamos en Arduino:



Ilustración 75: Selección del puerto en la plataforma ZPUino

DESARROLLO PRÁCTICO

➤ Diseñando circuitos básicos sobre la FPGA:

Gracias a DesignLab podemos diseñar circuitos para su posterior implementación en la FPGA. En este primer ejercicio, tras configurar nuestra placa, hemos probado a encender y apagar un LED con un switch como toma de contacto.

En primer lugar, creamos un nuevo proyecto de circuito FPGA pulsando sobre el icono con el fondo resaltado en blanco:



Ilustración 76: Creación proyecto FPGA

Se nos crea un sketch (programa) en blanco que no podremos editar hasta que lo guardemos, ya que es un archivo de 'sólo lectura'. Una vez hecho esto, podemos seguir con el siguiente paso, que es editar el circuito abriendo el ISE instalado anteriormente:



Ilustración 77: Editar circuito ZPUino

Hacemos doble clic sobre el archivo “Papilio_One_500K.sch” para situarnos en el nivel más alto y abrir el editor esquemático:

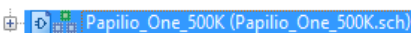


Ilustración 78: Abrir el editor esquemático

Creamos un sencillo circuito que tome una entrada, la invierta y la ponga en la salida. Para ello, clicamos sobre la pestaña “**Symbols**” y buscamos el inversor:

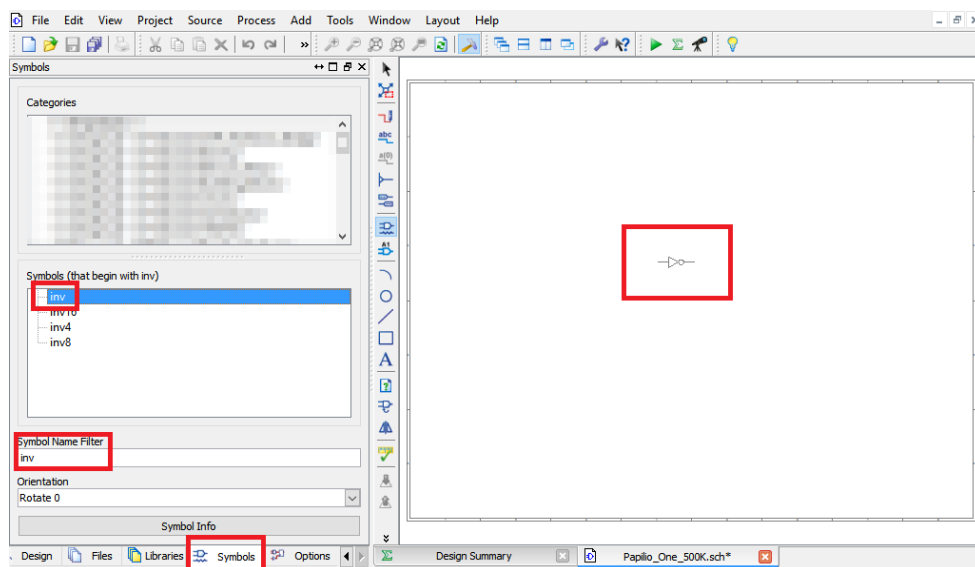


Ilustración 79: Arrastrando el inversor

Arrastramos hasta la ventana de trabajo y conectamos dos marcas de I/O, que indican al software qué pines externos conectas a tu FPGA. Es necesario ver que pines tiene disponibles Papilio, así que nos dirigimos a la pestaña “**Utility**”, que nos muestra los pines disponibles. Usamos “**WING_AL0**” y “**WING_AL1**”, disponibles según la imagen como Papilio One and Pro I/O Connectors:



Ilustración 80: Ver los pines disponibles

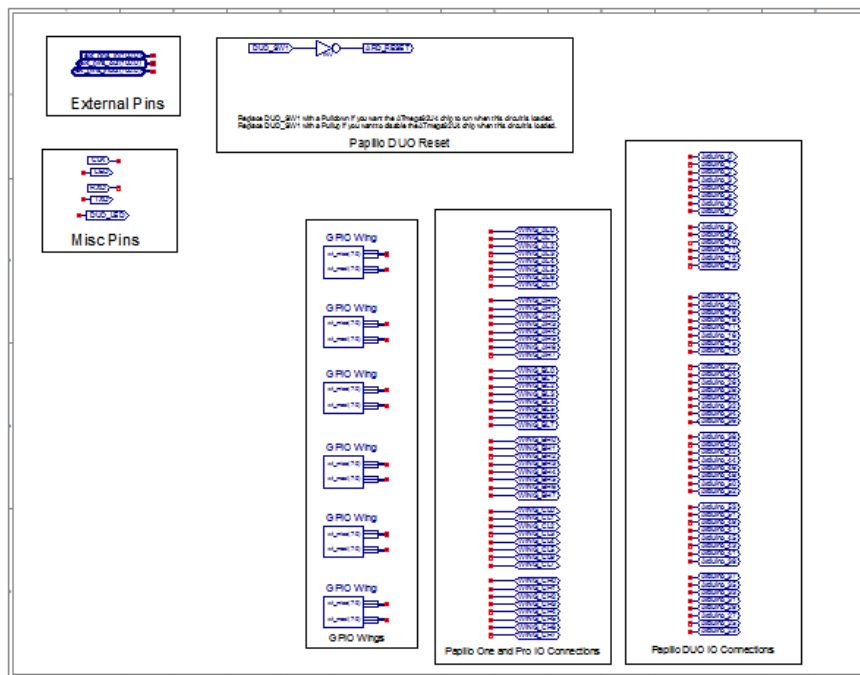


Ilustración 81: Conexiones Papilio

Así pues, de vuelta al editor esquemático, renombramos las marcas de I/O que colocamos anteriormente al inversor con el nombre de los pines que hemos dicho que vamos a usar pulsando con el botón derecho sobre las etiquetas y eligiendo “Rename Port”:

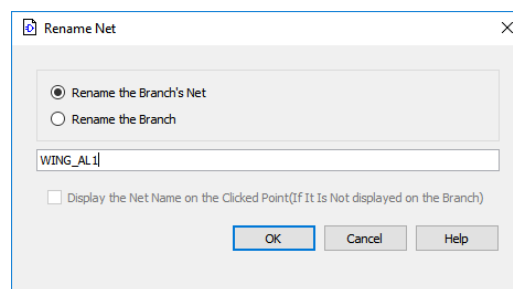


Ilustración 82: Renombrando I/O Papilio

De esta forma, a “WING_AL1” conectamos un LED y a “WING_AL0” conectamos el switch. Para terminar, vamos a la pestaña “Design”, seleccionamos el “Papilio_One_500K.sch” y damos doble click sobre “Generate Programming File” para sintetizar el circuito. La ventana de la consola muestra este mensaje: “Process “Generate Programming File” completed succesfully”, tras un tiempo. Sería así:



Ilustración 83: Diseño inversor

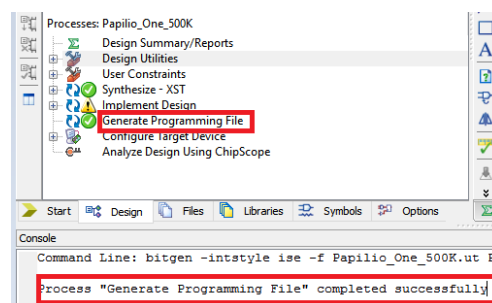


Ilustración 84: Generación de programa DesignLab

Una modificación muy importante a la hora de cargar correctamente el bitfile que genera ISE es, dentro de la carpeta “<proyecto>/circuit/500K/” borrar el archivo “papilio_one_500k.bit” y además renombrar el archivo “Papilio_One_500K.bit” a “papilio_one_500k.bit”. Esta diferencia de nombres impide la carga correcta del ‘.bit’ a la placa.

Volvemos al sketch inicial y, asegurándonos que hemos conectado nuestra Papilio al puerto correcto, hacemos click en “Load Circuit”. El circuito está cargado si aparece el mensaje “Done burning bitfile” y algo como esto:



Ilustración 85: Carga del circuito en la placa

```

Done burning bitfile
.....OK
Verifying :
.....Pass
Done.
SPI execution time 19042.1 ms
USB transactions: Write 6850 read 4681 retries 0
Using devlist.txt
JTAG chainpos: 0 Device IDCODE = 0x24001093 Desc: XC6SLX9

Using devlist.txt
ISC_Done = 0
ISC_Enabled = 0
House Cleaning = 1
DONE = 0
  
```

Ilustración 86: Ejecución de la carga del circuito en la placa

El resultado del ejercicio puede verse en el anexo audiovisual.

➤ Cargar SoC ZPUino y desarrollar sketches:

Durante este ejercicio, aprendemos a usar un ejemplo y cargar un circuito FPGA en la placa Papilio. El **sketch** a cargar es “**Papilio_Quickstart**”: Desde el menú Archivo -> Ejemplos -> Papilio_Quickstart

Una vez lo tengamos en nuestro proyecto, seleccionamos la placa y el puerto serie al que previamente debemos haberla conectado, como se indicaba en la introducción a la plataforma. De igual forma, cargamos el circuito FPGA a nuestra Papilio.

Finalmente, cargamos el sketch indicado anteriormente:



Ilustración 87: Carga del sketch en ZPUino

Comprobamos que nuestro sketch está corriendo adecuadamente abriendo el monitor serie:

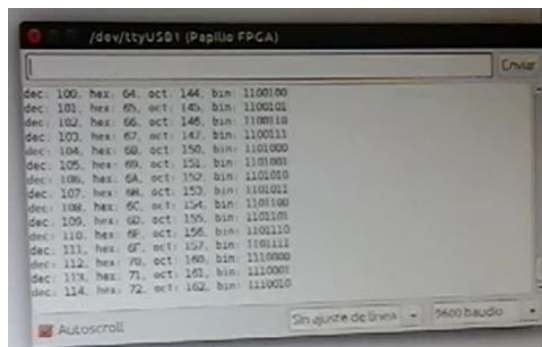


Ilustración 88: Monitor serie Papilio_Quickstart

Para comprobar el correcto funcionamiento, hemos colocado un LED en el pin 0 y un switch al pin que controla dicho LED, es decir, al pin 1. Observamos la salida del monitor serie (ver anexo audiovisual).

Para que el LED quede apagado o encendido en función de la posición del switch, alteramos el código así (hemos suprimido el ‘for’ para evitar mostrar todos los LEDs):

```

//This section blinks the LED's and keeps them solid if a button is pressed.
delay(200); // wait for a second
ledState = !ledState;
//for (int thisPin = 0; thisPin < buttonCount; thisPin++) {
// read the state of the pushbutton value:
buttonState = digitalRead(buttonPins[0]);

// check if the pushbutton is pressed.
// if it is, the buttonState is HIGH:
if (buttonState == HIGH) {
// turn LED on:
digitalWrite(ledPins[thisPin], HIGH);
Serial.println("LED: Encendido");
}
else {
// toggle LED:
digitalWrite(ledPins[thisPin], LOW);
Serial.println("LED: Apagado");
}
delay(3000);
//}
  
```

Ilustración 89: Modificación para mostrar estado LED ZPUino

Finalmente, la última modificación es poder controlar el LED por el puerto serie, esto es, desde la línea de comandos y evitando el uso del switch. Para ello empleamos los siguientes códigos en el IDE de ZPUino y en Python:

```
int led = 0;

void setup(){
  pinMode(led,OUTPUT);
  Serial.begin(9600);
}

void loop(){
  if(Serial.available()){
    char c = Serial.read();
    if(c == 'H'){
      digitalWrite(led,HIGH);
    }else if(c == 'L'){
      digitalWrite(led,LOW );
    }
  }
}
```

Ilustración 91: Código ZPUino
LED ON/OFF puerto serie

```
import serial

zpuino = serial.Serial('/dev/ttyUSB2', 9600)

print("Starting!")

while True:
  comando = raw_input('Introduce un comando: ') #Input
  zpuino.write(comando) #Mandar un comando hacia ZPUino
  if comando == 'H':
    print('LED ENCENDIDO')
  elif comando == 'L':
    print('LED APAGADO')
  zpuino.close() #Finalizamos la comunicacion
```

Ilustración 90: Código Python LED ON/OFF puerto serie Papilio

La ejecución final queda así:

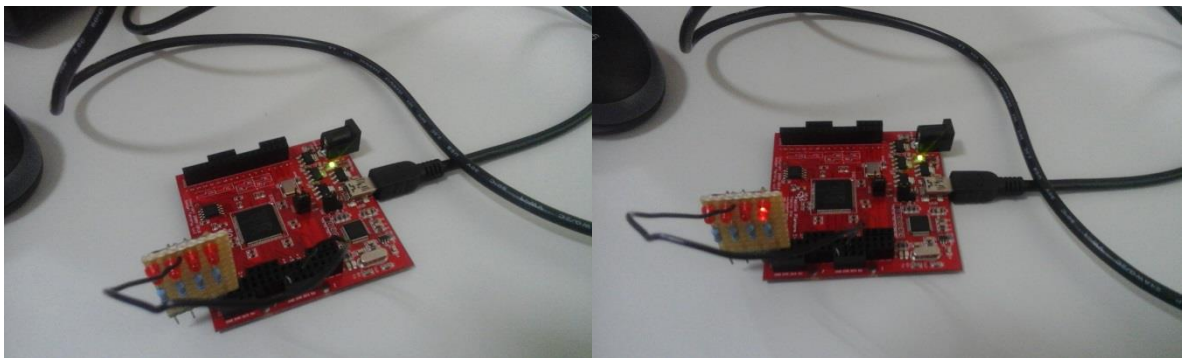


Ilustración 92: Ejecución LED ON/OFF por puerto serie Papilio

➤ Convirtiendo la placa Papilio en un analizador lógico:

El analizador lógico es capaz de capturar datos de un circuito digital y mostrarlos, para su posterior análisis, de modo similar a como lo hace un osciloscopio. La diferencia con este radica en que es capaz de visualizar las señales de múltiples canales. Además, puede medir la cantidad de estados lógicos, los tiempos entre cambios de nivel...

Para la ejecución de la práctica, existió un problema con el cliente software del analizador lógico con Java y fallaba la correcta ejecución en los PCs del laboratorio. La solución fue descargar el archivo <https://10.1.15.75/~bellido/ols-0.9.7.2-full.tar.gz> y, tras su extracción, ejecutar desde dicha carpeta extraída el comando **“sudo ./run.sh”**.

Nuestra placa Papilio puede funcionar como analizador lógico, siguiendo estos pasos.

En primer lugar, tras seleccionar nuestra placa y puerto, pulsamos sobre el icono del analizador lógico:



Ilustración 93: Icono analizador lógico

Después de esto, y tras aceptar sobrescribir el circuito, nos aparecerá esta pantalla de confirmación: **“Channels 0-15 are connected to the C Wing and channels 16-31 are connected to the A Wing.”**. Entonces, se nos abrirá el analizador lógico y pulsaremos dentro del cuadrado rojo, esto es, en **“Start capturing data from the logic analyzer”**:

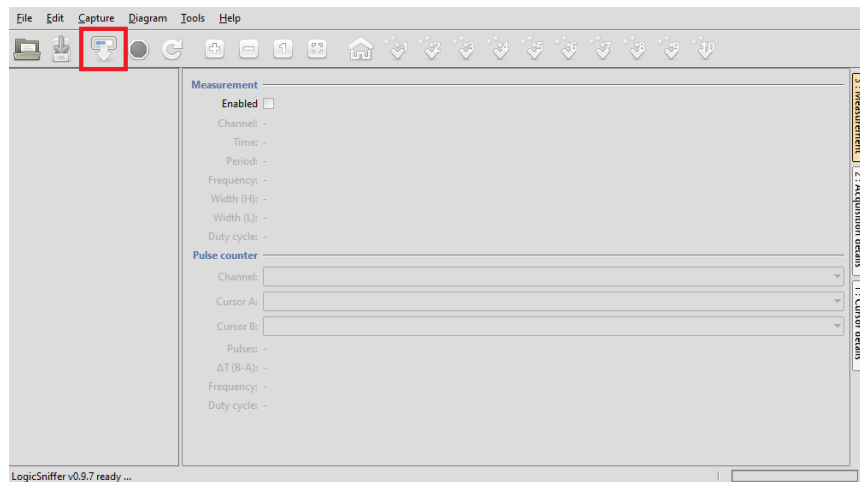


Ilustración 94: Analizador lógico

La configuración del mismo viene dada en estas dos capturas:

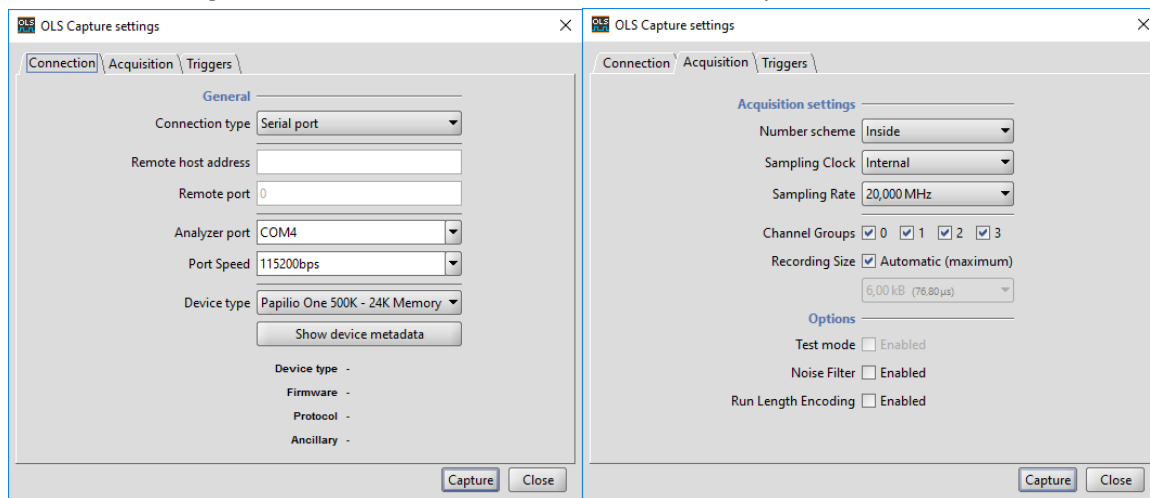


Ilustración 95: Configuración conexión analizador lógico

Para hacer nuestra Papilio con esta función, es necesario cargar el sketch de Fade en Arduino, placa que conectamos a Papilio:

```

/*
Fade

This example shows how to fade an LED on pin 9
using the analogWrite() function.
000000
This example code is in the public domain.
*/

int led = 9;           // the pin that the LED is attached to
int brightness = 0;    // how bright the LED is
int fadeAmount = 5;    // how many points to fade the LED by

// the setup routine runs once when you press reset:
void setup() {
  // declare pin 9 to be an output:
  pinMode(led, OUTPUT);
}

// the loop routine runs over and over again forever:
void loop() {
  // set the brightness of pin 9:
  analogWrite(led, brightness);

  // change the brightness for next time through the loop:
  brightness = brightness + fadeAmount;

  // reverse the direction of the fading at the ends of the fade:
  if (brightness == 0 || brightness == 255) {
    fadeAmount = -fadeAmount ;
  }
  // wait for 30 milliseconds to see the dimming effect
  delay(1);
}

```

Ilustración 96: Código Fade analizador lógico.

Una vez hecho el montaje y configurado el analizador, pulsamos sobre “Capture” y tenemos el siguiente resultado:

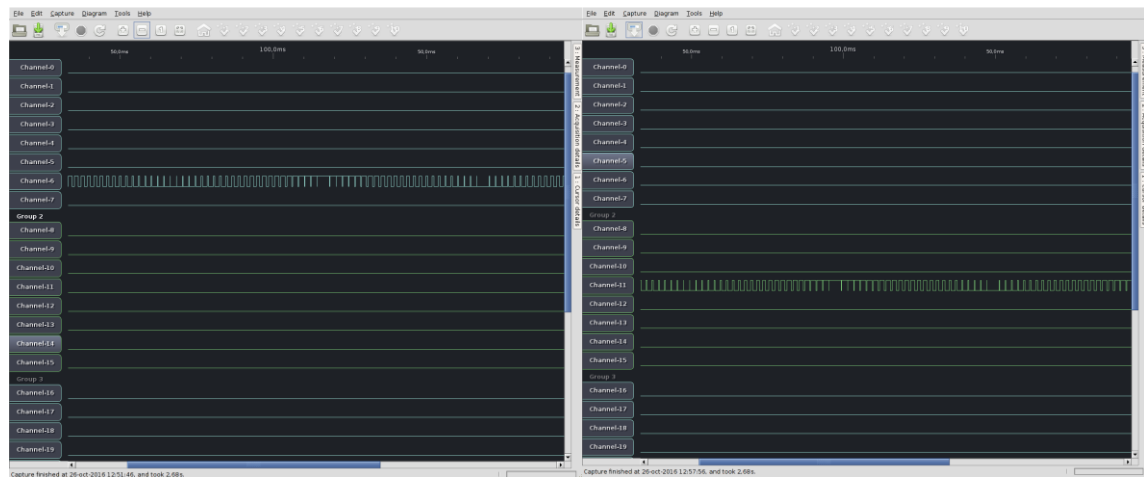


Ilustración 97: Ejecución del analizador lógico en varios canales

El montaje fue este:

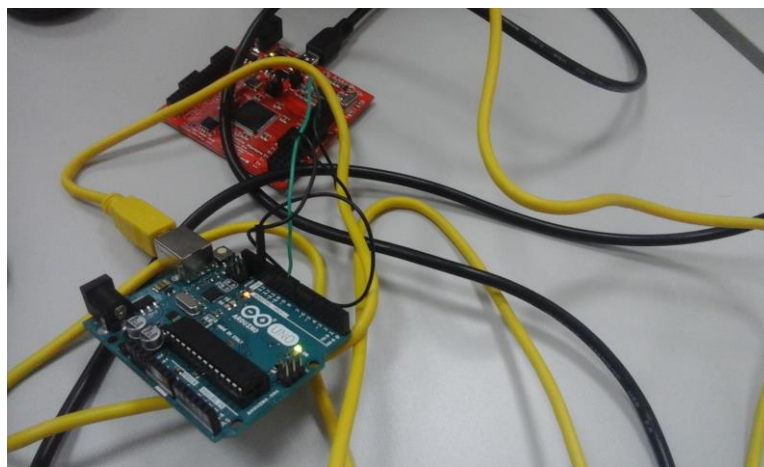


Ilustración 98: Montaje del analizador lógico

➤ Rediseñando el SoC. Añadiendo periféricos:

En esta última parte, desarrollamos un ejemplo básico de como añadir un periférico al SoC de ZPUino, sintetizarlo, generar el bit file, programarlo en la FPGA y utilizarlo desde un sketch.

El primer paso es crear el proyecto SoC ZPUino.

Para esto, dejamos Ctrl pulsado y nos dirigimos al mismo icono que crea una FPGA. El mensaje cambia y hacemos click sobre dicha imagen:



Ilustración 99: Creación del proyecto SoC en ZPUino

Entonces, se nos abre una ventana nueva y guardamos el proyecto. Acto seguido editamos el circuito pulsando sobre el icono correspondiente. Como hemos hecho anteriormente, navegamos hasta la pestaña “**Symbols**” tras pulsar doble click sobre “**Papilio_One_500K.sch**”. Comenzamos añadiendo una UART:

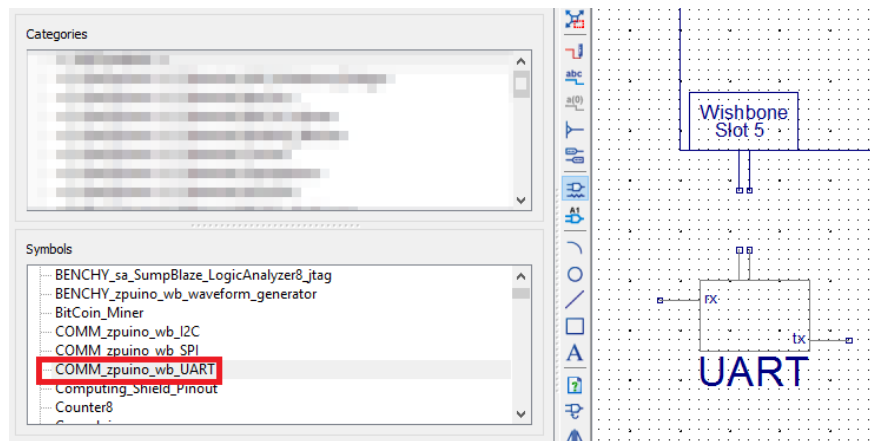


Ilustración 100: Añadiendo la UART

Continuamos con la supresión de dos pines. Es importante que también eliminemos los cables que unían a estos con el módulo:

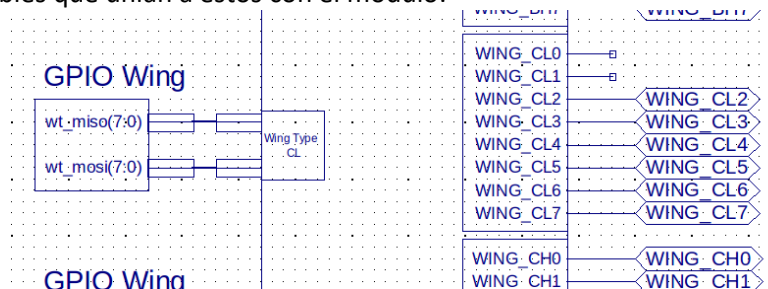


Ilustración 101: Supresión de pines para renombrar en la UART

Añadimos los marcadores a la UART que hemos creado y los renombramos con los nombres de los suprimidos. Además, conectamos la UART con el Wishbone, por ejemplo, número 5:

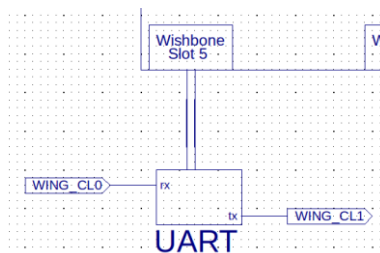


Ilustración 102: Conexión de la UART con el Wishbone

Como hemos hecho anteriormente, vamos a la pestaña “**Design**”, seleccionamos el “**Papilio_One_500K.sch**” y damos doble click sobre “**Generate Programming File**” para sintetizar el circuito. La ventana de consola muestra el mismo mensaje: “**Process “Generate Programming File” completed succesfully**”.

Volvemos al sketch del proyecto SoC ZPUino y hacemos estos cambios en el código:

```
HardwareSerial mySerial1(WishboneSlot(5));

int led = 13;

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

  pinMode(led, OUTPUT);
  mySerial1.begin(9600);
}

void loop() {
  mySerial1.write(1);
  digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
  delay(1000);             // wait for a second
  digitalWrite(led, LOW);  // turn the LED off by making the voltage LOW
  delay(1000);             // wait for a second
}
```

Ilustración 103: Código SoC ZPUino

Una vez hecho eso, y con nuestra placa conectada, pulsamos sobre “**Load Circuit**”. Una vez cargado el circuito, nos aseguramos de tener bien puesto el puerto de nuestra Papilio y cargamos el sketch.

Para completar el ejercicio, hacemos uso del anterior código de Python, respetando el puerto serie al cual se ha conectado la placa y tomamos también este código para ZPUino:

```
HardwareSerial mySerial1(WishboneSlot(5));

int led = 15;

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

  pinMode(led, OUTPUT);

  mySerial1.begin(9600);
}

void loop() {
  if(mySerial1.available()){
    char c =mySerial1.read();
    if(c == 'H'){
      digitalWrite(led,HIGH);
    }else if(c == 'L'){
      digitalWrite(led,LOW);
    }
  }
}
```

Ilustración 104: Código SoC ZPUino puerto serie

La ejecución del mismo enciende o apaga el LED conectado al pin 15 de nuestra Papilio. El montaje final queda así, teniendo en cuenta la conexión de la UART (www.ftdichip.com/Support/Documents/DataSheets/Cables/DS_TTL-232R_RPi.pdf):

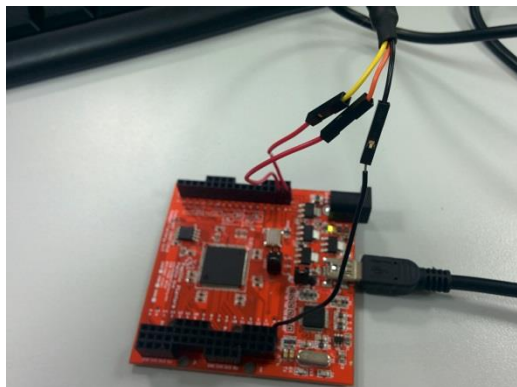


Ilustración 105: Montaje de la UART

La ejecución de este ejercicio fue completada conectando un LED en el pin indicado en cada apartado (no está disponible la documentación fotografiada o en vídeo).

Plataforma Raspberry Pi

INTRODUCCIÓN: PLATAFORMA Y ENTORNO DE TRABAJO

Raspberry Pi es lo que se conoce como **SBC** (Single Board Computer). Desarrollado en Reino Unido por la Fundación Raspberry Pi, su principal fin fue estimular la enseñanza de ciencias de la computación en las escuelas.

La plataforma dispone de contratos de distribución y venta con dos empresas, pero al mismo tiempo cualquiera puede convertirse en revendedor o redistribuidor, por lo que se entiende que es un producto con propiedad registrada pero de uso libre. Mantienen así, el control de la plataforma permitiendo su uso libre tanto a nivel educativo como particular. El asunto cambia si se pretende utilizar a nivel empresarial u obtener beneficios con su uso, ya que se debe consultar con la fundación.

En cuanto a versiones del hardware, Raspberry se caracteriza por tener dos modelos en venta: A y B. La principal diferencia entre estos es que el A no cuenta con puerto Ethernet, algo subsanable manualmente mediante un cable USB-Ethernet (aunque cerraría la única conexión USB con la que cuentan estas placas).

En el laboratorio empleamos la Raspberry Pi 3 Modelo B, lanzada al mercado en febrero de este mismo año, 2016. Como novedad con respecto a su predecesora, la RPi 2, cuenta con una CPU quad-core ARMv8 de 64 bits a 1.2 GHz, Wireless LAN 802.11n y Bluetooth 4.1, con tecnología BLE (Bluetooth Low Energy).

Desde el modelo B y desde el B+ de la RPi, cuentan con Ethernet y con 4 puertos USB. Además, poseen 1 GB de RAM, 40 pines GPIO, un puerto HDMI y ranura para Micro SD entre otras características. La alimentación de la misma es vía microUSB. Una característica importante de la RPi 3 es que es compatible con sus predecesoras.

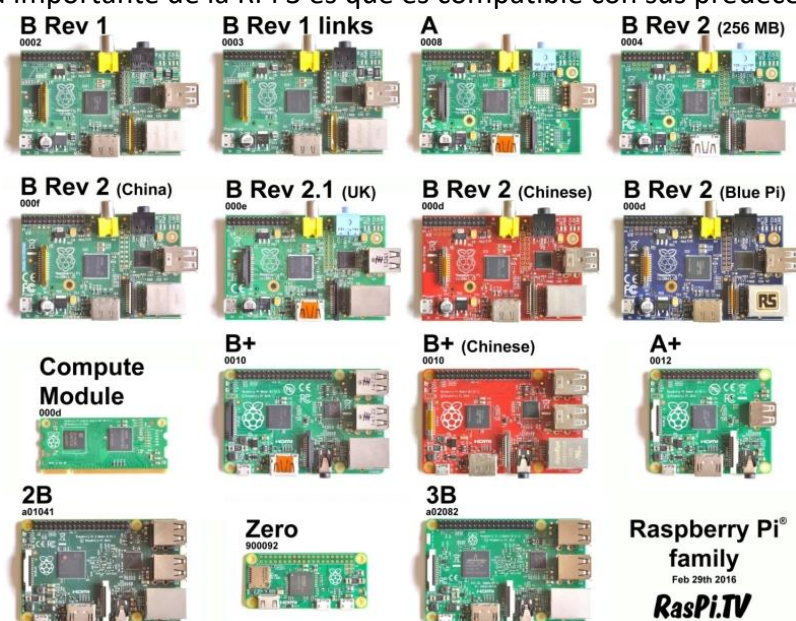


Ilustración 106: Modelos RPi

El software sí es open source, pues su sistema operativo oficial es una versión adaptada de **Debian**, denominada **Raspbian**, aunque permite otros sistemas operativos. En nuestro caso, utilizamos **Ubuntu MATE**. La imagen a descargar desde <https://10.1.15.78/~bellido> fue “**ubuntu-mate-16.04-desktop-armhf-raspberry-pi-resize.img**”. Durante la instalación del mismo tuve problemas con la μ SD, por lo que me hice cargo de poner la placa en marcha con el sistema instalándolo en casa.

En primer lugar, averiguamos donde tenemos montadas las particiones de la μ SD. Para ello, observamos las particiones que ya tenemos hechas con el comando “**df**”. Una vez hecho esto, introducimos la μ SD y lo ejecutamos de nuevo. Las nuevas particiones corresponden a la μ SD. Sabiendo ya cuales son, las desmontamos para sobrescribir en ellas con el comando “**umount /dev/nombreParticion**”. Es entonces cuando instalamos. Tras intentos fallidos como cambiar de RPi o μ SD, con ayuda del profesor, decidimos reformatear la μ SD inicial y reinstalar de nuevo:

```
practicass@mcr-81:~/Descargas$ sudo dd bs=4M if=ubuntu-mate-16.04-desktop-armhf-r
aspberry-pi-resize.img of=/dev/sdb
[sudo] password for practicas:
566+0 registros leídos
566+0 registros escritos
2373976064 bytes (2,4 GB, 2,2 GiB) copied, 108,346 s, 21,9 MB/s
765+0 registros leídos
765+0 registros escritos
3208642560 bytes (3,2 GB, 3,0 GiB) copied, 187,446 s, 17,1 MB/s
771+0 registros leídos
771+0 registros escritos
3233808384 bytes (3,2 GB, 3,0 GiB) copied, 189,854 s, 17,0 MB/s
1110+1 registros leídos
1110+1 registros escritos
4658233856 bytes (4,7 GB, 4,3 GiB) copied, 374,521 s, 12,4 MB/s
practicass@mcr-81:~/Descargas$
```

Ilustración 107: Escritura de Ubuntu MATE en μ SD

El porcentaje de finalización del proceso podemos verlo desde una nueva terminal con el comando: “**sudo pkill -USR1 -n -x dd**”. Sin embargo esto tampoco tuvo buen final.

Por tanto, me dispuse a realizar la práctica en casa. Una vez tenía la imagen descargada, introduje la tarjeta μ SD, con su adaptador a SD, en mi PC. Repasé la instalación hecha en clase y me di cuenta que, en una de las capturas, una de las particiones (las imágenes para RPi desde Linux suelen tener varias particiones, en nuestro caso eran dos) no estaba montada correctamente, pues no aparecía tras introducir la tarjeta:

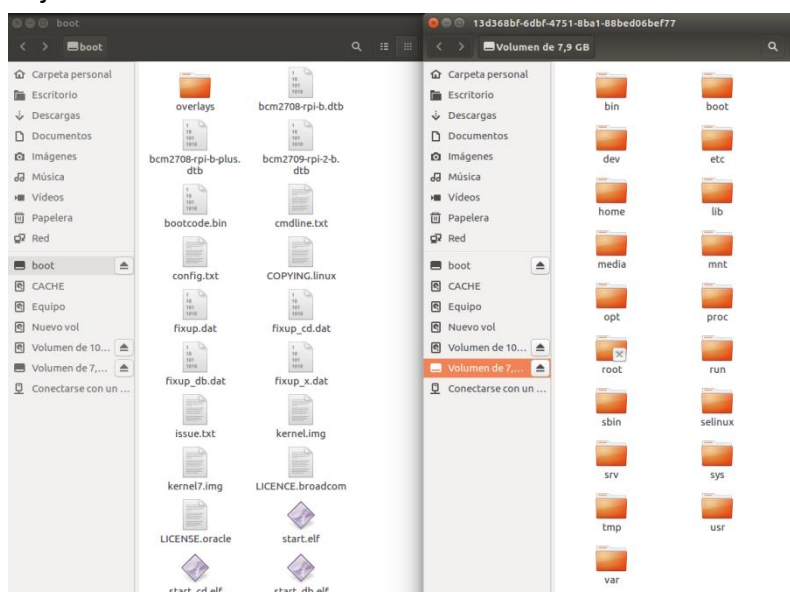


Ilustración 108: Partición montada de Ubuntu MATE

Así pues, acababa de abordar el posible problema del mal funcionamiento. Repetí el paso de ver el nombre que tenían mis particiones para intentar desmontarlas y reinstalar de nuevo:

```

ramon@ramon-K53SV:~$ df
S.ficheros bloques de 1K Usados Disponibles Uso% Montado en
udev 1956548 0 1956548 0% /dev
tmpfs 395328 6356 388972 2% /run
/dev/sda3 11973824 6565972 4776556 58% /
tmpfs 1976636 212 1976424 1% /dev/shm
tmpfs 5120 4 5116 1% /run/lock
tmpfs 1976636 0 1976636 0% /sys/fs/cgroup
tmpfs 395328 68 395260 1% /run/user/1000
ramon@ramon-K53SV:~$ df
S.ficheros bloques de 1K Usados Disponibles Uso% Montado en
udev 1956548 0 1956548 0% /dev
tmpfs 395328 6376 388952 2% /run
/dev/sda3 11973824 6565984 4776544 58% /
tmpfs 1976636 212 1976424 1% /dev/shm
tmpfs 5120 4 5116 1% /run/lock
tmpfs 1976636 0 1976636 0% /sys/fs/cgroup
tmpfs 395328 64 395264 1% /run/user/1000
/dev/mmcblk0p1 65480 20368 45112 32% /media/ramon/PI_ROOT
ramon@ramon-K53SV:~$ umount /dev/mmcblk0p1
ramon@ramon-K53SV:~$ df
S.ficheros bloques de 1K Usados Disponibles Uso% Montado en
udev 1956548 0 1956548 0% /dev
tmpfs 395328 6360 388968 2% /run
/dev/sda3 11973824 6566004 4776524 58% /
tmpfs 1976636 212 1976424 1% /dev/shm
tmpfs 5120 4 5116 1% /run/lock
tmpfs 1976636 0 1976636 0% /sys/fs/cgroup
tmpfs 395328 64 395264 1% /run/user/1000

```

Ilustración 109: Desmontando partición de Ubuntu MATE

La partición que había fallado era **"PI_ROOT"** (nombre conocido porque otros compañeros no tuvieron este tipo de fallo). Intenté desmontar la otra partición, que supuse se había creado con el mismo nombre, pero cambiando el dígito final ("**p1**" hace referencia a la partición 1): **"umount /dev/mmcblk0p2"**. Esto produjo una ventana de error diciendo que la partición que se pretendía desmontar no estaba disponible porque no había sido montada.

Introduje ese error en google y con el primer resultado llegué a la siguiente página: <http://www.dxsdata.com/2015/03/repair-corrupted-linux-partition-of-sd-card>.

Como indica el enlace, el sistema de ficheros estaba corrupto y era necesario repararlo. Entre las causas posibles que pueden desembocar a esto, se encuentran:

- Cesar la alimentación o resetear cuando el sistema está escribiendo. Esto podía ser evitado, así como la consecuencia de que, tras varias veces funcionando bien, el sistema de ficheros puede dañarse.
- Actualizar el sistema instalado puede que provoque que tengamos que lidiar con este problema, especialmente si tratamos con RPi.
- La tarjeta μ SD se ha dañado a causa del controlador interno o físicamente.

Probablemente, el conflicto haya sido que durante la descarga o escritura de la imagen se hayan perdido datos. También comentan en el mismo enlace que si la tarjeta es nueva, lo más probable es que solamente se haya dañado una de las particiones.

La solución fue ejecutar el comando:

```
ramon@ramon-K53SV:~$ sudo fsck -fy /dev/mmcblk0p2
[sudo] password for ramon:
fsck de util-linux 2.27.1
e2fsck 1.42.13 (17-May-2015)
Los «checksums» de uno o más descriptores de grupos de bloques son inválidos. ¿Arreglar? si

El «checksum» del descriptor de grupo 10 es 0xe6cf; debería ser 0xba65. ARREGLADO.
Paso 1: Verificando nodos-i, bloques y tamaños
Paso 2: Verificando la estructura de directorios
La entrada '.X11-unix' que está en /tmp (260103) tiene un nodo-i 84904 borrado/no utilizado. ¿Borrar? si
La entrada '.ICE-unix' que está en /tmp (260103) tiene un nodo-i 84905 borrado/no utilizado. ¿Borrar? si
La entrada '.XIM-unix' que está en /tmp (260103) tiene un nodo-i 84906 borrado/no utilizado. ¿Borrar? si
La entrada '.font-unix' que está en /tmp (260103) tiene un nodo-i 84907 borrado/no utilizado. ¿Borrar? si
La entrada '.Test-unix' que está en /tmp (260103) tiene un nodo-i 84908 borrado/no utilizado. ¿Borrar? si

Paso 3: Revisando la conectividad de directorios
Paso 4: Revisando las cuentas de referencia
La cuenta de referencia del nodo-i 260103 es 7, y debería ser 2. ¿Arreglar? si

Paso 5: Revisando el resumen de información de grupos
Diferencias del mapa de bits del bloque: -441699
¿Arreglar? si

La cuenta de bloques libres es incorrecta para el grupo #13 (0, contados=1).
¿Arreglar? si

La cuenta de bloques libres es incorrecta (165470, contados=165471).
¿Arreglar? si

La cuenta de nodos-i libres es incorrecta para el grupo #34 (0, contados=1).
¿Arreglar? si

La cuenta de nodos-i libres es incorrecta (96813, contados=96814).
¿Arreglar? si

PI_ROOT: ***** EL SISTEMA DE FICHEROS FUE MODIFICADO *****
PI_ROOT: 187666/284480 ficheros (0.1% no contiguos), 955153/1120624 bloques
ramon@ramon-K53SV:~$
```

Ilustración 110: Resolución error RPi

El comando se encargó de reestablecer el sistema de ficheros. Retiramos adecuadamente la tarjeta donde tenemos alojado el sistema de arranque Ubuntu MATE y desconectamos la RPi. En su lugar, conectamos a esta un teclado y un ratón vía USB, la pantalla de un PC (en mi caso el televisor) por HDMI, nuestra μ SD y el cable Ethernet. Finalmente, conectamos el μ USB para la alimentación y apareció la pantalla de configuración de Ubuntu MATE:

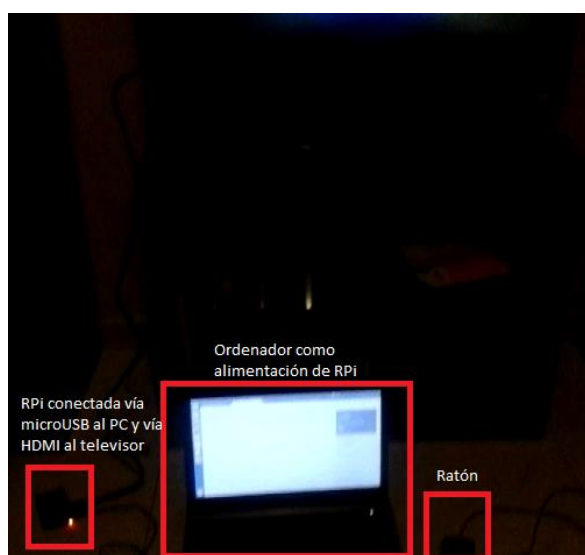


Ilustración 112: Conexión RPi casera

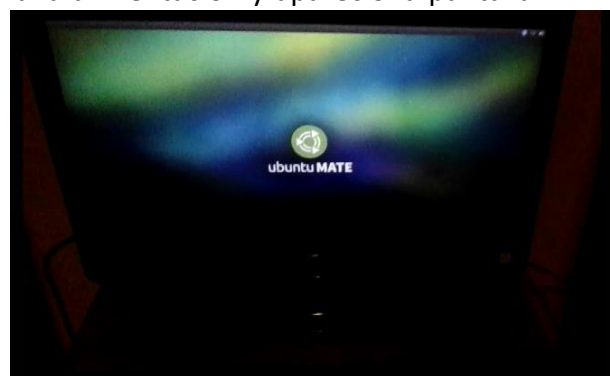


Ilustración 111: Inicialización de Ubuntu MATE

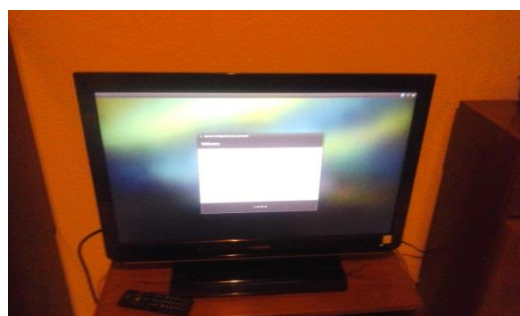


Ilustración 113: Instalación de Ubuntu MATE

De vuelta al laboratorio, configuré idioma, teclado, localización, fecha y hora. Es necesario crear un usuario nuevo, con una nueva password, datos necesarios para abordar las siguientes sesiones con esta plataforma (esto se guarda en la μ SD, tarjeta que cada alumno teníamos asignada).

Llevamos a cabo el “**resize**”, que permite al sistema de ficheros ocupar toda la μ SD. Así que, en la ventana de Welcome (Bienvenida), hacemos click sobre el icono de RPi:



Ilustración 114: Resize de RPi

Lo cual nos lleva a la siguiente pantalla, donde le damos al botón que, de esta forma tan sencilla, redimensiona el sistema de ficheros:

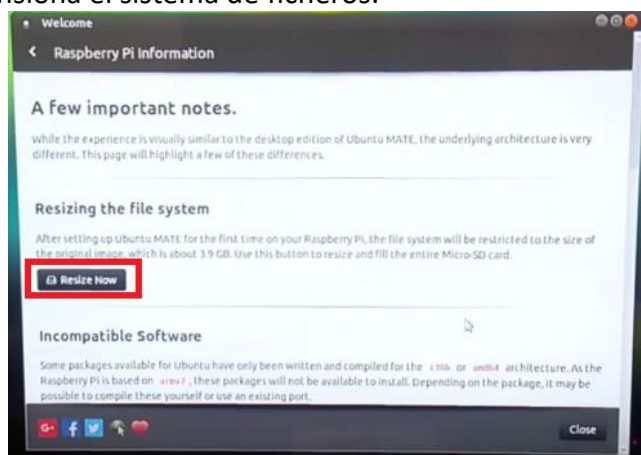


Ilustración 115: Botón de resize

El último paso es configurar Internet agregando la conexión manual cableada con la IP asignada al PC del laboratorio del que desenchufamos el cable Ethernet, con máscara 22, puerta de enlace 10.1.15.78 y servidores DNS 8.8.8.8. Comprobamos que tenemos conexión a Internet con un ping a Google.

Desgraciadamente, debido al cambio de RPi en cada sesión, fue necesario reconfigurar hora, fecha y acceso a Internet vía directa, es decir, sin usar el protocolo **SSH**. Sin embargo, una vez subsanados estos desajustes, pudimos realizar a cabo todas las sesiones desde el terminal mediante SSH.

Finalmente, y como comparativa con las otras dos plataformas que hemos usado, decir que RPi cuenta con módulos para cámara, por ejemplo. Y, aunque tiene productos como el [Sense-Hat](#), lo interesante es que pueden utilizar shields de Arduino con el correcto adaptador ([Bridge](#)).

DESARROLLO PRÁCTICO

Hemos cubierto una serie de seis tareas, desde el manejo de los GPIO (General Purpose Input/Output) hasta una cierta complejidad con el manejo web y el puerto Ethernet.

➤ **Manejo de GPIOs desde línea de comandos:**

La creación y accesibilidad a los pines de entrada y salida es algo que debemos llevar a cabo “manualmente” desde una terminal.

Aquí vemos la disposición de los pines de la RPi 3 B:

Raspberry Pi 3 GPIO Header

| Pin# | NAME | | NAME | Pin# |
|------|------------------------------------|--|------------------------------------|------|
| 01 | 3.3v DC Power | | DC Power 5v | 02 |
| 03 | GPIO02 (SDA1 , I ² C) | | DC Power 5v | 04 |
| 05 | GPIO03 (SCL1 , I ² C) | | Ground | 06 |
| 07 | GPIO04 (GPIO_GCLK) | | (TXD0) GPIO14 | 08 |
| 09 | Ground | | (RXD0) GPIO15 | 10 |
| 11 | GPIO17 (GPIO_GEN0) | | (GPIO_GEN1) GPIO18 | 12 |
| 13 | GPIO27 (GPIO_GEN2) | | Ground | 14 |
| 15 | GPIO22 (GPIO_GEN3) | | (GPIO_GEN4) GPIO23 | 16 |
| 17 | 3.3v DC Power | | (GPIO_GEN5) GPIO24 | 18 |
| 19 | GPIO10 (SPI_MOSI) | | Ground | 20 |
| 21 | GPIO09 (SPI_MISO) | | (GPIO_GEN6) GPIO25 | 22 |
| 23 | GPIO11 (SPI_CLK) | | (SPI_CE0_N) GPIO08 | 24 |
| 25 | Ground | | (SPI_CE1_N) GPIO07 | 26 |
| 27 | ID_SD (I ² C ID EEPROM) | | (I ² C ID EEPROM) ID_SC | 28 |
| 29 | GPIO05 | | Ground | 30 |
| 31 | GPIO06 | | GPIO12 | 32 |
| 33 | GPIO13 | | Ground | 34 |
| 35 | GPIO19 | | GPIO16 | 36 |
| 37 | GPIO26 | | GPIO20 | 38 |
| 39 | Ground | | GPIO21 | 40 |

Ilustración 116: GPIO RPi 3 B

En primer lugar hacemos el GPIO17 accesible mediante este comando: **“echo 17 > /sys/class/gpio/export”**. El siguiente paso es definir esta estructura que corresponde al GPIO17 físico como salida, pues vamos a conectar un LED: **“echo out > /sys/class/gpio/gpio17/direction”**.

Ya tenemos definido el GPIO17, ahora manipulamos su valor. Para encender nuestro LED: **“echo 1 > /sys/class/gpio/gpio17/value”**; y para apagarlo: **“echo 0 > /sys/class/gpio/gpio17/value”**.

Podemos eliminar la configuración de este GPIO con **“echo 17 > /sys/class/gpio/unexport”**

El montaje del circuito es conectar un LED, con su respectiva resistencia, al GPIO 17. La ejecución de este código es la siguiente:

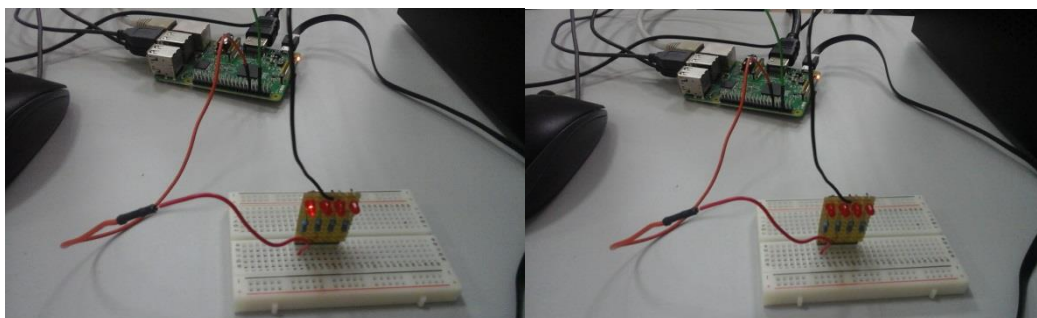


Ilustración 117: Ejecución manejo de GPIO LED ON/OFF

➤ **Ejemplos sencillos con PYTHON:**

En este segundo ejercicio, hacemos parpadear un par de LEDs. Eso sí, a diferencia del primero, en este usamos Python, evitando el uso de los comandos en la terminal de la parte anterior.

En nuestro caso, el sistema que hemos instalado ya tiene instalada la librería GPIO, aunque los pasos a seguir, si no la tuviésemos son sencillos. En primer lugar descargamos la librería: **"wget 'http://downloads.sourceforge.net/project/raspberrypi-gpio-python/RPi.GPIO-X.X.X.tar.gz'"** (donde X.X.X es la versión de la librería); acto seguido, descomprimos el archivo: **"tar zxvf RPi.GPIO-X.X.X.tar.gz"** y entramos en el directorio que se nos acaba de crear: **"cd RPi.GPIO-X.X.X"**. Tras instalar el paquete python-dev: **"sudo apt-get install python-dev"**, finalmente instalamos la librería: **"sudo python setup.py install"**.

Sobre la terminal, escribimos **"sudo nano blink.py"**, y dentro de este, el código en Python:

```
import RPi.GPIO as GPIO
import time
GPIO.setmode(GPIO.BCM)
GPIO.setup(17, GPIO.OUT) ## GPIO 17 como salida
GPIO.setup(27, GPIO.OUT) ## GPIO 27 como salida

def blink():
    print "Ejecucion iniciada..."
    iteracion = 0
    while iteracion < 30: ## Segundos que durara la funcion
        GPIO.output(17, True) ## Enciendo el 17
        GPIO.output(27, False) ## Apago el 27
        time.sleep(1) ## Esperamos 1 segundo
        GPIO.output(17, False) ## Apago el 17
        GPIO.output(27, True) ## Enciendo el 27
        time.sleep(1) ## Esperamos 1 segundo
        iteracion = iteracion + 2 ## Sumo 2 porque he hecho dos parpadeos
    print "Ejecucion finalizada"
    GPIO.cleanup() ## Hago una limpieza de los GPIO
blink() ## Hago la llamada a la funcion blink
```

Ilustración 118: Código control LEDs Python RPi

Ejecutamos en la consola con **"sudo python blink.py"**. El resultado lo podemos ver en el anexo audiovisual.

➤ **Conexión por SSH:**

En esta tarea nos conectamos a la RPi de forma remota. Para ello, usamos el protocolo SSH. SSH (Secure SHell), o intérprete de órdenes seguro, ya que la información va cifrada (a diferencia de telnet), es un protocolo sobre TCP que permite el acceso a máquinas remotas vía red, es decir, podemos manejar la computadora mediante comandos de consola y también ejecutar programas gráficos como Arduino.

Gracias a este protocolo podremos montar un servidor web que habilite el control de los GPIOs para el encendido y apagado de LEDs y un relé y la lectura de la temperatura del sensor tmp36GZ.

El comando empleado es: **"ssh -X ramchagar@10.1.15.82"**. Como se puede apreciar, el comando tiene mi usuario (ramchagar) y la IP del ordenador del que retiramos el cable Ethernet y que, por tanto, ocupaba la RPi. La ejecución del comando nos pide la contraseña para acceder remotamente, que no es otra que la que usamos al configurar Ubuntu MATE:


```

practicass@MCR-81:~$ ssh -X ramchagar@10.1.15.82
The authenticity of host '10.1.15.82 (10.1.15.82)' can't be established.
ECDSA key fingerprint is SHA256:nZz0LOUGXJpk3obv4m+MgBfQ0yLCr2a0dXVVeF+KYzA.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.1.15.82' (ECDSA) to the list of known hosts.
ramchagar@10.1.15.82's password:
Welcome to Ubuntu 16.04 LTS (GNU/Linux 4.1.19-v7+ armv7l)

 * Documentation:  https://help.ubuntu.com/

Pueden actualizarse 365 paquetes.
101 actualizaciones son de seguridad.

Last login: Wed Nov  2 10:33:09 2016
ramchagar@ramchagar-desktop:~$

```

Ilustración 119: Conexión a RPi vía SSH

➤ Servidor Web (LEDs):

El objetivo de este apartado fue hacer uso de la anterior tarea para montar el servidor web. La RPi tiene la capacidad de combinar framework web moderno con hardware y electrónica. Esta placa, además de conseguir datos de los servidores vía Internet, puede actuar como servidor propio. Existen herramientas como Apache o lighttpd que proporcionan archivos, ya sean HTML, imágenes, audio, vídeo o ejecutables, pero también hay herramientas que permiten crear servidores web extendiendo lenguajes de programación como Python, Ruby y JavaScript que generan HTML dinámico al recibir peticiones HTTP. Este modo permite tratar con eventos físicos, almacenamiento de datos o comprobar el valor de un sensor vía navegador.

En nuestro caso hemos empleado **Flask**, que es un framework web para Python que convierte la RPi en servidor web dinámico.

Como primer paso, y tras conectarnos vía SSH a nuestra RPi, es necesario instalar **pip**, un sistema para la gestión, instalación y administración de paquetes de software escritos en Python: “**sudo apt-get install python-pip**”. En consonancia con la orden anterior, procedemos a instalar Flask: “**sudo pip install flask**”.

Realizamos un pequeño programa con el que probar la instalación por medio de este código:

```

from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80, debug=True)

```

Ilustración 120: Código Python para probar el Flask

En las dos primeras líneas cargamos el módulo Flask en nuestro script de Python y creamos un objeto de tipo Flask de nombre ‘**app**’. Justo debajo, declaramos una ruta que ejecuta el código definido dentro de esta cada vez que desde el servidor web se accede a la misma. El código en cuestión es un simple mensaje de “**Hello World!**”. Las dos líneas finales mantienen al servidor “en escucha”, es decir, abre el servidor por el puerto 80 (HTTP) si ejecutamos este código directamente desde la línea de comandos.

La ejecución del mismo devuelve un código 200 porque se accedió a la ruta raíz ("/") con éxito. Sin embargo, muestra un 404 al acceder al "favicon.ico", ya que nuestro servidor no tiene:

```
ramchagar@ramchagar-desktop:~/Escritorio$ sudo python hello-flask.py
* Running on http://0.0.0.0:80/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger pin code: 268-753-387
10.1.15.81 - - [08/Nov/2016 09:24:02] "GET / HTTP/1.1" 200 -
10.1.15.81 - - [08/Nov/2016 09:24:02] "GET /favicon.ico HTTP/1.1" 404 -
```

Ilustración 121: Ejecución hello-flask.py en terminal

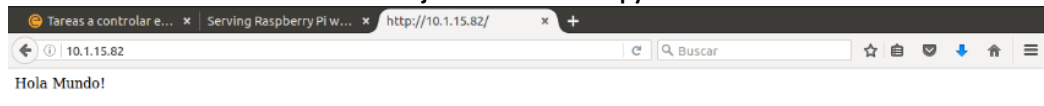


Ilustración 122: Acceso a la ruta raíz mostrando información con la modificación del idioma.

Flask cuenta con un “motor de plantillas” (**template engine**) llamado **Jinja2** que permite usar archivos HTML independientes con marcadores para insertar datos dinámicos donde queramos.

Una plantilla Jinja, inspirada en Python y Django, es un archivo de texto bajo cualquier formato que contiene variables y/o expresiones que son reemplazadas cuando son interpretadas y etiquetas que controlan la lógica de la misma.

Nosotros las hemos usado para algunos tipos de delimitadores que hacían más cómodo el hecho de mostrar datos como ‘{% ... %}’ para estructuras o ‘{{ ... }}’ para expresiones.

Así pues, hechas las presentaciones de última hora, bajo una carpeta creamos “**hello-template.py**” con el siguiente código:

```
from flask import Flask, render_template
import datetime
app = Flask(__name__)

@app.route("/")
def hello():
    now = datetime.datetime.now()
    timeString = now.strftime("%Y-%m-%d %H:%M")
    templateData = {
        'title': 'HELLO!',
        'time': timeString
    }
    return render_template('main.html', **templateData)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80, debug=True)
```

Ilustración 123: Código Python hello-template.py

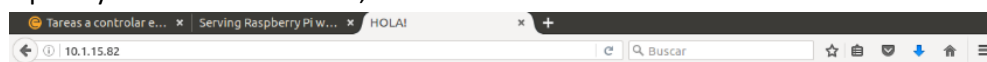
Líneas nuevas e interesantes a destacar pueden ser la obtención de la hora actual y el formato de la misma con “**now = datetime.datetime.now()**” y “**timeString = now.strftime(“%Y-%m-%d %H:%M”)**”. Aunque las que realmente importan son las cinco siguientes en las que se crea el ‘diccionario’ de variables ‘**templateData**’ donde se asocian claves a unos valores que se pasan bajo la llamada a “**render_template(..., ...)**” hacia “**main.html**”.

En la misma carpeta creamos un nuevo directorio llamado ‘**templates**’. Dentro de este último, creamos el archivo “**main.html**” con el código:

```
<!DOCTYPE html>
<html>
  <head>
    <title>{{ title }}</title>
  </head>
  <body>
    <h1>Hello, World!</h1>
    <h2>The date and time on the server is: {{ time }}</h2>
  </body>
</html>
```

Ilustración 124: Código main.html del ejemplo Python hello-template.py

Ejecutamos “**hello-template.py**” vía SSH y accedemos a la IP de nuestra RPi desde el navegador del ordenador contiguo al que pertenece el cable Ethernet conectado a Raspberry. Como consecuencia, obtenemos este resultado:



Hola mundo!

La fecha y la hora del servidor es: 2016-11-08 10:26

Ilustración 125: Acceso a la ruta raíz mostrando información con la modificación del idioma (fecha y hora)

Al adquirir cierto manejo con Python y Flask, incorporamos varios elementos a nuestro servidor web. La estructura fue sencilla: en nuestra carpeta del servidor web creamos el archivo “**weblamp.py**” y el directorio ‘**templates**’. Bajo este último, creamos el “**main.html**”.

En cuanto a los elementos, el primero de ellos consiste en un par de LEDs sobre los que controlamos su estado (ON/OFF). El montaje de este primer sub-apartado es sencillo, pues basta con conectar un par de LEDs a los pines 24 y 25.

Sobre el código, las modificaciones se basan en torno a la línea que importa la GPIO. Lo primero es poner el modo de este módulo, en nuestro caso “GPIO.BCM”. La diferencia con respecto al otro, “GPIO.BOARD”, radica en la forma de llamar a los pines:

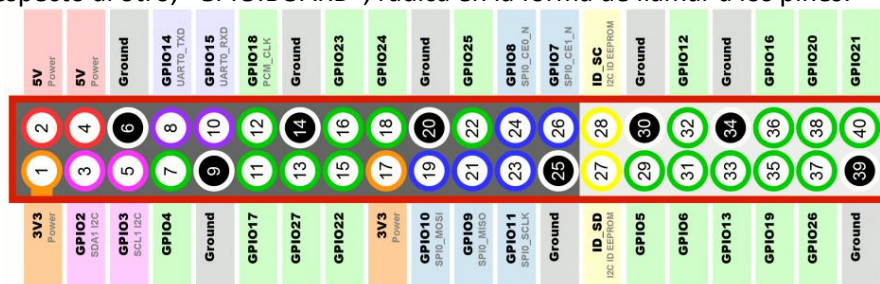


Ilustración 126: BOARD Vs. BCM

‘**BOARD**’ hace referencia a los pines por el número que ocupan en el diseño de la placa impresa, mientras que ‘**BCM**’ (Broadcom SOC channel) se refiere al nombre que aparece al lado de cada pin en la imagen anterior.

Inicializado el módulo GPIO, creamos un diccionario para los pines con estado cero lógico por defecto. Configuramos los pines como salida.

Una vez tenemos las estructuras inicializadas, definimos una función ‘**main**’ bajo la ruta raíz que aloja en ‘**templateData**’ los objetos ‘**pins**’ y los pasa al ‘**main**’. Además, definimos una nueva función bajo la ruta “/**<changePin>/<action>**” que se ejecuta cada vez que pulsamos en el “**main.html**” sobre el enlace que nos lleva a ella. Esta función interpreta el pin que hemos pulsado: cambia al estado que queremos poner el pin y lanza un mensaje de información. Esta interpretación la envía al “**main.html**” haciendo uso de la función mencionada antes “**render_template (...)**”

Hay una función que no hemos usado durante las prácticas, que es el ‘**Toggle**’. Esta función podría ser interesante para manejar switches en un pasillo largo de casa o en las escaleras, ya que podríamos usarlos como un conmutador, pues ‘**Toggle**’ cambia del estado actual al contrario.

```

import RPi.GPIO as GPIO
from flask import Flask, render_template, request
app = Flask(__name__)

GPIO.setmode(GPIO.BCM)

# Create a dictionary called pins to store the pin number, name, and pin state:
pins = {
    24 : {'name' : 'LED1', 'state' : GPIO.LOW},
    25 : {'name' : 'LED2', 'state' : GPIO.LOW}
}

# Set each pin as an output and make it low:
for pin in pins:
    GPIO.setup(pin, GPIO.OUT)
    GPIO.output(pin, GPIO.LOW)

@app.route("/")
def main():
    # For each pin, read the pin state and store it in the pins dictionary:
    for pin in pins:
        pins[pin]['state'] = GPIO.input(pin)
    # Put the pin dictionary into the template data dictionary:
    templateData = {
        'pins' : pins
    }
    # Pass the template data into the template main.html and return it to the user
    return render_template('main.html', **templateData)

# The function below is executed when someone requests a URL with the pin number and action in it:
@app.route("/<changePin>/<action>")
def action(changePin, action):
    # Convert the pin from the URL into an integer:
    changePin = int(changePin)
    # Get the device name for the pin being changed:
    deviceName = pins[changePin]['name']
    # If the action part of the URL is "on," execute the code indented below:
    if action == "on":
        # Set the pin high:
        GPIO.output(changePin, GPIO.HIGH)
        # Save the status message to be passed into the template:
        message = "Cambiado " + deviceName + " a encendido."
    if action == "off":
        GPIO.output(changePin, GPIO.LOW)
        message = "Cambiado " + deviceName + " a apagado."
    if action == "toggle":
        # Read the pin and set it to whatever it isn't (that is, toggle it):
        GPIO.output(changePin, not GPIO.input(changePin))
        message = "Toggled " + deviceName + " ."

    # For each pin, read the pin state and store it in the pins dictionary:
    for pin in pins:
        pins[pin]['state'] = GPIO.input(pin)

    # Along with the pin dictionary, put the message into the template data dictionary:
    templateData = {
        'message' : message,
        'pins' : pins
    }

    return render_template('main.html', **templateData)

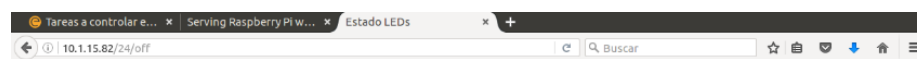
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80, debug=True)

```

Ilustración 127: Código Python servidor web (LEDs)

Editamos ahora el “main.html” de forma que muestre los mensajes de información y el estado de los LEDs por medio de las estructuras Jinja2. Es interesante apreciar que podemos acceder a una parte del objeto como el nombre del pin o el estado del mismo. El “main.html”, al compartirse y, prácticamente no cambiar en estructura HTML a lo largo de los cambios con el servidor web, se muestra al final de la memoria.

Ejecutamos en la RPi, de forma remota, el archivo con "sudo python weblamp.py" desde la carpeta que lo contiene en nuestra RPi. El resultado lo podemos ver aquí debajo:



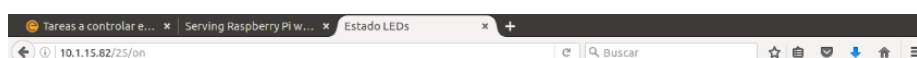
Dispositivos listados y su estado

El LED1 está actualmente apagado ([Encender](#))

El LED2 está actualmente encendido ([Apagar](#))

Cambiado LED1 a apagado.

Ilustración 128: Ejecución sobre navegador de servidor web LEDs (I)



Dispositivos listados y su estado

El LED1 está actualmente encendido ([Apagar](#))

El LED2 está actualmente encendido ([Apagar](#))

Cambiado LED2 a encendido.

Ilustración 129: Ejecución sobre navegador de servidor web LEDs (II)

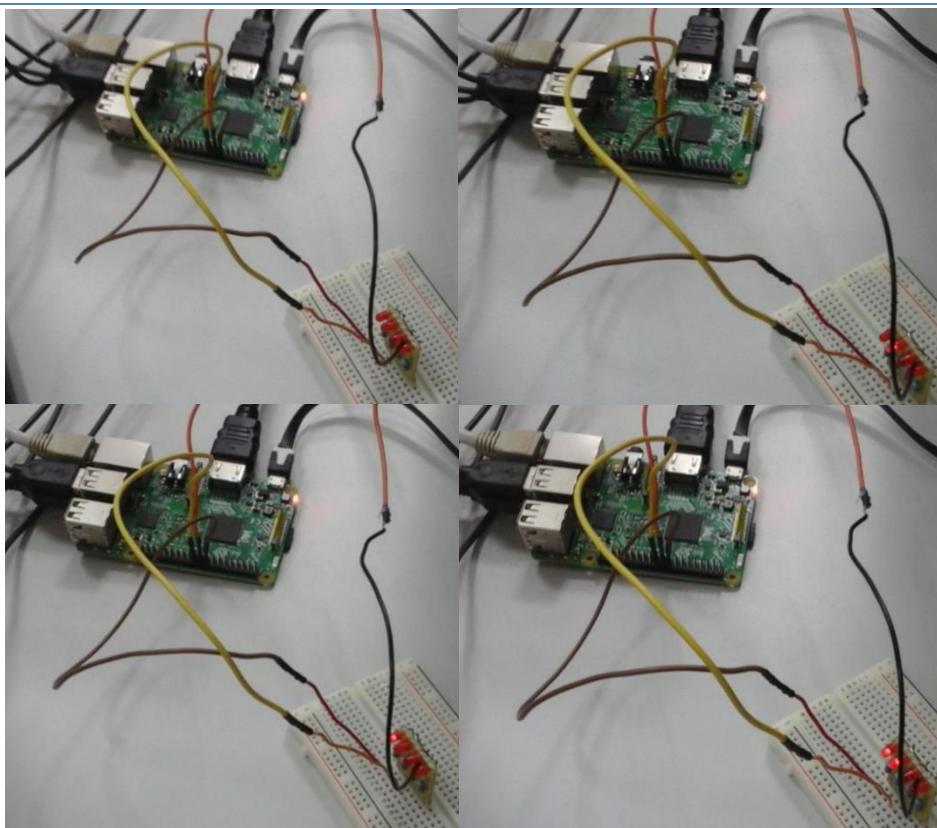


Ilustración 130: Ejecución y montaje servidor web LEDs

➤ **Servidor Web (Temperatura):**

Tanto esta tarea, como la posterior, se realizaron conectando los componentes a Arduino y, desde este, mediante el puerto serie, nuestra RPi recibía los datos que plasmaba a la página web.

El segundo sub-apartado se corresponde con la tarea 5, añadir un sensor de temperatura. Las modificaciones en código pasaban por crear un programa para Arduino que leyese el sensor y enviase, únicamente el dato leído, por puerto serie.

Entonces, el script en Python se encargaba de leerlo y mostrarlo en el archivo HTML. Para ello, inicialmente, importamos la biblioteca serial ("**import serial**"). Después, encima del '**for**' que pone los pines como salida, escribimos ("**ser = serial.Serial('/dev/ttyACM1', 9600)**", inicializando el puerto serial creado al conectar Arduino a la RPi). Justo bajo del '**def main():**' escribimos "**temps = ser.readline()**", para que al cargar la ruta raíz haga una lectura inicial de la temperatura. Este valor, se lo pasamos dentro de la estructura '**templateData**' con el par clave-valor "**'temps' : temps**" (separamos un par de otro mediante el uso de la coma ',').

El último cambio en Python fue crear una función que permitiese, cada vez que accediéramos a la ruta "**/temperature**", la actualización de la temperatura.

Los códigos, tanto Python como HTML, son una modificación de los anteriores, y se muestran también al final de la memoria.

La ejecución se muestra en el anexo audiovisual.

➤ **Servidor Web (Relé):**

Hemos llegado al final de estas prácticas de placas de desarrollo. La última tarea es conseguir hacer conmutar un relé que conectamos a Arduino, y este a su vez se conecta vía puerto serie a RPi.

Este es el tercer sub-apartado y las modificaciones en el fichero Python, con respecto a la anterior tarea, pasaron por definir el relé como el pin 17 de la RPi. Es cierto que no lo conectamos a la RPi, pero la definición como tal es para aprovechar el código desarrollado en el “**main.html**” y en la definición de ‘**action**’. Es decir, dentro de dicha definición, usamos los ‘**if**’ para ON u OFF; es dentro de estos ‘**if**’ donde añadimos nuevas estructuras de este tipo, ya que si el pin de la ruta corresponde al 17, entonces enviaremos por el serial el carácter que conmuta el estado del relé (quizás hubiese sido más realista usar la función ‘**toggle**’ para el relé, ya que este conmuta. De hecho, en el Python la ‘**L**’ la escribimos en el serial cuando se acciona el ON y la ‘**H**’ cuando es el OFF para que veamos la luz encenderse en el relé. Para que surtiese efecto esa modificación, en el “**main.html**” deberíamos diferenciar el pin del resto para cambiar la ruta si es el del relé el accionado).

Y ese carácter pasa por el serial hasta Arduino, que lo recoge en el código y en función de lo que reciba, escribe una u otra posición en el relé.

El HTML recibe de Python los estados de los pines (el relé pese a no estar conectado a la RPi tiene que cambiar su estado para que el HTML lo lea como cambiado).

Bajo estas líneas dejo el código en Arduino, Python y HTML (con pequeños detalles de CSS, el cual no pude integrar como una hoja de estilos aparte), respectivamente:

```
int pinRele = 8;
int sensorPin = 0;

void setup(){
  pinMode(pinRele, OUTPUT);
  Serial.begin(9600);
}

void loop(){
  float sensorVal;
  float temperatura;
  if(Serial.available() > 0){
    char c = Serial.read();
    if(c == 'H'){
      digitalWrite(pinRele, HIGH);
    }else if(c == 'L'){
      digitalWrite(pinRele, LOW);
    }
  }
  sensorVal = analogRead(sensorPin);
  temperatura = ((sensorVal/1024)*5 - .5)*100;
  Serial.println(temperatura);
  delay(2000);
}
```

Ilustración 131: Código Arduino servidor web (temperatura y relé)


```

import RPi.GPIO as GPIO
import serial
from flask import Flask, render_template, request
app = Flask(__name__)

GPIO.setmode(GPIO.BCM)

# Create a dictionary called pins to store the pin number, name, and pin state:
pins = {
    17 : {'name' : 'Relay', 'state' : GPIO.LOW},
    24 : {'name' : 'LED1', 'state' : GPIO.LOW},
    25 : {'name' : 'LED2', 'state' : GPIO.LOW}
}

ser = serial.Serial('/dev/ttyACM1', 9600)

# Set each pin as an output and make it low:
for pin in pins:
    GPIO.setup(pin, GPIO.OUT)
    GPIO.output(pin, GPIO.LOW)

@app.route("/")
def main():
    temps = ser.readline()
    # For each pin, read the pin state and store it in the pins dictionary:
    for pin in pins:
        pins[pin]['state'] = GPIO.input(pin)
    # Put the pin dictionary into the template data dictionary:
    templateData = {
        'pins' : pins,
        'temps' : temps
    }
    # Pass the template data into the template main.html and return it to the user
    return render_template('main.html', **templateData)

@app.route("/temperature")
def temp():
    temps = ser.readline()
    for pin in pins:
        pins[pin]['state'] = GPIO.input(pin)
    message = "La temperatura se ha refrescado"
    templateData = {
        'pins' : pins,
        'temps' : temps,
        'message' : message
    }
    return render_template('main.html', **templateData)

# The function below is executed when someone requests a URL with the pin number and action in it:
@app.route("/<changePin>/<action>")
def action(changePin, action):
    # Convert the pin from the URL into an integer:
    changePin = int(changePin)
    # Get the device name for the pin being changed:
    deviceName = pins[changePin]['name']
    # If the action part of the URL is "on," execute the code indented below:
    if action == "on":
        if changePin == 17:
            ser.write('L')
            # Set the pin high:
            GPIO.output(changePin, GPIO.HIGH)
            # Save the status message to be passed into the template:
            message = "Cambiado " + deviceName + " a encendido."
        if action == "off":
            if changePin == 17:
                ser.write('H')
                # Set the pin high:
                GPIO.output(changePin, GPIO.LOW)
                message = "Cambiado " + deviceName + " a apagado."
        if action == "toggle":
            # Read the pin and set it to whatever it isn't (that is, toggle it):
            GPIO.output(changePin, not GPIO.input(changePin))
            message = "Toggled " + deviceName + "."

    # For each pin, read the pin state and store it in the pins dictionary:
    for pin in pins:
        pins[pin]['state'] = GPIO.input(pin)

    # Along with the pin dictionary, put the message into the template data dictionary:
    templateData = {
        'message' : message,
        'pins' : pins
    }

    return render_template('main.html', **templateData)

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80, debug=True)

```

Ilustración 132: Código Python servidor web (completo)


```

<!DOCTYPE html>
<html>
<head>
  <title>Estado componentes RPi final</title>
</head>

<body style="margin:0; padding:0; top:0;">
<div id="titulo" style="background-color: green; width:100%; display: inline-block;">
  <h1 style="color: orange; text-align: center;">Dispositivos listados y su estado</h1>
</div>

  {% for pin in pins %}
  <p style="color: black; text-align: left;">El {{ pins[pin].name }}
  {% if pins[pin].state == true %}
    está actualmente encendido (<a style="text-decoration: none;" href="/{{pin}}/off">Apagar</a>)
  {% else %}
    está actualmente apagado (<a style="text-decoration: none;" href="/{{pin}}/on">Encender</a>)
  {% endif %}
  </p>
  {% endfor %}
  <p style="color: black; text-align: left;">La temperatura es: {{ temps }}°C (<a style="text-decoration: none;" href="/temperature">Refrescar</a>)
  {% if message %}
  <h2 style="color: blue; text-align: center;">{{ message }}</h2>
  {% endif %}

</body>
</html>

```

Ilustración 133: Código HTML (y CSS) servidor web (completo)

La ejecución y montaje quedan reflejados en el anexo audiovisual.

Conclusiones

Hemos conocido tres plataformas muy interesantes. Dos de ellas han sonado mucho estos últimos años (Arduino y RPi), la otra, ZPUino, nos abre un nuevo mundo de posibilidades con las FPGAs y los SoCs. También hemos tenido un primer contacto con el código desarrollado en Python y la interacción del mismo con los sensores y actuadores. Hemos dado un repaso y tenido la primera toma de contacto con componentes hardware como el motor DC o el relé.

Hemos percibido que lo simple que pueden aparentar ser algunos códigos o conexiones, puede tornarse en algo más complejo de lo que podían parecer a simple vista, ya que un error de programación o concepto, también por desconocimiento, nos ha dado en alguna ocasión, durante el desarrollo de las sesiones, algún pequeño quebradero de cabeza. Aunque también es cierto que, tras una breve meditación, la solución podía ejecutarse de manera sencilla.

Del desarrollo de estas prácticas me quedo con todos los conocimientos adquiridos como el uso de placas de expansión, el diseño de circuitos con DesignLab o el control de un sistema web domótico de forma remota. Me han parecido unas sesiones muy interesantes e instructivas.

Anexo Audiovisual

El anexo audiovisual pretende cubrir la ejecución de partes de la práctica para lograr plasmar mejor el resultado final.

- Arduino:.....https://youtu.be/_UMimCWhanc
- ZPUino:.....<https://youtu.be/20-F1NKAUEk>
- Raspberry Pi:.....<https://youtu.be/m2ypg7OZnDI>