



decsai.ugr.es

Universidad de Granada

Teoría de la Información y la Codificación

Grado en Ingeniería Informática

Seminario 1.- Introducción a Arduino. Diseño y construcción de plataforma para transmisión de datos por láser.



**Departamento de Ciencias de la
Computación e Inteligencia Artificial**

- 1. ¿Qué es Arduino?**
- 2. El procesador**
- 3. Compilador y bibliotecas para procesadores AVR**
- 4. Mi primer programa para AVR AtMega328p**
- 5. Comunicaciones en serie**
- 6. Mi primer programa para comunicaciones en serie: ECHO**
- 7. Recepción de datos desde sensores fotorresistores digitales**
- 8. Envío de datos por dispositivos emisores láser**
- 9. La arquitectura del sistema de emisión/recepción de datos por láser**
- 10. Ampliación de la biblioteca <ticcommardu.h>**
- 11. Ejemplo: Envío de datos por láser**
- 12. Ejemplo: Recepción de datos por láser**
- 13. Proyecto final de la práctica**

SESIÓN 1

- 1. ¿Qué es Arduino?**
- 2. El procesador**
- 3. Compilador y bibliotecas para procesadores AVR**
- 4. Mi primer programa para AVR AtMega328p**
- 5. Comunicaciones en serie**
- 6. Mi primer programa para comunicaciones en serie: ECHO**
- 7. Recepción de datos desde sensores fotorresistores digitales**
- 8. Envío de datos por dispositivos emisores láser**
- 9. La arquitectura del sistema de emisión/recepción de datos por láser**
- 10. Ampliación de la biblioteca <ticcommardu.h>**
- 11. Ejemplo: Envío de datos por láser**
- 12. Ejemplo: Recepción de datos por láser**
- 13. Proyecto final de la práctica**

- **Arduino** es un conjunto de placas controladoras y un entorno de programación, Open hardware y software.
- Facilitan la elaboración de proyectos de electrónica, automatismo, control, domótica, etc.
- Existen varios modelos de Arduino como son Uno, Leonardo, Mega...
- En el laboratorio utilizaremos el modelo **Arduino Uno**, por ser el más económico, el de mayor flexibilidad y posibilidades en proporción a su precio, y porque tiene la capacidad suficiente para la construcción de los prototipos de las prácticas.



- El desarrollo de proyectos con Arduino conlleva el uso de uno o varios elementos que se integran en la placa: entradas, salidas, alimentación, comunicación y shields de extensión.

- **Entradas:** son pines incrustados en la placa. Se utilizan para adquirir datos desde sensores u otros dispositivos externos. En la placa Arduino Uno son los pines digitales (del 0 al 13) y los analógicos (del A0 al A5).
- **Salidas:** los pines de salidas se utilizan para el envío de datos a dispositivos externos o a actuadores. En este caso los pines de salida son los pines digitales (0 a 13), que pueden configurarse como entrada o como salida.
- **Otros pines de interés:** TX (transmisión) y RX (lectura) también usados para comunicación en serie, RESET para resetear el sistema, Vin para alimentar la placa con fuentes de alimentación externa, y los pines ICSP para comunicación por puerto SPI.

- El desarrollo de proyectos con Arduino conlleva el uso de uno o varios elementos que se integran en la placa: entradas, salidas, alimentación, comunicación y shields de extensión.

- **Alimentación:** Considerados para la alimentación de sensores y actuadores, tales como los pines GND (del inglés GROUND, tierra), pines 5V que proporcionan 5 Voltios, pines 3.3V que proporciona 3.3 Voltios, los pines REF de referencia de voltaje. El pin Vin sirve para alimentar la placa de forma externa, aunque lo normal es alimentarlo por USB o por el conector de alimentación usando un voltaje de 5 a 12 Voltios.
- **Comunicación:** Lo más normal es utilizar comunicación serie por USB para cargar los programas en la placa o para enviar/recibir datos. No obstante, existen shields que permiten extender la capacidad de comunicación de la placa utilizando los pines ICSP (comunicación ISP), los pines 10 a 13 (también usados para comunicación ISP), los pines TX/RX o cualquiera de los digitales.

- El desarrollo de proyectos con Arduino conlleva el uso de uno o varios elementos que se integran en la placa: entradas, salidas, alimentación, comunicación y shields de extensión.
- **Shields:** Son otras placas externas que se insertan sobre Arduino para extender sus capacidades. Algunas de las más comunes son las de Wi-Fi, sensores, actuadores (motores), Pantallas LCD, relés, matrices LED's, GPS, etc.

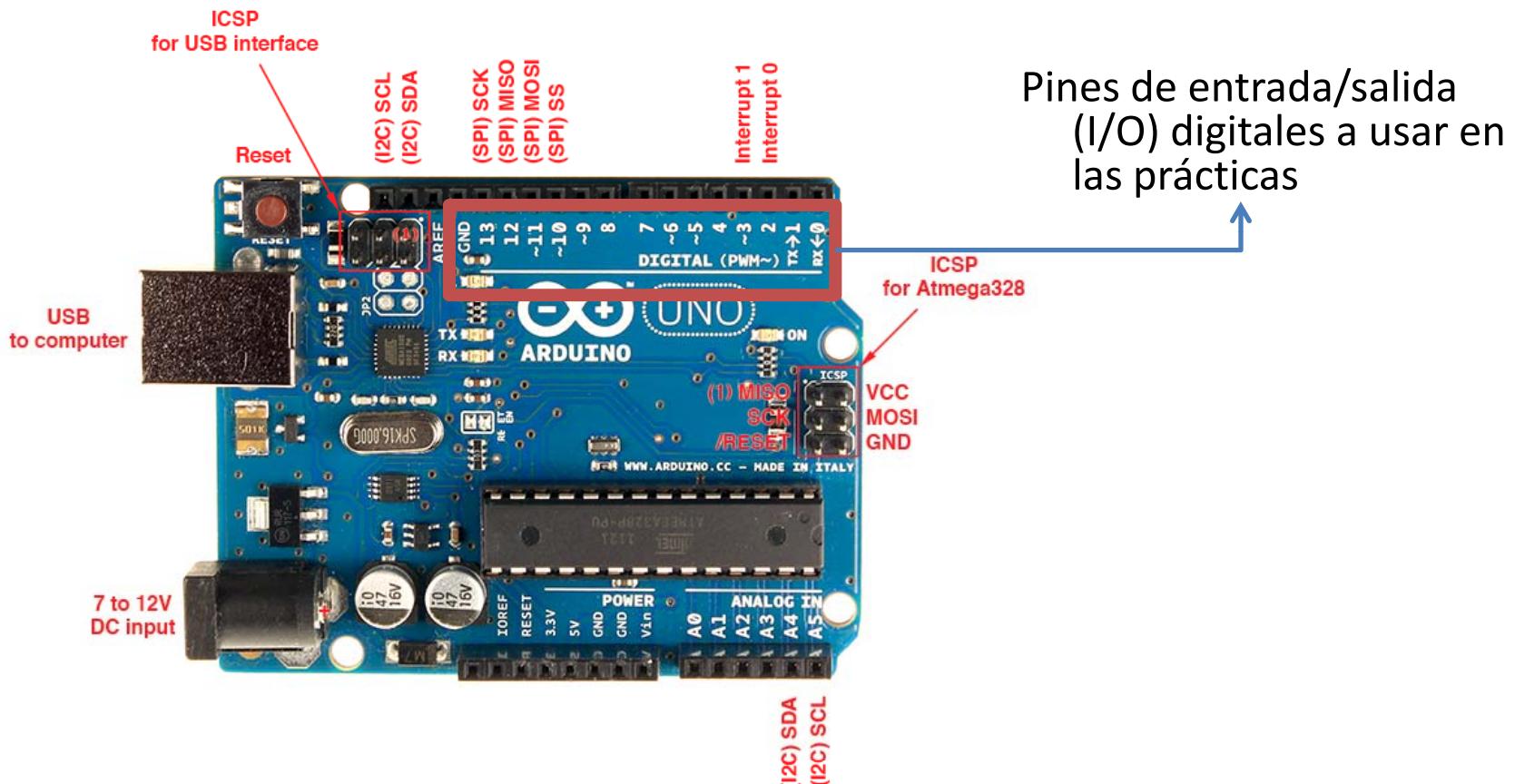


– Especificaciones técnicas:



Microcontroller	ATmega328P
Operating Voltage	5V
Input Voltage (recommended)	7-12V
Input Voltage (limit)	6-20V
Digital I/O Pins	14 (of which 6 provide PWM output)
PWM Digital I/O Pins	6
Analog Input Pins	6
DC Current per I/O Pin	20 mA
DC Current for 3.3V Pin	50 mA
Flash Memory	32 KB (ATmega328P) of which 0.5 KB used by bootloader
SRAM	2 KB (ATmega328P)
EEPROM	1 KB (ATmega328P)
Clock Speed	16 MHz
LED_BUILTIN	13
Length	68.6 mm
Width	53.4 mm
Weight	25 g

- Mapa de pines de la placa Arduino Uno (estándar):



- Arduino tiene su propio IDE de programación, con código similar a C. Es el utilizado principalmente por aquellos que se quieren iniciar en Arduino como hobby:

The screenshot shows the Arduino IDE interface. The main window displays the code for `ObstacleAvoidance.ino`. The code is written in a C-like syntax, defining `setup()` and `loop()` functions. The `setup()` function initializes the serial port at 9600 bps and attaches a digital pin to a motor. The `loop()` function checks if the robot is maneuvering; if not, it moves forward. It then calls `navigateWithWhiskers()` or `navigateWithSonar()`. The terminal window at the bottom shows the command-line interface used for building and uploading the sketch. The output includes the command `avr-objcopy -O ihex -R .eeprom` and the resulting file paths: `/var/folders/1v/84frnd63d37sg6gp3l2q332sw0000gn/T/build4867331055628351831.tmp/ObstacleAvoidance.cpp.eep`, `/var/folders/1v/84frnd63d37sg6gp3l2q332sw0000gn/T/build4867331055628351831.tmp/ObstacleAvoidance.cpp.elf`, and `/var/folders/1v/84frnd63d37sg6gp3l2q332sw0000gn/T/build4867331055628351831.tmp/ObstacleAvoidance.cpp.hex`. The message "Sketch uses 11,068 bytes (34%) of program storage space. Maximum is 32,256 bytes." is displayed.

```
ObstacleAvoidance | Arduino 1.5.6-r2

124
125 void setup() {
126     srand(millis());
127     Serial.begin(9600);
128
129     bot.attach();
130     bot.debug(true);
131
132     bot.setTurningSpeedPercent(80);
133
134     pinMode(leftWhiskerPin, INPUT);
135     pinMode(rightWhiskerPin, INPUT);
136 }
137
138 void loop() {
139     if (!bot.isManeuvering()) {
140         bot.goForward(speed);
141
142         // call our navigation processors one by one, but as soon as one of them
143         // starts maneuvering we skip the rest. If we bumped into whiskers, we sure
144         // don't need sonar to tell us we have a problem :)
145         navigateWithWhiskers() || navigateWithSonar(); // || ....
146     }
147 }
148

Done Saving.
/var/folders/1v/84frnd63d37sg6gp3l2q332sw0000gn/T/build4867331055628351831.tmp/ObstacleAvoidance.cpp.eep
/Applications/Arduino.app/Contents/Resources/Java/hardware/tools/avr/bin/avr-objcopy -O ihex -R .eeprom
/var/folders/1v/84frnd63d37sg6gp3l2q332sw0000gn/T/build4867331055628351831.tmp/ObstacleAvoidance.cpp.elf
/var/folders/1v/84frnd63d37sg6gp3l2q332sw0000gn/T/build4867331055628351831.tmp/ObstacleAvoidance.cpp.hex

Sketch uses 11,068 bytes (34%) of program storage space. Maximum is 32,256 bytes.
145
Arduino Uno on /dev/tty.usbserial-DA00WXY
```

- En las prácticas, nosotros no haremos uso de este IDE, sino del compilador GCC para procesadores AVR.

- Arduino es seguro, pero también puede romperse. Aquí enunciamos las 8 formas más comunes de destruir una placa Arduino:
 1. Conectar dos pines entre es arriesgado si no se sabe qué se hace. Si Especialmente si uno es tierra y el otro voltaje, la placa quedará dañada.
 2. Aplicar un voltaje superior a 5.5V a cualquier pin de entrada/salida.
 3. Al usar alimentación por Vin, invertir la corriente (conectar el positivo al negativo y viceversa).
 4. Conectar un voltaje superior al requerido en los pines de voltaje (por ejemplo, conectar 7V al pin de 5V o conectar 5V al pin de 3.3V).
 5. Conectar el voltaje directamente a tierra.
 6. Aplicar dos entradas de voltaje diferente para alimentar la placa (entrada externa Vin y entrada externa por el conector de alimentación).
 7. Aplicar un voltaje superior a 13V al pin RESET.
 8. Incluir en la placa una corriente superior a la soportada (la suma total de toda la corriente incluida en todos los pines de entrada/salida no puede superar los 200mA).

- 1. ¿Qué es Arduino?**
- 2. El procesador**
- 3. Compilador y bibliotecas para procesadores AVR**
- 4. Mi primer programa para AVR AtMega328p**
- 5. Comunicaciones en serie**
- 6. Mi primer programa para comunicaciones en serie: ECHO**
- 7. Recepción de datos desde sensores fotorresistores digitales**
- 8. Envío de datos por dispositivos emisores láser**
- 9. La arquitectura del sistema de emisión/recepción de datos por láser**
- 10. Ampliación de la biblioteca <ticcommardu.h>**
- 11. Ejemplo: Envío de datos por láser**
- 12. Ejemplo: Recepción de datos por láser**
- 13. Proyecto final de la práctica**

- Arduino Uno utiliza un microprocesador **AVR Atmega328 de Atmel**.



- Son procesadores de bajo consumo.
- Son de bajo precio.
- Tienen unas capacidades suficientes para el desarrollo de prototipos básicos.

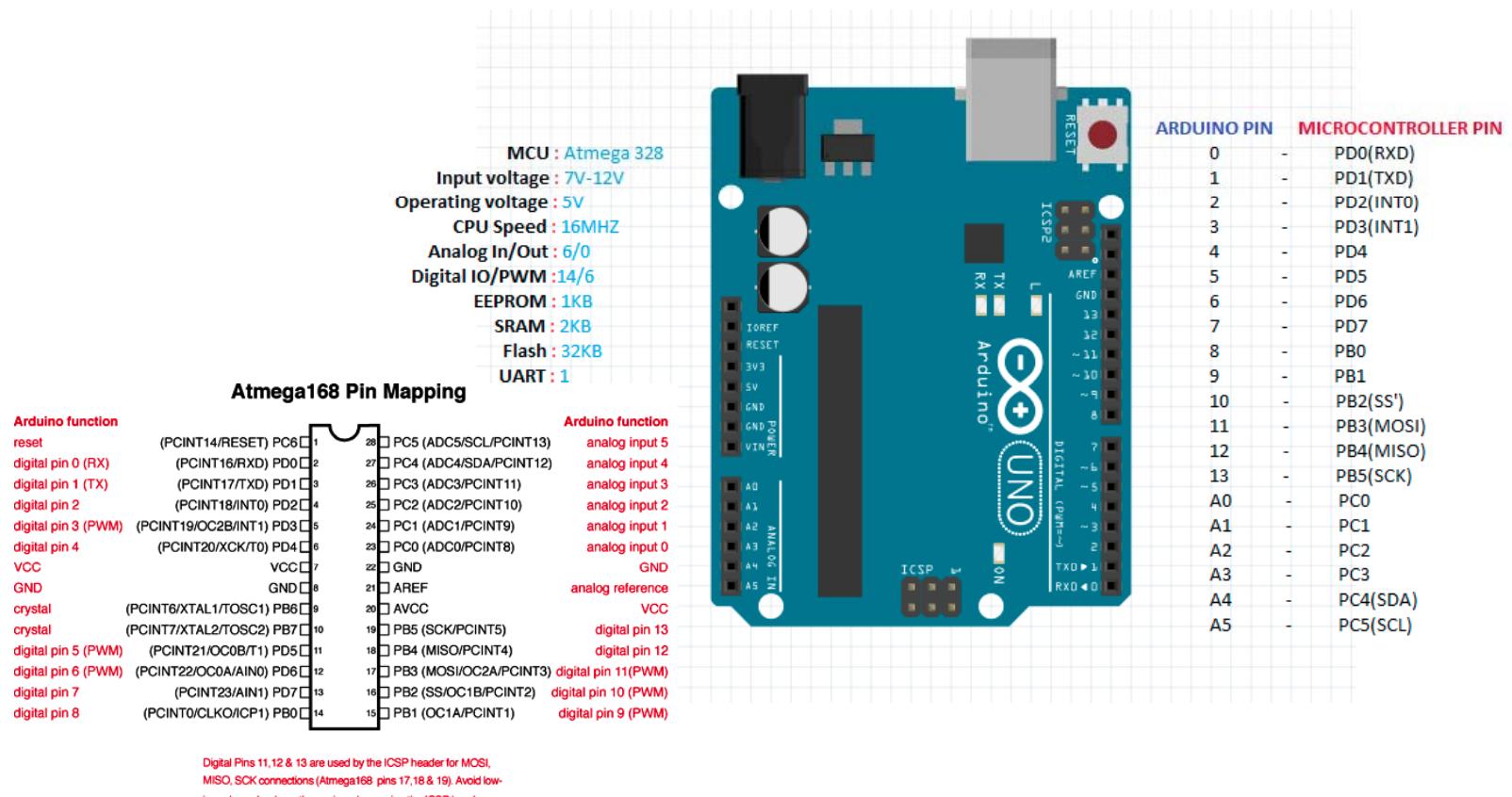
- El mapa de puertos del procesador es el siguiente:

Atmega168 Pin Mapping

Arduino function			Arduino function
reset	(PCINT14/RESET) PC6	1	28 □ PC5 (ADC5/SCL/PCINT13)
digital pin 0 (RX)	(PCINT16/RXD) PD0	2	27 □ PC4 (ADC4/SDA/PCINT12)
digital pin 1 (TX)	(PCINT17/TXD) PD1	3	26 □ PC3 (ADC3/PCINT11)
digital pin 2	(PCINT18/INT0) PD2	4	25 □ PC2 (ADC2/PCINT10)
digital pin 3 (PWM)	(PCINT19/OC2B/INT1) PD3	5	24 □ PC1 (ADC1/PCINT9)
digital pin 4	(PCINT20/XCK/T0) PD4	6	23 □ PC0 (ADC0/PCINT8)
VCC	VCC	7	22 □ GND
GND	GND	8	21 □ AREF
crystal	(PCINT6/XTAL1/TOSC1) PB6	9	20 □ AVCC
crystal	(PCINT7/XTAL2/TOSC2) PB7	10	19 □ PB5 (SCK/PCINT5)
digital pin 5 (PWM)	(PCINT21/OC0B/T1) PD5	11	18 □ PB4 (MISO/PCINT4)
digital pin 6 (PWM)	(PCINT22/OC0A/AIN0) PD6	12	17 □ PB3 (MOSI/OC2A/PCINT3)
digital pin 7	(PCINT23/AIN1) PD7	13	16 □ PB2 (SS/OC1B/PCINT2)
digital pin 8	(PCINT0/CLKO/ICP1) PB0	14	15 □ PB1 (OC1A/PCINT1)
			GND
			analog reference
			VCC
			digital pin 13
			digital pin 12
			digital pin 11(PWM)
			digital pin 10 (PWM)
			digital pin 9 (PWM)

Digital Pins 11,12 & 13 are used by the ICSP header for MOSI, MISO, SCK connections (Atmega168 pins 17,18 & 19). Avoid low-impedance loads on these pins when using the ICSP header.

- La correspondencia de los puertos con los pines de la placa Arduino Uno es la siguiente:



- 1. ¿Qué es Arduino?**
- 2. El procesador**
- 3. Compilador y bibliotecas para procesadores AVR**
- 4. Mi primer programa para AVR AtMega328p**
- 5. Comunicaciones en serie**
- 6. Mi primer programa para comunicaciones en serie: ECHO**
- 7. Recepción de datos desde sensores fotorresistores digitales**
- 8. Envío de datos por dispositivos emisores láser**
- 9. La arquitectura del sistema de emisión/recepción de datos por láser**
- 10. Ampliación de la biblioteca <ticcommardu.h>**
- 11. Ejemplo: Envío de datos por láser**
- 12. Ejemplo: Recepción de datos por láser**
- 13. Proyecto final de la práctica**

- En las prácticas, utilizaremos el lenguaje C para programar el microcontrolador **AVR Atmega328**. Existe una versión del compilador GCC para este tipo de microprocesadores. En Ubuntu, la instalación de los paquetes necesarios se realiza a través de *synaptic* o de *apt-get*:

— INSTALACION DEL COMPILEADOR Y UTILIDADES:

`sudo apt-get install gcc-avr` (*alternativamente, sudo apt-get install avr-gcc*)

`sudo apt-get install avrdude`

— INSTALACION DE BIBLIOTECAS PARA EL COMPILEADOR:

`sudo apt-get install avr-libc` (*alternativamente, sudo apt-get install libc-avr*)

– Ejemplo de instalación por línea de comandos:

```
manupc@manupcws: ~$ sudo apt-get install gcc-avr
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
gcc-avr ya está en su versión más reciente (1:4.9.2+Atmel3.5.0-1).
0 actualizados, 0 nuevos se instalarán, 0 para eliminar y 177 no actualizados.
manupc@manupcws: ~$ sudo apt-get install avrdude
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
avrdude ya está en su versión más reciente (6.2-5).
0 actualizados, 0 nuevos se instalarán, 0 para eliminar y 177 no actualizados.
manupc@manupcws: ~$ sudo apt-get install avr-libc
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
avr-libc ya está en su versión más reciente (1:1.8.0+Atmel3.5.0-1).
0 actualizados, 0 nuevos se instalarán, 0 para eliminar y 177 no actualizados.
manupc@manupcws: ~$
```

- 1. ¿Qué es Arduino?**
- 2. El procesador**
- 3. Compilador y bibliotecas para procesadores AVR**
- 4. Mi primer programa para AVR AtMega328p**
- 5. Comunicaciones en serie**
- 6. Mi primer programa para comunicaciones en serie: ECHO**
- 7. Recepción de datos desde sensores fotorresistores digitales**
- 8. Envío de datos por dispositivos emisores láser**
- 9. La arquitectura del sistema de emisión/recepción de datos por láser**
- 10. Ampliación de la biblioteca <ticcommardu.h>**
- 11. Ejemplo: Envío de datos por láser**
- 12. Ejemplo: Recepción de datos por láser**
- 13. Proyecto final de la práctica**

- **Bibliotecas y macros útiles:**

- **Biblioteca `<avr/io.h>`:** Contiene las definiciones de variables para direccionar puertos.
- **Biblioteca `<util/delay.h>`:** Contiene las definiciones de funciones para la ejecución del programa por los instantes de tiempo requeridos.
- Macro **`F_CPU`**: Indica cada cuánto tiempo se refresca el tick del procesador.
- Macro **`#define _BV(bit) (1 << (bit))`**. Definida en `<avr/io.h>`: Se utiliza para transformar un bit a byte.

En las prácticas, nuestros ficheros de código fuente comenzarán así:

```
// Utilizado para que el procesador pueda calcular el delay a partir del número de ticks del procesador
#define F_CPU 1000000UL

#include <avr/io.h>
#include <util/delay.h>
```

- Escribiremos el siguiente programa en un fichero **main.cpp**:

```
// Utilizado para el cálculo de ms en _delay_ms
#define F_CPU 1000000UL

#include <avr/io.h>
#include <util/delay.h>

#define BLINK_DELAY_MS 1000

int main (void)
{
    /* Pin 0 del puerto B del micro puesto como salida */
    DDRB |= _BV(DDRB0);

    while(1) {
        /* Mandamos señal de voltaje alto al pin 0 del puerto B */
        PORTB |= _BV(PORTB0);
        _delay_ms(BLINK_DELAY_MS);

        /* Mandamos señal de voltaje bajo al pin 0 del puerto B */
        PORTB &= ~_BV(PORTB0); // ~ es el NOT lógico a nivel de bits
        _delay_ms(BLINK_DELAY_MS);
    }
}
```

Qué vemos nuevo:

- Operadores a nivel de bits de C/C++:
 - $a \mid b$: Realiza el or a nivel de bits de a y b. Ejemplo:

unsigned char a= 1, b=4, c=a | b; // c vale 5: 00000101

- $00000001 \mid 00000100 = 00000101$
- $a \& b$: Realiza el and a nivel de bits de a y b. Ejemplo:

unsigned char a= 3, b=1, c=a & b; // c vale 1: 00000001

- $00000011 \& 00000001 = 00000001$

Los operadores tienen sus correspondientes $\mid=$, $\&=$.

- Operadores a nivel de bits de C/C++:
 - $\sim b$: Realiza el not a nivel de bits de a y b. Ejemplo:
unsigned char a= 1, c= $\sim a$; // c vale 254: 11111110
 - $\sim 00000001 = 11111110$
 - $A << n$: Desplaza todos los bits de A n posiciones hacia la izquierda, introduciendo n 0's por la derecha.
A= 1; // 0b00000001 A<<=2; → // 0b00000100
 - $A >> n$: Desplaza todos los bits de A n posiciones hacia la izquierda, introduciendo n 0's por la derecha.
A= 3; // 0b00000011 A>>=1; → // 0b00000001

– ¿Qué hemos hecho ??

```
// Utilizado para el cálculo de ms en _delay_ms
#define F_CPU 10000000UL

#include <avr/io.h>
#include <util/delay.h>

#define BLINK_DELAY_MS 1000

int main (void)
{
    /* Pin 0 del puerto B del micro puesto como salida */
    DDRB |= _BV(DDB0);

    while(1) {
        /* Mandamos señal de voltaje alto al pin 0 del puerto B */
        PORTB |= _BV(PORTB0);
        _delay_ms(BLINK_DELAY_MS);

        /* Mandamos señal de voltaje bajo al pin 0 del puerto B */
        PORTB &= ~_BV(PORTB0); // ~ es el NOT lógico a nivel de bits
        _delay_ms(BLINK_DELAY_MS);
    }
}
```

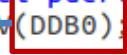
DDRB es el registro de direccionamiento de datos del puerto B del **microprocesador** (PB0-PB7).

– ¿Qué hemos hecho ??

```
// Utilizado para el cálculo de ms en _delay_ms
#define F_CPU 10000000UL

#include <avr/io.h>
#include <util/delay.h>

#define BLINK_DELAY_MS 1000

int main (void)
{
    /* Pin 0 del puerto B del micro puesto como salida */
    DDRB |= _BV(DDB0);  
  
    while(1) {
        /* Mandamos señal de voltaje alto al pin 0 del puerto B */
        PORTB |= _BV(PORTB0);
        _delay_ms(BLINK_DELAY_MS);

        /* Mandamos señal de voltaje bajo al pin 0 del puerto B */
        PORTB &= ~_BV(PORTB0); // ~ es el NOT lógico a nivel de bits
        _delay_ms(BLINK_DELAY_MS);
    }
}
```



DDB0 es la dirección del PIN 0 del puerto B del **microprocesador** (el pin PB0)

– ¿Qué hemos hecho ??

```
// Utilizado para el cálculo de ms en _delay_ms
#define F_CPU 10000000UL

#include <avr/io.h>
#include <util/delay.h>

#define BLINK_DELAY_MS 1000

int main (void)
{
    /* Pin 0 del puerto B del micro puesto como salida */
    DDRB |= _BV(DDB0);||

    while(1) {
        /* Mandamos señal de voltaje alto al pin 0 del puerto B */
        PORTB |= _BV(PORTB0);
        _delay_ms(BLINK_DELAY_MS);

        /* Mandamos señal de voltaje bajo al pin 0 del puerto B */
        PORTB &= ~_BV(PORTB0); // ~ es el NOT lógico a nivel de bits
        _delay_ms(BLINK_DELAY_MS);
    }
}
```

Haciendo esto estamos indicando que en el puerto B (microprocesador), el PIN 0 será de salida. Es decir, estamos haciendo $DDRB = DDRB | 0x01$.

```
// utilizado para el cálculo de ms en _delay_ms
#define F_CPU 10000000UL
```

```
#include <avr/io.h>
#include <util/delay.h
```

```
#define BLINK_DELAY_MS

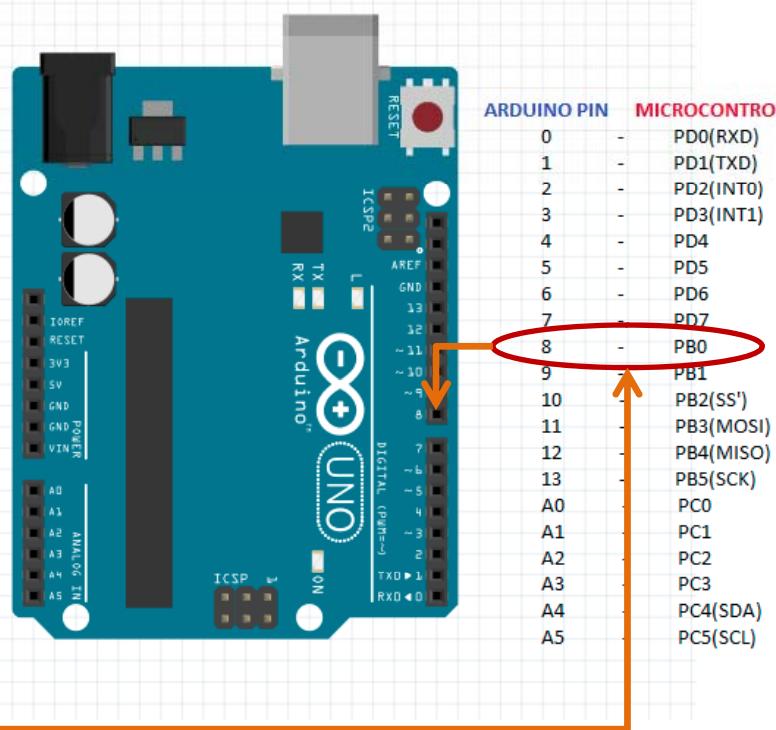
int main (void)
{
    /* Pin 0 del puerto B
    DDRB |= _BV(DDB0);
```

Atmega168 Pin Mapping

	Arduino function	PCINT14/RESET	PC6	PC5 (ADC5/SCL/PCINT13)	analog input 5
P	reset			(PCINT16/RXD) PD0	analog input 4
	digital pin 0 (RX)			(PCINT17/TXD) PD1	analog input 3
-	digital pin 1 (TX)			(PCINT18/INT0) PD2	analog input 2
	digital pin 2			(PCINT19/OC2B/INT1) PD3	analog input 1
	digital pin 3 (PWM)	(PCINT10/OC2B/INT1)	PD3	(PCINT20/XCK/T0) PD4	analog input 0
P	digital pin 4			VCC	GND
VCC				GND	
GND					
crystal		(PCINT6/XTAL1/TOSC1)	PB6		
crystal		(PCINT7/XTAL2/TOSC2)	PB7		
	digital pin 5 (PWM)	(PCINT21/OC0B/T1)	PD5		
	digital pin 6 (PWM)	(PCINT22/OC0A/AIN0)	PD6		
	digital pin 7	(PCINT23/AIN1)	PD7		
	digital pin 8	(PCINT0/CLKO/ICP1)	PB0		

Digital Pins 11,12 & 13 are used by the ICSP header for MOSI, MISO, SCK connections (Atmega168 pins 17,18 & 19). Avoid low impedance loads on these pins when using the ICSP header.

- ¿¿ Qué hemos hecho ??: Hemos dicho que el Pin digital 8 de Arduino es de salida.



– ¿Qué hemos hecho ??

```
// Utilizado para el cálculo de ms en _delay_ms
#define F_CPU 10000000UL

#include <avr/io.h>
#include <util/delay.h>

#define BLINK_DELAY_MS 1000

int main (void)
{
    /* Pin 0 del puerto B del micro puesto como salida */
    DDRB |= _BV(DDB0); // Linea resaltada con un cuadro rojo

    while(1) {
        /* Mandamos señal de voltaje alto al pin 0 del puerto B */
        PORTB |= _BV(PORTB0);
        _delay_ms(BLINK_DELAY_MS);

        /* Mandamos señal de voltaje bajo al pin 0 del puerto B */
        PORTB &= ~ BV(PORTB0); // ~ es el NOT lógico a nivel de bits
    }
}
```

Conclusión parcial: El puerto DDRB del microprocesador controla los pines PB0 a PB7 del microprocesador, de los cuales los pines PB0 a PB5 se corresponden con los pines digitales de I/O de Arduino Uno, desde el pin 8 hasta el pin 13.

```
// utilizado para el cálculo de ms en _delay_ms
#define F_CPU 10000000UL
```

```
#include <avr/io.h>
#include <util/delay.h
```

```
#define BLINK_DELAY_MS

int main (void)
{
    /* Pin 0 del puerto B
    DDRB |= _BV(DDB0); */

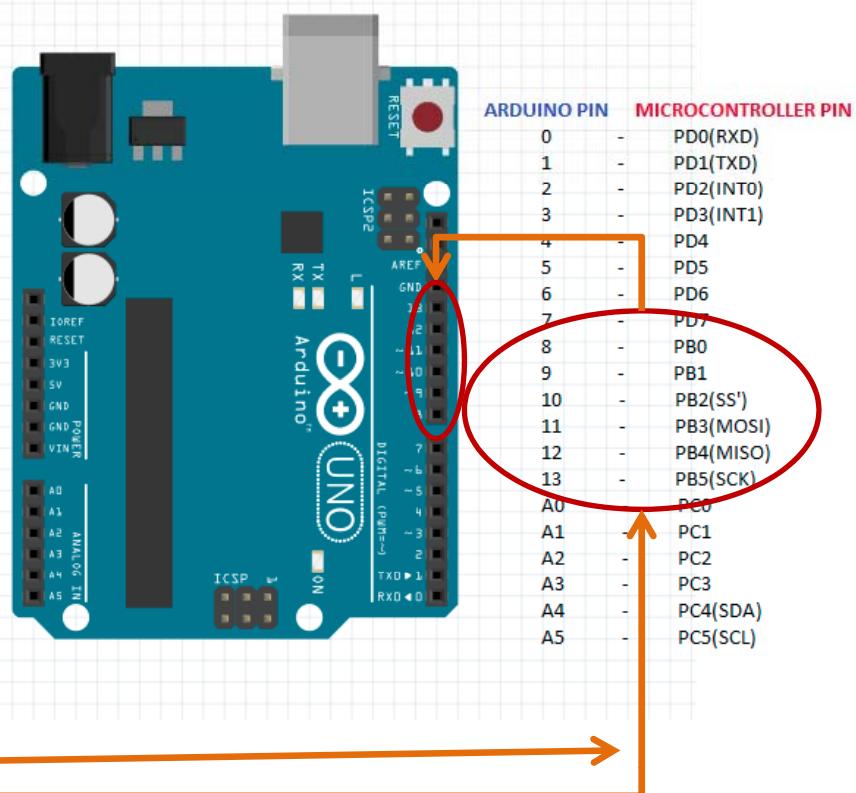
    wh
```

Atmega168 Pin Mapping

	Arduino function	PCINT function
P	reset	(PCINT14/RESET) PC6
	digital pin 0 (RX)	(PCINT16/RXD) PD0
	digital pin 1 (TX)	(PCINT17/TXD) PD1
	digital pin 2	(PCINT18/INT0) PD2
	digital pin 3 (PWM)	(PCINT19/OC2B/INT1) PD3
P	digital pin 4	(PCINT20/XCK/T0) PD4
VCC		VCC
GND		GND
crystal		(PCINT6/XTAL1/TOSC1) PB6
crystal		(PCINT7/XTAL2/TOSC2) PB7
	digital pin 5 (PWM)	(PCINT21/OC0B/T1) PD5
	digital pin 6 (PWM)	(PCINT22/OC0A/AIN0) PD6
	digital pin 7	(PCINT23/AN1) PD7
	digital pin 8	(PCINT0/CLKO/ICP1) PB0

Digital Pins 11,12 & 13 are used by the ICSP header for MOSI, MISO, SCK connections (Atmega168 pins 17,18 & 19). Avoid low-impedance loads on these pins when using the ICSP header.

- ¿¿ Qué hemos hecho ??: Hemos dicho que el Pin digital 8 de Arduino es de salida.



— Y después... ¿Qué hemos hecho ??

```
// Utilizado para el cálculo de ms en _delay_ms
#define F_CPU 10000000UL

#include <avr/io.h>
#include <util/delay.h>

#define BLINK_DELAY_MS 1000

int main (void)
{
    /* Pin 0 del puerto B del micro puesto como salida */
    DDRB |= _BV(DDB0);

    while(1) {
        /* Mandamos señal de voltaje alto al pin 0 del puerto B */
        PORTB |= _BV(PORTB0);
        _delay_ms(BLINK_DELAY_MS);

        /* Mandamos señal de voltaje bajo al pin 0 del puerto B */
        PORTB &= ~_BV(PORTB0); // ~ es el NOT lógico a nivel de bits
        _delay_ms(BLINK_DELAY_MS);
    }
}
```



PORTB es un byte que contiene los datos de salida existentes en los pines PBO-PB7 del **microprocesador**.

— Y después... ¿Qué hemos hecho ??

```
// Utilizado para el cálculo de ms en _delay_ms
#define F_CPU 10000000UL

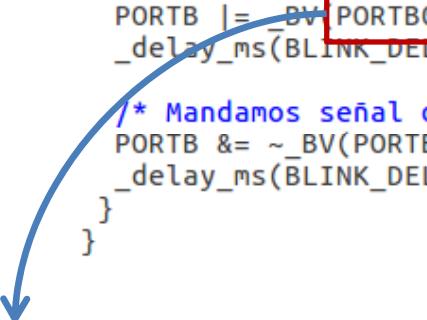
#include <avr/io.h>
#include <util/delay.h>

#define BLINK_DELAY_MS 1000

int main (void)
{
    /* Pin 0 del puerto B del micro puesto como salida */
    DDRB |= _BV(DDB0);

    while(1) {
        /* Mandamos señal de voltaje alto al pin 0 del puerto B */
        PORTB |= _BV(PORTB0);
        _delay_ms(BLINK_DELAY_MS);

        /* Mandamos señal de voltaje bajo al pin 0 del puerto B */
        PORTB &= ~_BV(PORTB0); // ~ es el NOT lógico a nivel de bits
        _delay_ms(BLINK_DELAY_MS);
    }
}
```



PORTB0 es el bit 0: Corresponde al pin PB0 del **microprocesador**.

— Y después... ¿Qué hemos hecho ??

```
// Utilizado para el cálculo de ms en _delay_ms
#define F_CPU 1000000UL

#include <avr/io.h>
#include <util/delay.h>

#define BLINK_DELAY_MS 1000

int main (void)
{
    /* Pin 0 del puerto B del micro puesto como salida */
    DDRB |= _BV(DDB0);

    while(1) {
        /* Mandamos señal de voltaje alto al pin 0 del puerto B */
        PORTB |= _BV(PORTB0);
        _delay_ms(BLINK_DELAY_MS);

        /* Mandamos señal de voltaje bajo al pin 0 del puerto B */
        PORTB &= ~_BV(PORTB0); // ~ es el NOT lógico a nivel de bits
        _delay_ms(BLINK_DELAY_MS);
    }
}
```



Haciendo esto estamos indicando al microprocesador que envíe voltaje alto por el pin 0 del puerto B. Es decir, estamos haciendo que los datos del puerto B sean $\text{PORTB} = \text{PORTB} | 0x01$. El Pin 8 de Arduino enviará voltaje.

— Y después... ¿Qué hemos hecho ??

```
// Utilizado para el cálculo de ms en _delay_ms
#define F_CPU 10000000UL

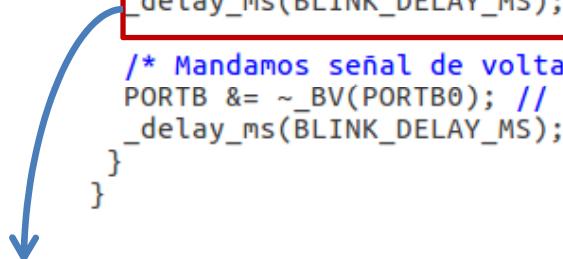
#include <avr/io.h>
#include <util/delay.h>

#define BLINK_DELAY_MS 1000

int main (void)
{
    /* Pin 0 del puerto B del micro puesto como salida */
    DDRB |= _BV(DDB0);

    while(1) {
        /* Mandamos señal de voltaje alto al pin 0 del puerto B */
        PORTB |= _BV(PORTB0);
        _delay_ms(BLINK_DELAY_MS);

        /* Mandamos señal de voltaje bajo al pin 0 del puerto B */
        PORTB &= ~_BV(PORTB0); // ~ es el NOT lógico a nivel de bits
        _delay_ms(BLINK_DELAY_MS);
    }
}
```



Aquí estamos esperando a que pase cierto tiempo

— Y después... ¿Qué hemos hecho ??

```
// Utilizado para el cálculo de ms en _delay_ms
#define F_CPU 10000000UL

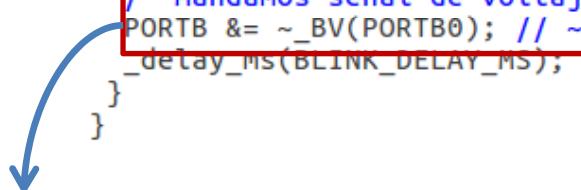
#include <avr/io.h>
#include <util/delay.h>

#define BLINK_DELAY_MS 1000

int main (void)
{
    /* Pin 0 del puerto B del micro puesto como salida */
    DDRB |= _BV(DDB0);

    while(1) {
        /* Mandamos señal de voltaje alto al pin 0 del puerto B */
        PORTB |= _BV(PORTB0);
        _delay_ms(BLINK_DELAY_MS);

        /* Mandamos señal de voltaje bajo al pin 0 del puerto B */
        PORTB &= ~_BV(PORTB0); // ~ es el NOT lógico a nivel de bits
        _delay_ms(BLINK_DELAY_MS);
    }
}
```



Aquí estamos haciendo la operación inversa: Poner a 0 la salida por el pin PB0 del microprocesador. El Pin 8 de Arduino enviará voltaje bajo (0V).

- Pasos para la compilación y envío del programa a la placa:

1. Compilación:

```
avr-gcc -Os -mmcu=atmega328p -c -o main.o main.cpp  
avr-gcc -mmcu=atmega328p main.o -o main  
avr-objcopy -O ihex -R .eeprom main main.hex
```

2. Búsqueda del puerto de conexión de Arduino Uno (*nota: Arduino Uno tiene que estar conectado al PC por el puerto USB*):

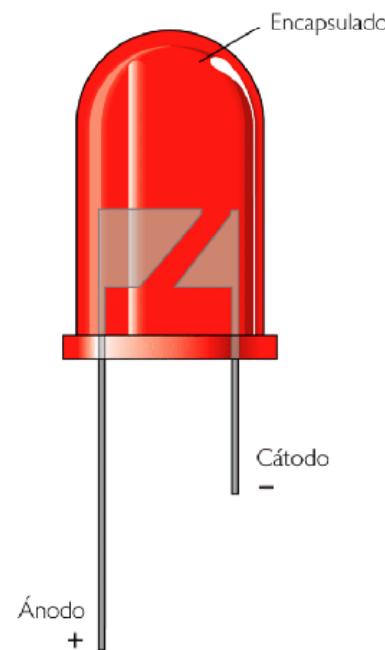
```
lsusb  
ls /dev/serial/by-id -l
```

3. Envío del programa a Arduino Uno:

```
sudo avrdude -F -V -c arduino -p ATMEGA328P -P /dev/ttyACM0 -b 115200 -U flash:w:main.hex
```

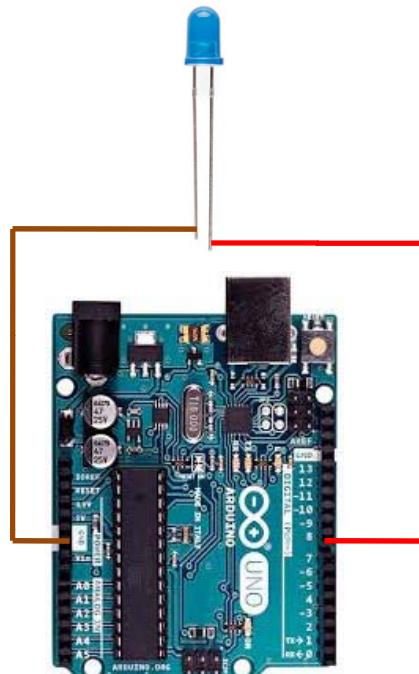
– Prueba de funcionamiento:

1. Seleccione un diodo LED en el laboratorio, y 2 cables de conexión macho-hembra.
2. Localice el cátodo (polo negativo) y el ánodo (polo positivo) del LED.



– Prueba de funcionamiento:

3. Conecte el extremo hembra de un cable al cátodo del LED, y el extremo macho a un pin GND de la placa Arduino.
4. Conecte el extremo hembra de otro cable al ánodo del LED, y el extremo macho al PIN DIGITAL 8.
5. Respuesta esperada: El LED debe parpadear.



— Ejercicio:

- Ahora, realice la misma prueba conectando el LED al pin digital 12 de la placa Arduino Uno y a la tierra (GND) colocada consecutivamente a este pin, como ilustra la figura.



Modifique el programa anterior para que la respuesta del LED sea la misma. Para ello debe:

1. Establecer el pin digital 12 de la placa Arduino Uno como salida.
2. En el bucle, enviar voltaje alto o bajo (según sea para encender o apagar el LED) al pin digital 12 de la placa.
3. **Revise el programa antes de enviarlo a la placa, para evitar escribir en puertos erróneos o que estén establecidos para lectura.**

SESIÓN 2

- 1. ¿Qué es Arduino?**
- 2. El procesador**
- 3. Compilador y bibliotecas para procesadores AVR**
- 4. Mi primer programa para AVR AtMega328p**
- 5. Comunicaciones en serie**
- 6. Mi primer programa para comunicaciones en serie: ECHO**
- 7. Recepción de datos desde sensores fotorresistores digitales**
- 8. Envío de datos por dispositivos emisores láser**
- 9. La arquitectura del sistema de emisión/recepción de datos por láser**
- 10. Ampliación de la biblioteca <ticcommardu.h>**
- 11. Ejemplo: Envío de datos por láser**
- 12. Ejemplo: Recepción de datos por láser**
- 13. Proyecto final de la práctica**

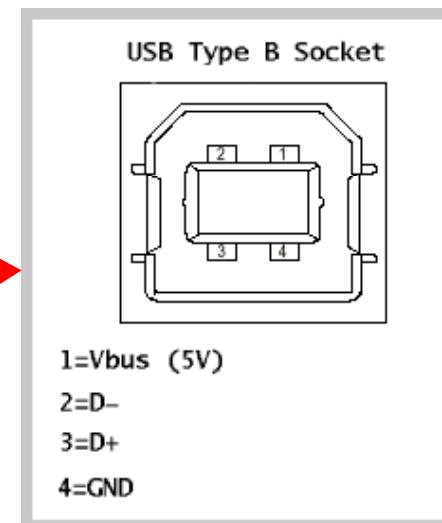
- **IMPORTANTE:** En Linux, el usuario tiene que formar parte del grupo **dialout**. En caso contrario, no podremos leer el puerto serie 0, en caso de poder, sólo se leerá basura. Tampoco podremos escribir (o, de poder, escribiremos basura).

```
sudo adduser ElUsuario dialout
```



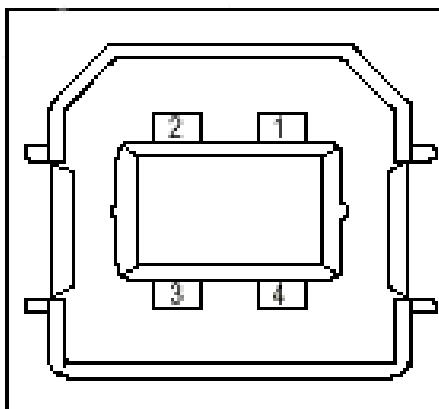
- Podemos comunicarnos con la placa Arduino de múltiples formas, aunque la mayoría de ellas requieren de shields adicionales:
 - Shield Wi-Fi
 - Shield Bluetooth
 - Pantallas táctiles
 - Etc.
- Todas las placas Arduino tienen posibilidad de comunicaciones por puerto serie (como mínimo un puerto serie). Así es como grabamos los programas en la memoria de la placa Arduino.
- Las **comunicaciones en serie** se realizan a través de **puertos UART**.

- El “**Bus Universal en Serie**” (**USB**) es un bus estándar que define las conexiones y protocolos para conectar, comunicar y alimentar periféricos y dispositivos electrónicos.
- En Arduino, la **comunicación por USB** se realiza a través del puerto serie **UART**. El conector utilizado en la placa es un **USB Tipo B**.



– Descripción del conector USB Tipo B:

USB Type B Socket



1=Vbus (5V)

2=D-

3=D+

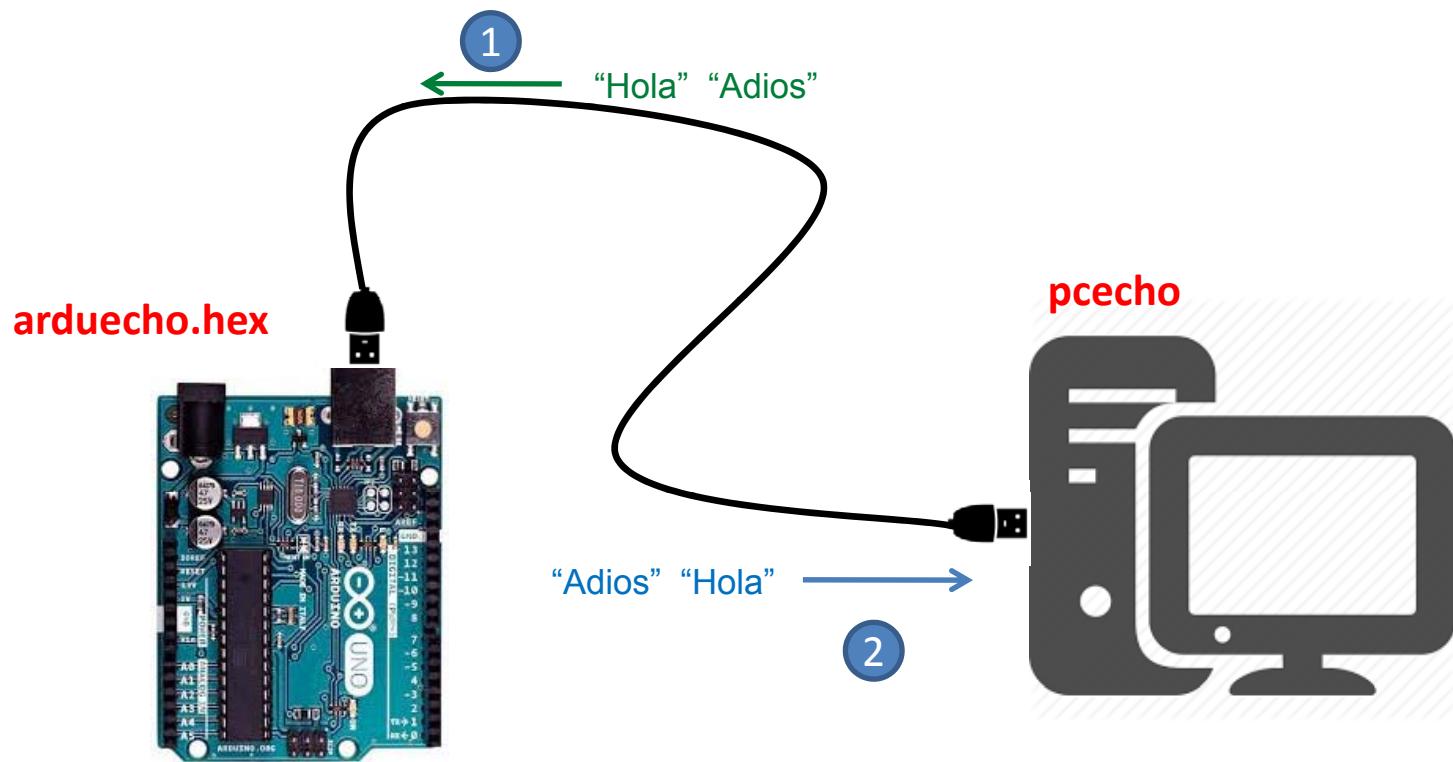
4=GND

- **Pin 1:** Alimentación con un voltaje de 5V DC
- **Pines 2 y 3:** Transmisión de datos
- **Pin 4:** Toma de tierra

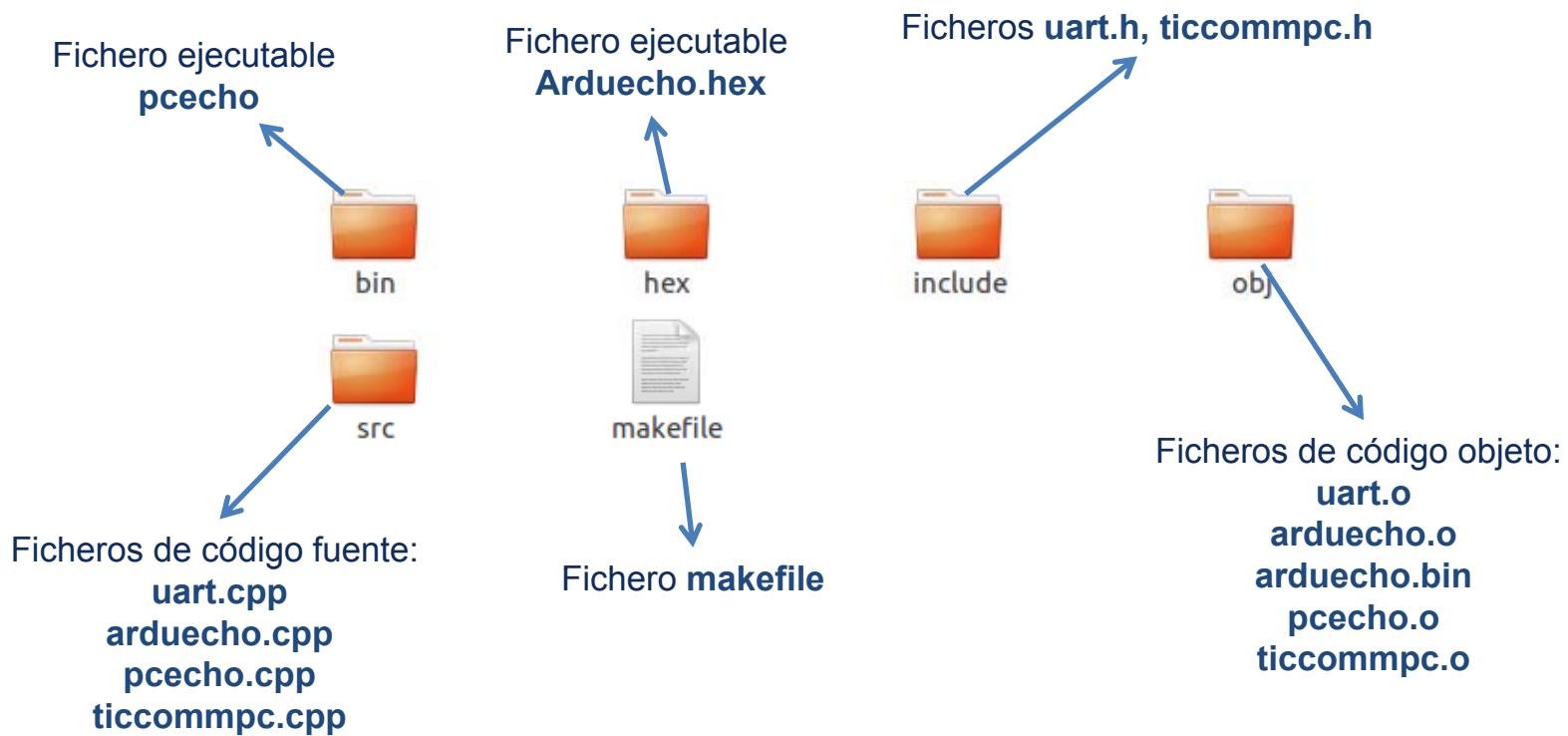
- 1. ¿Qué es Arduino?**
- 2. El procesador**
- 3. Compilador y bibliotecas para procesadores AVR**
- 4. Mi primer programa para AVR AtMega328p**
- 5. Comunicaciones en serie**
- 6. Mi primer programa para comunicaciones en serie: ECHO**
- 7. Recepción de datos desde sensores fotorresistores digitales**
- 8. Envío de datos por dispositivos emisores láser**
- 9. La arquitectura del sistema de emisión/recepción de datos por láser**
- 10. Ampliación de la biblioteca <ticcommardu.h>**
- 11. Ejemplo: Envío de datos por láser**
- 12. Ejemplo: Recepción de datos por láser**
- 13. Proyecto final de la práctica**

- Ejemplo de sistema para comunicación bidireccional PC-Arduino por puerto USB:
 - Construiremos un sistema cliente-servidor.
 - Arduino será el servidor (programa **arduecho.cpp**):
 - Estará escuchando el puerto USB para recibir datos.
 - Enviará por el puerto USB los mismos datos recibidos (implementación de un servicio ECHO).
 - Utilizaremos una biblioteca existente **<uart.h>** que nos facilite la tarea de enviar y recibir datos por **UART**.
 - El PC será el cliente (programa **pcecho.cpp**):
 - Se conectará al puerto USB para enviar datos recibidos desde la línea de comandos.
 - Esperará a que Arduino responda por el puerto serie USB, con datos de respuesta, y los mostrará por pantalla.

– Arquitectura general del sistema:

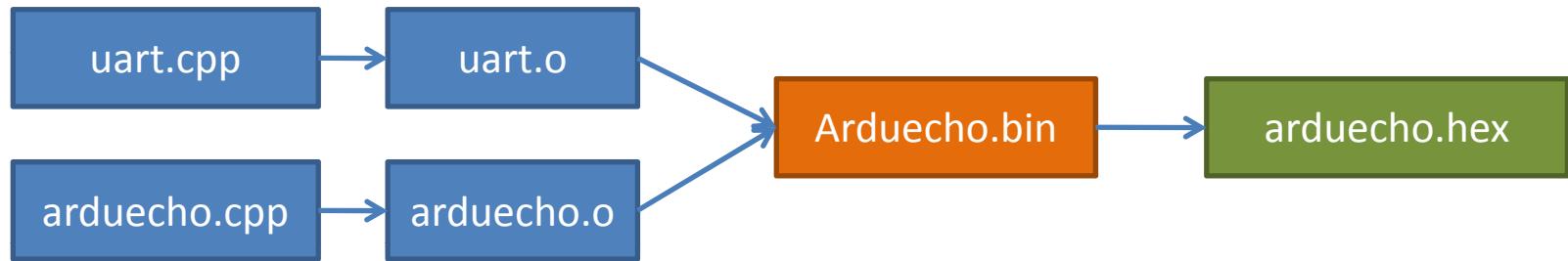


— Estructura de carpetas del proyecto:



- El programa arduecho: Modelo de compilación

- Compilaremos según el siguiente esquema:



- Crearemos un fichero makefile para evitar repetir estas operaciones varias veces en el futuro.

– El programa arduecho: Sección del fichero makefile

- El esquema del fichero makefile será como sigue:

```
# Puerto de comunicaciones donde se encuentra Arduino.
COMMPORT = /dev/ttyACM0

arduecho: arduechocpp uartcpp
    @echo Generando binario Arduino...
    avr-gcc -mmcu=atmega328p obj/arduecho.o obj/uart.o -o obj/arduecho.bin
    @echo Generando HEX Arduino...
    avr-objcopy -O ihex -R .eeprom obj/arduecho.bin hex/arduecho.hex
    @echo Arduecho compilado. Ejecute make send para enviarlo a la plataforma Arduino.

arduechocpp:
    @echo Compilando arduecho...
    avr-gcc -Os -mmcu=atmega328p -c -o obj/arduecho.o src/arduecho.cpp -Iinclude

uartcpp:
    @echo Compilando biblioteca UART...
    avr-gcc -Os -mmcu=atmega328p -c -o obj/uart.o src/uart.cpp -Iinclude

send:
    @echo Enviando arduecho a Arduino
    sudo avrdude -F -V -c arduino -p ATMEGA328P -P $(COMMPORT) -b 115200 -U flash:w:hex/arduecho.hex
```

– El programa src/arduecho.cpp: Código fuente (I)

```
3 // Ciclo de reloj del procesador: 16MHz
4 #define F_CPU 16000000UL
5
6 // Velocidad (en baudios) de las comunicaciones serie
7 #define UART_BAUD_RATE 9600
8
9 #include <avr/io.h>
10 #include <avr/interrupt.h>
11 #include <uart.h>
12 #include <util/delay.h>
13
14
15
16
17 int main(void) {
18
19     // Byte de datos recibido por puerto serie
20     // El byte menos significativo es el dato
21     // El segundo byte menos significativo contiene
22     //   información de error
23     unsigned int c;
24
25
26     // Inicialización del puerto UART con la velocidad
27     //   en baudios del puerto, y la velocidad del procesador
28     uart_init( UART_BAUD_SELECT(UART_BAUD_RATE,F_CPU) );
29
30
31     // Activación de las interrupciones hardware para
32     //   control del puerto serie
33     sei();
34 }
```

– El programa src/arduecho.cpp: Código fuente (I)

```
3 // Ciclo de reloj del procesador: 16MHz
4 #define F_CPU 16000000UL
5
6 // Velocidad (en baudios) de las comunicaciones serie
7 #define UART_BAUD_RATE 9600
8
9 #include <avr/io.h>
10 #include <avr/interrupt.h>
11 #include <uart.h>
12 #include <util/delay.h>
13
```

Biblioteca para el puerto UART

Velocidad del procesador a 16MHz

Velocidad del puerto serie: 9600 baud.

Biblioteca para gestión de interrupciones hardware

```
15
16
17 int main(void) {
18
19     // Byte de datos recibido por puerto serie
20     // El byte menos significativo es el dato
21     // El segundo byte menos significativo contiene
22     //   información de error
23     unsigned int c;
24
25
26     // Inicialización del puerto UART con la velocidad
27     //   en baudios del puerto, y la velocidad del procesador
28     uart_init( UART_BAUD_SELECT(UART_BAUD_RATE,F_CPU) );
29
30
31     // Activación de las interrupciones hardware para
32     //   control del puerto serie
33     sei();
34
```

– El programa src/arduecho.cpp: Código fuente (II)

```
3 // Ciclo de reloj del procesador: 16MHz
4 #define F_CPU 16000000UL
5
6 // Velocidad (en baudios) de las comunicaciones serie
7 #define UART_BAUD_RATE 9600
8
9 #include <avr/io.h>
10 #include <avr/interrupt.h>
11 #include <uart.h>
12 #include <util/delay.h>
13
14
15
16
17 int main(void) {
18
19     // Byte de datos recibido por puerto
20     // El byte menos significativo es el
21     // El segundo byte menos significativo contiene
22     //   información de error
23     unsigned int c;
24
25
26     // Inicialización del puerto UART con la velocidad
27     //   en baudios del puerto y la velocidad del procesador
28     uart_init( UART_BAUD_SELECT(UART_BAUD_RATE,F_CPU) );
29
30
31     // Activación de las interrupciones hardware para
32     //   control del puerto s
33     sei();
```

Inicialización del puerto UART en un procesador de 16MHz a una velocidad de 9600 baudios

Inicialización de las interrupciones hardware

- La gestión del puerto UART en procesadores AVR (y en cualquier procesador, por regla general), requiere de una inicialización.
- Esta inicialización implica también indicar cuál es la velocidad (en baudios) a la que se va a trabajar con el puerto (tradicionalmente, la velocidad del puerto serie más estándar es 9600, aunque un puerto USB puede trabajar cómodamente a 115200. Dependiendo de qué plataforma se use, a mayores velocidades se puede obtener errores en las comunicaciones.
 - Además, como el puerto UART es gestionado a través de interrupciones hardware, es necesario activar el módulo de gestión de interrupciones SEI.
- Deberemos realizar estas operaciones sólo una vez al principio del programa, e incluirlas en todos los programas que implementemos y que hagan uso de comunicaciones en serie por USB.

– El programa src/arduecho.cpp: Código fuente (III)

```
34
35
36 while (1) {
37
38     // Recogemos byte desde puerto UART
39     c = uart_getc();
40
41     // Flag sin datos activado: Dormimos
42     if ( c & UART_NO_DATA ) {
43         _delay_ms(1);
44
45         // Error de recepción de la trama de datos. Enviamos mensaje
46     } else if ( c & UART_FRAME_ERROR ) {
47         uart_puts_P("Error recibiendo trama.\n");
48
49         // Error de recepción de dato más rápido de lo que se pudo leer el anterior
50     } else if ( c & UART_OVERRUN_ERROR ) {
51         uart_puts_P("Error: La interrupción no pudo leer dato antes de recibir el siguiente.\n");
52
53
54         // Error de recepción: Buffer lleno. No estamos leyendo tan rápido como nos envían
55     } else if ( c & UART_BUFFER_OVERFLOW ) {
56         uart_puts_P("Error: El buffer está lleno. No se puede leer tan rápido como se envía.\n");
57
58         // Todo ok, respondemos con el mismo carácter
59     } else {
60         uart_putc( (unsigned char)c );
61     }
62 }
63
64 return 0;
65 }
```

– El programa src/arduecho.cpp: Código fuente (III)

```

34
35
36 while (1) {
37
38     // Recogemos byte desde puerto serial
39     c = uart_getc();
40
41     // Flag sin datos activado: Dormimos
42     if ( c & UART_NO_DATA ) {
43         _delay_ms(1);
44
45     // Error de recepción de la trama de datos. Enviamos error
46     } else if ( c & UART_FRAME_ERROR ) {
47         uart_puts_P("Error recibiendo trama.\n");
48
49     // Error de recepción de dato más rápido de lo que se pudo leer el anterior
50     } else if ( c & UART_OVERRUN_ERROR ) {
51         uart_puts_P("Error: La interrupción no pudo leer dato antes de recibir el siguiente.\n");
52
53
54     // Error de recepción: Buffer lleno. No estamos leyendo tan rápido como nos envían
55     } else if ( c & UART_BUFFER_OVERFLOW ) {
56         uart_puts_P("Error: El buffer está lleno. No se puede leer tan rápido como se envía.\n");
57
58     // Todo ok, respondemos con el mismo carácter
59     } else {
60         uart_putc( (unsigned char)c );
61     }
62 }
63
64 return 0;
65 }
```

Devuelve un entero (2 bytes)
 Byte 0: Dato leído por UART
 Byte 1: Control de errores

Envía una cadena de bytes por UART.
 Acabada en \n para enviar los datos del buffer UART

Envía un byte por UART

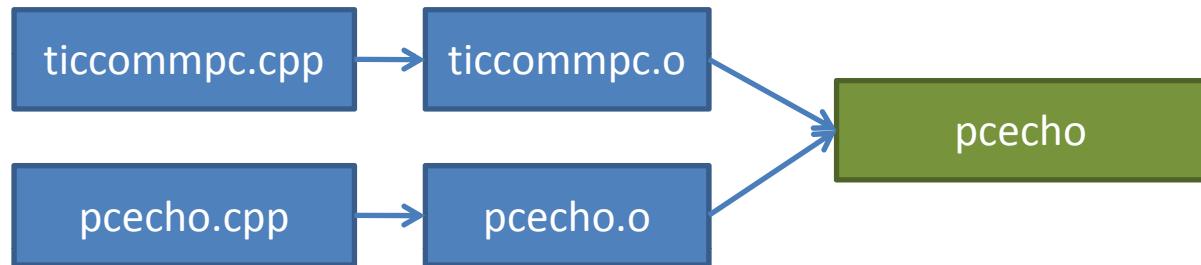
- La biblioteca `<uart.h>` que hemos utilizado proporciona los datos leídos desde el puerto UART en bytes, junto con otro byte de comprobación de errores:

Byte 1 (bits de errores)	Byte 0 (información)
--------------------------	----------------------

- En el byte más significativo encontramos los códigos de errores siguientes:
 - `UART_NO_DATA`: No hay datos en el puerto UART para leer.
 - `UART_FRAME_ERROR`: Error en la recepción del paquete de datos.
 - `UART_OVERRUN_ERROR`: La interrupción hardware está configurada lenta: Se ha recibido un dato antes de leer el que había en el puerto UART.
 - `UART_BUFFER_OVERFLOW`: El buffer del puerto UART está lleno. Nos están enviando datos con mayor velocidad de la que los estamos procesando.

- Con estos datos, ejecute el fichero makefile y envíe el programa:
 - **make arduecho** → Para compilar
 - **make send** → Para enviar el programa a Arduino
- Antes de enviar el programa, asegúrese de:
 1. Que es miembro del grupo **dialout**.
 2. Que el puerto de conexión de Arduino está bien configurado en la variable **COMMPORT** del fichero makefile.

- El programa pcecho: Modelo de compilación
 - Compilaremos según el siguiente esquema:



- Crearemos la biblioteca **ticcommppc** (ficheros src/ticcommppc.cpp e include/ticcommppc.h), para ayudarnos en el resto de las prácticas.
- La biblioteca **ticcommppc** contendrá funciones para gestión de comunicaciones serie entre el PC y Arduino.

- El programa pcecho: Fichero include/ticcommPC.h
 - De momento, contendrá una función para inicializar el puerto USB:

```
1
2 #ifndef __TICCOMMPC__
3
4 #define __TICCOMMPC__
5
6 /**
7  * Función para inicializar el puerto USB
8  *
9  * Entradas: Una cadena de caracteres con el puerto USB (ejemplo: "/dev/ttyACM0")
10 *
11 * Salidas: Un descriptor de fichero si el puerto se abrió correctamente, -1 si error.
12 */
13 int InicializarUSB(const char *portname);
14
15#endif
16
```

– El programa pcecho: Fichero src/ticcommppc.cpp (I)

- La implementación de la función requiere:
 1. Abrir el puerto de comunicaciones.
 2. Configurarlo.
- **Se necesita incluir bibliotecas de gestión y control de E/S**
- No entraremos en detalle en el significado de la configuración del puerto, más allá de lo necesario para la comunicación con Arduino.

```
1 #include <ticcommppc.h>
2 #include <unistd.h>
3 #include <fcntl.h>
4 #include <termios.h>
5
6
7 // Inicializar el puerto USB en el dispositivo dado por argumento
8 int InicializarUSB(const char *portname) {
9
10    int fd; // Descriptor de fichero para abrir el puerto
11    struct termios toptions; // Estructura para control del puerto USB
12
13    /* Abrimos puerto para E/S, sin bloqueo del puerto */
14    fd = open(portname, O_RDWR | O_NOCTTY);
15
16    if (fd== -1)
17        return -1;
18}
```

– El programa pcecho: Fichero src/ticcommppc.cpp (II)

```
19
20 // Cogemos la configuración actual de dispositivos /dev/tty
21 tcgetattr(fd, &options);
22
23
24 /* Personalizamos opciones */
25 cfsetispeed(&options, B9600); // Velocidad (baudios) de entrada de datos
26 cfsetospeed(&options, B9600); // Velocidad (baudios) de salida de datos
27
28 // Se envian 8 bits cada vez, sin bits de error de paridad ni bits de parada
29 toptions.c_cflag &= ~PARENB;
30 toptions.c_cflag &= ~CSTOPB;
31 toptions.c_cflag &= ~CSIZE;
32 toptions.c_cflag |= CS8;
33
34 // Sin control del flujo de datos por hardware
35 toptions.c_cflag &= ~CRTSCTS;
36
37 // Habilitamos para lectura y sin control de estado
38 toptions.c_cflag |= CREAD | CLOCAL;
39
40 // Deshabilitamos control E/S y caracteres de reinicio del puerto
41 toptions.c_iflag &= ~(IXON | IXOFF | IXANY);
42
43 // Deshabilitamos entradas canónicas, echo, caracteres de borrado del línea
44 // y señales del terminal
45 toptions.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
46
47 // Deshabilitamos post-procesamiento de salidas
48 toptions.c_oflag &= ~OPOST;
49
50 // Hacemos que devolvamos cada byte leido según llega (tamaño del buffer de llegada)
51 toptions.c_cc[VMIN] = 1;
52
53 // Sin espera desde que se reciben datos por puerto hasta que los cogemos
54 toptions.c_cc[VTIME] = 0;
55
56 // Establecemos las opciones del puerto
57 tcsetattr(fd, TCSANOW, &options);
```

– El programa pcecho: Fichero src/ticcommppc.cpp (II)

```
19
20 // Cogemos la configuración actual de dispositivos /dev/tty
21 tcgetattr(fd, &options);
22
23
24 /* Personalizamos opciones */
25 cfsetispeed(&options, B9600) // Velocidad
26 cfsetspeed(&options, B9600) // Velocidad
27
28 // Se envian 8 bits cada vez, sin bits de error de paridad ni bits de parada
29 toptions.c_cflag &= ~PARENB;
30 toptions.c_cflag &= ~CSTOPB;
31 toptions.c_cflag &= ~CSIZE;
32 toptions.c_cflag |= CS8;
33
34 // Sin control del flujo de datos por hardware
35 toptions.c_cflag &= ~CRTSCTS;
36
37 // Habilitamos para lectura y sin control de estado
38 toptions.c_cflag |= CREAD | CLOCAL;
39
40 // Deshabilitamos control E/S y caracteres de reinicio del puerto
41 toptions.c_iflag &= ~(IXON | IXOFF | IXANY);
42
43 // Deshabilitamos entradas canónicas, echo, caracteres de borrado del línea
44 // y señales del terminal
45 toptions.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
46
47 // Deshabilitamos post-procesamiento de salidas
48 toptions.c_oflag &= ~OPOST;
49
50 // Hacemos que devolvamos cada byte leido según llega (tamaño del buffer de llegada)
51 toptions.c_cc[VMIN] = 1;
52
53 // Sin espera desde que se reciben datos por puerto hasta que los cogemos
54 toptions.c_cc[VTIME] = 0;
55
56 // Establecemos las opciones del puerto
57 tcsetattr(fd, TCSANOW, &options);
```

¡OJO!: Los baudios incluidos aquí deben coincidir con la configuración en Arduino.

– El programa pcecho: Fichero src/ticcommppc.cpp (III)

```
58
59 // Esperamos inicialización
60 usleep(2000000);
61
62 // Envío de la basura que pudiese haber en el buffer
63 tcflush(fd, TCIFLUSH);
64
65 return fd;
66 }
```

– El programa pcecho: Fichero src/ticcommppc.cpp (III)

```
58  
59 // Esperamos la inicialización  
60 usleep(2000000);  
61  
62 // Envío de la basura que pu-  
63 tcflush(fd, TCIFLUSH);  
64  
65 return fd;  
66 }
```

¡IMPORTANTE!: Esperar un tiempo para que la configuración surta efecto (sobre todo en el extremo de las comunicaciones de Arduino).

– El programa pcecho: Fichero src/ticcommppc.cpp (III)

```
58  
59 // Esperamos inicialización  
60 usleep(1000000);  
61  
62  
63 // Envío de la basura que pudiese haber en el buffer  
64 tcflush(fd, TCIFLUSH);  
65  
66 return fd;  
67 }  
68
```

¡IMPORTANTE!: Vaciar el buffer antes de las comunicaciones.

Con la biblioteca implementada, ya podemos pasar al programa principal.

- **El programa pcecho: Fichero src/pcecho.cpp**
 - El programa principal realizará las siguientes operaciones:
 1. Abrir y configurar el puerto de comunicaciones USB.
 2. Pedir datos por entrada estándar y enviarlos a Arduino.
 3. Esperar a que Arduino responda con datos, y mostrarlos por pantalla.
 4. Realizar los pasos 2 y 3 hasta que no se introduzcan datos.
 5. **Cerrar el puerto de comunicaciones USB** y salir.

– El programa pcecho: Fichero src/pcecho.cpp (I)

- Inclusión de bibliotecas, definición de variables y abrir puerto:

```
1 #include <cstring>
2 #include <ticcomm.h>
3 #include <iostream>
4 #include <unistd.h>
5 #include <termios.h>
6
7 using namespace std;
8
9 // Puerto de comunicaciones con arduino
10 #define USBPORT "/dev/ttyACM0"
11
12
13
14 int main(int argc, char *argv[]) {
15
16     int fd; // Descriptor de fichero del puerto USB
17     char buf[128]; // Buffer de salida. Tamaño máximo de 128 caracteres
18     int ndata; // Número de datos en el buffer
19     int aux; // Variable auxiliar
20
21
22     // Inicializamos puerto
23     fd= InicializarUSB(USBPORT);
24     if (fd == -1) {
25         cout<<"Error inicializando puerto "<<USBPORT<<endl;
26         return 0;
27     }
28 }
```

– El programa pcecho: Fichero src/pcecho.cpp (II)

– Bucle principal:

```
29
30 // Bucle hasta que salgamos (cuando no introduzcamos nada)
31 do {
32     cout<<"Escriba algo para enviar a Arduino: ";
33     cin.getline(buf, 128);
34
35     ndata= strlen(buf); // Tamaño del buffer de salida
36
37     if (ndata>0) { // Enviamos si hay algo
38         aux= write(fd, buf, ndata);
39         tcflush(fd, TCIFLUSH);
40         cout << "Bytes enviados "<<aux<<"/"<<ndata<<endl;
41
42         // Comprobamos errores en el envío
43         if (aux < ndata) {
44             cout << "Error, el buffer tiene "<<ndata<<" bytes pero se enviaron "<<aux<<endl;
45             ndata= 0;
46         }
47
48         // Si se envió algún dato, esperamos respuesta de Arduino
49         if (aux>0) {
50             aux= read(fd, buf, 128);
51             if (aux > 0)
52                 cout<<"\tMensaje: "<< buf<<endl<<endl;
53             else
54                 cout<<"\tERROR RECIBIENDO DATOS\n";
55         }
56
57     }
58
59     // Se sale si no enviamos ningún caracter
60 } while (ndata > 0);
```

- El programa pcecho: Fichero src/pcecho.cpp (III)
 - Fin del programa:

```
61
62
63 // Cerramos el puerto USB
64 close(fd);
65 cout <<"\nFin del programa.\n\n";
66 return 0;
67 }
```

- Modificaremos el fichero makefile para incluir la compilación del programa pcecho.

– El programa pcecho: Sección del fichero makefile

```
25
26 pcecho: pcechocpp ticcommppcpp
27     @echo Generando fichero binario pcecho...
28     g++ -o bin/pcecho obj/pcecho.o obj/ticcommpc.o
29
30 pcechocpp:
31     @echo Compilando PCECHO...
32     g++ -c -o obj/pcecho.o src/pcecho.cpp -Iinclude
33
34
35 ticcommppcpp:
36     @echo Compilando fuentes de biblioteca TICCommPC...
37     g++ -c -o obj/ticcommpc.o src/ticcommpc.cpp -Iinclude
```

— El proyecto: makefile básico completo.

```
1
2 # Puerto de comunicaciones donde se encuentra Arduino.
3 COMMPORT = /dev/ttyACM0
4
5 arduecho: arduechocpp uartcpp
6     @echo Generando binario Arduino...
7     avr-gcc -mmcu=atmega328p obj/arduecho.o obj/uart.o -o obj/arduecho.bin
8     @echo Generando HEX Arduino...
9     avr-objcopy -O ihex -R .eeprom obj/arduecho.bin hex/arduecho.hex
10    @echo Arduecho compilado. Ejecute make send para enviarlo a la plataforma Arduino.
11
12 arduechocpp:
13     @echo Compilando arduecho...
14     avr-gcc -Os -mmcu=atmega328p -c -o obj/arduecho.o src/arduecho.cpp -Iinclude
15
16 uartcpp:
17     @echo Compilando biblioteca UART...
18     avr-gcc -Os -mmcu=atmega328p -c -o obj/uart.o src/uart.cpp -Iinclude
19
20
21 send:
22     @echo Enviando arduecho a Arduino
23     sudo avrdude -F -V -c arduino -p ATMEGA328P -P $(COMMPORT) -b 115200 -U flash:w:hex/arduecho.hex
24
25
26 pcecho: pcechocpp ticcommppc.cpp
27     @echo Generando fichero binario pcecho...
28     g++ -o bin/pcecho obj/pcecho.o obj/ticcommpc.o
29
30 pcechocpp:
31     @echo Compilando PCECHO...
32     g++ -c -o obj/pcecho.o src/pcecho.cpp -Iinclude
33
34
35 ticcommppc.cpp:
36     @echo Compilando fuentes de biblioteca TICCommPC...
37     g++ -c -o obj/ticcommpc.o src/ticcommpc.cpp -Iinclude
38 |
```

- El proyecto: makefile básico completo.
 - Para compilar el programa arduecho: **make arduecho**
 - Para compilar el programa pcecho: **make pcecho**
 - Para enviar el programa **arduecho** a la plataforma Arduino Uno:
make send
- Por comodidad y limpieza, se podrían añadir nuevos objetivos al makefile, tales como:
 - **all**: Compila pcecho, arduecho y lo envía a Arduino Uno
 - **clean**: Limpia las carpetas bin, obj y hex.

SESIÓN 3

- 1. ¿Qué es Arduino?**
- 2. El procesador**
- 3. Compilador y bibliotecas para procesadores AVR**
- 4. Mi primer programa para AVR AtMega328p**
- 5. Comunicaciones en serie**
- 6. Mi primer programa para comunicaciones en serie: ECHO**
- 7. Recepción de datos desde sensores fotorresistores digitales**
- 8. Envío de datos por dispositivos emisores láser**
- 9. La arquitectura del sistema de emisión/recepción de datos por láser**
- 10. Ampliación de la biblioteca <ticcommardu.h>**
- 11. Ejemplo: Envío de datos por láser**
- 12. Ejemplo: Recepción de datos por láser**
- 13. Proyecto final de la práctica**

- Recordemos que en las prácticas hemos estado construyendo 2 bibliotecas:
 - **<ticcommmpc.h>** para gestionar las comunicaciones en el extremo del PC, a través de USB, con una plataforma Arduino Uno.
 - **<ticcommardu.h>** para gestionar las comunicaciones en el extremo de Arduino, a través de USB, con un PC.

– Aspecto de <ticcommc.h> :

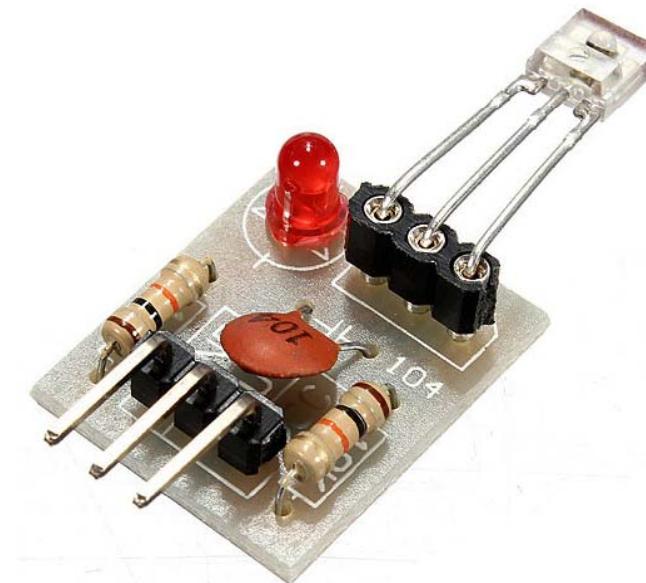
```
2 #ifndef __TICCOMMPC__
3
4 #define __TICCOMMPC__
5
6 /**
7 * Función para inicializar el puerto USB
8 *
9 * Entradas: Una cadena de caracteres con el puerto USB (ejemplo: "/dev/ttyACM0")
10 *
11 * Salidas: Un descriptor de fichero si el puerto se abrió correctamente, -1 si error.
12 */
13 int InicializarUSB(const char *portname);
14
15
16
17
18 /**
19 * Función para enviar un mensaje por USB a Arduino
20 *
21 * Entradas:
22 * fd: Descriptor de fichero asociado al puerto
23 * data: Cadena de datos a enviar
24 *
25 * Salidas: true si los datos se enviaron con éxito, false en otro caso
26 */
27 bool sendUSB(int &fd, const char *data);
28
29
30
31
32 /**
33 * Función para recibir un mensaje por USB desde Arduino
34 *
35 * Entradas:
36 * fd: Descriptor de fichero asociado al puerto
37 *
38 * Salidas: true si los datos se enviaron con éxito, false en otro caso
39 * en el parámetro data: Cadena de caracteres recibida
40 */
41 bool receiveUSB (int &fd, char *data);
42
43#endif
```

- Aspecto de **<ticcommardu.h>** :

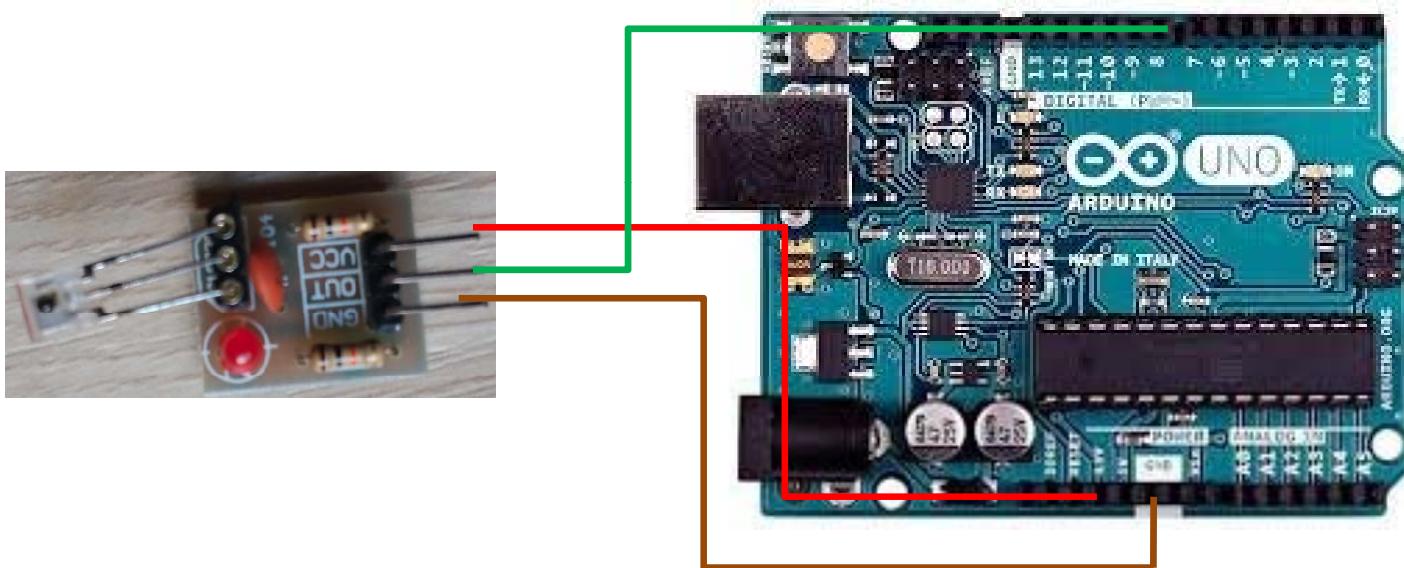
```
2 #ifndef __TICCOMMARDU__
3
4 #define __TICCOMMARDU__
5
6
7
8 /**
9 * Función para enviar un mensaje por USB a PC
10 *
11 * Entradas:
12 *   data: Cadena de datos a enviar.
13 *
14 * Salidas: true si los datos se enviaron con éxito, false en otro caso
15 */
16 bool arduSendUSB(const char *data);
17
18
19
20
21 /**
22 * Función para recibir un mensaje por USB desde PC
23 *
24 * Entradas: Ninguna
25 *
26 * Salidas: true si los datos se enviaron con éxito, false en otro caso
27 *   en el parámetro data: Cadena de caracteres recibida
28 */
29 bool arduReceiveUSB (char *data);
30
31#endif
```

¡No se puede continuar si no se dispone de estas bibliotecas!

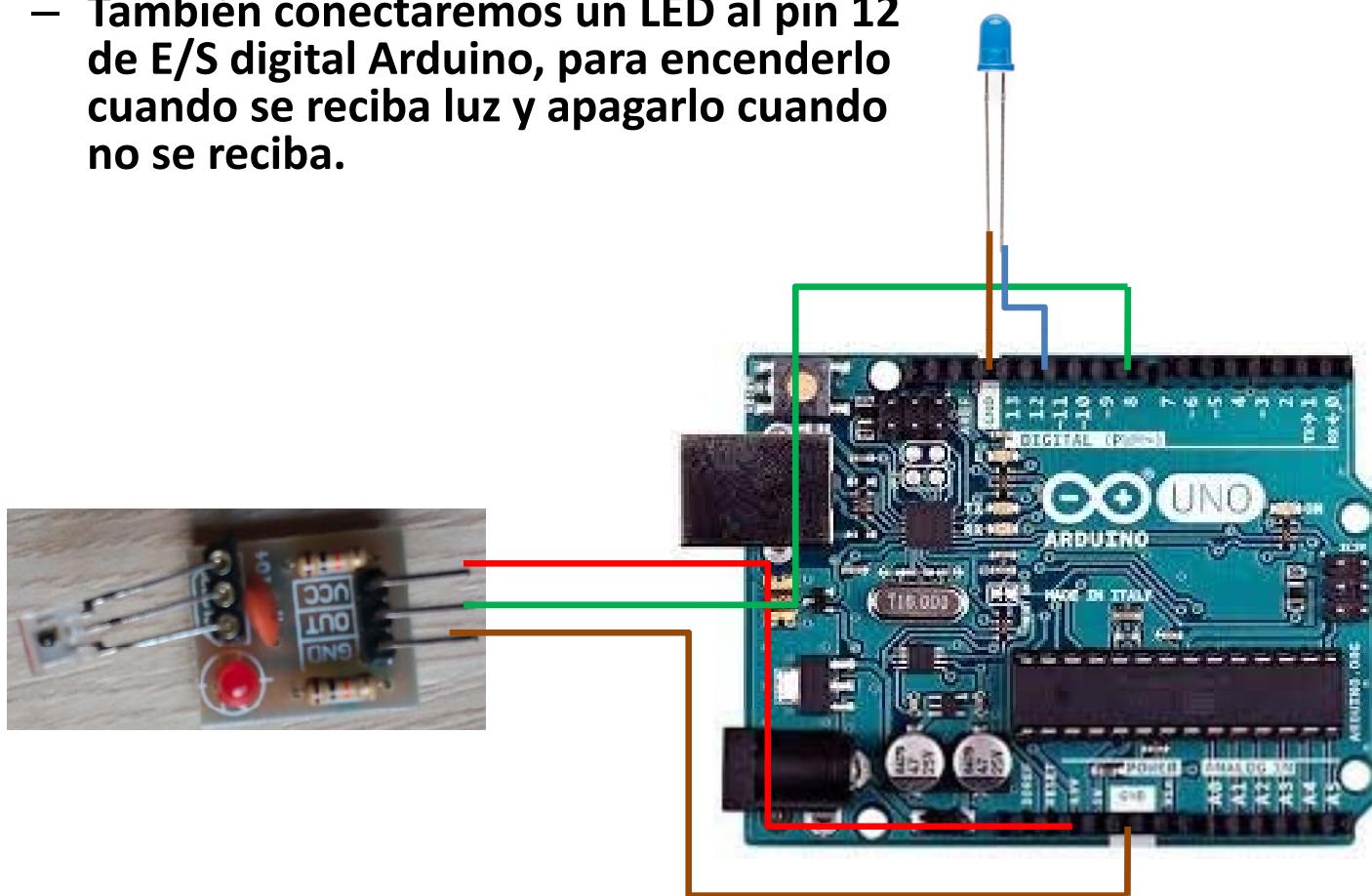
- Para construir los prototipos de emisión/recepción de información mediante láser, utilizaremos **fotorreceptores** capaces de captar la luz emitida mediante láser.
- El **Módulo receptor 2PCS** láser no modulador sensor Tubo es un fotorresistor que responde a la luz ambiental existente.
- Es un sensor que se conecta al puerto digital.
- Funciona a 3.3V.



- **Módulo receptor 2PCS láser no modulador sensor Tubo. Conexión con la placa:**
 - Utilizando cables de conexión de pines Macho-Hembra, conectaremos la patilla GND del sensor a tierra en la placa Arduino, la patilla VCC a la alimentación de 3.3V de la placa, y la salida del sensor VOUT al pin 8 de E/S digital:



- También conectaremos un LED al pin 12 de E/S digital Arduino, para encenderlo cuando se reciba luz y apagarlo cuando no se reciba.



- El programa **fotorreceptor** de Arduino Uno:
 - Debe configurar como entrada el Pin 8 de la placa.
 - Debe configurar como salida el Pin 12 de la placa.
 - En un bucle infinito:
 - Debe leer el Pin 8.
 - Si el Pin 8 > 0 (hay luz), entonces
 - Encender el LED
 - En otro caso,
 - Apagar el LED

– El programa **fotorreceptor (I)**:

```
1 #define F_CPU 16000000UL
2
3
4 #include <avr/io.h>
5 #include <util/delay.h>
6
7 // Tiempo a dormir en microsegundos
8 #define DELAY_US 500
9
10 int main (void) {
11
12     unsigned char dato; // dato a leer desde el sensor
13     bool currentLEDstate= false; // Estado actual del LED: encendido o apagado
14
15     // Ponemos el PIN 12 de la placa como salida
16     DDRB |= _BV(DDB4);
17
18     // Ponemos el PIN 8 de la placa como entrada
19     DDRB &= ~_BV(DDB0);
20
21
22     // Comenzamos con el LED apagado
23     PORTB &= ~_BV(PORTB4);
24     currentLEDstate= false;
25 }
```

- El programa **fotorreceptor (I)**:

```
1 #define F_CPU 16000000UL
2
3
4 #include <avr/io.h>
5 #include <util/delay.h>
6
7 // Tiempo a dormir en microsegundos
8 #define DELAY_US 500
9
10 int main (void) {
11
12     unsigned char dato; // dato a leer desde el sensor
13     bool currentLEDstate= false; // Estado actual del LED: encendido o apagado
14
15     // Ponemos el PIN 12 de la placa como salida
16     DDRB |= _BV(DDB4);
17
18     // Ponemos el PIN 8 de la placa como entrada
19     DDRB &= ~_BV(DDB0);
20
21
22     // Comenzamos con el LED apagado
23     PORTB &= ~_BV(PORTB4);
24     currentLEDstate= false;
```

Usaremos la variable para saber si el sensor recibe luz o no

- El programa **fotorreceptor (I)**:

```
1 #define F_CPU 16000000UL
2
3
4 #include <avr/io.h>
5 #include <util/delay.h>
6
7 // Tiempo a dormir en microsegundos
8 #define DELAY_US 500
9
10 int main (void) {
11
12     unsigned char dato; // dato a leer desde el sensor
13     bool currentLEDstate= false; // Estado actual del LED: encendido o apagado
14
15     // Ponemos el PIN 12 de la placa como salida
16     DDRB |= _BV(DDB4);
17
18     // Ponemos el PIN 8 de la placa como entrada
19     DDRB &= ~_BV(DDB0);
20
21
22     // Comenzamos con el LED apagado
23     PORTB &= ~_BV(PORTB4);
24     currentLEDstate= false;
```

Usaremos la variable para guardar el estado del LED (encendido=true)

– El programa fotorreceptor (I):

```
1 #define F_CPU 16000000UL
2
3
4 #include <avr/io.h>
5 #include <util/delay.h>
6
7 // Tiempo a dormir en microsegundos
8 #define DELAY_US 500
9
10 int main (void) {
11
12     unsigned char dato; // dato a leer desde el sensor
13     bool currentLEDstate= false; // Estado actual del LED: encendido o apagado
14
15     // Ponemos el PIN 12 de la placa como salida
16     DDRB |= _BV(DDB4); // Linea roja
17
18     // Ponemos el PIN 8 de la placa como entrada
19     DDRB &= ~_BV(DDB0);
20
21
22     // Comenzamos con el LED apagado
23     PORTB &= ~_BV(PORTB4);
24     currentLEDstate= false;
25 }
```

Pin PB4 del microprocesador como salida

- El programa **fotorreceptor (I)**:

```
1 #define F_CPU 16000000UL
2
3
4 #include <avr/io.h>
5 #include <util/delay.h>
6
7 // Tiempo a dormir en microsegundos
8 #define DELAY_US 500
9
10 int main (void) {
11
12     unsigned char dato; // dato a leer desde el sensor
13     bool currentLEDstate= false; // Estado actual del LED: encendido o apagado
14
15     // Ponemos el PIN 12 de la placa como salida
16     DDRB |= _BV(DDB4);
17
18     // Ponemos el PIN 8 de la placa como entrada
19     DDRB &= ~_BV(DDB0);
```

Pin PB0 del microprocesador como
entrada

```
20
21
22     // Comenzamos con el LED apagado
23     PORTB &= ~_BV(PORTB4);
24     currentLEDstate= false;
```

- El programa **fotorreceptor (I)**:

```
1 #define F_CPU 16000000UL
2
3
4 #include <avr/io.h>
5 #include <util/delay.h>
6
7 // Tiempo a dormir en microsegundos
8 #define DELAY_US 500
9
10 int main (void) {
11
12     unsigned char dato; // dato a leer desde el sensor
13     bool currentLEDstate= false; // Estado actual del LED: encendido o apagado
14
15     // Ponemos el PIN 12 de la placa como salida
16     DDRB |= _BV(DDB4);
17
18     // Ponemos el PIN 8 de la placa como entrada
19     DDRB &= ~_BV(DDB0);
20
21
22     // Comenzamos con el LED apagado
23     PORTB &= ~_BV(PORTB4);
24     currentLEDstate= false;
```

Apagamos el LED y ponemos su estado a false (apagado)

- El programa **fotorreceptor (II)**: El bucle principal

```
25
26 while(1) {
27
28     // Cogemos qué hay en el PIN 8 como entrada
29     dato = PINB & 0x01;
30
31     if (dato>0 && !currentLEDstate) { // Si hay luz, encendemos la LED
32         PORTB |= _BV(PORTB4);
33         currentLEDstate= true;
34     }
35
36     else if (dato==0 && currentLEDstate) { // en otro caso, lo apagamos
37         PORTB &= ~_BV(PORTB4);
38         currentLEDstate= false;
39     }
40
41     // Dormimos medio milisegundo
42     _delay_us(DELAY_US);
43 }
44
45 return 0;
46 }
```

Cogemos el bit correspondiente a PBO.

- El programa **fotorreceptor (II)**: El bucle principal

```
25
26 while(1) {
27
28     // Cogemos qué hay en el PIN 8 como entrada
29     dato = PINB & 0x01;
30
31     if (dato>0 && !currentLEDstate) { // 
32         PORTB |= _BV(PORTB4);
33         currentLEDstate= true;
34     }
35
36     else if (dato==0 && currentLEDstate) { // en otro caso, lo apagamos
37         PORTB &= ~_BV(PORTB4);
38         currentLEDstate= false;
39     }
40
41     // Dormimos medio milisegundo
42     _delay_us(DELAY_US);
43 }
44
45 return 0;
46 }
```

¡OJO!: PORTB se utiliza para SALIDAS

PINB se utiliza para ENTRADAS

- El programa **fotorreceptor (II)**: El bucle principal

```
25
26 while(1) {
27
28     // Cogemos qué hay en el PIN 8 como entrada
29     dato = PINB & 0x01;
30
31     if (dato>0 && !currentLEDstate) { // 
32         PORTB |= _BV(PORTB4);
33         currentLEDstate= true;
34     }
35
36     else if (dato==0 && currentLEDstate)
37         PORTB &= ~_BV(PORTB4);
38         currentLEDstate= false;
39     }
40
41     // Dormimos medio milisegundo
42     _delay_us(DELAY_US);
43 }
44
45 return 0;
46 }
```

Enviamos voltaje alto al pin PB4 del microprocesador cuando el LED estaba apagado y se recibe luz por el fotorresistor.

También cambiamos su estado.

- El programa **fotorreceptor (II)**: El bucle principal

```
25
26 while(1) {
27
28     // Cogemos qué hay en el PIN 8 como entrada
29     dato = PINB & 0x01;
30
31     if (dato>0 && !currentLEDstate) { // 
32         PORTB |= _BV(PORTB4);
33         currentLEDstate= true;
34     }
35
36     else if (dato==0 && currentLEDstate)
37         PORTB &= ~_BV(PORTB4);
38         currentLEDstate= false;
39     }
40
41     // Dormimos medio milisegundo
42     _delay_us(DELAY_US);
43 }
44
45 return 0;
46 }
```

Enviamos voltaje bajo al pin PB4 del microprocesador cuando el LED estaba encendido y no se recibe luz por el fotorresistor.
También cambiamos su estado.

- El programa **fotorreceptor (II)**: El bucle principal

```
25
26 while(1) {
27
28     // Cogemos qué hay en el PIN 8 como entrada
29     dato = PINB & 0x01;
30
31     if (dato>0 && !currentLEDstate) { // 
32         PORTB |= _BV(PORTB4);
33         currentLEDstate= true;
34     }
35
36     else if (dato==0 && currentLEDstate)
37         PORTB &= ~_BV(PORTB4);
38         currentLEDstate= false;
39     }
40
41     // Dormimos medio milisegundo
42     _delay_us(DELAY_US);
43 }
44
45 return 0;
46 }
```

Nueva función. Funciona igual que la ya conocida `_delay_ms`.

En este caso, duerme microsegundos.

- El makefile simple del programa **fotorreceptor**:

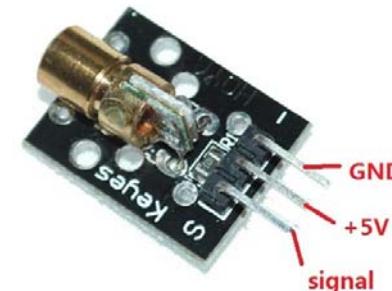
```
2
3 fotorreceptor:
4     @echo Compilando fotorreceptor...
5     @avr-gcc -Os -mmcu=atmega328p -c -o fotorreceptor.o fotorreceptor.cpp
6     @avr-gcc -mmcu=atmega328p fotorreceptor.o -o fotorreceptor.bin
7     @avr-objcopy -O ihex -R .eeprom fotorreceptor.bin fotorreceptor.hex
8     @echo Programa fotorreceptor compilado. Ejecute make sendfotorreceptor para enviarlo a Arduino.
9
10 sendfotorreceptor:
11     sudo avrdude -F -V -c arduino -p ATMEGA328P -P /dev/ttyACM0 -b 115200 -U flash:w:fotorreceptor.hex
12
13 clean:
14     rm -f -r *~
15     rm -f -r *.o
16     rm -f -r *.hex
17     rm -f -r *.bin|
```

- 1. ¿Qué es Arduino?**
- 2. El procesador**
- 3. Compilador y bibliotecas para procesadores AVR**
- 4. Mi primer programa para AVR AtMega328p**
- 5. Comunicaciones en serie**
- 6. Mi primer programa para comunicaciones en serie: ECHO**
- 7. Recepción de datos desde sensores fotorresistores digitales**
- 8. Envío de datos por dispositivos emisores láser**
- 9. La arquitectura del sistema de emisión/recepción de datos por láser**
- 10. Ampliación de la biblioteca <ticcommardu.h>**
- 11. Ejemplo: Envío de datos por láser**
- 12. Ejemplo: Recepción de datos por láser**
- 13. Proyecto final de la práctica**

- **IMPORTANTE:** El emisor láser que se utiliza en las prácticas tiene una longitud de onda relativamente alta (650nm), por lo que no produce daños erosivos ni quemaduras. Sin embargo, **el láser no se debe orientar directamente a los ojos, pues estos sí pueden ser dañados** ante un estímulo de estas características.

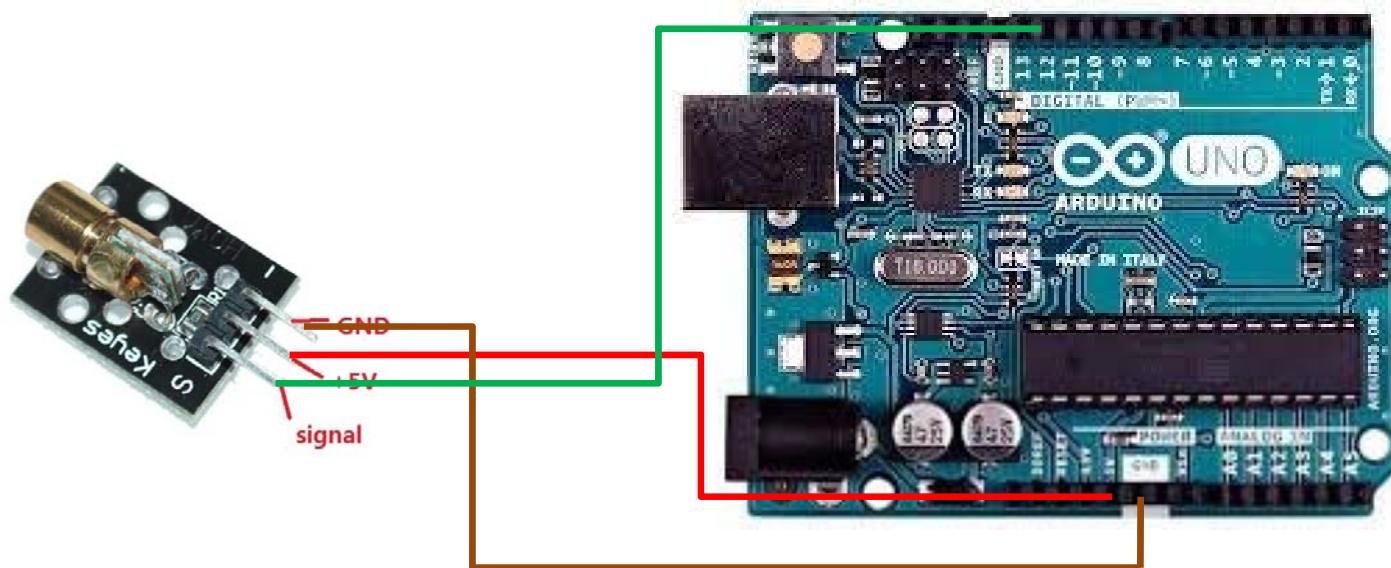


- Para construir los prototipos de emisión/recepción de información mediante láser, utilizaremos **emisores láser** capaces de emitir luz en el rango visible con una longitud de onda de 650nm.
- El **Módulo emisor láser KY-008** es un emisor adecuado a nuestras necesidades.
- Es un emisor que se conecta al puerto digital.
- Funciona a 5V.

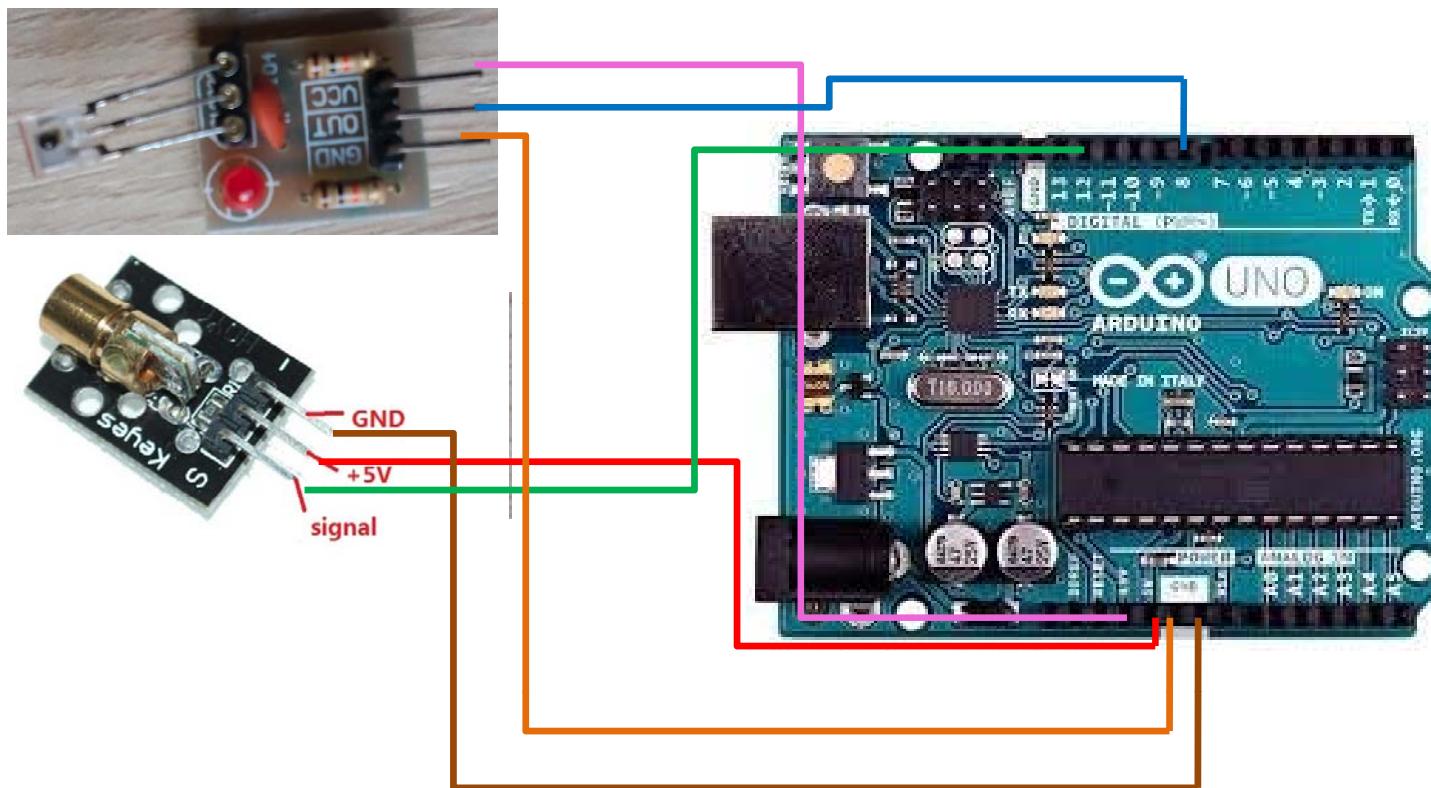


— Módulo emisor láser KY-008. Conexión con la placa:

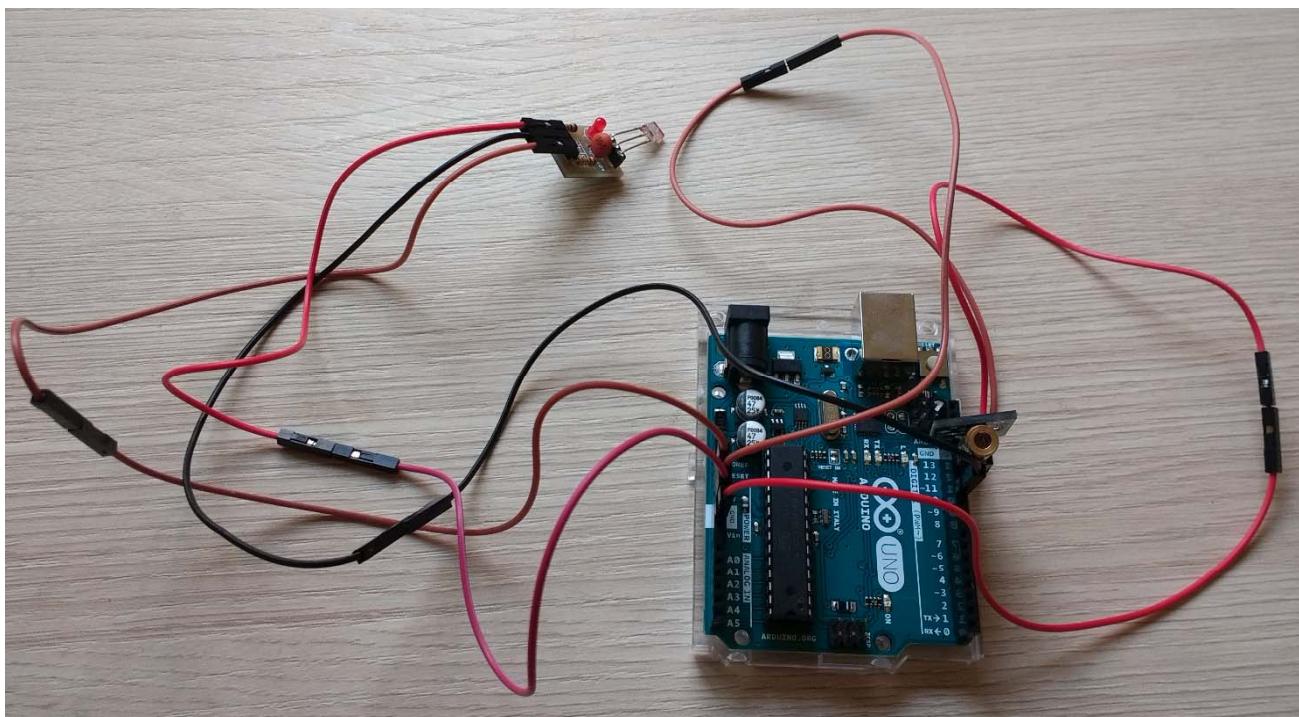
- Utilizando cables de conexión de pines Macho-Hembra, conectaremos la patilla GND del sensor a tierra en la placa Arduino, la patilla VCC a la alimentación de 5V de la placa, y la entrada del emisor SIGNAL al pin 12 de E/S digital:



- **Módulos emisor láser KY-008 y receptor 2PCS** láser no modulador sensor Tubo en la misma plataforma. **Conexión con la placa:**
 - Utilizaremos la misma configuración utilizada previamente para el ejemplo del fotorresistor:



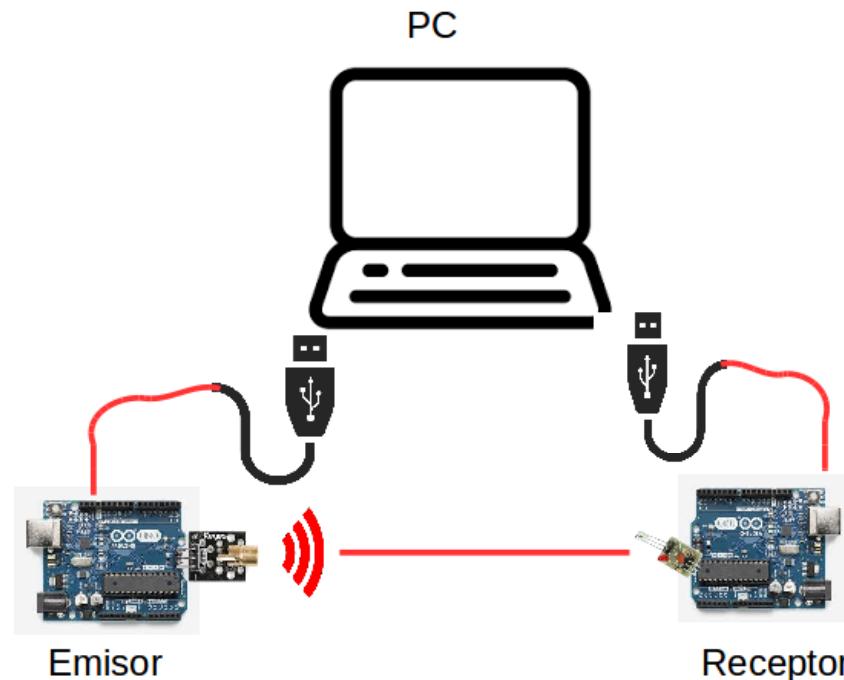
- Utilice el mismo programa **fotorreceptor.cpp** para comprobar que se emite luz láser cuando el fotorreceptor detecta iluminación, y que no se emite en caso contrario.
- **¡CUIDADO! NO ENFOCAR EL LÁSER DIRECTAMENTE A LOS OJOS**
- Ejemplo de interconexión en la placa:



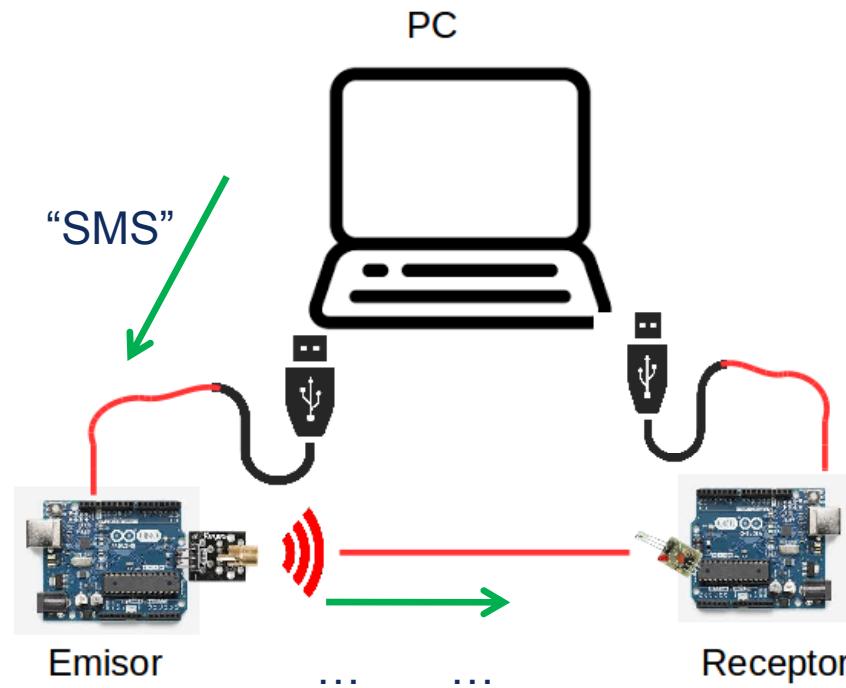
SESIÓN 4

- 1. ¿Qué es Arduino?**
- 2. El procesador**
- 3. Compilador y bibliotecas para procesadores AVR**
- 4. Mi primer programa para AVR AtMega328p**
- 5. Comunicaciones en serie**
- 6. Mi primer programa para comunicaciones en serie: ECHO**
- 7. Recepción de datos desde sensores fotorresistores digitales**
- 8. Envío de datos por dispositivos emisores láser**
- 9. La arquitectura del sistema de emisión/recepción de datos por láser**
- 10. Ampliación de la biblioteca <ticcommardu.h>**
- 11. Ejemplo: Envío de datos por láser**
- 12. Ejemplo: Recepción de datos por láser**
- 13. Proyecto final de la práctica**

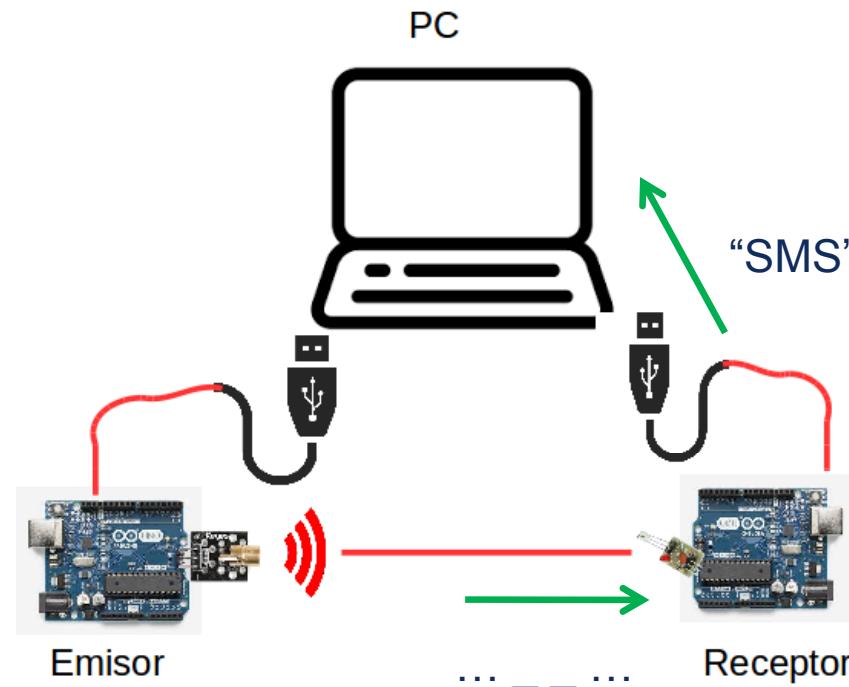
- **Utilizaremos 2 placas Arduino Uno:** Una como emisor, otra como receptor.
- Tanto el emisor como el receptor tendrán comunicaciones por USB con un PC (puede ser el mismo o diferente).



- Mediante el puerto USB, el PC del emisor enviará cadenas de datos al **Arduino emisor**.
- Este las codificará en señales luminosas, que serán transmitidas por el láser.



- **El Arduino receptor**, a través del fotorresistor, recibirá ráfagas luminosas desde el emisor láser, que interpretará y decodificará, transformándolas a símbolos ASCII.
- **Mediante el puerto USB**, el **Arduino receptor** enviará cadenas de caracteres con los símbolos ASCII decodificados del mensaje recibido.



- Tendremos **3 tipos de “símbolos” o “mensajes”:**
 - **Ráfagas cortas:** Asociadas con el símbolo binario 0 (punto).
 - **Ráfagas largas:** Asociadas con el símbolo binario 1 (raya).
 - **Sin emisión:** Fin de transmisión del símbolo.

“SMS” → ... _ _ ...

- **Cadena “SMS”:**
 - **“S”:** 3 ráfagas cortas, fin de transmisión
 - **“M”:** 2 ráfagas largas, fin de transmisión
 - **“S”:** 3 ráfagas cortas, fin de transmisión
 - **Fin del envío:** fin de transmisión

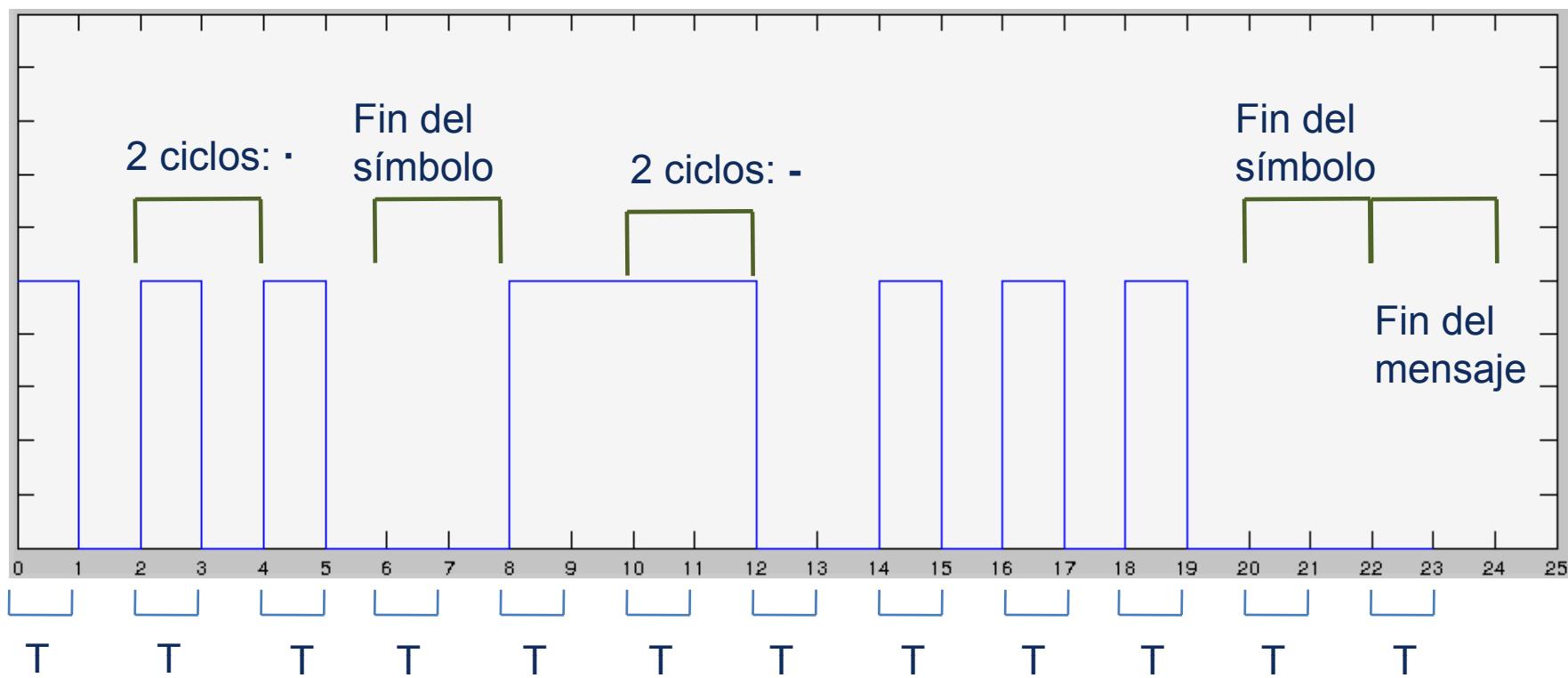
Consecuencia: Si encontramos dos mensajes *fin de transmisión consecutivos*, se ha terminado la comunicación desde el emisor al receptor.

Solución: Alternativa 1

- **Ciclo de reloj de la aplicación en Arduino:**
 - Para poder diferenciar entre ráfagas cortas y largas, será necesario establecer el tiempo mínimo que puede durar una ráfaga.
 - A este tiempo mínimo le llamaremos **ciclo de reloj de la aplicación**.
- **Así, podremos siempre decir que la transmisión de cualquier símbolo “0” o “1” durará 2 ciclos de reloj de la aplicación:**
- **Ráfaga corta:**
 - En el primer ciclo de reloj, se envía señal luminosa.
 - En el segundo, no se envía señal luminosa.
- **Ráfaga larga:**
 - En el primer ciclo de reloj, se envía señal luminosa.
 - En el segundo, también.
- **Fin de mensaje:**
 - No se envía señal luminosa en ninguno de los dos ciclos.

Solución: Alternativa 1

- Asumiendo que el ciclo de reloj de la aplicación es T , el siguiente esquema muestra cómo se codificaría el mensaje “... - - ...” en términos de “voltaje alto” o “voltaje bajo” de pines de la placa Arduino:



Solución: Alternativa 1

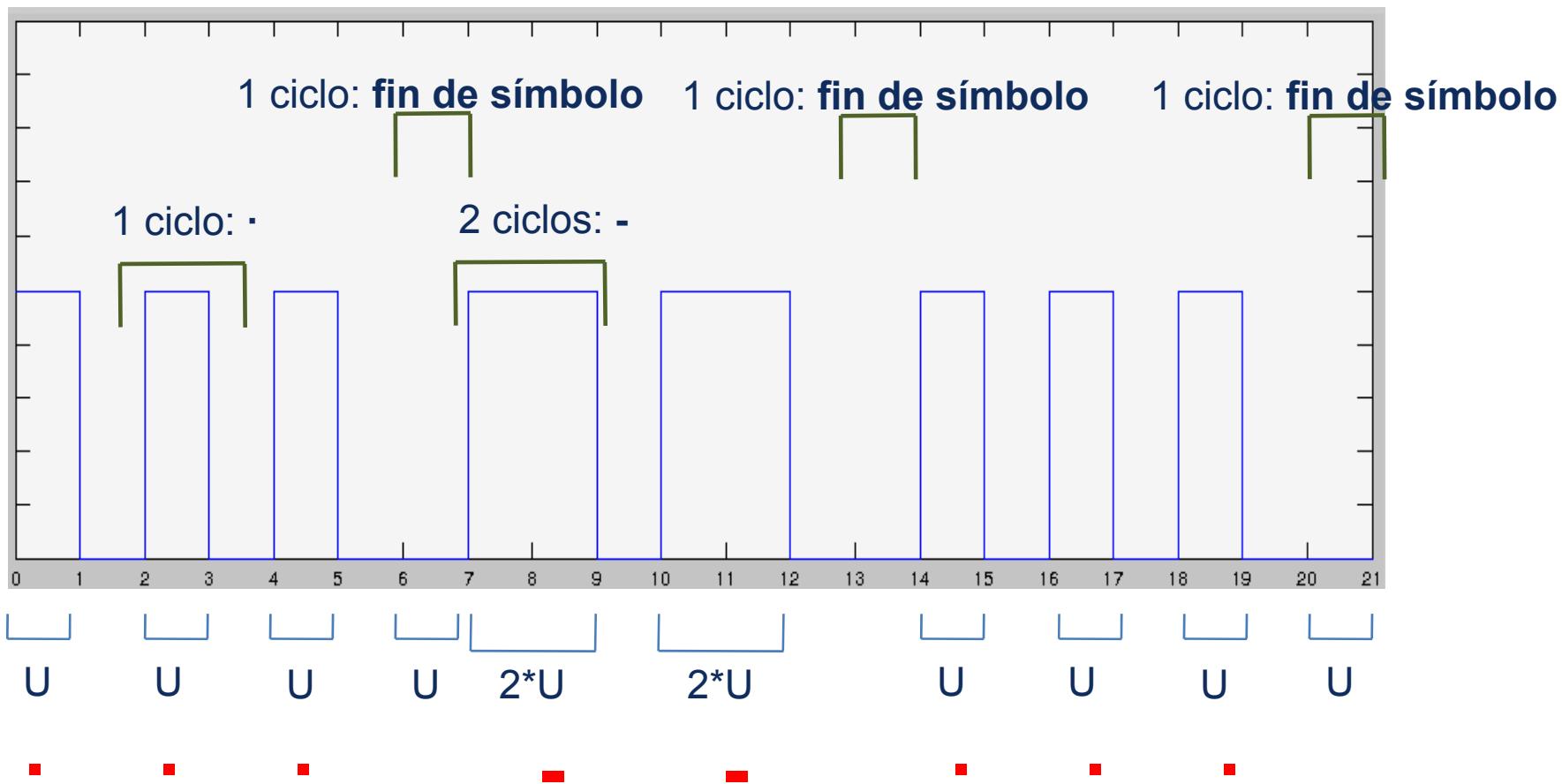
- Problemas de esta alternativa:
- ¿Cuánto debe durar T?
 - A mayor valor, menor velocidad.
 - Se deberá ajustar de acuerdo a la plataforma Arduino y a su velocidad de CPU y de comunicaciones por BUS.
- Los relojes internos de la aplicación Arduino emisor y receptor deben estar coordinados para que se envíe la información sin errores.
- ¿Qué ocurriría si un reloj (por ejemplo, el receptor) está trasladado δT ciclos del procesador con respecto al otro?

Solución: Alternativa 2

- Utilizar un umbral U para medir cuánto tiempo ha estado recibiendo luz el receptor:
 - Si está por debajo de un umbral, es una ráfaga corta.
 - Si está por encima del umbral, es una ráfaga larga.
- Limitaciones de esta alternativa:
 - Todas las ráfagas deben obligatoriamente finalizar con voltaje bajo. Todos los ciclos duran un tiempo U .
 - Ráfaga corta (2 ciclos):
 - En el primer ciclo de reloj, se envía señal luminosa.
 - En el segundo, no se envía señal luminosa.
 - Ráfaga larga (3 ciclos):
 - En el primer ciclo de reloj, se envía señal luminosa.
 - En el segundo, también.
 - En el tercero, no se envía señal luminosa.
 - Fin de símbolo (1 ciclo):
 - Ciclo único: No se envía señal luminosa.

Solución: Alternativa 2

- En las prácticas, utilizaremos la alternativa 2 porque es más sencilla de implementar.



– Parada estratégica (I):

- Llegados a este punto, conviene que diferenciamos correctamente entre dos términos:
 - Los **bits usados para codificar el símbolo que se desea enviar.**
 - Los **bits que realmente se transmiten por el medio físico.**
- En el ejemplo anterior, para transmitir un “0” del código (un punto), se han necesitado transmitir 2 ciclos por láser: Uno en voltaje alto y otro en voltaje bajo.
- Para transmitir un “1” del código (una raya), se han necesitado transmitir 3 ciclos por láser: Dos en voltaje alto y uno en voltaje bajo.
- Para evitar confusiones, en el resto de las prácticas de la asignatura nos referiremos a los bits de cada nivel de abstracción de la siguiente forma:
 - **codeBit:** Un bit de un código a transmitir.
 - **laserBit:** Un bit realmente transmitido por el medio físico.

– Parada estratégica (II):

- Por tanto, para transmitir en el ejemplo los **codeBits 000.11.000.**, se han necesitado un total de **22 laserBits**:

1010100110110010101000

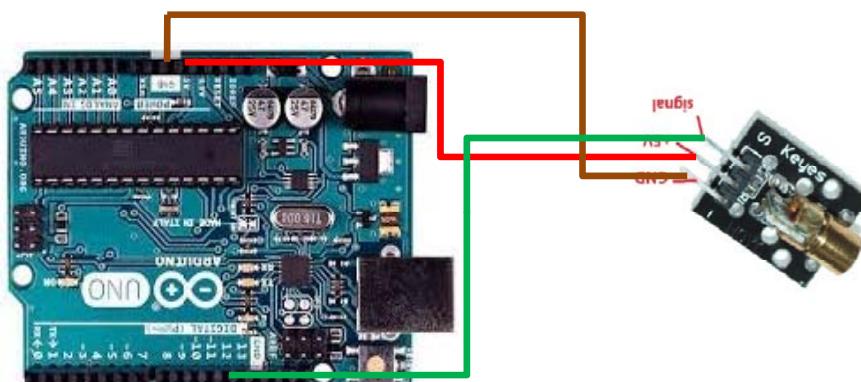
– Estos *lasetBits* se han utilizado para:

- Enviar 000 y el fin del símbolo (**7 laserBits** para **3 codeBits**)
- Enviar 11 y el fin del símbolo (**7 laserBits** para **2 codeBits**)
- Enviar 000 y el fin del símbolo (**7 laserBits** para **3 codeBits**)
- Enviar el fin del mensaje (**1 laserBit**)

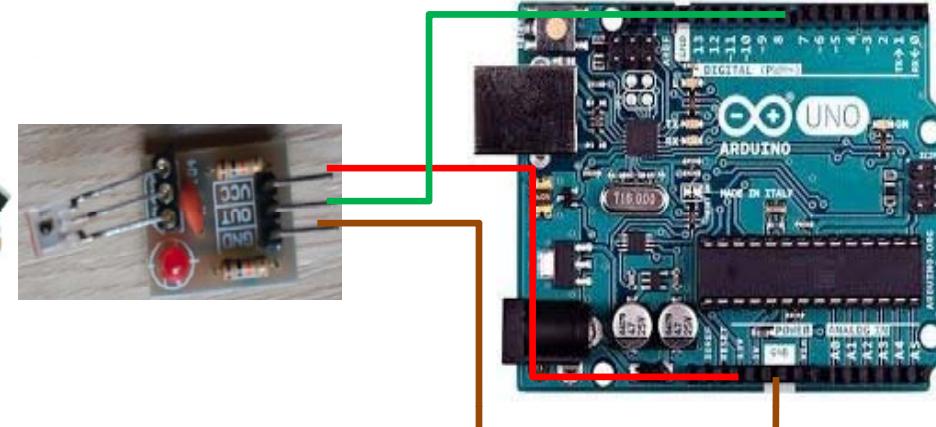
Usaremos esta nomenclatura durante el resto de las prácticas, para conocer si nos estamos refiriendo a bits de códigos o a bits de transmisión de datos.

– Un sencillo programa de envío:

- Enviaremos la información binaria (**codeBits**) 000.11.000. (· · · - - - · · ·) indefinidamente mediante el láser en una placa Arduino.
- Recibiremos la información por el Arduino con el fotorreceptor, guardándola en un buffer.
- Cuando el buffer esté lleno, enviaremos cadenas de caracteres con “0” o con “1” por el puerto USB, para comprobar qué se ha recibido.
- Necesitaremos 2 placas Arduino con las conexiones como se muestran en la figura (**fotorreceptor: Pin 8; emisor láser: Pin 12**):



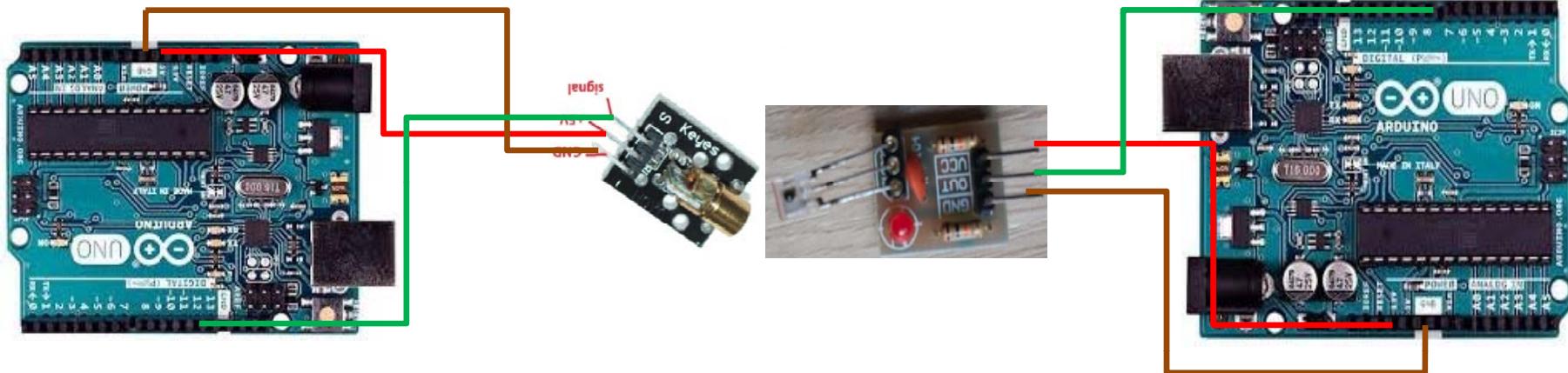
Placa emisora



Placa receptor

– Algunas consideraciones:

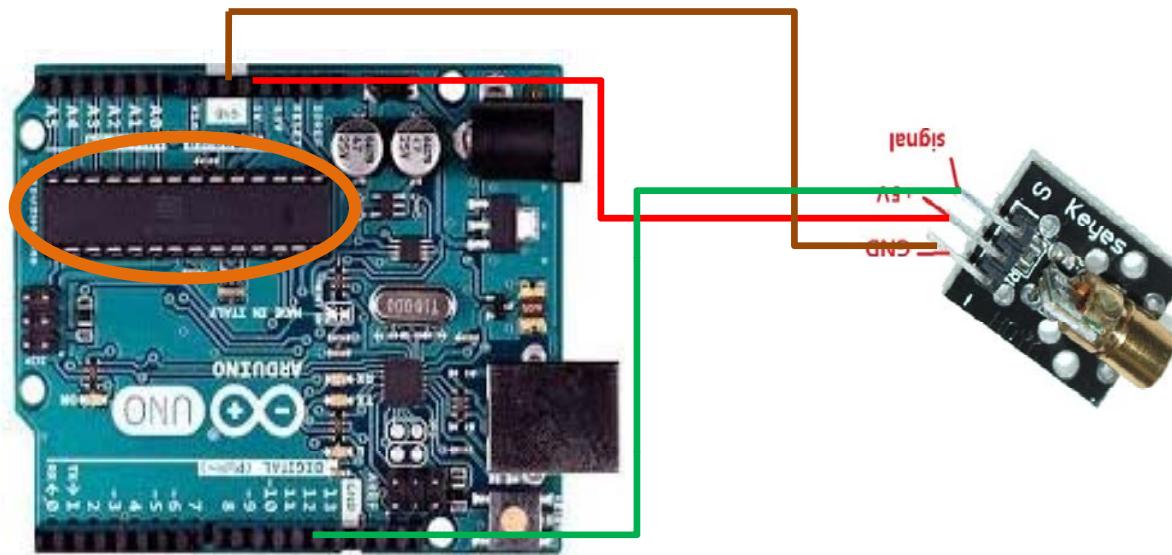
- Hay que asegurarse de que el emisor láser y el fotorreceptor están correctamente alineados para que se reciba información. Se puede hacer uso de los proyectos previos **LEDcontrol** y **SerialFotorresistor** para asegurarnos.
- En el receptor hay que muestrear, al menos, al doble de la frecuencia de envío de datos. Si se envían ráfagas con un tamaño de ciclo de **U**, el receptor debe muestrear a, como mucho, ciclos de **U/2**.



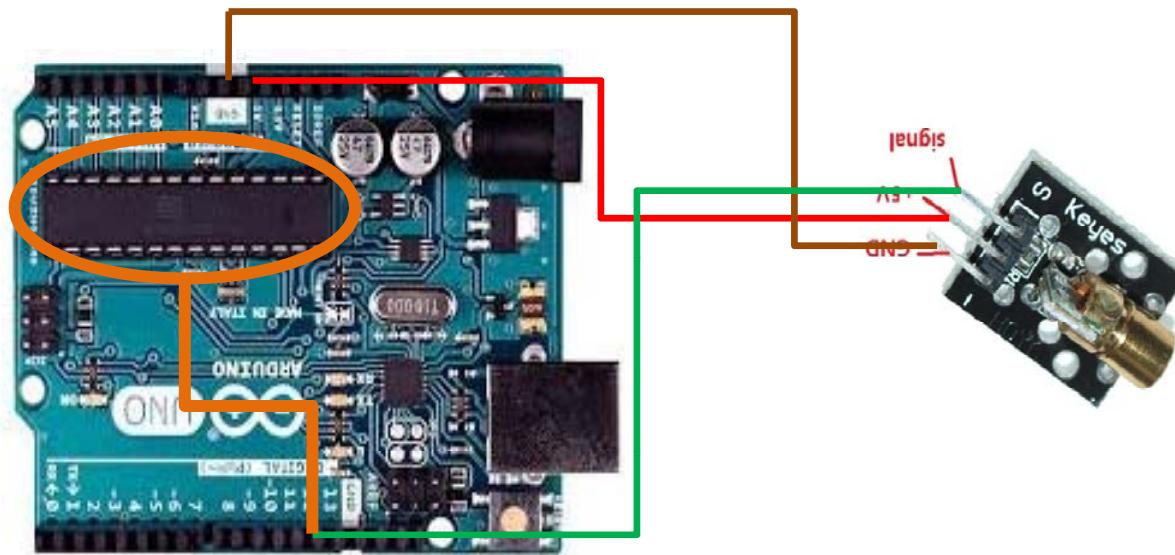
Placa emisora

Placa receptora

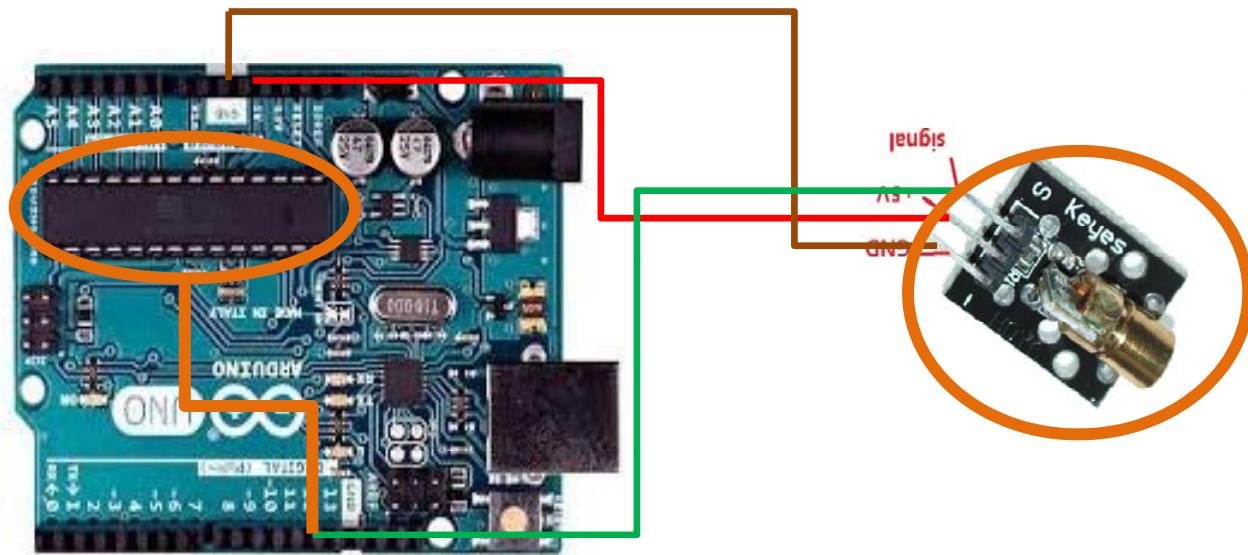
- El valor apropiado para U conlleva:
 - El tiempo que tarda el microprocesador en procesar la orden



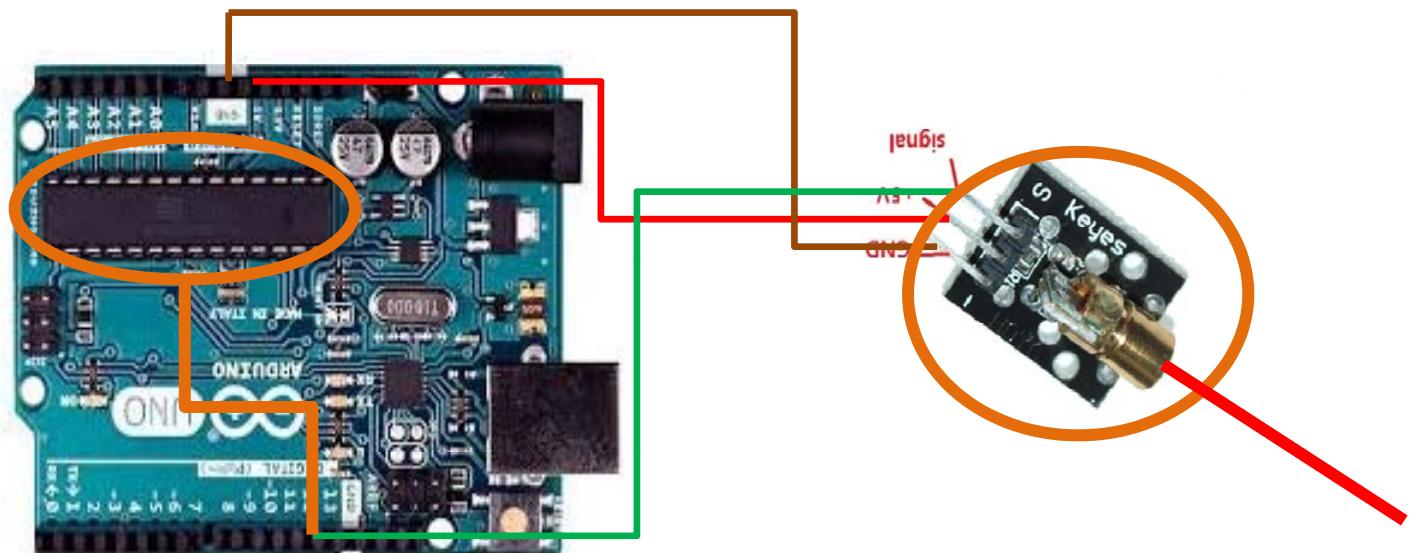
- El valor apropiado para U conlleva:
 - El tiempo que tarda el microprocesador en procesar la orden
 - + el tiempo que se tarda en activar el voltaje en el pin



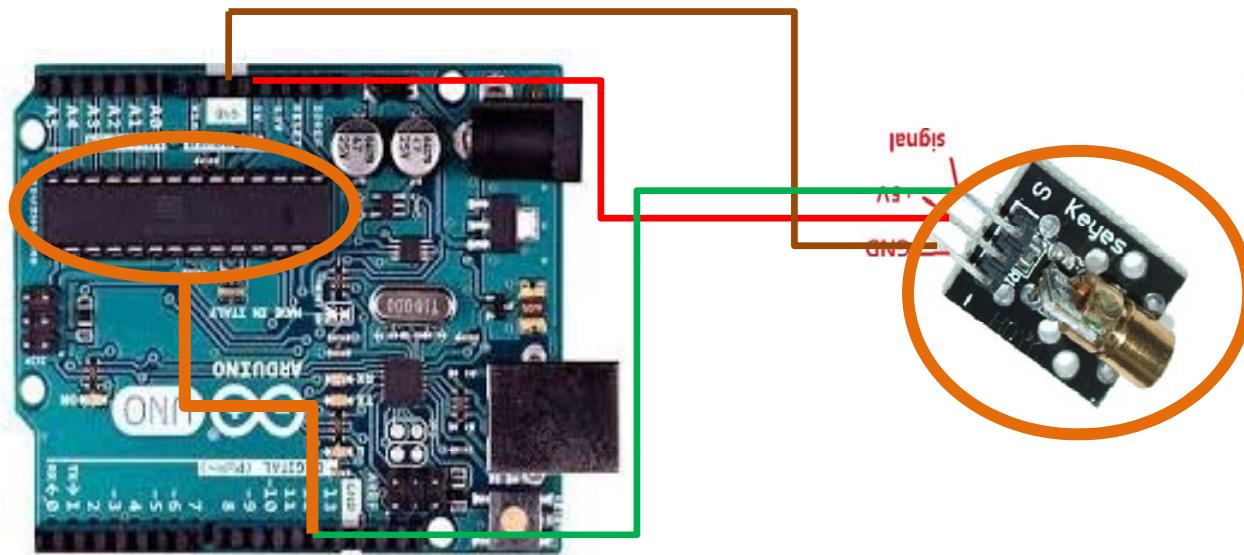
- El valor apropiado para U conlleva considerar:
 - El tiempo que tarda el microprocesador en procesar la orden
 - + el tiempo que se tarda en activar el voltaje en el pin
 - + el tiempo que el láser tarda en realizar su carga y encenderse



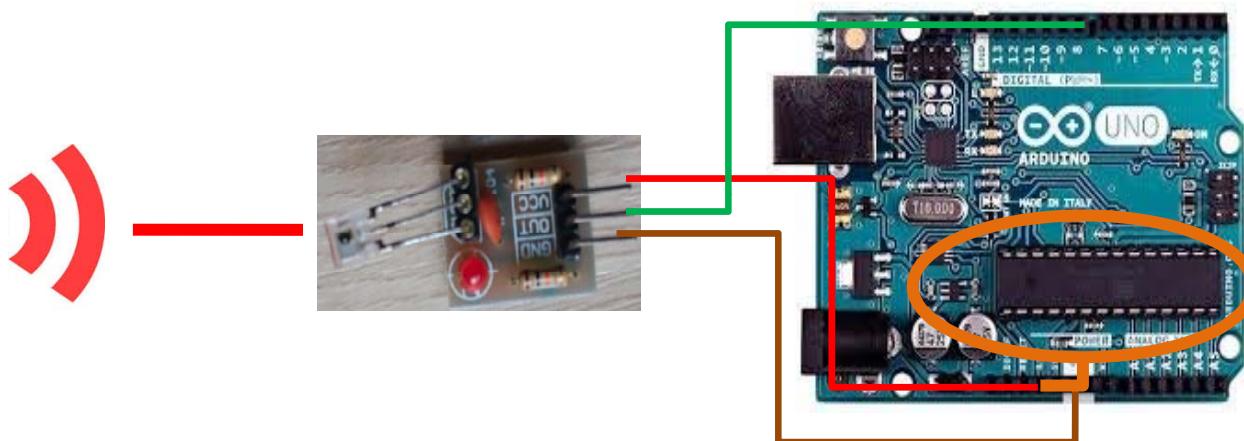
- El valor apropiado para U conlleva considerar:
 - El tiempo que tarda el microprocesador en procesar la orden
 - + el tiempo que se tarda en activar el voltaje en el pin
 - + el tiempo que el láser tarda en realizar su carga y encenderse
 - + el tiempo de transmisión por el canal



- El valor apropiado para U conlleva considerar:
 - El tiempo que tarda el microprocesador en procesar la orden
 - + el tiempo que se tarda en activar el voltaje en el pin
 - + el tiempo que el láser tarda en realizar su carga y encenderse
 - + el tiempo de transmisión por el canal
 - + el tiempo de volver a realizar las mismas acciones para apagar el láser



- El valor apropiado para U conlleva considerar:
 - Además, es necesario tener en cuenta cuánto tiempo tarda el fotorreceptor en detectar la luz, y cuánto tiempo se requiere para procesarla en el Arduino receptor.
- A menor valor de U, más velocidad de transmisión, pero mayor posibilidad de errores.
- Utilizaremos un valor no inferior a **U=15ms, y múltiplo de 3**, suficiente para las prácticas.



- 1. ¿Qué es Arduino?**
- 2. El procesador**
- 3. Compilador y bibliotecas para procesadores AVR**
- 4. Mi primer programa para AVR AtMega328p**
- 5. Comunicaciones en serie**
- 6. Mi primer programa para comunicaciones en serie: ECHO**
- 7. Recepción de datos desde sensores fotorresistores digitales**
- 8. Envío de datos por dispositivos emisores láser**
- 9. La arquitectura del sistema de emisión/recepción de datos por láser**
- 10. Ampliación de la biblioteca <ticcommardu.h>**
- 11. Ejemplo: Envío de datos por láser**
- 12. Ejemplo: Recepción de datos por láser**
- 13. Proyecto final de la práctica**

- Comenzaremos por ampliar la biblioteca <ticcommardu.h> para enviar “laser-bits” por el láser y recibirlos con el fotorreceptor. Necesitaremos:
 - Para enviar datos:
 - Una función *initLaserEmitter* que inicialice el Pin del láser como salida, y que inicialmente lo establezca a voltaje bajo (no emitir).
 - Una función *sendLaserBit* que envíe una ráfaga corta, larga o ninguna.
 - Para recibir datos:
 - Una función *initLaserReceiver* que inicialice el Pin del fotorreceptor como entrada.
 - Una función *recvLaserBit* que detecte la recepción de una ráfaga corta, larga o ninguna.

– Biblioteca <ticcommardu.h> (I):

```
1
2 #ifndef __TICCOMMARDU__
3
4 #define __TICCOMMARDU__
5
6 // Ciclo de reloj del procesador: 16MHz
7 #define F_CPU 16000000UL
8
9
10 // Tiempo de ciclo para comunicaciones por láser, en milisegundos
11 #define UMBRAL_U 15
12
13 // Tiempo de muestreo
14 #define SAMPLE_PERIOD (UMBRAL_U/3)
15
16
17 // Constante que representa una ráfaga corta (puntos)
18 #define LASER_DOT 0
19
20 // Constante que representa una ráfaga larga (raya)
21 #define LASER_DASH 1
22 |
23
24 // Constante que representa ninguna ráfaga
25 #define LASER_NONE 2
26
```

– Biblioteca <ticcommardu.h> (I):

```
1
2 #ifndef __TICCOMMARDU__
3
4 #define __TICCOMMARDU__
5
6 // Ciclo de reloj del procesador: 16MHz
7 #define F_CPU 16000000UL
8
9
10 // Tiempo de ciclo para comunicaciones por láser, en milisegundos
11 #define UMBRAL_U 15
12
13 // Tiempo de muestreo
14 #define SAMPLE_PERIOD (UMBRAL_U/3)
15
16
17 // Constante que representa una ráfaga corta (puntos)
18 #define LASER_DOT 0
19
20 // Constante que representa una ráfaga larga (raya)
21 #define LASER_DASH 1
22 |
23
24 // Constante que representa ninguna ráfaga
25 #define LASER_NONE 2
26
```

Pasamos aquí la velocidad de la CPU.
Será necesaria para controlar los ciclos de las ráfagas.

– Biblioteca <ticcommardu.h> (I):

```
1
2 #ifndef __TICCOMMARDU__
3
4 #define __TICCOMMARDU__
5
6 // Ciclo de reloj del procesador: 16MHz
7 #define F_CPU 16000000UL
8
9
10 // Tiempo de ciclo para comunicación
11 #define UMBRAL_U 15
12
13 // Tiempo de muestreo
14 #define SAMPLE_PERIOD (UMBRAL_U/3)
15
16
17 // Constante que representa una ráfaga corta (puntos)
18 #define LASER_DOT 0
19
20 // Constante que representa una ráfaga larga (raya)
21 #define LASER_DASH 1
22 |
23
24 // Constante que representa ninguna ráfaga
25 #define LASER_NONE 2
26
```

Tiempo mínimo que dura un ciclo de ráfaga.

– Biblioteca <ticcommardu.h> (I):

```
1
2 #ifndef __TICCOMMARDU__
3
4 #define __TICCOMMARDU__
5
6 // Ciclo de reloj del procesador: 16MHz
7 #define F_CPU 16000000UL
8
9
10 // Tiempo de ciclo para comunicación
11 #define UMBRAL_U 15
12
13 // Tiempo de muestreo
14 #define SAMPLE_PERIOD (UMBRAL_U/3)
15
16
17 // Constante que representa una ráfaga corta (puntos)
18 #define LASER_DOT 0
19
20 // Constante que representa una ráfaga larga (raya)
21 #define LASER_DASH 1
22 |
23
24 // Constante que representa ninguna ráfaga
25 #define LASER_NONE 2
26
```

La frecuencia de muestreo que utilizaremos para “mirar” qué se detecta en el fotoreceptor.

– Biblioteca <ticcommardu.h> (I):

```
1
2 #ifndef __TICCOMMARDU__
3
4 #define __TICCOMMARDU__
5
6 // Ciclo de reloj del procesador: 16MHz
7 #define F_CPU 16000000UL
8
9
10 // Tiempo de ciclo para comunicaciones por láser, en milisegundos
11 #define UMBRAL_U 15
12
13 // Tiempo de muestreo
14 #define SAMPLE_PERIOD (UMBRAL_U/3)
15
16
17 // Constante que representa una ráfaga corta (puntos)
18 #define LASER_DOT 0
19
20 // Constante que representa una ráfaga larga (raya)
21 #define LASER_DASH 1
22 |
23
24 // Constante que representa ninguna ráfaga
25 #define LASER_NONE 2
26
```

Constantes para simplificar el uso de envío/recepción de ráfagas cortas, largas o ninguna.

- Biblioteca <ticcommardu.h> (II):

- Funciones requeridas para inicializar el láser y para enviar una ráfaga.

```
26
27
28 /**
29 * Inicializa el emisor de láser, apagándolo
30 */
31 void initLaserEmitter();
32
33
34 /**
35 * Función para enviar por láser un símbolo.
36 *
37 * Entradas:
38 * what: Qué se envía. Puede ser uno de los valores siguientes:
39 * LASER_DOT para enviar una ráfaga corta,
40 * LASER_DASH para enviar una ráfaga larga
41 * LASER_NONE para pasar un ciclo sin enviar nada
42 *
43 */
44 void sendLaserBit(const unsigned char what);
45
```

- Biblioteca <ticcommardu.h> (III):
 - Funciones requeridas para inicializar el fotorreceptor y para detectar ráfagas.

```
46
47 /**
48 * Inicializa el receptor de láser
49 */
50 void initLaserReceiver();
51
52
53 /**
54 * Función para recibir un símbolo desde el fotorreceptor de láser.
55 *
56 * Entradas: Ninguna
57 *
58 * salidas
59 * what: Qué se ha recibido. Puede ser uno de los valores siguientes:
60 *      LASER_DOT para recibir una ráfaga corta,
61 *      LASER_DASH para recibir una ráfaga larga
62 *      LASER_NONE para indicar que no se ha recibido nada
63 *
64 */
65 void recvLaserBit(unsigned char & what);
66
```

- Biblioteca <ticcommardu.h> (IV):

- El resto del fichero sigue con las funciones que ya teníamos para gestión de comunicaciones por el puerto USB:

```
67
68
69 /**
70 * Función para enviar un mensaje por USB a PC
71 *
72 * Entradas:
73 *   data: Cadena de datos a enviar.
74 *
75 * Salidas: true si los datos se enviaron con éxito, false en otro caso
76 */
77 bool arduSendUSB(const char *data);
78
79
80
81
82 /**
83 * Función para recibir un mensaje por USB desde PC
84 *
85 * Entradas: Ninguna
86 *
87 * Salidas: true si los datos se enviaron con éxito, false en otro caso
88 *   en el parámetro data: Cadena de caracteres recibida
89 */
90 bool arduReceiveUSB (char *data);
91
92 #endif
93
```

- Fichero **ticcommardu.cpp (I)**:

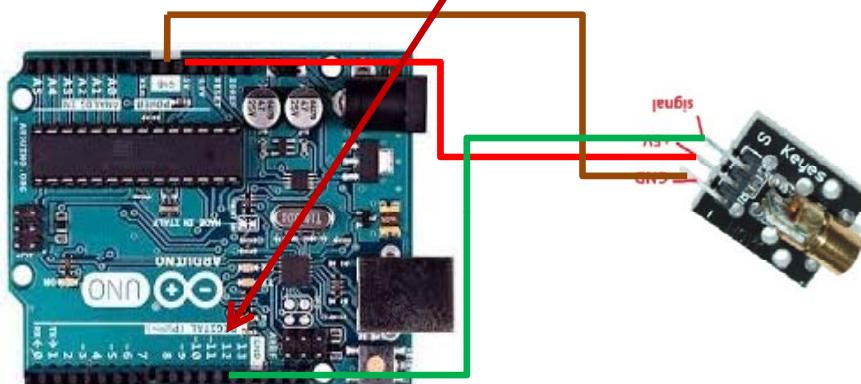
- El fichero requiere incluir nuevas bibliotecas para usar E/S y delays

```
1 #include <ticcommardu.h>
2 #include <uart.h>
3 #include <string.h>
4 #include <util/delay.h>
5
6
7 void initLaserEmitter() {
8
9 |
10 // Asumimos que está en el pin 12 de Arduino.
11 DDRB |= _BV(DDB4);
12
13 // Ponemos salida a baja
14 PORTB &= ~_BV(DDB4);
15 }
16
```

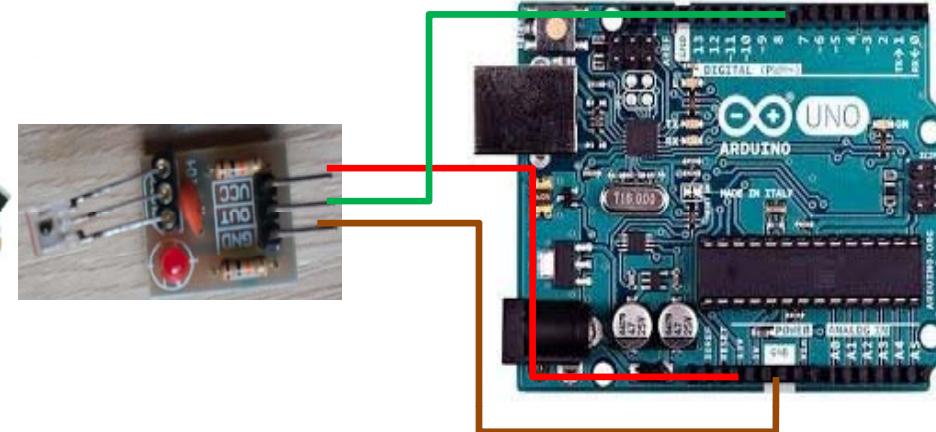
- Fichero **ticcommardu.cpp (I)**:
 - La inicialización del láser se realiza activando su puerto como de salida, y apagándolo inicialmente.
 - Asumimos que el puerto es el correspondiente al Pin 12 de la placa, según el esquema mostrado anteriormente.

```
1 #include <ticcommardu.h>
2 #include <uart.h>
3 #include <string.h>
4 #include <util/delay.h>
5
6
7 void initLaserEmitter() {
8
9 |
10 // Asumimos que está en el pin 12 de Arduino.
11 DDRB |= _BV(DDB4);
12
13 // Ponemos salida a baja
14 PORTB &= ~_BV(DDB4);
15 }
16
```

```
1 #include <ticcommardu.h>
2 #include <uart.h>
3 #include <string.h>
4 #include <util/delay.h>
5
6
7 void initLaserEmitter() {
8
9 |
10 // Asumimos que está en el pin 12 de Arduino.
11 DDRB |= _BV(DDB4);
12
13 // Ponemos salida a baja
14 PORTB &= ~_BV(DDB4);
15 }
16
```



Placa emisora



Placa receptora

- Fichero **ticcommardu.cpp (II):**

- El envío de ráfagas consiste en activar el láser, esperar a que termine la ráfaga, apagarlo, y esperar a que pase un ciclo.

```
17
18 void sendLaserBit(const unsigned char what) {
19
20 // Ponemos salida a alta si hay que enviar ráfaga
21 if (what != LASER_NONE)
22     PORTB |= _BV(DDB4);
23
24 // Esperamos el tiempo de ciclo
25 if (what != LASER_DASH)
26     _delay_ms(UMBRAL_U);
27 else
28     _delay_ms(2*UMBRAL_U);
29
30 // Ponemos salida a baja
31 PORTB &= ~_BV(DDB4);
32
33 // Y esperamos un ciclo
34 if (what != LASER_NONE)
35     _delay_ms(UMBRAL_U);
36 }
37
```

- Fichero **ticcommardu.cpp (III)**:

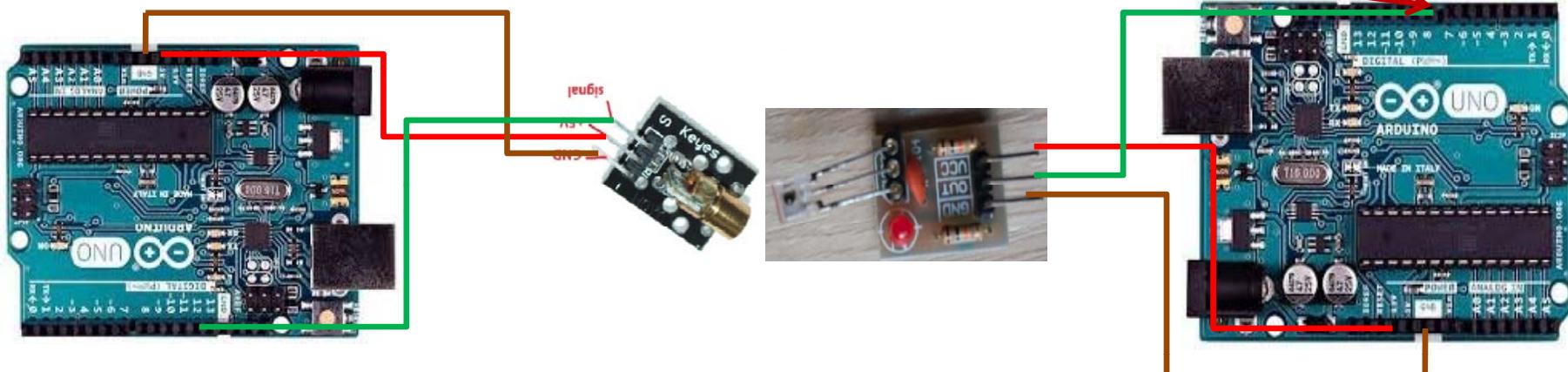
- La inicialización del receptor se realiza activando su puerto como de entrada.
- Asumimos que el puerto es el correspondiente al Pin 8 de la placa, según el esquema mostrado anteriormente.

```
39 void initLaserReceiver(){  
40  
41     // Asumimos que está en el pin 8 de Arduino.  
42     DDRB &= ~_BV(DDB0);  
43 }  
44
```

– Fichero **ticcommardu.cpp** (III):

- La inicialización del receptor se realiza activando su puerto como de entrada.
- Asumimos que el puerto es el correspondiente al Pin 8 de la placa, según el esquema mostrado anteriormente.

```
39 void initLaserReceiver(){  
40  
41     // Asumimos que está en el pin 8 de Arduino.  
42     DDRB &= ~_BV(DDB0);  
43 }  
44
```



Placa emisora

Placa receptora

- Fichero **ticcommardu.cpp**:

- La lectura (recepción de datos por el fotorreceptor) es algo más compleja:
 - Debemos contar cuántos “laser-Bit” a 0 hay y, si el valor es superior a un umbral (considerando la frecuencia de muestreo), se asume que no se ha enviado nada.
 - En caso contrario, debemos contar cuántos “laser-Bit” a 1 hay y, si el valor es superior a un umbral (considerando la frecuencia de muestreo), se asume que se ha enviado una ráfaga larga.

– Fichero **ticcommardu.cpp (IV)**:

- Parte 1: Contar cuántos “laser-Bit” a 0 hay y, si el valor es superior a un umbral, devolver **LASER_NONE**.

```
45
46 void recvLaserBit(unsigned char & what) {
47
48     unsigned char cBajo=0; // Contador de veces que se detecta voltaje bajo
49     unsigned char cAlto=0; // Contador de veces que se detecta voltaje alto
50     unsigned char dato; // Dato a medir
51
52     // Inicialmente suponemos que está en bajo
53     // Comprobamos si es mensaje LASER_NONE
54     do {
55
56         // Asumimos que está en el PIN 8
57         dato= PINB & 0x01;
58
59         if (dato>0)
60             cAlto++;
61         else
62             cBajo++;
63
64         if (cBajo>3) {
65             what= LASER_NONE;
66             return;
67         }
68
69         // Esperamos al siguiente muestreo
70         _delay_ms( SAMPLE_PERIOD );
71
72     } while (dato == 0);
73 }
```

- Fichero **ticcommardu.cpp (IV)**:

- Parte 1: Contar cuántos “laser-Bit” a 0 hay y, si el valor es superior a un umbral, devolver **LASER_NONE**.

```
45
46 void recvLaserBit(unsigned char & what) {
47
48     unsigned char cBajo=0; // Contador de veces que se detecta voltaje bajo
49     unsigned char cAlto=0; // Contador de veces que se detecta voltaje alto
50     unsigned char dato; // Dato a medir
51
52     // Inicialmente suponemos que está en bajo
53     // Comprobamos si es mensaje LASER_NONE
54     do {
55
56         // Asumimos que está en el PIN 8
57         dato= PINB & 0x01;
58
59         if (dato>0)
60             cAlto++;
61         else
62             cBajo++;
63
64         if (cBajo>3) {
65             what= LASER_NONE;
66             return;
67         }
68
69         // Esperamos al siguiente muestreo
70         _delay_ms( SAMPLE_PERIOD );
71
72     } while (dato == 0);
73 }
```

Medimos el pin 8 de la placa (PBO del procesador, primer bit de PINB)

- Fichero **ticcommardu.cpp (IV)**:

- Parte 1: Contar cuántos “laser-Bit” a 0 hay y, si el valor es superior a un umbral, devolver **LASER_NONE**.

```
45
46 void recvLaserBit(unsigned char & what) {
47
48     unsigned char cBajo=0; // Contador de veces que se detecta voltaje bajo
49     unsigned char cAlto=0; // Contador de veces que se detecta voltaje alto
50     unsigned char dato; // Dato a medir
51
52     // Inicialmente suponemos que está en bajo
53     // Comprobamos si es mensaje LASER_NONE
54     do {
55
56         // Asumimos que está en el PIN 8
57         dato= PINB & 0x01;
58
59         if (dato>0)
60             cAlto++;
61         else
62             cBajo++;
63
64         if (cBajo>3) {
65             what= LASER_NONE;
66             return;
67         }
68
69         // Esperamos al siguiente muestreo
70         _delay_ms( SAMPLE_PERIOD );
71
72     } while (dato == 0);
73 }
```

Si se detecta luz, debemos empezar a contar desde ya.

- Fichero **ticcommardu.cpp (IV)**:

- Parte 1: Contar cuántos “laser-Bit” a 0 hay y, si el valor es superior a un umbral, devolver **LASER_NONE**.

```
45
46 void recvLaserBit(unsigned char & what) {
47
48     unsigned char cBajo=0; // Contador de veces que se detecta voltaje bajo
49     unsigned char cAlto=0; // Contador de veces que se detecta voltaje alto
50     unsigned char dato; // Dato a medir
51
52     // Inicialmente suponemos que está en bajo
53     // Comprobamos si es mensaje LASER_NONE
54     do {
55
56         // Asumimos que está en el PIN 8
57         dato= PINB & 0x01;
58
59         if (dato>0)
60             cAlto++;
61         else
62             cBajo++;
63
64         if (cBajo>3) {
65             what= LASER_NONE;
66             return;
67     }
68
69     // Esperamos al siguiente muestreo
70     _delay_ms( SAMPLE_PERIOD );
71
72 } while (dato == 0);
```

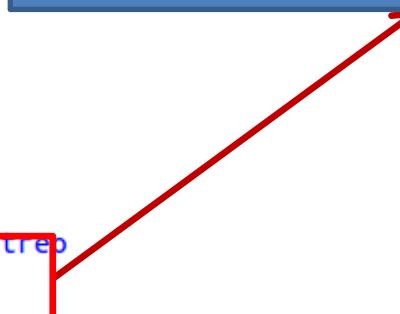
Dos ciclos seguidos, muestrados a U/3 ms, dan 6 valores muestrados. Si todos ellos son 0V, entonces se detecta que no hay ráfaga

- Fichero **ticcommardu.cpp (IV)**:

- Parte 1: Contar cuántos “laser-Bit” a 0 hay y, si el valor es superior a un umbral, devolver **LASER_NONE**.

```
45
46 void recvLaserBit(unsigned char & what) {
47
48     unsigned char cBajo=0; // Contador de veces que se detecta voltaje bajo
49     unsigned char cAlto=0; // Contador de veces que se detecta voltaje alto
50     unsigned char dato; // Dato a medir
51
52     // Inicialmente suponemos que está en bajo
53     // Comprobamos si es mensaje LASER_NONE
54     do {
55
56         // Asumimos que está en el PIN 8
57         dato= PINB & 0x01;
58
59         if (dato>0)
60             cAlto++;
61         else
62             cBajo++;
63
64         if (cBajo>3) {
65             what= LASER_NONE;
66             return;
67         }
68
69         // Esperamos al siguiente muestreo
70         _delay_ms( SAMPLE_PERIOD );
71
72     } while (dato == 0);
73 }
```

Esperamos el tiempo de muestreo para volver a medir



- Fichero **ticcommardu.cpp (IV)**:

- Parte 1: Contar cuántos “laser-Bit” a 0 hay y, si el valor es superior a un umbral, devolver **LASER_NONE**.

```
45
46 void recvLaserBit(unsigned char & what) {
47
48     unsigned char cBajo=0; // Contador de veces que se detecta voltaje bajo
49     unsigned char cAlto=0; // Contador de veces que se detecta voltaje alto
50     unsigned char dato; // Dato a medir
51
52     // Inicialmente suponemos que está en bajo
53     // Comprobamos si es mensaje LASER_NONE
54     do {
55
56         // Asumimos que está en el PIN 8
57         dato= PINB & 0x01;
58
59         if (dato>0)
60             cAlto++;
61         else
62             cBajo++;
63
64         if (cBajo>3) {
65             what= LASER_NONE;
66             return;
67         }
68
69         // Esperamos al siguiente muestreo
70         _delay_ms( SAMPLE_PERIOD );
71
72     } while (dato == 0);
73 }
```

Sólo saldremos de aquí cuando detectemos que hay ráfaga.

- Fichero **ticcommardu.cpp (V)**:

- Parte 2: Contar cuántos “laser-Bit” a 1 hay y, si el valor es superior a un umbral, devolver **LASER_DASH**. En otro caso, **LASER_DOT**.
- El código es similar a detectar **LASER_NONE**:

```
73
74
75 // Si llegamos aquí, está llegando una ráfaga
76 // Comprobamos si es ráfaga corta o larga
77 do {
78
79     // Asumimos que está en el PIN 8
80     dato= PINB & 0x01;
81
82     if (dato>0)
83         cAlto++;
84
85     // Esperamos al siguiente muestreo
86     _delay_ms( SAMPLE_PERIOD );
87
88 } while (dato == 1);
89
90 // Si hemos muestreado más de 3 veces lo mismo, entonces es raya
91 if (cAlto>3)
92     what= LASER_DASH;
93
94 else // En otro caso, es un punto
95     what= LASER_DOT;
96
97 }
98 }
```

- Fichero **ticcommardu.cpp (VI)**:

- El resto del fichero contiene la implementación de las funciones **arduSendUSB** y **arduReceiveUSB** que ya habíamos desarrollado previamente.

- 1. ¿Qué es Arduino?**
- 2. El procesador**
- 3. Compilador y bibliotecas para procesadores AVR**
- 4. Mi primer programa para AVR AtMega328p**
- 5. Comunicaciones en serie**
- 6. Mi primer programa para comunicaciones en serie: ECHO**
- 7. Recepción de datos desde sensores fotorresistores digitales**
- 8. Envío de datos por dispositivos emisores láser**
- 9. La arquitectura del sistema de emisión/recepción de datos por láser**
- 10. Ampliación de la biblioteca <ticcommardu.h>**
- 11. Ejemplo: Envío de datos por láser**
- 12. Ejemplo: Recepción de datos por láser**
- 13. Proyecto final de la práctica**

- Ahora elaboraremos un pequeño ejemplo de prueba para comprobar que la emisión de láser funciona.
- Para este ejemplo, establecer **UMBRAL_U=500** (medio segundo), para poder comprobar visualmente que el sistema de emisión funciona.
- Crearemos un programa “EmisorArdu”, que emita el siguiente mensaje indefinidamente (*codeBits*):

• • • — — • • •

0 0 0 1 1 0 0 0

Programa **emisorArdu** para la plataforma Arduino Uno:

```
3 #include <ticcommardu.h>
4 #include <util/delay.h>
5
6
7 int main(void) {
8
9     initLaserEmitter(); // Iniciamos el láser
10
11    while (1) {
12
13        // Enviamos: ... -- ...
14        sendLaserBit(LASER_DOT); sendLaserBit(LASER_DOT); sendLaserBit(LASER_DOT);
15        sendLaserBit(LASER_NONE); // Fin de símbolo
16
17        sendLaserBit(LASER_DASH); sendLaserBit(LASER_DASH);
18        sendLaserBit(LASER_NONE); // Fin de símbolo
19
20        sendLaserBit(LASER_DOT); sendLaserBit(LASER_DOT); sendLaserBit(LASER_DOT);
21        sendLaserBit(LASER_NONE); // Fin de símbolo
22
23    }
24
25    return 0;
26 }
27
```

Makefile para el programa **emisorArdu** (asumiendo que Arduino está conectado a **/dev/ttyACM1**):

```
5 COMMPORTEMISOR = /dev/ttyACM1
6
7 emisorArdu: ticcommarducpp emisorArducpp uartcpp
8     @echo Generando binario Arduino emisor...
9     avr-gcc -mmcu=atmega328p obj/ticcommardu.o obj/uart.o obj/emisorArdu.o -o obj/emisorArdu.bin
10    @echo Generando HEX Arduino emisor...
11    avr-objcopy -O ihex -R .eeprom obj/emisorArdu.bin hex/emisorArdu.hex
12    @echo emisorArdu compilado. Ejecute make sendEmisor para enviarlo a la plataforma Arduino.
13
14 emisorArducpp:
15     @echo Compilando emisorArdu...
16     avr-gcc -Os -mmcu=atmega328p -c -o obj/emisorArdu.o src/emisorArdu.cpp -Iinclude
17
18 sendEmisor:
19     @echo Enviando emisorArdu a Arduino
20     sudo avrdude -F -V -c arduino -p ATMEGA328P -P $(COMMPORTEMISOR) -b 115200 -U flash:w:hex/emisorArdu.hex
21
22 ticcommarducpp:
23     @echo Compilando TICCommArdu...
24     avr-gcc -Os -mmcu=atmega328p -c -o obj/ticcommardu.o src/ticcommardu.cpp -Iinclude
25
26 uartcpp:
27     @echo Compilando biblioteca UART...
28     avr-gcc -Os -mmcu=atmega328p -c -o obj/uart.o src/uart.cpp -Iinclude
29
```

- 1. ¿Qué es Arduino?**
- 2. El procesador**
- 3. Compilador y bibliotecas para procesadores AVR**
- 4. Mi primer programa para AVR AtMega328p**
- 5. Comunicaciones en serie**
- 6. Mi primer programa para comunicaciones en serie: ECHO**
- 7. Recepción de datos desde sensores fotorresistores digitales**
- 8. Envío de datos por dispositivos emisores láser**
- 9. La arquitectura del sistema de emisión/recepción de datos por láser**
- 10. Ampliación de la biblioteca <ticcommardu.h>**
- 11. Ejemplo: Envío de datos por láser**
- 12. Ejemplo: Recepción de datos por láser**
- 13. Proyecto final de la práctica**

- Ahora elaboraremos un pequeño ejemplo de prueba para comprobar que la recepción de láser funciona.
- Para este ejemplo, establecer **UMBRAL_U=15 tanto en el emisor como en el receptor.**
- Crearemos dos aplicaciones:
 - Un programa **ReceptorPC**, que reciba cadenas de caracteres por USB desde una plataforma Arduino y las muestre por pantalla.
 - Un programa **ReceptorArdu**, que reciba ráfagas por láser y los vaya guardando en un buffer. Cuando el buffer se encuentre lleno, lo enviará a un programa “ReceptorPC” a través de USB.
- En el ejemplo, asumiremos que el receptor se conecta al puerto serie por **/dev/ttyACM0** y que emisor se conecta en **/dev/ttyACM1**.

– Fichero receptorPC.cpp:

- Su implementación será muy parecida al programa **fotoPC** realizado en el cuaderno de prácticas:

```
1 #include <cstring>
2 #include <ticcomm.h>
3 #include <iostream>
4 #include <unistd.h>
5 #include <termios.h>
6
7 using namespace std;
8
9 // Puerto de comunicaciones con arduino
10#define USBPORT "/dev/ttyACM0"
11
12
13 int main(int argc, char *argv[]) {
14
15    int fd; // Descriptor de fichero del puerto USB
16    char buf[129]; // Buffer de salida. Tamaño máximo de 128 caracteres
17
18
19    // Inicializamos puerto
20    fd= InicializarUSB(USBPORT);
21    if (fd == -1) {
22        cout<<"Error inicializando puerto "<<USBPORT<<endl;
23        return 0;
24    }
25
26
27    // Bucle principal
28    while (1) {
29
30        // Leemos datos del USB
31        if (receiveUSB (fd, buf)) {
32            cout << "Mensaje recibido: "<<buf<<endl;
33        } else {
34            cout<<"\tERROR RECIBIENDO DATOS\n\n";
35            close(fd);
36            return 0;
37        }
38    }
39
40    // Cerramos el puerto USB
41    close(fd);
42    return 0;
43 }
```

– Fichero receptorPC.cpp:

- Su implementación será muy parecida al programa **fotoPC** realizado en el cuaderno de prácticas:

Sólo hay que:

1. Inicializar el USB.
2. Recibir el buffer
3. Mostrarlo por pantalla.

```
1 #include <cstring>
2 #include <ticcomm.h>
3 #include <iostream>
4 #include <unistd.h>
5 #include <termios.h>
6
7 using namespace std;
8
9 // Puerto de comunicaciones con arduino
10 #define USBPORT "/dev/ttyACM0"
11
12
13 int main(int argc, char *argv[]) {
14
15     int fd; // Descriptor de fichero del puerto USB
16     char buf[129]; // Buffer de salida. Tamaño máximo de 128 caracteres
17
18
19     // Inicializamos puerto
20     fd= InicializarUSB(USBPORT);
21     if (fd == -1) {
22         cout<<"Error inicializando puerto "<<USBPORT<<endl;
23         return 0;
24     }
25
26
27     // Bucle principal
28     while (1) {
29
30         // Leemos datos del USB
31         if (receiveUSB (fd, buf)) {
32             cout << "Mensaje recibido: "<<buf<<endl;
33         } else {
34             cout<<"\tERROR RECIBIENDO DATOS\n\n";
35             close(fd);
36             return 0;
37         }
38     }
39
40     // Cerramos el puerto USB
41     close(fd);
42     return 0;
43 }
```

- Makefile del fichero receptorPC.cpp:

```
58
59 receptorPC: receptorPCcpp ticcommppccpp
60     @echo Generando fichero binario receptorPC...
61     g++ -o bin/receptorPC obj/receptorPC.o obj/ticcommppc.o
62
63 receptorPCcpp:
64     @echo Compilando receptorPC...
65     g++ -c -o obj/receptorPC.o src/receptorPC.cpp -Iinclude
66
67
68 ticcommppccpp:
69     @echo Compilando fuentes de biblioteca TICCommPC...
70     g++ -c -o obj/ticcommppc.o src/ticcommppc.cpp -Iinclude
71
72
73 clean:
74     rm -f -r *~
75     rm -f -r bin/*
76     rm -f -r obj/*
77     rm -f -r hex/*
78     rm -f -r include/*~
79     rm -f -r src/*~
80
```

– Fichero receptorArdu.cpp (I):

- Primero inicializaremos las comunicaciones por el puerto UART y configuraremos el receptor láser en nuestra biblioteca:

```
1 // Velocidad (en baudios) de las comunicaciones serie
2 #define UART_BAUD_RATE 9600
3
4
5 #include <ticcommardu.h>
6 #include <avr/interrupt.h>
7 #include <uart.h>
8
9
10 int main(void) {
11
12     unsigned char dato; // Dato a leer desde el fotoreceptor
13     char cadena[101];
14     unsigned int n= 0;
15
16
17     // Inicialización del puerto UART con la velocidad
18     // en baudios del puerto, y la velocidad del procesador
19     uart_init( UART_BAUD_SELECT(UART_BAUD_RATE,F_CPU) );
20
21     // Activación de las interrupciones hardware para
22     // control del puerto serie
23     sei();
24
25
26     // Inicializamos el fotorreceptor
27     initLaserReceiver();
```

– Fichero receptorArdu.cpp (I):

- Primero inicializaremos las comunicaciones por el puerto UART y configuraremos el receptor láser en nuestra biblioteca:

```
1 // Velocidad (en baudios) de las comunicaciones serie
2 #define UART_BAUD_RATE 9600
3
4
5 #include <ticcommardu.h>
6 #include <avr/interrupt.h>
7 #include <uart.h>
8
9
10 int main(void) {
11
12     unsigned char dato; // Dato a leer desde el puerto serie
13     char cadena[101];
14     unsigned int n= 0;
15
16
17     // Inicialización del puerto UART con la velocidad
18     // en baudios del puerto, y la velocidad de reloj
19     uart_init( UART_BAUD_SELECT(UART_BAUD_RATE,
20
21         // Activación de las interrupciones hardware
22         // control del puerto serie
23         sei());
24
25
26     // Inicializamos el fotorreceptor
27     initLaserReceiver();
```

Datos a usar para la recepción por el fotorreceptor y para llenar el buffer de salida por USB.

“cadena” es el buffer que enviaremos por USB. En total, 100 bytes útiles.

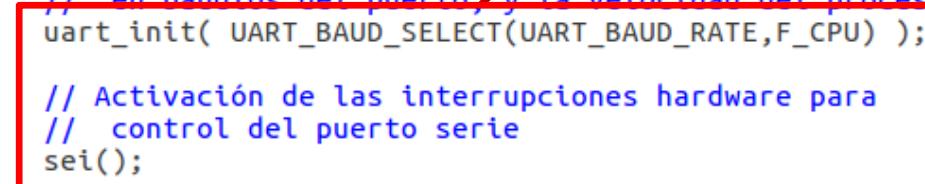
“n” es el número de bytes de “cadena” que llevamos llenos.

– Fichero receptorArdu.cpp (I):

- Primero inicializaremos las comunicaciones por el puerto UART y configuraremos el receptor láser en nuestra biblioteca:

```
1 // Velocidad (en baudios) de las comunicaciones serie
2 #define UART_BAUD_RATE 9600
3
4
5 #include <ticcommardu.h>
6 #include <avr/interrupt.h>
7 #include <uart.h>
8
9
10 int main(void) {
11
12     unsigned char dato; // Dato a leer desde el fotorreceptor
13     char cadena[101];
14     unsigned int n= 0;
15
16
17     // Inicialización del puerto UART con la velocidad
18     // en baudios del puerto, y la velocidad del procesador
19     uart_init( UART_BAUD_SELECT(UART_BAUD_RATE,F_CPU) );
20
21     // Activación de las interrupciones hardware para
22     // control del puerto serie
23     sei();
24
25
26     // Inicializamos el fotorreceptor
27     initLaserReceiver();
```

Inicialización del puerto UART
para USB.



– Fichero receptorArdu.cpp (I):

- Primero inicializaremos las comunicaciones por el puerto UART y configuraremos el receptor láser en nuestra biblioteca:

```
1 // Velocidad (en baudios) de las comunicaciones serie
2 #define UART_BAUD_RATE 9600
3
4
5 #include <ticcommardu.h>
6 #include <avr/interrupt.h>
7 #include <uart.h>
8
9
10 int main(void) {
11
12     unsigned char dato; // Dato a leer desde
13     char cadena[101];
14     unsigned int n= 0;
15
16
17     // Inicialización del puerto UART con la velocidad
18     // en baudios del puerto, y la velocidad del procesador
19     uart_init( UART_BAUD_SELECT(UART_BAUD_RATE,F_CPU) );
20
21     // Activación de las interrupciones hardware para
22     // control del puerto serie
23     sei();
24
25
26     // Inicializamos el fotorreceptor
27     initLaserReceiver();
```

Inicialización del fotorreceptor.
¡OJO!: Debe estar conectado en el
pin 8 de la placa.

– Fichero receptorArdu.cpp (II):

- Bucle infinito que envía cadenas de 100 bytes útiles

```
28
29     while (1) {
30         n= 0;
31         // rellenamos el buffer
32         for (int i= 0; i<100; i++) {
33
34             // Recibimos dato
35             recvLaserBit(dato);
36
37             // Si es ráfaga corta, incluimos en el buffer un 0
38             if (dato == LASER_DOT)
39                 cadena[n]= '0';
40
41             // Si es ráfaga larga, incluimos en el buffer un 1
42             else if (dato == LASER_DASH)
43                 cadena[n]= '1';
44
45             // Si no se recibe nada, incluimos en el buffer un ?
46             else
47                 cadena[n]= '?';
48
49             n++; // Pasamos al siguiente carácter en cadena
50     }
51
52     // Enviamos el mensaje al PC
53     cadena[n]= '\0';
54     arduSendUSB(cadena);
55 }
56
57 return 0;
58 }
```

– Fichero receptorArdu.cpp (II):

- Bucle infinito que envía cadenas de 100 bytes útiles

```
28
29     while (1) {
30         n= 0;
31         // rellenamos el buffer
32         for (int i= 0; i<100; i++) {
33
34             // Recibimos dato
35             recvLaserBit(dato);
36
37             // Si es ráfaga corta, incluimos en el buffer un 0
38             if (dato == LASER_DOT)
39                 cadena[n]= '0';
40
41             // Si es ráfaga larga, incluimos en el buffer un 1
42             else if (dato == LASER_DASH)
43                 cadena[n]= '1';
44
45             // Si no se recibe nada, incluimos en el buffer un ?
46             else
47                 cadena[n]= '?';
48
49             n++; // Pasamos al siguiente carácter en cadena
50     }
51
52     // Enviamos el mensaje al PC
53     cadena[n]= '\0';
54     arduSendUSB(cadena);
55 }
56
57 return 0;
58 }
```

Inicialmente, “cadena” está vacío.

– Fichero receptorArdu.cpp (II):

- Bucle infinito que envía cadenas de 100 bytes útiles

```
28  while (1) {  
29      n= 0;  
30      // rellenamos el buffer  
31      for (int i= 0; i<100; i++) {  
32          // Recibimos dato  
33          recvLaserBit(dato);  
34  
35          // Si es ráfaga corta, incluimos en el buffer un 0  
36          if (dato == LASER_DOT)  
37              cadena[n]= '0';  
38  
39          // Si es ráfaga larga, incluimos en el buffer un 1  
40          else if (dato == LASER_DASH)  
41              cadena[n]= '1';  
42  
43          // Si no se recibe nada, incluimos en el buffer un ?  
44          else  
45              cadena[n]='?';  
46  
47          n++; // Pasamos al siguiente carácter en cadena  
48      }  
49  
50      // Enviamos el mensaje al PC  
51      cadena[n]= '\0';  
52      arduSendUSB(cadena);  
53  
54  }  
55  
56  return 0;  
57 }  
58 }
```

Rellenamos los 100 caracteres útiles de “cadena”.

— Fichero receptorArdu.cpp (II):

- Bucle infinito que envía cadenas de 100 bytes útiles

```
28  while (1) {  
29      n= 0;  
30      // rellenamos el buffer  
31      for (int i= 0; i<100; i++) {  
32          // Recibimos dato  
33          recvLaserBit(dato);  
34  
35          // Si es ráfaga corta, incluimos en el buffer un 0  
36          if (dato == LASER_DOT)  
37              cadena[n]= '0';  
38  
39          // Si es ráfaga larga, incluimos en el buffer un 1  
40          else if (dato == LASER_DASH)  
41              cadena[n]= '1';  
42  
43          // Si no se recibe nada, incluimos en el buffer un ?  
44          else  
45              cadena[n]='?';  
46  
47          n++; // Pasamos al siguiente carácter en cadena  
48      }  
49  
50      // Enviamos el mensaje al PC  
51      cadena[n]= '\0';  
52      arduSendUSB(cadena);  
53  
54  }  
55  
56  return 0;  
57 }  
58 }
```

Recibimos en “dato” qué se está detectando por el fotorreceptor.

— Fichero receptorArdu.cpp (II):

- Bucle infinito que envía cadenas de 100 bytes útiles

```
28  while (1) {  
29      n= 0;  
30      // rellenamos el buffer  
31      for (int i= 0; i<100; i++) {  
32          // Recibimos dato  
33          recvLaserBit(dato);  
34  
35          // Si es ráfaga corta, incluimos en el buffer un 0  
36          if (dato == LASER_DOT)  
37              cadena[n]= '0';  
38  
39          // Si es ráfaga larga, incluimos en el buffer un 1  
40          else if (dato == LASER_DASH)  
41              cadena[n]= '1';  
42  
43          // Si no se recibe nada, incluimos en el buffer un ?  
44          else  
45              cadena[n]='?';  
46  
47          n++; // Pasamos al siguiente carácter en cadena  
48      }  
49  
50      // Enviamos el mensaje al PC  
51      cadena[n]= '\0';  
52      arduSendUSB(cadena);  
53  }  
54  
55  return 0;  
56 }  
57 }
```

Si es una ráfaga corta, ponemos el carácter '0' en el buffer

— Fichero receptorArdu.cpp (II):

- Bucle infinito que envía cadenas de 100 bytes útiles

```
28  while (1) {  
29      n= 0;  
30      // rellenamos el buffer  
31      for (int i= 0; i<100; i++) {  
32          // Recibimos dato  
33          recvLaserBit(dato);  
34  
35          // Si es ráfaga corta, incluimos en el buffer un 0  
36          if (dato == LASER_DOT)  
37              cadena[n]= '0';  
38  
39          // Si es ráfaga larga, incluimos en el buffer un 1  
40          else if (dato == LASER_DASH)  
41              cadena[n]= '1';  
42  
43          // Si no se recibe nada, incluimos en el buffer un ?  
44          else  
45              cadena[n]='?';  
46  
47          n++; // Pasamos al siguiente carácter en cadena  
48      }  
49  
50      // Enviamos el mensaje al PC  
51      cadena[n]= '\0';  
52      arduSendUSB(cadena);  
53  
54  }  
55  
56  return 0;  
57 }  
58 }
```

Si es una ráfaga larga, ponemos el carácter '1' en el buffer

— Fichero receptorArdu.cpp (II):

- Bucle infinito que envía cadenas de 100 bytes útiles

```
28  while (1) {  
29      n= 0;  
30      // rellenamos el buffer  
31      for (int i= 0; i<100; i++) {  
32          // Recibimos dato  
33          recvLaserBit(dato);  
34  
35          // Si es ráfaga corta, incluimos en el buffer un 0  
36          if (dato == LASER_DOT)  
37              cadena[n]= '0';  
38  
39          // Si es ráfaga larga, incluimos en el buffer un 1  
40          else if (dato == LASER_DASH)  
41              cadena[n]= '1';  
42  
43          // Si no se recibe nada, incluimos en el buffer un ?  
44          else  
45              cadena[n]='?';  
46  
47          n++; // Pasamos al siguiente carácter en cadena  
48      }  
49  
50      // Enviamos el mensaje al PC  
51      cadena[n]= '\0';  
52      arduSendUSB(cadena);  
53  
54  }  
55  
56  
57  return 0;  
58 }
```

Si no se detectan ráfagas, ponemos el carácter '?' en el buffer

— Fichero receptorArdu.cpp (II):

- Bucle infinito que envía cadenas de 100 bytes útiles

```
28
29     while (1) {
30         n= 0;
31         // rellenamos el buffer
32         for (int i= 0; i<100; i++) {
33
34             // Recibimos dato
35             recvLaserBit(dato);
36
37             // Si es ráfaga corta, incluimos en el buffer un 0
38             if (dato == LASER_DOT)
39                 cadena[n]= '0';
40
41             // Si es ráfaga larga, incluim
42             else if (dato == LASER_DASH)
43                 cadena[n]= '1';
44
45             // Si no se recibe nada, incluimos en el buffer un ?
46             else
47                 cadena[n]= '?';
48
49             n++; // Pasamos al siguiente carácter en cadena
50     }
51
52     // Enviamos el mensaje al PC
53     cadena[n]= '\0';
54     arduSendUSB(cadena);
55 }
56
57 return 0;
58 }
```

Pasamos a llenar el siguiente carácter del buffer.

— Fichero receptorArdu.cpp (II):

- Bucle infinito que envía cadenas de 100 bytes útiles

```
28
29     while (1) {
30         n= 0;
31         // rellenamos el buffer
32         for (int i= 0; i<100; i++) {
33
34             // Recibimos dato
35             recvLaserBit(dato);
36
37             // Si es ráfaga corta, incluimos en el buffer un 0
38             if (dato == LASER_DOT)
39                 cadena[n]= '0';
40
41             // Si es ráfaga larga, inclu
42             else if (dato == LASER_DASH)
43                 cadena[n]= '1';
44
45             // Si no se recibe nada, inc
46             else
47                 cadena[n]= '?';
48
49             n++; // Pasamos al siguiente carácter en cadena
50     }
51
52     // Enviamos el mensaje al PC
53     cadena[n]= '\0';
54     arduSendUSB(cadena);
55 }
56
57 return 0;
58 }
```

Al llenar los 100 bytes, terminamos poniendo en la última posición del buffer el carácter fin de cadena y lo enviamos por USB.

– Makefile del fichero receptorArdu.cpp:

```
COMMPORTRECEPTOR = /dev/ttyACM0

receptorArdu: ticcommarducpp uartcpp receptorArduCPP
    @echo Generando binario Arduino receptor...
    avr-gcc -mmcu=atmega328p obj/ticcommardu.o obj/uart.o obj/receptorArdu.o -o obj/receptorArdu.bin
    @echo Generando HEX Arduino receptor...
    avr-objcopy -O ihex -R .eeprom obj/receptorArdu.bin hex/receptorArdu.hex
    @echo receptorArdu compilado. Ejecute make sendReceptor para enviarlo a la plataforma Arduino.

receptorArduCPP:
    @echo Compilando receptorArdu...
    avr-gcc -Os -mmcu=atmega328p -c -o obj/receptorArdu.o src/receptorArdu.cpp -Iinclude

sendReceptor:
    @echo Enviando receptorArdu a Arduino
    sudo avrdude -F -V -c arduino -p ATMEGA328P -P $(COMMPORTRECEPTOR) -b 115200 -U flash:w:hex/receptorArdu.hex

ticcommarducpp:
    @echo Compilando TICCommArdu...
    avr-gcc -Os -mmcu=atmega328p -c -o obj/ticcommardu.o src/ticcommardu.cpp -Iinclude

uartCPP:
    @echo Compilando biblioteca UART...
    avr-gcc -Os -mmcu=atmega328p -c -o obj/uart.o src/uart.cpp -Iinclude
```

- Debemos compilar:
 - El receptor: **make receptorArdu**
 - El emisor: **make emisorArdu**
 - El receptor en el PC: **make receptorPC**
 - Enviar el receptor: **make sendReceptor**
 - Enviar el emisor: **make sendEmisor**
- También debemos asegurarnos de que:
 - El receptor se encuentra en una zona sin iluminación (o muy poca)
 - El emisor láser está alineado con el fotorreceptor, para que este pueda recibir la información

- Al ejecutar el receptorPC, encontramos la siguiente salida:

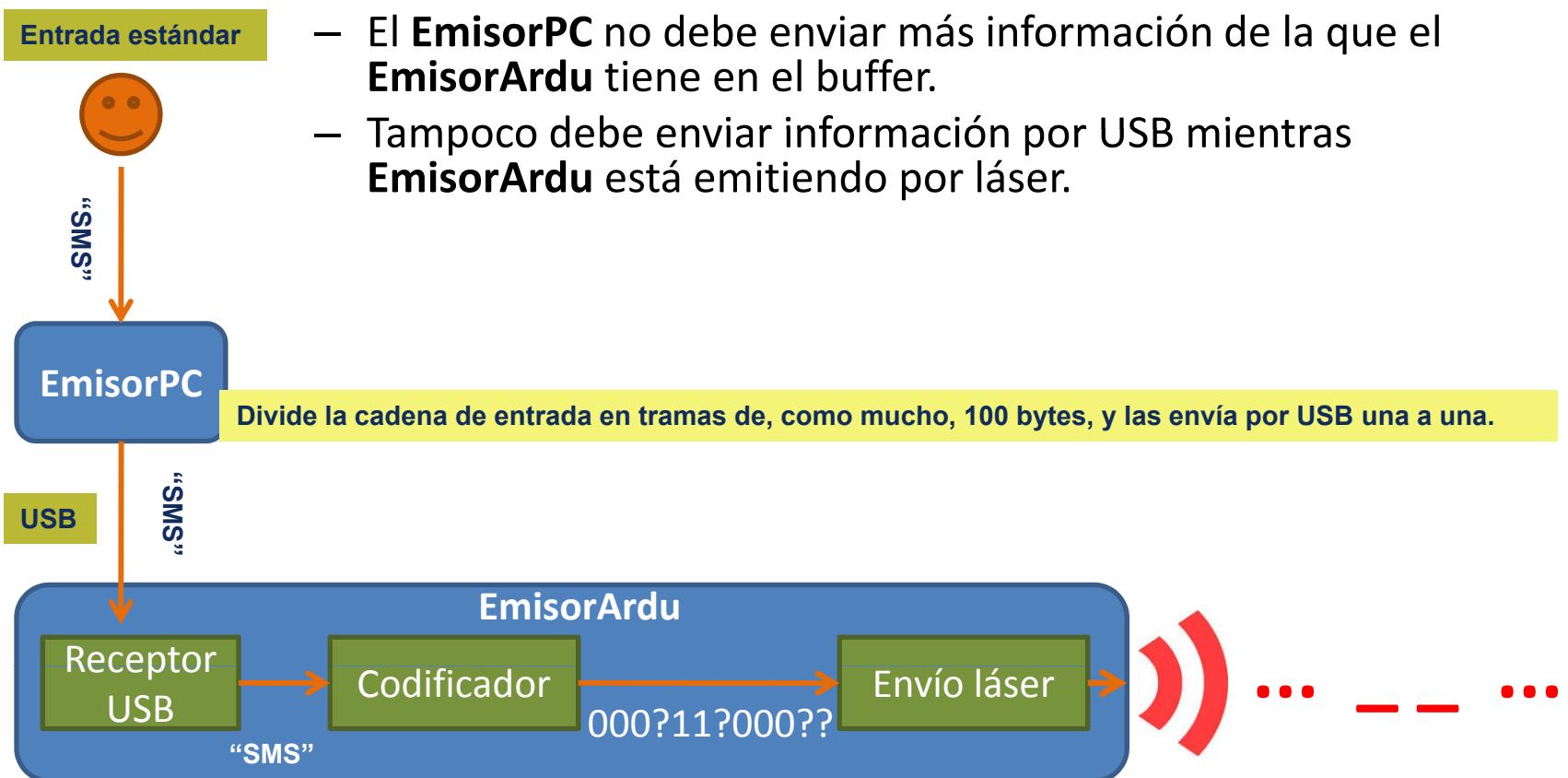
```
manupc@manupcws: ~/Dropbox/docencia/TIC/Practicas/Seminario1/codigos/Sesion4/emis or$ ./bin/receptorPC
Mensaje recibido: 00?000?11?000?000?11?000?000?11?000?000?11?000?000?11?000?000?
11?000?000?11?000?000?11?000?000?11?000?000?11?000?000?11?000?000?11?000?000?
Mensaje recibido: 000?11?000?000?11?000?000?11?000?000?11?000?000?11?000?000?11?
000?000?11?000?000?11?000?000?11?000?000?0
Mensaje recibido: ?11?000?000?11?000?000?11?000?000?11?000?000?11?000?000?11?000?
?000?11?000?000?11?000?000?11?000?000?
Mensaje recibido: 0?000?000?11?000?000?11?000?000?11?000?000?11?000?000?11?000?000?
00?11?000?000?11?000?000?11?000?000?11
Mensaje recibido: 00?000?11?000?000?11?000?000?11?000?000?11?000?000?11?000?000?11?000?000?
11?000?000?11?000?000?11?000?000?11?000?
^C
manupc@manupcws: ~/Dropbox/docencia/TIC/Practicas/Seminario1/codigos/Sesion4/emis or$
```

SESIÓN 5

- 1. ¿Qué es Arduino?**
- 2. El procesador**
- 3. Compilador y bibliotecas para procesadores AVR**
- 4. Mi primer programa para AVR AtMega328p**
- 5. Comunicaciones en serie**
- 6. Mi primer programa para comunicaciones en serie: ECHO**
- 7. Recepción de datos desde sensores fotorresistores digitales**
- 8. Envío de datos por dispositivos emisores láser**
- 9. La arquitectura del sistema de emisión/recepción de datos por láser**
- 10. Ampliación de la biblioteca <ticcommardu.h>**
- 11. Ejemplo: Envío de datos por láser**
- 12. Ejemplo: Recepción de datos por láser**
- 13. Proyecto final de la práctica**

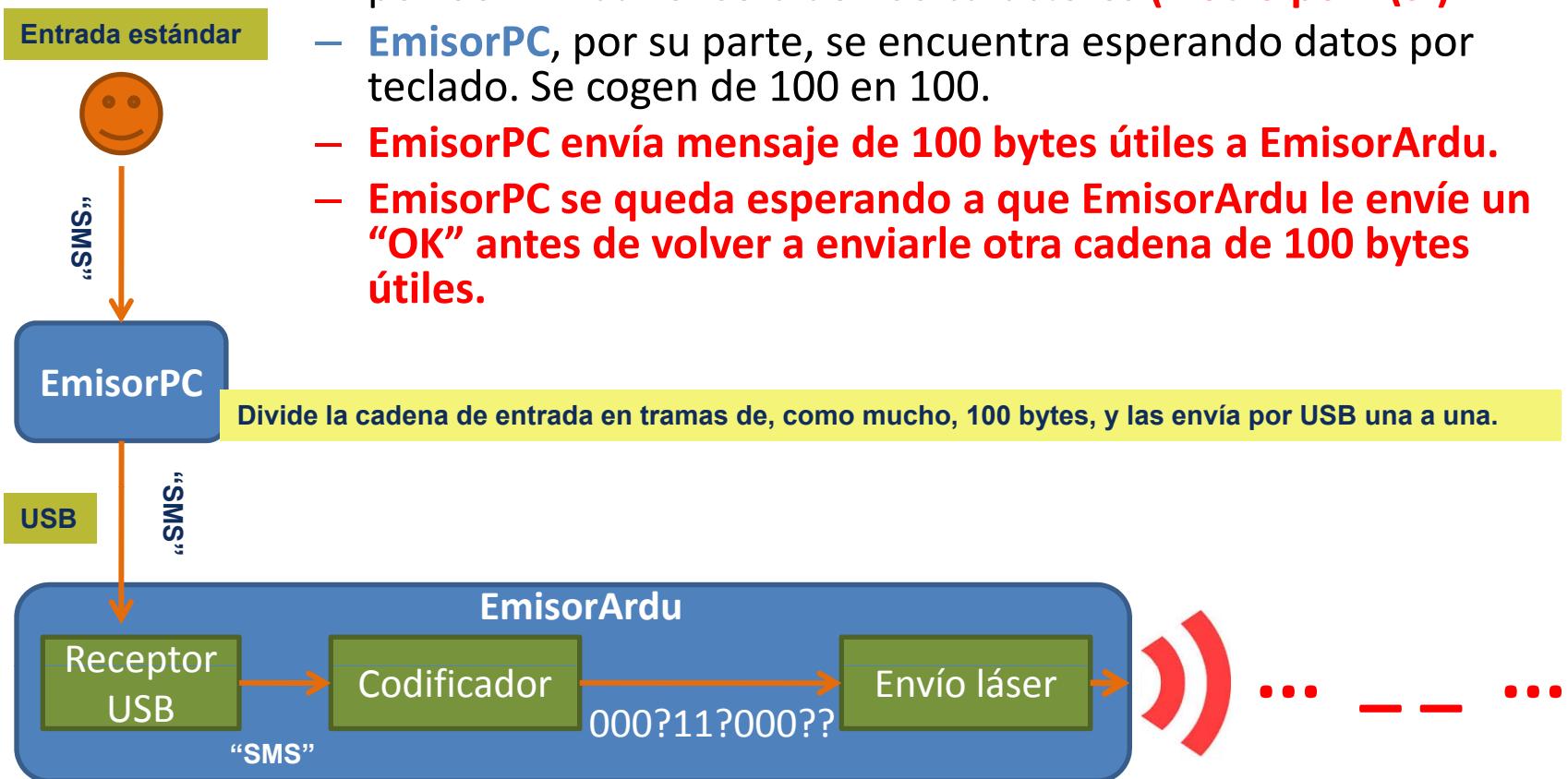
- El **proyecto final de la práctica** consiste en terminar de construir las bibliotecas básicas de comunicaciones para poder implementar códigos en los siguientes seminarios.
- Se debe construir un sistema compuesto por 4 programas o aplicaciones:
 - **EmisorPC:** programa que envía un texto a una placa emisora Arduino Uno.
 - **EmisorArdu:** programa que recibe un texto por USB, lo codifica y lo envía a través de láser.
 - **ReceptorArdu:** programa de una placa receptora Arduino Uno que recibe un mensaje a través del fotorreceptor láser, lo decodifica en texto y lo envía por USB.
 - **ReceptorPC:** programa que recibe un mensaje de texto por USB y lo muestra por pantalla.

- Para realizar el proyecto final, es necesario considerar que todas las aplicaciones deben estar bien coordinadas para obtener los resultados esperados:

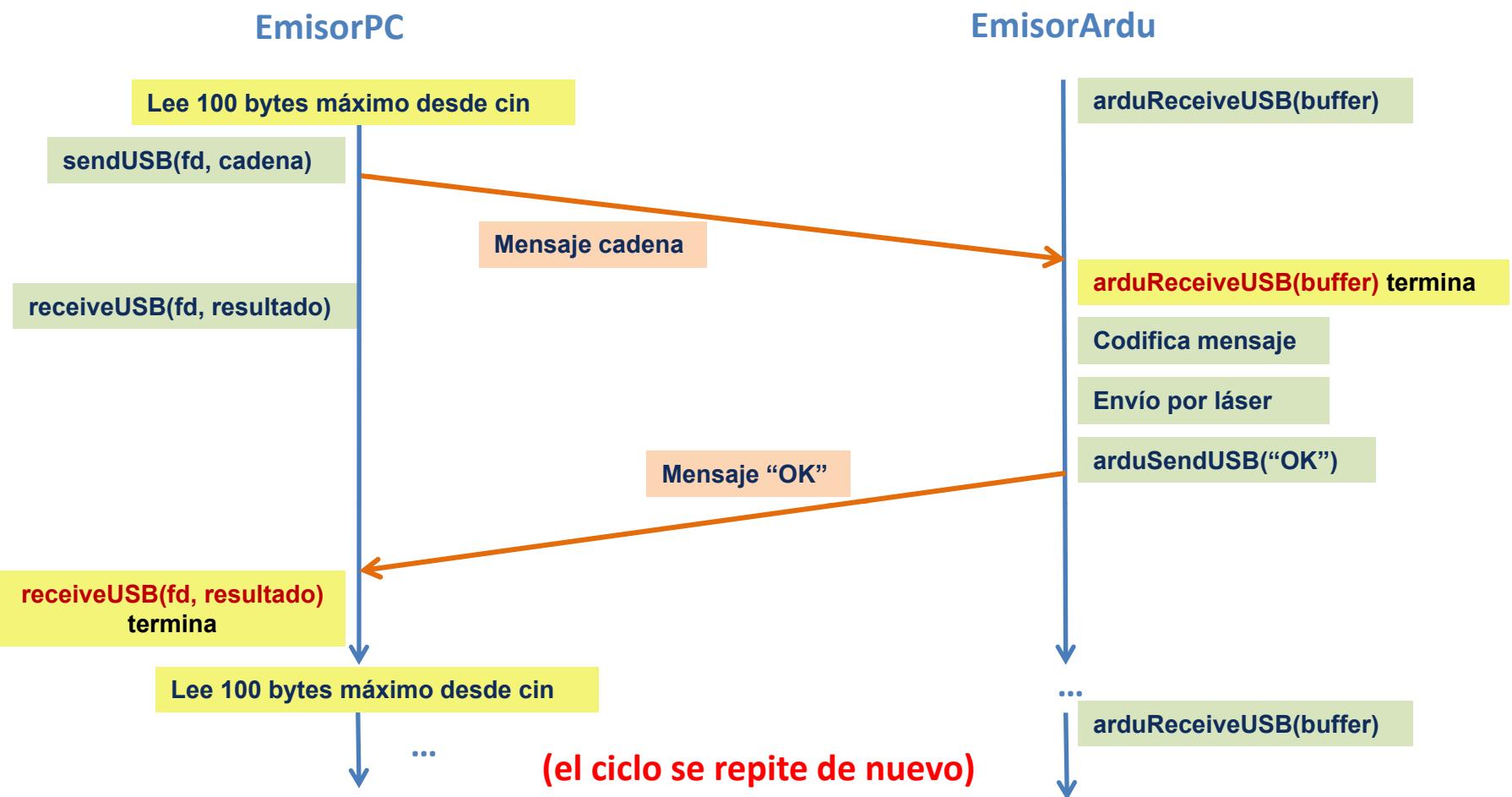


– Cómo coordinar el EmisorPC con el EmisorArdu:

- Inicialmente, **EmisorArdu** se encuentra esperando un mensaje por USB. El buffer será de 100 caracteres (**+ otro por '\0'**)
- **EmisorPC**, por su parte, se encuentra esperando datos por teclado. Se cogen de 100 en 100.
- **EmisorPC envía mensaje de 100 bytes útiles a EmisorArdu.**
- **EmisorPC se queda esperando a que EmisorArdu le envíe un “OK” antes de volver a enviarle otra cadena de 100 bytes útiles.**



- Diagrama del protocolo de comunicaciones entre EmisorPC y EmisorArdu:



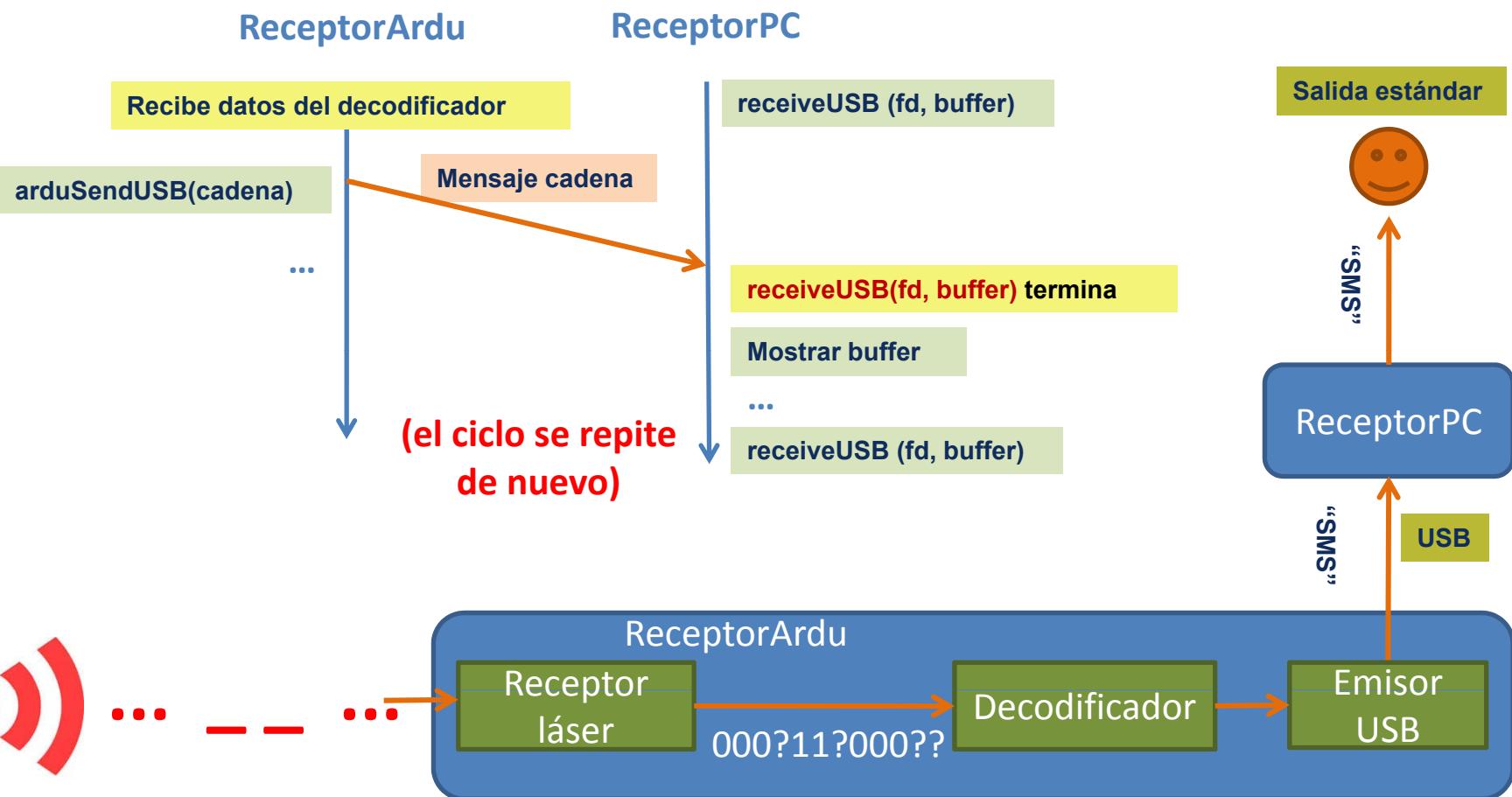
- Para realizar el proyecto final, es necesario considerar que todas las aplicaciones deben estar bien coordinadas para obtener los resultados esperados:
 - El **ReceptorPC** debe tener configurado el buffer de entrada con el mismo tamaño de envío del buffer de salida por USB de **ReceptorArdu**.
 - El **EmisorArdu** debe coordinarse bien con **ReceptorArdu**, para evitar que se pierda parte del mensaje o que se mezcle con basura:

```
manupc@manupcws: ~/Dropbox/docencia/TIC/Practicas/Seminario1/codigos/Sesion4/emisor$ ./bin/receptorPC
Writing | #####
avrdude: 970 bytes of flash written
avrdude: safemode: Fuses OK (E:00, H:00, L:00)
avrdude done. Thank you.

manupc@manupcws:~/Dropbox/docencia/TIC/Practicas/Seminario1/codigos/Sesion4/emisor$ ./bin/receptorPC
Mensaje recibido: 00?000?11?000?000?11?000?000?11?000?000?11?000?000?
11?000?000?11?000?000?11?000?000?11?00
Mensaje recibido: 000?21?000?000?21?000?000?11?000?000?.11?000?000?11?000?000?11?
00?000?11?000?000?11?000?000?
Mensaje recibido: ?11?000?000?11?000?000?11?000?000?11?000?000?11?000?000?11?000?
00?11?000?000?11?000?000?11?000?000?
Mensaje recibido: 0?000?000?11?000?000?11?000?000?11?000?000?11?000?000?11?000?000?
0?11?000?000?11?000?000?11?000?000?11?
Mensaje recibido: 00?000?11?000?000?11?000?000?11?000?000?11?000?000?11?000?000?
11?000?000?11?000?000?11?000?000?11?000?000?
^C
manupc@manupcws:~/Dropbox/docencia/TIC/Practicas/Seminario1/codigos/Sesion4/emisor$
```

¡AQUÍ se ha comenzado a leer el mensaje tarde! Se recibe "partido"

- Cómo coordinar el ReceptorArdu con el ReceptorPC:
- Es la opción más simple:



– Cómo coordinar el EmisorArdu con el ReceptorArdu:

- **La clave está en el receptor:** Saber cuándo comienza el mensaje y cuándo acaba.
- Diseñaremos el receptor para que pueda estar en dos estados: “Recibiendo” o “En Espera”.
- **Fácil de implementar con una variable bool**
- Mientras el receptor esté en espera, el buffer de entrada de datos por láser (llamémosle **Laser**) estará vacío (componentes útiles **nLaser=0**).
- El tamaño máximo del buffer **Laser** debe ser mínimo de 100 para poder almacenar un máximo de 100 símbolos recibidos, dado que este valor es el que hemos utilizado para el emisor.

- **Inicialmente, ReceptorArdu se encuentra en el estado “En Espera”.**

- Cómo coordinar el EmisorArdu con el ReceptorArdu:
 - Transiciones entre estados (ReceptorArdu):
 - Del estado “En Espera” a “Recibiendo”:
 - Cuando se esté en espera y se reciba un dato por láser distinto de LASER_NONE, se pasará al estado “Recibiendo” y se llenará el buffer.
 - Del estado “Recibiendo” a “En Espera” :
 - Cuando se reciba dos LASER_NONE seguidos, se pasará al estado “En espera” y:
 - Se decodificará el mensaje recibido
 - Se enviará por USB al receptorPC
 - Se inicializará el tamaño del buffer a 0 para prepararnos a recibir otra trama de datos.

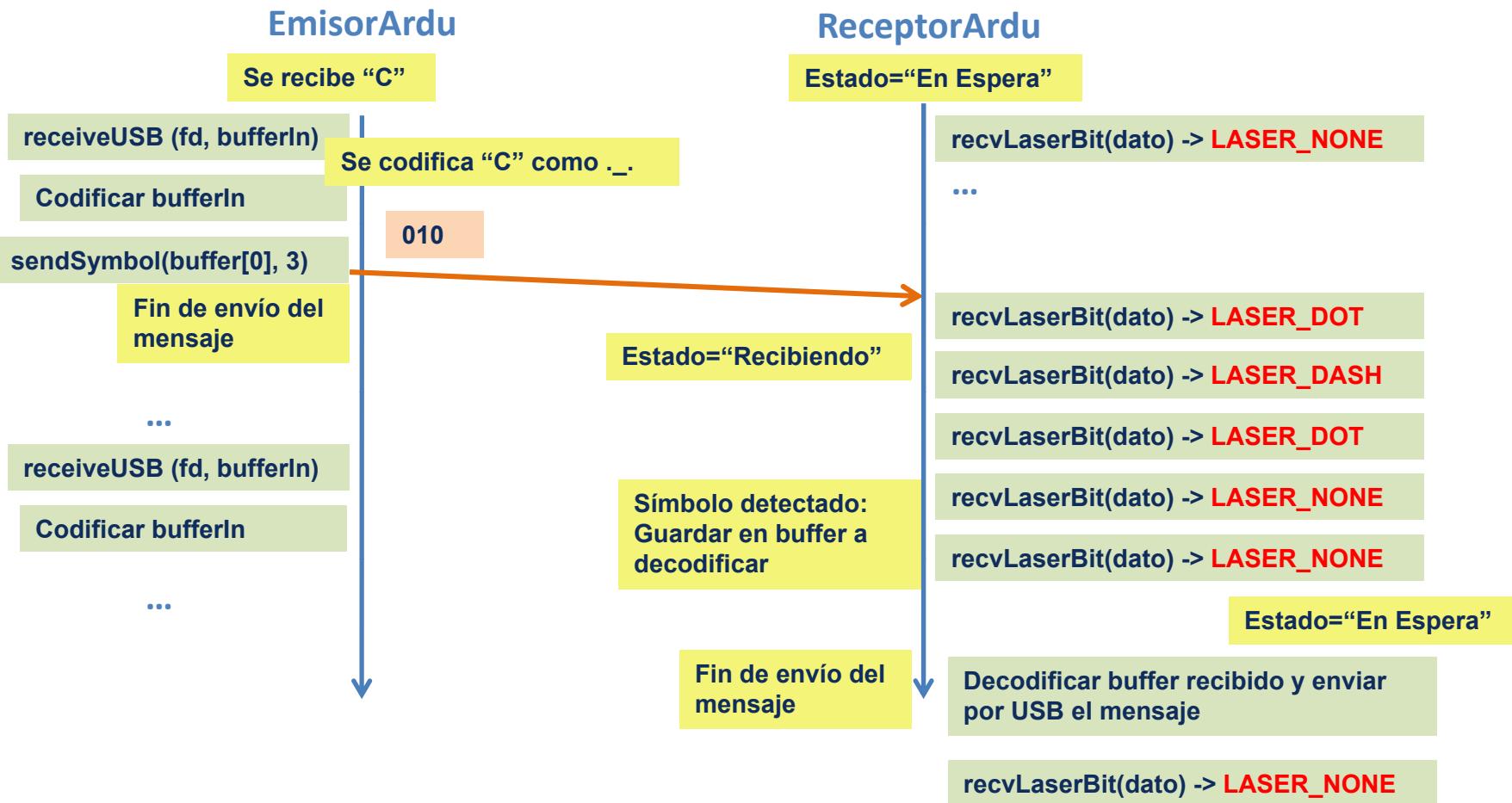
– Cómo coordinar el EmisorArdu con el ReceptorArdu:

- Vamos a ver dos ejemplos.
- Supongamos que tenemos un alfabeto de 4 símbolos “A”, “B”, “C”, y “D”.
- Asumiremos que codificador realiza las siguientes codificaciones sobre el alfabeto:

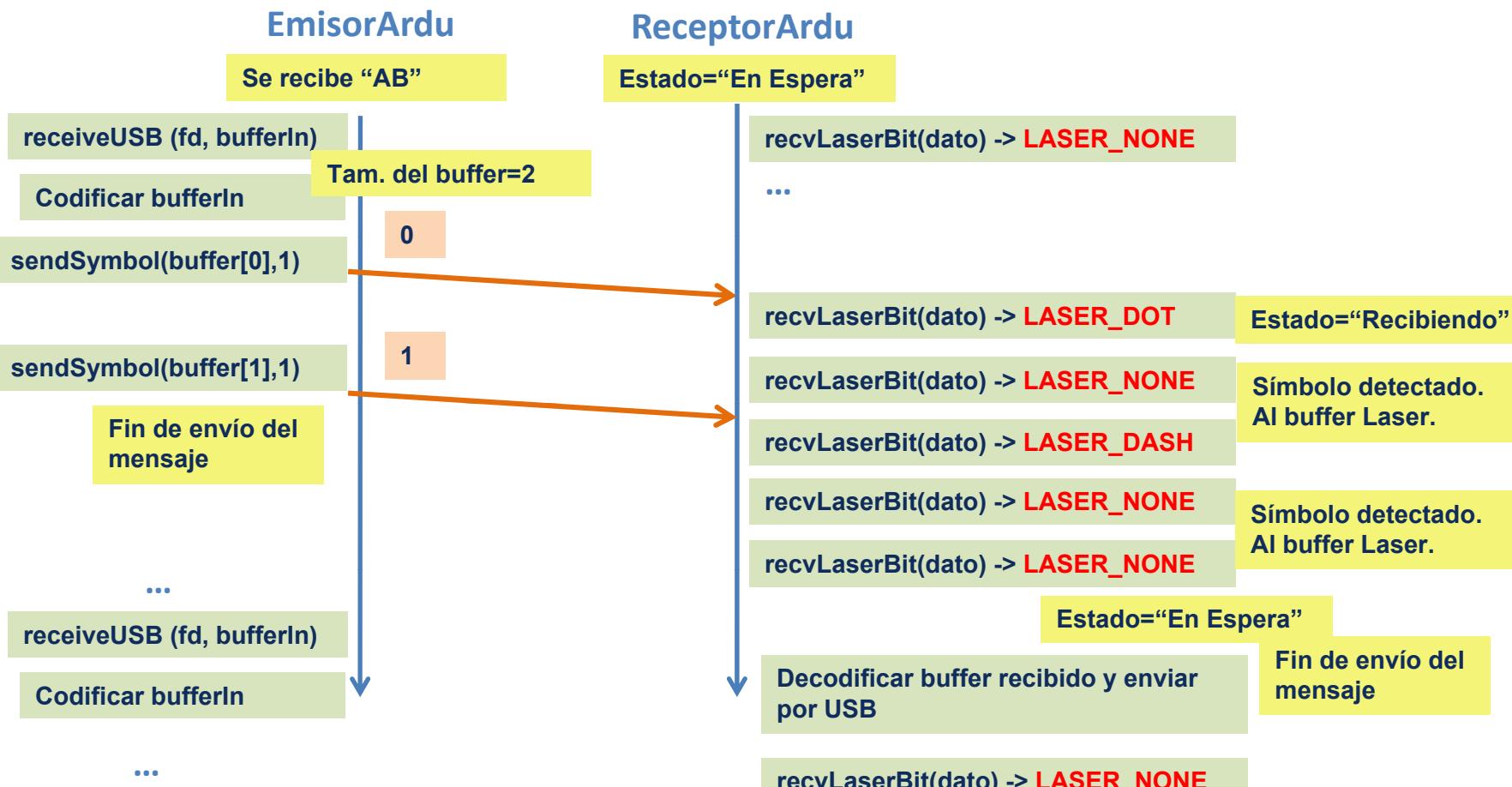
Símbolo	Codificación
“A”	.
“B”	-
“C”	._.
“D”	..

- En el ejemplo 1, enviaremos sólo el mensaje de la cadena “C”.
- En el ejemplo 2, enviaremos el mensaje de la cadena “AB”.

– Ejemplo de coordinación para el envío de “C”



– Ejemplo de coordinación para el envío de “AB”



– Cuestiones adicionales:

- Las próximas prácticas estarán orientadas íntegramente a los módulos de codificación y decodificación.
- Para poder reutilizar el código en las próximas prácticas, debemos hacer un diseño simple y flexible, que nos permita reutilizar los prototipos de las funciones en las prácticas futuras, cambiando únicamente la implementación del codificador y del decodificador.
- Por tanto, asumiremos que tendremos 2 funciones para codificar y 2 funciones para decodificar. Se deberán implementar con los prototipos indicados a continuación.

- Codificador:

- Codificador de un mensaje:

- ENTRADAS: Debe tener un mensaje a codificar como entrada (llamémosle **orig**), junto con su tamaño útil (llamémosle **nOrig**).

- SALIDAS:

- Debe tener un mensaje codificado como salida (llamémosle **codif**). El tamaño de **codif** será el mismo que el de **orig**.

- Durante la codificación, **orig[0]** se guardará codificado en **codif[0]**, **orig[1]** en **codif[1]**, etc.

- Debe tener un vector (llamémosle **util**) para indicar cuántos bits útiles hay codificados en cada componente **i** de **codif[i]**.

- Ejemplo de entradas y salidas para codificar “ABDC”:

Orig-> {‘A’, ‘B’, ‘C’, ‘D’}

Codif -> {XXXXXXX**0** , XXXXXXX**1** , XXXXX**010** , XXXXX**00**}

Tam -> {1, 1, 3, 2}

NOTA: Se ha marcado con X los bits que no se utilizan en el vector de datos codificados

- Codificador:

- Prototipo del codificador:

```
void codificador(const char *orig, const int nOrig, char *codif,  
                 unsigned char *util);
```

- Para simplificar, además utilizaremos otra función que codifique un símbolo únicamente:
 - **ENTRADAS:** Un símbolo a codificar (**corig**)
 - **SALIDAS:**
 - Un byte con la codificación del símbolo (**ccodif**)
 - Un byte sin signo con el número de bits usados para codificar el símbolo en **ccodif** (**nUtil**).

```
void codificaSimbolo(const char corig, char &ccodif,  
                      unsigned char &nUtil);
```

- Así, la **implementación** del codificador se reduce a un bucle que llama a **codificaSímbolo** por cada una de las componentes del mensaje a codificar.

- Decodificador:

- Decodificador de un mensaje:

- **ENTRADAS:** Debe tener un mensaje a decodificar como entrada (llamémosle **codif**), junto con su tamaño útil (llamémosle **nCodif**) y un vector con el número de bits útiles (llamémosle **utiles**) de cada símbolo a decodificar.

- **SALIDAS:**

- Debe tener un mensaje decodificado como salida (llamémosle **decodif**). El tamaño de **decodif** será el mismo que el de **codif**. Durante la decodificación, **codif[0]** se guardará decodificado en **decodif[0]**, **codif[1]** en **decodif[1]**, etc.

- Ejemplo de entradas y salidas para decodificar {0, 1, 010, 00}:

Codif -> {XXXXXXX**0** , XXXXXXX**1** , XXXXX**010** , XXXXXX**00**}

utiles -> {1, 1, 3, 2}

decodif-> {'A', 'B', 'C', 'D'}

NOTA: Se ha marcado con X los bits que no se utilizan en el vector de datos codificados

- Decodificador:
 - Prototipo del decodificador:

```
void decodificador(const char *codif, const unsigned char *utiles,  
                   const int nCodif, char *decodif);
```

- Para simplificar, además utilizaremos otra función que decodifique un símbolo únicamente:
 - **ENTRADAS:**
 - Un símbolo a decodificar (**ccodif**)
 - El número de bits útiles del símbolo (**nUtils**)
 - **SALIDAS:**
 - Un byte con la decodificación del símbolo (**decodif**)

```
void decodificaSímbolo(const char ccodif, const unsigned char nUtils,  
                       char &decodif);
```

- Así, la **implementación** del decodificador se reduce a un bucle que llama a **decodificaSímbolo** por cada una de las componentes del mensaje a codificar.

- Qué hay que hacer en el proyecto de la práctica actual:
 - Crear una nueva biblioteca <arducodif.h> que contenga las 4 funciones para codificar y decodificar.
 - Implementar la codificación y decodificación de símbolos del código Morse.
 - Implementar los mecanismos de coordinación estudiados entre:
 - emisorPC y emisorArdu
 - emisorArdu y receptorArdu
 - receptorArdu y receptorPC
 - Obtener una **versión funcional** que permita enviar mensajes en código Morse desde un **emisorPC** y visualizarlos en un **receptorPC**.



decsai.ugr.es

Universidad de Granada

Teoría de la Información y la Codificación

Grado en Ingeniería Informática

Seminario 1.- Introducción a Arduino. Diseño y construcción de plataforma para transmisión de datos por láser.



**Departamento de Ciencias de la
Computación e Inteligencia Artificial**