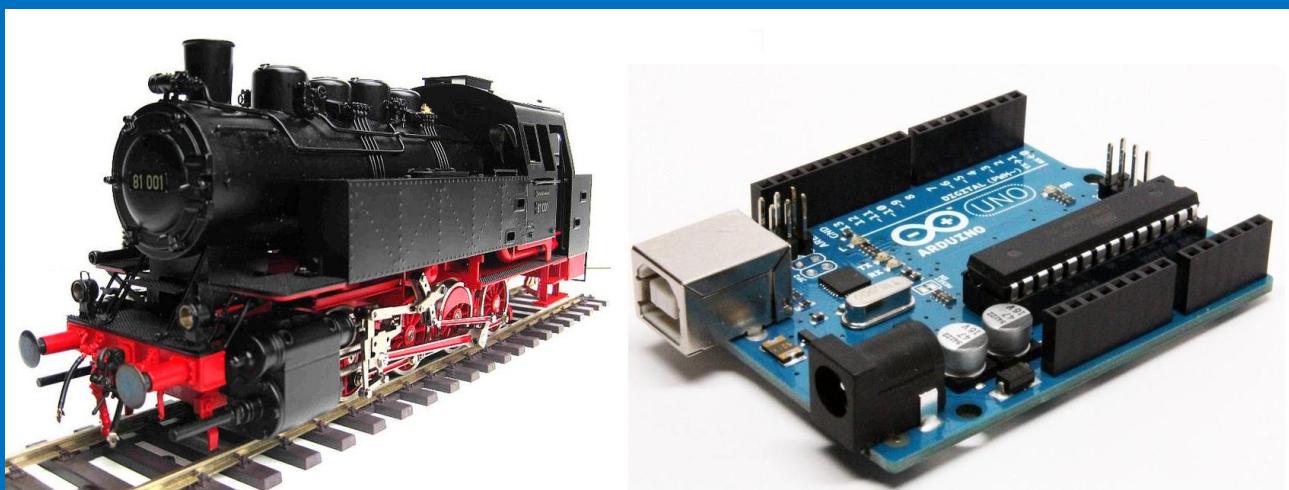




Lenguaje de programación Arduino

Guía práctica de iniciación aplicada al
Modelismo Ferroviario



Paco Cañada

Esta pequeña guía práctica surge como un hilo sobre "Nociones básicas para programar en C del Arduino o entender los listados", en el foro CTMS (Control Tren Modelo por Software - <http://ctms1.com/>). Se ha trasladado aquí el contenido de ese hilo en forma de libro pensando en los aficionados al modelismo ferroviario que desean acercarse a la programación de Arduino para aplicarlo en sus maquetas.

Paco Cañada
<http://usuaris.tinet.cat/fmco/>

Primera edición: marzo, 2020
Segunda edición: marzo, 2020
Tercera edición: marzo, 2020
Cuarta edición: marzo, 2020
Quinta edición: abril, 2020
Sexta edición: abril, 2020
Séptima edición: mayo, 2020
Octava edición: agosto, 2020
Novena edición: octubre, 2020
Décima edición: octubre, 2020
Undécima edición: febrero, 2022
Duodécima edición: agosto, 2022

(21 julio 2019 al 29 de febrero de 2020)
[\(21 julio 2019 al 21 de marzo de 2020\)](#)
(21 julio 2019 al 26 de marzo de 2020)
(21 julio 2019 al 31 de marzo de 2020)
(21 julio 2019 al 10 de abril de 2020)
(21 julio 2019 al 24 de abril de 2020)
(21 julio 2019 al 11 de mayo de 2020)
(21 julio 2019 al 14 de agosto de 2020)
(21 julio 2019 al 16 de octubre de 2020)
(21 julio 2019 al 12 de diciembre de 2020)
(21 julio 2019 al 20 de febrero de 2022)

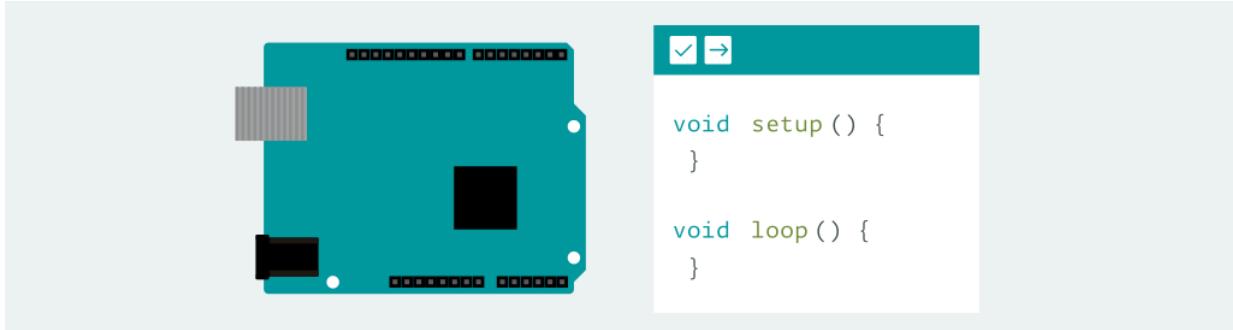
(Comprende las entradas al hilo desde el 21 julio 2019 al 2 de agosto de 2022)

Contenido

Lenguaje de programación Arduino.....	5
1. Nociones básicas	7
2. Ejemplo clásico: Blink.....	9
3. Otro ejemplo simple	11
4. Ejemplo simple para la maqueta: Luces obras.....	13
5. Librerías incluidas en Arduino IDE	15
6. Otras librerías.....	19
7. Otras temporizaciones: Luces de edificio aleatorias.....	21
8. Librerías incompatibles: Placa giratoria por segmento y sonido.	25
9. Uso de 'shields': Plataforma deslizante con motor paso a paso	31
10. Los Strings: Botón Pulsador de Acción	43
11. Comunicaciones (Serie e I2C): Reloj-Pantalla de andén.....	51
12. Bits y PWM: Tren lanzadera analógico	61
13. De la idea al programa: Paso a nivel DCC	71
14. Bits problemáticos: Bits serie y retroseñalización S88.....	87
15. Memoria EEPROM: Decodificador de accesorios multiusos.....	97
16. Interrupciones: Mando XpressNet para panel	119
17. Gestor de tarjetas y las colas: Lanzador de rutas XpressNet	141
18. Sketch en varios archivos: Megafonía para la maqueta	157
19. Crear librería: Decodificador DCC para señales RENFE.....	173
20. Máquina de estados: Control de un funicular	191
21. Baterías y Neopixels: Diorama en un tarro de cristal.....	203
22. Pantalla táctil: TCO Xpressnet táctil.....	213
Anexo I. Referencia al Lenguaje	247
Anexo II. Librerías.....	249

Lenguaje de programación Arduino

Un mini tutorial para programar el Arduino o por lo menos entender los listados.



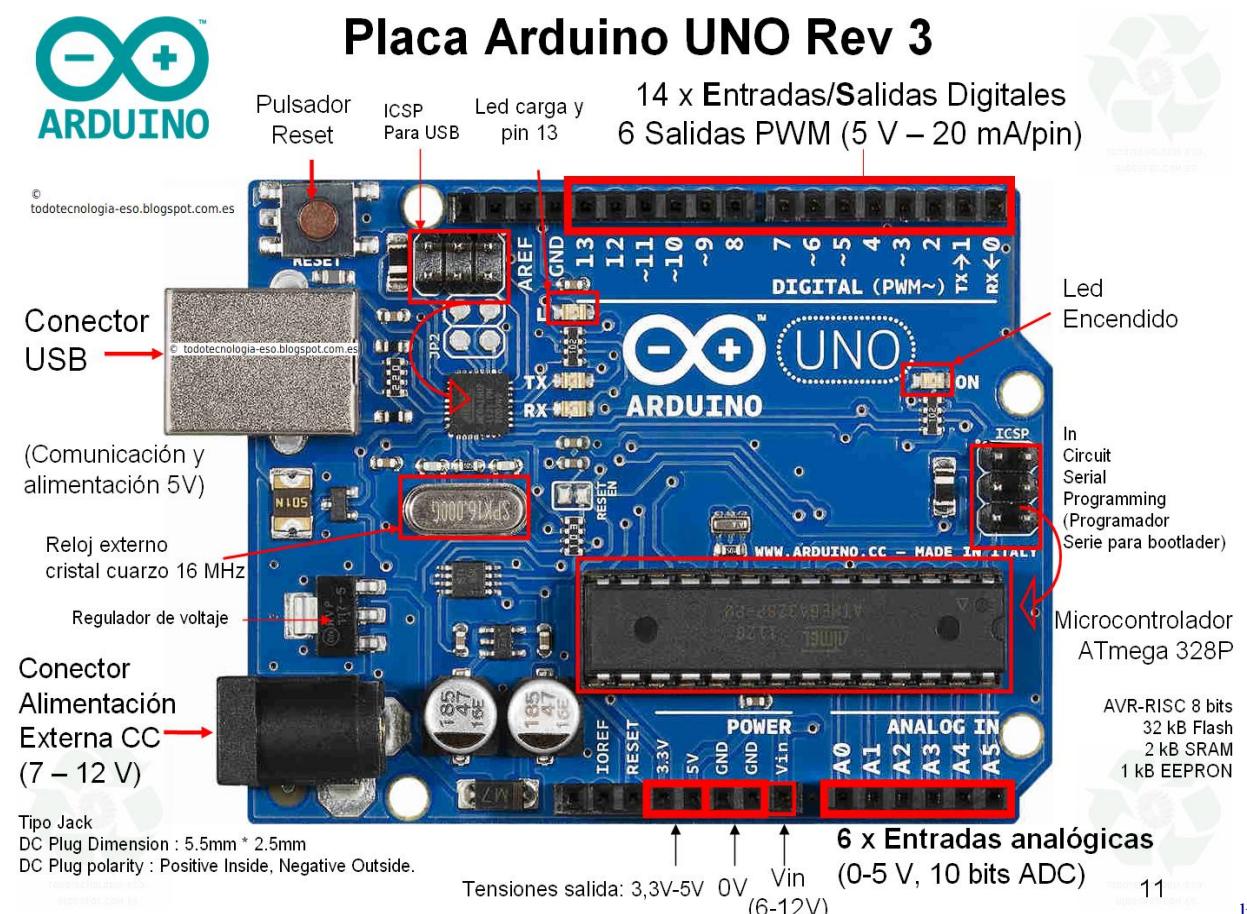
El lenguaje de programación de Arduino que se usa con el Arduino IDE es C++ aunque tiene algunas diferencias ya que deriva de Wiring y de avr-libc

Un programa de Arduino se denomina **sketch** o proyecto y tiene la extensión **.ino**

Importante: para que funcione el sketch, el nombre del fichero debe estar en un directorio con el mismo nombre que el sketch.

Cuando iniciamos el Arduino IDE nos aparece un sketch con la estructura mínima del programa, aquí es donde debemos poner nuestras instrucciones para que el Arduino haga lo que queramos.

La placa más usada es el Arduino Uno, aquí tenéis una descripción de sus conexiones:



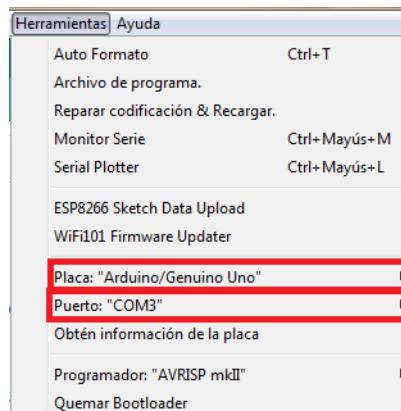
El Arduino IDE lo podemos descargar desde la página oficial de Arduino para nuestro sistema operativo (Windows, Linux & Raspberry Pi y MacOS). Para Windows está disponible la versión instalable con *drivers* y la versión como archivo ZIP en que tendremos que instalar manualmente los *drivers*:

<https://www.arduino.cc/en/software>

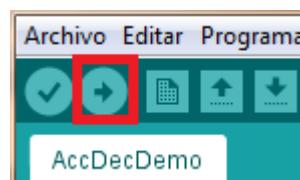
Si usamos clones orientales de placas Arduino es probable que monten el circuito CH340G como interface USB-Serie por lo que para usarlas en entorno Windows tendremos que instalar su correspondiente *driver*.

Desde el mismo Arduino IDE se carga el programa en la placa Arduino, a través de la conexión USB del mismo.

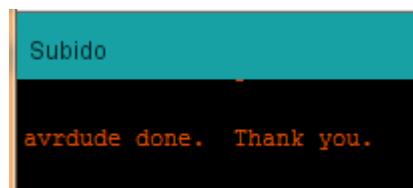
Una vez hemos abierto o completado el **sketch** correspondiente, en el Arduino IDE comprobamos que hemos seleccionado la placa Arduino que tenemos y el puerto al que está conectada. Quizás necesitemos instalar algún *driver* previamente para que nos aparezca listado el puerto al que está conectado.



Para cargar el programa pulsaremos sobre el botón de ‘Subir’ programa.



Tras compilar el programa, si todo va bien nos informará de que ha sido subido.



Empezamos!

Paco

1. Nociones básicas

El lenguaje de Arduino es sensible a mayúsculas y minúsculas (*case sensitive*), no es lo mismo escribir **valor** que **Valor** o **VALOR**

Cada instrucción acaba con ;

Las llaves {} sirven para agrupar instrucciones en diferentes construcciones (funciones, condicionales, bucles, ...). Vigila que cada {} tenga su } según la indentación (sangrado). En el Arduino IDE colocando el cursor junto a } te indica cual es su {

Las // sirven para añadir comentarios al programa. Lo que se escriba detrás de las // en la línea es ignorado

Si los comentarios son más largos de una línea se puede usar /* al iniciar y */ al finalizar el comentario, lo que este entre /* y */ será ignorado

```
// esto es un comentario  
/* esto tambien es un comentario  
pero en varias lineas */
```

Las funciones son piezas de código que dejan un resultado en función de los parámetros de entrada. Como parámetros podemos pasárselos de tipo: **byte**, números enteros **int**, caracteres **char**, números en coma flotante **float** u otro tipo de datos y devolverán un resultado de un determinado tipo o ninguno (**void**)

Data Types	Size in Bytes	Can contain:
boolean	1	true (1) or false (0)
char	1	ASCII character or signed value between -128 and 127
unsigned char, byte, uint8_t	1	ASCII character or unsigned value between 0 and 255
int, short	2	signed value between -32,768 and 32,767
unsigned int, word, uint16_t	2	unsigned value between 0 and 65,535 signed value between -2,147,483,648 and 2,147,483,647
long	4	unsigned value between 0 and 4,294,967,295
unsigned long, uint32_t	4	floating point value between -3.4028235E+38 and 3.4028235E+38 (Note that double is the same as a float on this platform.)
float, double	4	

[link](#)

Hay que definir las funciones y luego se pueden usar en nuestro código. Para definirlas se indica el tipo de dato que devolverán, el nombre de la función y entre () se ponen los nombres de los parámetros que les pasamos precedidos con su tipo y separados por , luego entre {} se escriben las instrucciones que darán el resultado en función de los parámetros.

```
int miFuncion (int parametro1, char parametro2) {  
// aquí van las instrucciones  
}
```

Para usarlas como una instrucción más se escribe su nombre y entre () los parámetros que necesiten separados por , y se finaliza con ; al ser ahora una instrucción.

```
miFuncion (100, 'a');
```

Las variables para almacenar resultados hay que definirlas con el tipo de datos previamente, si se usa una variable sin haber definido su tipo dará error ya que no sabrá qué tipo de datos guardar. Se define escribiendo el tipo, su nombre y ; o si se quiere inicializar con un valor: el tipo, su nombre = y el valor compatible con el tipo y para finalizar ;

```
int miVariable;  
int miOtraVariable = 5;
```

Si se definen al principio, fuera de las funciones son variables globales que se pueden usar en todo el programa. Si se definen dentro de una función son variables locales que solo se utilizan en esa función, por lo que puede haber varias variables con el mismo nombre pero cada una se tendrá en cuenta solo en la función en que ha sido definida.

Las constantes son como las variables pero su valor no puede ser cambiado, se definen como las variables pero precedidas de **const**

```
const int miConstante = 3;
```

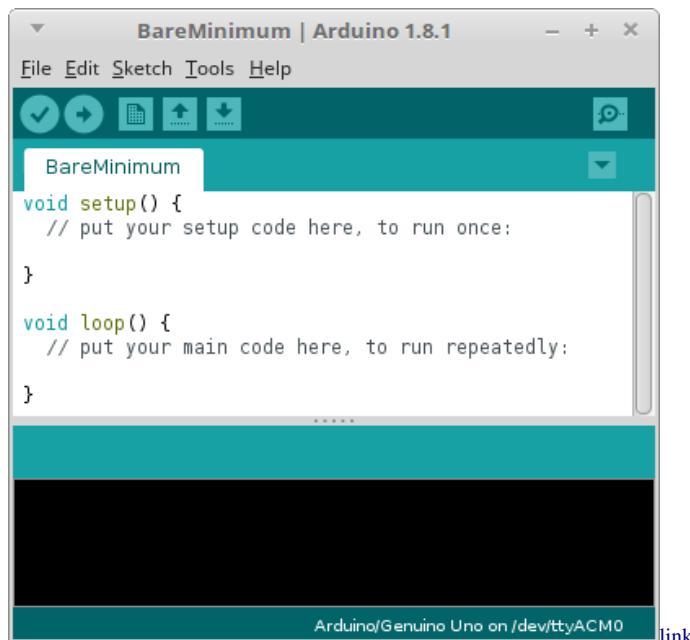
Para usar las variables y las constantes sólo hay que escribir su nombre.

```
miVariable = miConstante + 1;      // ahora miVariable tendrá asignado el valor 4
```

Las instrucciones que comienzan por **#** son directivas del compilador, no son instrucciones, por lo que no acaban en **;**

#include se usa para incluir librerías. Son paquetes de funciones ya escritas que nos simplificaran escribir nuestro código.

#define sustituirá en el programa cada ocurrencia del nombre o caracteres escritos a continuación por los caracteres que siguen hasta el final de la línea. Aunque su valor es invariable no son como las constantes, hay otras directivas del compilador que pueden tratar con ellas.



Ahora podemos interpretar el sketch que aparece por defecto en el Arduino IDE.

Es la definición de dos funciones **setup()** y **loop()** que siempre han de estar presentes en todos los programas de Arduino.

setup() se ejecuta una sola vez al dar tensión o pulsar el botón Reset. Nos sirve para las configuraciones iniciales.

loop() es el programa principal. Las instrucciones de esta función se ejecutarán y cuando finalice se volverá a ejecutar desde el principio de la función una y otra vez sin parar.

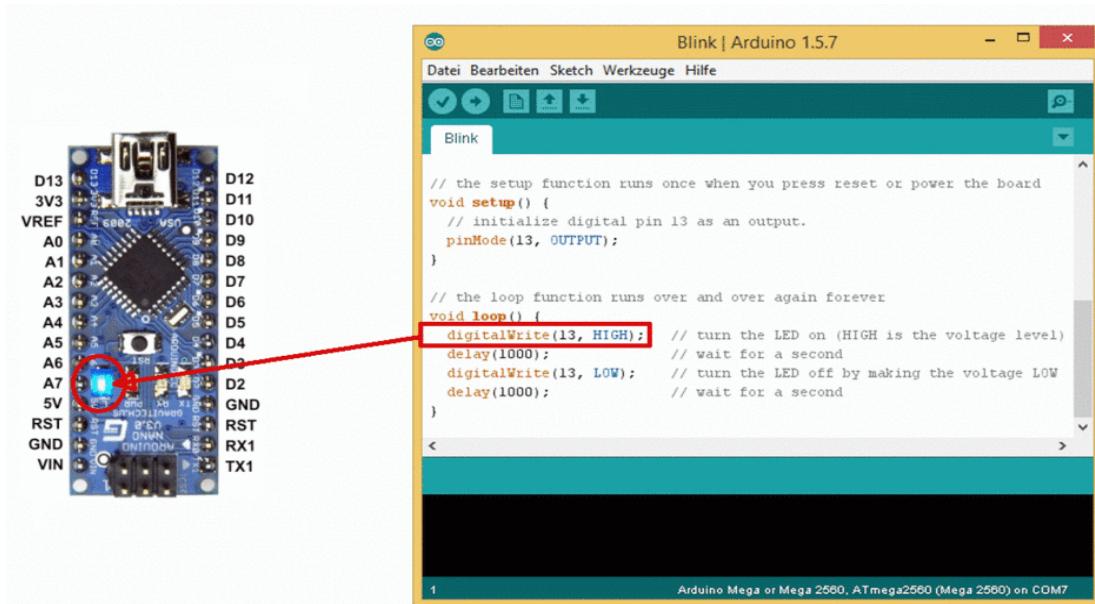
El tipo de datos que devuelve es **void**, o sea, no devuelven datos y no tienen parámetros para ejecutarlas.

La descripción de las diferentes funciones, variables y estructuras del lenguaje Arduino los tenéis aquí:

<https://www.arduino.cc/reference/en/>

Para escribir el programa dentro de estas funciones cada maestrillo tiene su librillo, así que depende de la pericia y conocimiento de cada uno, veamos algunos ejemplos:

2. Ejemplo clásico: Blink



[link ani](#)

El Arduino tiene un LED en la placa que está conectado al pin D13, se puede hacer que parpadee con este simple programa.

En la función **setup()** que se ejecuta al iniciar vamos a configurar el pin 13 como salida digital para poder encender y apagar el LED, para ello usamos la función **pinMode()** que como parámetros necesita el pin y el tipo que puede ser salida (**OUTPUT**), entrada (**INPUT**) o entrada con el pull-up activado (**INPUT_PULLUP**).

```
void setup() {  
    pinMode(13, OUTPUT);      // define pin 13 como salida  
}
```

En la función **loop()** o programa principal vamos a encender y luego apagar el LED con una pausa intermedia de un segundo, una vez acabe como la función **loop()** volverá a empezar, volverá a encenderse y apagarse con lo que tendremos el efecto del LED que parpadea.

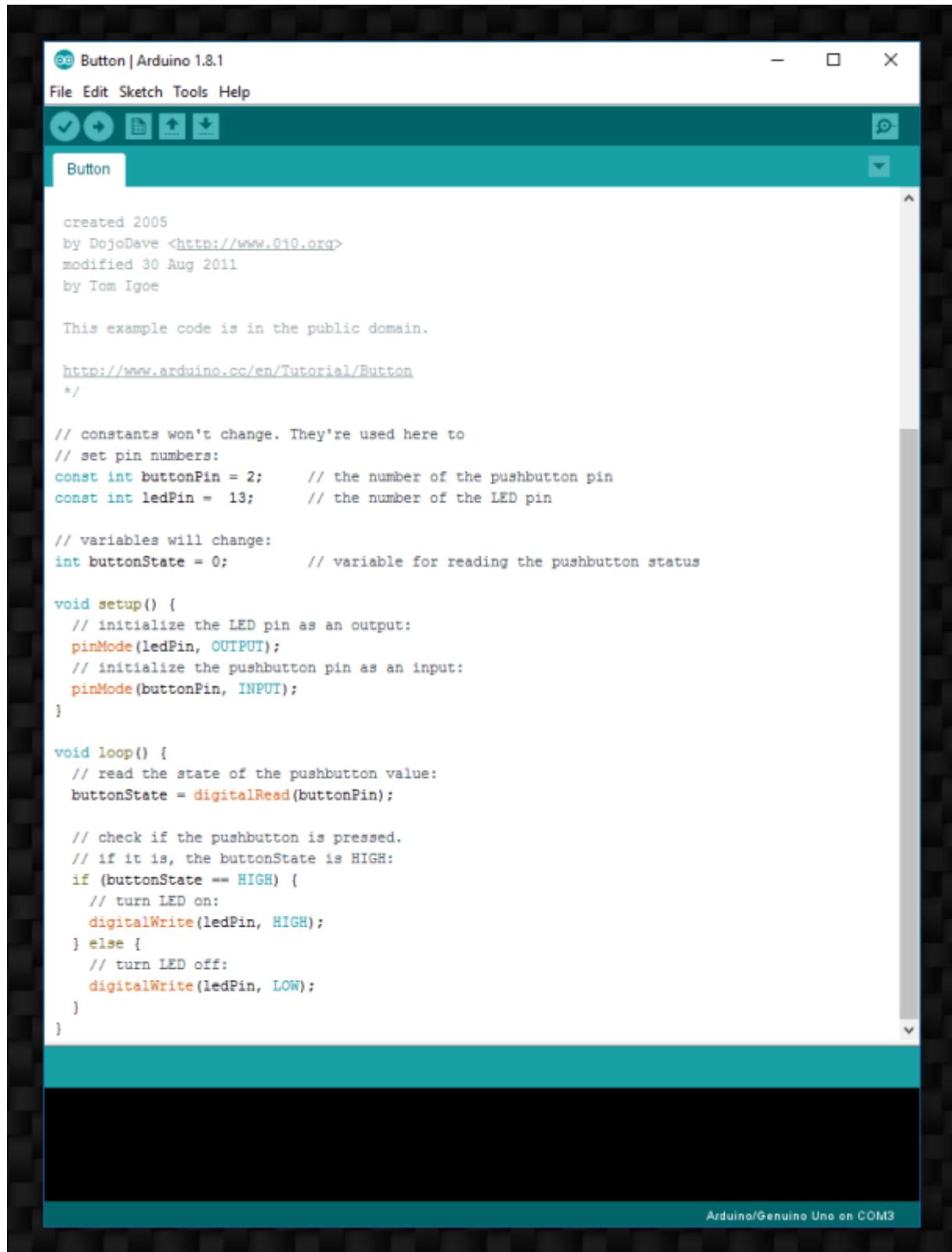
La función **digitalWrite()** coloca el pin indicado en el primer parámetro (13, el LED de la placa) en el estado indicado en el segundo parámetro (**HIGH** para encender el LED, o **LOW** para apagarlo) . HIGH pone 5V en la salida y LOW pone 0V en la salida.

La función **delay()** hace una pausa de los milisegundos indicados como parámetro. 1000 ms = 1 segundo.

```
void loop() {  
    digitalWrite(13, HIGH);    // enciende el LED (HIGH pone 5V en la salida)  
    delay(1000);              // espera un segundo (1000ms)  
    digitalWrite(13, LOW);     // apaga el LED con LOW (pone 0V en la salida)  
    delay(1000);              // espera un segundo (1000ms)  
}
```


3. Otro ejemplo simple

Aunque es matar moscas a cañonazos, para que veamos un poco más del lenguaje Arduino vamos a encender el LED de la placa Arduino cuando se pulse un botón y se apague cuando se suelte.



The screenshot shows the Arduino IDE interface with the title bar "Button | Arduino 1.8.1". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with icons for file operations. The main window displays the "Button" example sketch. The code is as follows:

```
created 2005
by DojoDave <http://www.010.org>
modified 30 Aug 2011
by Tom Igoe

This example code is in the public domain.

http://www.arduino.cc/en/Tutorial/Button
 */

// constants won't change. They're used here to
// set pin numbers:
const int buttonPin = 2;      // the number of the pushbutton pin
const int ledPin = 13;        // the number of the LED pin

// variables will change:
int buttonState = 0;          // variable for reading the pushbutton status

void setup() {
  // initialize the LED pin as an output:
  pinMode(ledPin, OUTPUT);
  // initialize the pushbutton pin as an input:
  pinMode(buttonPin, INPUT);
}

void loop() {
  // read the state of the pushbutton value:
  buttonState = digitalRead(buttonPin);

  // check if the pushbutton is pressed.
  // if it is, the buttonState is HIGH:
  if (buttonState == HIGH) {
    // turn LED on:
    digitalWrite(ledPin, HIGH);
  } else {
    // turn LED off:
    digitalWrite(ledPin, LOW);
  }
}
```

At the bottom of the IDE, the status bar shows "Arduino/Genuino Uno en COM3". A link icon is located at the bottom right of the status bar.

Primero vamos a definir unas constantes para que sea el programa mas fácil de leer y de poder cambiar luego los valores.

El valor constante `buttonPin` será el pin donde conectaremos el pulsador (D2)

El valor constante `ledPin` será el pin donde conectamos el LED (en este caso D13, el pin de la placa).

También vamos a definir `buttonState` que es una variable global y la inicializamos a 0 que nos permitirá almacenar el valor leído del pin donde está el pulsador.

En `setup()` mediante `pinMode()` le indicamos que el pin del LED es de salida (`OUTPUT`), y que el pin del pulsador es de entrada (`INPUT`)

En este caso el pulsador lo conectamos entre 5V y la entrada D2, además le colocamos una resistencia de pull-down de 1K entre D2 y GND.

Cuando pulsemos el botón tendemos en el pin 5V (estado `HIGH`) y cuando no esté pulsado tendremos 0V (estado `LOW`).

En `loop()` vamos a usar la función `digitalRead ()` que necesita como parámetro el pin que ha de leer y devolverá el estado (`HIGH` o `LOW`) que almacenaremos en la variable `buttonState`.

Para saber si encendemos el LED o lo apagamos usamos un condicional `if ()` para comparar la variable con un estado determinado.

El condicional `if()` necesita como parámetro un valor lógico cierto o falso (valor 0), si el valor es cierto (no es cero) se ejecutarán las instrucciones entre las {} que le siguen.

Opcionalmente si se quieren ejecutar algunas instrucciones en el caso de que el valor lógico haya sido falso, se añade `else` y en sus {} se ponen las instrucciones para el caso falso.

Para comparar `buttonState` con `HIGH`, se usa el comparador lógico == que deja un valor cierto o falso para el `if()` según sean los valores iguales o no (un error común es poner = pero esto no es un operador lógico sino uno de asignación, sería como el que habíamos usado para dar un valor inicial a la variable `buttonState`).

En caso de que la comparación sea cierta ejecutamos `digitalWrite()` para encender el LED y sino (`else`) lo apagamos.

Aquí el video:

<https://youtu.be/tCaiPQxt3vM>



[link](#)

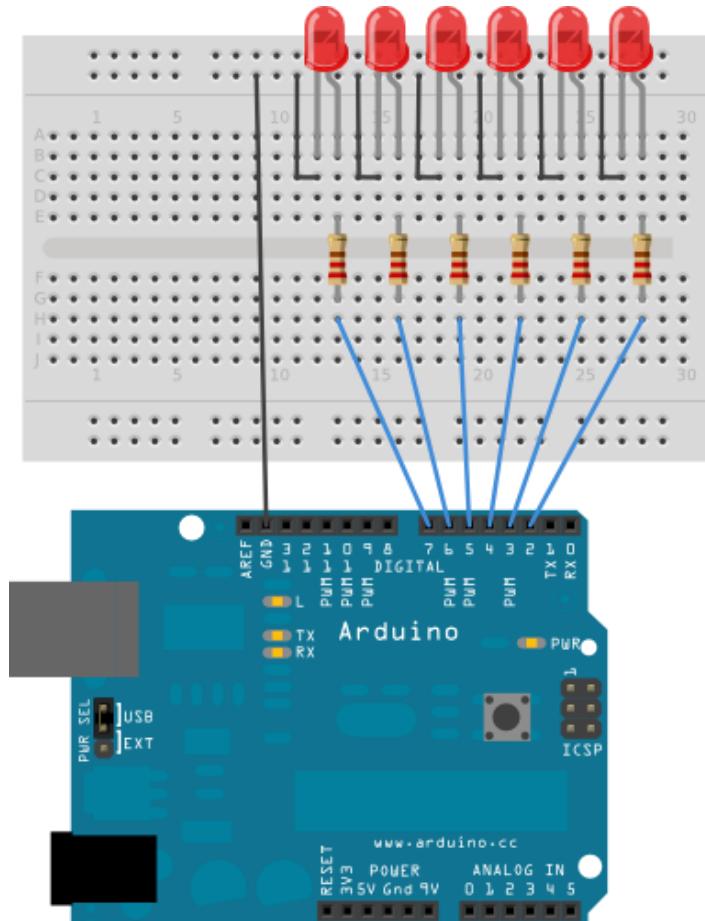
Si usamos el pulsador con una resistencia de pull-up (pulsador entre GND y D2, y resistencia entre D2 y 5V) la lógica se invierte por lo que la comparación del estado del botón la tendríamos que hacer con `LOW`, en lugar de con `HIGH`.

En este último caso si nos queremos ahorrar la resistencia y los cables son cortos, podemos definir en `pinMode()` el pulsador como `INPUT_PULLUP` con lo que se activará la resistencia de pull-up interna del Arduino.

4. Ejemplo simple para la maqueta: Luces obras

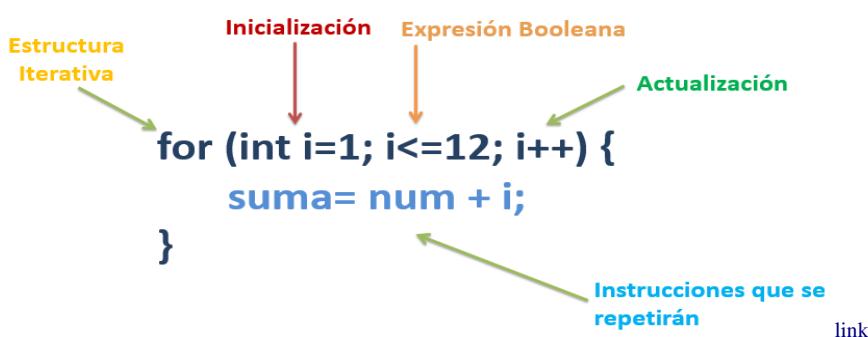
El efecto de luces de obras de carretera es fácil de recrear. Se tiene que ir encendiendo un tiempo corto un LED y luego apagarlo para seguir haciendo lo mismo con el siguiente.

Por ejemplo con 6 LED conectados a las salidas D2 a D7



[link](#)

En el programa vamos a usar un bucle para ir pasando de un pin a otro. Los bucles **for()** se usan de la siguiente manera:



[link](#)

El bucle **for()** necesita tres parámetros. **Atención!** en realidad son instrucciones por lo que están separados por ;

El primer parámetro es la inicialización de la variable que lleva la cuenta al valor inicial, incluso podemos indicar el tipo de dato si no la habíamos definido antes, así será una variable local. En el ejemplo la variable **i** será 1 al empezar.

El segundo parámetro es una condición, mientras se cumpla se ejecutarán las instrucciones entre las **{ }**. En el ejemplo, mientras la variable **i** sea menor o igual (**<=**) a 12.

El tercer parámetro se usa para la actualización de la variable, cada vez que se hayan ejecutado las instrucciones entre {} se actualizará según esta instrucción.

En el ejemplo, se incrementa en 1. Estas expresiones serían iguales:

```
i = i + 1;    // forma normal  
i += 1;       // forma abreviada  
i++;         // con operador de post-incremento
```

Las instrucciones que se ejecutarán en cada iteración son poner la variable **suma** al valor de la suma de las variables **num** e **i**

el programa para las luces de obras haciendo uso de bucles podría ser así:

```
int tiempo = 100; // Cuanto mas alto, mas lento el efecto  
int pin;  
  
void setup() {  
    for (pin = 2; pin < 8; pin++) {  
        pinMode (pin, OUTPUT);  
    }  
}  
  
void loop() {  
    for (pin = 2; pin < 8; pin++) {  
        digitalWrite (pin, HIGH);  
        delay (tiempo);  
        digitalWrite (pin, LOW);  
    }  
    delay (1000);  
}
```

Definimos las variables **pin** y **tiempo** para variar fácilmente el tiempo.

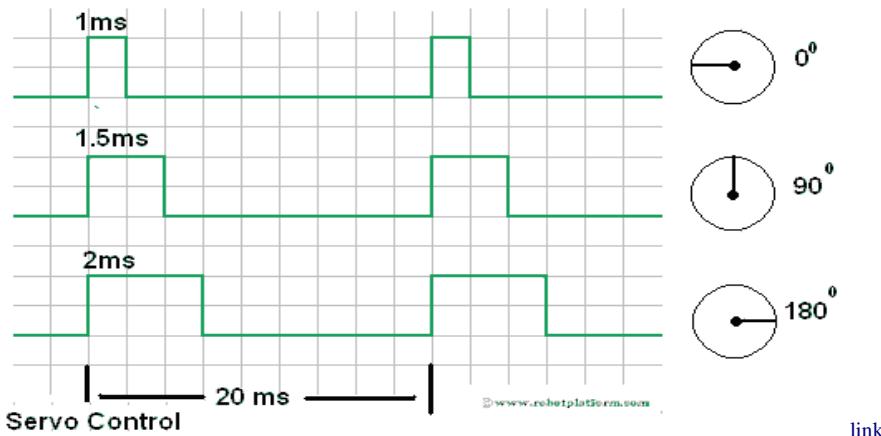
En **setup()** hacemos un bucle desde 2 hasta 7 (cuando llegue la cuenta a 8 la condición será falsa y dejará de ejecutarse el bucle) en el que se inicializa en **pin** de la cuenta como salida (**OUTPUT**)

En **loop()** hacemos un bucle también de 2 a 7 que enciende el LED de la cuenta, espera un **tiempo** en milisegundos indicado en la variable **tiempo**, y apaga el LED.

Cuando finalice la cuenta esperamos un segundo, con todo apagado. Como se volverá a ejecutar **loop()** volveremos a tener la misma secuencia.

5. Librerías incluidas en Arduino IDE

Si queremos mover un servo debemos generar una señal como la de la figura según la posición del servo:



[link](#)

Podemos generar la señal de forma parecida al ejemplo [Blink](#) anterior ya que se trata de encender y apagar un pin con un determinado tiempo pero no es muy práctico.

Arduino tiene unas librerías (conjunto de funciones) que proporcionan funcionalidades extra para interactuar con el hardware o manipular datos.

Aquí tenéis la referencia de las incluidas en el Arduino IDE:

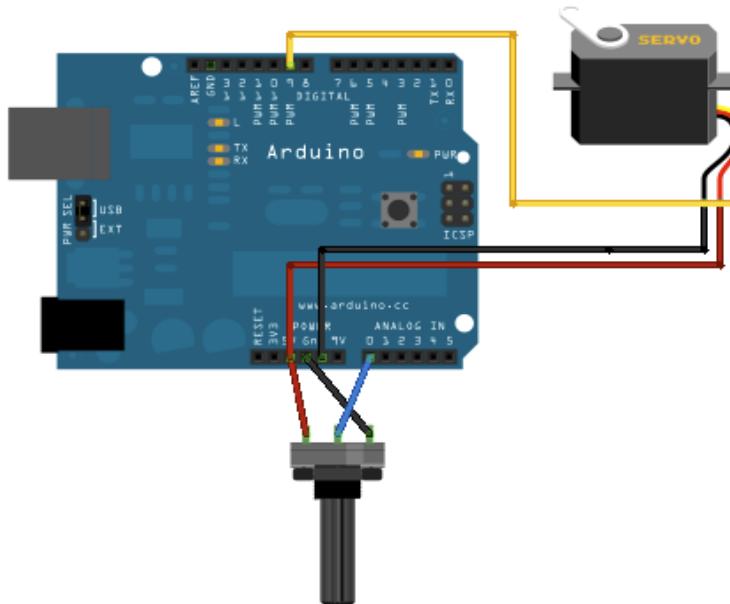
<https://www.arduino.cc/en/Reference/Libraries>

Una de estas librerías permite generar la señal para controlar los servos. Tiene definidas unas funciones útiles para definir el pin donde generar la señal y el periodo de la misma.

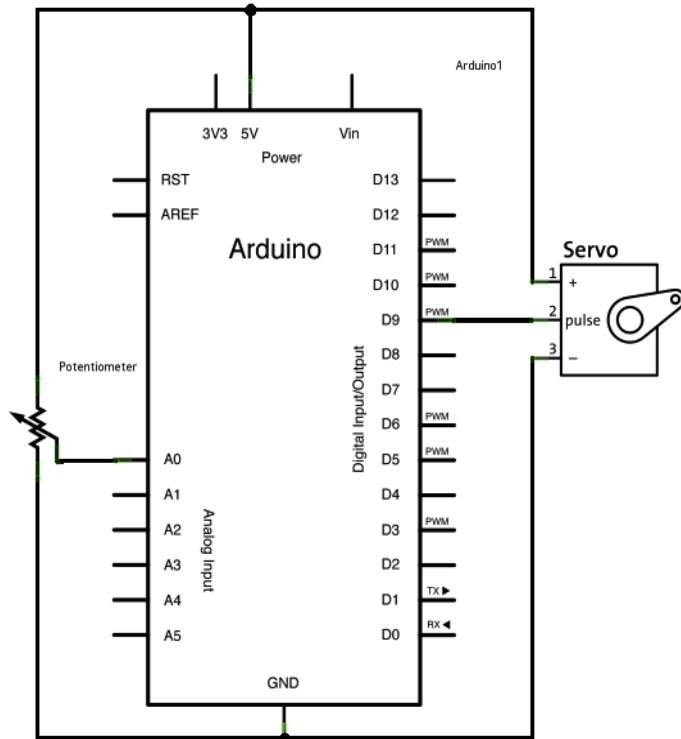
<https://www.arduino.cc/en/Reference/Servo>

Para usar las funciones incluidas en las librerías debemos mirar su documentación y ejemplos para ver como se usan cada una de ellas.

Como ejemplo, vamos a mover un servo conectado en el pin D9 en función de la posición de un potenciómetro conectado a la entrada analógica A0



[link](#)



[link](#)

_1__SERVO Arduino 1.6.8

```
#include <Servo.h>

Servo myservo; //Creamos el objeto llamado myservo

int potpin = A0; //pin del potenciómetro al pin A0
int val; // Se crea una variable para un buen desarrollo

void setup()
{
  myservo.attach(9); //usamos el pin PWM 9
}

void loop()
{
  val = analogRead(potpín); // leemos el pin A0
  val = map(val, 0, 1023, 0, 180); //usamos la función map() para convertir valores
  myservo.write(val); //escribimos en el servo el valor obtenido
  delay(15);
}
```

Guardado.

El nombre del proyecto debe ser modificado. El nombre del proyecto debe consistir solo Ademas debe contener menos de 64 caracteres.

2 Arduino/Genuino Uno en /dev/cu.usbmodem1411 [link](#)

Empezamos incluyendo la librería de servos en nuestro programa para poder usar sus funciones, para ello usamos la directiva `#include` con el nombre de la librería entre <>

Esta librería trabaja con objetos, hay que definir un objeto por cada servo, luego le pediremos que ejecute las funciones para ese servo.

El objeto `Servo` será `myservo` y así nos referimos a él cuando queramos asignarle el pin donde está conectado nuestro servo o moverlo

Definimos un par de variables de tipo `int` (numero entero de -32768 a 32767), `potpin` que será el pin donde está conectado el potenciómetro (`A0` es un valor ya definido con `#define` en el sistema Arduino y será sustituido por su valor de número entero por el compilador) y `val` que contendrá el valor leído de la entrada analógica (entre 0 para 0V y 1023 para 5V)

En la función `setup()` que se ejecuta al principio y sirve principalmente para configurar cosas, le indicaremos al objeto `myservo` donde tiene conectado el servo que tiene que controlar. Para ello usamos la función de la librería `attach()` con el parámetro del número de pin. Como es para un objeto se escribe el objeto, un `.` y la función con sus parámetros.

En la función `loop()` leeremos el pin analógico en la variable `val` y convertiremos el valor leído (de 0 a 1023) en una valor para controlar el servo (entre 0 y 180) y moveremos el servo a esa posición, dejaremos un tiempo para que se genere la señal, este ciclo se volverá a repetir por lo que si movemos el potenciómetro se moverá el servo al mismo tiempo.

La función `analogRead()` necesita como parámetro el pin analógico a leer, en este caso lo tenemos en la variable `potpin` que hemos definido al principio y devuelve un resultado que lo asignamos a la variable `val`

Para convertir el valor leído a un valor usable por el servo, la función `map()` convierte el primer parámetro, comprendido entre el segundo y tercer parámetro a un valor entre el cuarto parámetro y el quinto parámetro, o sea, básicamente hace una regla de tres. El resultado lo volvemos a dejar en la variable `val` ya que su valor anterior ya no nos hace falta.

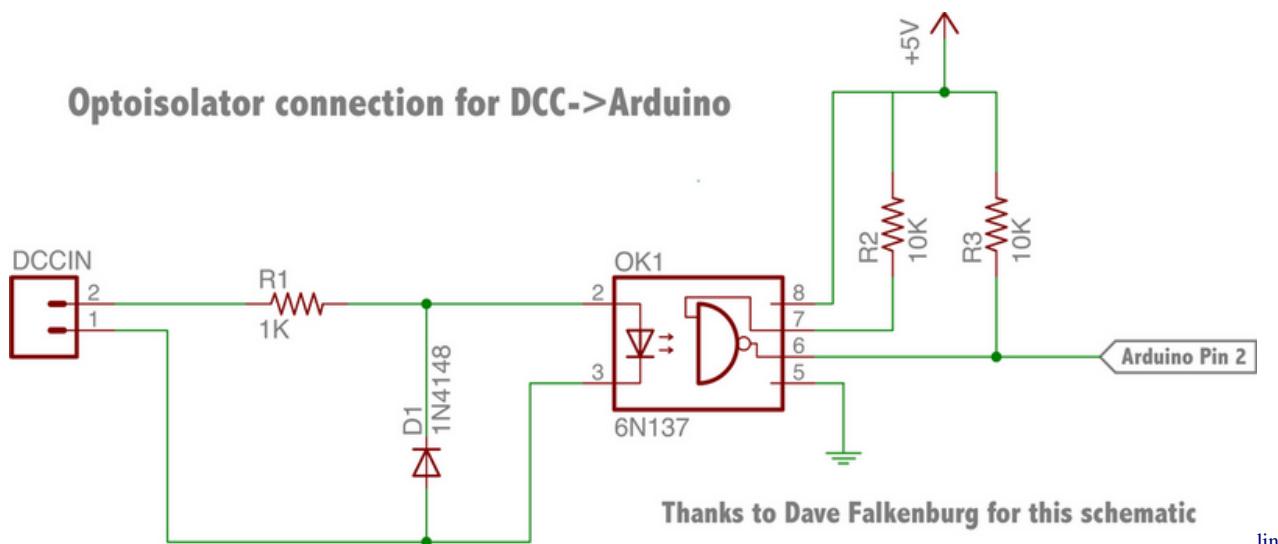
La función `write()` de la librería `Servo` genera la señal para mover un servo a una posición según el parámetro que se le pase. Le tenemos que poner el objeto al que queremos que le realice la función. A ese objeto ya le habíamos indicado anteriormente en `setup()` el pin al que estaba conectado el servo por lo que `write()` ya sabe dónde generar la señal.

Finalmente con la función `delay()` dejamos un pequeño tiempo para generar la señal, está en 15ms aunque sería mejor subirlo a 20ms que es el periodo de la señal de servos.

6. Otras librerías

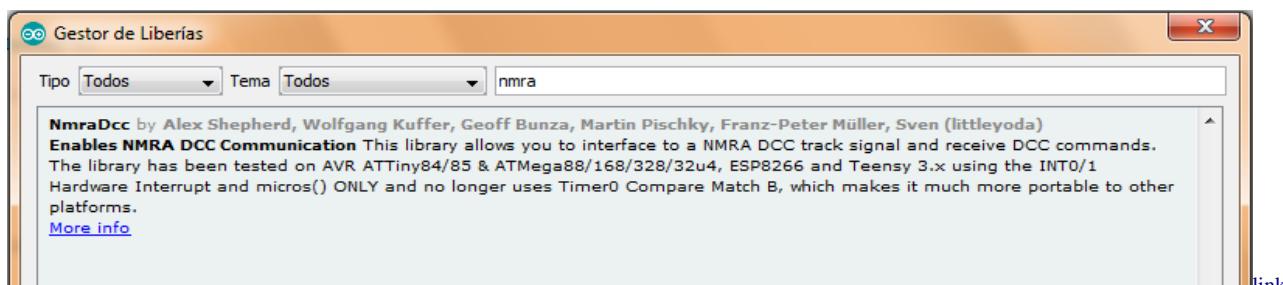
Hay librerías creadas por terceros que nos pueden ser útiles para nuestro diseño. Como no vienen instaladas en el Arduino IDE tendremos que instalarlas antes de poder usarlas en nuestro programa.

Un ejemplo de estas es la librería que nos permite decodificar la señal DCC usando un pequeño circuito que con un opto aísla la señal de vía de nuestro Arduino. Proporcionando esta señal al pin D2 y usando esta librería podemos hacer nuestro propio descodificador de accesorios.



Para instalar la librería, hay que ir al menú Programa del Arduino IDE, elegir la opción Incluir Librería. Dependiendo de la librería que queramos usar hay que elegir entre Gestionar librería o Añadir librería .ZIP

En nuestro caso vamos a usar la librería NmraDcc que está disponible a través de la opción de menú Gestionar librería, la buscamos en el Gestor de librerías y la instalamos.



En este caso no hay un manual que explique cada función pero la librería con ejemplos de uso de sus funciones la tenemos aquí:

<https://github.com/mrrwa/NmraDcc>

Como ejemplo vamos a hacer un simple descodificador de accesorios que encenderá y apagará el LED de la placa Arduino (el conectado al pin D13) al mover el desvío 6 en nuestra central DCC.

```

Archivo Editar Programa Herramientas Ayuda
AccDecDemo
1 // Demostració de la llibreria NmraDcc com descodificador d'accessoris -- Paco Cañada 2019
2
3 #include <NmraDcc.h>           // Llibreria DCC
4
5 NmraDcc Dcc;                  // Crea el objecte DCC
6                                         // Definim unes constants que pot canviar l'usuari
7 const byte DCC_PIN = 2;          // DCC pin
8 const byte LED_PIN = 13;         // LED pin
9 const int DCC_ADDRESS = 6;       // Adreça accessori
10
11 void setup() {                 // Inicialització
12     pinMode(LED_PIN, OUTPUT);   // LED apagat per defecte
13     digitalWrite(LED_PIN, LOW);
14                                         // Configurem pin i tipus de descodificador
15     Dcc.pin(digitalPinToInterrupt(DCC_PIN), DCC_PIN, 1);
16     Dcc.initAccessoryDecoder( MAN_ID_DIY, 1, FLAGS_OUTPUT_ADDRESS_MODE, 0 );
17 }
18
19 void loop() {                  // Bucle principal
20     Dcc.process();             // Procesa la descodificacio de la senyal DCC
21 }
22
23 // Quan arriba un paquet DCC per accessoris s'executa aquesta rutina
24 void notifyDccAccTurnoutOutput( uint16_t Addr, uint8_t Direction, uint8_t OutputPower ) {
25     if( ( Addr == DCC_ADDRESS ) && OutputPower ) { // Si es la nostra adreça i esta activa
26         if (Direction == 0)                         // Segons sigui recte/desviat
27             digitalWrite(LED_PIN, LOW);            // apaguem LED
28         else
29             digitalWrite(LED_PIN, HIGH);           // encenem LED
30     }
31 }
32

```

[link](#)

Primero incluimos la librería para trabajar con ella `#include <nmraDcc.h>`

Al igual que la de servos funciona con objetos por lo que definimos un objeto `NmraDcc` para la señal `Dcc`

Para que luego sea más sencillo leer el programa y cambiar los valores definimos unas constantes para el pin donde lee la señal DCC, el pin que vamos a controlar (en este caso el LED del pin D13) y la dirección DCC a la que responderá.

En `setup()` usamos las funciones `pinMode()` y `digitalWrite ()` para indicar que el pin del LED es salida y que lo queremos apagado.

Luego usamos las funciones de la librería, como podemos ver en sus ejemplos, para indicar el pin por donde leerá la señal DCC y inicializar la librería para que sea como un decodificador de accesorios pasándole como parámetros el ID y versión de fabricante (en este caso DIY) y el tipo de decodificador de accesorios según las constantes definidas en la librería y que se pueden ver en sus ejemplos.

En el bucle principal no necesitamos nada más que se use la función `process()` de la librería con el objeto `Dcc` para ir procesando los paquetes DCC que llegan.

Cuando llega un paquete DCC de accesorios la librería hace que se ejecute la función `notifyDccAccTurnoutOutput()` y pone los parámetros del paquete que ha recibido.

Nosotros tenemos que escribir esta función y comparar los parámetros recibidos con los que queremos que el LED responda.

Así que con un condicional `if()` comparamos (`==`) el parámetro `Addr` con nuestra dirección que habíamos definido como una constante al principio y además (`&&`) que `OutputPower` sea cierto, o sea que se esté activando la salida en el paquete DCC.

Si es nuestra dirección y se activa usamos otro condicional `if()` para comparar (`==`) el parámetro `Direction`, que nos dice si es recto o desviado. Si es 0 apagamos el LED mediante `digitalWrite ()` y si no es 0 lo encendemos.

Nota: En el último `if/else` no hemos colocado las `{ }` ya que no son necesarias cuando sólo se usa una instrucción

7. Otras temporizaciones: Luces de edificio aleatorias.

La función **delay()** detiene el programa los milisegundos que se le pasen como parámetro. A veces esto no es deseable ya que tenemos que controlar varias cosas a la vez y esta detención nos complica la gestión del tiempo.

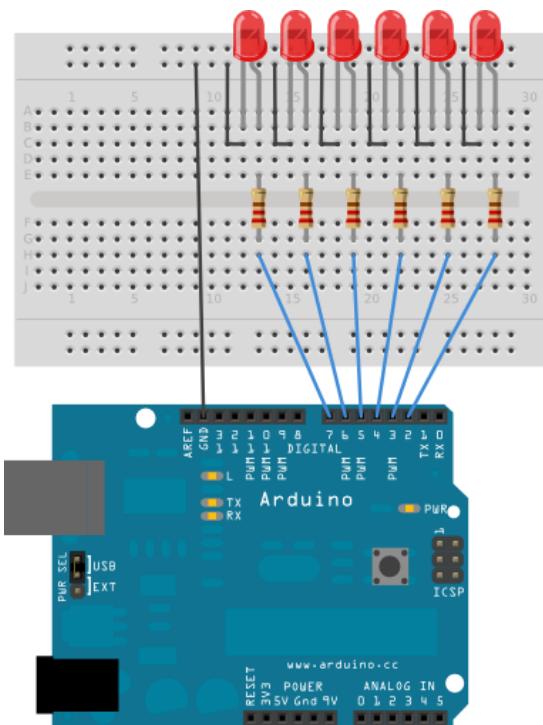
En lenguaje Arduino la función **millis()** nos devuelve los milisegundos que han pasado desde que el programa empezó a correr. El valor es del tipo **unsigned long** por lo que volverá a cero al cabo de 49 días.

Podemos comparar el tiempo actual devuelto por **millis()** con otro valor que proceda de la suma de un tiempo anterior más un tiempo de espera que hayamos definido, si la suma es menor que el tiempo actual quiere decir que ya ha pasado el tiempo de espera. Hacer varios tiempos de espera y saber si ya han pasado se reduce a comparar los valores guardados con el tiempo actual.

A veces queremos que esos tiempos de espera sean aleatorios de manera que no parezca algo preestablecido o irreal para ello existe una función que genera un numero pseudo-aleatorio cada vez que se la llama. La función **random()** tiene dos variantes, si se le llama con un solo parámetro genera un numero entre 0 y el parámetro, si se le llama con dos parámetros genera un numero entre el primer y el segundo parámetro.

Aunque cada vez que se llama a **random()** el valor es aleatorio como se genera a través de una fórmula matemática los valores se repiten entre subsecuentes ejecuciones del sketch. Si necesitamos más aleatoriedad podemos inicializar el generador de números aleatorios con la función **randomSeed()**, si ponemos un número diferente cada vez la secuencia variará. Una manera es leer un pin analógico que no esté conectado con lo que se leerá el ruido que será diferente cada vez.

Si tomamos el montaje que habíamos usado para las luces de obras podemos convertirlo en uno para iluminar las diferentes ventanas de un edificio de forma totalmente aleatoria.



[link](#)

En este ejemplo, para hacer el código más compacto vamos a hacer uso de los arrays (matriz). Los arrays son conjuntos de valores a los que se accede con un índice, se definen de forma análoga a las variables: tipo nombre y entre **[]** se indica la cantidad de elementos del tipo definido que podrá almacenar, se finaliza con ;

```
int miArray[3];
```

Si se quiere inicializar con valores determinados, se escribe el tipo, nombre, entre **[]** la cantidad de elementos totales = y entre **{ }** separados por , los valores iniciales y se finaliza con ;

```
int miOtroArray[3] = {10, 20, 30};
```

El uso de los arrays es parecido a las variables, se escribe su nombre y entre [] el índice del dato a acceder. El primer valor siempre tiene el índice 0.

```
int diez = miOtroArray[0];
```

El programa seria este:

```
/* Iluminacion aleatoria de varias luces en un edificio con diferentes
   tiempos de apagado y encendido - Paco Cañada 2019 */
```

```
#define numPins 6                                // numero de LEDs a controlar

int ledPin [] = { 2, 3, 4, 5, 6, 7};             // pines en los que estan conectados los LED
unsigned long finTiempo [numPins];               // fin de tiempo para encender o apagar LED
int estadoPin [numPins];                         // Estado actual del LED

long tiempoMinimoApagado = 10000;                // tiempo minimo que el LED esta apagado
long tiempoMaximoApagado = 25000;                // tiempo maximo que el LED esta apagado
long tiempoMinimoEncendido = 5000;               // tiempo minimo que el LED esta encendido
long tiempoMaximoEncendido = 20000;              // tiempo maximo que el LED esta encendido

void setup () {
    randomSeed (analogRead(A0));                  // A0 es un pin analogico no usado, al aire. Lee el ruido.

    for (int n = 0; n < numPins ; n++) {
        pinMode (ledPin[n], OUTPUT);
        digitalWrite (ledPin[n], LOW);
        estadoPin[n] = LOW;
        finTiempo[n] = nuevoTiempo(LOW);
    }
}

void loop () {
    for (int n = 0; n < numPins; n++) {           // Bucle para todos los LED
        if (millis() > finTiempo[n]) { // si el tiempo actual es mayor que el fin de tiempo del pin
            if (estadoPin[n] == LOW) {           // Mira el estado actual del pin si estaba apagado
                digitalWrite(ledPin[n], HIGH);
                estadoPin[n] = HIGH;
            }
            else {                           // si por el contrario estaba encendido
                digitalWrite(ledPin[n], LOW);
                estadoPin[n] = LOW;
            }
            finTiempo[n] = nuevoTiempo (estadoPin[n]); // asigna un nuevo tiempo para el cambio de estado
        }
    }
}

unsigned long nuevoTiempo (int estado) { // tiempo aleatorio a partir del tiempo actual y estado
    unsigned long tiempo;

    if (estado == HIGH)                      // tiempo aleatorio segun estado
        tiempo = random (tiempoMinimoEncendido, tiempoMaximoEncendido);
    else
        tiempo = random (tiempoMinimoApagado, tiempoMaximoApagado);
    tiempo += millis();                     // tiempo actual
    return (tiempo);                        // devuelve el tiempo calculado
}
```

Primero definimos cuantas salidas para LEDs (o en este caso, elementos de los arrays) tenemos para que sea luego sencillo cambiarlo para adaptarlo a nuestras necesidades.

La directiva **#define** hace que el compilador sustituya la palabra **numPins** por el numero 6, también serviría para sustituir cualquier otra cadena de caracteres. Lo podríamos haber hecho como una constante con **const int numPins = 6;** pero así vemos más cosas del lenguaje y queda más chulo.

La idea es tener varios arrays para las diferentes cosas que necesitamos (pin, tiempo, estado) y con el índice acceder al dato en concreto. El índice vendrá de un bucle **for()** que nos recorrerá todos los datos. Cuando se tenga que cambiar de estado asignaremos un nuevo tiempo aleatorio en función del estado .

Así que definimos un array llamado `ledPin[]` que como elementos tendrá los pines en los que tengamos conectados los LEDs. Si os dais cuenta entre `[]` no hemos puesto la cantidad de elementos, le hemos dejado el trabajo de contarlos al compilador ya que le definimos los elementos a continuación. Para que esto funcione bien hemos de poner al menos tantos elementos como vayamos a acceder sino faltarán datos y se producirán errores inesperados al ejecutar el programa, ya que no sabrá a qué pin acceder.

Los pines los he escrito en orden pero no es necesario, y si modificamos `numPins` tendremos que tener tantos elementos como `numPins`. Si quisieramos más salidas, los pines analógicos también se podrían usar, por ejemplo escribiríamos el elemento `A5` para usar el pin analógico A5

El array `finTiempo[]` contiene el tiempo en milisegundos que usaremos para comparar con `millis()` para saber cuando tenemos que hacer un cambio de estado. Usamos el tipo `unsigned long` como devuelve `millis()`

El array `estadoPin[]` nos indicará el estado actual del pin. (`HIGH` o `LOW`)

También definimos unos valores para que los podamos cambiar fácilmente de los tiempos límites de encendido y apagado. Los definimos del tipo `long` como requiere la función `random()`

En `setup()` inicializamos la semilla de la función pseudo aleatoria leyendo el pin analógico `A0` que no utilizamos, aunque no es estrictamente necesario nos servirá para que la secuencia sea más aleatoria.

Hacemos un bucle `for()` para inicializar los diferentes arrays y pines, en este caso irá de 0 a 5 ya que `numPins` es 6.

Primero ponemos el pin como salida, así que tomamos un elemento del array `ledPin[]` según el índice del bucle y lo ponemos como salida (`OUTPUT`) usando `pinMode()`

Con `digitalWrite()` hacemos igual, tomamos un elemento del array `ledPin[]` y lo ponemos a `LOW` (apagado) El elemento correspondiente del array `estadoPin[]` también lo ponemos a `LOW` ya que nos indica el estado actual del pin.

Como última cosa, en el bucle asignamos un tiempo a un elemento del array `finTiempo[]` con la llamada a la función `nuevoTiempo()`

La función `nuevoTiempo()` la hemos escrito nosotros más adelante para simplificar el listado y evitar repeticiones de código. Devuelve un dato `unsigned long`, como `millis()`, que es un tiempo aleatorio en función del parámetro estado que le pasemos.

Para ello según el estado genera con `random()` un valor aleatorio entre los valores que definimos al principio del programa y lo asigna a la variable local `tiempo`. Después le sumamos el tiempo actual devuelto por `millis()` con lo que `tiempo` será un tiempo futuro.

Con `return(tiempo)` finalizamos la función devolviendo como resultado la variable `tiempo`.

Estas expresiones son equivalentes:

```
tiempo = tiempo + millis(); // forma normal  
tiempo += millis(); // operador compuesto
```

En `loop()` hacemos un bucle `for()` al igual que en `setup()` pero esta vez comparamos el tiempo actual con el tiempo previsto de cada elemento para el cambio de estado, si el tiempo actual es mayor, hay que cambiar de estado el pin y calcular un nuevo tiempo aleatorio para que vuelva a cambiar.

En el `if()` miramos si el tiempo actual `millis()` es mayor que `finTiempo[]` del elemento, si es así toca cambiar de estado.

Con otro `if()` miramos el estado actual del elemento, si es `LOW` cambiamos el pin con `digitalWrite()` a `HIGH`, y actualizamos también el elemento en el array `estadoPin[]`. Con el `else`, si no era `LOW`, los cambiamos a `LOW`.

Finalmente calculamos un nuevo tiempo en función del nuevo estado del elemento.

A medida que se repita `loop()` el tiempo devuelto por `millis()` irá cambiando e irá llegando a los diferentes `finTiempo[]`

Nota: Observar las `{ }` y la indentación (sangrado) para comprobar visualmente que instrucciones se ejecutan dentro de cada `if() / else() / for()`

8. Librerías incompatibles: Placa giratoria por segmento y sonido.

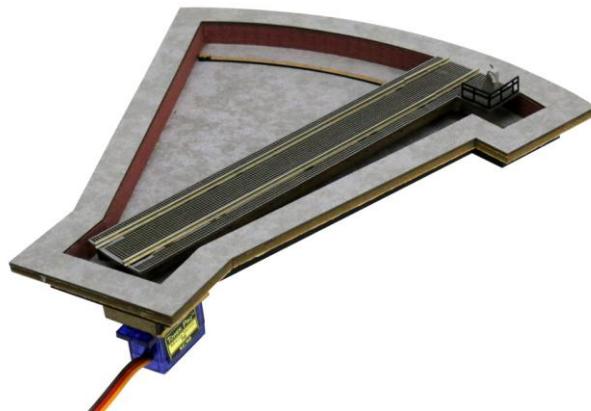
Una de las ventajas del Arduino es que hay librerías para casi todo por lo que nos facilita mucho la creación de nuestros proyectos sin tener que preocuparnos por los detalles técnicos.

Como contrapartida, al no saber qué recursos del micro se usan nos podemos encontrar el problema de que incluyamos varias librerías que funcionan perfectamente por separado pero juntas en un sketch no funcionen como se esperan debido a que hacen uso de los mismos recursos.

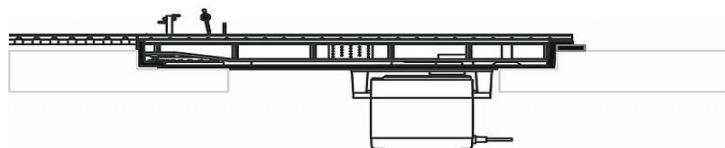
En este ejemplo, vamos a controlar una de estas placas por segmento que se mueven gracias a un servo y además vamos a ponerle sonido 😊



[link](#)



[link](#)

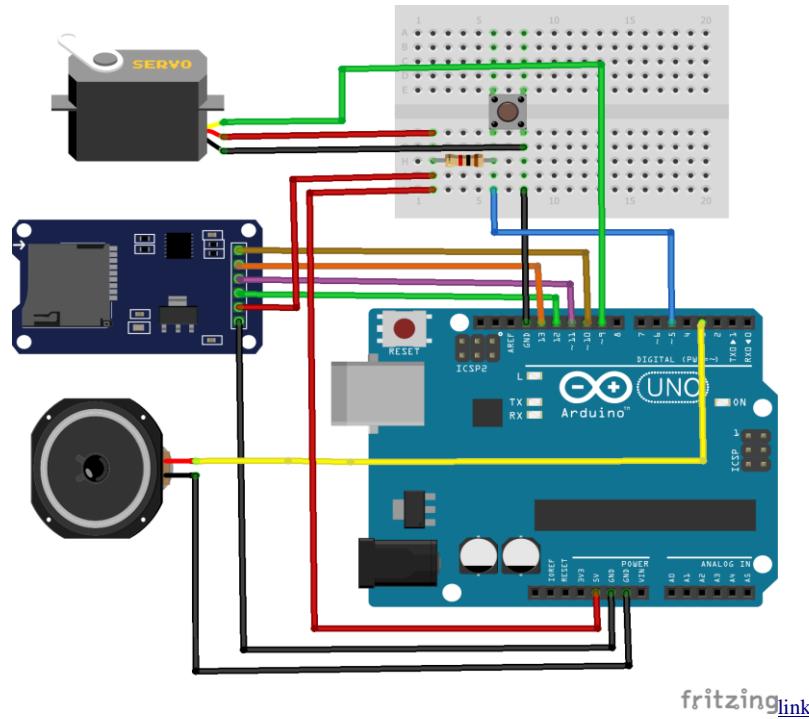


[link](#)

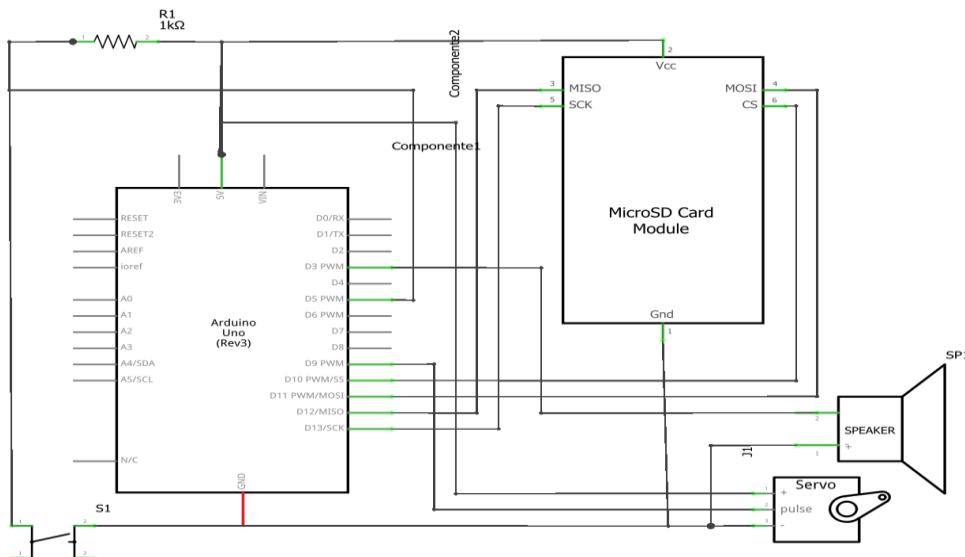
Para el servo utilizamos librería `Servo.h` que ya hemos visto, para el sonido vamos a usar la librería `TMRpcm.h` que reproduce archivos .wav desde una tarjeta SD. Para ello tendremos que instalarla desde el **Gestor de librerías**. (**Programa -> Incluir librería -> Gestionar librería**)

<https://github.com/TMRh20/TMRpcm>

Necesitaremos un lector de SD y un altavoz para la parte de sonido y un servo y un pulsador para el control de la placa giratoria. Haremos que cada vez que pulsemos el botón la placa giratoria se mueva a la siguiente salida, mientras esté en movimiento se oirá el sonido, al llegar a la salida se detendrá y se parará el sonido.



fritzing [link](#)



fritzing [link](#)

En este pagina hay muchos sonidos que nos pueden valer para nuestra maqueta:

<https://www.ittproducts.com/GL.html>

Nosotros vamos a usar este sonido para la placa giratoria:

<https://www.ittproducts.com/media/g1320-1.mp3>

El sonido para que lo pueda reproducir la librería se tiene que convertir a .wav (24-32kHz, mono, PCM unsigned 8 bit).

Lo podemos hacer con programas como el Audacity (<https://www.audacityteam.org/>) o bien hacerlo online:

<https://audio.online-convert.com/es/convertir-a-wav>



[link](#)

El problema que nos encontramos es que las librerías **Servo.h** y **TMRpcm.h** son incompatibles cuando usamos un Arduino Uno/Nano. El servo no se mueve al activar el sonido.

En un Arduino Uno/Nano sólo hay tres *timers* (temporizadores): Timer0 (el que usa **delay()** / **millis()** / **micros()**), Timer1 (el que usa **Servo**) y Timer2 (que usa **tone()**).

Ambas librerías hacen uso del Timer1 reprogramándolo para sus necesidades por lo que interfieren una en la otra. (Un Arduino Mega tiene más temporizadores que puede usar la librería **TMRpcm** por lo que no tendría esta incompatibilidad)

Si en nuestro diseño tuviésemos que usar los pines PWM mediante la función **analogWrite()** hay que tener en cuenta que también hace uso de los timers, según el pin PWM usa un timer u otro:

<https://www.arduino.cc/reference/en/language/functions/analog-io/analogwrite/>

Conociendo de donde viene la incompatibilidad, si queremos usar ambas librerías en un Arduino Uno/Nano tenemos que hacer que una de las librerías use otro temporizador (el Timer2) pero como nosotros no hemos escrito el código de la librería, esto puede ser una dura tarea.

Buscando en internet encontramos la librería **ServoTimer2.h** que controla hasta 8 servos usando el Timer2.

<https://github.com/nabontra/ServoTimer2>

También encontramos la librería **PWMServo.h** que al no usar interrupciones el movimiento es más estable, genera el pulso del servo mediante PWM, pero usa el Timer1

<https://github.com/PaulStoffregen/PWMServo>

Por suerte, en la librería **TMRpcm** han previsto que se pueda usar el Timer2 para ello se ha de modificar el archivo **pcmConfig.h** como en la imagen (el archivo está en **Mis Documentos/Arduino/libraries/TMRpcm**). Activar el uso del Timer2 y usar un buffer de 128 bytes:

```
***** GENERAL USER DEFINES *****
See https://github.com/TMRh20/TMRpcm/wiki for info on usage

Override the default size of the buffers (MAX 254). There are 2 buffers, so memory usage will be double this number
Defaults to 64bytes for Uno etc. 254 for Mega etc. note: In multi mode there are 4 buffers*
#define bufferSize 128 //must be an even number

/* Uncomment to run the SD card at full speed (half speed is default for standard SD lib)*/
#define SD_FULLSPEED

/* HANDLE_TAGS - This option allows proper playback of WAV files with embedded metadata*/
//#define HANDLE_TAGS

/* Ethernet shield support etc. The library outputs on both timer pins, 9 and 10 on Uno by default. Uncommenting this
will disable output on the 2nd timer pin and should allow it to function with shields etc that use Uno pin 10 (TIMER1 COMPB).*/
##define DISABLE_SPEAKER2

/* Use 8-bit TIMER2 - If using an Uno, Nano, etc and need TIMER1 for other things*/
#define USE_TIMER2
```

[link](#)

Por ahora tenemos cuatro soluciones:

- 1 - Usar la librería `Servotimer2.h` en lugar de la `Servo.h`
- 2 - Modificar el archivo `pcmConfig.h` de la librería `TMRpcm` y usar `Servo.h`
- 3 - Modificar el archivo `pcmConfig.h` de la librería `TMRpcm` y usar `PWMServo.h`
- 4 - Usar un Arduino Mega

Después de unas pruebas vamos a escoger la opción 3, así tendremos un movimiento del servo más estable que no le afectará el uso intensivo de interrupciones de la librería `TMRpcm`

A veces no resulta tan fácil encontrar el origen de las incompatibilidades por lo que no nos queda más remedio que indagar en Internet si alguien encontró con los mismos problemas que nosotros y lo más importante, encontró la solución o tendremos que cambiar nuestro diseño.

Otra dificultad que nos encontramos es inherente a la propia construcción de la placa giratoria, el ángulo del segmento de las comerciales es de 37,5°, nosotros podemos mover el servo con la función `write()` de grado en grado ya que el parámetro que se le pasa es la posición en grados. Resulta evidente que con esa precisión vamos a notar los saltos durante el movimiento.

Tenemos que buscar la manera de suavizarlo: La librería `Servo` tiene la función `writeMicroseconds()` que nos permite mover el servo variando el pulso de microsegundo en microsegundo, como vimos anteriormente 1500us corresponden a la posición central del servo (90°), la posición 0° a 1000us y la posición 180° a 2000us

Para tres vías de salida y 37,5° de segmento, con la función `writeMicroseconds()` tendremos unas 100 posiciones intermedias entre vía y vía, mientras que con `write()` solo serían unas 18 posiciones intermedias, así que si nos movemos de 1us en 1us el movimiento será más suave.

Lamentablemente la opción escogida, librería `PWMServo`, no tiene la función `writeMicroseconds()` por lo que nos tendremos que conformar con el movimiento grado a grado.

En el programa compararemos la posición del servo con la posición de cada una de las vías de salida, si no estamos sobre ninguna de ellas moveremos el servo 1 grado en una dirección dada por una variable y mantendremos el sonido. Si estamos sobre una salida pararemos el movimiento del servo y el sonido. En función de si la salida es un extremo u otro del segmento de la placa giratoria tendremos que cambiar la variable que indica la dirección de movimiento. En la/las vías intermedias no hay que cambiar la dirección de movimiento.

También en las salidas tenemos que comprobar el pulsador para iniciar el movimiento hacia la siguiente vía de salida y poner en marcha el sonido si se pulsa.

El programa no es muy complicado pero hay que escribir una gran cantidad de `if()` para ir comparando cada una de las opciones (posición servo/vía, vía extremo/vía intermedia, pulsador, dirección de movimiento)

La estructura `switch...case` nos permite hacer varias comparaciones de una manera más simple, sería como encadenar varios `if()else()`

La instrucción `switch` compara una variable con un valor constante especificado por los `case`, cuando coincide la variable con el valor ejecuta las instrucciones que encuentre hasta la instrucción `break`. Incluso es posible usando `default` ejecutar instrucciones en caso de que la variable no coincida con ninguno de los `case`

Este sería el programa:

```
// Control de placa giratoria de segmento con sonido - Paco Cañada 2019
// Las librerías Servo.h y TMRpcm son incompatibles en Arduino Uno/Nano.
// Las dos usan el Timer1, tenemos que cambiar la librería TMRpcm para que use el Timer2
// y usamos PWMServo para mas estabilidad

// Board           SERVO_PIN_A   SERVO_PIN_B   SERVO_PIN_C
// ----          -----        -----        -----
// Arduino Uno, Duemilanove    9          10          (none)
// Arduino Mega      11         12          13

#include <PWMServo.h>           // Libreria para el Servo
#include <SPI.h>                 // librerias para la SD
#include <SD.h>                  //
#include <TMRpcm.h>               // Libreria para el sonido desde SD

TMRpcm Audio;                   // Objeto para generar audio
PWMServo servoPlaca;           // Objeto para mover el servo

#define pinPulsador 5             // Pin del pulsador
#define pinServo SERVO_PIN_A     // Pin del servo
#define pinSD 10                  // Pin de seleccion de la SD
```

```

#define Vial 72           // Posiciones del servo para cada vía
#define Via2 90
#define Via3 108

int posicion;           // Posición actual del servo
int dirección;          // Dirección 1 o -1

void setup() {
    pinMode (pinPulsador, INPUT);
    servoPlaca.attach(pinServo);
    servoPlaca.write(Via2);
    posicion = Via2;
    dirección = 1;
    delay(1000);
    Audio.speakerPin = 3;
    Audio.setVolume(4);
    Audio.quality(1);
    SD.begin(pinSD);
}

void loop() {
    switch (posicion) { // actuamos según la posición en que estamos
        case Vial:
            dirección = 1;
            CompruebaPosición ();
            break;
        case Via2:
            CompruebaPosición (); // Comprueba pulsador y sonido
            break;
        case Via3:
            dirección = -1;
            CompruebaPosición (); // Cambiamos dirección hacia anterior vía
            break;
        default:
            MueveServo (); // Movemos el servo
            break;
    }
}

void CompruebaPosición () {
    if (Audio.isPlaying()) // Si hemos llegado, paramos el sonido
        Audio.disable();
    if (servoPlaca.attached())
        servoPlaca.detach(); // y desconectamos el servo
    if (digitalRead (pinPulsador) == LOW) { // Si pulsamos el pulsador
        Audio.play("placa.wav"); // Ponemos el sonido
        servoPlaca.attach(pinServo); // Conectamos el servo
        MueveServo (); // Empezamos a mover el servo
    }
}

void MueveServo () {
    posicion += dirección; // Movemos el servo un poco según dirección
    servoPlaca.write (posicion);
    delay(38);
}

```

Primero incluimos todas las librerías necesarias con **#include** (Recordad de modificar el fichero **pcmConfig.h** del la librería **TMRpcm** para que use el Timer2)

Luego definimos los objetos para el servo y el audio según la librería. Con **#define** establecemos unos valores para los pines que usamos (y que podemos cambiar según necesitemos)

Los **case** necesitan valores constantes, aquí usamos **#define** que nos hace mas legible el código para establecer las posiciones del servo de cada una de las vías de salida (en nuestro caso tres) con las que comparar la variable **posición**.

Otra variable que necesitamos es **dirección** para saber hacia dónde nos movemos. Como vamos a mover el servo 1 grado más o 1 grado menos de donde estamos, podemos hacer que la variable **dirección** sea 1 o -1 de manera que sumándola a **posición** ya tengamos la siguiente posición.

En el **setup()** indicamos que el pin del botón es de entrada, conectamos el servo y lo movemos a una posición conocida que guardamos en **posición** (mejor que sea una vía intermedia así tendremos margen de movimiento a ambos lados). Cargamos la variable **dirección** con 1 por defecto y esperamos un poco a que el servo se coloque en posición.

Finalmente inicializamos las librerías de la tarjeta SD y del sonido según sus ejemplos.

El `loop()` gracias a `switch...case` es bastante sencillo, `switch` como parámetro tiene la variable `posicion` que es con la que hará las comparaciones. Entre `{ }` del `switch` se colocan los `case` con los valores constantes con los que queremos comparar y :

Si la variable coincide con la constante se ejecutan las instrucciones, hasta el `break`

Nuestros `case` son las posiciones de las diferentes vías de salida: La `Via1` situada en el extremo tiene que modificar la variable `direccion` para que el servo no se mueva hacia afuera sino que se mueva hacia adentro en el próximo desplazamiento. Para el `case` de la `Via3` ocurre lo mismo pero ahora con el signo cambiado. En la vía intermedia no hay que cambiar la variable `direccion` ya que el futuro movimiento tiene que seguir en la dirección actual para barrer todas las posiciones.

Cuando se llega a una vía (coincide el `case`) hay que detener el sonido y parar el servo, además hay que comprobar el botón para ver si hay que iniciar un nuevo desplazamiento, esto lo vamos a hacer con nuestra función `CompruebaPosicion()` de forma que no repitamos código.

Como llegamos a la función `CompruebaPosicion()` porque ha coincidido `posicion` con un `case` de una vía quiere decir que hemos llegado al final del desplazamiento por tanto miramos si el sonido se está reproduciendo con la función `isPlaying()` del objeto `Audio`, si es así lo detendremos con `disable()` y también desconectamos el servo con la función `detach()` del objeto `Servo`. Esto último es opcional, pero lo hacemos para disminuir el consumo y las pequeñas vibraciones.

Ahora comprobamos si se ha pulsado el botón, (la entrada está a `LOW`) para ir a la siguiente vía de salida. Si es así, reproducimos el sonido del movimiento con la función `play()` del objeto `Audio` con el parámetro del nombre del fichero `.wav` entre "

Conectamos de nuevo el pin del servo e iniciamos el movimiento del servo, esto lo vamos a hacer con nuestra función `MueveServo()`

La función `MueveServo()` simplemente asigna a `posicion` la suma de `posicion` más `direccion` (recordemos que es 1 o -1), mueve el servo a esa posición con `write()` y espera un poco a que se produzca el pulso para el servo.

Como `MueveServo()` cambia la variable `posicion` en la próxima ejecución de `loop()` ya no coincidirá con ninguno de los `case`, por lo que se ejecutarán las instrucciones que siguen a `default`

En `default` lo único que hacemos es llamar a `MueveServo()` que moverá un poco más el servo y seguirá entrando en `default` hasta que `posicion` coincida con una de las vías donde entrará en uno de los `case` y allí se detendrá el sonido y el movimiento del servo

Nota: Si tuviéramos 4 vías de salida en lugar de tres, las dos intermedias harían lo mismo por lo que se podría poner así:

```
case Via2:  
case Via2a:  
    CompruebaPosicion ();           // Comprueba pulsador y sonido  
    break;
```

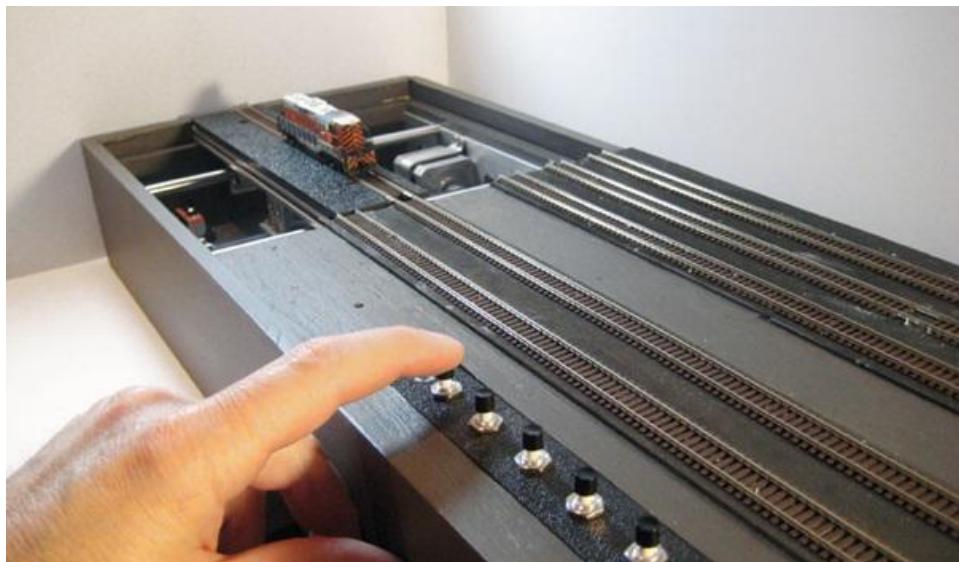
Tanto si se entra por `Via2` como por `Via2a` se ejecutan las mismas instrucciones hasta el `break`

9. Uso de 'shields': Plataforma deslizante con motor paso a paso

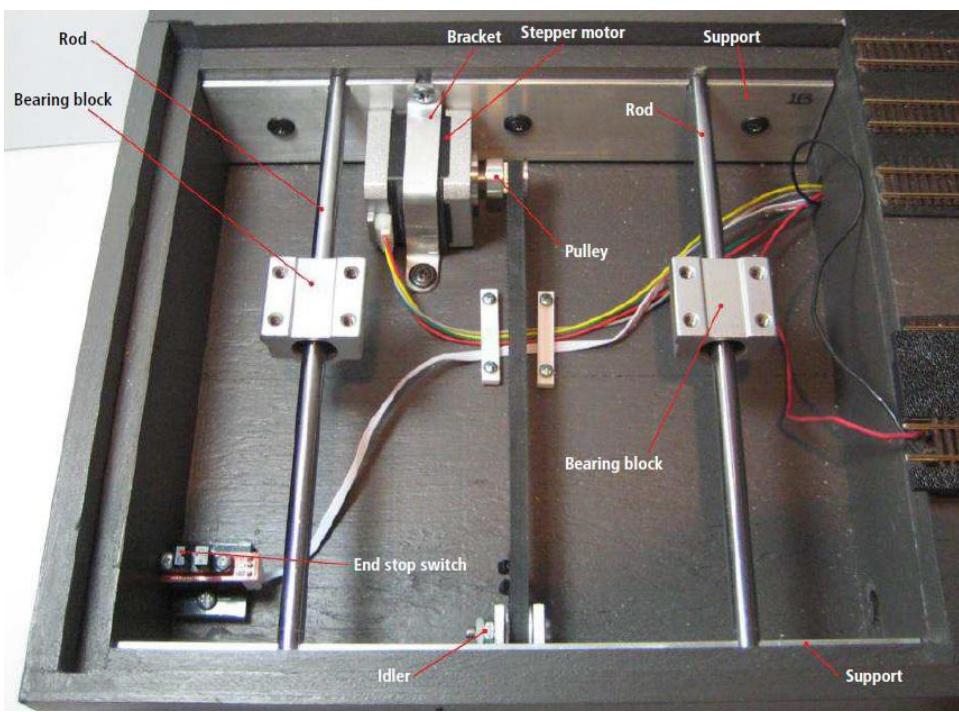
Para el siguiente proyecto vamos a necesitar un poco de bricolaje y hacer uso de circuitos electrónicos adicionales para controlar un motor paso a paso.

Gracias a la fácil disponibilidad de piezas mecánicas para la construcción de impresoras 3D las podemos aprovechar para construir una plataforma deslizante artesanal como esta:

<https://mrr.trains.com/how-to/dcc-electrical/2018/06/arduino-control-code-for-traverser>



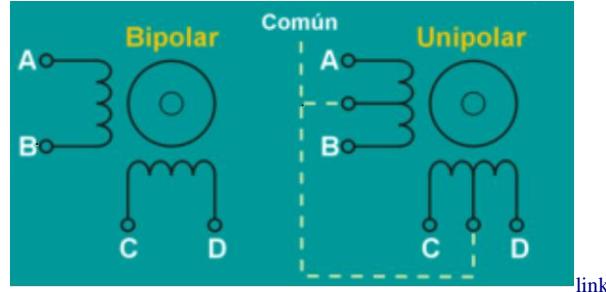
[link](#)



Dejaremos para los manitas la parte mecánica y nos centraremos en la parte electrónica y la programación.

Los motores paso a paso tienen una construcción especial con varios bobinados. Dependiendo de cómo se controlen esos bobinados podemos hacer que se mantenga en una posición o que gire en uno u otro sentido. Las diferentes posiciones en las que puede quedar estacionario se denominan pasos. Para impresoras 3D es habitual encontrar motores con 200 pasos/revolución

Hay dos tipos de motores paso a paso, los unipolares y los bipolares dependiendo de la conexión de los bobinados:



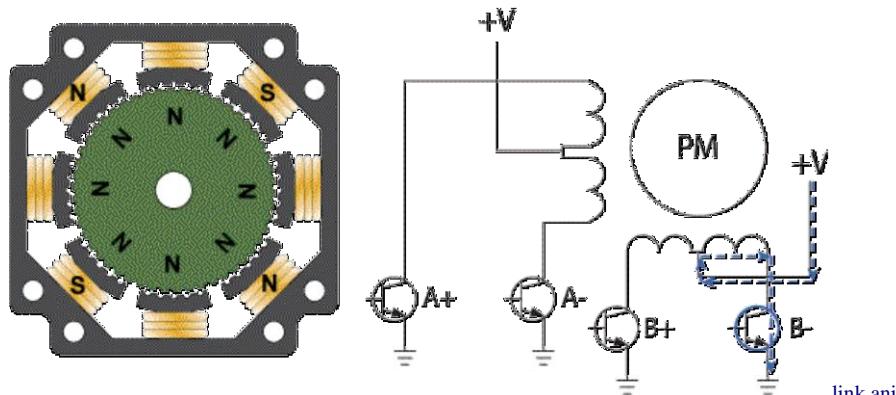
[link](#)

Motor unipolar:



[link](#)

Un motor unipolar (como el 28BYJ-48) suele tener 5 o 6 cables y es el de más sencillo de controlar, solo requiere que se activen los bobinados uno tras otro para hacerlo girar:



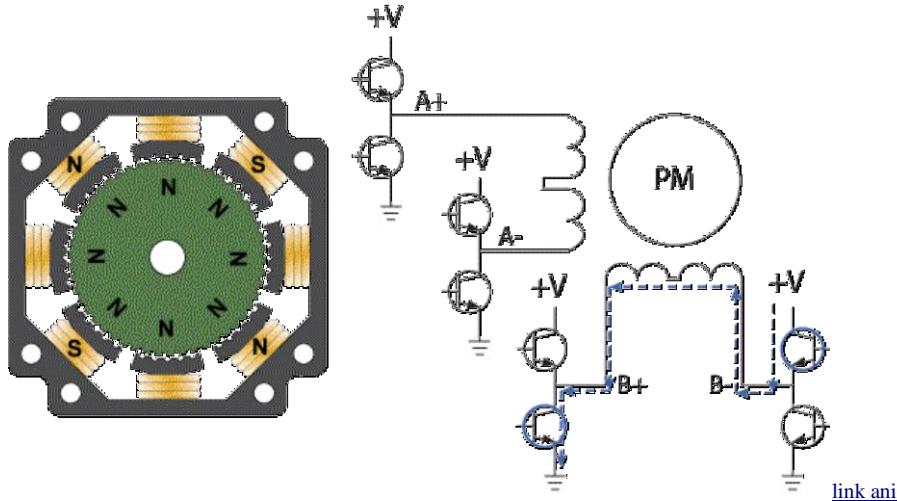
[link ani](#)

Motor bipolar:



[link](#)

Un motor bipolar tiene 4 cables y para controlarlo no solo activamos uno u otro bobinado, sino que tenemos que cambiar la polaridad para conseguir que gire:

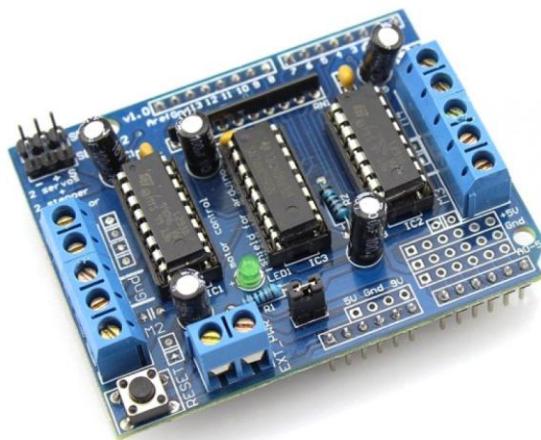


[link ani](#)

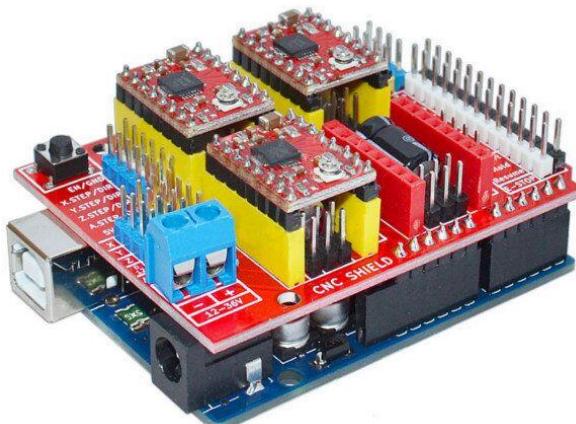
Nota: Un motor unipolar podemos controlarlo como un bipolar si no conectamos la toma intermedia de cada bobina.

Un ‘shield’ en Arduino es una placa que se apila sobre el Arduino o sobre otro ‘shield’, de manera que nos permite ampliar el hardware o para dar funcionalidad extra a un Arduino.

Entre ellas hay que pueden controlar un motor paso a paso:

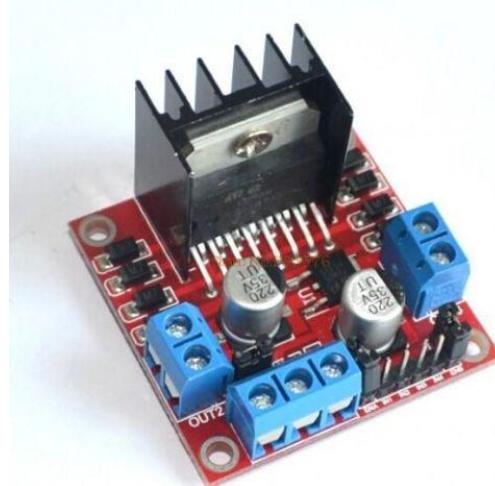


[link](#)



[link](#)

Otras placas de extensión no se pinchan en el Arduino pero también pueden ser controladas desde los pines analógicos y digitales para los motores paso a paso:



[link](#)

[link](#)

Yo voy a utilizar un viejo motor bipolar del desguace de una impresora o fax y la shield de motores (Arduino motor shield v3) que tenia por un cajón porque la compré para probar la central digital DCC++ por Arduino: (<https://sites.google.com/site/dccpsite/home>)



[link](#)

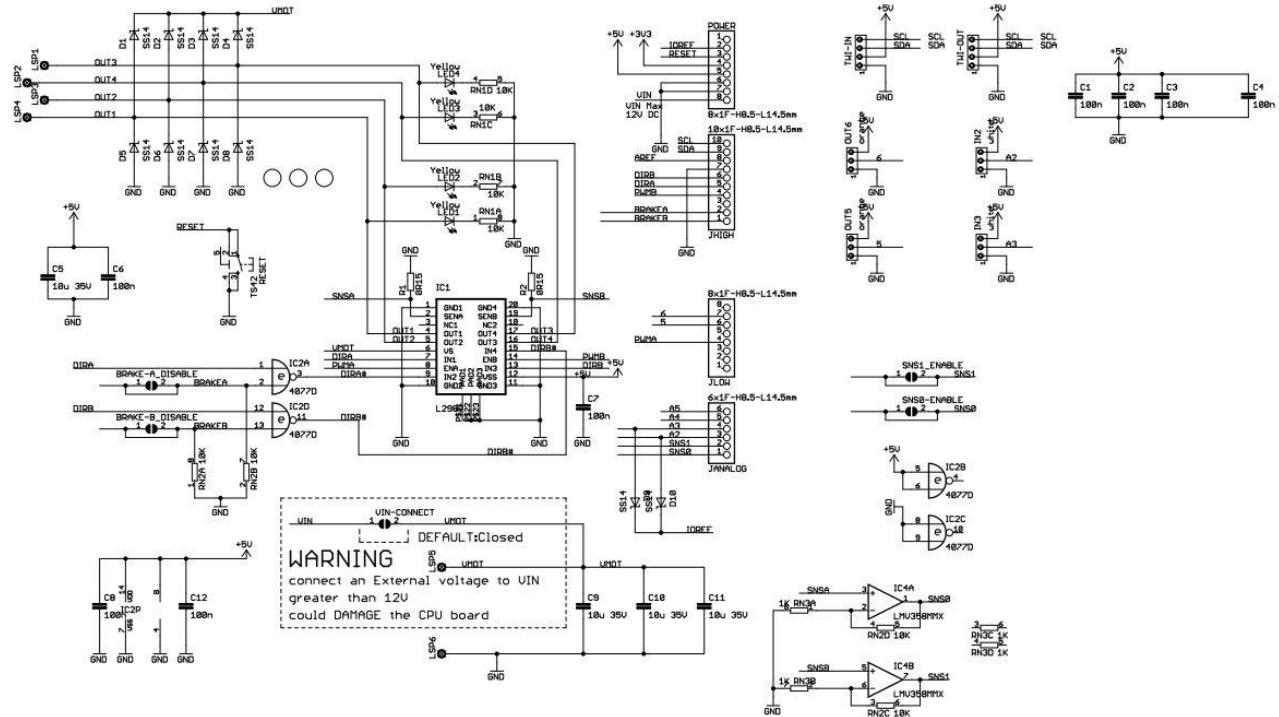
En esta shield si el motor necesita otra tensión diferente de la que admite el Arduino 7V..12V, debemos cortar el jumper (puente) de la conexión Vin sino podríamos dañar la placa Arduino Uno. En este caso alimentaremos independientemente el Arduino de la shield



[link](#)

Con las shields es interesante tener su esquema para saber que pines usa y cuales nos deja libres para otros usos (u otras shields) de forma que no tengamos incompatibilidades de hardware.

Para la Motor shield rev3 este es su esquema:



[link](#)

Como vemos esta shield está basada en un integrado L298 que puede controlar dos motores de continua o uno paso a paso bipolar con 2A por canal. Estos son los pines que usa:

Function	pins per Ch. A	pins per Ch. B
Direction	D12	D13
PWM	D3	D11
Brake	D9	D8
Current Sensing	A0	A1

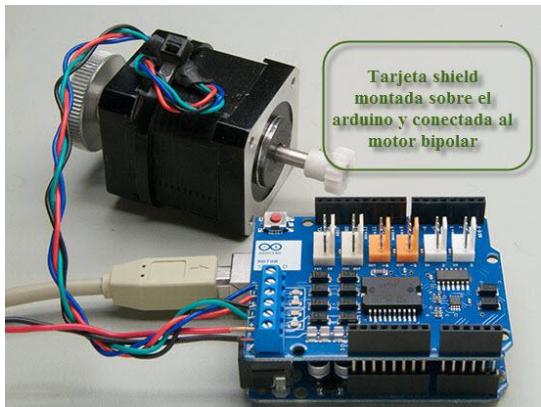
[link](#)

Las entradas analógicas A0 y A1 nos dan la corriente del motor/bobina a razón de 1,65 V/A: Para 2A daría 3,3V.

Además del jumper de Vin tiene otros jumper en la PCB que se pueden cortar y utilizarlos para otros fines sino necesitamos esas señales para el control del motor : SNS0 (A0), SNS1 (A1) , BRAKEA (D9) y BRAKEB (D8).

También tiene una serie de conectores que nos dan acceso a las señales A2, A3, A4 (SDA), A5 (SCL), D5 y D6.

Finalmente las bobinas (o motores) se conectan una en A- y A+ y la otra en B- y B+



Nota: Con esta shield no podríamos ponerle sonido como el proyecto anterior de placa giratoria ya que usa los pines del SPI necesarios para el lector SD. Si quisiéramos sonido, tendríamos que usar otra placa diferente de control de motores.

IMPORTANTE: Si usas motores de impresora 3D (NEMA17 o similar) ten en cuenta que se eligen por el par (torque) por lo que no suelen dar la tensión de funcionamiento sino la resistencia y la corriente por fase, con ayuda de la ley de Ohm: $V = I \times R$ podemos saber la tensión de funcionamiento, que suele ser baja, por lo que quizás esta no sea la shield más adecuada para vuestro motor bipolar. En ese caso mejor usar los Pololus o la CNC Shield:

<http://carlini.es/manejar-un-motor-stepper-con-un-driver-a4988-y-arduino/>

Si no tenemos el datasheet (hoja de datos) del motor paso a paso quizás no sabremos cuales son los pares de cables que van conectados a cada bobina.

Con un multímetro podemos medir la resistencia entre dos cables y según la medida averiguar cual corresponde a cada bobina.

Otro método mas manual pero que curiosamente funciona es este:

- 1- Con el motor bipolar desconectado gira un poco el eje con la mano
- 2- Conecta un par de cables entre ellos
- 3- Gira un poco el eje con la mano. Si es más difícil de girar que antes has encontrado los dos cables de la misma bobina

<https://steppermotor.top/como-identificar-las-bobinas-de-tu-motor-pap/>

La librería `Stepper.h` incluida en Arduino es una manera simple de controlar un motor paso a paso tanto unipolares como bipolares:

<https://www.arduino.cc/en/Reference/Stepper>

El inconveniente es que su función `step()` que hace girar el motor los pasos que se le pasen como parámetro, es bloqueante (como vimos con `delay()`). Detiene el programa hasta que el motor haya terminado de moverse.

Si establecemos la velocidad con `setSpeed()` en 1 RPM (revolución por minuto) con un motor de 200 pasos por revolución al ejecutar `step(200)` tardará un minuto en ejecutarse para girar una revolución, lo cual puede ser inaceptable para nuestro programa.

Una librería mas avanzada que podemos instalar desde el Gestor de Librerías es la librería `AccelStepper`:

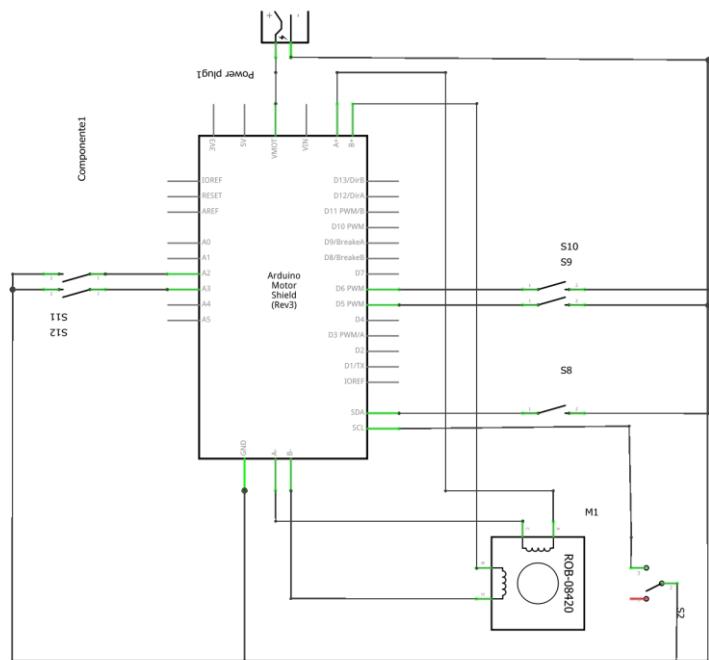
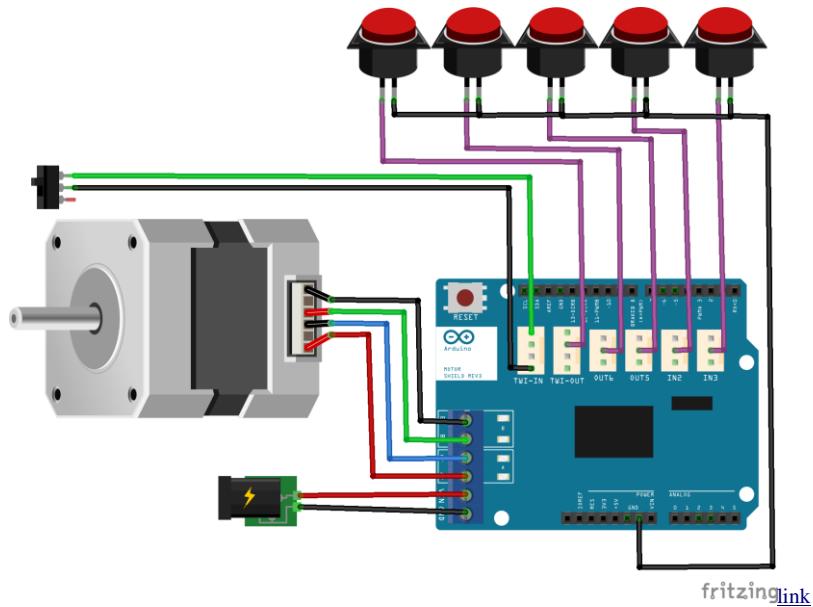
<https://www.airspayce.com/mikem/arduino/AccelStepper/index.html>

Esta librería, no es bloqueante, controla el motor paso a paso y además se le puede indicar una aceleración/deceleración para el movimiento.

Nuestra plataforma tendrá un motor paso a paso para el movimiento, un interruptor de final de carrera que accionará la plataforma al llegar a un extremo y un pulsador para cada vía.

Para el programa haremos que al dar tensión la plataforma se mueva despacio en dirección al interruptor final de carrera, cuando lo toque pararemos y haremos que retroceda y vuelva a buscar más lentamente el final de carrera, ese punto será el paso 0. Luego moveremos la plataforma hasta el paso que corresponda con la primera vía.

El resto es fácil, esperaremos a que se pulse el pulsador de una vía, momento en que llevaremos la plataforma a la vía correspondiente.



Vamos a ver otra forma de organizar los datos. Podemos crear una especie de gran variable que contenga otras variables. En nuestro caso, crearemos una estructura para definir como es un punto de paro, que a su vez contendrá información del pulsador asociado y de la posición de ese punto de paro.

Como esta gran variable es como cualquier otra variable podemos hacer arrays con ella.

Para crearla se utiliza **struct** y el nombre de este tipo de estructura, luego entre **{ }** se definen como siempre cada una de las variables que contendrá y se finaliza con **;**

```
struct miStruct {
    int miNumero;
    char miCaracter;
};
```

Para definirla, así como en las variables escribíamos el tipo y su nombre, aquí se escribe el nombre de la **struct** que hemos creado y el nombre que le vamos a dar y acabamos con **;**

```
miStruct misVariables;
```

Si queremos inicializarla con valores se escribe el nombre de la struct creada, su nombre = entre {} los valores separados por , de cada una de las variables que componen la struct y se acaba con ;

```
miStruct misVariables = {10, 'a'};
```

Para usar el valor de una de las variables que forma la struct, se escribe el nombre (de la struct definida) después . y luego el nombre de la variable de la struct

```
char letra = misVariables.miCaracter;
```

Para determinados bucles vamos a usar la estructura **while()** que repite las instrucciones entre {} hasta que la expresión entre () sea falsa. La estructura **do...while()** funciona de la misma manera pero la condición entre () se comprueba al final por lo que las instrucciones entre {} se ejecutan al menos una vez.

```
while (letra == 'a') {
// instrucciones a ejecutar
}

do {
// instrucciones a ejecutar
} while ((letra == 'a'));
```

Gracias a esta estructura el programa será más sencillo para mover el motor hasta que llegue a un determinado punto.

El programa seria este:

```
// Control de placa deslizante con motor paso a paso - Paco Cañada 2019

#include <AccelStepper.h> // Libreria motores paso a paso con
aceleracion/deceleracion

// Placa Arduino Motor Shield rev3
// salidos digitales:
const int dirA = 12; // dir: Control del motor por la libreria
const int dirB = 13;
const int pwmA = 3; // pwm: Tiene que estar a HIGH
const int pwmB = 11;
const int brakeA = 9; // brake: Tiene que estar a LOW si no se ha cortado su
jumper en la shield
const int brakeB = 8;
// entradas analogicas
const int sns0 = A0; // Lectura de la corriente del motor/bobina. (1.65V/A)
const int sns1 = A1;
// entradas pulsadores (conector en la shield)
const int pinA2 = A2; // Pulsar para ir a la via
const int pinA3 = A3;
const int pinD5 = 5;
    const int pinD6 = 6;
const int pinSDA = A4;
// entrada final de carrera
const int pinSCL = A5; // Para buscar el punto 0

#define numVias 5 // Numero de vias/pulsadores de nuestra plataforma
deslizante

#define Vial 600 // Pasos en que estan las vias. Deteminados por prueba y
error
#define Via2 750
#define Via3 900
#define Via4 1050
#define Via5 1200

#define FinCarrera pinSCL // Para que sea mas facil de interpretar listado

struct PuntoParo { // struct de datos para el punto, contiene posicion y
pulsador asociado
    int pulsador;
    long posicion;
};

PuntoParo Via[] = { // Array de puntos de parada
    { pinA2, Vial },
    { pinA3, Via2 },
    { pinD5, Via3 },
    { pinD6, Via4 },
    { pinSDA, Via5},
```

```

};

#define MaxVelocidad 200 // Velocidad normal (pasos por segundo)
#define IniVelocidad 100 // Velocidad al buscar final de carrera
#define ZeroVelocidad 50 // Velocidad al buscar el paso 0
#define Aceleracion 20 // Aceleracion (pasos por segundo al cuadrado)

// Define el objeto y el tipo de interface para AccelStepper:
// Con Arduino motor shield rev3 usar 2 -> FULL2WIRE < 2 wire stepper, 2 motor pins required (H-Bridge)
// Con Pololu A4988/DRV8825 usar 1 -> DRIVER < Stepper Driver, 2 driver pins required (driver board)
// Con motor unipolar y ULN2803 usar 4 -> FULL4WIRE < 4 wire full stepper, 4 motor pins required (4 transistors)
// AccelStepper stepper( 4, pinA1, pinA2, pinB1, pinB2);

AccelStepper stepper( 2, dirA, dirB); // se usan pines dir_a y dir_b

void setup()
{
    pinMode(pwmA, OUTPUT); // Inicializa pines de salida del Arduino motor shield
    pinMode(pwmB, OUTPUT);
    pinMode(brakeA, OUTPUT);
    pinMode(brakeB, OUTPUT);

    digitalWrite(pwmA, HIGH);
    digitalWrite(pwmB, HIGH);
    digitalWrite(brakeA, LOW);
    digitalWrite(brakeB, LOW);

    pinMode(FinCarrera, INPUT_PULLUP); // Inicializa pines de entrada (pulsadores y final de carrera)
    for (int n=0; n<numVias; n++)
        pinMode(Via[n].pulsador, INPUT_PULLUP);

    BuscaZero(); // Busca el paso 0

    stepper.setMaxSpeed(MaxVelocidad);
    stepper.setAcceleration(Aceleracion);
    stepper.moveTo(Via[0].posicion); // Vamos a la primera via
}

void loop(){
    if (stepper.distanceToGo() == 0) {
        stepper.run(); // let the AccelStepper to disable motor current after stop
        CompruebaBotones();
    }
    stepper.run();
}

void BuscaZero () {
    long distancia;

    distancia = Via[0].posicion / 2; // Para asegurar encontrar el final de carrera, preveemos movernos
    distancia += Via[numVias -1].posicion; // la distancia a la ultima via mas la mitad hasta la primera
    stepper.setMaxSpeed(IniVelocidad);
    stepper.setAcceleration(Aceleracion);
    stepper.moveTo(-distancia); // Posicion sera relativa al punto actual (antihorario)

    while (digitalRead(FinCarrera) == HIGH) { // Mover mientras no se pulse el final de carrera
        stepper.run();
    }

    stepper.setCurrentPosition(0); // Paso 0 provisional
    distancia = IniVelocidad; // alejarse un segundo de velocidad de busqueda
    stepper.setMaxSpeed(ZeroVelocidad); // pero mas despacio para menos inercia
    stepper.moveTo(distancia); // Distancia absoluta (horario)
    while (stepper.distanceToGo() != 0) { // Mover hasta alejarse para soltar el final de carrera
        stepper.run();
    }
    stepper.moveTo(-distancia); // Distancia absoluta desde el cero provisional para asegura que toca
    do { // Movemos hasta que pulse el final de carrera
        stepper.run();
    }
}

```

```

} while (digitalRead(FinCarrera) == HIGH);

stepper.setCurrentPosition(0); // ya hemos encontrado el paso 0
delay(1000); // Esperamos para que nos vean
}

void CompruebaBotones() {
    for (int n=0; n< numVias; n++) {
        if (digitalRead(Via[n].pulsador) == LOW) // Mira si esta pulsado
            stepper.moveTo(Via[n].posicion); // pone el nuevo destino
    }
}

```

Primero incluimos la librería y definimos todos los pines de la placa *shield*, de esta manera tendremos información de cómo está conectada y los pines ocupados por si mas tarde añadimos otras funcionalidades.

También definimos cuantas vías tenemos y las posiciones en pasos desde el final de carrera obtenidas a base de prueba y error. (Dependerá de los pasos por vuelta del motor, la placa de control, si se usa correa o varilla, y la distancia entre vías)

Luego renombramos el fin de carrera a algo más coherente que la señal en que está conectada.

Ahora definimos nuestra estructura de datos para los puntos de paro con **struct**, contendrá dos variables: **pulsador**, que será el pin asociado del pulsador para ir a esa vía y **posicion**, pasos donde está situada la vía (un **long** como requiere luego la librería)

Con esta estructura hacemos un array de puntos de paro llamado **Via[]** y los inicializamos. Observar las **{ }** para ir asignando valores a cada una de las variables de la **struct** y las , para el array y dentro de cada struct

Definimos también la aceleración y diferentes valores para velocidad para que sea fácil adaptarlas, tendremos una velocidad de desplazamiento normal, otra más lenta para ir al final de carrera y otra aún más lenta para determinar el paso 0.

Finalmente definimos el objeto **stepper** con los valores adecuados para inicializar la librería **AccelStepper** según la placa de control utilizada.

En **setup()** colocamos los pines de la shield con los valores adecuados. También hacemos un bucle para inicializar los pines de los pulsadores como entrada, estos valores están en el array de puntos de paro **Via[]**, concretamente en la variable **pulsador**, observar el . entre ambos nombres.

Hemos escrito una función llamada **BuscaZero()** que nos hará el ciclo de búsqueda del paso 0. En ella tendremos una variable local llamada **distanzia** para ir calculando las posiciones a las que prevemos movernos, es tipo **long** como necesitan las funciones de la librería.

Para buscar el paso 0 en la función **BuscaZero()** haremos que la plataforma se mueva hacia el final de carrera, para asegurarnos que lo tocará tendrá que desplazarse al menos la distancia de la última vía y poco más. Ese poco más puede ser cualquier distancia pero para hacer el programa lo más genérico posible haremos que sea la mitad del recorrido hasta la primera vía.

Con las funciones de la librería estableceremos el movimiento que deseamos, con **setMaxSpeed()** pondremos la velocidad de búsqueda, con **setAcceleration()** la aceleración y con **move()** diremos que se mueva la distancia que hemos calculado en modo relativo (desde la posición actual), el signo negativo hará que se acerque al final de carrera

Para que se mueva hay que llamar a la función **run()** al menos una vez por paso, así que utilizaremos un bucle **while()** que llamará a **run()** mientras el final de carrera este a **HIGH** (sin pulsar).

Cuando se pulse finalizará el bucle y con **setCurrentPosition()** estableceremos provisionalmente el último paso como paso 0. Como por las inercias del movimiento ese puede que no sea exactamente el paso 0, nos alejaremos más despacio una pequeña distancia y lo volveremos a buscar más lentamente.

La distancia puede ser cualquiera que nos asegure que el final de carrera no está pulsado, podríamos movernos y mirar el pulsador a ver cuando deja de estar pulsado pero vamos a movernos una distancia concreta. La velocidad a que nos movíamos la habíamos establecido en pasos por segundo, así que si queremos retroceder a la posición donde estábamos un segundo antes de tocar el final de carrera, la distancia en pasos será igual a la velocidad (**IniVelocidad**)

Ahora ponemos una velocidad más lenta con **setMaxSpeed()** y establecemos la posición absoluta propuesta desde el paso 0 provisional con **moveTo()**. Con otro bucle **while()** ejecutamos **run()** pero ahora la condición es que hayamos hecho el recorrido, o sea que la distancia para llegar, que devuelve la función **distanceToGo()** sea 0

Cuando ha finalizado el movimiento establecemos un nuevo movimiento para llegar a esta misma distancia pero al otro lado del paso 0 provisional (movimiento absoluto y signo negativo) así nos aseguramos que tocará el final de carrera.

Con un nuevo bucle, pero esta vez `do...while()` moveremos con `run()` mientras el final de carrera no esté pulsado (a **HIGH**). Cuando lo toque finalizará el bucle y estableceremos este como el paso 0 definitivo. Finalmente con `delay()` esperaremos un segundo para que se vea donde ha tocado.

Después de la búsqueda del paso 0 en el `setup()` solo nos queda poner la velocidad y aceleración normal e indicar que queremos ir a la posición de la primera vía. El primer valor (índice 0) valor `posicion` del array de puntos de paro `Via[]`

En el `loop()` solo hay que ir ejecutando `run()` para que se mueva el motor paso a paso si es necesario y comprobar si hemos llegado a un destino, la función `distanceToGo()` devuelve 0, en ese caso comprobaremos los botones con nuestra función `CompruebaBotones()` para ver si tenemos que ir a otro destino.

La función `CompruebaBotones()` simplemente hace un bucle `for()` por todos los botones, que están en la variable `pulsador` del array de puntos de paro y si esta pulsado (**LOW**) establece el nuevo destino con `moveTo()` a la posición absoluta guardada en la variable `posicion` del array de puntos de paro para ese índice. En el `loop()` la función `run()` irá moviendo el motor.

Nota: Si se pulsan varios botones, como el bucle `for()` recorre los pulsadores del primero al último se irán estableciendo los destinos, por lo que el destino final será el del pulsador con el número de vía más alta

10. Los Strings: Botón Pulsador de Acción

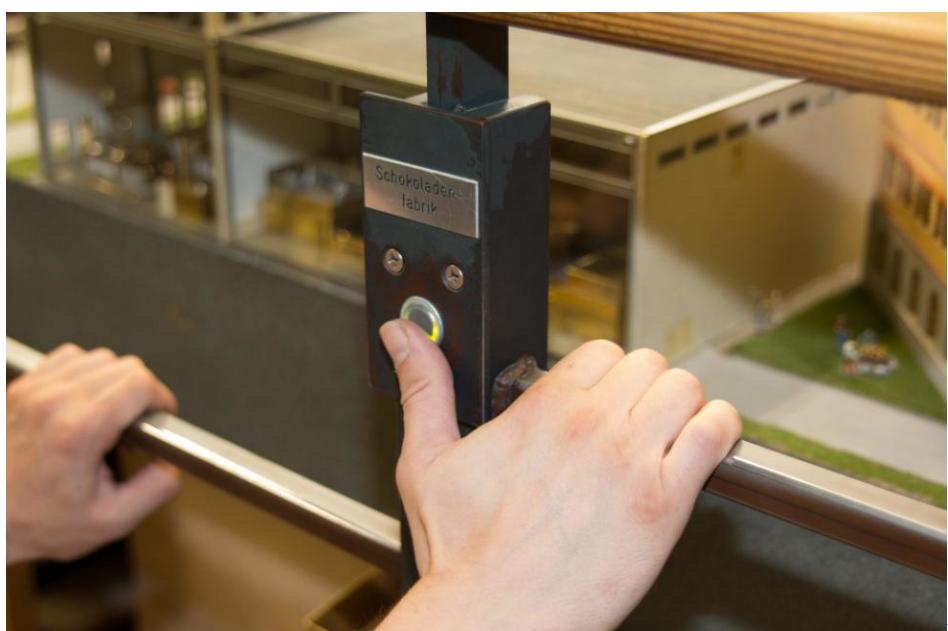
Los Strings en Arduino son parecidos a los arrays de caracteres pero con más posibilidades ya que se pueden usar ciertas operaciones con ellos que con arrays habría que implementar con código. Serían como un objeto de una librería con sus funciones propias.

<https://www.arduino.cc/reference/en/language/variables/data-types/stringobject/>

Al contrario que los arrays no ocupan una posición de memoria fija sino que se asigna dinámicamente por lo que son más lentos y tienden a ocupar mucha memoria al manipularlos, a cambio ofrecen más funcionalidad para trabajar con cadenas de caracteres.

Los Strings normalmente se usan para enviar textos al puerto serie o a una pantalla conectada al Arduino. En micros con poca memoria RAM como el Arduino Uno es preferible usar arrays en lugar de Strings o hacer que se almacenen en la ROM en lugar de la RAM

En este ejemplo, para ver cómo funcionan los Strings los vamos a usar para codificar las órdenes que se ejecutarán al pulsar un Botón de Acción como los que hay en la maqueta de Miniatur Wunderland para realizar diferentes animaciones:



O como en mi diseño KDaktion:

http://usuaris.tinet.cat/fmco/dccacc_sp.html#kdaktion



[link](#)

La idea es tener un String con las órdenes codificadas a ejecutar que el Arduino irá interpretando y ejecutando para controlar hasta 3 servos y 10 salidas digitales de forma que sea fácil de modificar para adaptarlas a los diferentes efectos y animaciones que queramos realizar al pulsar el botón.

Las órdenes serán un String formado por simples caracteres en mayúsculas, seguidas de números si la orden necesita un parámetro. Por ejemplo "**ILOH3P1000L3H0**"

las órdenes serán las siguientes:

I: Espera la pulsación del Botón de Acción

H: Pone a nivel HIGH una salida digital, necesita un parámetro (entre 0 y 9) que será la salida

L: Pone a nivel LOW una salida digital, necesita un parámetro (entre 0 y 9) que será la salida

P: Hace una pausa, necesita un parámetro que será la duración en milisegundos (hasta 65535)

En el String de ejemplo, si se conecta un LED en la salida 0 y otro en la 3 se realizaría la siguiente secuencia:

Se espera a que se pulse el Botón de Acción, se apaga el LED de la salida 0, se enciende el de la salida 3, hace una pausa de 1000ms (un segundo) apaga el LED de la salida 3 y enciende el de la salida 0.

Haremos que una vez ejecutado el String, se vuelva a cargar, de forma que se repita. En el ejemplo, volvería a esperar la pulsación del Botón de Acción.

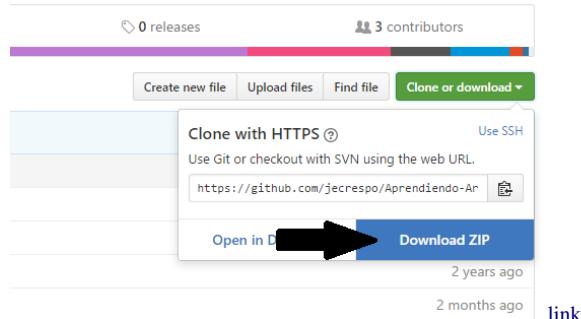
También tendremos un String que se ejecutará una sola vez al dar tensión al Arduino o hacer un reset de forma que podremos colocar las salidas en una determinada posición inicial.

Para los servos definimos otras órdenes relacionadas con ellos, como para las posibles animaciones nos interesa controlar la velocidad de movimiento de los servos en lugar de la librería Servo.h vamos a usar la librería **VarSpeedServo.h**

https://github.com/netlabtoolkit/VarSpeedServo?utm_source=rb-community&utm_medium=forum&utm_campaign=better-arduino-servo-library

Esta librería nos permite controlar hasta 8 servos y controlar su velocidad de movimiento (0: máxima velocidad, 1 a 255: velocidad desde lenta a rápida).

Para instalarla, la descargamos en formato ZIP, y en el Arduino IDE vamos a **Programa->Incluir Librería->Añadir Librería ZIP...**





Las órdenes que podremos colocar en el String son:

X: Servo X, necesita un parámetro que será la posición a la que se ha de mover (entre 0 y 180)

Y: Servo Y, necesita un parámetro que será la posición a la que se ha de mover (entre 0 y 180)

Z: Servo Z, necesita un parámetro que será la posición a la que se ha de mover (entre 0 y 180)

V: velocidad de movimiento del servo, necesita un parámetro que será la velocidad (entre 0 y 255). Se ha de usar antes de X, Y o Z para que les afecte.

W: Espera a que acabe el movimiento de los servos y los desconecta para evitar vibraciones

También vamos a implementar unas órdenes para poder repetir una serie de acciones un determinado número de veces, lo que nos facilitará hacer parpadeos de un LED, por ejemplo.

R: Repite las órdenes hasta la orden F, necesita un parámetro que es el número de repeticiones (entre 2 y 65535)

F: Fin del bucle, vuelve al principio del bucle si no se han realizado todas la repeticiones.

En este ejemplo: "**R10H3P500L3P500F**", repetimos 10 veces: encender el LED de la salida 3, hacemos una pausa de 0,5s, apagamos el LED y hacemos otra pausa de 0,5s. Lo que nos dará un parpadeo durante 10 segundos.

Finalmente para que nos sea fácil identificar las diferentes fases de la animación tendremos una orden que nos permitirá escribir anotaciones.

<: Ignora el texto hasta >

Así una animación completa podría quedar en un String como este, fácil de modificar e interpretar:

"<inicio>IH0H3P1000L3<servo>X25WX150W<parpadeo>R10H3P500L3P500F<fin>LO"

El programa sería este:

```
// Control de un pulsador de acción con tres servos y 10 salidas digitales - Paco Cañada 2019

#include <VarSpeedServo.h> // https://github.com/netlab toolkit/VarSpeedServo?utm_source=rb-community&utm_medium=forum&utm_campaign=better-arduino-servo-library

/* PulsadorAcción lee y ejecuta un String que contiene los siguientes comandos:
 *  Xnn Ynn Znn   Servo a posición nn
 *  Vnn           Velocidad servo nn
 *  W             Espere hasta que acabe el movimiento de los servos
 *  Hn            Salida digital n a HIGH
 *  Ln            Salida digital n a LOW
 *  I             Espera pulsador
 *  Pnn           Pausa de nn ms (máximo 65535ms)
 *  Rnn           Repetir n veces hasta F
 *  F             Fin de bucle, vuelve a R hasta ejecutar n veces
 *  <...>         Comentario
 */

// Acciones que se ejecutan tras el reset
String Reset      = "H0<servo inicio>V20X25Y50Z180";
// Acciones que se repiten
```

```

String Acciones = "<boton>WL0IH0<acciones>H4P1000L4x100y20z50WH3X25Y50Z180WL3<parpadeo espera
final>R10H0P500L0P500F";

#define servoXpin 12 // Definicion de los pines
#define servoYpin 11
#define servoZpin 10
#define pulsadorPin 13

int digitalPins[] = {A0,A1,2,3,4,5,6,7,8,9};

VarSpeedServo servoX; // Objetos de la libreria VarSpeedServo
VarSpeedServo servoY;
VarSpeedServo servoZ;

String Operacion; // Acciones a ejecutar
int pos; // Posicion en el String Operacion que se esta
decodificando
unsigned int parametro; // Valor del parametro
int velocidad; // Velocidad del servo
int bucle=0; // contador para el bucle
int buclepos=0; // posicion a la que ha de volver el bucle

void setup() {
  int n;

  for (n=0; n < sizeof(digitalPins); n++) { // Inicializa salidas digitales
    pinMode (digitalPins[n], OUTPUT);
    digitalWrite (digitalPins[n], LOW);
  }
  pinMode (pulsadorPin,INPUT_PULLUP); // Inicializa pin pulsador
  velocidad = 30; // velocidad servo por defecto
  pos = 0; // inicializa decodificacion al inicio del String
  Operacion = Reset; // Primero ejecutamos el String Reset
  Operacion.toUpperCase(); // Pone los comandos en mayusculas
}

void loop() {
  if (pos >= Operacion.length()) { // Si hemos alcanzado el final del String, cargamos las
acciones
    Operacion = Acciones; // Pone los comandos en mayusculas
    Operacion.toUpperCase(); // inicializa el puntero de decodificacion
  }
  if (isAlpha (Operacion.charAt(pos)) || isPunct (Operacion.charAt(pos))) {
    switch (Operacion.charAt(pos++)) {
      case 'I': // Esperar a que se pulse el boton
        while (digitalRead(pulsadorPin) == HIGH);
        break;
      case 'H': // salida digital a HIGH
        parametro = ExtraeNumero(9);
        digitalWrite (digitalPins[parametro],HIGH);
        break;
      case 'L': // salida digital a LOW
        parametro = ExtraeNumero(9);
        digitalWrite (digitalPins[parametro],LOW);
        break;
      case 'P': // Pausa de ms
        parametro = ExtraeNumero(65535);
        delay (parametro);
        break;
      case 'V': // velocidad del servo
        velocidad = ExtraeNumero(255);
        break;
      case 'X': // Mueve Servo X
        parametro = ExtraeNumero(180);
        servoX.attach(servoXpin);
        servoX.write(parametro,velocidad);
        break;
      case 'Y': // Mueve Servo Y
        parametro = ExtraeNumero(180);
        servoY.attach(servoYpin);
        servoY.write(parametro,velocidad);
        break;
      case 'Z': // Mueve Servo Z
        parametro = ExtraeNumero(180);
        servoZ.attach(servoZpin);
        servoZ.write(parametro,velocidad);
        break;
      case 'W': // Espera a que acabe el movimiento de los servos
    }
  }
}

```

```

servoX.wait();                                // Espera a Servo X
servoX.detach();
servoY.wait();                                // Espera a Servo Y
servoY.detach();
servoZ.wait();                                // Espera a Servo Z
servoZ.detach();
break;
case 'R':
    bucle = ExtraeNumero(65535);           // Lee el numero de repeticiones
    buclepos = pos;                         // guarda la posicion a volver
    break;
case 'F':
    bucle--;                               // decrementa contador de bucle
    if (bucle > 0)                      // si quedan bucles por hacer, cambia la posicion de decodificacion
        pos = buclepos;
    break;
case '<':
    parametro = Operacion.indexOf('>', pos);
    if (parametro == -1)
        pos = Operacion.length();
    else
        pos = parametro + 1;
    break;
}
else {
    ExtraeNumero(0);
}
}

int ExtraeNumero (unsigned int maximo) {
String numero;
unsigned int valor;

numero = "";
while (isDigit (Operacion.charAt(pos)) && (pos < Operacion.length()) ) { // lee caracter mientras
sea un numero
    numero.concat (Operacion.charAt(pos++));                           // y lo añade al nuevo string
}
valor = min (numero.toInt(), maximo);          // convierte el string en numero
sin que pase del maximo
return (valor);                                // devuelve el valor obtenido
}

```

Primero incluimos la librería **VarSpeedServo.h** y definimos dos Strings con las órdenes a ejecutar.

El String **Reset** serán las órdenes que se ejecutarán únicamente al dar tensión al Arduino o hacer un reset y nos servirá para posicionar inicialmente los servos y salidas.

El String **Acciones** serán las órdenes que queremos ejecutar para realizar la animación. Una vez ejecutada se volverá a cargar.

Después definimos los pines donde están conectados los servos y el Botón Pulsador.

También definimos un array con los pines dónde van conectadas las 10 salidas digitales. Ya que los pines D0 y D1 son usados normalmente por el puerto serie (por si nos hiciera falta usarlo en un futuro) hemos definido las salidas 0 y 1 en los pines A0 y A1 respectivamente.

Luego definimos los objetos de la librería, en este caso los tres servos y una serie de variables globales. **Operacion** será el String que contiene las órdenes que se están ejecutando. **pos** almacenará el índice dentro del String de la instrucción que se interpretará. **parametro** guardará el número leído que corresponde al parámetro de la instrucción, **velocidad** es uno de esos parámetros leídos que corresponde a la instrucción V (velocidad del servo) y finalmente dos variables para poder ejecutar un bucle, **bucle** que almacena el número de repeticiones y **buclepos** que guardará el índice dentro del String donde ha de volver si queda repeticiones por hacer.

En el **setup()** inicializamos los pines digitales de salida leyéndolos del array **digitalPins[]**. En el bucle **for** para el límite en lugar de poner 10 usamos la función **sizeof()** que nos devuelve el tamaño de la variable pasada como parámetro, así si cambiamos la cantidad de pines en el array no tendremos que modificar el valor del límite del bucle **for** ya que **sizeof()** nos lo calculará.

Luego inicializamos el pin del pulsador como entrada. También daremos una velocidad por defecto a los servos por si en el String de la animación no usamos la orden V.

Ponemos a cero el índice `pos` para que decodifique el String desde el principio y copiamos en el String `Operacion` el String `Reset` simplemente usando el operador de asignación =

Con arrays para hacer la copia tendríamos que conocer previamente la longitud del array para hacer un bucle que fuera copiando carácter a carácter de uno a otro y que el array de `Operacion` fuera lo suficiente grande para contener al array de `reset`

Para finalizar se usa la función `toUpperCase()` que pasa a mayúsculas todo el contenido de un String por si por descuido hemos usado minúsculas para las órdenes.

En el `loop()` primero comprobamos que el índice `pos` para la interpretación no haya llegado al final del String `Operacion`, para ello lo comparamos con la función `length()` que nos da la longitud actual del String. Si hemos llegado al final copiamos el String `Acciones` en `Operacion`, ponemos las órdenes en mayúsculas y ponemos el índice de nuevo al principio.

NOTA: La memoria del Arduino es limitada y necesitamos que se puedan almacenar los Strings tanto el original como la copia, por ello la longitud máxima del String queda limitada por el mismo y por las variables usadas. Vigilad de no abusar de los comentarios.

Ahora leemos en la posición del índice del String el carácter con `charAt()`, este debería corresponder a una orden, que es una letra o un signo de puntuación.

En el `if` hacemos la comparación, si el carácter es alfabético, la función `isAlpha()` devuelve `true`, si fuera un signo de puntuación la función `isPunct()` devolvería `true`. Usamos la operación lógica || que devuelve `true` si `isAlpha()` o `isPunct()` ha devuelto `true`.

Si el carácter era una orden, usamos un `switch` para comparar el carácter leído con cada una de nuestras posibles órdenes.

Observad que en este `charAt()` hemos post-incrementado el índice con `++` así lo dejamos apuntando al siguiente carácter que puede ser un parámetro para esta orden o una nueva orden.

En el `case` de la orden I leemos la entrada mediante un bucle `while` mientras siga a `HIGH` (sin pulsar), al pulsar será `LOW` y acabará el bucle. Observad que como no tenemos que hacer nada más, no hemos usado las `}` y hemos puesto el ; de fin de instrucción

En el `case` de la orden H, como necesitamos un parámetro numérico, llamamos a nuestra función `ExtraeNumero()` para leer el parámetro y lo usamos para poner a `HIGH` con `digitalWrite()` el pin correspondiente almacenado en el array `digitalPins[]`

A nuestra función `ExtraeNumero()` le vamos a pasar un parámetro que será el número más alto que podemos devolver por si hemos cometido un error al escribir el String con las órdenes. Inicializamos el String `numero`, que usaremos para ir guardando los números leídos, a una cadena vacía: "".

Realizamos un bucle `while` en el que comprobamos que el carácter apuntado por el índice `pos` es un número con `isDigit()`, que devolverá `true`, y no hemos llegado al final del String de órdenes por lo que la longitud del String devuelta por `length()` es menor que `pos`. El operador && es el y lógico.

Con `charAt()` se lee el carácter numérico y lo añadimos con `concat()` al String `numero`. Además avanzamos el índice con `++`

Si no era un número, hemos acabado de extraer el parámetro numérico y `numero` contendrá los caracteres del número, así que con `toInt()` lo convertimos en un número entero.

Además comprobamos que no exceda del máximo, para ello la función `min()` devuelve el menor de los valores, nuestro número convertido o el valor máximo que pasamos como parámetro a la función `ExtraeNumero()`. Finalmente devolvemos con `return()` el valor.

Para el `case` de la orden L el proceso es el mismo que para la orden H pero poniendo la salida a `LOW`

Para la orden P, leemos el número desde el String con `ExtraeNumero()` y lo usamos como parámetro para `delay()`

El `case` de V simplemente lee el número con `ExtraeNumero()` y lo guarda en la variable `velocidad`.

Para los servos, las tres órdenes X, Y y Z son iguales, conectar el servo a su pin con `attach()`, leer el número que indica la posición a donde se tiene que mover desde el String y pasarlo como parámetro a la función `write()` de la librería `VarSpeedServo`, el segundo parámetro es la velocidad que lo tenemos guardado en la variable `velocidad`.

Para la orden W simplemente usamos la función `wait()` de la librería con cada uno de los objetos `VarSpeedServo` lo que hará que se espere a que todos los servos hayan acabado su movimiento para continuar. Además los desconectamos con `detach()` para evitar vibraciones.

En las ordenes para realizar un bucle necesitamos saber cuántas veces tenemos que repetir el bucle que lo guardaremos en la variable `bucle` y dónde tenemos que volver en la secuencia en cada repetición que lo guardamos en la variable `buclepos`.

En el `case` R leemos el número de repeticiones con `ExtraeNumero()` y lo guardamos en la variable `bucle`. Ahora el índice `pos` apunta a la primera orden que se ha de repetir así que lo copiamos en `buclepos`.

Al final del bucle con la orden F decrementamos `bucle`, el número de repeticiones, con el operador `--`, si aún queda por realizar algún bucle modificamos el índice `pos` con el valor `buclepos` de forma que la siguiente orden que decodificaremos será la primera del bucle.

NOTA: No se puede anidar los bucles (poner un bucle dentro de otro). Esto lo dejo como ejercicio 😊

Para los comentarios que empiezan con < tendremos que buscar el carácter > que indica el final del comentario. La función `indexOf()` localiza el parámetro, un carácter, dentro del String. Si lo localiza devuelve la posición en la que lo ha localizado y si no devuelve el valor -1

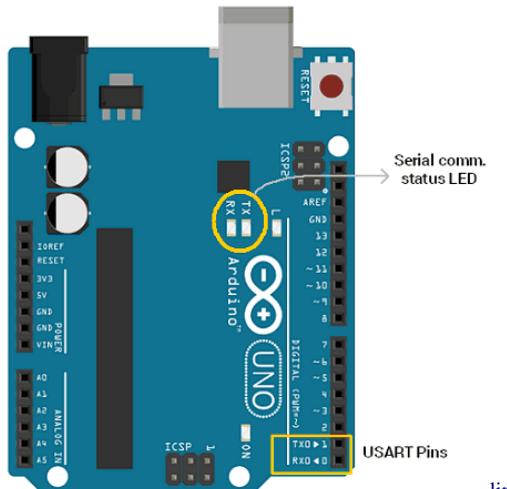
Si hay varios comentarios debemos usar `indexOf()` con dos parámetros, el carácter y la posición desde donde buscar, `pos` en este caso

Si encuentra el carácter > actualizamos el índice `pos` con la posición siguiente al carácter >, pero si no lo localiza actualizamos `pos` con la longitud del String de forma que apunte a su final.

Solo nos queda poner en el `else` qué hacer si por error la orden no era un carácter alfabético o un signo de puntuación, por ejemplo, hemos escrito el parámetro pero se nos ha olvidado escribir la orden. En este caso lo leeremos con `ExtraeNumero()` pero no haremos nada con él, así avanzará el índice `pos` hasta una nueva orden.

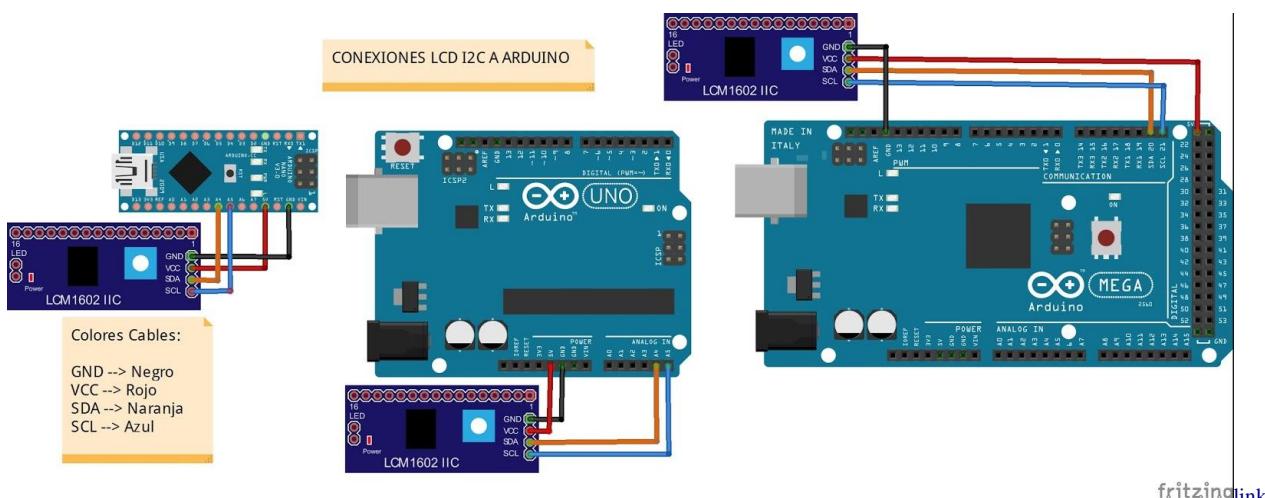
11. Comunicaciones (Serie e I2C): Reloj-Pantalla de andén

Algunos puertos de entrada/salida del Arduino tienen funciones alternativas que permiten establecer comunicaciones con otros dispositivos. El más conocido es el interface Serie (pines D0:RX y D1:TX) que es el que está conectado al chip del conector USB por donde cargamos el programa en nuestro Arduino.



[link](#)

Otro interface es el I2C (pines A4: SDA y A5: SCL) que permite comunicarse con pantallas LCD y OLED, relojes RTC, expansiones de puertos, módulos PWM, memorias EEPROM, inclinómetros, etc.



[fritzing](#) [link](#)

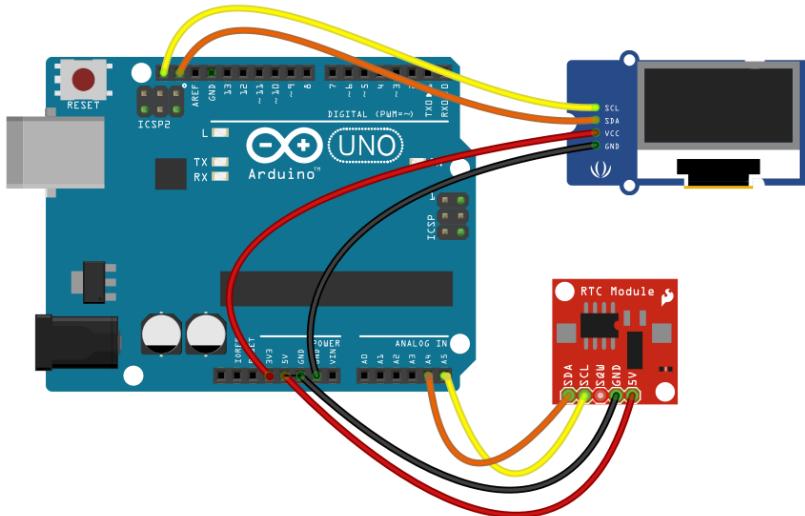
El interface SPI (pines D13: SCK, D12: MISO, D11: MOSI y D10: SS) permiten comunicarse con lectoras de tarjetas SD, pantallas OLED, módulos Ethernet, etc.

En este proyecto vamos a usar el bus I2C para comunicarnos con una pantalla OLED y un reloj RTC (Reloj de Tiempo Real) para mostrar la hora real y los mensajes de los trenes como en una pantalla de andén enviando órdenes desde el puerto serie.



[link](#)

Este es el esquema de conexión:



[link](#)

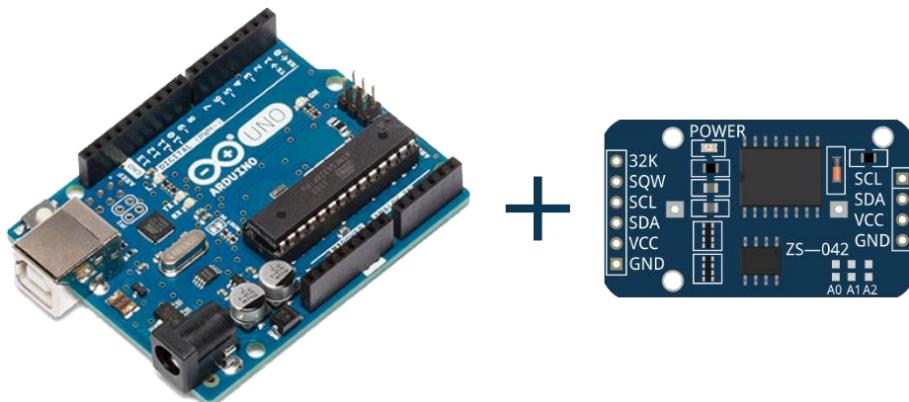
El interface I2C requiere que incluyamos la librería **Wire** del entorno Arduino:

<https://www.arduino.cc/en/Reference/Wire>

Para el reloj RTC vamos a usar un módulo con el chip DS3231, que además tiene una memoria EEPROM pero que no usaremos en este proyecto. Este módulo tiene una pila de botón tipo CR2032 que mantiene el cómputo de la hora aunque se quite la alimentación.

Además el DS3231 tiene un sensor de temperatura que aunque no es muy preciso ($\pm 3^{\circ}\text{C}$) como otros módulos específicos para temperatura como el DHT11 le dará un plus a nuestra pantalla de andén.

El reloj no solo mantiene la hora, minutos y segundos, también mantiene la fecha (día, mes y año) y el día de la semana. Incluso se le pueden programar un par de alarmas aunque nosotros en este proyecto solo usaremos la hora actual.



[link](#)

De las librerías disponibles para los módulos RTC que soporten el DS3231, he escogido la **uRTCLib** disponible a través del Gestor de Librerías ya que parece más ligera que otras.

<https://github.com/Naguissa/uRTCLib>

Como pantalla OLED conectada al bus I2C usaremos una con el chip SSD1306 de 128x32 pixels (0.91") que permite unas 4 líneas de caracteres, es rectangular y parece más adecuada para un andén que una cuadrada de 128x64 (0.96"). Estas pantallas se alimentan a 3.3V

EC Buying



OLED LCD Display Module [link](#)

El bus I2C es lento comparado con el SPI para transferir datos a una pantalla, pero para pequeños tamaños de imagen sigue siendo válido. Como ventaja, solo usa dos pines para comunicarse con ella.

Para estas pantallas hay varias librerías, una de las más usadas es la de Adafruit:

https://github.com/adafruit/Adafruit_SSD1306

Esta librería permite escribir texto y dibujar gráficos en la pantalla, aunque para ello necesita de otra librería, la [Adafruit-GFX-Library](#) que tiene las primitivas de gráficos y una gran variedad de fuentes de texto:

<https://github.com/adafruit/Adafruit-GFX-Library>

El problema con esta librería es que necesita mucha RAM ya que usa un buffer del tamaño de la pantalla completa para las escrituras que luego transfiere en bloque para tener una adecuada velocidad de refresco. En micros con poca memoria RAM como el Arduino Uno deja poco espacio para las variables y strings.

Otra alternativa es la [u8glib](#), esta usa un buffer más pequeño pero requiere que el repintado de la pantalla se repita en un bucle con lo que la velocidad de refresco es lenta.

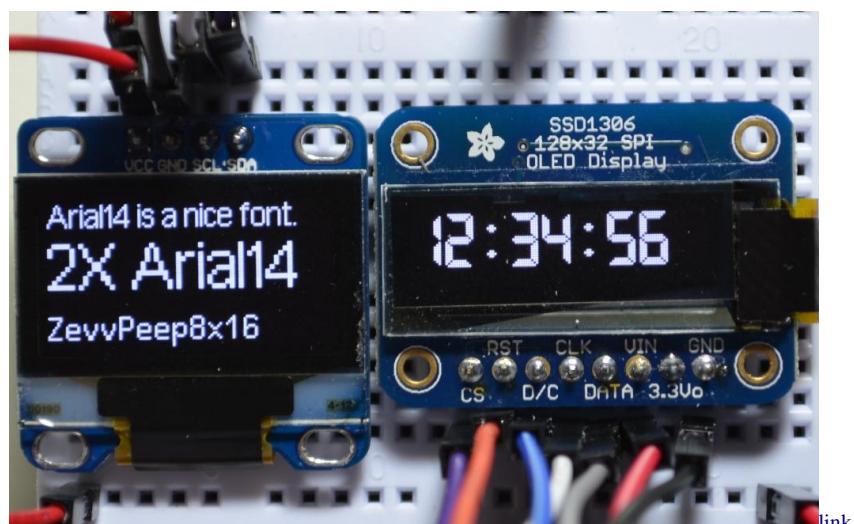
<https://github.com/olikraus/u8glib>

Una nueva versión de esta librería es la [u8g2](#) en la que se puede elegir entre el modo de buffer completo o parcial para adaptarlo a la capacidad de nuestro Arduino.

<https://github.com/olikraus/u8g2>

Finalmente he escogido la librería [SSD1306Ascii](#), que requiere muy poca cantidad de RAM, como contrapartida solo permite texto, aunque tiene bastantes fuentes de texto.

<https://github.com/greiman/SSD1306Ascii>

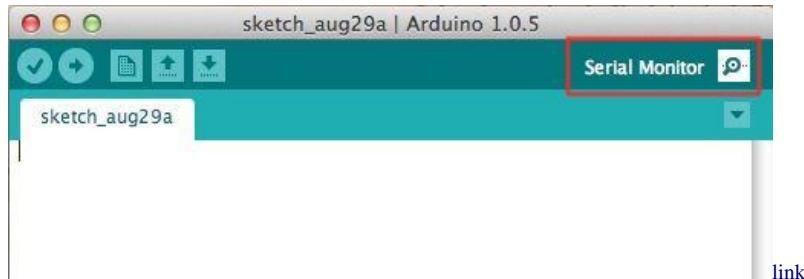


[link](#)

Para el puerto serie utilizaremos las funciones incluidas en Arduino del objeto **Serial**:

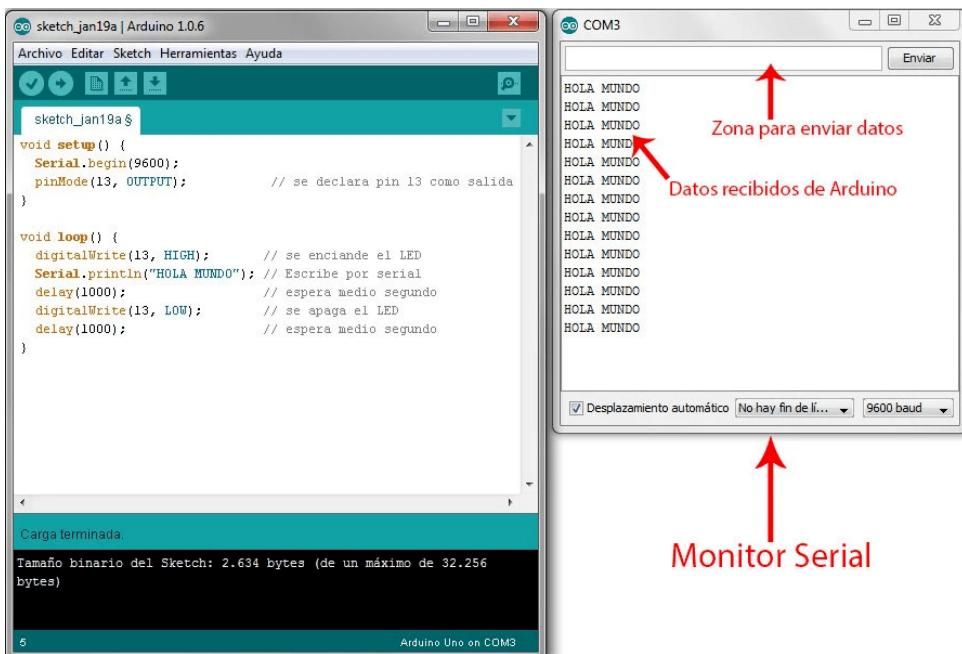
<https://www.arduino.cc/reference/en/language/functions/communication/serial/>

Nos comunicaremos con el Arduino desde el Monitor Serie. Se accede desde el menú **Herramientas -> Monitor Serie** o desde el ícono de la parte superior derecha del Arduino IDE. En **Herramientas -> Puerto** tiene que elegirse el puerto que corresponde a nuestro Arduino.



El monitor de puerto serie es una pequeña utilidad integrada dentro del Arduino IDE que nos permite enviar y recibir fácilmente información a través del puerto serie. Su uso es muy sencillo, y dispone de dos zonas, una que muestra los datos recibidos, y otra para enviarlos.

En la parte inferior derecha del Monitor Serie tenemos que poner la misma velocidad en baudios que la que definimos en nuestro sketch para visualizar correctamente los datos.



Las funciones principales a utilizar con el puerto Serie son:

Serial.begin(velocidad) inicializa el puerto serie, velocidad serán los baudios por segundo con los que nos comunicaremos.

Serial.print(mensaje) Utilizado para enviar datos como texto. Si mensaje es un número, será enviado como sus caracteres equivalentes para representar ese número; si queremos enviar textos, se escribe entre ' para caracteres, y " para cadenas de texto.

Serial.println(mensaje) Igual que **print()** pero agregando al final el carácter no imprimible retorno de carro (ASCII 13, o '\r'); y el carácter de nueva línea o enter (ASCII 10, o '\n').

Podemos ahorrar memoria RAM usando la macro **F()** si tenemos que enviar grandes cantidades de texto aunque en este proyecto no nos vamos a preocupar en exceso ya que la librería de la pantalla usa poca memoria.

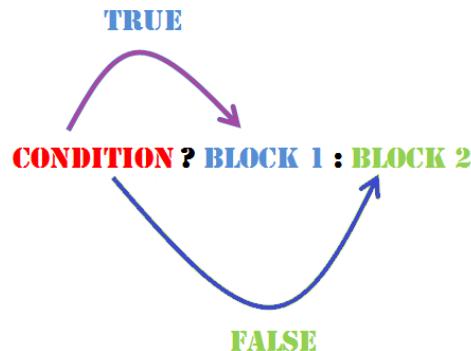
```
Serial.print ("Este texto se copia en RAM");  
Serial.print (F("Este texto no."));
```

Para recibir datos normalmente comprobamos si hay caracteres en el buffer de recepción con `Serial.available()` y en caso de que los haya los leemos uno a uno con `Serial.read()` lo que puede ser algo farragoso para interpretar.

Por suerte hay funciones como `Serial.parseInt()` que leen e interpretan los caracteres numéricos y nos los devuelven como un `int`. La interpretación se detiene si se recibe un carácter no numérico o se acaba el tiempo de espera para recibir caracteres (por defecto, un segundo).

Para recibir textos en un `String` la función `Serial.readStringUntil()` lee la cadena de texto hasta que encuentra el carácter que se le pase como parámetro, normalmente '`\n`' para recibir una línea completa.

Del lenguaje Arduino también veremos cómo escribir los números hexadecimales (se les precede de `0x` para distinguirlos de los textos) y una alternativa al `if-else` como condicional, el operador ternario:



Con esta construcción se escribe la condición a probar luego `?` la instrucción en caso de resultar verdadera, luego `:` y la instrucción en caso de resultar falsa.

Este código sería equivalente:

```
(n>1) ? x = 1 : x = 0 ;      // operador condicional ternario  
if (n > 1)                  // if-else  
  x = 1;  
else  
  x = 0;
```

En nuestro proyecto leeremos únicamente la hora, minutos y segundos del reloj RTC y cada segundo refrescaremos la nueva hora en pantalla. Además cada cierto tiempo leeremos y mostraremos la temperatura durante unos segundos.

Usaremos el Monitor Serie en el que el Arduino nos podrá mostrar un menú de órdenes con el que podremos leer la hora y la temperatura y poner en hora el reloj.

Además las órdenes nos permitirán elegir entre mostrar la hora/temperatura y uno de los varios mensajes definidos en un array, de esta forma tendremos un panel de anuncios de un andén.

El anuncio consistirá en tres líneas, la primera con el tipo de tren, la segunda con el destino con una letra de mayor tamaño y la tercera la haremos móvil para mostrar un mensaje largo del tipo estaciones en las que el tren efectúa parada.

Video: <https://www.youtube.com/watch?v=ld8-SR83kwg>



[link](#)

El programa es el siguiente:

```
// Control de una Pantalla OLED y reloj para panel de estacion - Paco Cañada 2020

#include "Wire.h"           // Libreria I2C
#include "uRTCLib.h"         // Libreria RTC DS3231
https://github.com/Naguissa/uRTCLib/blob/master/src/uRTCLib.h
#include "SSD1306Ascii.h"     // Librerias OLED SSD1306 https://github.com/greiman/SSD1306Ascii
#include "SSD1306AsciiWire.h"

uRTCLib rtc(0x68);          // La direccion I2C del modulo de reloj DS3231

#define I2C_ADDRESS 0x3C        // la direccion I2C de la pantalla OLED SSD1306
#define RST_PIN -1
SSD1306AsciiWire oled;

int hora,minutos,segundos;    // variables para mostrar hora y mensaje
int ultSegundo=99;
bool muestraHora;
int mensaje;

TickerState state;           // Mantiene puntero a la cola de mensajes en movimiento
uint32_t tickTime = 0;

#define maxMensaje 5           // Numero de mensajes en la pantalla. Se definen en el array
text[]

const char* text[] = {
  "RE REGIONAL EXPRESS", "BARCELONA", "EFECTUA PARADA EN VILASECA, TARRAGONA, ALTAFULLA, TORREDEMBARRA, SANT VICENS Y VILANOVA",
  "AVE 03062", "MADRID ATOCHA", "EFECTUA PARADA EN TARRAGONA CAMP, LLEIDA PIRINEUS, ZARAGOZA DELICIAS Y GUADALAJARA",
  "RE REGIONAL", "AEROPUERTO", "DIRECTO A AEROPUERTO",
  "T.HOTEL 10156", "BARCELONA SANTS", "PARA ESTE TREN NO SON VALIDOS LOS BILLETES DE CERCANIAS",
  "HD DIURNO 17194", "BADAJOZ", "EFECTUA PARADA EN LEGANES Y LUEGO SIGUE RECTO"
};

void setup() {
  Serial.begin(115200);          // Usamos 115200 baud para el monitor serie
  Serial.println ("Pantalla Reloj OLED by Paco");
  Serial.println ("? para ver comandos\n");

  Wire.begin();                  // Inicializacion puerto I2C. SDA = A4, SCL = A5
  Wire.setClock(400000L);

  rtc.refresh();                // lee modulo RTC DS3231
  ultSegundo = rtc.second();
  muestraHora = true;

  oled.begin(&Adafruit128x32, I2C_ADDRESS); // Inicializacion pantalla OLED 128x32 SSD1306
  oled.clear();
  oled.setFont(Verdana12_bold);
  oled.setCursor (22,1);
  oled.print ("Design by Paco");
  delay (2000);
}

void loop() {
  if (Serial.available() > 0)      // Si hay caracteres en el puerto serie los interpretamos
    entradaSerie();

  rtc.refresh();                  // Actualizamos desde el reloj RTC la hora actual
  segundos = rtc.second();
  if (muestraHora) {              // Mostrar hora
    if ((ultSegundo != segundos)) { // Cada nuevo segundo actualizamos la pantalla
      minutos = rtc.minute();
      hora = rtc.hour();
      ultSegundo = segundos;
      if ((segundos > 25) && (segundos < 35))
        oledTemp();                // Temperatura
      else {
        oled.setFont(lcdnums14x24); // Hora
        oled.clear();
        oled.setCursor (8,1);       // columna en pixels. fila en 8 filas de pixel.
        oledHora();
      }
    }
  }
}
```

```

        else {                                // Mostrar mensaje movil
            if (tickTime <= millis()) {
                tickTime = millis() + 30;
                int8_t rtn = oled.tickerTick(&state);
                if (rtn <= 0)           // Si fin de mensaje, vuelve a cargarlo
                    oled.tickerText(&state, text[(mensaje*3)-1]);
            }
        }
    }

void entradaSerie () {
    int c = Serial.read ();                  // Lee caracter del comando
    switch (c) {
        case '#':                         // Comando #HH:MM:SS poner en hora
            hora = Serial.parseInt();
            minutos = Serial.parseInt();
            segundos = Serial.parseInt();
            hora = constrain(hora, 0, 23);
            minutos = constrain(minutos, 0, 59);
            segundos = constrain(segundos, 0, 59);
            // RTCLib::set(byte second, byte minute, byte hour, byte dayOfWeek, byte dayOfMonth, byte
month, byte year)
            rtc.set(segundos, minutos, hora, 1, 23, 02, 20);
            rtc.refresh();
            Serial.print ("Nueva ");
        case '@':                          // Comando @ leer hora
            Serial.print("Hora: ");
            Serial.print(rtc.hour());
            Serial.print(':');
            Serial.print(rtc.minute());
            Serial.print(':');
            Serial.println(rtc.second());
            break;
        case 'T':                           // Comando T lee temperatura
            Serial.print("Temp: ");
            Serial.print(rtc.temp());
            Serial.println (" °C");
            break;
        case '-':                           // Comando - muestra hora
            muestraHora = true;
            break;
        case '+':                           // Comando +N muestra mensaje N en la pantalla OLED
            mensaje = Serial.parseInt();
            mensaje = constrain (mensaje,1,maxMensaje);
            muestraHora = false;
            oled.clear ();
            oled.setFont(lcd5x7);
            oled.println (text[(mensaje*3)-3]);          // Primera linea mensaje. Tipo tren
            oled.setFont(Verdana12_bold);
            oled.println (text[(mensaje*3)-2]);          // Segunda linea mensaje. Destino
            oled.tickerInit(&state, Adafruit5x7, 3);   // mensaje movil en linea 3
            break;
        case '\n':                          // filtra caracter nueva linea
        case '\r':                          // filtra caracter retorno del carro
            break;
        case '?':                           // Comando ? ayuda
        default:                           // Cualquier otro comando no soportado muestra ayuda
            Serial.println (F("Introduzca uno de estos comandos:"));
            Serial.println (F("@      Lee hora actual"));
            Serial.println (F("T      Lee temperatura actual"));
            Serial.println (F("#HH:MM:SS Poner en hora"));
            Serial.println (F("+N     Muestra mensaje N en la pantalla"));
            Serial.println (F("-     Muestra hora en la pantalla"));
            Serial.println (F("?     Esta ayuda\n"));
            break;
    }
}

void oledHora() {
    if (hora<10)                      // muestra hora actual en la pantalla OLED
        oled.print ("0");
    oled.print(hora);
    (minutos < 10) ? oled.print (:0") : oled.print (:");
    oled.print(minutos);
    (segundos < 10) ? oled.print (:0") : oled.print (:");
    oled.print(segundos);
}

```

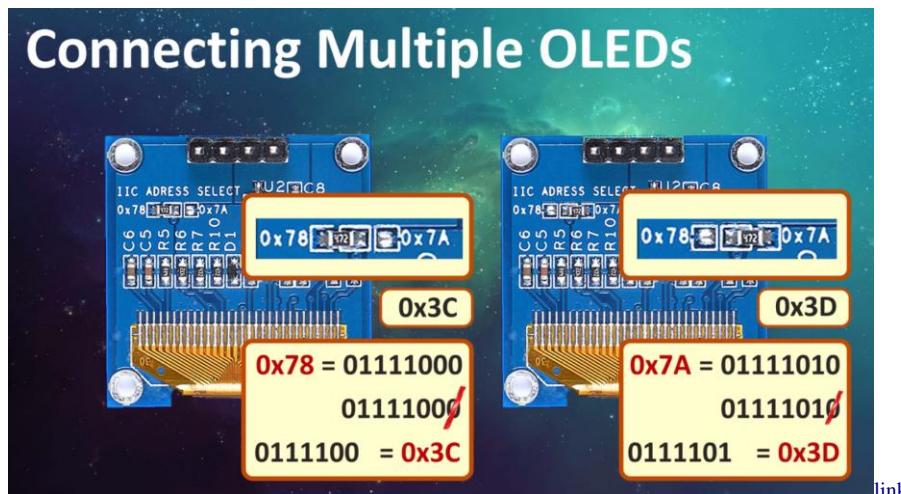
```

void oledTemp () {
    oled.setFont(Verdana12 bold);           // muestra temperatura en pantalla OLED
    oled.clear();                          // columna en pixels. fila en 8 filas de pixel.
    oled.setCursor (40,0);
    oled.set2X ();
    oled.print(int (rtc.temp()));
    oled.print(" °C");
    oled.set1X ();
    oled.setCursor (44,3);
    oled.setFont(lcd5x7);
    oledHora();
}

```

Incluimos las librerías necesarias e inicializamos el objeto rtc para el reloj DS3231 indicando la dirección I2C del mismo (0x68). También indicamos la dirección de la pantalla (0x3C) e inicializamos el objeto oled.

Los chips para I2C tienen una dirección con la que nos comunicamos con ellos, algunos dan la posibilidad de elegir otra colocando unos puentes con lo que se pueden tener varios iguales pero con diferente dirección. Por ejemplo para cambiar dirección del bus I2C, de una pantalla OLED 128x64



[link](#)

Tras definir unas variables para leer la hora (`hora`, `minuto`, `segundo`), el último segundo mostrado (`ultSegundo`), si mostramos la hora (`muestraHora`) y el mensaje que se muestra (`mensaje`) definimos las variables `state` y `tickTime` para la librería `SSD1306Ascii` que nos permitirán manejar una línea de texto móvil en la pantalla.

En `maxMensaje` indicamos el numero de mensajes que podremos elegir mostrar y que están contenidos en el array `text[]`, cada mensaje son tres cadenas, el tipo de tren, destino y mensaje móvil, así que como `maxMensaje` es 5 tendremos que tener 15 cadenas en el array.

La cadena tipo de tren y destino han de tener como máximo los caracteres que quepan en un línea de la pantalla OLED, el mensaje móvil puede tener cualquier tamaño pero tened en cuenta el consumo de RAM así que no los hagáis muy largos.

Observad que el tipo del array es `char*` que indica que es un array de punteros a caracteres (cadenas).

En `setup()` inicializamos el puerto serie con `Serial.begin()` a 115200 baudios, para nuestro caso podemos elegir cualquier velocidad (9600, 19200, etc..) mientras corresponda con la del Monitor Serie, para otros casos dependerá del módulo con el que nos comuniquemos.

Luego enviamos un par de mensajes con `Serial.println()` para comprobar que la comunicación funciona e informar al usuario de que hay disponible un menú de ayuda.

Inicializamos el bus I2C con `Wire.begin()` y establecemos la velocidad del bus al máximo (400kHz) con `Wire.setClock()`, esto último realmente no es necesario. Observad la `L` al final del número que indica que la cifra se trate como un `long` en lugar de como un `int` que evidentemente estaría fuera de rango.

Ahora ya podemos refrescar los registros de tiempo del objeto `rtc` con `refresh()` y asignar a la variable `ultSegundo`, el segundo acabado de refrescar con `second()`. También pondremos la variable booleana `muestraHora` a `true` para que por defecto mostremos la hora actual.

El objeto `oled` lo inicializamos con `begin()` indicando el tipo de pantalla y su dirección I2C como requiere esta función. Observad el `&` en el dato tipo de pantalla, esto indica que no se le pasa el valor de ese dato sino que se le pasa un puntero a ese dato así la función podrá manipularlo.

Luego usaremos diferentes funciones de la librería para mostrar el mensaje de bienvenida 'Design by Paco' en la pantalla durante un par de segundos. Vanidoso que es uno 😊

`clear()` borra la pantalla, `setFont()` selecciona una de las fuentes de texto, `setCursor(columna, fila)` indica la posición de la pantalla donde se empezará a escribir. La columna se indica en pixels (entre 0 y 127) y la fila en 8 pixels, como tenemos disponibles 32 pixels, fila irá de 0 a 3.

En `loop()` comprobaremos el puerto serie por si recibimos una instrucción, leeremos el reloj y actualizaremos la hora o temperatura en la pantalla cada segundo si `muestraHora` es `true`, en caso contrario se está mostrando un mensaje y si ha pasado un tiempo desplazaremos el mensaje móvil de la pantalla.

Si `Serial.available()` devuelve un valor mayor que cero indicando que hay caracteres en el buffer los leeremos con nuestra función `entradaSerie()` que mirará qué comando introduce el usuario desde el Monitor Serie.

En `entradaSerie()` leemos el primer carácter del buffer con `Serial.read()` y con un `switch` compararemos el carácter con cada una de las ordenes disponibles para el usuario en los diferentes `case`.

Para el `case '#'` que se corresponde con el comando `#` en el que el usuario introduce la hora, minutos y segundos separados por `:` en el formato `#HH:MM:SS`, como ya hemos leído el `#` quedan por leer del buffer unos valores numéricos separados por un carácter no numérico.

La función `Serial.parseInt()` nos es muy útil ya que leerá hasta el carácter `:` descartándolo y convirtiendo los caracteres numéricos en su valor y devolviéndonos un `int` con el valor total. Sin esta práctica función tendríamos que hacer nosotros la conversión a base de `Serial.read()` y comparando el carácter recibido con uno numérico para convertirlo en su valor y llevar la cuenta para el valor final.

Lo repetiremos tres veces para leer el número correspondiente para las variables `hora`, `minutos` y `segundos`.

La función `constrain()` devuelve un valor entre el segundo y tercer parámetro del valor indicado como primer parámetro.

No vamos a hacer un chequeo completo de lo que ha introducido el usuario pero como mínimo aseguraremos que los valores estén dentro del rango de horas, minutos y segundos. Si el usuario introduce `#45:85,6` obtendremos `23:59:06` que al menos esta dentro del rango. Si solo introduce `#` o falta algún valor, debido al timeout (fin de tiempo de espera) la función `Serial.parseInt()` devolverá un valor que será un cero con lo que para `#` serán las `00:00:00` y para las `#12:15` serán las `12:15:00`.

Finalmente con la función `set()` pondremos en hora el RTC. Solo actualizamos los valores de hora, minutos y segundos, el resto: día de la semana, día, mes y año lo ponemos a un valor cualquiera.

Ahora refrescamos los registros de la hora actual y mostramos y respondemos al usuario con la cadena "Nueva ". Observad que en este `case` no hemos finalizado con `break` así que se seguirá ejecutando lo que viene a continuación, que es la respuesta al comando `@`

El `case` para `@` hace que se envíe al usuario la hora actual. Con `Serial.print()` enviamos un texto o un valor que obtenemos del objeto `rtc` (directamente del RTC), `horas` con `hour()`, `minutos` con `minute()` y `segundos` con `second()`, el último es un `Serial.println()` que nos enviará al final el carácter nueva línea `\n` con lo que el siguiente texto que se envíe aparecerá en la siguiente línea del Monitor Serie.

Aquí si finalizamos con `break`, por lo que si hemos entrado por `#` el usuario verá "Nueva Hora: 23:59:06" y si hemos entrado por `@` solo verá "Hora: 23:59:06"

El `case T` lee la temperatura del RTC con `temp()` y se la muestra al usuario.

El `case -` pone `muestraHora` a `true` para que se muestre la hora en la pantalla OLED

El `case +` de forma parecida a `#` lee un parámetro numérico del buffer, que limita con `constrain()` a un valor entre 1 y `maxMensaje` (el numero de mensajes definidos en el array `text[]`) y lo guarda en la variable `mensaje`. Se pone `muestraHora` a valor `false` para indicar que se está mostrando un mensaje y no la hora y a continuación lee las cadenas del mensaje del array y las muestra en la pantalla OLED.

Primero borra la pantalla, con `clear()` y selecciona una fuente pequeña para la cadena del tipo de tren con `setFont()`. Con `println()` imprime la cadena en la primera línea de pantalla. El índice de la cadena a mostrar será el numero que

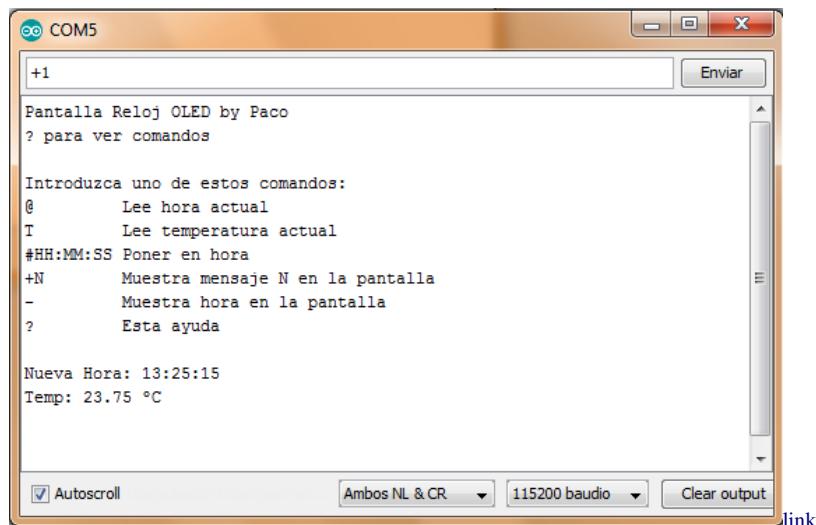
ha introducido el usuario que está en la variable `mensaje`, multiplicado por 3 ya que cada mensaje son 3 cadenas pero restando 3, ya que el índice del array empieza en 0. Si el usuario introduce +1 el cálculo dará 0 que es la primera cadena del array, si introduce +2 será 3 que es la cuarta cadena del array, etc..

Ahora seleccionamos una fuente de texto más grande para el destino y hacemos el cálculo para imprimir la segunda cadena del mensaje y la imprimimos.

De mostrar el mensaje móvil se encargará la librería `SSD1306Ascii`, con `tickerInit()` inicializamos su presentación. El primer parámetro es un puntero a una variable del tipo `TickerState` para su control, el segundo es la fuente que usará y el tercero es la línea de la pantalla en que lo mostrará.

El `case ?` y para cualquier otro comando del que no tengamos definida respuesta (`default`) mostrará el menú de ayuda, utilizaremos la macro `F()` con `Serial.println()` para imprimir las diferentes líneas y ahorrar memoria RAM

Los `case \n` y `\r` evitarán que se nos muestre el menú tras introducir un comando si tenemos seleccionado en el Monitor Serie algún tipo de ajuste de línea.



Siguiendo en el `loop()` refrescamos la hora del reloj y leeremos los segundos, si tenemos que mostrar la hora en la pantalla(`muestraHora` es `true`), comprobamos si se ha cambiado de segundo comparando `segundos` con `ultSegundo`, en ese caso leemos los minutos y horas y actualizamos `ultSegundo`.

Normalmente mostraremos la hora con `oledHora()`, pero también mostraremos la temperatura durante unos segundos, he escogido hacerlo cada minuto entre los segundos 25 y 35, así que si `segundos` está entre esos dos valores llamaremos a la función `oledTemp()`

Nuestra función `oledTemp()` seleccionara un tipo de letra grande, que haremos el doble de grande con `set2X()` e imprimiremos la temperatura leída con `temp()`, luego pondremos con `set1X()` la letra normal a tamaño normal, y una fuente pequeña e imprimiremos la hora con nuestra función `oledHora()`.

`oledHora()` imprimirá la hora en formato HH:MM:SS por lo que si algún dato solo tiene un dígito, está entre 0 y 9, añadirá un '0' antes de imprimirlo. Para el primer dato (`hora`) usamos un `if()` y para los siguientes el operador ternario, imprimiendo ':' o ':0' según el caso.

Si no mostramos la hora (`muestraHora` es `false`), es que se está mostrando uno de los mensajes, así que tenemos que ir moviendo el mensaje móvil. Si ya ha pasado un tiempo desde la última vez que se movió, `tickTime` es menor que el tiempo devuelto por `millis()`, se actualiza `tickTime` para dentro de 30ms. Variando este valor haremos que el mensaje vaya más rápido o más lento.

Llamamos a `tickerTick()` con el puntero a `state` para que nos mueva el texto. El valor devuelto con esta función si es 0 o negativo ($<= 0$) indica que se ha terminado de mostrar completamente la línea móvil, así que la volvemos a cargar con `tickerText()` para que muestre el mensaje del array `text[]` correspondiente.

Si cambiamos $<= 0$ por $<= 1$ no se esperará a acabar completamente el mensaje sino que lo mostrará a continuación.

12. Bits y PWM: Tren lanzadera analógico

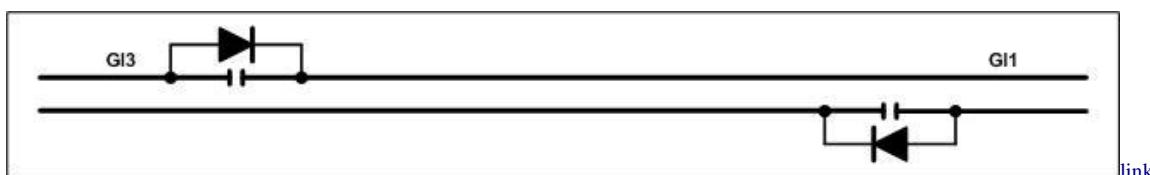
Ahora tomamos un sencillo montaje del maquetismo analógico, un tren lanzadera, y vamos a añadirle nuevas funcionalidades aprovechando las opciones del lenguaje Arduino y su hardware.



En un tren lanzadera en analógico se usa un relé para invertir la polaridad de la corriente de tracción una vez se llega a una de las estaciones de los extremos.

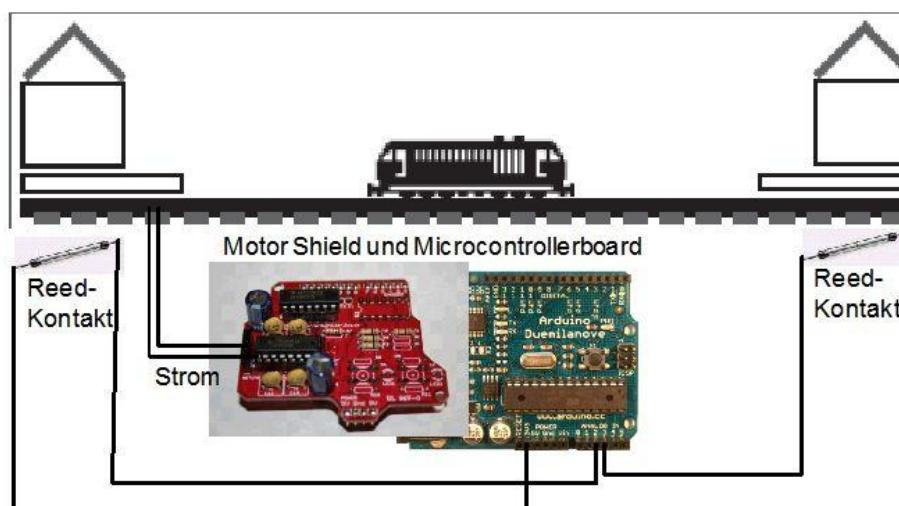
En su forma más simple, se monta un imán en la locomotora y un sensor Reed en cada una de las estaciones, este Reed cuando la locomotora se sitúa sobre él cierra el circuito de una de las bobinas de un relé biestable que hace que bascule e invierta la polaridad de la vía con lo que el tren cambia bruscamente de dirección y se dirige hacia la otra estación donde al llegar a su Reed activa la otra bobina del relé basculando nuevamente para cambiar la polaridad de la vía.

Otra versión que evita los cambios bruscos de dirección, sustituye los Reed por un temporizador y divide la vía en tramos. Los tramos extremos que corresponden a las estaciones se alimentan a través de un diodo que solo conducen cuando la polaridad que proporciona el relé hace que el tren vaya hacia la otra estación, si no es así, el tren se detiene bruscamente en ese tramo y espera a que el temporizador haga bascular el relé.

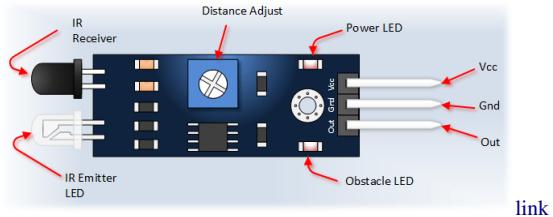


Usar el Arduino como un temporizador usando `delay()` o leer unos Reed para hacer bascular un relé es una tarea sencilla pero ya hemos visto que fácil se resuelve con la electrónica normal.

Una de las primeras mejoras que se nos ocurre es controlar la velocidad para así evitar las paradas bruscas. Ya que tenemos unas placas de control de motores (como la *shield* usada en la plataforma deslizante) podemos prescindir del relé y hacer un frenado suave una vez detectemos que hemos llegado a una estación, por lo que tendríamos algo parecido a esto:

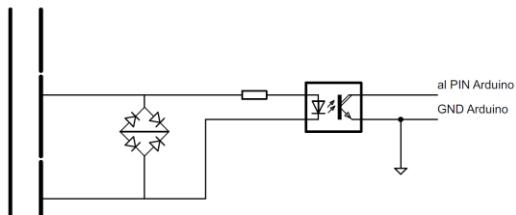


Si no queremos montar imanes, los Reed los podemos sustituir por detectores de infrarrojos:



[link](#)

Con estos sensores tendríamos una detección puntual, quizás sería mejor usar detectores de consumo como hacemos en DCC:



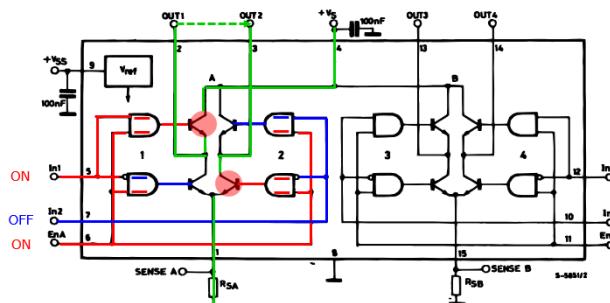
[link](#)

Un problema de usar detectores de consumo en analógico es que cuando el tren este parado, no tendremos tensión y no veremos el consumo. Esto lo podemos solucionar alimentando brevemente la vía (del orden de microsegundos), el pulso es tan corto que no moverá el motor pero veremos que hay consumo. También tendríamos que usar otro tipo de opto como el PC814 con diodos LED en anti paralelo, de otra forma no detectaríamos en una de las dos polaridades de la vía, aunque lo podríamos solventar de la misma forma que la detección en parado con un pulso con la polaridad correcta aunque no sé cómo afectaría a la larga al motor.

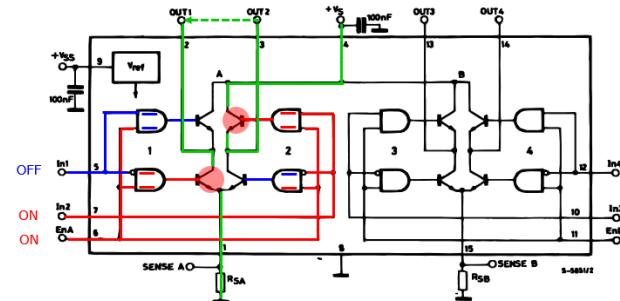
Si usamos la Motor Shield rev3 que vimos en un capítulo anterior ([9. Uso de 'shields': Plataforma deslizante con motor paso a paso](#)) ya incluye un manera de ver el consumo, de hecho hay dos, una por salida. Podemos medir el consumo (a razón de 1.65V/A) de cada salida con `analogRead()` lo que nos sirve para el tramo de las estaciones.

Para entender como detecta el consumo hay que ver cómo funciona el chip L298 cuando alimenta la vía:

● Transistor that turns on
— Path through which motor driving current flows
— Path with HIGH state
— Path with LOW state



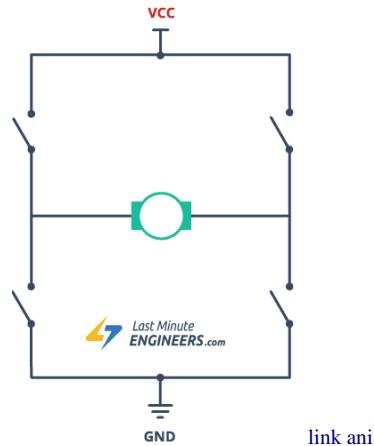
● Transistor that turns on
— Path with HIGH state
— Path with LOW state



Según el esquema de la Motor Shield rev3, cuando la salida PWMA (EnA) esta a **HIGH** se activa la salida A, la polaridad depende de DIRA (si BRAKEA está a **LOW**) que activa los transistores del L298 (entrada In1 o In2). Vemos que la corriente (en verde) circula atravesando el motor (OUT1-OUT2) y finalmente la resistencia donde medimos el consumo por el pin SNSA (A0), así que da igual la polaridad siempre que PWMA (EnA) esté a **HIGH** podremos medir la corriente en SNSA.

Si PWMA está a **LOW** ocurrirá como en la salida B (lado derecho de la imagen) no se activaran los transistores y no tendremos paso de corriente por la resistencia por lo que no tendremos medida de corriente.

En la siguiente imagen vemos de forma simplificada como se puede controlar el sentido de giro del motor (los transistores serian los interruptores) y medir corriente por el polo del motor que se conecta a masa.

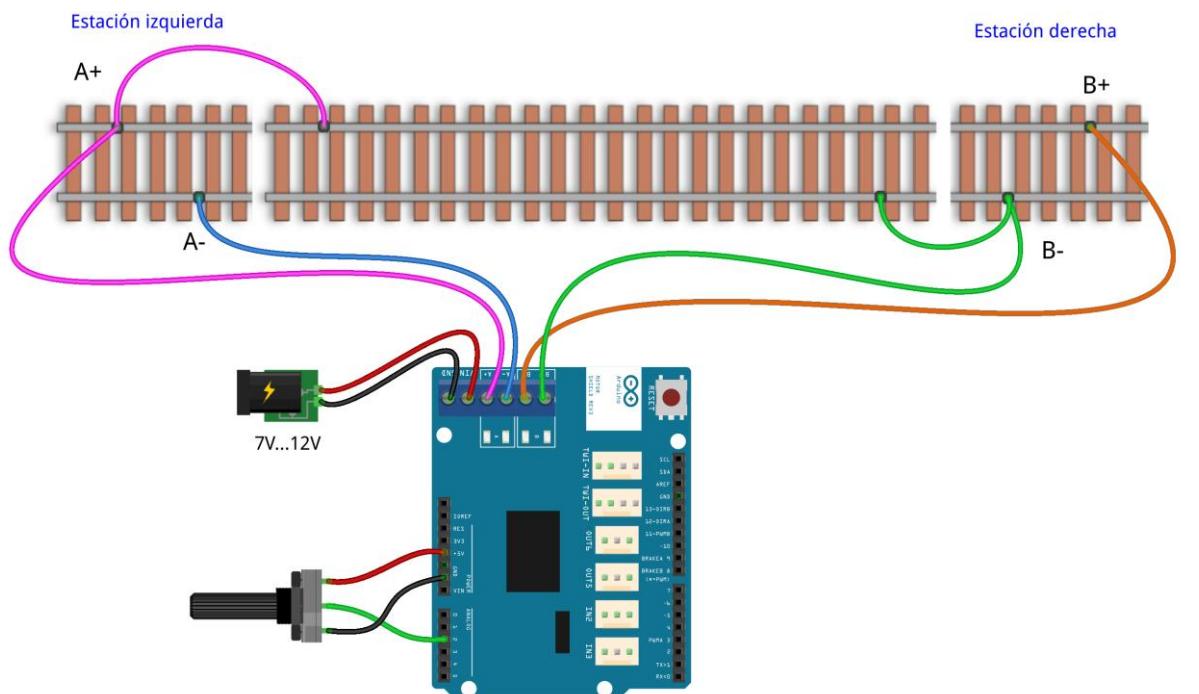


[Last Minute
ENGINEERS.com](#)

[link ani](#)

Si encontramos la manera de alimentar el tramo intermedio discerniendo cuando se está en él o cada una de las estaciones no necesitaríamos detectores de consumo externos.

Vamos a alimentar la vía así:



[fritzing](#)[link](#)

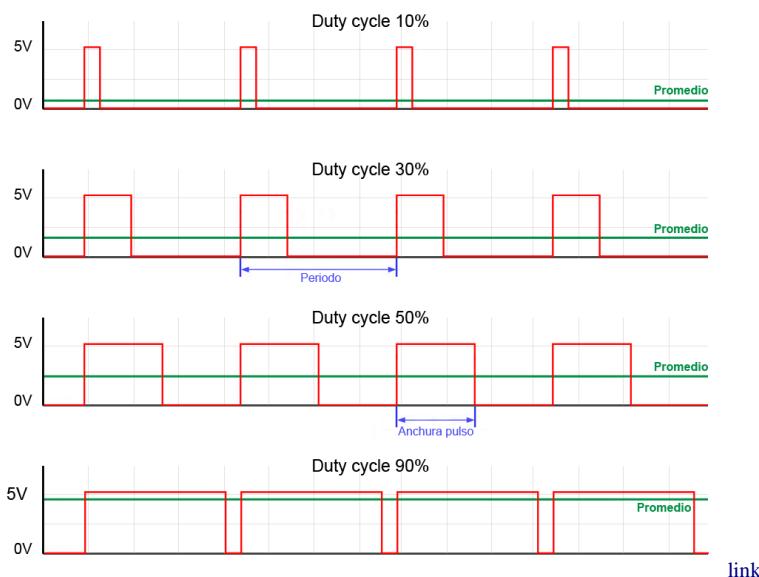
Para que esto funcione correctamente y sin cortocircuitos ambas salidas A y B han de estar en fase, (tienen que tener la misma polaridad a la vez), así no hay cortocircuito al pasar de un tramo a otro. Recordad que la resistencia de senseo está en el lado del transistor que conecta a masa el pin de salida.

Así, si A+ y B+ son positivas y A- y B- son masa el tren irá de derecha a izquierda, si está sobre la estación derecha o en el tramo central habrá consumo de corriente en la salida B y no en la A. Si esta sobre la estación izquierda habrá consumo de corriente en la salida A pero no en la B.

De forma parecida si A+ y B+ son masa y A- y B- son positivas el tren irá de izquierda a derecha, si está sobre la estación izquierda o en el tramo central habrá consumo de corriente en la salida A y no en la B. Si esta sobre la estación derecha habrá consumo de corriente en la salida B pero no en la A.

Simplificando: Cuando el tren se mueve hacia la izquierda (A+ y B+ son positivas) si hay consumo en A debemos frenar. Cuando el tren se mueve hacia la derecha (A+ y B+ son masa) si hay consumo en B hay que frenar.

Las salidas del L298 ponen en la vía la tensión de alimentación con lo que el tren iría a la máxima velocidad. Para acelerar y frenar, y para controlar la velocidad de marcha ya que no tenemos una salida analógica vamos a usar la modulación por ancho de impulso (PWM), que es una señal cuadrada de frecuencia constante, en la que podemos variar la duración del pulso (*duty cycle*) obteniendo una tensión media proporcional al ancho del pulso.



[link](#)

En Arduino Uno, Mini y Nano, disponemos de 6 salidas PWM de 8bits (valor entre 0 y 255) en los pines 3, 5, 6, 9, 10 y 11 a través de la función `analogWrite()`

<https://www.arduino.cc/reference/en/language/functions/analog-io/analogwrite/>

Las funciones PWM por hardware emplean los *Timer* (temporizador) para generar la onda de salida. Cada salida conectada a un mismo temporizador comparte la misma frecuencia, aunque pueden tener distintos *duty cycles*.

El Timer0 controla las salidas PWM 5 y 6.

El Timer1 controla las salidas PWM 9 y 10.

El Timer2 controla las salidas PWM 3 y 11.

Ya que necesitamos que ambas salidas del L298 estén a la misma tensión al mismo tiempo tenemos que escoger el par de salidas que usen un mismo Timer y usar el mismo *duty cycle*, afortunadamente en la *shield* de motores v3 los pines PWM usados son el D3 y el D11 que son controlados por el Timer 2.

La frecuencia estándar para las salidas PWM en Arduino Uno, Mini y Nano es de 490Hz para todos los pines, excepto para el 5 y 6 cuya frecuencia es de 980Hz.

La frecuencia de los PWM se puede modificar cambiando el preescalado de los *Timer* correspondientes. El preescalador es un valor de 3 bits almacenado en los tres bits menos significativos del correspondiente registro de Timer/Counter: TCCR0B, TCCR1B y TCCR2B.

Para el Timer2, la frecuencia es de 31250Hz, y los preescalados de 1, 8, 32, 64, 128, 256, y 1024.

<https://arduinoinfo.mywikis.net/wiki/Arduino-PWM-Frequency>

En nuestro caso, los motores de locomotoras funcionan mejor con una frecuencia baja de alrededor de 100Hz o menor por lo que cambiaremos el preescalado del Timer 2.

Podemos utilizar un patrón (máscara) con los bits a borrar, poner a 0 los tres menos significativos, y usar el operador binario & (Y binario, diferente del Y lógico **&&**) y luego poner a 1 los bits necesarios con el operador | (O binario, diferente del O lógico **||**). Observad la **B** delante del número para indicar que es un numero binario.

```
TCCR2B = (TCCR2B & B11111000) | B00000110; // set timer 2 divisor to 256  
// for PWM frequency of 122.55 Hz for D3 & D11
```

Por otra parte, para trabajar con bits hay una serie de funciones en lenguaje Arduino, entre ellas **bitSet()** para poner un bit a 1 y **bitClear()** para poner un bit a 0. También obtendremos unos 122Hz modificando el prescalado del Timer 2 a 256 cambiando el valor de sus bits de la siguiente forma:

```
bitSet(TCCR2B, 2); // Timer 2 divisor: 256, frecuencia 122.55 Hz (8ms)  
bitSet(TCCR2B, 1);  
bitClear(TCCR2B, 0);
```

La velocidad máxima la podremos regular con un potenciómetro conectado en **A2**, leeremos su valor con **analogRead()**, entre 0 y 1023, y lo adaptaremos para usarlo con el PWM mediante **analogWrite()**, entre 0 y 255.

Podríamos usar la función **map()** que realiza una serie de operaciones matemáticas para obtener el valor final. Esta es su definición:

```
long map(long x, long in_min, long in_max, long out_min, long out_max) {  
    return (x - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;  
}
```

En nuestro caso una manera más rápida de cálculo es usando bits con el operador **>>** que desplaza los bits hacia la derecha, cada desplazamiento sería equivalente a dividir por 2.

```
valor = analogRead (A2) >> 2;
```

Esto nos dividiría el valor leído en la entrada analógica por 4, al hacer dos desplazamientos.

De forma análoga el operador **<<** desplaza los bits hacia la izquierda, cada desplazamiento equivaldría a multiplicar por 2.

Cuando el tren pare en una estación estaremos un tiempo antes de reiniciar la marcha en sentido contrario para que los pasajeros puedan subir y bajar del tren.

Podemos usar un valor numérico para distinguir cada una de las fases (acelerar, correr, frenar, esperar) pero una manera que facilita la lectura del código es usar una enumeración.

El tipo de datos **enum** enumera automáticamente cualquier lista de identificadores que se le pase entre **{ }** separados por , asignándoles valores de 0, 1, 2, etc.

Como es un tipo de datos, la variable que se defina solo contendrá alguno de estos identificadores válidos. Se puede asignar un valor entero a cualquier identificador de la enumeración con **=**, al siguiente identificador se le asignará el siguiente numero entero.

El programa seria este.

```
// Tren lanzadera analogico - Paco Cañada 2020  
  
// Placa Arduino Motor Shield rev3  
// salidas digitales:  
const int dirA = 12; // dir: Polaridad de la salida  
const int dirB = 13; // pwm: activa/desactiva tension salida  
const int pwmA = 3; // brake: Tiene que estar a LOW si no se ha cortado su  
const int pwmB = 11; // jumper en la shield  
const int brakeA = 9; // entradas analogicas  
const int brakeB = 8; // Lectura de la corriente del motor. (1.65V/A)  
// entradas analogicas  
const int snsA = A0; // Potenciómetro  
const int snsB = A1; // sentido de marcha  
const int poti = A2;  
  
#define deAaB LOW  
#define deBaA HIGH
```

```

#define tiempoEspera 10000          // Tiempo de espera en estacion: 10s
#define numPulsos 12                // numero de pulsos antes de cambiar de velocidad
#define minCorriente 6              // Corriente minima para deteccion

int velActual;                   // velocidad actual del tren
int velObjetivo;                // velocidad objetivo del tren (potenciómetro)
int dirActual;                  // dirección actual del tren
int SensorA, SensorB;          // lectura de la corriente
unsigned long AccDec;           // temporizador para acelerar/frenar

enum Estado {ACELERA, CORRE, FRENA, ESPERA};

Estado Fase;

void setup()
{
    pinMode(pwmA, OUTPUT);        // Inicializa pines de salida del Arduino motor shield v3
    pinMode(pwmB, OUTPUT);
    pinMode(brakeA, OUTPUT);
    pinMode(brakeB, OUTPUT);
    pinMode(dirA, OUTPUT);
    pinMode(dirB, OUTPUT);

    bitSet(TCCR2B,2);            // Timer 2 divisor: 256, frecuencia PWM en D3 y D11:
122.55 Hz (8ms)
    bitSet(TCCR2B,1);
    bitClear(TCCR2B,0);

    setVelocidad (0);            // Detiene el tren
    digitalWrite(brakeA, LOW);   // Liberamos el freno
    digitalWrite(brakeB, LOW);
    buscaTren ();               // localiza el tren y pone la dirección correcta
    AccDec = millis();           // inicializamos tiempo temporizador
    leeVelObjetivo ();           // velocidad objetivo según potenciómetro
    Fase = ACELERA;
}

void loop() {
    switch (Fase) {
        case ACELERA:
            if ((millis() - AccDec) > (numPulsos * 8)) { // si tiempo mayor de numPulsos de 8ms (122hz)
                AccDec = millis();
                velActual = (velActual > 251) ? 255 : velActual + 4; // Aumentamos velocidad
                setVelocidad (velActual);                            // Leemos velocidad objetivo según potenciómetro
                leeVelObjetivo();                                     // Comprobamos si ha llegado a la velocidad objetivo
            }
            if (velActual >= velObjetivo)                      // Si ha llegado a la estación frena el tren
                Fase = CORRE;
            if (enEstacion())
                Fase = FRENA;
            break;
        case CORRE:
            leeVelObjetivo ();                                // Actualizamos velocidad según potenciómetro
            if (velActual != velObjetivo)
                setVelocidad (velObjetivo);
            if (enEstacion())
                Fase = FRENA;
            break;
        case FRENA:
            if ((millis() - AccDec) > (numPulsos * 8)) { // si tiempo mayor de numPulsos de 8ms (122hz)
                AccDec = millis();
                velActual = (velActual > 4) ? velActual - 4 : 0; // Disminuimos velocidad
                setVelocidad (velActual);
            }
            if (velActual == 0)
                Fase = ESPERA;                             // Si ha frenado, tenemos que esperar en la estación
            break;
        case ESPERA:
            delay (tiempoEspera);                         // Esperamos a que suban y bajen los pasajeros
            cambiaDireccion();                           // Vamos en dirección hacia la otra estación
            Fase = ACELERA;
            break;
    }
}

void setVelocidad (int velocidad) {
    analogWrite (pwmA,velocidad);                    // pins como PWM
    analogWrite (pwmB,velocidad);
    velActual = velocidad;
}

```

```

}

void setDireccion (int sentido) {
    digitalWrite (dirA, sentido);
    digitalWrite (dirB, sentido);
    dirActual = sentido;
}

void cambiaDireccion () {
    dirActual = (dirActual == deAaB) ? deBaA : deAaB ;
    setDireccion (dirActual);
}

void leeVelObjetivo() {
    int valor = analogRead (poti);           // lee potenciómetro: valor de 0 a 1023
    valor = valor >> 4;                   // se divide entre 16. Son 64 escalones de velocidad PWM.
    velObjetivo = valor << 2;             // se multiplica por 4, para que este entre 0 y 255.
}

void leeAnalog () {
    SensorA = analogRead (snsA);           // Lee corriente del L298
    SensorB = analogRead (snsB);
}

void leeSensores () {
    if (velActual > 0)                  // Solo si esta en marcha
        leeAnalog ();                  // lee corriente
}

void buscaTren () {
    setDireccion (deAaB);                // dirección de A a B por defecto
    digitalWrite(pwmA,HIGH);            // activa tensión salida. pins como salida digital
    digitalWrite(pwmB,HIGH);
    delay (1);
    leeAnalog ();
    setVelocidad (0);                  // esperamos un poco a que estabilize
    if (SensorB > minCorriente)       // Leemos la corriente
        setDireccion (deBaA);          // pins como PWM y velocidad 0
    // Si esta en la estación B, cambia la dirección
}

bool enEstacion () {
    leeSensores();                     // Lee corriente del sensor según dirección
    if ((dirActual == deAaB) && (SensorB > minCorriente))
        return (true);
    if ((dirActual == deBaA) && (SensorA > minCorriente))
        return (true);
    return (false);
}

```

Primero definimos todos los pines de la placa shield, de esta manera tendremos información de cómo está conectada y los pines ocupados además del pin para el potenciómetro.

También definimos unos valores para el estado del pin de dirección según vayamos de derecha a izquierda (**deBaA**) o viceversa (**deAaB**), el tiempo de espera en la estación(**tiempoEspera**), cuánto pulsos se han de producir antes de cambiar de velocidad para que sea progresiva (**numPulsos**), cuantos más pulsos mas tardará en acelerar y **FRENAR!**

Definimos el valor mínimo de consumo de corriente (**minCorriente**) para que se considere que hay ocupación, esto lo hacemos porque la placa tiene en sus salidas unos LED que indican su activación y que consumen algo de corriente que nos podría llevar a error.

Como variables definimos la velocidad actual del tren (**velActual**) y la velocidad objetivo (**velObjetivo**) según la lectura de la posición del potenciómetro, la dirección actual del tren (**dirActual**), el valor leído de corriente en los sensores (**SensorA** y **SensorB**) y por último un temporizador para acelerar/frenar (**AccDec**).

Vamos a usar un **enum** llamado **Estado** para los valores de las distintas fases o estados: **ACELERA**, **CORRE**, **FRENA** y **ESPERA**, que compararemos en un **switch()**, así que definimos la variable **Fase** del tipo de este **enum**

En **setup()** inicializamos los diferentes pines como salidas (**OUTPUT**) y cambiamos la frecuencia del PWM del Timer2 modificando los bits de su registro **TCCR2B** para 122,55Hz lo que nos da un periodo de unos 8ms

Primero, por seguridad, detenemos el tren poniendo el parámetro de nuestra función **setVelocidad()** a 0.

En la función `setVelocidad()` mediante `analogWrite()` se ponen ambos pines `pwmA` y `pwmB` a la vez como PWM con el duty cycle pasado como parámetro (de 0 a 255), además guardamos este valor en la variable `velActual`.

Siguiendo en `setup()`, se libera el freno de los motores poniendo las salidas `brakeA` y `brakeB` a `LOW` lo que permite que haya tensión según el PWM en las salidas del L298.

Tenemos que localizar el tren para poner el sentido de movimiento inicial correcto por lo que nuestra función `buscaTren()` llama a nuestra función `setDireccion()` con la dirección `deAaB` como parámetro inicialmente.

`setDireccion()` escribe el parámetro a la vez en los dos pines `dirA` y `dirB` con `digitalWrite()` para que ambas salidas estén en fase, además lo guarda en la variable `dirActual`.

Una vez seleccionada una dirección activamos las salidas `pwmA` y `pwmB` durante 1ms con lo que podremos leer un consumo en los sensores con `leeAnalog()`.

Al usar `digitalWrite()` la función PWM de los pines se desconecta (usando `digitalRead()` también ocurriría) por lo que tenemos que recordar volver a ponerlos como PWM mas adelante.

También podríamos haber usado `setVelocidad(255)` ya que con el valor 255 la función `analogWrite()` pone la salida a `HIGH` y con 0 la pone a `LOW`, pero así se ve más claro lo que queremos hacer.

Nuestra función `leeAnalog()` lee ambos sensores con `analogRead()` y guarda su valor en las variables `SensorA` y `SensorB` respectivamente.

Una vez leída la corriente, volvemos a poner los pines en función PWM llamando a `setVelocidad()` con velocidad 0.

Si en el `SensorB` se lee más corriente que la mínima `minCorriente` indicando ocupación, se cambia la dirección de movimiento llamando a `setDireccion()` con el parámetro `deBaA`.

En `setup()` seguimos inicializando el temporizador `AccDec` con `millis()` y leemos la velocidad objetivo mediante nuestra función `leeVelObjetivo()`.

En `leeVelObjetivo()` leemos la entrada analógica `poti` del potenciómetro. Como en cada paso de aceleración vamos a aumentar el duty cycle en 4 para que no sea excesivamente lenta vamos a convertir este valor en uno adecuado a `analogWrite()` pero de 4 en 4.

Si usamos `map()` o dividimos valor entre 4 estará entre 0 y 255 pero de 1 en 1. Así que con el operador `>>` dividimos entre 16 (valor pasara de entre 0 y 1023 a otro valor entre 0 y 63) luego con el operador `<<` se multiplicara por 4 con lo que ya estará entre 0 y 255, además irá de 4 en 4.

Ya solo queda poner `Fase` a `ACELERA` antes de entrar en el `loop()`

El `loop()` es bastante simple, con un `switch()` compararemos `Fase` con los distintos estados y actuaremos en consecuencia.

En el `case ACELERA` comprobamos que haya pasado un tiempo suficiente para cambiar de velocidad (el tiempo del número de pulsos de 8ms definidos en `numPulsos`)

Si ha pasado el tiempo, se actualiza `AccDec` y se incrementa `velActual` en 4 si no es mayor de 252 usando el operador ternario y se cambia el duty cycle con `setVelocidad()`, además leemos el potenciómetro con `leeVelObjetivo()`.

Ahora, la parte de control, comprobamos si se ha alcanzado la velocidad objetivo para cambiar de fase a `CORRE`, y por último la parte de seguridad, si se ha llegado a la estación se pasará a la fase `FRENA`.

La función `enEstacion()` devolverá `true` o `false` en función de si hay consumo en el sensor de la estación, así que llama a `leeSensores()` para actualizar `SensorA` y `SensorB`.

`leeSensores()` llama a `leeAnalog()` solo si `velActual` es mayor que 0, o sea se está moviendo el tren.

Luego en los `if()` según la dirección actual y (`&&`) si hay consumo de corriente en el sensor correspondiente se devolvería `true`. Si no es el caso, se devolvería `false`.

En el `case CORRE` leemos el potenciómetro y ponemos la velocidad actual a la velocidad objetivo. Luego si se llega a la estación se cambia la fase a `FRENA`

En el `case FRENA` de forma parecida al `case ACELERA` se disminuye la velocidad de 4 en 4 a medida que vaya pasando el tiempo. Si la velocidad actual es 0 pasaremos a la fase `ESPERA`.

En el **case ESPERA** simplemente esperamos un tiempo con **delay()** para que los pasajeros puedan subir y bajar del tren y cambiamos la dirección de movimiento con **cambiaDireccion()** antes de volver a la fase **ACELERA**.

En **cambiaDireccion()** según el valor de **dirActual** mediante el operador ternario se cambia a la contraria y se actualizan los pines de dirección del L298 con **setDireccion()**

Video: <https://youtu.be/ybu8wsuDhA8>



Como ejercicio, ya que hay pines libres, se pueden poner unos semáforos a la entrada y/o salida de cada una de las estaciones que cambien su aspecto en concordancia con el movimiento del tren. También como mejora se podría leer otro potenciómetro para poder variar el tiempo de espera en las estaciones.

13. De la idea al programa: Paso a nivel DCC

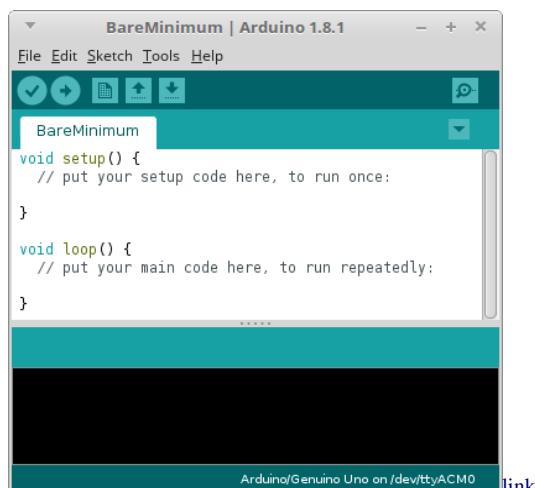
Ahora que ya tenemos una noción de las diferentes instrucciones, librerías y estructuras del Lenguaje Arduino nos falta lo más difícil: Plasmar una idea en un programa ... y que funcione!

La idea:

"Quiero usar Arduino como un decodificador DCC para que cuando desde la central doy una orden sencilla (bajar/subir las barreras de un paso a nivel) se encargue de ejecutar una secuencia más o menos compleja, por ejemplo:

Activar la campana y las luces, bajar la barrera, desactivar la campana y activar la señal luminosa que indica al maquinista que el paso de nivel es seguro (esto al menos en la DB). Luego, el proceso correspondiente para levantar la barrera."

Con estas especificaciones y tras habernos leído, incluso más de una vez, todos los capítulos anteriores nos plantamos delante de esto:



¿Qué hago?

¿Has encontrado la solución a la adivinanza? -- preguntó el Sombrerero, dirigiéndose de nuevo a Alicia.

No. Me doy por vencida. ¿Cuál es la solución?

No tengo la menor idea -dijo el Sombrerero.

Alicia en el País de las Maravillas, libro de Lewis Carroll, diácono anglicano, lógico, matemático, fotógrafo y escritor británico (1832-1898)

Las reacciones a la pregunta de qué hacer ahora son:

1- Uff! esto es muy difícil, Arduino no es para mí.

2- Empezamos a escribir: `pinMode()` por aquí, `digitalWrite()` por allá,....

3- Mensaje en el foro: ¿Quién me escribe el programa para un paso a nivel ...

4- Parar y pensar

La respuesta 1 es una reacción razonable cuando no se tiene experiencia, pero si algo no se empieza difícilmente se acabará. La respuesta 3 dependerá de lo receptivos que estén los foreros con más experiencia. Aunque la respuesta 2 parece una buena idea, la correcta es la 4.

La tarea no es complicada, al leerla otra vez ya nos vamos haciendo la idea de las instrucciones básicas que vamos a usar, sin embargo, falta información importante:

- Que usamos para mover las barreras? Motor, servo, relé,...?
- Cuantas luces/señales necesitamos? Son intermitentes?
- Para el sonido de la campana, generamos un tono o reproducimos un archivo MP3?
- La señal DCC como la decodificamos? Usamos una dirección de locomotora o de accesorio?
- Cuándo y cómo pasamos de un paso de la secuencia a otro?
- Las acciones (sonido, luz intermitente, barrera) son simultáneas o se hacen una detrás de otra?
- Solo queremos control por DCC o también sería útil tener pulsadores para subir y bajar las barreras?
- Cual es la posición inicial, barreras subidas o bajadas?

Algunas de las respuestas dependerán de lo que tengamos montado en la maqueta. Como es un ejemplo, vamos a intentar responderlas:

Usaremos un servo para mover las barreras, para la campana nos bastará con un tono y usaremos un altavoz de 8 ohm, para las luces usaremos LED, para la señal DCC usaremos una dirección de accesorios que será la 6, de momento sólo queremos usarlo por DCC ya que controlamos la maqueta por software desde nuestro PC aunque podríamos tener en cuenta lo de los pulsadores para un futuro y la posición inicial será barreras subidas con todo apagado.

Para la secuencia, sino la conocemos en detalle, buscaremos información o la supondremos según nuestra experiencia.



[link](#)

La secuencia será esta:

- **Orden bajar barreras:** Activar luz intermitente carretera y tono de aviso, luego bajar barrera y cuando este bajada, activar luz intermitente maquinista y apagar tono
- **Orden subir barreras:** Apagar luz maquinista, subir barrera y cuando este subida, apagar luz carretera.

¿Que librerías necesito?

No siempre se necesitan librerías, depende de lo que vayamos a controlar, de la complejidad del control y de la experiencia en programación.

En nuestro caso, ya que tenemos un servo usaremos alguna de las librerías que los controlan que hemos usado en ejemplos anteriores. Para no complicarnos usaremos la librería [Servo.h](#) incluida en el Arduino IDE que tiene bastante documentación:

<https://www.arduino.cc/en/Reference/Servo>

Para decodificar la señal DCC ya que es algo bastante complejo, usaremos la librería [NmraDcc](#) que vimos en un capítulo anterior ([6. Otras librerías: Descodificador de Accesorios DCC](#))

<https://github.com/mrrwa/NmraDcc>

Para el tono del altavoz no necesitamos librerías, usaremos las instrucción [tone\(\)](#) del Arduino que genera un tono en un pin de la frecuencia que le indiquemos.

<https://www.arduino.cc/reference/en/language/functions/advanced-io/tone/>

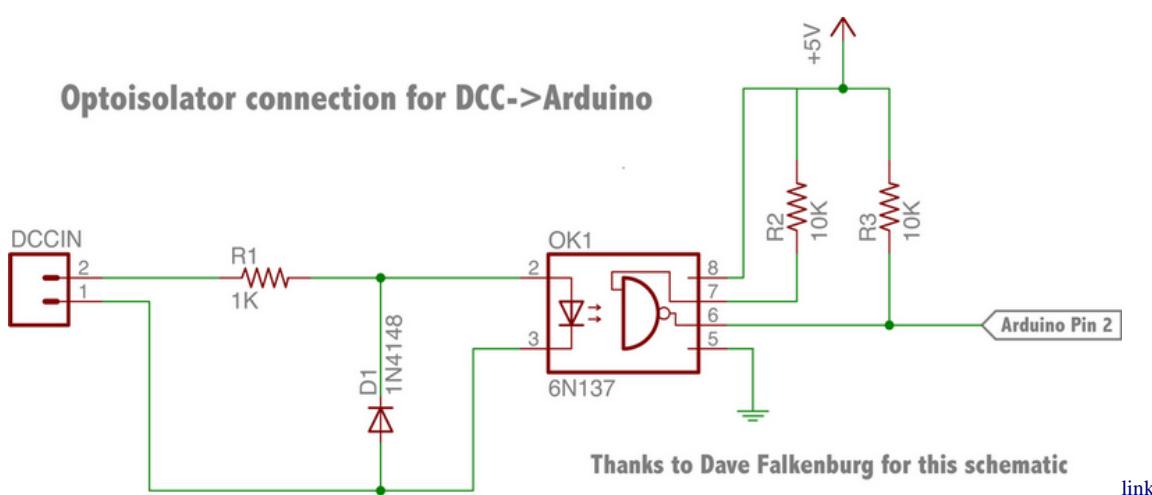
¿Como lo conecto?

Tenemos que mirar si necesitamos conectar algún elemento a un pin específico por lo que no se podrá conectar otro elemento diferente a ese pin, por ejemplo para un Arduino Uno:

- | | |
|---|---------------------------|
| - Si usamos comunicaciones serie (Serial): | D0 y D1 |
| - Si usamos comunicaciones I2C (Wire): | A4 y A5 |
| - Si usamos el SPI (lector SD, ...): | D11, D12 y D13 |
| - Si necesitamos PWM por hardware: | D3, D5, D6, D9, D10 o D11 |
| - Si requiere una entrada analógica (potenciómetro, LDR,...): | A0, A1, A2, A3, A4 o A5 |
| - Si usa interrupciones externas (entrada DCC,...): | D2 (habitual) o D3 |
| - Si la librería necesita una conexión específica: | Mirar librería |

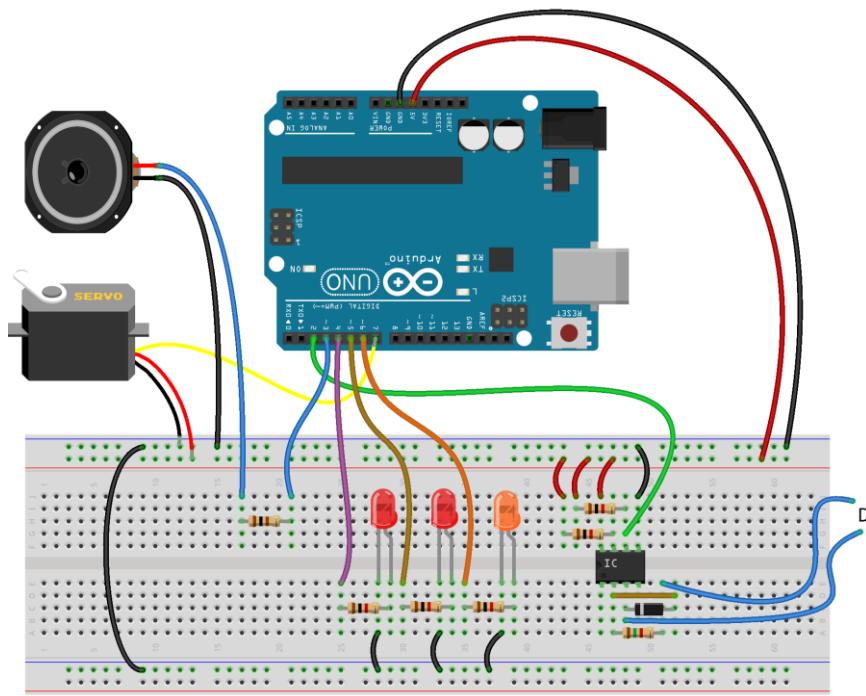
También puede ser necesario montar algún elemento o circuito electrónico por lo que tendremos que mirar en caso de que vaya montado en una *shield* que pines usa ya que seguramente no los podremos cambiar.

Para nuestro programa sólo tiene requerimientos la librería DCC que usa interrupciones externas, la señal DCC del circuito electrónico según el esquema la conectaremos a D2, el resto de elementos los podemos conectar a cualquier pin.



También hay que tener en cuenta que los LED necesitan su resistencia y que el altavoz de 8 ohm también necesitará una resistencia de 100 ohm en serie.

Este es el circuito que vamos a utilizar:



[link](#)

¿Como empiezo?

“Empieza por el principio - dijo el Rey con gravedad - y sigue hasta llegar al final; allí te paras”.

Alicia en el País de las Maravillas.

Una vez tenemos claros los objetivos y los medios de que disponemos, podemos empezar a desarrollar una estrategia para realizarlo. No es que haya una manera mejor que otra si al final se consigue el mismo resultado pero seguramente una será más eficiente o incluso más legible que otra por si en un futuro queremos modificar o añadir alguna cosa más.

Una manera habitual de resolver estos problemas es el diseño descendente (*top-down*), que es dividir el problema en varias tareas que a su vez se divide en otras más pequeñas dejando los detalles (instrucciones) para el final.

Otro método es el diseño *bottom-up* o Factorización muy usado en el lenguaje FORTH, en este caso se hacen pequeñas piezas de código que se pueden probar y verificar que funcionan sin errores y con ellas ir montando otras piezas de código mayores a base de reutilizar estas pequeñas piezas.

Para nuestro ejemplo, usando el método de diseño descendente, escribiríamos en una hoja de papel en blanco:

- 1- Poner barrera arriba y todo apagado
- 2- Esperar orden de bajada
- 3- Encender luces carretera, sonido y bajar barrera
- 4- Apagar sonido y encender luz maquinista
- 5- Esperar orden de subida
- 6- Apagar luz maquinista y subir barrera
- 7- Apagar luces carretera
- 8- Volver a empezar (puede ser desde 1 o 2)

Podemos ir subdividiendo las tareas describiéndolas con más detalle, por ejemplo la 3:

3- Encender luces carretera, sonido y bajar barrera

- a- Activar luces intermitentes carretera, período 1 segundo
- b- Generar tono (alternativo 277Hz y 244Hz cada 350ms)
- c- Mover servo hasta posición barrera bajada

Seguimos subdividiendo, en este caso el movimiento del servo. Como queremos un movimiento suave de bajada en lugar de mover el servo directamente a la posición final, lo haremos poco a poco. (cada 20ms. Aumentándolo iría más lento, por ejemplo: 60ms)

- c1- Conectar servo al pín
- c2- Mover un poco el servo hacia la posición de barrera bajada
- c3- Esperar 20ms a que se mueva
- c4- Ha llegado a la posición de barrera bajada? No: continuar en c2
- c5- Desconectar el servo

Para las luces de carretera cuando están activas:

- a1- Encender luz derecha y apagar izquierda
- a2- Esperar 0,5 segundos
- a3- Apagar luz derecha y encender izquierda
- a4- Esperar 0,5 segundos
- a5- Continuar en a1

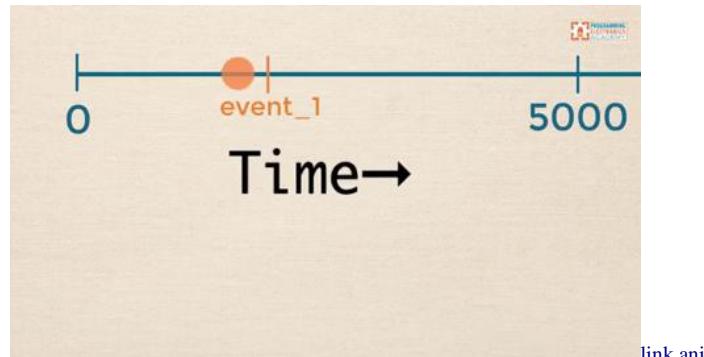
Para el tono cuando está activo:

- b1- Poner tono de 277Hz
- b2- Esperar 350ms
- b3- Poner tono de 244Hz
- b4- Esperar 350ms
- b5- Continuar en b1

Aquí ya nos damos cuenta que si todo esto se tiene que realizar a la vez, todas esas esperas nos van a interferir unas con otras y no conseguiremos los períodos establecidos. Así que una primera estrategia en las esperas es sustituir los típicos `delay()` por temporizadores que hagan uso de `millis()`

El temporizador normalmente se realiza restando del tiempo actual devuelto por `millis()` el tiempo inicial almacenado en la variable del temporizador y si es mayor o igual que la espera se actualiza el tiempo inicial con el tiempo actual y se ejecutan las acciones, algo así :

```
if (millis() - tiempoTemporizador >= espera) {  
    tiempoTemporizador = millis();  
    // acciones a ejecutar  
}
```



[link ani](#)

Ya vemos que vamos a necesitar varias variables para temporizadores, el tipo tendrá que ser `unsigned long` como el valor devuelto por `millis()` para que la resta de un resultado correcto.

Como las nombramos, **t1**, **t2**, **t3**? o mejor como **Temporizador1**, **Temporizador2**, **Temporizador3**?

Podemos nombrarlas como queramos pero un nombre claro nos permitirá comprender el código, un **t1** no parece la mejor forma de relacionarlo con el temporizador de las luces.

```
unsigned long tiempoLuzCarretera;
unsigned long tiempoTono;
unsigned long tiempoServo;
```

Este cambio nos obliga a repensar las tareas, pero sólo a las que le afecte el cambio, el resto de la planificación sigue igual. Así la intermitencia de luces de carretera pasaría a:

- 1- Sí no han pasado 0,5s no hacer nada.
- 2- Actualizar tiempo
- 3- Si la luz está encendida, apagarla
- 4- Si la luz está apagada, encenderla

Esto descrito así, que parece correcto, tiene un fallo: Nunca se apagaría la luz una vez encendida:

En el paso 3 si la luz está encendida, la apagamos. En el paso 4 vemos que la luz está apagada así que la volvemos a encender.

En este caso necesitamos algo que nos indique en qué fase están las luces para proceder. Usaremos una variable auxiliar para saber la fase, del tipo **bool** nos bastará, además con el operador **!** la invertiremos fácilmente.

```
bool faseLuzCarretera;
```

Ahora tendremos esto para la intermitencia:

- a1- Sí no han pasado 0,5s no hacer nada.
- a2- Actualizar tiempo
- a3- Si Fase es false: Encender luz derecha y apagar izquierda
- a4- Si Fase es true: Apagar luz derecha y encender izquierda
- a5- Cambiar Fase

Como ya no podemos subdividir más podemos intentar escribir el código de esta parte. Puede ser parecido a este:

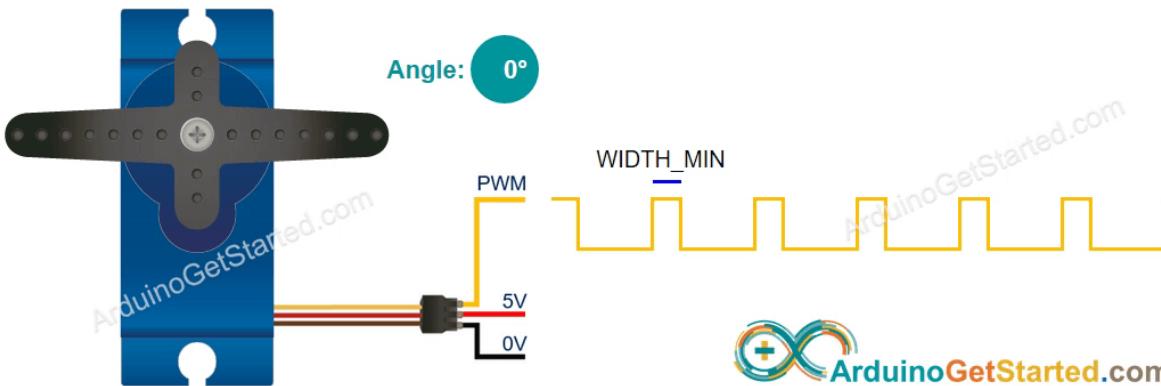
```
if (millis() - tiempoLuzCarretera >= 500) { // 1- Si no han pasado 0,5s (500ms) no hacer nada.
    tiempoLuzCarretera = millis(); // 2- Actualizar tiempo
    if (faseLuzCarretera == false) { // 3- Si Fase es false: Encender luz derecha y apagar izquierda
        digitalWrite (pinLuzDerecha, HIGH);
        digitalWrite (pinLuzIzquierda, LOW);
    }
    else { // 4- Si Fase es true: Apagar luz derecha y encender izquierda
        digitalWrite (pinLuzDerecha, LOW);
        digitalWrite (pinLuzIzquierda, HIGH);
    }
    faseLuzCarretera = ! faseLuzCarretera; // 5- Cambiar Fase
}
```

El tono es parecido:

- b1- Sí no han pasado 350ms no hacer nada.
- b2- Actualizar tiempo
- b3- Si Fase es false: generar tono de 277 Hz
- b4- Si Fase es true: generar tono de 244Hz
- b5- Cambiar Fase

y el código quedaría una cosa así:

```
if (millis() - tiempoTono >= 350) { // 1- Si no han pasado 350ms no hacer nada.
    tiempoTono = millis(); // 2- Actualizar tiempo
    if (faseTono == false)
        tone (pinAltavoz, 277); // 3- Si Fase es false: generar tono de 277 Hz
    else
        tone (pinAltavoz, 244); // 4- Si Fase es true: generar tono de 244Hz
    faseTono = ! faseTono; // 5- Cambiar Fase
}
```



[link ani](#)

Para el movimiento del servo lo replanteamos así:

- c1- Conectar servo al pin
- c2- Mover un poco el servo
- c3- Ha llegado a la posición de barrera bajada? No: continuar en c2
- c4- Desconectar el servo

El paso c2 se subdividiría así:

- 1- Si no han pasado 20ms no hacer nada
- 2- Actualizar tiempo
- 3- Actualizar posición hacia barrera bajada
- 4- Mover el servo a nueva posición

El código de este paso sería algo así:

```
if (millis() - tiempoServo >= 20) {      // 1- Si no han pasado 20ms no hacer nada.
    tiempoServo = millis();                // 2- Actualizar tiempo
    posicionServo = posicionServo - 1;     // 3- Actualizar posición hacia barrera bajada
    servoBarrera.write (posicionServo);   // 4- Mover el servo a nueva posición
}
```

Es fácil imaginar que la luz para el maquinista que también será intermitente será parecida a estas, así su código será parecido a esto:

```
if (millis() - tiempoLuzMaquinista >= 500) { // 1- Si no han pasado 0,5s (500ms) no hacer nada.
    tiempoLuzMaquinista = millis();           // 2- Actualizar tiempo
    if (faseLuzMaquinista == false)
        digitalWrite (pinLuzMaquinista, HIGH); // 3- Si Fase es 0: Encender luz
    else
        digitalWrite (pinLuzMaquinista, LOW);  // 4- Si Fase es 1: Apagar luz
    faseLuzMaquinista = ! faseLuzMaquinista; // 5- Cambiar Fase
}
```

Todos estos trozos de código que ya tenemos hay que ir ejecutándolos para que realicen la tarea que habíamos escrito en un principio:

3- Encender luces carretera, sonido y bajar barrera

Los ponemos todos estos trozos juntos uno detrás de otro o creamos una función para cada uno o varios de ellos?

Volvamos al papel, a lo que escribimos para esta tarea:

- a- Activar luces intermitentes carretera, periodo 1 segundo
- b- Generar tono (alternativo 277Hz y 244Hz cada 350ms)
- c- Mover servo hasta posición barrera bajada

Tendríamos que cambiarlo un poco ya que hay que ejecutarlos periódicamente con un bucle para que avance **millis()**:

- a- Conectar servo al pin
- b- Secuencia de luces de carretera, periodo 1 segundo
- c- Secuencia de tono (alternativo 277Hz y 244Hz cada 350ms)
- d- Mover un poco la barrera hacia abajo
- e- Ha llegado a la posición de barrera bajada? No: continuar en b
- f- Desconectar el servo

Esto podría quedar como código así:

```
servoBarrera.attach (pinServo); // a- Conectar servo al pin
do {
    intermitenciaLuzCarretera(); // b- Secuencia de luces de carretera
    generarTono(); // c- Secuencia de tono
    moverBarreraAbajo(); // d- Mover un poco la barrera hacia abajo
} while (posicionServo > posicionBarreraBajada); // e- Repetir hasta posición de barrera bajada
servoBarrera.detach (); // f- Desconectar servo
```

Por claridad al leer el código parece que la mejor opción es que cada trozo de código sea una función, serán del tipo **void** ya que no han de devolver ningún resultado y no tendrán parámetros ya que no hacen falta.

También podríamos haber escrito:

- a- Conectar servo al pin
- b- Mientras no se llegue a la posición de barrera bajada
 - b1- Secuencia de luces de carretera, periodo 1 segundo
 - b2- Secuencia de tono (alternativo 277Hz y 244Hz cada 350ms)
 - b3- Mover un poco la barrera hacia abajo
- c- Desconectar el servo

Y el código sería:

```
servoBarrera.attach (pinServo); // a- Conectar servo al pin
while (posicionServo > posicionBarreraBajada) { // b- Mientras no se llegue a la posición de
barrera bajada
    intermitenciaLuzCarretera(); // b1- Secuencia de luces de carretera
    generarTono(); // b2- Secuencia de tono
    moverBarreraAbajo(); // b3- Mover un poco la barrera hacia abajo
}
servoBarrera.detach (pinServo); // c- Desconectar servo
```

Cuál es la mejor? Se puede llegar al mismo resultado por dos caminos distintos, las dos hacen lo mismo la única diferencia es que una comprueba la posición antes de empezar y otra al final.

while() no hará nada si ya estamos con la barrera bajada, mientras que **do-while()** hará las secuencias al menos una vez.

En ambos casos como moveremos más de una vez el servo no tiene mucha importancia hacerlo de una manera u otra.

¿Por dónde sigo?

“¿Podrías decirme, por favor, qué camino debo seguir para salir de aquí?
-Esto depende en gran parte del sitio al que quieras llegar -dijo el Gato.
-No me importa mucho el sitio... -dijo Alicia.
-Entonces tampoco importa mucho el camino que tomes -dijo el Gato.
- ... siempre que llegue a alguna parte -añadió Alicia como explicación.
- ¡Oh, siempre llegarás a alguna parte -aseguró el Gato- si caminas lo suficiente!”

Alicia en el País de las Maravillas

Vamos por otra tarea, la 1 parece fácil y además es la candidata perfecta para ir en el **setup()**, ya que nos pone las condiciones iniciales:

1- Poner barrera arriba y todo apagado

Casi que podríamos escribir su código directamente, pero vamos a desarrollarla:

- a- Inicializar los pines como salidas
- b- Apagar todas las salidas
- c- Mover el servo a la posición de barrera subida
- d- Inicializar librería DCC (nos habíamos olvidado!!!!) 😊😊

Para colocar el servo tenemos dos opciones, moverlo ahora o solo inicializar la posición y cuando llegue la orden de bajar barrera la función de mover el servo ya lo colocará en su sitio.

Elegimos la primera:

```
posicionServo = posicionBarreraSubida; // Posición del servo inicial
servoBarrera.attach (pinServo); // Conectar servo
servoBarrera.write (posicionServo); // Mover servo
delay (1000); // Esperamos un poco a que llegue
servoBarrera.detach (pinServo); // Desconectar servo
```

Para inicializar la librería DCC, lo hacemos como indican sus ejemplos, indicamos el pin y la interrupción que usa y lo inicializamos como un decodificador de accesorios para que podamos capturar la orden de subir y bajar barrera:

```
Dcc.pin (digitalPinToInterrupt (pinDCC), pinDCC, 1);
Dcc.initAccessoryDecoder (MAN_ID DIY, 1, FLAGS_OUTPUT_ADDRESS_MODE, 0);
```

En Arduino Uno hay dos pines para interrupciones externas, D2 y D3 pero en otro tipo de Arduino hay más, **digitalPinToInterrupt()** proporciona el valor adecuado a la interrupción que se tiene que capturar según el pin que le pasemos.

Para las luces y el altavoz es fácil al ser salidas digitales:

```
pinMode(pinAltavoz, OUTPUT);
pinMode(pinLuzDerecha, OUTPUT);
pinMode(pinLuzIzquierda, OUTPUT);
pinMode(pinLuzMaquinista, OUTPUT);

digitalWrite(pinAltavoz, LOW);
digitalWrite(pinLuzDerecha, LOW);
digitalWrite(pinLuzIzquierda, LOW);
digitalWrite(pinLuzMaquinista, LOW);
```

Seguimos con las otras tareas:

6- Apagar luz maquinista y subir barrera

Para subir la barrera lo hacemos de forma parecida a bajar barrera, hay que tener en cuenta que siguen las luces intermitentes de carretera.

- a- Apagar luz Maquinista
- b- Conectar servo al pín
- c- Secuencia de luces de carretera, período 1 segundo
- d- Mover un poco la barrera hacia arriba
- e- Ha llegado a la posición de barrera subida? No: continuar en b
- f- Desconectar el servo

El código es fácil de realizar con la experiencia de la otra función.

```
digitalWrite (pinLuzMaquinista, LOW); // a- Apagar luz Maquinista
servoBarrera.attach (pinServo); // b- Conectar servo al pin
do {
    intermitenciaLuzCarretera(); // c- Secuencia de luces de carretera
    moverBarreraArriba(); // d- Mover un poco la barrera hacia arriba
} while (posicionServo < posicionBarreraSubida); // e- Repetir hasta posición de barrera subida
servoBarrera.detach (); // f- Desconectar servo
```

La función **moverBarreraArriba()** es como **moverBarreraAbajo()** pero actualizando la posición del servo hacia la posición de subida.

```
void moverBarreraArriba () {
    if (millis () - tiempoServo > 20) { //1- Si no han pasado 20ms no hacer nada.
        tiempoServo = millis(); // 2- Actualizar tiempo
        posicionServo = posicionServo + 1; // 3- Actualizar posición hacia barrera subida
        servoBarrera.write (posicionServo); // 4- Mover el servo a nueva posición
    }
}
```

Siguientes tareas:

7- Apagar luces carretera

Este casi no necesita explicación:

```
digitalWrite (pinLuzDerecha, LOW);
digitalWrite (pinLuzIzquierda, LOW);
```

4 - Apagar sonido y encender luz maquinista

Para apagar el sonido usamos **noTone()**, encenderemos la luz intermitente del maquinista y luego en la siguiente tarea se realizará la intermitencia:

```
noTone (pinAltavoz);
digitalWrite (pinLuzMaquinista, HIGH);
```

Lo más difícil

“Pero, ¡ay!, o las cerraduras eran demasiado grandes, o la llave era demasiado pequeña, lo cierto es que no pudo abrir ninguna puerta. Sin embargo, al dar la vuelta por segunda vez, descubrió una cortinilla que no había visto antes, y detrás había una puertecita de unos dos palmos de altura. Probó la llave de oro en la cerradura, y vio con alegría que ajustaba bien” .

Alicia en el País de las Maravillas

Nos quedan dos tareas que parecían fáciles pero que no sabemos cómo realizar:

- 2- Esperar orden de bajada
- 5- Esperar orden de subida

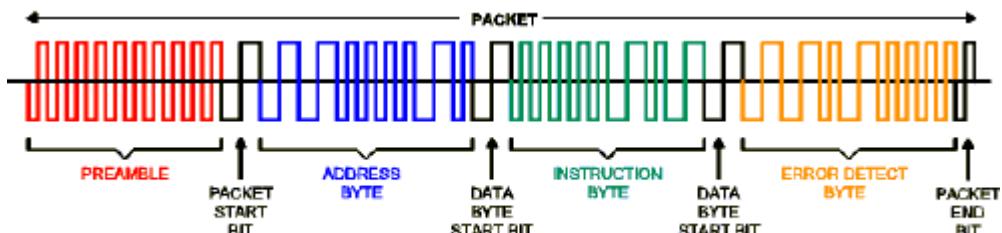
Si la orden se diera cuando se pulsa un botón se reduciría a:

- a- intermitencia de las luces si están activas
- b- Leer estado del botón
- c- Si no está pulsado, continuar en a

Pero lo que tenemos con la librería DCC es que se ejecuta la función `notifyDccAccTurnoutOutput()` cuando llega una orden DCC de accesorios, esta función tenemos que desarrollarla nosotros a nuestra conveniencia según los parámetros que nos lleguen, nosotros no podemos llamarla, además es del tipo `void` por lo que no nos devolvería ningún parámetro.

También tenemos que llamar a `Dcc.process()` para ir decodificando la señal de forma continua o no recibiremos las ordenes.

Para complicarlo un poco más, las centrales DCC suelen repetir la orden de accesorios mientras se mantiene pulsado el botón en el mando por lo que nos pueden llegar varios paquetes DCC iguales de activación de accesorio, con las consiguientes llamadas a la función `notifyDccAccTurnoutOutput()`, incluso algunas obvian el enviar la orden DCC de desactivación (Lenz en particular).



[link](#)

Vamos a ver la función en detalle.

```
void notifyDccAccTurnoutOutput( uint16_t Addr, uint8_t Direction, uint8_t OutputPower )
```

El parámetro `Addr` es la dirección de accesorio que deberemos comparar con la dirección que deseamos que muevan las barreras.

`Direction` indica si es una salida u otra (rojo / verde, recto / desviado), por ejemplo 0 sería bajar barrera y 1 subir barrera.

Finalmente `OutputPower` indica si es un paquete de activación cuando es 1 o desactivación cuando es 0. (Con Lenz solo nos llegarán a 1)

Lo más simple que se me ocurre es que al decodificar el paquete pongamos una variable booleana a **true** cuando corresponda con nuestra dirección así nuestra tarea se parecerá mucho a leer un pulsador pero en su lugar leeremos un variable.

Nuestra tarea de espera de orden quedaría así:

- a- Poner variable a false
- b- Intermisión de las luces si están activas
- c- Procesar señal DCC
- d- Leer variable
- e- Si variable es false continuar en b

y la función **notifyDccAccTurnoutOutput()** la escribiríamos así:

- 1- Si Addr no es nuestra dirección de accesorios no hacer nada
- 2- Si OutputPower no es de activación no hacer nada
- 3- Si Direction es 1 poner variable Subir a true
- 4- Si Direction es 0 poner variable Bajar a true.

```
void notifyDccAccTurnoutOutput( uint16_t Addr, uint8_t Direction, uint8_t OutputPower ) {  
    if (Addr == miDireccionDCC) {  
        if (OutputPower == 1) {  
            if (Direction == 1)  
                subirBarrera = true;  
            else  
                bajarBarrera = true;  
        }  
    }  
}
```

Las tareas quedarían así:

2- Esperar orden de bajada

```
bajarBarrera = false;  
do {  
    Dcc.process();  
} while (bajarBarrera == false);
```

5- Esperar orden de subida

```
subirBarrera = false;  
do {  
    intermitenciaLuzCarretera();  
    intermitenciaLuzMaquinista();  
    Dcc.process();  
} while (subirBarrera == false);
```

Dcc.process() también se tendría que ejecutar en cualquier otro bucle de espera así que lo añadiremos en los bucles **do-while()** que hemos escrito anteriormente para mover el servo.

El programa

“Todo tiene una moraleja, sólo falta saber encontrarla”.

Alicia en el País de las Maravillas

8- Volver a empezar (puede ser desde 1 o 2)

La última tarea ya nos la hace el **loop()**, una vez finalice con las tareas vuelve a empezar desde el principio del **loop()**.

La tarea 1 la pondremos en el **setup()** así que volverá a empezar desde la tarea 2.

Ya hemos definido todas las tareas y tenemos plasmado su código, ahora solo queda unirlo todo y poner las definiciones de variables y constantes que nos faltan de acuerdo a nuestro esquema y librerías:

```
#include <Servo.h>                                // Librerías
#include <NmraDcc.h>

const int pinAltavoz = 3;                            // definición de la conexión de los pines
const int pinLuzDerecha = 4;
const int pinLuzIzquierda = 5;
const int pinLuzMaquinista = 6;
const int pinServo = 7;
const int pinDCC = 2;

const int posicionBarreraSubida = 120;   // límites de movimiento del servo
const int posicionBarreraBajada = 10;

const int miDireccionDCC = 6;                      // Dirección de accesorio DCC a la que respondemos

Servo servoBarrera;                                // Objeto para la librería Servo.h
NmraDcc Dcc;                                       // Objeto para la librería NmraDcc.h
```



[link](#)

Y aquí todo junto:

```
// Paso a nivel DCC - Paco Cañada 2020
#include <Servo.h>                                // Librerías
#include <NmraDcc.h>

const int pinAltavoz = 3;                           // definición de la conexión de los pines
const int pinLuzDerecha = 4;
const int pinLuzIzquierda = 5;
const int pinLuzMaquinista = 6;
const int pinServo = 7;
const int pinDCC = 2;

const int posicionBarreraSubida = 120;             // límites de movimiento del servo
const int posicionBarreraBajada = 10;

const int miDireccionDCC = 6;                       // Dirección de accesorio DCC a la que respondemos

Servo servoBarrera;                               // Objeto para la librería Servo.h
NmraDcc Dcc;                                     // Objeto para la librería NmraDcc.h

unsigned long tiempoLuzCarretera;
unsigned long tiempoTono;
unsigned long tiempoServo;
unsigned long tiempoLuzMaquinista;

bool faseLuzCarretera;                            // variables para las fases
bool faseLuzMaquinista;
bool faseTono;

bool subirBarrera;                               // variables para las ordenes
bool bajarBarrera;

int posicionServo;                               // variable para el movimiento del servo

void setup() {
pinMode(pinAltavoz, OUTPUT);                     // 1- Poner barrera arriba y todo apagado
pinMode(pinLuzDerecha, OUTPUT);
pinMode(pinLuzIzquierda, OUTPUT);
pinMode(pinLuzMaquinista, OUTPUT);

digitalWrite(pinAltavoz, LOW);
digitalWrite(pinLuzDerecha, LOW);
digitalWrite(pinLuzIzquierda, LOW);
digitalWrite(pinLuzMaquinista, LOW);

posicionServo = posicionBarreraSubida;
servoBarrera.attach (pinServo);
servoBarrera.write (posicionServo);
delay (1000);
servoBarrera.detach ();

Dcc.pin (digitalPinToInterrupt (pinDCC), pinDCC, 1);
Dcc.initAccessoryDecoder (MAN_ID DIY, 1, FLAGS_OUTPUT_ADDRESS_MODE, 0);
}

void loop() {
bajarBarrera = false;                           // 2- Esperar orden de bajada
do {
  Dcc.process();
} while (bajarBarrera == false);

servoBarrera.attach (pinServo);                  // 3- Encender luces carretera, sonido y bajar barrera
do {
  Dcc.process();
  intermitenciaLuzCarretera();
  generarTono();
  moverBarreraAbajo();
} while (posicionServo > posicionBarreraBajada);
servoBarrera.detach ();

noTone(pinAltavoz);                            // 4 - Apagar sonido y encender luz maquinista
digitalWrite(pinLuzMaquinista,HIGH);
```

```

subirBarrera = false;                                // 5- Esperar orden de subida
do {
    intermitenciaLuzCarretera();
    intermitenciaLuzMaquinista();
    Dcc.process();
} while (subirBarrera == false);

digitalWrite (pinLuzMaquinista, LOW);                // 6- Apagar luz maquinista y subir barrera
servoBarrera.attach (pinServo);
do {
    Dcc.process();
    intermitenciaLuzCarretera();
    moverBarreraArriba();
} while (posicionServo < posicionBarreraSubida);
servoBarrera.detach ();

digitalWrite(pinLuzDerecha, LOW);                    // 7- Apagar luces carretera
digitalWrite(pinLuzIzquierda, LOW);
}

void intermitenciaLuzCarretera() {
    if (millis() - tiempoLuzCarretera >= 500) { // 1- Si no han pasado 0,5s (500ms) no hacer nada.
        tiempoLuzCarretera = millis();           // 2- Actualizar tiempo
        if (faseLuzCarretera == false) {          // 3- Si Fase es false: Encender luz derecha y apagar
izquierda
            digitalWrite (pinLuzDerecha, HIGH);
            digitalWrite (pinLuzIzquierda, LOW);
        }
        else {                                    // 4- Si Fase es true: Apagar luz derecha y encender
izquierda
            digitalWrite (pinLuzDerecha, LOW);
            digitalWrite (pinLuzIzquierda, HIGH);
        }
        faseLuzCarretera = ! faseLuzCarretera;    // 5- Cambiar Fase
    }
}

void generarTono() {
    if (millis() - tiempoTono >= 350) {           // 1- Si no han pasado 350ms no hacer nada.
        tiempoTono = millis();                     // 2- Actualizar tiempo
        if (faseTono == false) {                   // 3- Si Fase es false: generar tono de 277 Hz
            tone (pinAltavoz, 277);
        }
        else {                                    // 4- Si Fase es true: generar tono de 244Hz
            tone (pinAltavoz, 244);
        }
        faseTono = ! faseTono;                    // 5- Cambiar Fase
    }
}

void moverBarreraAbajo() {
    if (millis() - tiempoServo >= 20) {          // 1- Si no han pasado 20ms no hacer nada.
        tiempoServo = millis();                   // 2- Actualizar tiempo
        posicionServo = posicionServo - 1;       // 3- Actualizar posición hacia barrera bajada
        servoBarrera.write (posicionServo);
    }
}

void intermitenciaLuzMaquinista() {
    if (millis() - tiempoLuzMaquinista >= 500) { // 1- Si no han pasado 0,5s (500ms) no hacer nada.
        tiempoLuzMaquinista = millis();           // 2- Actualizar tiempo
        if (faseLuzMaquinista == false)
            digitalWrite (pinLuzMaquinista, HIGH);
        else
            digitalWrite (pinLuzMaquinista, LOW);
        faseLuzMaquinista = ! faseLuzMaquinista; // 5- Cambiar Fase
    }
}

void moverBarreraArriba(){
    if (millis() - tiempoServo > 20) {           // 1- Si no han pasado 20ms no hacer nada.
        tiempoServo = millis();                   // 2- Actualizar tiempo
        posicionServo = posicionServo + 1;       // 3- Actualizar posición hacia barrera subida
        servoBarrera.write (posicionServo);
    }
}

```

```
void notifyDccAccTurnoutOutput( uint16_t Addr, uint8_t Direction, uint8_t OutputPower ) {  
    if (Addr == miDireccionDCC) {  
        if (OutputPower == 1) {  
            if (Direction == 1)  
                subirBarrera = true;  
            else  
                bajarBarrera = true;  
        }  
    }  
}
```

Lo probamos y funciona!!!! Ya tenemos nuestro primer proyecto!!!

Probablemente se puedan mejorar muchas cosas pero lo importante es que hemos conseguido que funcione.

Ejercicio:

Añadir los pulsadores para subir y bajar barreras en D8 y D9, se conectarán a masa al pulsarlos y llevaran resistencia de pull-up a 5V

Pistas:

Solo requiere modificar las tareas de esperar orden de subida o bajada e inicializar los pines en el **setup()**.

Los botones estarán pulsados cuando se lea **LOW** en la entrada.

14. Bits problemáticos: Bits serie y retroseñalización S88

Con el control por ordenador se abren nuevas posibilidades de juego, entre otras:

- En modo automático podemos establecer rutas, horarios, trenes lanzadera ...
- Ser maquinista de nuestro propio tren respetando la señalización e integrarnos dentro del resto de circulaciones automáticas.
- Centrarnos en realizar maniobras mientras el resto de la maqueta continúa con vida propia.
- Podemos convertirnos en jefe de estación o controlador de CTC controlando la circulación de los trenes.

En este último caso, y sobre todo cuando se controlan los trenes en modo manual con otros amigos pululando por la maqueta y tenemos que ir cambiando desvíos en el sinóptico clicando en la pantalla del ordenador se pierde la visión de la maqueta y no se disfruta tanto como si tuviéramos un tablero de control óptico (TCO) desde el que contemplar y controlar nuestro pequeño mundo.

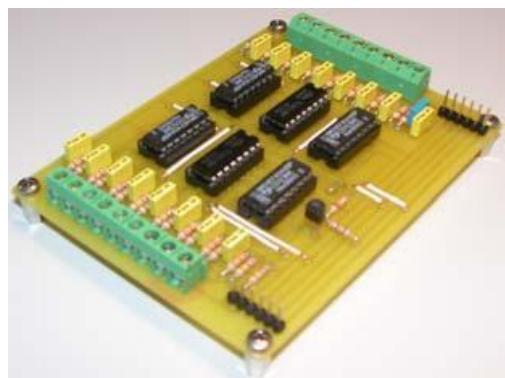


[link](#)

La idea:

Tengo un par de módulos S88 de 16 entradas que no uso ya que no tienen detectores de consumo así que no les he encontrado mucha utilidad hasta ahora para el control por ordenador de la maqueta.

http://usuaris.tinet.cat/fmco/s88_sp.html#S88



[link](#)

La necesidad agudiza el ingenio así que he pensado que me podrían servir para conectarles pulsadores y tener mi propio TCO. Sólo hay que encontrar la manera de leerlos y pasárselos al ordenador, así desde el ordenador podría hacer que un botón cambie un desvío o genere una ruta o lo que se me ocurra como si se hubiera clicado en la pantalla del ordenador.

Investigando...

Los S88 no son más que un chip que almacena el estado de las entradas en un registro de desplazamiento y que se puede leer en serie desplazando los bits de ese registro, además la salida de un chip alimenta la entrada de otro, es por esto que los S88 se conectan un detrás de otro como una cadena y las direcciones de las entradas dependen de la posición en que esté conectado en la cadena. El S88-N no es más que un cambio de conector (de 6 pins a RJ45).

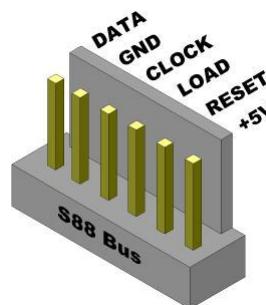


[link](#)

En Arduino la instrucción **shiftOut()** nos permite escribir en registros de desplazamiento como el chip 74595 usado en las pantallitas LED de 7 segmentos y la instrucción **shiftIn()** que nos permite leer registros de desplazamiento (como el chip 4021)

```
byte shiftIn(dataPin, clockPin, bitOrder)
```

shiftIn() nos devuelve un byte (nos lee 8 bits) simplemente pasándole el pin en el que se leen los datos (**dataPin**) y el que usa como reloj para desplazar los bits (**clockPin**), además podemos escoger con **bitOrder** si lee primero el bit de menor peso (**LSBFIRST**) o el bit de mayor peso (**MSBFIRST**)

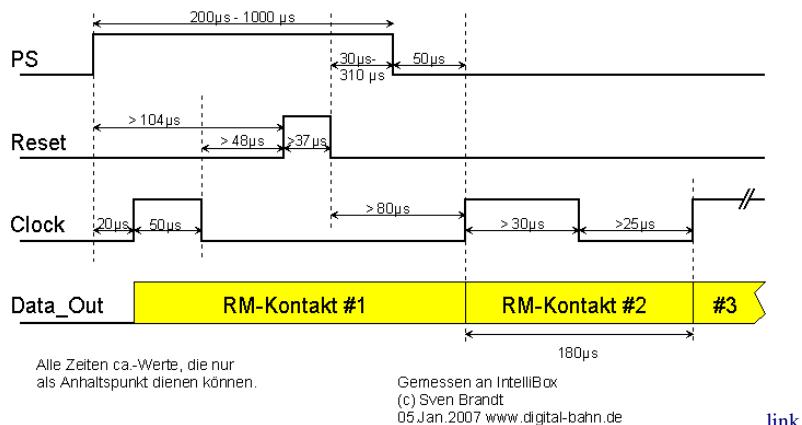


[link](#)

Hasta ahora no hay especificaciones oficiales para el bus s88, pero hay bastante información en internet para ver cómo hay que controlar sus señales:

<http://www.s88-n.eu/s88-timing-en.html>

https://www.digital-bahn.de/info_tech/s88.htm



[link](#)

Solo hay que generar las señales con este patrón usando **digitalWrite()** para empezar a leer los S88.

Ahora solo hay que pasar los datos al programa de control (Traincontroller, Rocail, iTrain, Windigipet,...). Estos programas permiten conectar varios interfaces para el control de maqueta, por lo que vamos a hacer que nuestro Arduino sea uno de esos interfaces.

Estos interfaces de las centrales digitales se comunican con el ordenador según un protocolo, si encontramos un protocolo simple de implementar que se pueda comunicar con él por el puerto serie (usaríamos **Serial**) y que soporte retroseñalización nos sería fácil conseguir nuestro objetivo sin elementos adicionales.

Algunos de estos protocolos son públicos, así que vamos a ver cuales hay disponibles:

Protocolo	Centrales	Enlace
Xpressnet	Lenz, Roco, DR5000	xpressnet-lan-usb-23151-v1.pdf
Loconet	Digitrax, Intellibox II, DR5000	loconetpersonaledition.pdf
Z21	Roco Z21, DR5000	z21-lan-protokoll.pdf
P50	Marklin 6050, Intellibox, Tams	interface.txt
HSI88	LDT HSI-88	hsi88_command-codes_en.pdf
RC-Link	TAMS RC-Link	RC-Talk-Protokoll.txt

El más simple de todos es el antiguo P50 usado por el interface Märklin 6050/6051 además es soportado por la mayor parte de los programas de control. Funciona por el puerto serie a 2400 baudios, 8 bits datos y 2 bits de stop con control de flujo CTS.

En este protocolo los comandos de locomotoras y accesorios están compuestos por dos bytes. El primer byte define el tipo de comando, mientras que el segundo byte representa la dirección del decodificador. Todos los demás comandos están compuestos de un solo byte.

Los únicos comandos que implican una respuesta de la interfaz son aquellos que se ocupan de los módulos de retroseñalización que envían dos bytes por módulo S88 que se corresponden con sus entradas.

Comando	Dirección	Descripción
1..31	1..80	Control locomotoras
32	-	Desactivación accesorio
33,34	0..255	Control accesorios
64..85	1..80	Control funciones locomotora F1..F4
96,97	-	Go, Stop
128 + n	-	Lectura de todos los S88 hasta el n (n=1..31)
192 + n	-	Lectura de un solo S88, el n (n=1..31)

En el programa de control de trenes solo configuraremos detecciones en este interfaz por lo que en principio no deberían llegar comandos para locomotoras y accesorios aunque los tendremos en cuenta para filtrarlos.

Es un interfaz bastante lento, tardaría 0,3s en enviar los datos de los 31 módulos S88 (9ms por módulo) pero para pocos modulos es más que suficiente.

Curiosidad: HSI-88 tardaría 0,1s para los 31 módulos (6ms por módulo). Loconet tardaría unos 1,2s para las 496 entradas (2,5ms por entrada) si cambiaseen todas a la vez.

Lectura de los S88

Para leer los S88 hay que generar las señales como el patrón para que se carguen los datos en los registros de desplazamiento y luego ya podemos leerlos. Es algo simple a base de **digitalWrite()** y esperas.

De entre los tiempos existentes vamos a escoger uno intermedio 50us como en NanoX, la espera la haremos con **delayMicroseconds()** en lugar de **delay()** ya que este último sería para esperas de milisegundos.

```
#define S88_TIME 50          // Temporizacion para S88. 50us como en NanoX-S88 de Paco
```

Para iniciar la lectura reproducimos el patrón:

```
void iniS88 () {
    digitalWrite (pinLoad, HIGH);      // Activamos Load/PS
    delayMicroseconds (S88_TIME);
    pulsoS88(pinClock);             // Pulso Clock
    pulsoS88(pinReset);            // Pulso Reset
    digitalWrite (pinLoad, LOW);     // Desactivamos Load
    delayMicroseconds (S88_TIME);
}
```

Para el pulso ya que básicamente hacemos lo mismo cambiando el pin, lo he puesto en una función aparte, en la que pasamos como parámetro el pin a controlar.

```
void pulsoS88 (int pinS88) {
    digitalWrite (pinS88, HIGH);      // Pulso
    delayMicroseconds (S88_TIME);
    digitalWrite (pinS88, LOW);
    delayMicroseconds (S88_TIME);
}
```

Los datos recibidos de la lectura de los S88 los guardamos en un array, ya que como máximo leemos 31 módulos y son de 16 bits (dos bytes) lo hacemos de 62 bytes:

```
byte datosS88[62];                      // Buffer para los datos leidos (31 modulos * 2 bytes)
```

La lectura de los S88 es generar el patrón y un bucle de los módulos que tengo (16 entradas: 2 bytes) que leemos con **shiftIn()** y guardamos en el array

```
#define S88_MODULOS 2                  // Numeros de modulos S88 de 16 entradas. Entre 1 y 31

void leeS88 () {
    iniS88();                         // Empezamos lectura
    for (int i=0; i< (S88_MODULOS * 2); i++) // Leemos los modulos
        datosS88[i] = shiftIn (pData, pinClock, MSBFIRST);
}
```

Interfaz P50 (Märklin 6050)

Este protocolo es bastante sencillo, se comunica a 2400baudios, 8 bits de datos, 2 bits de stop y sin paridad. En el **setup()** configuraremos Serial para esta configuración, como se aparta de lo normal (**SERIAL_8N1**), ya que usa dos bits de stop en lugar de uno, añadiremos otro parámetro en **Serial.begin()** con el valor adecuado.

```
Serial.begin (2400, SERIAL_8N2); // 2400 baudios, 2 stop bits, sin paridad
```

También inicializaremos el array de los datos leídos

```
for (int i=0; i<62; i++)          // Borramos buffer
    datosS88[i]=0;
```

Para decodificar los comandos haremos una función que lea el dato del puerto serie y en función del comando proceda.

Si es un comando de control de locomotoras (1..31 / 64..80) o de accesorios (33 o 34) leerá el siguiente byte (la dirección) disponible en el buffer aunque no hará ninguna acción.

```
void leeDireccion() {
    while (Serial.available() == 0); // Espera a que llegue el dato
    Serial.read();
}
```

Si es un comando de lectura de varios S88 (128 + n) calcularemos cuantos datos se han de enviar y se enviarán con **Serial.write()** los datos correspondientes del array. Usamos éste en lugar de **Serial.print()** ya que son datos binarios.

Si es un comando de lectura de un solo S88 (192 + n) calcularemos la posición en el array y enviaremos dos bytes.

El resto de comandos simplemente los ignoraremos.

```
void entradaSerie() {
    int c, n, i;

    c = Serial.read();                // leemos comando
    if (c >= 1 && c <= 31)           // Control locomotoras
        leeDireccion();
    if (c == 33 || c == 34)           // Control accesorios
        leeDireccion();
    if (c >= 64 && c <= 85)         // Control funcion locomotoras F1..F4
        leeDireccion();
```

```

if (c >= 129 && c <= 159) {           // Lectura de todos los S88, hasta el n
    n = (c - 128) * 2;
    for (i=0; i < n; i++)
        Serial.write (datosS88[i]);
}
if (c >= 193 && c <= 223) {           // Lectura de un solo S88, el n
    n = (c - 193) * 2;
    Serial.write (datosS88[n]);
    Serial.write (datosS88[n+1]);
}
}

```

El resto del programa:

Definimos los pines donde se conecta el S88:

```

const int pinClock = 3;
const int pinLoad = 4;
const int pinReset = 5;
const int pinData = 6;

```

Añadimos la inicialización en el **setup()**

```

pinMode (pinClock, OUTPUT);
pinMode (pinLoad, OUTPUT);
pinMode (pinReset, OUTPUT);
pinMode (pinData, INPUT_PULLUP);

digitalWrite (pinClock, LOW);      // para asegurarnos que las señales estan en el estado correcto
digitalWrite (pinLoad, LOW);
digitalWrite (pinReset, LOW);

```

Y en el **loop()** escribimos el bucle del programa:

```

void loop() {
    leeS88();
    if (Serial.available() > 0)      // Si hay caracteres en el puerto serie los interpretamos
        entradaSerie();
}

```

Ya podemos probarlo con nuestro programa de control por PC escogiendo como interfaz el Märklin 6050/6051.

Probando...

La primera en la frente... 😞



[link](#)

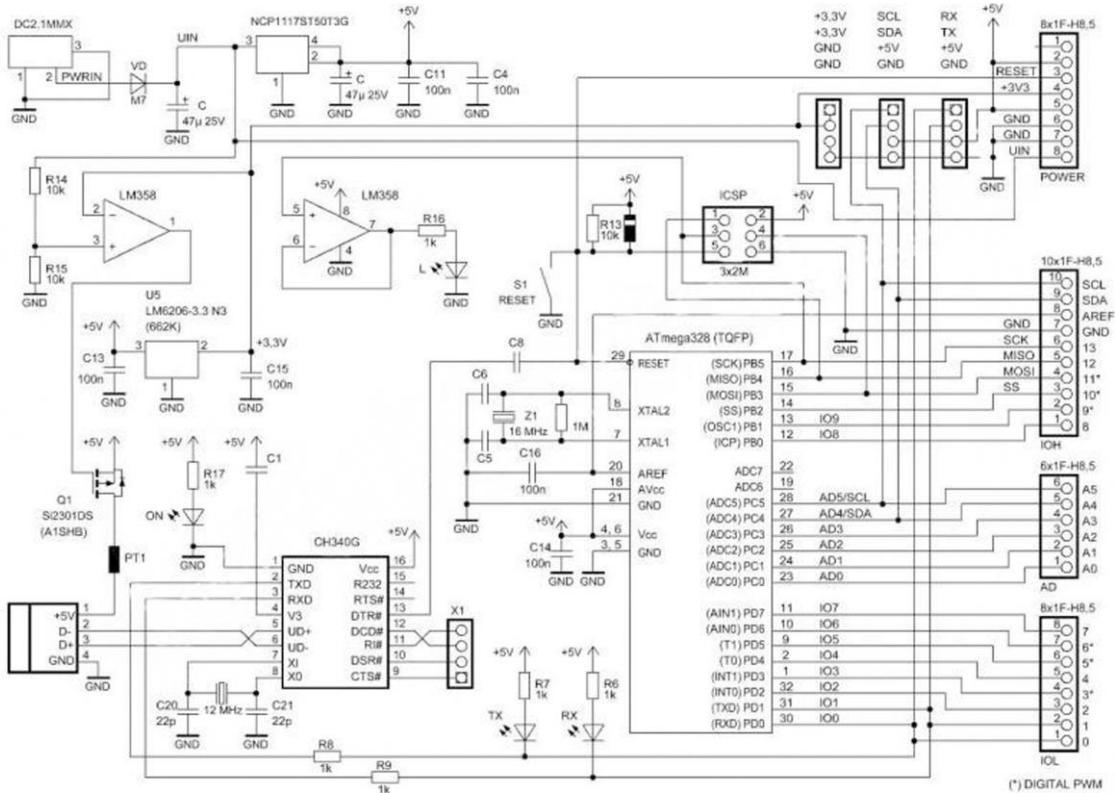
A pesar de configurar todo correctamente y elegir el protocolo más simple para no complicarnos con el programa no podemos comunicar con el programa de control.

Todos se quejan de que no pueden comunicar por la señal CTS, del control de flujo. Esta señal proviene del interfaz y le indica al ordenador cuando puede enviar datos.

Normalmente podemos configurar el puerto serie para que no use control de flujo pero los programas de control de trenes ya tienen asumido que con la interfaz Märklin hay que activarlo así que no te dan opción a desactivarlo.

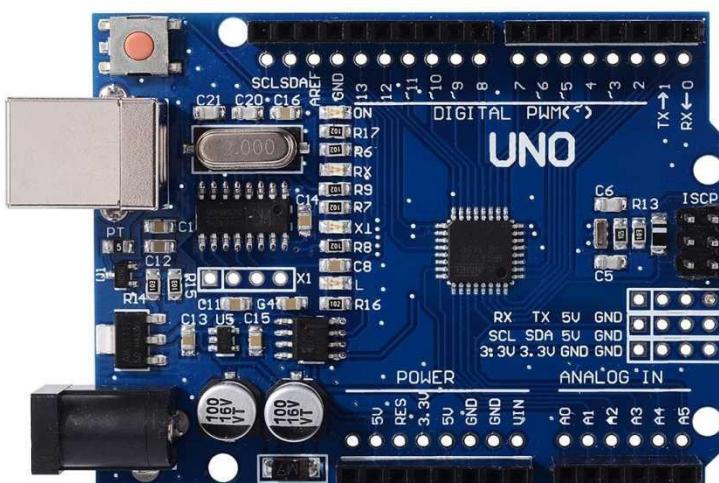
Leyendo por internet parece que en el Arduino original no se da este problema ya que usa un micro especialmente programado como convertidor USB-Serie, pero yo tengo un clon chino que usa el chip CH340G.

Este es el esquema del Arduino chino con USB-Serie CH340G:



Fijaos en el conector X1 debajo del chip USB, el primer pin (el marcado con cuadradito) es la señal CTS, que va al pin 9 del CH340G.

[link](#)



[link](#)

Tenemos que poner a **LOW** ese pin para que se active la comunicación. Lo conectaremos a GND.

Segundo problema...

Parece que todo funciona y me detecta todas las entradas del S88...menos la primera y además están desplazadas, la 2 es la 1, la 3 es la 2, etc. 😞

Este es algo más difícil, tiene que ver en cómo funciona `shiftIn()` y el chip 4014 de registro de desplazamiento de los S88.

Hemos puesto las señales como el patrón y tenemos `Clock` a `LOW` y el primer dato ya disponible en `Data`.

`ShiftIn()` primero pone a `Clock` a `HIGH` y luego lee el dato y finalmente pone `Clock` a `LOW` otra vez, y así para los 8 bits que ha de leer. Esta es su definición:

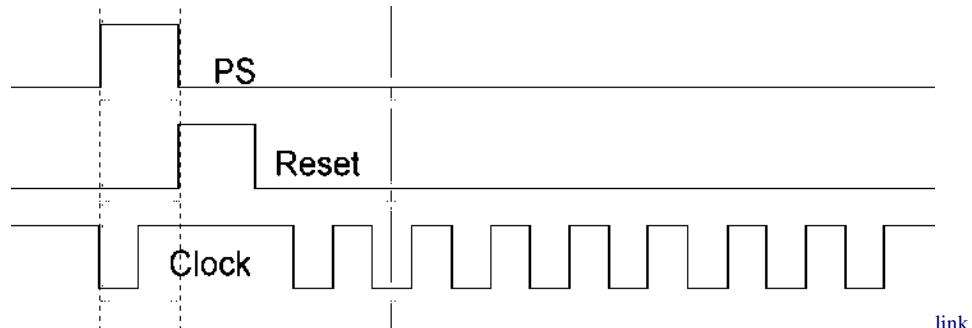
```
uint8_t shiftIn(uint8_t dataPin, uint8_t clockPin, uint8_t bitOrder) {
    uint8_t value = 0;
    uint8_t i;

    for (i = 0; i < 8; ++i) {
        digitalWrite(clockPin, HIGH);
        if (bitOrder == LSBFIRST)
            value |= digitalRead(dataPin) << i;
        else
            value |= digitalRead(dataPin) << (7 - i);
        digitalWrite(clockPin, LOW);
    }
    return value;
}
```

Con `<<` desplaza el bit leido a la posición que le toca y con `|=` lo añade a `value` para ir componiendo el valor.

Al poner el `Clock` a `HIGH` ya tenemos el siguiente dato en `Data` por lo que nos hemos saltado el primero en la lectura.

La ECOS usa un patrón diferente en el que `Clock` está a `HIGH` por defecto así que `shiftIn()` no cambiaría `Clock` cuando leyera la señal `Data`.



[link](#)

Tendríamos que cambiar el patrón como lo genera la ECOS:

```
void iniS88 () {
    digitalWrite (pinClock, HIGH); // clock a HIGH por defecto
    digitalWrite (pinLoad, HIGH); // Activamos Load/PS
    delayMicroseconds (S88_TIME);
    digitalWrite (pinClock, LOW);
    delayMicroseconds (S88_TIME);
    digitalWrite (pinClock, HIGH);
    delayMicroseconds (S88_TIME);
    digitalWrite (pinLoad, LOW); // Desactivamos Load
    digitalWrite (pinReset, HIGH); // Pulso Reset
    delayMicroseconds (S88_TIME);
    digitalWrite (pinReset, LOW);
    delayMicroseconds (S88_TIME);
}
```

Ahora ya nos lee correctamente todas las entradas. 😊

Tercer problema....

Este problema yo no lo tengo ya que mis S88 son a base de chips 4014 y 4044 pero hay quien usa S88 a base de micro controladores o incluso a base de Arduino.

<https://www.eliberia.es/index.php/digital/2-uncategorised/13-arduino-como-retromodulo-s88-arduin-as-s88-feedback-module>

Como se ve en la definición de `shiftIn()`, no hay ningún tiempo de espera entre poner `Clock` a `HIGH` y leer el pin `Data`, lo que en un Arduino Uno nos da un pulso alto de sólo 10us y unos 6us de bajo lo que suele ser demasiado rápido para estos tipos de S88.

Podemos escribir nuestro propio `shiftIn()` para este tipo de S88 y actualizamos nuestro `leeS88()` con lo que la lectura será más lenta:

```
byte shiftInS88 () {
    byte value = 0;
    int i;
    for (i = 7; i >= 0; i--) {
        digitalWrite (pinClock, HIGH);           // Clock a HIGH y espera
        delayMicroseconds (S88_TIME);
        value |= digitalRead (pData) << i;    // Leemos dato y guardamos bit
        digitalWrite (pinClock, LOW);           // Clock a LOW y espera
        delayMicroseconds (S88_TIME);
    }
    return (value);
}

void leeS88() {
    iniS88();                                // Empezamos lectura
    for (int i=0; i< (S88_MODULOS * 2); i++) // Leemos los modulos
        datosS88[i] = shiftInS88();
}
```

El programa

El programa queda así:

```
// Interfaz S88 con protocolo P50 (Maerklin 6050/6051) - Paco Cañada 2020

#define S88_MODULOS 2                      // Numeros de modulos S88 de 16 entradas. Entre 1 y 31
#define S88_TIME 50                         // Temporizacion para S88. 50us como en NanoX-S88 de Paco

const int pinClock = 3;
const int pinLoad = 4;
const int pinReset = 5;
const int pinData = 6;

byte datosS88[62];                      // Buffer para los datos leidos (31 modulos * 2 bytes)

void setup() {
    pinMode (pinClock, OUTPUT);
    pinMode (pinLoad, OUTPUT);
    pinMode (pinReset, OUTPUT);
    pinMode (pinData, INPUT_PULLUP);

    digitalWrite (pinClock, LOW);          // para asegurarnos que las señales estan en el estado correcto
    digitalWrite (pinLoad, LOW);
    digitalWrite (pinReset, LOW);

    Serial.begin (2400, SERIAL_8N2);      // 2400 baudios, 2 stops bits, sin paridad
    for (int i=0; i<62; i++)            // Borramos buffer
        datosS88[i]=0;
}

void loop() {
    leeS88();
    if (Serial.available() > 0)         // Si hay caracteres en el puerto serie los interpretamos
        entradaSerie();
}

void iniS88 () {
    digitalWrite (pinClock, HIGH); // clock a HIGH por defecto
    digitalWrite (pinLoad, HIGH); // Activamos Load/PS
    delayMicroseconds (S88_TIME);
    digitalWrite (pinClock, LOW);
```

```

delayMicroseconds (S88_TIME);
digitalWrite (pinClock, HIGH);
delayMicroseconds (S88_TIME);
digitalWrite (pinLoad, LOW); // Desactivamos Load
digitalWrite (pinReset, HIGH); // Pulso Reset
delayMicroseconds (S88_TIME);
digitalWrite (pinReset, LOW);
delayMicroseconds (S88_TIME);
}

byte shiftInS88 () {
    byte value = 0;
    int i;
    for (i = 7; i >= 0; i--) {
        digitalWrite (pinClock, HIGH); // Clock a HIGH y espera
        delayMicroseconds (S88_TIME);
        value |= digitalRead (pinData) << i; // Leemos dato y guardamos bit
        digitalWrite (pinClock, LOW); // Clock a LOW y espera
        delayMicroseconds (S88_TIME);
    }
    return (value);
}

void leeS88() {
    iniS88(); // Empezamos lectura
    for (int i=0; i< (S88_MODULOS * 2); i++) // Leemos los modulos
        datosS88[i] = shiftInS88();
}

void leeDireccion() {
    while (Serial.available() == 0); // Espera a que llegue el dato
    Serial.read();
}

void entradaSerie() {
    int c, n, i;

    c = Serial.read(); // leemos comando
    if (c >= 1 && c <= 31) // Control locomotoras
        leeDireccion();
    if (c == 33 || c == 34) // Control accesorios
        leeDireccion();
    if (c >= 64 && c <= 85) // Control funcion locomotoras F1..F4
        leeDireccion();
    if (c >= 129 && c <= 159) { // Lectura de todos los S88, hasta el n
        //digitalWrite (pinCTS,HIGH);
        n = (c - 128) * 2;
        for (i=0; i < n; i++)
            Serial.write (datosS88[i]);
        //digitalWrite (pinCTS,LOW);
    }
    if (c >= 193 && c <= 223) { // Lectura de un solo S88, el n
        //digitalWrite (pinCTS,HIGH);
        n = (c - 193) * 2;
        Serial.write (datosS88[n]);
        Serial.write (datosS88[n+1]);
        //digitalWrite (pinCTS,LOW);
    }
}

```

NOTA:

Si en lugar de TCO (pulsadores) se usa como detección de ocupación **DEBE** usarse con detectores optoacoplados en los S88.

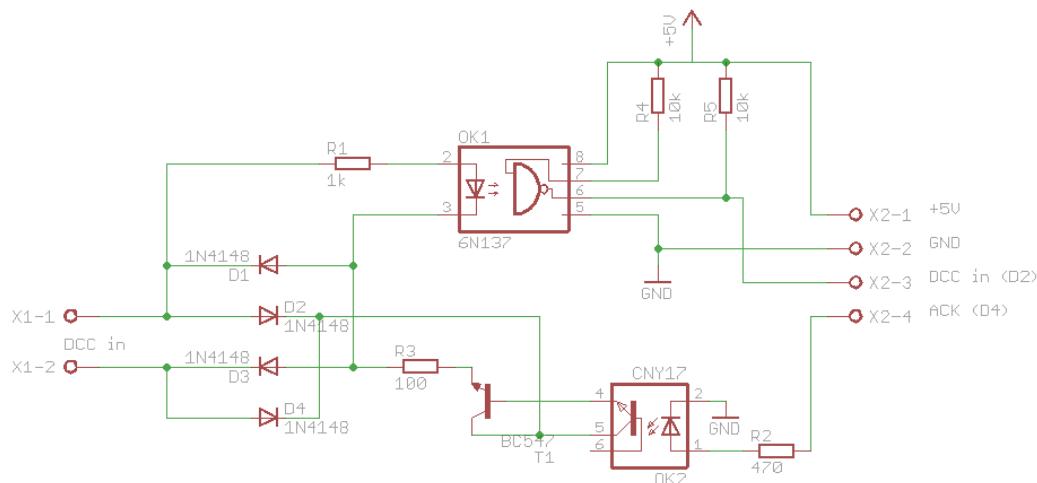
La señal DTR del interfaz USB-Serie en todos los Arduinos va conectada a su Reset para permitir arrancar el *bootloader* y programar el programa en el mismo. Normalmente no es problema para los programas de control de trenes pero alguno (Windigipet) se queja, la segunda vez que lo arranquemos ya detectara bien nuestro interfaz S88.

15. Memoria EEPROM: Decodificador de accesorios multiusos

Una de las principales ventajas de hacer un decodificador de accesorios propio con el Arduino es que lo podemos adaptar a nuestras necesidades. Así no necesitamos que todas las salidas sean iguales como la mayoría de los comerciales, por ejemplo.

Podemos tener más salidas de luces e incluir servos en un mismo decodificador incluso podemos variar los parámetros (dirección, movimiento, ...) cambiando las CV necesarias.

La librería [Nmradcc.h](#) tiene previsto el uso de CV, de hecho usa las CV29, CV7 y CV8 (además puede usar las CV9, CV17, CV18). También permite leer las CV pero necesitamos incluir la electrónica para la lectura de CV (pulso ACK) y programar alguna que otra rutina.



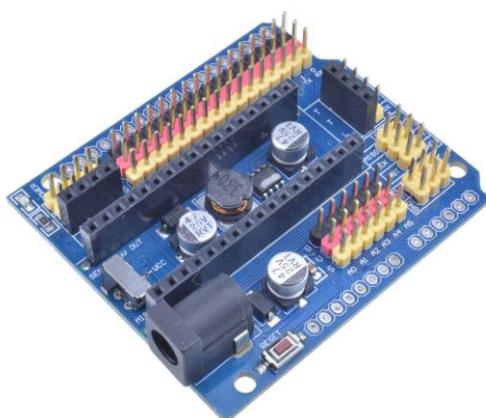
Cuando se tienen muchas CV en un decodificador es algo engorroso de manejar, y además cuando se quieren programar en los decodificadores de accesorios implica sacarlos del sitio y/o conectarlos a la vía de programación.

En un Arduino Uno/Nano tenemos un puerto serie USB que nos permite enviar textos que podríamos aprovechar para cambiar los valores de configuración (o CV) mediante lenguaje natural sin sacar ni desconectar el Arduino del sitio.

La idea

Hacer un decodificador de accesorios multiusos que sus salidas se puedan configurar como salida fija (luces, relé), parpadeante (luces), impulsos (relé biestable, bobinas) o servos, además cada salida puede tener una dirección distinta no necesitando que sea correlativa o única.

Para configurarlo se hará a través del puerto serie USB con opciones y menús en castellano.



[link](#)

Pensando...

Vamos a usar los máximos pines disponibles y queremos obtener diferentes funciones según programemos. Vamos a ver los requisitos para cada opción:

- **Salida fija**: Al recibir la orden DCC con nuestra dirección, si es la posición programada (rojo/verde) se activará la salida si es la otra posición se desactivará.

- **Salida parpadeante**: Al recibir la orden DCC con nuestra dirección, si es la posición programada (rojo/verde) se activará la salida parpadeando a un ritmo sincronizado para todas las salidas parpadeantes, si es la otra posición se desactivará.

- **Salida Servo**: Al recibir la orden DCC con nuestra dirección, se moverá a un ángulo programado según sea la posición que se active (rojo/verde) con una velocidad programada.

- **Salida impulso**: Al recibir la orden DCC con nuestra dirección, si es la posición programada (rojo/verde) se activará la salida durante un tiempo programado.

Al dar alimentación las salidas estarán apagadas. Si la CV8 nos es 13 (DIY-Do It Yourself) cargaremos una configuración por defecto con todos los pines como salida Fija, así desde DCC podremos resetear el decodificador. Para los servos seleccionaremos uno de los dos ángulos programados como inicial aunque no lo moveremos.

Si todas estas configuraciones las hiciéramos con CV tendríamos una gran cantidad de ellas para programar por lo que hacer un menú y programar las diferentes salidas desde el ordenador parece una buena idea, usaremos **Serial**.

Para no complicarnos, la entrada que realice el usuario para la configuración será numérica, así el menú mostrará las opciones y/o el rango para cada configuración:

Introduzca tipo de salida (0:FIJA, 1:FLASH, 2:SERVO, 3:PULSO)

Introduzca duracion pulso en ms (1-5100)

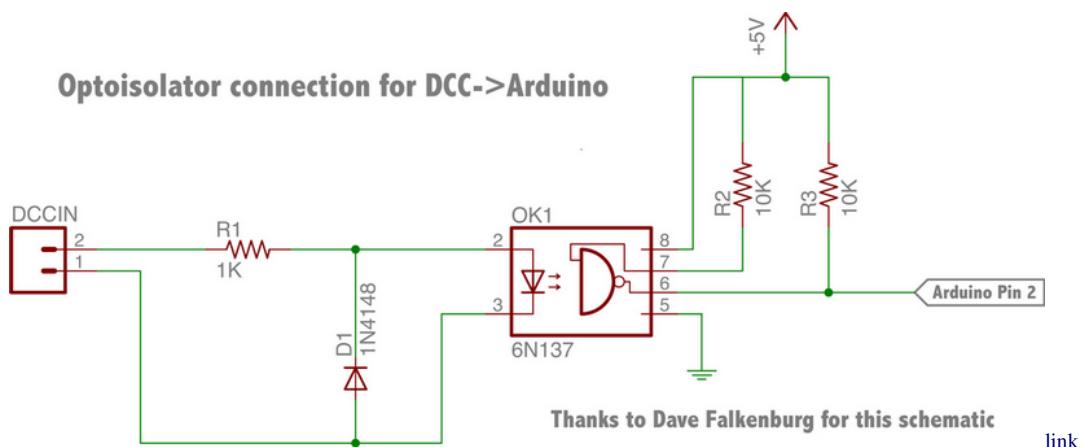
Las configuraciones para que no se pierdan cuando se apaga el decodificador las guardaremos en la memoria EEPROM, esta es una memoria que retiene los datos aunque se pierda la alimentación. Es donde se guardan las CV.

En un Arduino Uno/Nano es de 1024 bytes (1K) y en un Mega es de 4096 bytes (4K), permite un número limitado de escrituras, unas 100.000 en un único byte, por lo tanto, hay que evitar reescribir en el mismo byte para aumentar la vida útil de la EEPROM. Usaremos la librería **EEPROM.h**

<https://www.arduino.cc/en/Reference/EEPROM>

Para los servos usaremos la librería **Servo.h** que permite un máximo de 12 servos en un Arduino Uno/Nano, pero nos impide usar **analogWrite()** con los pines D9 y D10 (las usadas por el Timer1) pero como no necesitamos PWM nos sirve perfectamente.

Para decodificar la señal DCC usaremos la librería **Nmraddrcc.h** con el siguiente circuito:



Hecho un cesto, hecho ciento

Refrán popular

Cogemos nuestro lápiz y cuaderno y escribimos lo que necesitaríamos para una salida:

```
Setup()
    Si CV8 no es 13, guardar configuración por defecto.
    Leer configuración (pín, tipo, dirección,...) e inicializar
Loop()
    Si se recibe orden DCC según configuración: activar/desactivar salida
    Generar la secuencia adecuada (fija, flash, servo, pulso) si esta activada
    Si se recibe orden desde el puerto serie: mostrar, activar/desactivar o configurar salida

Configurar salida:
    Mostrar configuración actual
    Pedir nuevos datos (pín, tipo, dirección)
    Pedir datos según tipo
    Guardar o cancelar?
    Volver a leer configuración si ha cambiado
```

Lo que programemos para un pin nos servirá para otro, así que en lugar de repetir código intentaremos hacer uso de bucles, arrays y otras estructuras. Escribiremos pequeñas funciones, cuando se pueda, para que se puedan usar en varios sitios y para ayudar a la legibilidad. También evitaremos poner valores numéricos haciendo uso de **enum** y **#define** que permiten descripciones más claras de forma que podamos adaptarlos sin tener que tocar el código del programa.

Vamos a listar los pines que se pueden usar como salidas. Dejamos D0 y D1 ya que son para el puerto serie y D2 que es para la entrada DCC, para facilidad de uso los sustituimos por A0, A1, A2, así no tenemos que ir poniendo etiquetas a los pines. Dejamos libres A3, A4 y A5 por si nos interesa en el futuro usar el I2C, pulsadores o las entradas analógicas (también los podríamos añadir ahora y tener 17 salidas)

```
#define MAX_PIN 14      // numero maximo de pins a controlar
int salida[MAX_PIN]={A0, A1, A2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};
```

En el **setup()** para inicializarlos como salidas y apagarlos lo hacemos en un bucle, además definimos dos estados para ellos, así podremos cambiarlos a nuestro interés sin cambiar el código:

```
#define PIN_ON HIGH
#define PIN_OFF LOW

for (int n=0; n< MAX_PIN; n++) {                      // inicializar salidas
    pinMode (salida[n],OUTPUT);
    digitalWrite (salida[n],PIN_OFF);
}
```

Con la librería **Servo.h** Vamos a hacer un array de Servos para poder acceder a los objetos fácilmente y una variable para saber cuántos servos estamos usando ya que no podemos usar más de 12.

```
#define MAX_SERVO 12      // Numero maximo de servos a controlar (hasta 12)

Servo servo[MAX_SERVO];

int servosUsados;                                // contador de salidas de servo
```

NOTA: Los servos consumen una potencia considerable, por lo que si necesitamos controlar más de uno o dos, probablemente necesitemos alimentarlos desde un alimentador separado (es decir, no del pin de + 5V en su Arduino). Asegúrate de conectar las masas del Arduino y la fuente de alimentación externa.

La idea es que cada pin tenga su propia configuración, por lo que necesitará unos valores genéricos (dirección, tipo) y otros específicos (Par: rojo/verde, Fase: A o B, etc.) que dependiendo del tipo de salida tendrán unos valores u otros. Estos valores serán lo que guardaremos en la EEPROM.

Tipo	Dirección	Valor A	Valor B	Valor C
Luz Fija	1..2044	Par	-	-
Luz Flash	1..2044	Par	Fase	-
Servo	1..2044	Posición A	Posición B	Velocidad
Pulso	1..2044	Par	Duración	-

También necesitaremos unas variables auxiliares para el estado actual de la salida y para temporizaciones. Además para controlar el servo necesitaremos un par de variables más, la posición actual y el índice del array de objetos Servo

Como tenemos diferentes variables para cada pin podemos agruparlas en una estructura y hacer un array para todos los pines.

```
struct pinDef {                                // configuracion de las salidas
    byte tipo;
    int direccionDCC;
    byte valorA;
    byte valorB;
    byte valorC;
    unsigned long tiempo;                      // tiempo para la salida
    bool activado;                            // estado actual de la salida
    byte posServo;                            // posicion actual del servo
    int indice;                               // Indice al array de objetos servo
};

struct pinDef pinSalida[MAX_PIN];
```

Para los valores que contendrán tipo y Par (**valorA**) enumeramos los valores para tener un código mas claro:

```
enum tipoSalida { FIJA, FLASH, SERVO, PULSO};    // valores para tipo
enum valorPar { ROJO, VERDE};                     // valores para Par
```

Para cargar los datos en esta estructura tendremos que leerlos de la EEPROM para lo que usaremos las funciones de la librería **EEPROM**. La función **EEPROM.read()** lee el byte de la dirección que le pasemos (0 .. 1023 en caso del Arduino Uno/Nano)

Hay que tener en cuenta que la librería **Nmradcc.h** tiene previsto el uso de CV, que guarda en la EEPROM, y usa las CV29, CV7 y CV8 (además puede usar las CV1, CV9, CV17, CV18) así que elegiremos las nuestras a partir de la CV112 que están definidas en la NMRA como específicas para el fabricante.

Leeremos 6 bytes por salida (**tipo**, **direccionDCC**, **valorA**, **valorB**, **valorC**). Hay que tener en cuenta que **direccionDCC** son dos bytes por lo que usaremos el operador **<<** para colocar el byte alto en su sitio.

Estamos accediendo a variables dentro de la estructura **pinSalida[]** que es un array. Fijaros en los **[]** y el **.** para acceder a cada dato y en los **++** para actualizar la dirección EEPROM para la siguiente lectura.

```
#define CFG_BASE 112      // Direccion base de la EEPROM para las configuraciones

void leeConfigSalida (int numSalida) {    // Lee la configuracion de una salida desde la EEPROM
    int n = (numSalida * 6) + CFG_BASE;
    pinSalida[numSalida].tipo = EEPROM.read(n++);
    pinSalida[numSalida].direccionDCC = (EEPROM.read(n++) << 8) + EEPROM.read(n++);
    pinSalida[numSalida].valorA = EEPROM.read(n++);
    pinSalida[numSalida].valorB = EEPROM.read(n++);
    pinSalida[numSalida].valorC = EEPROM.read(n++);
}
```

Para inicializar todas las configuraciones, leemos la parte de la EEPROM y el resto le damos valores por defecto, también iremos contando los servos en caso de que los haya, hasta el máximo, y les asignaremos un índice para el array de objetos Servo y una posición por defecto.

```
void leeConfiguracion () {    // Lee todas las configuraciones y desactiva salidas por defecto
    servosUsados = 0;           // contador de servos
    for (int n=0 ; n < MAX_PIN; n++) {
        leeConfigSalida (n);    // lee configuracion desde la EEPROM
        pinSalida[n].tiempo = 0;  // resetea tiempo
        pinSalida[n].activado = false; // salida desactivada
        pinSalida[n].posServo = pinSalida[n].valorA; // posicion por defecto del servo
```

```

    if ((pinSalida[n].tipo == SERVO) && (servosUsados < MAX_SERVO)) { // si es un servo
        pinSalida[n].indice = servosUsados;                                // asignamos un indice
        servo[servosUsados].write (pinSalida[n].posServo); // y una posicion
        servosUsados++;                                                 // incrementamos el contador de servos
    }
    else {
        pinSalida[n].indice = 0;                                         // no es un servo o hay demasiados
    }
}
}

```

Para guardar la configuración escribiremos en la EEPROM forma similar. Con la función `EEPROM.write()` se guarda un byte (tarda unos 3.3ms) pero para minimizar las escrituras en la EEPROM usaremos `EEPROM.update()` que solo escribe si el valor es diferente del leído.

Para `direccionDCC` haremos uso de las funciones `highByte()` que nos da el byte alto y `lowByte()` que nos da el byte bajo del `int` que le pasemos como parámetro.

```

void escribeConfigSalida (int numSalida) {
    int n = (numSalida * 6) + CFG_BASE;
    EEPROM.update (n++, pinSalida[numSalida].tipo);
    EEPROM.update (n++, highByte(pinSalida[numSalida]. direccionDCC));
    EEPROM.update (n++, lowByte(pinSalida[numSalida]. direccionDCC));
    EEPROM.update (n++, pinSalida[numSalida].valorA);
    EEPROM.update (n++, pinSalida[numSalida].valorB);
    EEPROM.update (n, pinSalida[numSalida].valorC);
}

```

Inicialmente la EEPROM contendrá valores aleatorios (normalmente 255) así que en el `setup()` antes de inicializar la librería `NmraDCC` leeremos la CV8 (`CV_MANUFACTURER_ID`) y si no es 13 (`MAN_ID_DIY`) como la pasaremos a la librería resetearmos la configuración a salidas fijas a partir de la dirección 1, alternando el par de salida con lo que tendríamos la configuración de un decodificador de semáforos de dos aspectos.

Finalmente escribiremos `MAN_ID_DIY` en la CV8 para que no se vuelva a resetear.

```

void resetConfiguracion () {
    if (EEPROM.read (CV_MANUFACTURER_ID) != MAN_ID_DIY) {           // Si CV8 no es 13 reseteamos
        configuracion
        for (int n=0; n < MAX_PIN; n++) {
            pinSalida[n].tipo = FIJA;
            pinSalida[n].direccionDCC = (n >> 1) + 1;                  // la misma dirección para dos salidas
            pinSalida[n].valorA = (n & 0x01) ? VERDE : ROJO;          // Pares: ROJO, Impares: VERDE
            pinSalida[n].valorB = 0;
            pinSalida[n].valorC = 0;                                    // Lo guardamos en la EEPROM
        }
        EEPROM.update (CV_MANUFACTURER_ID, MAN_ID_DIY);
    }
}

```

Lo podríamos hacer a través de la función de la librería `notifyCVResetFactoryDefault()` como en sus ejemplos pero tendríamos que escribir una larga lista de números de CV (del 112 al 195) y sus valores, así es más compacto.

Con `notifyCVChange()` la librería nos informa de que se ha cambiado una CV, así que verificaremos si era la CV8 para resetear el decodificador.

```

void notifyCVChange( uint16_t CV, uint8_t Value) {
    resetConfiguracion(); // Si cambia una CV comprobamos si hay que resetear
}

```

Para inicializar la librería DCC, lo hacemos como indican sus ejemplos, indicamos el pin y la interrupción que usa y lo inicializamos como un decodificador de accesorios:

```

const int pinDCC = 2;

Dcc.pin (digitalPinToInterrupt (pinDCC), pinDCC, 1);
Dcc.initAccessoryDecoder (MAN_ID_DIY, 1, FLAGS_OUTPUT_ADDRESS_MODE, 0);

```

Activando...

La idea es que cuando se reciba por DCC un paquete de activación accesarios (se ejecutará `notifyDccAccTurnoutOutput()`) para que sea rápido de interpretar y podamos atender a otros comandos sólo comprobamos si la dirección coincide con alguna de nuestras salidas.

Según el tipo de salida si coincide también el par (rojo/verde) ponemos a `true` la variable `activado`, sino coincide el par lo pondremos a `false`.

Para la salida pulso además pondremos `tiempo` al tiempo actual con `millis()` ya que lo necesitaremos para hacer el temporizador.

Para los servos pondremos activado según el par, ROJO: `false`, VERDE: `true`, conectaremos el objeto servo al pin con `attach()` según el índice para que se pueda mover y pondremos `tiempo` al tiempo actual ya que lo necesitamos para el temporizador de la velocidad.

```
void notifyDccAccTurnoutOutput ( uint16_t Addr, uint8_t Direction, uint8_t OutputPower ) {
    if (OutputPower == 1) {                                     // solo comandos de activacion
        for (int n= 0; n < MAX_PIN; n++) {
            if (Addr == pinSalida[n].direccionDCC) {           // Si coincide la direccion
                activaSalida (n, Direction);                   // activa salida segun tipo
            }
        }
    }
}

void activaSalida (int numSalida, byte Direction) {
    switch (pinSalida[numSalida].tipo) {                      // activar segun tipo
        case FIJA:
        case FLASH:
            pinSalida[numSalida].activado = (pinSalida[numSalida].valorA == Direction) ? true : false;
            break;
        case PULSO:
            pinSalida[numSalida].activado = (pinSalida[numSalida].valorA == Direction) ? true : false;
            pinSalida[numSalida].tiempo = millis();           // inicializa tiempo para temporizador
            break;
        case SERVO:
            pinSalida[numSalida].activado = (Direction == ROJO) ? false : true;
            servo[pinSalida[numSalida].indice].attach (salida[numSalida]); // conecta pin al objeto
            servo correspondiente
            pinSalida[numSalida].tiempo = millis();           // inicializa tiempo para temporizador
            break;
    }
}
```

Dando vida...

Ahora que ya tenemos todo definido y preparado y sabemos cuándo se han de activar las salidas vamos a darles vida. Primero vamos a controlarlas.

La salida fija es bastante simple, si está activada se enciende, si está desactivada se apaga:

```
void salidaFija (int numSalida) {
    if (pinSalida[numSalida].activado == true)
        digitalWrite (salida[numSalida] , PIN_ON);
    else
        digitalWrite (salida[numSalida] , PIN_OFF);
}
```

Para la salida pulso cuando esta activa hacemos un temporizador que nos desactive la salida cuando ha pasado el tiempo establecido, como tenemos la duración (`valorB`) y cuando se ha activado en la rutina DCC nos ha puesto el tiempo inicial es fácil.

El resto es exactamente igual como la salida fija, así que la llamamos y no repetimos código.

El valor de la duración es como máximo 255 así que lo multiplicaremos por 20 para tener un rango entre 20ms y 5,1s más adecuado para manejar desvíos de bobina y de motores (para desvíos se necesita hardware adicional: transistor, ULN2803, L293D,...)

```
#define MULT_PULSO 20 // Multiplicador para el pulso

void salidaPulso (int numSalida) {
    if (pinSalida[numSalida].activado == true)
        if (millis() - pinSalida[numSalida].tiempo > (pinSalida[numSalida].valorB * MULT_PULSO)) // Entre 20ms y 5,1s
            pinSalida[numSalida].activado = false;
    salidaFija (numSalida);
}
```

Para la salida parpadeante tenemos que hacer dos cosas, como queremos que todas las salidas estén sincronizadas tenemos que hacer un temporizador general en el **loop()** que vaya cambiando el valor de una variable para que podamos compararla con la fase (**valorB**) en la función de control de la salida.

```
#define PARPADEO 500 // Parpadeo cada 0,5s

unsigned long tiempoFlash;

enum faseFlash {FASE_A, FASE_B};

byte Fase;

if (millis() - tiempoFlash > PARPADEO) { // temporizador para el parpadeo
    tiempoFlash = millis();
    Fase = (Fase == FASE_A) ? FASE_B : FASE_A; // cambia la fase
}
```

En la función de control comparamos la fase del parpadeo con la establecida en la configuración, cuando esta activa si coincide se encenderá en cualquier otro caso se apagará.

```
void salidaFlash (int numSalida) {
    if (pinSalida[numSalida].activado == true) { // si esta activado comprueba la fase
        if (Fase == pinSalida[numSalida].valorB)
            digitalWrite (salida[numSalida] , PIN_ON); // si coincide la fase se enciende
        else
            digitalWrite (salida[numSalida] , PIN_OFF); // si no coincide se apaga
    }
    else
        digitalWrite (salida[numSalida] , PIN_OFF); // si esta desactivado se apaga
}
```

Para la salida servo tenemos que comprobar si la posición actual coincide con la posición final (**valorA** o **valorB**) según **activado**.

Una vez ha pasado un tiempo según **valorC** (velocidad), los ms por grado programados, si coincide la posición final y está conectado lo desconectaremos con **detach()** para evitar vibraciones y bajar el consumo.

Si no coincide moveremos el servo un grado hacia la posición final con **write()**

```
void salidaServo (int numSalida) {
    byte posFinal = (pinSalida[numSalida].activado == true) ? pinSalida[numSalida].valorB : pinSalida[numSalida].valorA;
    if (millis() - pinSalida[numSalida].tiempo > (pinSalida[numSalida].valorC)){ //espera segun velocidad
        pinSalida[numSalida].tiempo = millis();
        if (pinSalida[numSalida].posServo == posFinal) {
            if (servo[pinSalida[numSalida].indice].attached()) { //si esta en posicion y conectado lo desconectamos
                while (digitalRead(salida[numSalida]) == HIGH); // Esperamos a que acabe el pulso
                servo[pinSalida[numSalida].indice].detach(); // lo desconectamos
                digitalWrite(salida[numSalida],LOW); // nos aseguramos de que esta desconectado
            }
        }
        else {
            if (posFinal > pinSalida[numSalida].posServo) // actualiza posicion
                pinSalida[numSalida].posServo++;
            else
                pinSalida[numSalida].posServo--;
            servo[pinSalida[numSalida].indice].write(pinSalida[numSalida].posServo); //muestra hacia posicion final
        }
    }
}
```

Ahora solo nos queda en el loop() hacer un bucle con todos los pines que nos vaya llamando a las funciones de salida correspondientes según el tipo. También hay que llamar a `Dcc.process()` para que vayamos recibiendo los comandos por DCC.

```
for (int n=0; n < MAX_PIN; n++) {
    Dcc.process();
    switch (pinSalida[n].tipo) {
        case FIJA:
            salidaFija(n);
            break;
        case FLASH:
            salidaFlash(n);
            break;
        case PULSO:
            salidaPulso(n);
            break;
        case SERVO:
            salidaServo(n);
            break;
    }
}
```

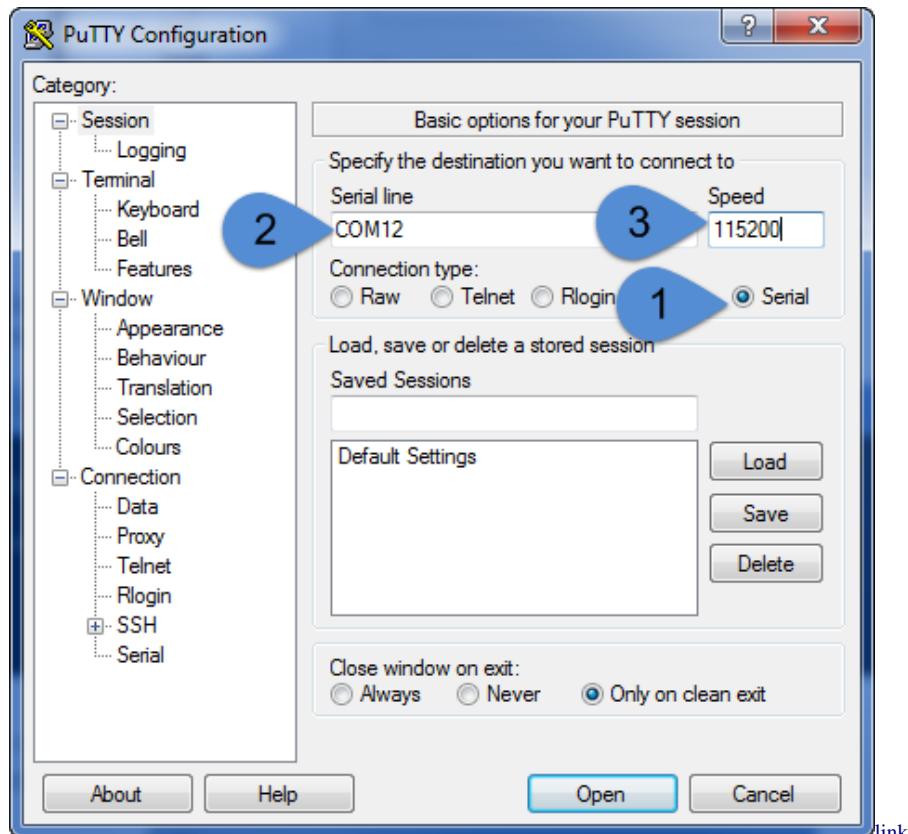
Con esto ya tendríamos un decodificador de accesorios multiusos DCC plenamente funcional.

Se pueden programar las CV (hay 84 en total para las salidas) y obtener una función distinta en cada pin. Se puede poner la misma dirección a varias salidas por lo que se activarían a la vez

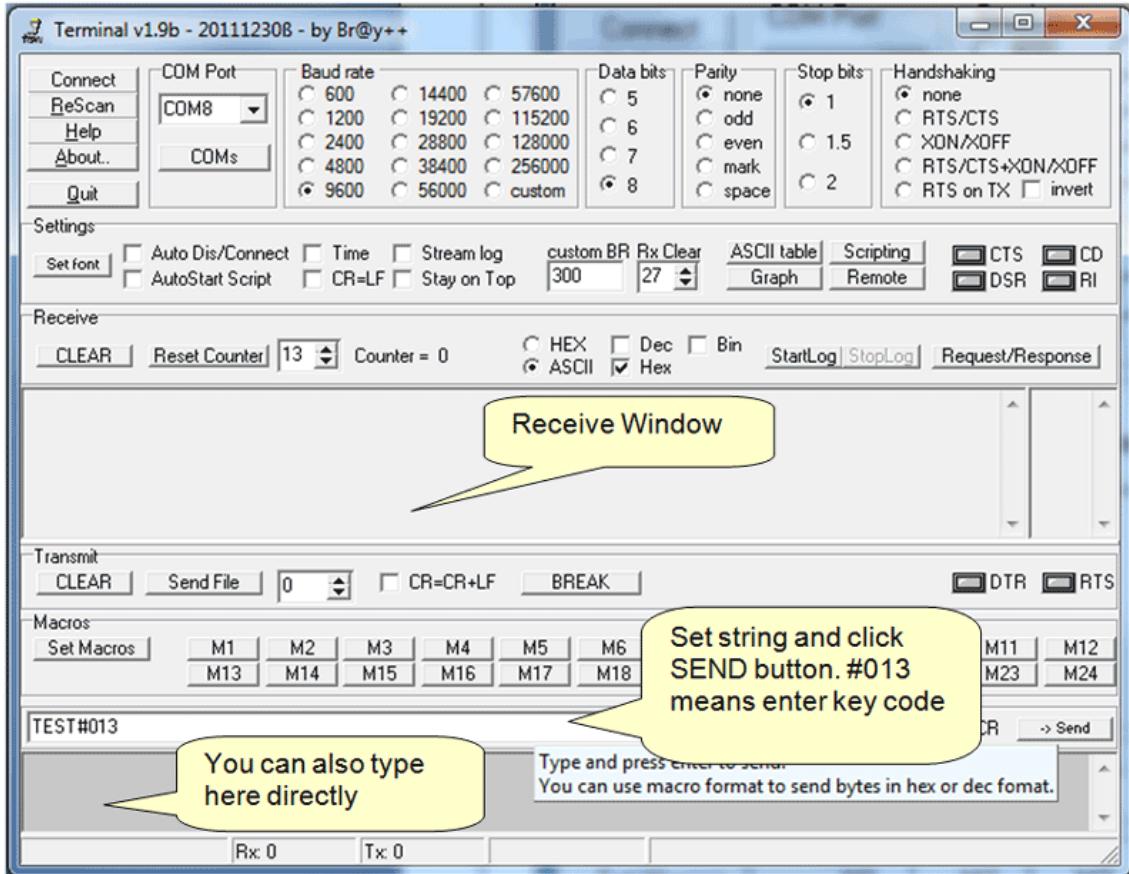
Comunicandonos...

Para cambiar la configuración en lugar de ir modificando CV, que también lo podemos hacer pero es engorroso, lo haremos cómodamente desde el ordenador y en castellano usando el Monitor Serie u otro programa de terminal como el PuTTY, o el Terminal que son bastante ligeros y no requieren instalación.

<https://www.putty.org/>



[link](#)



[link](#)

La velocidad no es importante mientras coincida con la establecida en el ordenador aunque nos conviene que sea rápida.

```
Serial.begin (115200);
```

Una de las cosas que queremos es que nos muestre según el tipo de salida un mensaje con sus características:

```
Salida 0: Direccion 1 ROJO - FIJA estado: ON
Salida 1: Direccion 1 VERDE - FLASH FASE A estado: OFF
Salida 2: Direccion 5 ROJO - PULSO de duracion 200ms
Salida 3: Direccion 3 SERVO - ROJO: 0 VERDE:180 velocidad 30ms/grado angulo actual 0
...
...
```

Podemos listar todas las salidas y ver su estado según el tipo o si se esta configurando una salida solo ver esa salida sin indicar su estado ya que se inicializará después.

Usaremos la macro **F()** en los textos de **Serial.print()** que imprimirá directamente desde la memoria Flash (la de programa) y nos ahorrará algo de memoria RAM.

Podemos usar el carácter **\n** para cambiar de línea. También comprobaremos si se ha superado el número máximo de servos, en cuyo caso pondremos un mensaje de alerta.

Para formatear el texto (nos salga alineado) usaremos la función **sprintf()** que escribe un texto en un array de caracteres (de tamaño suficiente) según el formato que le pasemos y las variables.

En formato se usa por cada variable: **%d** para enteros, **%u** para enteros sin signo, **%s** para arrays de texto, **%X** para números hexadecimales, **%c** para caracteres, **%f** para números en coma flotante. Si se pone un numero después de % el array será de esa longitud rellenando con espacios o con ceros si pone un 0:

Example	Variable Type	Value	Format Specifier	Description of format specifier	Output
1	char*	"ArduinoBasics"	%s	string	ArduinoBasics
2	char	'a'	%c	character	a
3	int	49	%c	character	1
4	int	49	%d	signed int	49
5	int	49	%8d	fixed width (8 characters)	_____49
6	int	49	%07d	fixed width (7 characters)-zero prefix	0000049
7	float	99.9911	%f	float	99.991096
8	float	99.9911	%.1f	float (1 decimal place)	100.0
9	float	99.9911	%.3f	float (3 decimal places)	99.991
10	float	99.9911	%e	scientific notation	9.999110e+01
11	float	99.9911	%g	shortest representation of %e or %f	99.9911
12	int	49	%#o	unsigned octal	061
13	int	49	%x	unsigned hex	31
14	int	49	%#x	unsigned hex	0x31
15	long long int	1234567890123456789LL	%lld	long long int	1234567890123456789

[link](#)

```

void printConfiguracionSalidas () {
    Serial.println (F("\nConfiguracion de las salidas\n"));
    for (int n=0; n < MAX_PIN; n++) {
        printSalida (n, true);
        Dcc.process(); // ir procesando comandos DCC
    }
    if (servosUsados > MAX_SERVO) {
        Serial.print (F("\nATENCION: Hay demasiados servos. Maximo "));
        Serial.println (MAX_SERVO);
    }
}

void printSalida (int numSalida, bool verEstado) {
    int valor;
    char dato[80]; // Tamaño suficiente para una linea
    sprintf (dato,"Salida %2d - Direccion:%5d", numSalida, pinSalida[numSalida].direccionDCC); // "Salida xx - Direccion: yyyy"
    Serial.print (dato);
    if (pinSalida[numSalida].tipo != SERVO) { // si no es servo imprime ROJO o VERDE segun par
        if (pinSalida[numSalida].valorA == ROJO)
            Serial.print (F(" ROJO - "));
        else
            Serial.print (F(" VERDE - "));
    }
    switch (pinSalida[numSalida].tipo) { // imprime configuracion segun tipo
        case FIJA:
            Serial.print(F("FIJA"));
            printEstado(numSalida, verEstado);
            break;
    }
}

```

```

        case FLASH:
            if (pinSalida[numSalida].valorB == FASE_A)
                Serial.print (F("FLASH FASE A"));
            else
                Serial.print (F("FLASH FASE B"));
                printEstado(numSalida, verEstado);
            break;
        case PULSO:
            Serial.print (F("PULSO de duracion "));
            valor = (int)pinSalida[numSalida].valorB * MULT_PULSO;
            Serial.print (valor);
            Serial.println(F("ms"));
            break;
        case SERVO:
            sprintf (dato," SERVO - ROJO: %d VERDE: %d velocidad %dms/grado angulo actual %d",
            pinSalida[numSalida].valorA, pinSalida[numSalida].valorB, pinSalida[numSalida].valorC,
            pinSalida[numSalida].posServo);
            Serial.println (dato);
            break;
    }
}

void printEstado (int numSalida, bool verEstado) {
    if (verEstado) {
        Serial.print (F(" estado: "));
        if (pinSalida[numSalida].activado)
            Serial.println(F("ON"));
        else
            Serial.println(F("OFF"));
    }
    else
        Serial.println("");
}

```

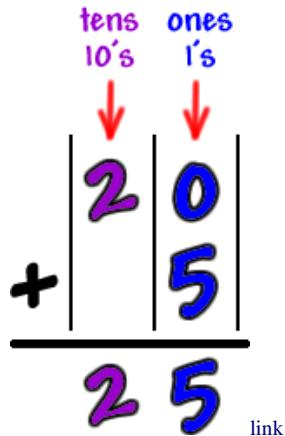
Configurando...

Para cambiar la configuración de una salida necesitamos unas variables globales auxiliares que vayan guardando los diferentes valores que va entrando el usuario:

```
int selPin=0, selDireccion, selTipo, selValorA, selValorB, selValorC; //variables para configurar salida
```

Los valores que introducirá el usuario serán numéricos, entre 0 y un máximo por lo que haremos una función que espere a que el usuario escriba los números y que componen el valor y pulse ENTER, luego como precaución fijaremos ese valor para que esté dentro del rango.

Como es una espera iremos llamando a `Dcc.process()` para que vaya procesando las ordenes.



[link](#)

Para componer el numero, inicialmente pondremos `valor` a 0 y comprobaremos que el carácter que ha introducido sea un dígito, si es así multiplicaremos `valor` por 10 y le sumaremos el nuevo dígito. Solo permitiremos que pulse ENTER (carácter `\r`) si ha introducido algún dígito, así evitamos errores.

```

int entradaSerie (int maximo) {
    int n=0,c=0,valor=0;
    do {
        if (Serial.available()>0) { // espera a que llegue un caracter procesando
DCC
        c = Serial.read();
        if (isDigit(c)) {
            valor = (valor * 10) + (c - '0'); // el digito leido se añade a valor
            n++;
        }
    } else {
        Dcc.process();
    }
} while ((c != '\r') || (n == 0)); // hasta que se pulse enter
valor = constrain (valor, 0, maximo); // lo ajustamos
return (valor);
}

```

Una de las entradas que le pediremos al usuario es que indique el par (tecla roja/verde) que quiere asignar a esa salida, así que le informamos de que queremos que entre y le indicamos los valores numéricos que corresponden a las opciones:

Introduzca posicion de activacion de la salida FIJA (0:ROJO, 1:VERDE)

Llamaremos a `entradaSerie()` con 1 como número máximo y guardaremos su elección en la variable. Despues le informamos de su elección.

```

void entradaPar (int tipoSalida) {
    Serial.print (F("\nIntroduzca posicion de activacion de la salida "));
    switch (tipoSalida) {
        case FIJA:
            Serial.print (F("FIJA"));
            break;
        case FLASH:
            Serial.print (F("FLASH"));
            break;
        case PULSO:
            Serial.print (F("PULSO"));
            break;
    }
    Serial.println (F(" (0:ROJO, 1:VERDE)"));
    selValorA = entradaSerie(1);
    Serial.print (F("Nueva posicion: "));
    if (selValorA == ROJO)
        Serial.println (F("ROJO"));
    else
        Serial.println (F("VERDE"));
}

```

Para cada tipo de salida requeriremos los datos necesarios:

```

void configuraFIJA() {
    entradaPar (FIJA);
    selValorB = 0;
    selValorC = 0;
}

void configuraFLASH() {
    entradaPar (FLASH);
    Serial.println (F("\nIntroduzca Fase (0:FASE A, 1:FASE B)"));
    selValorB = entradaSerie(1);
    Serial.print (F("Nueva fase: "));
    if (selValorA == FASE_A)
        Serial.println (F("FASE A"));
    else
        Serial.println (F("FASE B"));
    selValorC = 0;
}

```

```

void configuraPULSO() {
    entradaPar (PULSO);
    Serial.print (F("\nIntroduzca duracion pulso en ms (1-"));
    int valor = 255 * MULT_PULSO;
    Serial.print (valor);
    Serial.println (F("")));
    selValorB = entradaSerie(valor) / MULT_PULSO;
    selValorC = 0;
}

void configuraSERVO() {
    Serial.println (F("\nIntroduzca angulo posicion ROJO (0-180) "));
    selValorA = entradaSerie(180);
    Serial.print (F("\nNuevo angulo (ROJO): "));
    Serial.println (selValorA);
    Serial.println (F("\nIntroduzca angulo posicion VERDE (0-180) "));
    selValorB = entradaSerie(180);
    Serial.print (F("\nNuevo angulo (VERDE): "));
    Serial.println (selValorB);
    Serial.println (F("\nIntroduzca velocidad (0-255) Usual: 5..50ms por grado"));
    selValorC = entradaSerie(255);
}

```

Como cuando se configura una salida no estamos en el **loop()** sino en un bucle de espera que solo procesa la señal DCC y no actualiza las salidas mientras escribe el usuario podría darse el caso de que alguna salida tipo pulso quedase activa lo que podría ser perjudicial si estuviera conectado a una bobina así que las desactivaremos al entrar a configurar una salida.

```

void desactivaPulsos () {
    for (int n=0; n< MAX_PIN; n++) {                                // desconecta las salidas PULSO
        if (pinSalida[n].tipo == PULSO) {
            pinSalida[n].activado = false;
            digitalWrite (salida[n],PIN_OFF);
        }
    }
}

```

Para configurar una salida le pediremos en numero de salida a configurar y le mostraremos el estado actual. Los valores los iremos guardando en las diferentes variables auxiliares.

Luego le pedimos la dirección DCC para esa salida. Como la dirección 0 no es válida usaremos ese valor para cancelar la operación.

Le pediremos el tipo de salida y según él llamaremos a las funciones definidas anteriormente que les pedirán los parámetros adecuados.

Cargaremos la nueva configuración y le daremos la oportunidad de guardar o cancelar los cambios. Si desea guardarla la escribiremos en la EEPROM y recargaremos la configuración del decodificador. Si cancela leeremos los datos originales de la salida desde la EEPROM.

```

void configuraSalida() {
    desactivaPulsos();                                         // desconecta las salidas pulso por seguridad
    Serial.print (F("\nIntroduzca salida a configurar entre 0 y "));
    Serial.println (MAX_PIN - 1);
    selPin = entradaSerie (MAX_PIN - 1);
    Serial.println (F("\nConfiguracion actual:"));
    printSalida (selPin, true);                               // Imprimimos configuracion pin seleccionado
    Serial.println (F("\nIntroduzca Direccion DCC (1-2044, 0: Salir)"));
    selDireccion = entradaSerie(2044);
    if (selDireccion == 0) {
        Serial.println (F("\nCambios cancelados"));
        return;
    }
    Serial.print (F("Nueva direccion: "));
    Serial.println (selDireccion);
    Serial.println (F("\nIntroduzca tipo de salida (0:FIJA, 1:FLASH, 2:SERVO, 3:PULSO)"));
    selTipo = entradaSerie(3);
    switch (selTipo) {
        case FIJA:
            configuraFIJA();
            break;
        case FLASH:
            configuraFLASH();
            break;
    }
}

```

```

        case PULSO:
            configuraPULSO();
            break;
        case SERVO:
            configuraSERVO();
            break;
    }
    pinSalida[selPin].tipo = selTipo;
    pinSalida[selPin].direccionDCC = selDireccion;
    pinSalida[selPin].valorA = selValorA;
    pinSalida[selPin].valorB = selValorB;
    pinSalida[selPin].valorC = selValorC;
    Serial.println(F("\nNueva configuracion:"));
    printSalida(selPin, false);
    Serial.println(F("\nCorrecto? (0:NO, 1:SI)"));
    if (entradaSerie(1)) {
        escribeConfigSalida(selPin);
        leeConfiguracion();
        Serial.println(F("\nCambios guardados"));
    }
    else {
        leeConfigSalida(selPin);
        Serial.println(F("\nCambios cancelados"));
    }
}

```

Ya solo queda añadir en el `loop()` el menú para que sepamos que opciones tenemos disponibles (información, configuración, mover ultima salida configurada, ayuda):

```

Introduzca uno de estos comandos
I: Informacion salidas
C: Cambia salida
-: Posicion ROJO. Salida 0
+: Posicion VERDE. Salida 0
?: Esta ayuda

```

```

if (Serial.available() >0) {
    int c = Serial.read(); // lee caracter comando
    switch (c) {
        case 'I': // I: Informacion
            printConfiguracionSalidas ();
            break;
        case 'C': // C: Configurar salida
            configuraSalida();
            break;
        case '-': // -: ultima salida configurada a ROJO
            activaSalida (selPin, ROJO);
            break;
        case '+': // +: ultima salida configurada a VERDE
            activaSalida (selPin, VERDE);
            break;
        case '\n':
        case '\r':
            break;
        case '?': // ?: Ayuda
        default:
            Serial.println (F("\nIntroduzca uno de estos comandos"));
            Serial.println (F("I: Informacion salidas"));
            Serial.println (F("C: Cambia salida"));
            Serial.print (F("-: Posicion ROJO. Salida "));
            Serial.print (selPin);
            Serial.print (F("+: Posicion VERDE. Salida "));
            Serial.println (selPin);
            Serial.println (F("?: Esta ayuda"));
            break;
    }
}

```

Otra cosa que nos queda por hacer en un buen decodificador de accesorios es que guarde el estado de las salidas para que al iniciar quede todo como la ultima vez.

Establecemos una zona a partir de la CV230 para guardar el estado, lo haríamos al final de la función `activaSalida()` una vez establecido el valor de activado/desactivado. Así también evitamos movimientos bruscos de los servos. Para evitar que las salidas pulso se activen al recibir alimentación consumiendo una gran cantidad de corriente las guardaremos como desactivadas.

```
#define ACT_BASE 230      // Direccion base para guardar el estado (CV230)

byte estado = (pinSalida[numSalida].activado) ? 1 : 0; // guarda estado actual
if (pinSalida[numSalida].tipo == PULSO)
    estado=0;
EEPROM.update(numSalida + ACT_BASE, estado);
```

En **leeConfiguracion()** cambiaríamos la asignación de activado y la posición actual del servo

```
if (EEPROM.read (n + ACT_BASE) == 0) {           // recupera el ultimo estado guardado
    pinSalida[n].activado = false;
    pinSalida[n].posServo = pinSalida[n].valorA; // posicion por defecto del servo
}
else {
    pinSalida[n].activado = true;
    pinSalida[n].posServo = pinSalida[n].valorB; // posicion por defecto del servo
}
```

En **resetConfiguracion()** ponemos el ultimo estado como desactivado

```
EEPROM.update (n + ACT_BASE, 0);                  // guardamos ultimo estado como desactivado
```

El programa

Ya tenemos escrito todas las rutinas así que el programa queda así:

```
// Decodificador de accesorios Multiusos - Paco Cañada 2020
#include <Servo.h>          // Librerias
#include <NmraDcc.h>

#define MAX_PIN     14      // Numero maximo de pins a controlar
#define MAX_SERVO   12      // Numero maximo de servos a controlar (hasta 12)
#define CFG_BASE   112      // Direccion base de la EEPROM para las configuraciones (CV112)
#define ACT_BASE   230      // Direccion base para guardar el estado (CV230)
#define PARPADEO  500      // Parpadeo cada 0,5s
#define MULT_PULSO 20      // Multiplicador para el pulso

#define PIN_ON HIGH
#define PIN_OFF LOW

Servo servo[MAX_SERVO];                      // Objetos
NmraDcc Dcc;

int salida[MAX_PIN]={A0, A1, A2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 };
const int pinDCC = 2;

struct pinDef {                                // configuracion de las salidas
    byte tipo;
    int direccionDCC;
    byte valorA;
    byte valorB;
    byte valorC;
    unsigned long tiempo;                      // tiempo para la salida
    bool activado;                            // estado actual de la salida
    byte posServo;                            // posicion actual del servo
    int indice;                               // Indice al array de objetos servo
};

struct pinDef pinSalida[MAX_PIN];

int servosUsados;                            // contador de salidas de servo
unsigned long tiempoFlash;                   // variables para flash
byte Fase;
int selPin=0, selDireccion, selTipo, selValorA, selValorB, selValorC; // variables para configurar salida

enum tipoSalida {FIJA, FLASH, SERVO, PULSO}; // valores para tipo
enum valorPar {ROJO, VERDE};                 // valores para Par
enum faseFlash {FASE_A, FASE_B};             // valores para fase

void setup() {
    for(int n=0; n< MAX_PIN; n++) {          // inicializar salidas
        pinMode (salida[n],OUTPUT);
        digitalWrite (salida[n],PIN_OFF);
    }
    Serial.begin (115200);
    Serial.println (F("Decodificador de accesorios Multiusos by Paco Cañada\n"));
    resetConfiguracion ();                    // comprueba CV8 y resetea si es necesario
    Dcc.pin (digitalPinToInterrupt (pinDCC), pinDCC, 1);
    Dcc.initAccessoryDecoder (MAN_ID_DIY, 1, FLAGS_OUTPUT_ADDRESS_MODE, 0);
    leeConfiguracion ();                     // lee configuracion de los pines
    tiempoFlash = millis();                  // tiempo para temporizador de parpadeo
    Serial.println (F("? para ver comandos"));
}
```

```

}

void loop() {
    if (millis() - tiempoFlash > PARPADEO) {      // temporizador para el parpadeo
        tiempoFlash = millis();
        Fase = (Fase == FASE_A) ? FASE_B : FASE_A; // cambia la fase
    }
    for (int n=0; n < MAX_PIN; n++) {
        Dcc.process();
        switch (pinSalida[n].tipo) {
            case FIJA:
                salidaFija(n);
                break;
            case FLASH:
                salidaFlash(n);
                break;
            case PULSO:
                salidaPulso(n);
                break;
            case SERVO:
                salidaServo(n);
                break;
        }
    }
    if (Serial.available() >0) {
        int c = Serial.read();                      // lee caracter comando
        switch (c) {
            case 'I':                           // I: Informacion
                printConfiguracionSalidas ();
                break;
            case 'C':                           // C: Configurar salida
                configuraSalida();
                break;
            case '-':                           // -: ultima salida configurada a ROJO
                activaSalida (selPin, ROJO);
                break;
            case '+':                           // +: ultima salida configurada a VERDE
                activaSalida (selPin, VERDE);
                break;
            case '\n':
            case '\r':
                break;
            case '?':                           // ?: Ayuda
            default:
                Serial.println (F("\nIntroduzca uno de estos comandos"));
                Serial.println (F("I: Informacion salidas"));
                Serial.println (F("C: Cambia salida"));
                Serial.print   (F("-: Posicion ROJO. Salida "));
                Serial.println (selPin);
                Serial.print   (F("+: Posicion VERDE. Salida "));
                Serial.println (selPin);
                Serial.println (F("?: Esta ayuda"));
                break;
        }
    }
}

void leeConfigSalida (int numSalida) {           // Lee la configuracion de una salida desde la EEPROM
    int n = (numSalida * 6) + CFG_BASE;
    pinSalida[numSalida].tipo = EEPROM.read(n++);
    pinSalida[numSalida].direccionDCC = (EEPROM.read(n++) << 8) + EEPROM.read(n++);
    pinSalida[numSalida].valorA = EEPROM.read(n++);
    pinSalida[numSalida].valorB = EEPROM.read(n++);
    pinSalida[numSalida].valorC = EEPROM.read(n);
}

void leeConfiguracion () { // Lee todas las configuraciones y desactiva salidas por defecto
    servosUsados = 0;          // contador de servos
    for (int n=0 ; n < MAX_PIN; n++) {
        leeConfigSalida (n);      // lee configuracion desde la EEPROM
        pinSalida[n].tiempo = 0;    // resetea tiempo
        if (EEPROM.read (n + ACT_BASE) == 0) { // recupera el ultimo estado guardado
            pinSalida[n].activado = false;
            pinSalida[n].posServo = pinSalida[n].valorA; // posicion por defecto del servo
        }
        else {
            pinSalida[n].activado = true;
            pinSalida[n].posServo = pinSalida[n].valorB; // posicion por defecto del servo
        }
        if ((pinSalida[n].tipo == SERVO) && (servosUsados < MAX_SERVO)) { // si es un servo
            pinSalida[n].indice = servosUsados; // asignamos un indice
            servo[servosUsados].write (pinSalida[n].posServo); // una posicion
            servosUsados++; // incrementamos el contador de servos
        }
        else {
            pinSalida[n].indice = 0; // no es un servo o hay demasiados
        }
    }
}

```

```

void escribeConfigSalida (int numSalida) {
    int n = (numSalida * 6) + CFG_BASE;
    EEPROM.update (n++, pinSalida[numSalida].tipo);
    EEPROM.update (n++, highByte(pinSalida[numSalida].direccionDCC));
    EEPROM.update (n++, lowByte(pinSalida[numSalida].direccionDCC));
    EEPROM.update (n++, pinSalida[numSalida].valorA);
    EEPROM.update (n++, pinSalida[numSalida].valorB);
    EEPROM.update (n, pinSalida[numSalida].valorC);
}

void resetConfiguracion () {
    if (EEPROM.read (CV_MANUFACTURER_ID) != MAN_ID DIY) { // Si CV8 no es 13 reseteamos configuracion
        Serial.println (F("Reseteando configuracion\n"));
        for (int n=0; n < MAX_PIN; n++) {
            pinSalida[n].tipo = FIJA;
            pinSalida[n].direccionDCC = (n >> 1) + 1; // la misma direccion para dos salidas
            pinSalida[n].valorA = (n & 0x01) ? VERDE : ROJO; // Pares: ROJO, Impares: VERDE
            pinSalida[n].valorB = 0;
            pinSalida[n].valorC = 0;
            escribeConfigSalida (n); // Lo guardamos en la EEPROM
            EEPROM.update (n + ACT_BASE, 0); // guardamos ultimo estado como desactivado
        }
        EEPROM.update (CV_MANUFACTURER_ID, MAN_ID DIY);
    }
}

void activaSalida (int numSalida, byte Direction) {
    switch (pinSalida[numSalida].tipo) { // activar segun tipo
        case FIJA:
        case FLASH:
            pinSalida[numSalida].activado = (pinSalida[numSalida].valorA == Direction) ? true : false;
            break;
        case PULSO:
            pinSalida[numSalida].activado = (pinSalida[numSalida].valorA == Direction) ? true : false;
            pinSalida[numSalida].tiempo = millis(); // inicializa tiempo para temporizador
            break;
        case SERVO:
            pinSalida[numSalida].activado = (Direction == ROJO) ? false : true;
            servo[pinSalida[numSalida].indice].attach (salida[numSalida]); // conecta pin al objeto servo
            correspondiente
            pinSalida[numSalida].tiempo = millis(); // inicializa tiempo para temporizador
            break;
    }
    byte estado = (pinSalida[numSalida].activado) ? 1 : 0; // guarda estado actual
    if (pinSalida[numSalida].tipo == PULSO)
        estado=0;
    EEPROM.update(numSalida + ACT_BASE, estado);
}

void notifyDccAccTurnoutOutput( uint16_t Addr, uint8_t Direction, uint8_t OutputPower ) {
    if (OutputPower == 1) { // solo comandos de activacion
        for (int n= 0; n < MAX_PIN; n++) {
            if (Addr == pinSalida[n].direccionDCC) { // Si coincide la direccion
                activaSalida (n, Direction); // activa salida segun tipo
            }
        }
    }
}

void notifyCVChange( uint16_t CV, uint8_t Value) {
    resetConfiguracion(); // Si cambia una CV comprobamos si hay que resetear
}

void salidaFija (int numSalida) {
    if (pinSalida[numSalida].activado == true) // pone la salida segun activado
        digitalWrite (salida[numSalida],PIN_ON);
    else
        digitalWrite (salida[numSalida],PIN_OFF);
}

void salidaPulso (int numSalida) {
    if (pinSalida[numSalida].activado == true) // si esta activado comprueba tiempo
        if (millis() - pinSalida[numSalida].tiempo > (pinSalida[numSalida].valorB * MULT_PULSO)) // Entre 20ms y 5,1s
            pinSalida[numSalida].activado = false;
    salidaFija (numSalida);
}

void salidaFlash (int numSalida) {
    if (pinSalida[numSalida].activado == true) // si esta activado comprueba la fase
        if (Fase == pinSalida[numSalida].valorB)
            digitalWrite (salida[numSalida],PIN_ON); // si coincide la fase se enciende
        else
            digitalWrite (salida[numSalida],PIN_OFF); // si no coincide se apaga
    else
        digitalWrite (salida[numSalida],PIN_OFF); // si esta desactivado se apaga
}

void salidaServo (int numSalida) {
}

```

```

byte posFinal = (pinSalida[numSalida].activado == true) ? pinSalida[numSalida].valorB :
pinSalida[numSalida].valorA;
if (millis() - pinSalida[numSalida].tiempo > (pinSalida[numSalida].valorC)) { // espera segun velocidad
    pinSalida[numSalida].tiempo = millis();
    if (pinSalida[numSalida].posServo == posFinal) {
        if (servo[pinSalida[numSalida].indice].attached()) { // si esta en posicion y conectado lo desconectamos
            while (digitalRead(salida[numSalida]) == HIGH); // Esperamos a que acabe el pulso
            servo[pinSalida[numSalida].indice].detach(); // lo desconectamos
            digitalWrite(salida[numSalida], LOW); // nos aseguramos de que esta desconectado
        }
    } else {
        if (posFinal > pinSalida[numSalida].posServo) // actualiza posicion
            pinSalida[numSalida].posServo++;
        else
            pinSalida[numSalida].posServo--;
        servo[pinSalida[numSalida].indice].write(pinSalida[numSalida].posServo); // mueve hacia posicion final
    }
}
}

void printEstado (int numSalida, bool verEstado) {
    if (verEstado) {
        Serial.print(F(" estado: "));
        if (pinSalida[numSalida].activado)
            Serial.println(F("ON"));
        else
            Serial.println(F("OFF"));
    } else
        Serial.println("");
}

void printSalida (int numSalida, bool verEstado) {
    int valor;
    char dato[80]; // Tamaño suficiente para una linea
    sprintf(dato, "Salida %d - Direccion:%d", numSalida, pinSalida[numSalida].direccionDCC); // "Salida xx - Direccion: yyyy"
    Serial.print(dato);
    if (pinSalida[numSalida].tipo != SERVO) { // si no es servo imprime ROJO o VERDE segun par
        if (pinSalida[numSalida].valorA == ROJO)
            Serial.print(F(" ROJO - "));
        else
            Serial.print(F(" VERDE - "));
    }
    switch (pinSalida[numSalida].tipo) { // imprime configuracion segun tipo
        case FIJA:
            Serial.print(F("FIJA"));
            printEstado(numSalida, verEstado);
            break;
        case FLASH:
            if (pinSalida[numSalida].valorB == FASE_A)
                Serial.print(F("FLASH FASE A"));
            else
                Serial.print(F("FLASH FASE B"));
            printEstado(numSalida, verEstado);
            break;
        case PULSO:
            Serial.print(F("PULSO de duracion "));
            valor = (int)pinSalida[numSalida].valorB * MULT_PULSO;
            Serial.print(valor);
            Serial.println(F("ms"));
            break;
        case SERVO:
            sprintf(dato, " SERVO - ROJO: %d VERDE: %d velocidad %dms/grado angulo actual %d",
                    pinSalida[numSalida].valorA, pinSalida[numSalida].valorB, pinSalida[numSalida].valorC,
                    pinSalida[numSalida].posServo);
            Serial.println(dato);
            break;
    }
}

void printConfiguracionSalidas () {
    Serial.println(F("\nConfiguracion de las salidas\n"));
    for (int n=0; n < MAX_PIN; n++) {
        printSalida(n, true);
        Dcc.process(); // ir procesando comandos DCC
    }
    if (servosUsados > MAX_SERVO) {
        Serial.print(F("\nATENCION: Hay demasiados servos. Maximo "));
        Serial.println(MAX_SERVO);
    }
}

int entradaSerie (int maximo) {
    int n=0, c=0, valor=0;
    do {
        if (Serial.available() > 0) // espera a que llegue un caracter procesando DCC
            c = Serial.read();
        if (isDigit(c)) {

```

```

        valor = (valor * 10) + (c - '0');           // el digito leido se añade a valor
        n++;
    }
}
else {
    Dcc.process();
}
} while ((c != '\r') || (n == 0));           // hasta que se pulse enter
valor = constrain (valor, 0, maximo);         // lo ajustamos
return (valor);
}

void desactivaPulsos () {
    for (int n=0; n< MAX_PIN; n++) {           // desconecta las salidas PULSO
        if (pinSalida[n].tipo == PULSO) {
            pinSalida[n].activado = false;
            digitalWrite (salida[n],PIN_OFF);
        }
    }
}

void configuraSalida() {
    desactivaPulsos();                         // desconecta las salidas pulso por seguridad
    Serial.print (F("\nIntroduzca salida a configurar entre 0 y "));
    Serial.println (MAX_PIN - 1);
    selPin = entradaSerie (MAX_PIN - 1);
    Serial.println (F("\nConfiguracion actual:"));
    printSalida (selPin, true);                // Imprimimos configuracion pin seleccionado
    Serial.println (F("\nIntroduzca Direccion DCC (1-2044, 0: Salir)"));
    selDireccion = entradaSerie(2044);
    if (selDireccion == 0) {
        Serial.println (F("\nCambios cancelados"));
        return;
    }
    Serial.print (F("Nueva direccion: "));
    Serial.println (selDireccion);
    Serial.println (F("\nIntroduzca tipo de salida (0:FIJA, 1:FLASH, 2:SERVO, 3:PULSO)"));
    selTipo = entradaSerie(3);
    switch (selTipo) {
        case FIJA:
            configuraFIJA();
            break;
        case FLASH:
            configuraFLASH();
            break;
        case PULSO:
            configuraPULSO();
            break;
        case SERVO:
            configuraSERVO();
            break;
    }
    pinSalida[selPin].tipo = selTipo;
    pinSalida[selPin].direccionDCC = selDireccion;
    pinSalida[selPin].valorA = selValorA;
    pinSalida[selPin].valorB = selValorB;
    pinSalida[selPin].valorC = selValorC;
    Serial.println (F("\nNueva configuracion:"));
    printSalida (selPin, false);                // Imprimimos configuracion pin seleccionado
    Serial.println (F("\nCorrecto? (0:NO, 1:SI)"));
    if (entradaSerie(1)) {
        escribeConfigSalida (selPin);           // correcto: guardar en EEPROM
        leeConfiguracion ();                  // lee configuracion de todos los pines
        Serial.println (F("\nCambios guardados"));
    }
    else {
        leeConfigSalida(selPin);              // incorrecto: leer configuracion desde EEPROM
        Serial.println (F("\nCambios cancelados"));
    }
}

void entradaPar (int tipoSalida) {
    Serial.print (F("\nIntroduzca posicion de activacion de la salida "));
    switch (tipoSalida) {
        case FIJA:
            Serial.print (F("FIJA"));
            break;
        case FLASH:
            Serial.print (F("FLASH"));
            break;
        case PULSO:
            Serial.print (F("PULSO"));
            break;
    }
    Serial.println (F(" (0:ROJO, 1:VERDE)"));
    selValorA = entradaSerie(1);
    Serial.print (F("Nueva posicion: "));
    if (selValorA == ROJO)
        Serial.println (F("ROJO"));
}

```

```

    else
        Serial.println (F("VERDE"));
}

void configuraFIJA() {
    entradaPar (FIJA);
    selValorB = 0;
    selValorC = 0;
}

void configuraFLASH() {
    entradaPar (FLASH);
    Serial.println (F("\nIntroduzca Fase (0:FASE A, 1:FASE B)"));
    selValorB = entradaSerie(1);
    Serial.print (F("Nueva fase: "));
    if (selValorA == FASE_A)
        Serial.println (F("FASE A"));
    else
        Serial.println (F("FASE B"));
    selValorC = 0;
}

void configuraPULSO() {
    entradaPar (PULSO);
    Serial.print (F("\nIntroduzca duracion pulso en ms (1-"));
    int valor = 255 * MULT_PULSO;
    Serial.print (valor);
    Serial.println (F("")));
    selValorB = entradaSerie(valor) / MULT_PULSO;
    selValorC = 0;
}

void configuraSERVO() {
    Serial.println (F("\nIntroduzca angulo posicion ROJO (0-180) "));
    selValorA = entradaSerie(180);
    Serial.print (F("\nNuevo angulo (ROJO): "));
    Serial.println (selValorA);
    Serial.println (F("\nIntroduzca angulo posicion VERDE (0-180) "));
    selValorB = entradaSerie(180);
    Serial.print (F("\nNuevo angulo (VERDE): "));
    Serial.println (selValorB);
    Serial.println (F("\nIntroduzca velocidad (0-255) Usual: 5..50ms por grado"));
    selValorC = entradaSerie(255);
}

```

Ejemplos

Semáforo rojo/verde. La dirección 6 controla las luces de un semáforo de dos aspectos en las salidas 10 y 11 (LEDs a 5V o bien añadir un ULN2803 para otras tensiones):

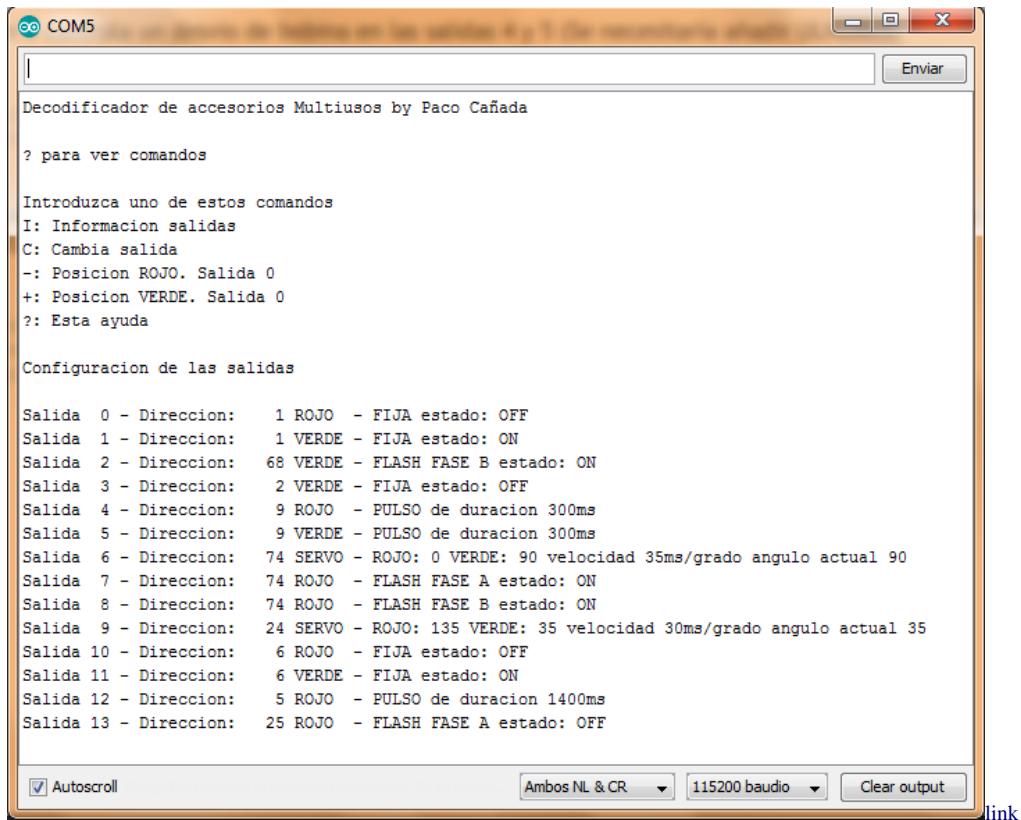
```
Salida 10 - Direccion: 6 ROJO - FIJA estado: ON
Salida 11 - Direccion: 6 VERDE - FIJA estado: OFF
```

Desvío. La dirección 9 controla un desvío de bobina en las salidas 4 y 5 (Se necesitaría añadir ULN2803)

```
Salida 4 - Direccion: 9 ROJO - PULSO de duracion 300ms
Salida 5 - Direccion: 9 VERDE - PULSO de duracion 300ms
```

Paso a nivel. La dirección 74 ROJO activa las salidas 7 y 8 parpadeantes en diferente fase y baja el servo de la salida 6 a 0 grados. La 74 VERDE apaga las salidas parpadeantes y sube el servo a 90 grados:

```
Salida 6 - Direccion: 74 SERVO - ROJO: 0 VERDE: 90 velocidad 35ms/grado angulo actual 90
Salida 7 - Direccion: 74 ROJO - FLASH FASE A estado: OFF
Salida 8 - Direccion: 74 ROJO - FLASH FASE B estado: OFF
```

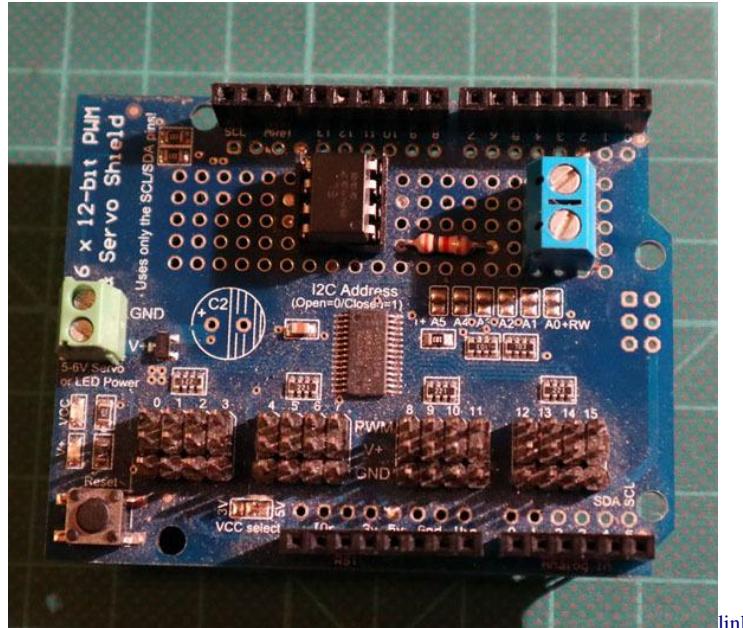


NOTA: La salida 13 es la que tiene el LED de la placa, al resetear parpadea por la comunicación con el *bootloader* por lo que es mejor no usarla para servos o bobinas así se evitan movimientos indeseados.

Hasta el infinito y más allá!

Buzz Lightyear (Toy Story, 1995)

Con esta shield podemos tener 15 salidas más 16 servos por I2C ya que incorpora el chip PCA9685 y tiene una zona de pruebas para poder montar el circuito del optoacoplador para la señal DCC.



[link](#)

Necesitaremos añadir la librería [Wire.h](#) para trabajar con I2C y la [Adafruit_PWM_ServoDriver.h](#) para esta shield.

<https://github.com/adafruit/Adafruit-PWM-Servo-Driver-Library>

Habría que adaptar las escrituras a las salidas para que si es una de las controladas por el chip PCA9685 se haga a través del I2C pero valdría prácticamente todo el código aquí descrito, así que si alguien quiere intentarlo, lo dejo como ejercicio 😊

16. Interrupciones: Mando XpressNet para panel

Algunos tienen un panel de control óptico (TCO) de la maqueta o de la estación desde donde controlan los desvíos, puede ser interesante tener también un mando Xpressnet en el propio panel para controlar algún tren cuando haga maniobras y dejar el Multimaus y el LH101 para el resto de la maqueta.



La idea

Algo simple parecido al pequeño LH01 de Lenz, en el que pueda controlar la velocidad, las funciones y se pueda elegir la locomotora en un pequeño display.

Para la velocidad usaremos un encoder rotativo con pulsador en el mismo mando en lugar de un potenciómetro, así no tendremos el indeseado efecto de cambio de velocidad si no corresponde con la que lleva la maquina cuando la seleccionemos.

Haremos como el LH01 de Lenz, en la pantalla mostraremos la dirección de la locomotora o cuando la controlemos pondremos el paso de velocidad y el sentido. Girando el mando variaremos la velocidad, si lo pulsamos estando parada le cambiamos el sentido, si esta en marcha la frenaremos poniendo la velocidad a cero.

Si no encontramos encoder rotativo con pulsador podemos ponerlo aparte, también necesitaremos otro pulsador para acceder a un menú para cambiar de locomotora y elegir funciones.

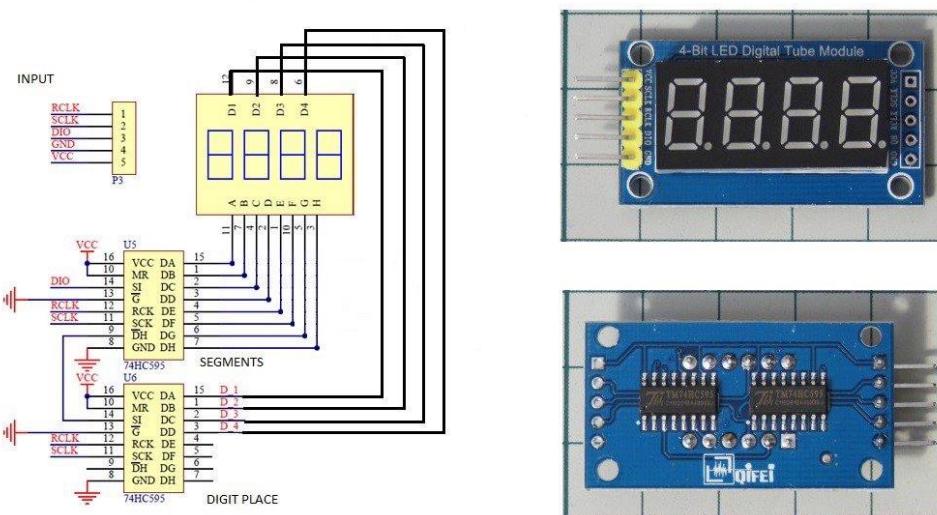
Pulsando el botón entraremos o saldremos del menú. El menú será simple, se podrá seleccionar con el mando entre elegir dirección de locomotora o cambiar estado de una función, al pulsarlo se entrará en esa opción.

Si elegimos función, girando el mando seleccionamos un numero entre 0 y 28, al pulsarlo activaremos o desactivaremos la función (se mostrará el estado de la función).

Si elegimos locomotora, girando el mando elegimos un dígito, al pulsarlo pasaremos al siguiente dígito. El dígito se señalará encendiéndose el punto decimal, al pulsar el botón se saldrá del menú y se podrá controlar la nueva locomotora.

Eligiendo...

Necesitaremos un display de 4 bits 4 dígitos 7 segmentos controlado por unos chips 74HC595.



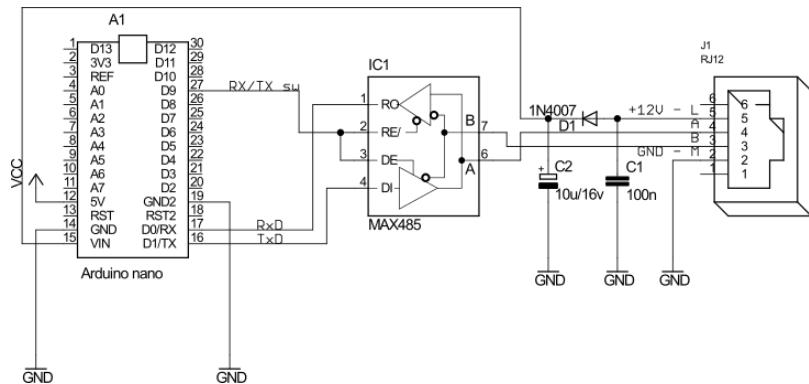
[link](#)

Para la conexión Xpressnet (LMAB) necesitaremos un MAX485 que cablearemos adecuadamente o lo tendremos en un modulo:



[link](#)

La alimentación la tomaremos del bus Xpressnet directamente alimentando al Arduino por el pin Vin a través de un diodo:

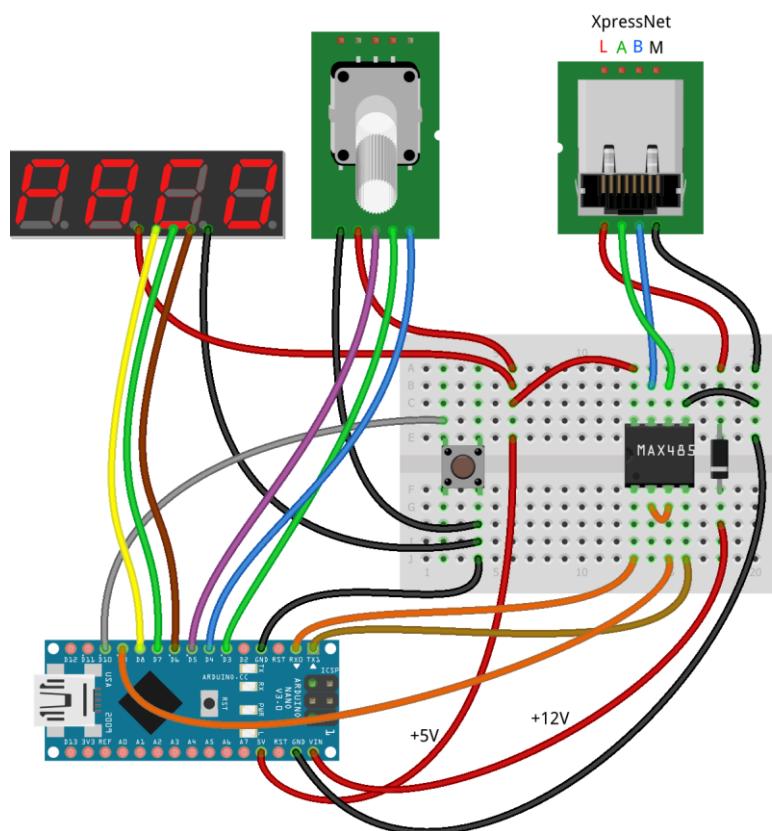


[link](#)

Para comunicarnos con la central usaremos la librería **XpressNet.h** como esclavo que instalaremos desde Programa-> Añadir librería ZIP

<https://sourceforge.net/projects/pgahtow/files/Arduino%20%28v1.0%29%20libraries/XpressNet.zip/download>

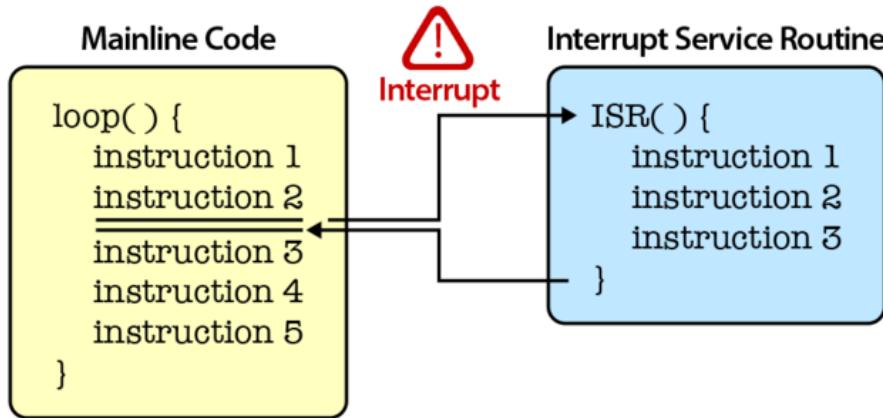
Lo conectaremos así:



fritzing [link](#)

Interrumpiendo...

En este proyecto vamos a ver el uso de interrupciones, cuando se produce una interrupción el programa que se ejecuta se detiene y pasa a ejecutarse la función de la interrupción, una vez que esta finaliza vuelve al punto donde se interrumpió.



[link](#)

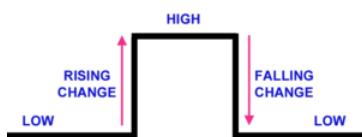
Las funciones de servicio de interrupción (ISR) son especiales, no tienen parámetros y no devuelven nada. Generalmente tienen que ser tan cortas y rápidas como sea posible.

Dentro de una interrupción no se pueden utilizar `delay()` y el valor devuelto por `millis()` no se incrementará, la recepción de datos serie podría perder datos si se tarda mucho y si se comparten variables con el código principal han de ser globales y se han de declarar como `volatile`.

Ha de tenerse en cuenta que podría darse el caso de que estemos leyendo o escribiendo una variable de varios bytes (`int`, `long`, array...) compartida con la interrupción y esta saltase justo en medio de la lectura y leyera o cambiase el valor, por lo que seguramente tendríamos un dato erróneo. En estos casos hay que utilizar `nolimits()` para evitar las interrupciones antes de modificar/leer esos datos y una vez realizado usar `interrupts()` para volverlas a activar. (hay quien utiliza `sei()` y `cli()` para activar/desactivar interrupciones)

Hay interrupciones que se ejecutan cuando cambia un pin, otras cuando algún Timer llega al final de la cuenta, cuando se finaliza el envío/recepción por el puerto serie, etc.

Las relacionadas con un pin pueden activarse cuando la entrada cambia (**CHANGE**), lo hace de HIGH a LOW (**FALLING**), de LOW a HIGH (**RISING**) o por nivel **LOW**



[link](#)

En un Arduino Uno/Nano estas interrupciones están disponibles en los pines D2 y D3, en el Arduino Mega en D2, D3, D18, D19, D20 Y D21. Se activan con `attachInterrupt()` donde se le pasan como parámetros la interrupción, la función a ejecutar en la interrupción y el modo (**CHANGE**, **FALLING**, **RISING**, **LOW**). Para averiguar la interrupción que corresponde a cada pin se usa la función `digitalPinToInterrupt()`. Por ejemplo, para ejecutar nuestra función `funcionISR()` cada vez que cambie el pin D2:

```
void funcionISR () {
// El código a ejecutar en la interrupción
}

attachInterrupt (digitalPinToInterrupt(2), funcionISR, CHANGE);
```

Hay otra forma de usarlas, que es usando ISR, pero es algo mas liado ya que depende del tipo de interrupción (hay 25), hay que saber con que vector se corresponde y probablemente tengamos que configurar registros internos del micro, por ejemplo:

```
ISR (PCINT0_vect)
{
// uno de los pines D8 a D13 ha cambiado
}
```

Una de las librerías que trabaja con interrupciones es la [TimerOne.h](#) que usa el Timer1 (el mismo que [Servo.h](#) por lo que no es compatible) accesible a través del [Gestor de Librerías](#) que nos permite ejecutar una función periódicamente (hasta 8388480 microsegundos) reprogramando el Timer1.

<https://github.com/PaulStoffregen/TimerOne>

Por ejemplo para que ejecutemos la función cada 250ms:

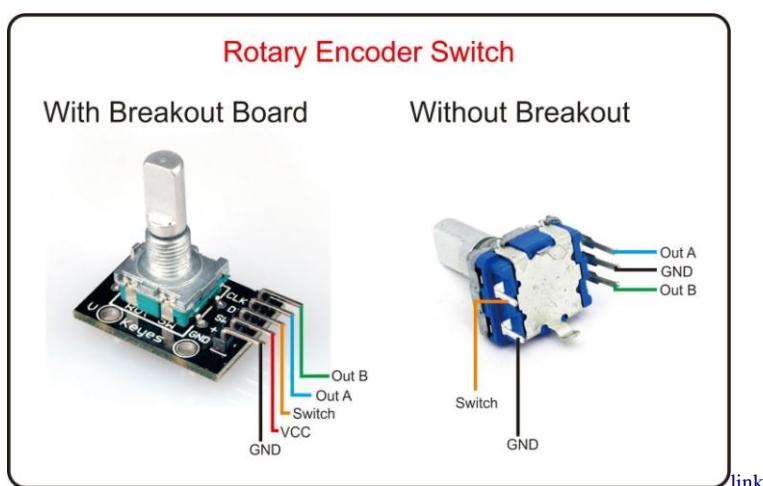
```
void funcionISR () {
// El código a ejecutar en la interrupción
}

Timer1.initialize(250000);           // Dispara cada 250 ms
Timer1.attachInterrupt(funcionISR); // Activa la interrupcion
```

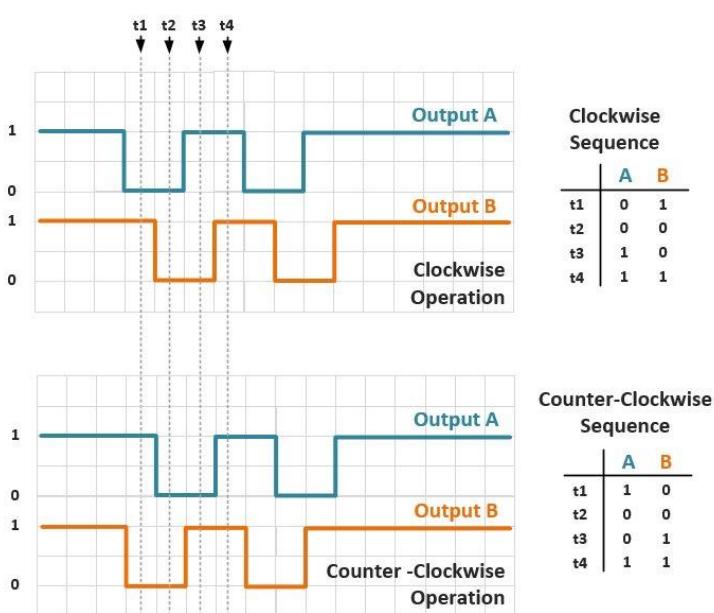
Hay otras librerías que también trabajan con interrupciones, por ejemplo [Servo.h](#), la de decodificación de la señal DCC ([NmraDcc.h](#)) y la de comunicación XpressNet ([Xpressnet.h](#)).

Controlando

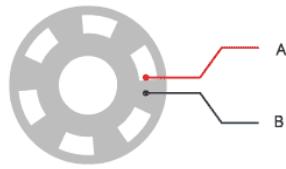
Para la velocidad y selección de datos vamos a usar un encoder rotativo



Para un encoder cuando gira tenemos esta salida en sus dos pines, si nos fijamos cuando una señal (Output A) cambia la otra (Output B) estará al mismo nivel o al contrario dependiendo de si se gira en sentido horario (CW) o antihorario (CCW).

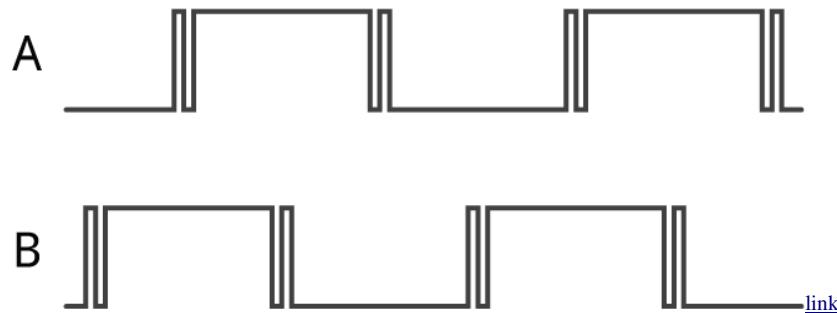


[link](#)



[link ani](#)

Uno de los problemas que se presentan es que las señales no son tan bonitas como las pintan sino que en realidad cuando cambia de estado se producen una serie de rebotes que hacen que la señal sea parecida a esta:



[link](#)

Vamos a leerlo usando interrupciones así no tendremos que ir leyéndolo constantemente, para ello uno de los pines ha de estar conectado a un pin de interrupción (D2 o D3), así cuando se mueva saltará la interrupción y haremos que nos actualice una variable entre 0 y el valor que le digamos ([encoderMax](#)), para evitar problemas haremos que el tipo sea **byte**. Además activaremos una bandera (*flag*) para indicar que el valor ha cambiado ([encoderCambio](#)).

En la interrupción tendremos que comprobar el estado de los pines y compararlos con el estado en la anterior interrupción para ver si realmente ha cambiado o es un rebote, ya que una señal permanece estable cuando la otra cambia podremos comprobar cuando ha habido un cambio.

Después miraremos si las entradas están al mismo nivel o al contrario para incrementar o decrementar el valor actual.

```
const int pinOutA = 3;           // pines del encoder
const int pinOutB = 4;
const int pinSwitch = 5;
const int pinPulsador = 10;      // pin del pulsador

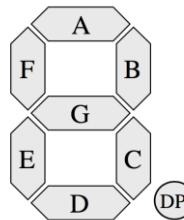
int outA, outB, copiaOutA, copiaOutB; // valor de las entradas del encoder que usados por ISR
volatile byte encoderValor;          // estos valores del encoder son compartidos por ISR y programa
volatile byte encoderMax;
volatile bool encoderCambio;

// en setup
pinMode (pinOutA, INPUT_PULLUP);    // entradas
pinMode (pinOutB, INPUT_PULLUP);
pinMode (pinSwitch, INPUT_PULLUP);
pinMode (pinPulsador, INPUT_PULLUP);
copiaOutA = digitalRead (pinOutA);
copiaOutB = digitalRead (pinOutB);
attachInterrupt (digitalPinToInterrupt (pinOutA), encoderISR, CHANGE);

// la interrupcion
void encoderISR () {               // interrupcion encoder
    outA = digitalRead (pinOutA);
    outB = digitalRead (pinOutB);
    if (outA != copiaOutA) {        // evitamos rebotes
        copiaOutA = outA;
        if (outB != copiaOutB) {
            copiaOutB = outB;
            if (outA == outB)        // comprueba sentido de giro
                encoderValor = (encoderValor >= encoderMax) ? encoderMax : ++encoderValor; // CW, hasta maximo
            else
                encoderValor = (encoderValor <= 0) ? 0 : --encoderValor; // CCW, hasta 0
            encoderCambio = true;
        }
    }
}
```

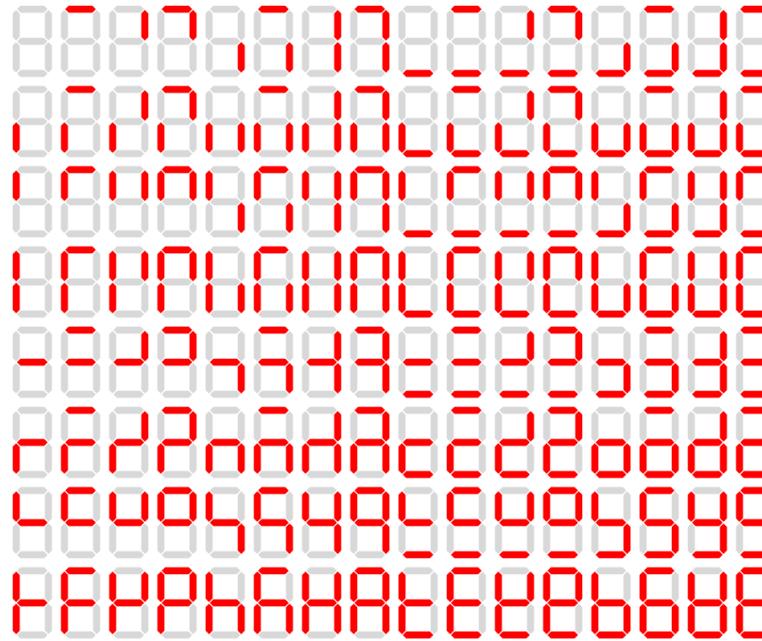
Para controlar la pantalla solo necesitaremos tres pines (Data, Clock y Latch) ya que lleva dos registros de desplazamiento 74HC595 que nos conectan uno los dígitos y el otro los segmentos.

Para encender los segmentos tenemos que poner Latch a **LOW** y enviarle con **shiftOut()** un dato con un patrón de bits a encender (a=bit0, b=bit1,..., g=bit6, DP=bit7)



[link](#)

Estas son todas las combinaciones que se pueden conseguir:



[link](#)

Necesitamos definir los patrones de los números y letras, queremos escribir además de números: OFF, StoP, Pro para el estado de a central y Loco, Func para el menu y ', para la mostrar el sentido. Observad la **B** de los valores en binario.

```
static const byte patron[] =           // 7 segmentos valores para 0..9, espacio, F,L,t,o,P,r,c,u,n,°
{
//  patron caracter |   0 = segmento on
//  dpgFEDCBA |   1 = segmento off
B11000000, //0 |       A
B11111001, //1 |       -----
B10100100, //2 |       F |       | B
B10110000, //3 |       |       G | 
B10011001, //4 |       -----|
B10010010, //5 |       E |       | C
B10000010, //6 |       |       | 
B11111000, //7 |       ----- dp
B10000000, //8 |       D
B10011000, //9 |
B11111111, // espacio
B10001110, // F
B11000111, // L -
B10000111, // t
B10100011, // o -
B10001100, // P
B10101111, // r -
B10100111, // c -
B11100011, // u -
B10101011, // n -
B11011111, // ' -
B11101111, // , -
B10011100 // ° -
```

```

#define CH_ESPACIO 10           // Posicion de los caracteres en la tabla de patrones
#define CH_F 11
#define CH_L 12
#define CH_t 13
#define CH_o 14
#define CH_P 15
#define CH_r 16
#define CH_c 17
#define CH_u 18
#define CH_n 19
#define CH_ADELANTE 20
#define CH_ATRAS 21
#define CH_BOLA 22

```

Con otro **shiftOut()** tenemos que enviar el bit del dígito que queremos encender y apagar los otros ya que todos reciben la misma combinación a,b,c..g,DP, finalmente volver a poner Latch a **HIGH**.

Para encender los cuatro, cada uno con su dígito, tenemos que ir enviando los patrones a cada uno e ir repitiéndolos lo suficientemente rápido para que la persistencia del ojo vea los cuatro encendidos. Es el mismo principio que el cine (24 fotogramas por segundo) o la televisión.

Como tenemos 4 dígitos y los encendemos 40 veces por segundo para que no parpadeen así que necesitamos cambiar de dígito cada 6,25ms. Podríamos hacer la temporización en el **loop()** pero como estamos viendo interrupciones vamos a usar una para hacerlo.

Usaremos la librería **TimerOne.h** que nos ejecutará la función exactamente cada 6,25ms. Como la interrupción si usásemos **shiftOut()** dentro de ella sería muy lenta ya que ha de enviar 16 bits de datos y en ese tiempo se podría perder algún dato serie (XpressNet va a 62500 baudios, 160us por byte) sólo activaremos una bandera para indicar que ya se puede mostrar el dígito, también llevaremos la cuenta del dígito.

```

const int pinData = 6;           // DIO pines pantalla 7 segmentos chip 74HC595
const int pinLatch = 7;          // RCLK
const int pinClock = 8;          // SCLK
byte digitos[4];               // patrones a mostrar en cada digito.
uint8_t enciendeDP = -1;         // digito a mostrar con el punto decimal encendido
volatile bool actualizaPantalla; // Compartido con interrupcion
volatile byte contadorISR = 0;   // contador de la interrupcion

// En el setup()
pinMode (pinData,OUTPUT);
pinMode (pinLatch,OUTPUT);
pinMode (pinClock,OUTPUT);
Timer1.initialize(6250);         // Dispara cada 6.25 ms (40 veces/segundo por digito)
Timer1.attachInterrupt(pantallaISR); // Activa la interrupcion

// La interrupcion
void pantallaISR() {            // interrupcion pantalla
    contadorISR++;
    if (contadorISR == 4)        // cuenta de 0 a 3
        contadorISR = 0;          // empieza con los miles, luego centenas, decenas y unidades
    actualizaPantalla = true;
}

```

En el **loop()** si tenemos que actualizar la pantalla enviamos los datos de los patrones con **shiftOut()** y el dígito a activar. Además activaremos el punto decimal si coincide con el dígito a mostrar.

```

void enviaPantalla() {
    int segmentos, caracter;
    byte contador;
    actualizaPantalla = false;
    contador = contadorISR;           // lo copiamos por si cambia ya que es compartido con ISR
    caracter = digitos[contador];     // caracter a mostrar en el digito
    segmentos = patron[caracter];     // patron de segmentos segun el caracter.
    if (enciendeDP == contador)       // si este digito necesita el punto decimal se enciende
        bitClear (segmentos, 7);
    digitalWrite (pinLatch,LOW);       // inicia transmision
    shiftOut (pinData, pinClock, MSBFIRST, segmentos); // patron segmentos
    shiftOut (pinData, pinClock, MSBFIRST, 1 << (contador)); // activa digito
    digitalWrite (pinLatch, HIGH);
}

```

Para mostrar números los mostraremos alineados a la derecha según la cantidad de cifras que necesitemos.

```
#define UNIDADES 0          // posicion en el array digitos[]
#define DECENAS 1
#define CENTENAS 2
#define MILES 3

void mostrarNumero (int valor, int cifras) {
    digitos[UNIDADES] = valor % 10;      // guardamos el resto de la division en el array
    if (cifras >1)
        digitos[DECENAS] = (valor / 10) % 10;
    if (cifras >2)
        digitos[CENTENAS] = (valor / 100) % 10;
    if (cifras >3)
        digitos[MILES] = (valor / 1000) % 10;
}
```

Definiremos si nosotros estamos en modo normal, menú o error para mostrar los mensajes correspondientes a cada modo en la pantalla

```
enum modos {NORMAL, MENU_SEL, MENU_LOCO, MENU_FUNC, ERROR_XN}; // modo en el que estoy
modos miModo;
```

XpressNet

Vamos a ver una nueva estructura de datos similar a **struct** que es **union**, a diferencia de **struct** en que una variable se almacena a continuación de otra, en **union** todas ocupan el mismo espacio.

De esta forma es posible cargar un valor en una variable y leerlo por partes con otra. Por ejemplo:

```
union datos {
    int entero; // 16 bits
    byte parte[2]; // 8 bits x 2
} Valor;

Valor.entero = 12345; // 16 bits
byte alto = Valor.parte[1]; // byte alto (8bits)
byte bajo = Valor.parte[0]; // byte bajo (8bits)
```

La conexión Xpressnet requiere definir la dirección en el bus Xpressnet (entre 1 y 31) y el pin de control de transmisión en su función **start()**. Usaremos la dirección 25 que suele estar desocupada.

Definimos los datos la locomotora que controlamos, inicializamos la librería y le pedimos a la central el estado en que se encuentra con **getPower()**. En esta librería el estado de las funciones F0 a F28 nos lo pasará en cuatro bytes así que las agruparemos con un **long** en un **union** para manejarlas.

He comprobado que es mejor pedir los datos a la central a través de la librería una vez ya has recibido la respuesta, así se evitan problemas con las temporizaciones. Así que definiremos unas banderas (**flag**) para cuando queramos hacer peticiones a la central (estado de la locomotora y sus funciones)

```
int miLocomotora; // Datos de la locomotora que estoy controlando
uint8_t miVelocidad;
uint8_t miSentido;
uint8_t miPasos;
union funciones {
    uint8_t Xpress[4]; // array para funciones, F0F4, F5F12, F13F20 y F21F28
    unsigned long Bits; // long para acceder a los bits
} miFuncion;

const int pinTXRX = 9; // pin de control del MAX485
const int miDireccionXpressnet = 25; // Direccion del dispositivo en el bus Xpressnet

XpressNetClass XpressNet;

bool pideInfoLoco; // pedir informacion de la locomotora
bool pideInfoFunc; // pedir indormacion del estado de las funciones
bool pedirFuncPendiente; // falta pedir informacion del estado de las funciones
```

```

// en setup()
miLocomotora = 3;                                // Inicialmente controlamos la locomotora 3 a 28 pasos
miPasos = PASOS28;
encoderMax = 28;
encoderValor = 0;
pideInfoLoco = false;
pideInfoFunc = false;
pedirFuncPendiente = false;
miModo = ERROR_XN;                                // Mostramos '0003' hasta que la central responda
mostrarNumero (miLocomotora, 4);
XpressNet.start (miDireccionXpressnet, pinTXRX); // inicializamos libreria
XpressNet.getPower();                             // Pedimos el estado actual de la central

```

En el **loop()** hay que llamar constantemente a la librería:

```
XpressNet.receive();
```

Paso a paso

Antes de hacer mas cosas tenemos que preparar algunas funciones para que nos faciliten el trabajo con esta librería. Una es saber la velocidad máxima que podremos dar con el encoder que dependerá de los pasos en que trabaje la locomotora.

```

#define PASOS14 0                      // pasos segun los valores de la libreria
#define PASOS28 2
#define PASOS128 3

byte velocidadMaxima (uint8_t pasos) {
    switch (pasos) {                // devuelve el numero de paso mas alto para el encoder
        case PASOS14:
            return (14);
            break;
        case PASOS28:
            return (28);
            break;
        case PASOS128:
            return (126);
            break;
    }
}

```

Cuando usemos direcciones largas (dirección de locomotora de 100 hasta 9999) tenemos que escribir el byte alto de la dirección con el valor adecuado (como hacemos con la CV17) así que escribimos una función que nos lo calcule.

```

uint8_t highByteCV17 (int direccion) {
    if (direccion > 99)           // si es direccion larga coloca bits altos a uno
        direccion |= 0xC000;
    return (highByte(direccion));
}

```

Para controlar la locomotora necesitamos pedirle a la central, si se ha activado el flag, que nos diga su velocidad actual, sentido, pasos y funciones F0 a F12 con **getLocoInfo()** y activaremos un flag para indicar que aun tenemos que pedir la información de las funciones (F13 a F28). Nos lo notificará luego con la función **notifyLokAll()**

```

void pideInformacionLocomotora () {      // informacion de locomotora.
    pideInfoLoco = false;
    pedirFuncPendiente = true;
    XpressNet.getLocoInfo (highByteCV17(miLocomotora), lowByte(miLocomotora));
}

```

La información del estado de las funciones de la locomotora cuando se active el flag lo pediremos con **getLocoFunc()**

```

void pideInformacionFunciones () {      // informacion de funciones.
    pideInfoFunc = false;
    XpressNet.getLocoFunc (highByteCV17(miLocomotora), lowByte(miLocomotora));
}

```

Otra es enviar a la central la nueva velocidad con `setLocoDrive()` teniendo en cuenta los pasos con los que trabajamos y poniendo en el sitio correcto los valores como requiere la función `setLocoDrive()` de la librería.

```
void nuevaVelocidad(uint8_t valor) {      // enviar nueva velocidad como requiere libreria
    int velocidad;
    miVelocidad = valor;
    switch (miPasos) {                      // nos saltamos el paso de parada de emergencia
        case PASOS14:
        case PASOS128:
            if (valor>0)
                velocidad = valor + 1;
            else
                velocidad = 0;
            break;
        case PASOS28:                         // ponemos los 28 pasos en su formato
            if (valor > 0)                   // pasamos velocidad entre 0 y 31 (0 0 0 S4 S3 S2 S1 S0)
                velocidad = valor + 3;       // nos saltamos pasos parada de emergencia
            else
                velocidad = 0;
            bitWrite (velocidad, 5, bitRead (velocidad,0)); // copia bit 0          (0 0 S0 S4 S3 S2 S1 S0)
            velocidad >>= 1;                 // coloca en su sitio        (0 0 0 S0 S4 S3 S2 S1)
            break;
    }
    if (miSentido)                         // añade el sentido
        velocidad |= 0x80;
    XpressNet.setLocoDrive (highByteCV17(miLocomotora), lowByte(miLocomotora), miPasos, velocidad);
}
```

Para las funciones como solo tenemos que cambiarlas al pulsar, lo haremos a través de la función `setLocoFunc()`.

```
#define FUNC_OFF 0                     // control de funciones según los valores de la librería
#define FUNC_ON 1
#define FUNC_CHANGE 2

void cambiaFuncion (int numFuncion) { // activamos o desactivamos la función segun su estado
    bool estado = bitRead (miFuncion.Bits, numFuncion);
    if (estado) {
        XpressNet.setLocoFunc (highByteCV17(miLocomotora), lowByte(miLocomotora), FUNC_OFF, numFuncion);
        bitClear (miFuncion.Bits, numFuncion);
    }
    else {
        XpressNet.setLocoFunc (highByteCV17(miLocomotora), lowByte(miLocomotora), FUNC_ON, numFuncion);
        bitSet (miFuncion.Bits, numFuncion);
    }
}
```

Otra función que nos interesa es actualizar la pantalla con la nueva información si hemos hecho un cambio (velocidad, función, modo, etc...) escribiendo los caracteres en el array de pantalla dependiendo del modo mostraremos una cosa u otra, también actualizaremos el valor máximo del encoder y su valor.

```
uint8_t pantallaMenu = 0;           // menu seleccionado: 0:Locomotora, 1:Funciones
uint8_t pantallaDigito = 0;         // digito seleccionado en seleccion de locomotora
uint8_t pantallaFuncion = 0;        // valor de la función seleccionada

void nuevaPantalla() {              // mostrar datos en pantalla segun mi modo
    switch (miModo) {
        case NORMAL:                  // mostrar sentido y velocidad
            mostrarNumero (miVelocidad,3);
            if (miSentido)
                digitos[MILES] = CH_ADELANTE;
            else
                digitos[MILES] = CH_ATRAS;
            encoderMax = velocidadMaxima(miPasos);
            encoderValor = miVelocidad;
            break;
        case MENU_SEL:                // mostrar 'Loco' o 'Func'
            if (pantallaMenu == MENU_FUNC) { // menu seleccionado: Locomotora o Funciones
                digitos[MILES]=CH_F;
                digitos[CENTENAS]=CH_u;
                digitos[DECENAS]=CH_n;
                digitos[UNIDADES]=CH_c;
            }
            else {
                digitos[MILES]=CH_L;
                digitos[CENTENAS]=CH_o;
            }
    }
}
```

```

        digitos[DECENAS]=CH_c;
        digitos[UNIDADES]=CH_o;
    }
    encoderMax = 1;
    encoderValor = (pantallaMenu == MENU_FUNC) ? 1 : 0;
    break;
case MENU_LOCO:                                // mostrar numero de locomotora y punto decimal
    mostrarNumero (miLocomotora,4);
    enciendeDP = pantallaDigito;                // Enciende el punto decimal
    encoderMax = 9;
    encoderValor = digitos[pantallaDigito];   // en digitos[] ya esta el valor calculado
    break;
case MENU_FUNC:                                // mostrar funcion y estado
    mostrarNumero (pantallaFuncion,2);
    digitos[CENTENAS] = CH_F;
    if (bitRead (miFuncion.Bits, pantallaFuncion)) // vemos si la funcion esta activa
        digitos[MILES] = CH_BOLA;               // mostrar funcion activa
    else
        digitos[MILES] = CH_ESPACIO;          // mostrar funcion apagada
    encoderMax = 28;
    encoderValor = pantallaFuncion;
    break;
case ERROR_XN:                                 // dejar la pantalla igual
    break;
}
}

```

Notificando

Una de las notificaciones que recibimos es el estado de la central mediante `NotifyXNetPower()`, pondremos en pantalla la información correspondiente, así que escribimos las letras en el array `digitos[]`, si entramos en modo normal pedimos información de nuestra locomotora por si ha cambiado.

```

uint8_t estadoCentral;                      // Estado de la central

void notifyXNetPower (uint8_t State) { // La libreria llama a esta funcion cuando cambia estado de
    la central
    estadoCentral = State;
    enciendeDP = -1; // Apagamos punto decimal
    switch (estadoCentral) {
        case csNormal:                         // Al entrar en modo normal mostramos la direccion de
nuestra locomotora
            miModo = NORMAL;                  // pedimos informacion de la locomotora por si ha cambiado
            mostrarNumero (miLocomotora,4);
            encoderMax = velocidadMaxima (miPasos);
            encoderValor = miVelocidad;
            pideInfoLoco = true;
            break;
        case csShortCircuit:                  // Corto circuito - OFF
        case csTrackVoltageOff:             // Sin tension en via - OFF
            digitos[MILES]= 0;
            digitos[CENTENAS]= CH_F;
            digitos[DECENAS]= CH_F;
            digitos[UNIDADES]= CH_ESPACIO;
            miModo = ERROR_XN;
            break;
        case csEmergencyStop:               // Parada emergencia - StoP
            digitos[MILES]= 5;
            digitos[CENTENAS]= CH_t;
            digitos[DECENAS]= CH_o;
            digitos[UNIDADES]= CH_P;
            miModo = ERROR_XN;
            break;
        case csServiceMode:                 // Programacion en modo servicio - Pro
            digitos[MILES]= CH_P;
            digitos[CENTENAS]= CH_r;
            digitos[DECENAS]= CH_o;
            digitos[UNIDADES]= CH_ESPACIO;
            miModo = ERROR_XN;
            break;
    }
}

```

Otra notificación es `notifyLokAll()` con el estado actual de una locomotora, así que actualizamos nuestros datos si son de la locomotora que estamos controlando.

En el byte alto de la dirección borramos los bits correspondientes a direcciones largas para hacer la comparación.

Calcularemos la velocidad actual para que vaya entre 0 y la máxima según los pasos tal como la usaremos con el encoder.

Observad que al convertir la velocidad nos puede dar un valor negativo que tendremos que corregir como cero, los valores negativos en datos tipo `byte` tienen el bit 7 activado (valores 128 a 255) así que si es el caso pondremos la velocidad como cero.

Si tenemos activado el flag de pedir información de funciones aún pendiente (`pedirFuncPendiente`) activaremos el flag de pedir estado de funciones (`pideInfoFunc`) que se nos notificarán correctamente en la próxima llamada a esta función.

En caso contrario actualizamos los datos de las funciones rellenando el array en el `union` recolocando los bits en el orden adecuado a través de `long`

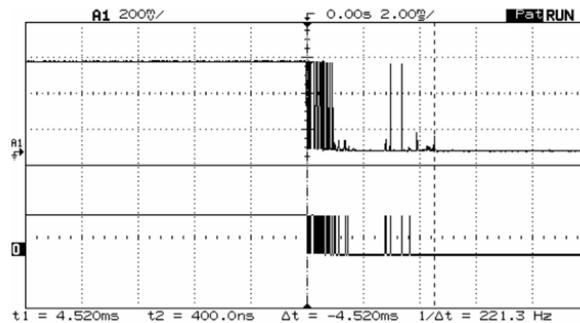
```
void notifyLokAll(uint8_t Adr_High, uint8_t Adr_Low, boolean Busy, uint8_t Steps, uint8_t Speed,
                  uint8_t Direction, uint8_t F0, uint8_t F1, uint8_t F2, uint8_t F3, boolean Req) {
    int Loco = ((Adr_High << 8) & 0x3F) + (Adr_Low);
    if (Loco != miLocomotora) // si no es nuestra locomotora no hace nada
        return;
    miSentido = Direction; // sentido
    miPasos = Steps; // pasos
    switch (Steps) { // ajusta mi Velocidad segun los pasos
        case PASOS14:
        case PASOS128:
            miVelocidad = Speed - 1; // eliminamos paso de stop de emergencia
            break;
        case PASOS28:
            Speed <<=1; // Mover bits a su sitio (0 0 0 S0 S4 S3 S2 S1)
            bitWrite (Speed, 0, bitRead(Speed,5)); // Hacemos sitio para el bit (0 0 S0 S4 S3 S2 S1 0 )
            Speed &= 0x1F; // colocamos bit (0 0 S0 S4 S3 S2 S1 S0)
            miVelocidad = Speed - 3; // Limpiamos (0 0 0 S4 S3 S2 S1 S0);
            break;
    }
    if (miVelocidad > 127) // eliminamos pasos de stop de emergencia
        miVelocidad = 0;
    if (pedirFuncPendiente) { // Comprobamos si falta pedir informacion de las funciones
        pedirFuncPendiente = false;
        pideInfoFunc = true;
    }
    else {
        miFuncion.Xpress[3] = F3; // F3: F28F27F26F25F24F23F22F21
        miFuncion.Xpress[2] = F2; // F2: F20F19F18F17F16F15F14F13
        miFuncion.Xpress[1] = F1; // F1: F12F11F10 F9 F8 F7 F6 F5
        // ponemos F0: x x x F0 F4 F3 F2 F1 asi F4 F3 F2 F1 F0 x x x para reordenarlos
        miFuncion.Xpress[0] = F0 <<4; // F4 F3 F2 F1 0 0 0 0
        if (bitRead(F0, 4))
            bitSet (miFuncion.Xpress[0], 3); // F4 F3 F2 F1 F0 0 0 0
        miFuncion.Bits = miFuncion.Bits >> 3; // desplazamos los 29 bits a su sitio: 000F28F27...F0
    }
    nuevaPantalla();
}
```

Esta notificación no es necesaria pero nos sirve para ver si hay actividad en el bus encendiéndo un LED, usaremos el de la placa. La constante `LED_BUILTIN` contiene el valor adecuado para la placa que usamos, en Arduino Uno/Nano es el pin D13.

```
void notifyXNetStatus (uint8_t State) { // Funcion de la libreria para mostrar estado en un LED
    digitalWrite(LED_BUILTIN, State); // usamos el LED del Arduino (Arduino Uno/Nano: pin 13)
}
```

Manejando

La interrupcion nos supervisa el encoder pero los pulsadores tenemos que leerlos aparte, haremos que se nos active una bandera cuando se haya apretado.



[link](#)

Al igual que con el encoder, cuando se pulsan se genera ruido hasta que hacen buen contacto (hasta 1 o 2ms después) con lo que podríamos tener lecturas erróneas, para eliminar estos ruidos se suele colocar un condensador de 100nF en paralelo con el pulsador.

También podemos comparar el estado del pulsador con un estado anterior y no darlo por bueno hasta que ha pasado un tiempo suficiente.

```
unsigned long tiempoBotones;      // para lectura de los botones
const unsigned long antiRebote = 50; // temporizador antirebote
int estadoPulsador;
int estadoSwitch;
bool pulsadorOn;
bool switchOn;

void leePulsadores () {
    int entradaBoton;
    if (millis() - tiempoBotones > antiRebote) { // lee cada cierto tiempo
        tiempoBotones = millis();
        entradaBoton = digitalRead (pinPulsador); // comprueba cambio en boton pulsador
        if (estadoPulsador != entradaBoton) {
            estadoPulsador = entradaBoton;
            if (estadoPulsador == LOW)
                pulsadorOn = true;
        }
        entradaBoton = digitalRead (pinSwitch); // comprueba cambio en boton del encoder
        if (estadoSwitch != entradaBoton) {
            estadoSwitch = entradaBoton;
            if (estadoSwitch == LOW)
                switchOn = true;
        }
    }
}
```

Ahora ya podemos escribir las funciones para cada uno de los controles según el modo en el que estemos actualizando luego la pantalla.

```
void controlPulsador () {                                // acciones cuando se pulsa el boton segun el modo
    pulsadorOn = false;
    switch (miModo) {
        case NORMAL:                                     // en modo normal pasa al menu de seleccion
            miModo = MENU_SEL;
            break;
        case MENU_LOCO:                                  // en menu loco pasa a normal,pide informacion de la nueva locomotora
            miModo = NORMAL;
            enciendeDP= -1;                               // apaga punto decimal
            pideInfoLoco = true;
            return;                                         // no muestra velocidad, espera a que llegue la informacion
            break;
        case MENU_SEL:                                    // en los otros menu pasa a normal
        case MENU_FUNC:
            miModo = NORMAL;
            break;
        case ERROR_XN:                                   // en error no hace nada
            break;
    }
    nuevaPantalla();                                    // actualiza datos en pantalla
}
```

```

void controlSwitch () {
    switchOn = false;
    switch (miModo) {
        case NORMAL:
            if (miVelocidad == 0)
                miSentido = ! miSentido;
            nuevaVelocidad (0);
            break;
        case MENU_SEL:
            if (pantallaMenu == MENU_FUNC)
                miModo = MENU_FUNC;
            else {
                miModo = MENU_LOCO;
                pantallaDigito = UNIDADES;
            }
            break;
        case MENU_LOCO:
            switch (pantallaDigito) {
                case UNIDADES:
                    pantallaDigito = DECENAS;
                    break;
                case DECENAS:
                    pantallaDigito = CENTENAS;
                    break;
                case CENTENAS:
                    pantallaDigito = MILES;
                    break;
                case MILES:
                    pantallaDigito = UNIDADES;
                    break;
            }
            break;
        case MENU_FUNC:
            cambiaFuncion (pantallaFuncion);
            break;
        case ERROR_XN:
            programacion
                if (estadoCentral != csServiceMode)
                    XpressNet.setPower(csNormal);
                break;
    }
    nuevaPantalla();
}

void controlEncoder () {
    encoderCambio=false;
    switch (miModo) {
        case NORMAL:
            nuevaVelocidad (encoderValor);
            break;
        case MENU_LOCO:
            digitos[pantallaDigito] = encoderValor;
            miLocomotora = (digitos[MILES] * 1000) + (digitos[CENTENAS] * 100) + (digitos[DECENAS] * 10) +
            digitos[UNIDADES];
            break;
        case MENU_SEL:
            if (encoderValor == 1)
                pantallaMenu = MENU_FUNC;
            else
                pantallaMenu = MENU_LOCO;
            break;
        case MENU_FUNC:
            pantallaFuncion = encoderValor;
            break;
        case ERROR_XN:
            break;
    }
    nuevaPantalla();
}

```

// acciones cuando se pulsa el boton del encoder

// en modo normal paramos o cambiamos de sentido parados

// cambiamos sentido si estamos parados

// paramos, con el sentido adecuado.

// en seleccion menu, entra en el menu correspondiente

// menu seleccionado: Locomotora, Funciones

// el primer digito a cambiar son las unidades

// en menu loco pasa al siguiente digito.

// en menu funcion activa/desactiva funcion

// en error reactiva la central si no estamos en modo de

// actualiza datos en pantalla

// acciones cuando se gira el encoder

// en modo normal cambia la velocidad

// en menu loco actualiza digito

// en menu seleccion muestra la opcion

// en menu funcion selecciona nueva funcion

// en error no hace nada

// actualiza datos en pantalla

El **loop()** queda así de sencillo:

```
void loop() {
    XpressNet.receive(); // llama a la libreria para que vaya recibiendo paquetes
    if (pideInfoLoco)
        pideInformacionLocomotora();
    if (pideInfoFunc)
        pideInformacionFunciones();
    leePulsadores(); // comprueba el estado de los pulsadores
    if (actualizaPantalla) // cada 6ms. Interrupcion TimerOne
        enviaPantalla();
    if (pulsadorOn) // si se pulsa Pulsador
        controlPulsador();
    if (switchOn) // si se pulsa boton encoder
        controlSwitch();
    if (encoderCambio) // si se mueve encoder. Interrupcion pin
        controlEncoder();
}
```

El programa

Aquí está el listado completo.

```
// Mando XpressNet para panel - Paco Cañada 2020
#include <XpressNet.h>
#include <TimerOne.h>

const int pinOutA = 3; // pines del encoder
const int pinOutB = 4;
const int pinSwitch = 5;
const int pinPulsador = 10; // pin del pulsador

const int pinData = 6; // DIO pines pantalla 7 segmentos chip 74HC595
const int pinLatch = 7; // RCLK
const int pinClock = 8; // SCLK

const int pinTXRX = 9; // pin de control del MAX485
const int miDireccionXpressnet = 25; // Dirección del dispositivo en el bus Xpressnet

XpressNetClass XpressNet;

int miLocomotora; // Datos de la locomotora que estoy controlando
uint8_t miVelocidad;
uint8_t miSentido;
uint8_t miPasos;
union funciones {
    uint8_t Xpress[4]; // array para funciones, F0F4, F5F12, F13F20 y F21F28
    unsigned long Bits; // long para acceder a los bits
} miFuncion;

uint8_t estadoCentral; // Estado de la central
bool pideInfoLoco; // pedir informacion de la locomotora
bool pideInfoFunc; // pedir indormacion del estado de las funciones
bool pedirFuncPendiente; // falta pedir informacion del estado de las funciones

#define PASOS14 0 // pasos segun los valores de la libreria
#define PASOS28 2
#define PASOS128 3

#define FUNC_OFF 0 // control de funciones según los valores de la libreria
#define FUNC_ON 1
#define FUNC_CHANGE 2

int outA, outB, copiaOutA, copiaOutB; // valor de las entradas del encoder que usados por ISR
volatile byte encoderValor; // estos valores del encoder son compartidos por ISR y programa
volatile byte encoderMax;
volatile bool encoderCambio;

unsigned long tiempoBotones; // para lectura de los botones
const unsigned long antiRebote = 50; // temporizador antirebote
int estadoPulsador;
int estadoSwitch;
bool pulsadorOn;
bool switchOn;

byte digitos[4]; // patrones a mostrar en cada digito.
uint8_t enciendeDP = -1; // digito a mostrar con el punto decimal encendido
volatile bool actualizaPantalla; // Compartido con interrupcion
volatile byte contadorISR = 0; // contador de la interrupcion

uint8_t pantallaMenu = 0; // menu seleccionado: 0:Locomotora, 1:Funciones
uint8_t pantallaDigito = 0; // digito seleccionado en seleccion de locomotora
uint8_t pantallaFuncion = 0; // valor de la funcion seleccionada
```

```

#define UNIDADES 0           // posicion en el array digitos[]
#define DECENAS 1
#define CENTENAS 2
#define MILES 3

#define CH_ESPACIO 10        // Posicion de los caracteres en la tabla de patrones
#define CH_F 11
#define CH_L 12
#define CH_t 13
#define CH_o 14
#define CH_P 15
#define CH_r 16
#define CH_c 17
#define CH_u 18
#define CH_n 19
#define CH_ADELANTE 20
#define CH_ATRAS 21
#define CH_BOLA 22

static const byte patron[] =      // 7 segmentos valores para 0..9, espacio, F,L,t,o,P,r,c,u,n,°
{
//  patron caracter | 0 = segmento on
//  dpGFEDCBA | 1 = segmento off
//  B11000000, //0 |       A
B11111001, //1 |      -----
B10100100, //2 |   F |       | B
B10110000, //3 |       | G | 
B10011001, //4 |      -----
B10010010, //5 |   E |       | C
B10000010, //6 |       |       |
B11111000, //7 |      ----- dp
B10000000, //8 |       D
B10011000, //9 |
B11111111, // espacio
B10001110, // F
B11000111, // L -
B10000111, // t
B10100011, // o -
B10001100, // P
B10101111, // r -
B10100111, // c -
B11100011, // u -
B10101011, // n -
B11011111, // ' -
B11101111, // , -
B10011100 // ° -
};

enum modos {NORMAL, MENU_SEL, MENU_LOCO, MENU_FUNC, ERROR_XN}; // modo en el que estoy
modos miModo;

void setup() {
pinMode (pinOutA,INPUT_PULLUP);           // entradas
pinMode (pinOutB, INPUT_PULLUP);
pinMode (pinSwitch, INPUT_PULLUP);
pinMode (pinPulsador, INPUT_PULLUP);
copiaOutA = digitalRead (pinOutA);
copiaOutB = digitalRead (pinOutB);
attachInterrupt (digitalPinToInterrupt (pinOutA), encoderISR, CHANGE);
pinMode (pData,OUTPUT);
pinMode (pinLatch,OUTPUT);
pinMode (pinClock,OUTPUT);
Timer1.initialize(6250);                  // Dispara cada 6.25 ms (40 veces/segundo por digito)
Timer1.attachInterrupt(pantallaISR);     // Activa la interrupcion
miLocomotora = 3;                       // Inicialmente controlamos la locomotora 3 a 28 pasos
miPasos = PASOS28;
encoderMax = 28;
encoderValor = 0;
pideInfoLoco = false;
pideInfoFunc = false;
pedirFuncPendiente = false;
miModo = ERROR_XN;                      // Mostramos '0003' hasta que la central responda
mostrarNumero (miLocomotora, 4);
XpressNet.start (miDireccionXpressnet, pinTXRX); // inicializamos libreria
XpressNet.getPower();                   // Pedimos el estado actual de la central
}

void loop() {
XpressNet.receive();                    // llama a la libreria para que vaya recibiendo paquetes
if (pideInfoLoco)
    pideInformacionLocomotora();
if (pideInfoFunc)
    pideInformacionFunciones();
leePulsadores();                      // comprueba el estado de los pulsadores
// cada 6ms. Interrupcion TimerOne
if (actualizaPantalla)
    enviaPantalla();                  // si se pulsa Pulsador
if (pulsadorOn)
    controlPulsador();
}

```

```

if (switchOn)                                // si se pulsa boton encoder
    controlSwitch();
if (encoderCambio)                           // si se mueve encoder. Interrupcion pin
    controlEncoder();
}

void encoderISR () {                         // interrupcion encoder
    outA = digitalRead (pinOutA);
    outB = digitalRead (pinOutB);
    if (outA != copiaOutA) {                  // evitamos rebotes
        copiaOutA = outA;
        if ( outB != copiaOutB) {
            copiaOutB = outB;
            if (outA == outB)                // comprueba sentido de giro
                encoderValor = (encoderValor >= encoderMax) ? encoderMax : ++encoderValor; // CW, hasta maximo
            else
                encoderValor = (encoderValor <=0) ? 0 : --encoderValor; // CCW, hasta 0
            encoderCambio = true;
        }
    }
}

void pantallaISR() {                         // interrupcion pantalla
    contadorISR++;
    if (contadorISR == 4)
        contadorISR = 0;                    // cuenta de 0 a 3
    actualizaPantalla = true;
}

void enviaPantalla() {
    int segmentos, caracter;
    byte contador;
    actualizaPantalla = false;
    contador = contadorISR;               // lo copiamos por si cambia ya que es compartido con ISR
    caracter = digitos[contador];         // caracter a mostrar en el digito
    segmentos = patron[caracter];         // patron de segmentos segun el caracter.
    if (enciendeDP == contador)          // si este digito necesita el punto decimal se enciende
        bitClear (segmentos, 7);
    digitalWrite (pinLatch,LOW);          // inicia transmision
    shiftOut (pinData, pinClock, MSBFIRST, segmentos); // patron segmentos
    shiftOut (pinData, pinClock, MSBFIRST, 1 << (contador)); // activa digito
    digitalWrite (pinLatch, HIGH);
}

void mostrarNumero (int valor, int cifras) {
    digitos[UNIDADES] = valor % 10;       // guardamos el resto de la division en el array
    if (cifras >1)
        digitos[DECENAS] = (valor / 10) % 10;
    if (cifras >2)
        digitos[CENTENAS] = (valor / 100) % 10;
    if (cifras >3)
        digitos[MILES] = (valor / 1000) % 10;
}

uint8_t highByteCV17 (int direccion) {
    if (direccion > 99)                  // si es direccion larga coloca bits altos a uno
        direccion |= 0xC000;
    return (highByte(direccion));
}

void pideInformacionLocomotora () {        // informacion de locomotora.
    pideInfoLoco = false;
    pedirFuncPendiente = true;
    XpressNet.getLocoInfo (highByteCV17(miLocomotora), lowByte(miLocomotora));
}

void pideInformacionFunciones () {          // informacion de funciones.
    pideInfoFunc = false;
    XpressNet.getLocoFunc (highByteCV17(miLocomotora), lowByte(miLocomotora));
}

byte velocidadMaxima (uint8_t pasos) {
    switch (pasos) {                   // devuelve el numero de paso mas alto para el encoder
        case PASOS14:
            return (14);
        break;
        case PASOS28:
            return (28);
        break;
        case PASOS128:
            return (126);
        break;
    }
}

void nuevaVelocidad(uint8_t valor) {        // enviar nueva velocidad como requiere libreria
    int velocidad;
    miVelocidad = valor;
    switch (miPasos) {                 // nos saltamos el paso de parada de emergencia
        case PASOS14:

```

```

case PASOS128:
if (valor>0)
    velocidad = valor + 1;
else
    velocidad = 0;
break;
case PASOS28:
if (valor > 0) // ponemos los 28 pasos en su formato
    velocidad = valor + 3; // pasamos velocidad entre 0 y 31 (0 0 0 S4 S3 S2 S1 S0)
else // nos saltamos pasos parada de emergencia
    velocidad = 0;
bitWrite (velocidad, 5, bitRead (velocidad,0)); // copia bit 0      (0 0 S0 S4 S3 S2 S1 S0)
velocidad >>= 1; // coloca en su sitio      (0 0 0 S0 S4 S3 S2 S1)
break;
}
if (miSentido) // añade el sentido
    velocidad |= 0x80;
XpressNet.setLocoDrive (highByteCV17(miLocomotora), lowByte(miLocomotora), miPasos, velocidad);
}

void cambiaFuncion (int numFuncion) { // activamos o desactivamos la funcion segun su estado
bool estado = bitRead (miFuncion.Bits, numFuncion);
if (estado) {
    XpressNet.setLocoFunc (highByteCV17(miLocomotora), lowByte(miLocomotora), FUNC_OFF, numFuncion);
    bitClear (miFuncion.Bits, numFuncion);
}
else {
    XpressNet.setLocoFunc (highByteCV17(miLocomotora), lowByte(miLocomotora), FUNC_ON, numFuncion);
    bitSet (miFuncion.Bits, numFuncion);
}
}

void notifyXNetPower (uint8_t State) { // La libreria llama a esta funcion cuando cambia estado de la central
    estadoCentral = State;
    enciendeDP = -1; // Apagamos punto decimal
switch (estadoCentral) {
    case csNormal: // Al entrar en modo normal mostramos la direccion de nuestra locomotora
        miModo = NORMAL; // pedimos informacion de la locomotora por si ha cambiado
        mostrarNumero (miLocomotora,4);
        encoderMax = velocidadMaxima (miPasos);
        encoderValor = miVelocidad;
        pideInfoLoco = true;
        break;
    case csShortCircuit: // Corto circuito - OFF
    case csTrackVoltageOff: // Sin tension en via - OFF
        digitos[MILES]= 0;
        digitos[CENTENAS]= CH_F;
        digitos[DECENAS]= CH_F;
        digitos[UNIDADES]= CH_ESPACIO;
        miModo = ERROR_XN;
        break;
    case csEmergencyStop: // Parada emergencia - Stop
        digitos[MILES]= 5;
        digitos[CENTENAS]= CH_t;
        digitos[DECENAS]= CH_o;
        digitos[UNIDADES]= CH_P;
        miModo = ERROR_XN;
        break;
    case csServiceMode: // Programacion en modo servicio - Pro
        digitos[MILES]= CH_P;
        digitos[CENTENAS]= CH_r;
        digitos[DECENAS]= CH_o;
        digitos[UNIDADES]= CH_ESPACIO;
        miModo = ERROR_XN;
        break;
}
}

void notifyLokAll(uint8_t Adr_High, uint8_t Adr_Low, boolean Busy, uint8_t Steps, uint8_t Speed,
                 uint8_t Direction, uint8_t F0, uint8_t F1, uint8_t F2, uint8_t F3, boolean Reg) {
int Loco = ((Adr_High << 8) & 0x3F) + (Adr_Low);
if (Loco != miLocomotora) // si no es nuestra locomotora no hace nada
    return;
miSentido = Direction; // sentido
miPasos = Steps; // pasos
switch (Steps) {
    case PASOS14: // ajusta mi Velocidad segun los pasos
    case PASOS128:
        miVelocidad = Speed - 1; // eliminamos paso de stop de emergencia
        break;
    case PASOS28: // Mover bits a su sitio (0 0 0 S0 S4 S3 S2 S1)
        Speed <<=1; // Hacemos sitio para el bit (0 0 S0 S4 S3 S2 S1 0 )
        bitWrite (Speed, 0, bitRead(Speed,5)); // colocamos bit (0 0 S0 S4 S3 S2 S1 S0)
        Speed &= 0x1F; // Limpiamos (0 0 0 S4 S3 S2 S1 S0);
        miVelocidad = Speed - 3; // eliminamos pasos de stop de emergencia
        break;
}
if (miVelocidad > 127) // Corregimos ajuste (unsigned)
    miVelocidad = 0;
if (pedirFuncPendiente) // Comprobamos si falta pedir informacion de las funciones

```

```

    pedirFuncPendiente = false;
    pideInfoFunc = true;
}
else {
    miFuncion.Xpress[3] = F3;           // F3: F28F27F26F25F24F23F22F21
    miFuncion.Xpress[2] = F2;           // F2: F20F19F18F17F16F15F14F13
    miFuncion.Xpress[1] = F1;           // F1: F12F11F10 F9 F8 F7 F6 F5
// ponemos F0: x x x F0 F4 F3 F2 F1 asi F4 F3 F2 F1 F0 x x x para reordenarlos
    miFuncion.Xpress[0] = F0 <<4;      // F4 F3 F2 F1 0 0 0 0
    if (bitRead(F0, 4))
        bitSet (miFuncion.Xpress[0], 3); // F4 F3 F2 F1 F0 0 0 0
    miFuncion.Bits = miFuncion.Bits >> 3; // desplazamos los 29 bits a su sitio: 000F28F27...F0
}
nuevaPantalla();
}

void notifyXNetStatus (uint8_t State) { // Funcion de la libreria para mostrar estado en un LED
    digitalWrite(LED_BUILTIN, State); // usamos el LED del Arduino (Arduino Uno/Nano: pin 13)
}

void nuevaPantalla() { // mostrar datos en pantalla segun mi modo
    switch (miModo) {
        case NORMAL: // mostrar numero (miVelocidad,3);
            mostrarNumero (miVelocidad,3);
            if (miSentido)
                digitos[MILES] = CH_ADELANTE;
            else
                digitos[MILES] = CH_ATRAS;
            encoderMax = velocidadMaxima(miPasos);
            encoderValor = miVelocidad;
            break;
        case MENU_SEL: // mostrar 'Loco' o 'Func'
            if (pantallaMenu == MENU_FUNC) { // menu seleccionado: Locomotora o Funciones
                digitos[MILES]=CH_F;
                digitos[CENTENAS]=CH_u;
                digitos[DECENAS]=CH_n;
                digitos[UNIDADES]=CH_c;
            }
            else {
                digitos[MILES]=CH_L;
                digitos[CENTENAS]=CH_o;
                digitos[DECENAS]=CH_c;
                digitos[UNIDADES]=CH_o;
            }
            encoderMax = 1;
            encoderValor = (pantallaMenu == MENU_FUNC) ? 1 : 0;
            break;
        case MENU_LOCO: // mostrar numero de locomotora y punto decimal
            mostrarNumero (miLocomotora,4);
            enciendeDP = pantallaDigito; // Enciende el punto decimal
            encoderMax = 9;
            encoderValor = digitos[pantallaDigito]; // en digitos[] ya esta el valor calculado
            break;
        case MENU_FUNC: // mostrar funcion y estado
            mostrarNumero (pantallaFuncion,2);
            digitos[CENTENAS] = CH_F;
            if (bitRead (miFuncion.Bits, pantallaFuncion)) // vemos si la funcion esta activa
                digitos[MILES] = CH_BOLA; // mostrar funcion activa
            else
                digitos[MILES] = CH_ESPACIO; // mostrar funcion apagada
            encoderMax = 28;
            encoderValor = pantallaFuncion;
            break;
        case ERROR_XN: // dejar la pantalla igual
            break;
    }
}

void leePulsadores () {
    int entradaBoton;
    if (millis() - tiempoBotones > antiRebote) { // lee cada cierto tiempo
        tiempoBotones = millis();
        entradaBoton = digitalRead (pinPulsador); // comprueba cambio en boton pulsador
        if (estadoPulsador != entradaBoton)
            estadoPulsador = entradaBoton;
        if (estadoPulsador == LOW)
            pulsadorOn = true;
    }
    entradaBoton = digitalRead (pinSwitch); // comprueba cambio en boton del encoder
    if (estadoSwitch != entradaBoton) {
        estadoSwitch = entradaBoton;
        if (estadoSwitch == LOW)
            switchOn = true;
    }
}

void controlPulsador () { // acciones cuando se pulsa el boton segun el modo
    pulsadorOn = false;
    switch (miModo) {

```

```

case NORMAL:                                // en modo normal pasa al menu de seleccion
    miModo = MENU_SEL;
    break;
case MENU_LOCO:                            // en menu loco pasa a normal,pide informacion de la nueva locomotora
    miModo = NORMAL;
    enciendeDP= -1;
    pideInfoLoco = true;
    return;
    break;
case MENU_SEL:                             // en los otros menu pasa a normal
case MENU_FUNC:
    miModo = NORMAL;
    break;
case ERROR_XN:                           // en error no hace nada
    break;
}
nuevaPantalla();                         // actualiza datos en pantalla

void controlSwitch () {
    switchOn = false;
    switch (miModo) {
        case NORMAL:
            if (miVelocidad == 0)
                miSentido = ! miSentido;
            nuevaVelocidad (0);
            break;
        case MENU_SEL:
            if (pantallaMenu == MENU_FUNC)
                miModo = MENU_FUNC;
            else {
                miModo = MENU_LOCO;
                pantallaDigito = UNIDADES;
            }
            break;
        case MENU_LOCO:
            switch (pantallaDigito) {
                case UNIDADES:
                    pantallaDigito = DECENAS;
                    break;
                case DECENAS:
                    pantallaDigito = CENTENAS;
                    break;
                case CENTENAS:
                    pantallaDigito = MILES;
                    break;
                case MILES:
                    pantallaDigito = UNIDADES;
                    break;
            }
            break;
        case MENU_FUNC:
            cambiaFuncion (pantallaFuncion);
            break;
        case ERROR_XN:
            if (estadoCentral != csServiceMode)
                XpressNet.setPower(csNormal);
            break;
    }
    nuevaPantalla();                         // actualiza datos en pantalla
}

void controlEncoder () {
    encoderCambio=false;
    switch (miModo) {
        case NORMAL:
            nuevaVelocidad (encoderValor);           // en modo normal cambia la velocidad
            break;
        case MENU_LOCO:                          // en menu loco actualiza digito
            digitos[pantallaDigito] = encoderValor;
            miLocomotora = (digitos[MILES] * 1000) + (digitos[CENTENAS] * 100) + (digitos[DECENAS] * 10) +
digitos[UNIDADES];
            break;
        case MENU_SEL:                           // en menu seleccion muestra la opcion
            if (encoderValor == 1)
                pantallaMenu = MENU_FUNC;
            else
                pantallaMenu = MENU_LOCO;
            break;
        case MENU_FUNC:
            pantallaFuncion = encoderValor;         // en menu funcion selecciona nueva funcion
            break;
        case ERROR_XN:
            break;
    }
    nuevaPantalla();                         // actualiza datos en pantalla
}

```

Solución de problemas

Si al programar el Arduino da error hay que desconectar D0 (RX) del pin 1 del MAX485. Esto es debido a que la señal del chip USB llega a D0 a través de una resistencia y la del MAX485 llega directa, por lo que la señal del MAX485 prevalece sobre la del puerto serie. Con el Arduino Mega no ocurre ya que el MAX se conecta a D18 (TX) y D19 (RX), pero no vamos a dedicar un Arduino Mega para este controlador tan simple.

Si el encoder funciona la revés, decrementa en lugar de incrementar, intercambia los pines OutA y OutB

17. Gestor de tarjetas y las colas: Lanzador de rutas XpressNet

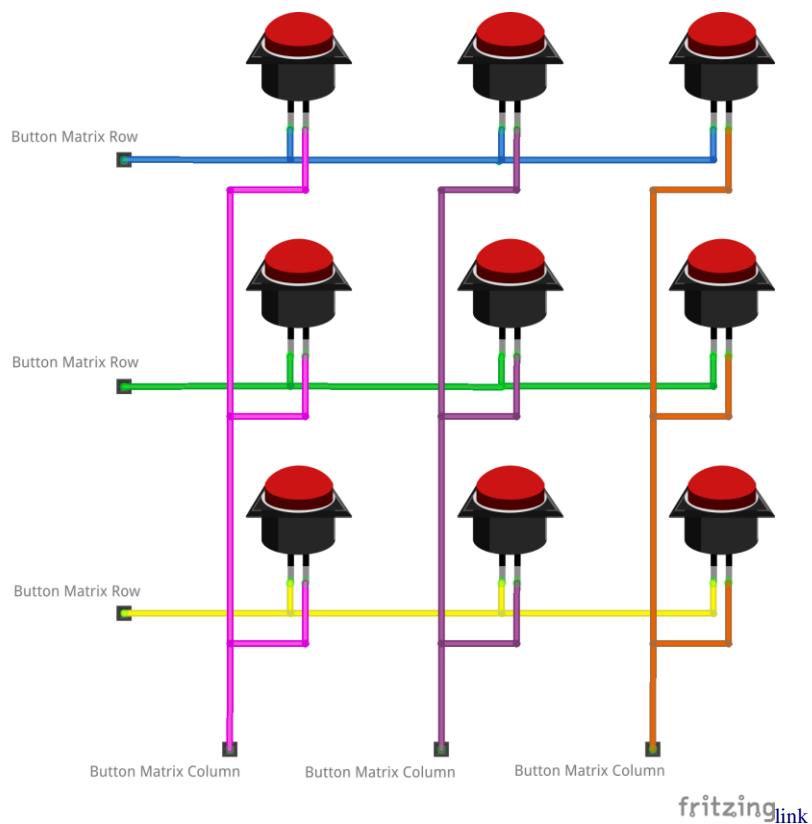
Mi diseño [XbusTCO](#) permite accionar los accesorios digitales desde un TCO con botones (o interruptores) a través de XpressNet, aunque tiene ya unos cuantos años (lo diseñé en 2005) es utilizado por bastantes aficionados a lo largo y ancho del mundo.

Una de las cosas que le falta, ya que no encontré en su momento una manera amigable para el usuario de implementarlo, es que pulsando uno de sus botones se ejecutara una ruta colocando los diferentes desvíos y señales en una posición concreta.

La idea

El Arduino tiene un puerto serie que se utilizará para la comunicación por el bus XpressNet que se podría reprogramar en un momento determinado para tener también comunicación con el ordenador, mostrando unos sencillos menús de texto para programar fácilmente los accesorios y las rutas que se ejecutarán cuando se pulse uno de sus botones.

Los botones los dispondremos en forma de matriz (filas y columnas), así usando pocos pines podemos leer muchos botones. Con 8 pines (4x4) podemos leer 16, con 9 (4x5) serían 20, con 15 (7x8) leeríamos hasta 56.



fritzing [link](#)

La librería [Keypad.h](#) permite leer teclados dispuestos en forma de matriz, además no necesita que se instalen resistencias o diodos en los pines para la correcta lectura de los botones y está disponible a través del Gestor de Librerías del Arduino IDE.

<https://playground.arduino.cc/Code/Keypad/>

Empiezan los problemas

El primer problema que encontramos es que no se puede usar **Serial** y la librería **XpressNet.h** en el Arduino Uno/Nano a la vez. Ambas intentan colocar su rutina de interrupción ISR para controlar el único puerto serie y el compilador se queja indicando error en el `_vector_18`:

```
HardwareSerial0.cpp.o (symbol from plugin): In function `Serial':  
.text+0x0): multiple definition of `__vector_18'  
sketch/USART_ISR.c.o (symbol from plugin):(.text+0x0): first defined here  
HardwareSerial0.cpp.o (symbol from plugin): In function `Serial':  
.text+0x0): multiple definition of `__vector_19'  
sketch/USART_ISR.c.o (symbol from plugin):(.text+0x0): first defined here  
collect2: error: ld returned 1 exit status  
exit status 1  
Error compiling for board Arduino/Genuino Uno.
```

[link](#)

Aunque pudiéramos conseguir usar solo uno a la vez no se podría compilar por la repetición de la rutina ISR. Así que tenemos que buscar otra solución.

La primera que se me ocurre es usar la librería **SoftwareSerial.h** que simula por software un puerto serie en cualquier otro pin digital, incluso se pueden crear varios puertos:

<https://www.arduino.cc/en/Reference/SoftwareSerial>

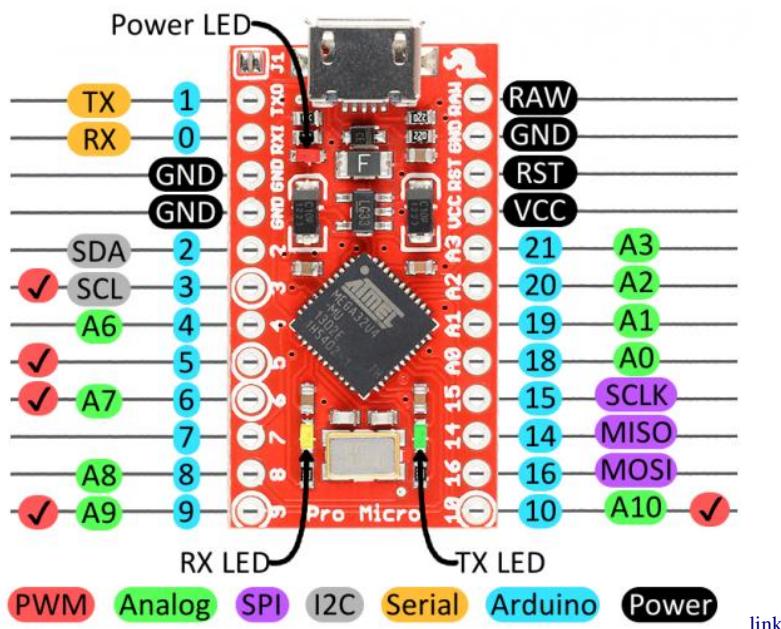
Lamentablemente **SoftwareSerial.h** no soporta el uso de 9 bits que requiere el protocolo XpressNet, esto supone que lo tengamos que dedicar a la interacción con el usuario y por tanto tener que usar un convertidor USB-Serie para conectarlo con el PC pero existiría uno en el Arduino que quedaría sin uso (solo para cargar el programa del Arduino).

Otra solución es usar un Arduino que disponga de más puertos serie como el Arduino Mega 2560, pero me parece un desfalco usarlo cuando solo necesito un puerto serie de 9 bits y unos cuantos pines para los botones.

Buscando he encontrado el Arduino Pro Micro que es bastante barato y suficiente para lo que quiero ya que tiene un puerto serie y una conexión USB implementada.

Arduino Pro Micro

Finalmente, he decidido usar un Arduino Pro Micro que lleva el USB integrado en el propio chip y además tiene un puerto serie independiente:



Algunas diferencias con el Arduino Uno:

- Usa un ATMega 32U4 en lugar de 328p.
- Tiene la misma Flash ROM 32K (aunque 4K se los queda el driver USB) y EEPROM (1K) que el Uno aunque un poco mas de RAM (2,5K).
- No tiene las salidas digitales D11, D12, D13 aunque tiene las D14, D15 y D16 y tampoco el LED de D13 aunque tiene dos LED (RX y TX) que se podrían usar.
- Tiene 9 entradas analógicas y los pines del I2C (SDA y SCL) están en D2 y D3.
- Interrupciones en D0, D1, D2, D3, D7

Está pensado para trabajar con baterías, J1 es para puenteear el regulador de tensión (yo no lo haría forastero!!) si se alimenta de USB.

ATENCION: NO puenteéis J1. Yo no lo tengo puenteado ya que voy a usar a la vez USB y alimentación por RAW.

ATENCION: Hay dos versiones la de 3.3V a 8MHz y la más común de 5V a 16Mhz (la que uso yo). No equivocaros al comprarlo y elegirlo en el Arduino IDE o lo inutilizareis como un ladrillo (*brick*) aunque se podría recuperar siguiendo estos complicados pasos:

<https://enioea2hw.wordpress.com/2017/05/03/arduino-micro-pro-bricked/>

<https://learn.sparkfun.com/tutorials/pro-micro-fio-v3-hookup-guide/troubleshooting-and-faq>

En Windows 10 debería instalarse automáticamente el driver para el Arduino Pro Micro pero si tenéis otro sistema operativo hay que instalarlo, para ello primero hay que bajarlo del repositorio:

https://github.com/sparkfun/Arduino_Boards

Descomprimir el archivo zip y en la carpeta ...Arduino_Boards-master/sparkfun/avr/signed_driver tendréis los archivos .inf y .cat del driver.

Podéis seguir esta guía:

https://learn.sparkfun.com/tutorials/pro-micro-fio-v3-hookup-guide/all#windows_driver

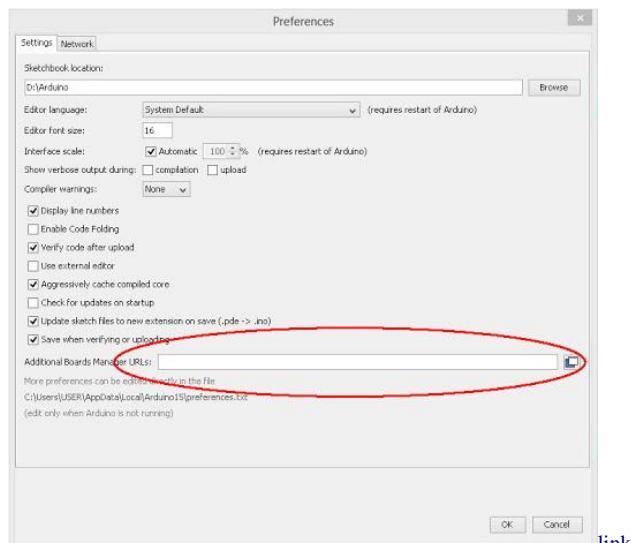
El Gestor de Tarjetas

Un nuevo problema que nos encontramos es que no está en la lista del menú Herramientas -> Placa debido a que ha sido desarrollado por SparkFun y no por la gente de Arduino.

Tendremos que añadirlo a la lista usando el Gestor de Tarjetas, esto permite añadir al Arduino IDE nuevos tipos de tarjeta (Pro Micro, ESP8266, ESP32,...) para que se puedan programar desde el entorno Arduino.

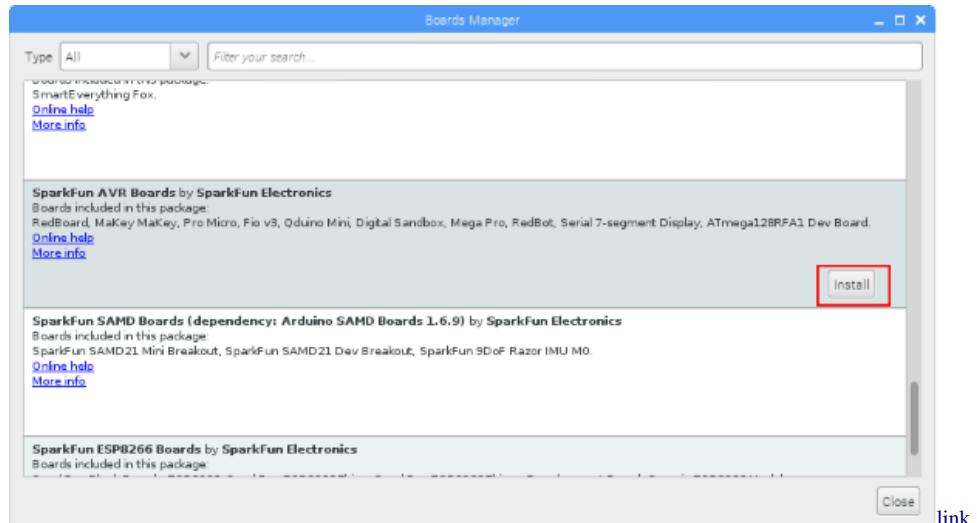
Para añadir el Arduino Pro Micro hay que ir a Archivo -> Preferencias y en el campo 'Gestor de URLs adicionales de tarjetas' y en él añadir la siguiente línea:

https://raw.githubusercontent.com/sparkfun/Arduino_Boards/master/IDE_Board_Manager/package_sparkfun_index.json



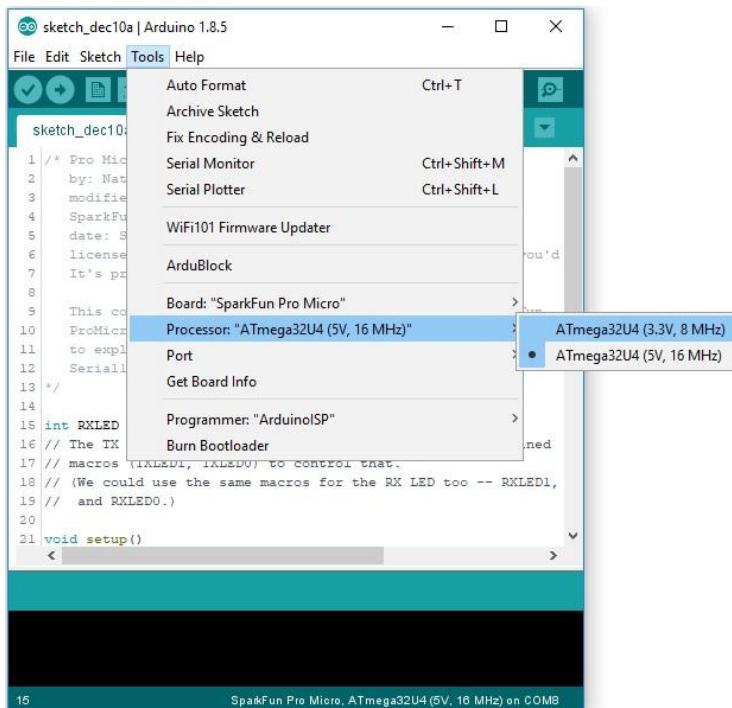
[link](#)

Luego hay que ir a **Herramientas** -> **Placa** -> **Gestor de Tarjetas** y buscar sparkfun, aparecerá la 'SparkFun AVR boards' que tendremos que instalar:



[link](#)

Ahora ya tendremos el Arduino Pro Micro en la lista de placas.



[link](#)

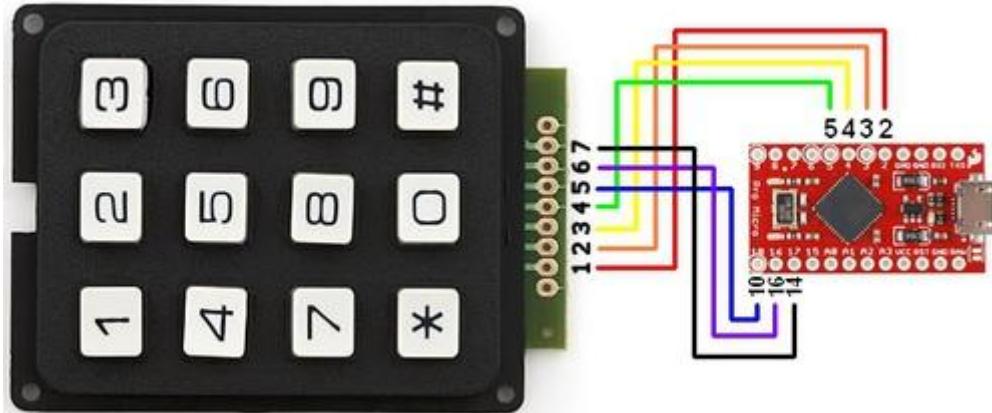
ATENCION: No os equivoquéis al seleccionar vuestra placa ya que hay dos modelos, el que yo uso es el 'SparkFun Pro Micro' 'ATmega32U4 5V 16MHz'. Si os equivocáis cargareis un gestor de arranque (*bootloader*) erróneo que impedirá programarlo y comunicaros con el PC.

Una vez instalado en **Herramientas** -> **Puerto** seleccionáis el puerto del Arduino Pro Micro. Este genera dos puertos, el que os saldrá en la lista es el de comunicación normal, cuando se resetea (activando dos veces seguidas el pin RST) o cuando se programa aparece un segundo puerto que es el que usa el Arduino IDE para programarlo con el bootloader.

Leyendo botones...

La librería **Keypad.h** lee una serie de botones dispuestos en forma de matriz (filas, columnas) y con la función **getKey()** nos da el código de tecla que le hayamos asignado cuando se pulsa , si no hay ningún botón pulsado nos devuelve 0

Para las pruebas he usado un teclado 3x4 como en la imagen:



[link](#)

Hay que definir las conexiones y el código que queremos que devuelva cada tecla para ello se usa un array de dos dimensiones con el que inicializaremos el objeto **Keypad** usando la función **Keypad()**. Observad el orden de los datos y recordar que el cero no se puede usar ya que indica que no se ha pulsado ninguna tecla:

```
#define FILAS 4 // Filas y Columnas de la matriz
#define COLUMNAS 3

char keys[FILAS][COLUMNAS] = { // array dos dimensiones
    {1,2,3},
    {4,5,6},
    {7,8,9},
    {10,11,12}
};

byte filaPins[FILAS] = {5, 4, 3, 2}; // pins de las filas del teclado
byte colPins[COLUMNAS] = {10, 16, 14}; // pins de las columnas del teclado
Keypad keypad = Keypad( makeKeymap(keys), filaPins, colPins, FILAS, COLUMNAS);
```

Cuando se pulse una tecla/botón queremos que se envíe una ruta programada. La ruta consistirá en una serie de direcciones de accesorios (desvíos o señales) en una posición determinada (rojo/verde). Las direcciones de accesorios en XpressNet van de la 0 a la 1023 que se corresponden con los accesorios 1 a 1024.

Para contener la ruta usaremos un array de tantos elementos como botones (filas por columnas) que contendrán hasta un máximo de accesorios por ruta, en nuestro ejemplo 8.

El dato del array será el numero de accesorio que ocupa 10 bits por lo que necesitaremos un **int** para contenerlo, además dedicaremos el bit 15 para indicar la posición (0: Rojo, 1: Verde) por lo que lo definiremos como **unsigned int**.

```
#define MAX_ACCESESORIOS 8 // Maximo de accesorios a enviar por
cada tecla
const int totalRutas = FILAS * COLUMNAS; // numero de rutas
unsigned int MemoriaAccesorios[totalRutas][MAX_ACCESESORIOS]; // Memoria para accesorios
```

Estos accesorios de las rutas se leerán de la EEPROM cada vez que arranque el Arduino. Para comprobar la integridad de los datos pondremos un firma en las dos primeras posiciones de la EEPROM, los caracteres OK. Si al leer estas posiciones están estos caracteres supondremos que los datos son correctos y los cargaremos en el array **MemoriaAccesorios[]** leyendo los dos bytes de cada dato. Si no lo llenaremos con el valor 0xFFFF que lo usaremos como marca de un accesorio no definido.

```
#define INICIO_EEPROM 4 // Direccion a partir de donde se guardan los datos
void leeEEPROM () { // Leer memoria de rutas de la EEPROM
    int direccionEE, n, i;
    unsigned int dato;
    if ((EEPROM.read(0) != 'O') || (EEPROM.read(1) != 'K')){ //Si las dos primeras posiciones no es
'OK'
```

```

        for (n=0; n< totalRutas; n++) // Borra la memoria de rutas
            for (i=0; i< MAX_ACCESESORIOS; i++)
                MemoriaAccesorios[n][i] = 0xFFFF;
    }
    else {
        direccionEE = INICIO_EEPROM;
        for (n=0; n< totalRutas; n++) // Lee la memoria de rutas de la EEPROM
            for (i=0; i< MAX_ACCESESORIOS; i++) {
                dato = (EEPROM.read (direccionEE++) << 8) + EEPROM.read (direccionEE++);
                MemoriaAccesorios[n][i] = dato;
            }
    }
}

```

Para guardar los datos en EEPROM una vez que el usuario ha definido los accesorios de cada ruta haremos igual, guardando los dos bytes de cada dato con **EEPROM.update()** y poniendo la firma en las dos primeras posiciones.

```

void guardaEEPROM () {
    int direccionEE, n, i;
    direccionEE = INICIO_EEPROM;
    for (n=0; n< totalRutas; n++) // Guarda la memoria de rutas de la EEPROM
        for (i=0; i< MAX_ACCESESORIOS; i++) {
            EEPROM.update (direccionEE++, highByte(MemoriaAccesorios[n][i]));
            EEPROM.update (direccionEE++, lowByte(MemoriaAccesorios[n][i]));
        }
    EEPROM.update (0, 'O'); // Marca 'OK' de datos validos
    EEPROM.update (1, 'K');
    Serial.println (F("Rutas guardadas en EEPROM"));
}

```

La capacidad máxima de la EEPROM es de sólo 1K (1024 bytes) si usamos muchos botones y definimos muchos accesorios por ruta quizás desbordemos la capacidad de la misma.

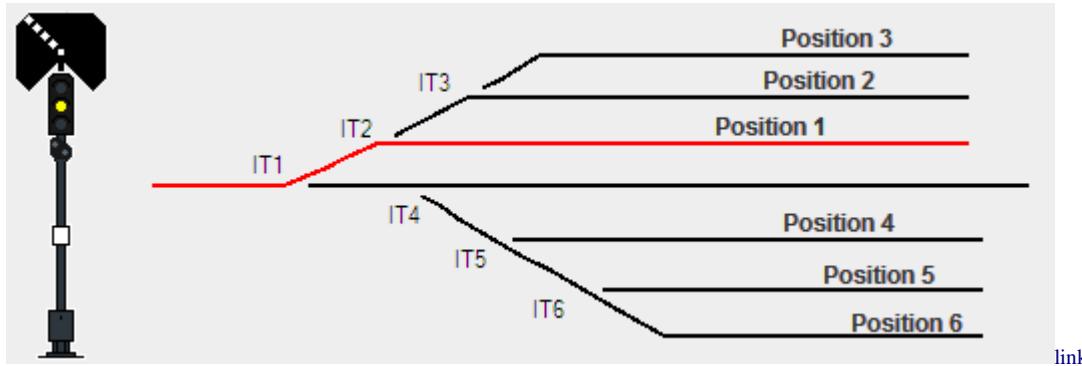
Podemos aprovecharnos del compilador para que nos detecte esta situación indeseada y nos indique un error a la hora de compilar. Para ello usaremos las directivas **#if** y **#endif** para la comprobación y la directiva **#error** para mostrar un mensaje de error en el momento de compilar. Recordad que el array es de unsigned int (2 bytes por dato) y que no empezamos a guardar datos en la primera posición de la EEPROM.

```

#if ((MAX_ACCESESORIOS * FILAS * COLUMNAS) > 510) // Comprobar que no se exceda capacidad EEPROM (1K)
#error "Los datos exceden la capacidad de la EEPROM"
#endif

```

Las Rutas



Para que el usuario vea las rutas que hay definidas haremos una función que mostrará una de las rutas que le pasemos. Leeremos desde el array los accesorios que la componen e imprimiremos el numero de desvío y leeremos con **bitRead()** el bit 15 para mostrar R (rojo) o V (verde) según corresponda. Si el dato es 0xFFFF no mostraremos nada.

Una mejora que se me ha ocurrido es que si en lugar de ejecutar una ruta simplemente queremos mover un desvío (si el pulsador lo hemos montado haciendo un TCO) es usar el bit 14 para indicar que se cambie la posición, así el bit 15 cambiará cada vez que se pulse ese botón. Esto lo indicaremos mostrando una C (Cambio)

```

void listaUnaRuta (int ruta) {
    int i;
    unsigned int dato;
    Serial.print (F("Ruta "));
    Serial.print (ruta + 1);
    Serial.print (F(": < "));
    for (i=0; i< MAX_ACCESESORIOS; i++) {
        dato = MemoriaAccesorios[ruta][i];
        if (dato != 0xFFFF) { // 0xFFFF es accesorio no definido
            Serial.print ((dato & 0x03FF) + 1); // accesorio 1 a 1024
            if (bitRead (dato,14)) // Cambia posicion si bit 14 activado
                Serial.print (F("C "));
            else {
                if (bitRead (dato,15)) // posicion Rojo/Verde segun bit 15
                    Serial.print (F("V "));
                else
                    Serial.print (F("R "));
            }
        }
    }
    Serial.println (F(">")); // fin ruta
}

```

Para listar todas las rutas solo hay que hacer un bucle:

```

void listadoRutas () {
    for (int n=0; n< totalRutas; n++) { // Lista cada ruta
        listaUnaRuta (n);
    }
}

```

Para la introducción de los datos de las rutas el usuario conectará temporalmente el Arduino al ordenador y mediante el Monitor Serie (o el PuTTY) podrá definir las diferentes rutas.

En el Arduino Pro Micro, el puerto USB se usa con **Serial** mientras que el puerto serie (D0 y D1) se usaría con **Serial1**. En nuestro caso la velocidad del puerto puede ser cualquiera ya que es un USB implementado en el propio chip (en el caso de usar un Arduino Mega sería mejor limitarlo a 9600 baudios).

Para recibir los caracteres del usuario usaremos el mismo método simple de introducción de datos que vimos en el capítulo [15. Memoria EEPROM: Decodificador de accesorios multiusos](#) haciendo que el usuario introduzca sólo números entre 0 y un valor máximo. Como es una espera iremos llamando a la librería Xpressnet para que vaya recibiendo datos por si lo tenemos conectado a la central.

```

int entradaSerie (int maximo) {
    unsigned int n=0,c=0,valor=0;
    do {
        if (Serial.available()>0) { // espera a que llegue un caracter procesando Xpressnet
            c = Serial.read();
            if (isDigit(c)) {
                valor = (valor * 10) + (c - '0'); // el digito leido se añade a valor
                n++;
            }
        }
        XpressNet.receive(); // llama a la libreria para que vaya recibiendo paquetes
    } while ((c != '\r') || (n == 0)); // hasta que se pulse enter
    valor = constrain (valor, 0, maximo); // lo ajustamos
    return (valor);
}

```

Para definir una ruta, le pediremos al usuario que indique cual quiere modificar y le pediremos uno a uno los accesorios que la componen y su posición (rojo/verde o cambio).

El número cero lo usaremos para el accesorio no definido (0xFFFF).

Para activar el bit 15 o el bit 14 usaremos **bit()** que nos devuelve el valor del bit (peso) con el que haremos una operación OR con el numero de desvío.

Al finalizar mostraremos la ruta recién creada.

```

void entradaRutas () {
    int ruta, accesorio, posicion, n;
    Serial.print (F("Introduzca Ruta a modificar: (0: Cancelar, 1-"));
    Serial.print (totalRutas);
    Serial.println (F("")));
    ruta = entradaSerie (totalRutas);
    if (ruta==0)
        return;
    Serial.print (F("Defina los "));
    Serial.print (MAX_ACCESESORIOS);
    Serial.print (F(" accesorios de la ruta "));
    Serial.println (ruta);
    for (n=0; n< MAX_ACCESESORIOS; n++) {
        Serial.print (F("Introduzca accesorio "));
        Serial.print (n + 1);
        Serial.println(F(" (0: No definir, 1-1024)"));
        accesorio = entradaSerie (1024);
        if (accesorio > 0) {
            Serial.println (F("Posicion - 0: Rojo, 1: Verde, 2: Cambia"));
            posicion = entradaSerie (2);
            switch (posicion) {
                case 0:
                    MemoriaAccesorios[ruta - 1][n] = accesorio - 1;           // Rojo: bit 15 a 0
                    break;
                case 1:
                    MemoriaAccesorios[ruta - 1][n] = (accesorio - 1) | bit(15); // Verde: bit 15 a 1
                    break;
                case 2:
                    MemoriaAccesorios[ruta - 1][n] = (accesorio - 1) | bit(14); // Cambio: bit 14 a 1
                    break;
            }
        }
        else
            MemoriaAccesorios[ruta - 1][n] = 0xFFFF;                  // no definido
    }
    listaUnaRuta (ruta - 1);
}

```

Al usuario le daremos un menú para que pueda elegir la operación que quiere realizar: Listar, Entrar o Guardar las rutas. Así como una ayuda.

```

void entradaComandos () {
    int c = Serial.read();                                // lee caracter comando
    switch (c) {
        case 'L':                                         // L: Listado de rutas
            Serial.println (F("\nLanza Rutas Xpressnet by Paco Cañada 2020"));
            listadoRutas();
            break;
        case 'E':                                         // E: Entrada de nuevas rutas
            entradaRutas();
            break;
        case 'G':                                         // G: Guardar rutas en EEPROM
            Serial.print (F("\nGuardar todas las Rutas en EEPROM. Esta seguro? - 0: No, 1: Si"));
            c = entradaSerie (1);
            if (c == 1){
                guardaEEPROM();
            }
            else
                Serial.println (F("Guardar rutas cancelado"));
            break;
        case '\n':                                         // descartar
        case '\r':
            break;
        case '?':                                         // ?: Ayuda
        default:
            Serial.println (F("\nIntroduzca uno de estos comandos"));
            Serial.println (F("L: Listado Rutas"));
            Serial.println (F("E: Entrada Rutas"));
            Serial.println (F("G: Guardar Rutas en EEPROM"));
            Serial.println (F("?: Esta ayuda"));
            break;
    }
}

```

Las colas

Cuando se pulsa un botón se han de enviar varios accesorios a la central, si los enviamos todos de una vez dependiendo del consumo de los mismos quizás el decodificador, la fuente de alimentación o la unidad de descarga capacitiva (CDU) no puedan recuperarse a tiempo para mover el siguiente desvío.

Para evitar problemas tendremos que poner un tiempo de espera entre el envío de una orden de mover accesorio y otra. Esto implica que se tardará un tiempo en enviar todos los accesorios de la ruta durante el cual al usuario le puede haber dado por pulsar otro botón de otra ruta por lo que si esperamos a enviarlos todos podemos perder una orden y no ejecutar la segunda ruta.

La solución para esto es que en lugar de enviar directamente los accesorios, los colocaremos en una cola a la espera de ser enviados.

Hay dos tipos las colas **FIFO** (First in, first out) en el que el primero que entra es el primero que sale, como la cola para acceder a un autobús y las pilas **LIFO** (Last in, first out) en el que el último que entra es el primero que sale, como en una pila de libros



[link](#)

Vamos a hacer una cola FIFO en la que ir guardando los accesorios a mover, necesitaremos un array para guardar los datos, un par de índices, para saber donde está el primer elemento a sacar y el ultimo que llegar y una variable para saber cuántos hay en la cola.

```
#define NUM_ACC_COLA 32
unsigned int accesoriosFIFO [NUM_ACC_COLA];
int ultimoLlegar;
int primeroSalir;
int enCola;

void borraFIFO () {
    ultimoLlegar=0;                                // indice cabeza cola
    primeroSalir=0;                               // indice final cola
    enCola=0;                                     // numero de accesorios en cola
}
```

```
// Maximo numero de desvios en cola
// cola FIFO
// indice cabeza cola
// indice final cola
// numero de accesorios en cola
```

```
// indice cabeza cola
// indice final cola
// numero de accesorios en cola
```

Cuando metamos un accesorio en la cola avanzaremos el puntero sin salirnos del array y allí guardaremos el accesorio. La operación **%** nos da el resto de una división, así el puntero siempre estará dentro de los límites del array.

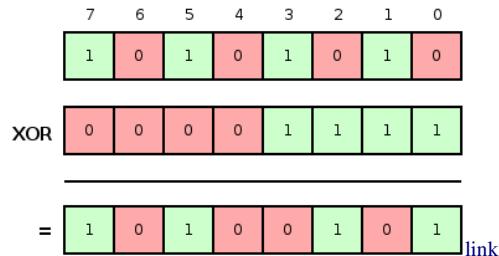
```
void writeFIFO (unsigned int instrucion) {
    ultimoLlegar = (ultimoLlegar + 1) % NUM_ACC_COLA;      // avanza puntero
    enCola++;
    accesoriosFIFO[ultimoLlegar] = instrucion;           // lo guarda en la cola
}
```

Para sacar un elemento hacemos igual, avanzamos el puntero y leemos el valor

```
unsigned int readFIFO () {
    primeroSalir = (primeroSalir + 1) % NUM_ACC_COLA;      //avanza puntero
    enCola--;
    return (accesoriosFIFO[primeroSalir]);
}
```

Para enviar una ruta al pulsar un botón meteremos en la cola todos los accesorios definidos en la ruta. Si no hay definido un accesorio (0xFFFF) no lo metemos, y si tiene activado el bit 14 cambiamos la posición (bit 15) y lo metemos en la cola.

La operación ^ hace una operación XOR bit a bit, si coinciden es resultado es cero, pero si son diferentes deja un uno, con bit(15) solo tenemos activado el bit 15 lo que hace en la práctica es que en el resultado se invierta el bit 15 al realizar la operación.

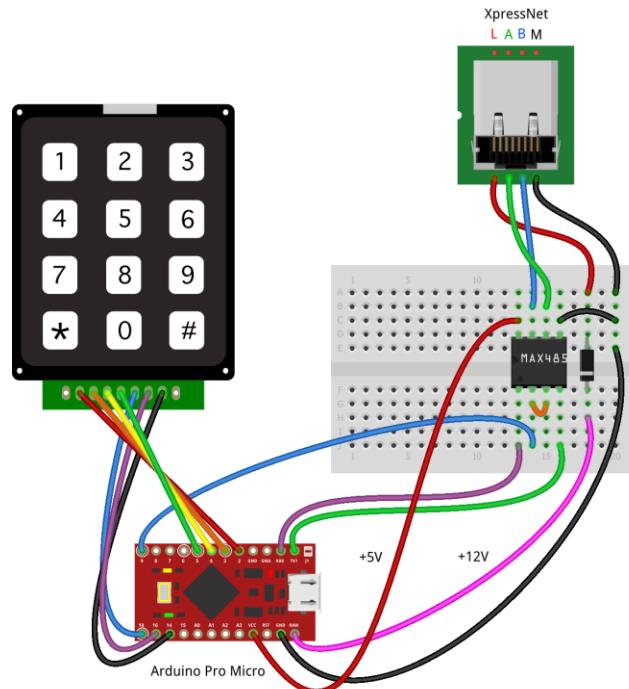


Si la cola está prácticamente llena esperaremos a que se vacíe llamando a nuestra función `enviaColaAccesorios()` que leerá la cola para enviarla a la central

```
void enviaRuta (char tecla) {
    unsigned int accesorio;
    for (int n=0; n < MAX_ACCESORIOS ; n++) {
        while (enCola >= NUM_ACC_COLA -2) // Si la cola esta llena espera a que haya hueco
            enviaColaAccesorios();
        accesorio = MemoriaAccesorios[tecla-1][n];
        if (accesorio != 0xFFFF) // Si esta definido el accesorio, lo pone en la cola
            if (bitRead (accesorio,14)) // Cambio de posicion si bit 14 activado
                accesorio ^= bit(15);
            MemoriaAccesorios[tecla-1][n] = accesorio;
        }
        writeFIFO (accesorio);
    }
}
```

Parcheando...

La conexión con la central XpressNet se hace a través de un MAX485 conectado al puerto serie por lo que mi prototipo ha quedado así:



[fritzing](#)[link](#)

Para poder mover los accesorios hay que conectar con la central mediante el protocolo XpressNet, por suerte la librería [XpressNet.h](#) nos simplifica las cosas ya que nos proporciona una funciones para poder comunicarnos.

Definimos el objeto, la dirección del bus y el pin por donde controlaremos el MAX485, además de una variable para saber el estado de la central y en el [setup\(\)](#) inicializaremos la librería y le pediremos el estado actual con [getPower\(\)](#)

```
XpressNetClass XpressNet; // Objeto XpressNet  
const int pinTXRX = 9; // pin del MAX485  
const int miDireccionXpressnet = 24; // direccion Xpressnet  
byte csStatus; // estado de la central  
  
// en setup()  
XpressNet.start (miDireccionXpressnet, pinTXRX); // inicializamos libreria  
csStatus = csTrackVoltageOff; // Suponemos que la central esta sin tension en via  
XpressNet.getPower(); // Pedimos el estado actual de la central
```

Pero... resulta que este código que hemos utilizado otras veces no se compila, nos da error.
El error proviene de los registros usados para controlar el puerto serie.

La librería intenta usar los registros del puerto [Serial](#), pero nosotros estamos conectando por [Serial1](#). Si compilamos para un Arduino Mega 2560 no da este error ya que elige los registros correctos.

Esto es debido a que la librería no ha contemplado que el Arduino Pro Micro se ha de conectar como el arduino Mega y piensa que es como el Arduino Uno.

La manera de arreglarlo es parchear la librería [Xpressnet.h](#) para que reconozca al Arduino Pro Micro.

Para ello hay que cambiar la siguiente línea del archivo [XpressNet.h](#) que se encuentra en [Mis Documentos -> Arduino -> libraries](#) añadiendo la definición para el Arduino Pro Micro ([_AVR_ATmega32U4_](#)) a la detección del tipo de placa.

NOTA: El Arduino IDE no deja abrir este archivo hay que utilizar otro editor como el Notepad++

<https://notepad-plus-plus.org/>

```
/* From the ATmega datasheet: */  
-----  
// Which serial port is used, if we have more than one on the chip?  
// note that the 328s (the currently produced "smaller" chips) only  
// have one serial port, so we force this.  
// Maybe we are running on a MEGA chip with more than 1 port? If so, you  
// can put the serial port to port 1, and use the port 0 for status messages  
// to your PC.  
#if defined(_AVR_ATmega1280_) || defined(_AVR_ATmega2560_) || defined(_AVR_ATmega32U4_) //Arduino MEGA  
#define SERIAL_PORT_1  
#undef SERIAL_PORT_0  
  
#elif defined(_AVR_ATmega1284P_) || defined(_AVR_ATmega644P_) //Sanguino (other pins!)  
#define SERIAL_PORT_1  
#undef SERIAL_PORT_0  
  
#else //others Arduino UNO  
#define SERIAL_PORT_0  
#undef SERIAL_PORT_1
```

[link](#)

La librería nos informará del estado de la central con [notifyXNetPower\(\)](#) así que guardaremos su estado en la variable [csStatus](#)

```
void notifyXNetPower(uint8_t State){ // La libreria llama a esta funcion cuando cambia estado de la central  
    csStatus = State;  
}
```

La librería también nos informa del estado de la comunicación con [notifyXNetStatus\(\)](#) por si queremos encender un LED. El Arduino Pro Micro no tiene el LED conectado al pin 13 (de hecho no hay pin 13 disponible) pero podemos usar en su lugar el LED RX que está conectado al pin 17

```
const int RXLED = 17; // LED RX indica estado de la central (Pro Micro: 17, Mega: 13)  
void notifyXNetStatus (uint8_t State) { // Funcion de la libreria para mostrar estado en un LED  
    digitalWrite(RXLED, State); // usamos el LED del Arduino  
}
```

Moviendo accesorios

Ya solo nos queda sacar los datos de la cola e ir enviándolos a la central con los tiempos adecuados. Para mover un accesorio hay que mandar la orden de activación y un tiempo después la orden de desactivación, normalmente con 100ms es suficiente para la mayoría de decodificadores. Luego hay que esperar un tiempo para que le de tiempo al condensador/CDU a que se recargue y ya podremos mover el siguiente accesorio.

```
#define ACC_TIEMPO_ON 100 // tiempo de activacion de accesorios en ms
#define ACC_TIEMPO_CDU 500 // tiempo de espera para recarga unidad descarga capacitiva (CDU) en ms
unsigned long tiempoAccesorio; // envio de accesorios
bool recargando, esperaAccesorio;
unsigned int accEnviado;
```

Para enviar un accesorio ponemos los datos como requiere la librería en función de la posición que tenemos en el bit 15 y de si es una orden de activación o desactivación.

```
void enviaAccesorio (unsigned int instruccion, bool activa) {
    byte adrH, pos; // Accessory Decoder operation request (AAAAAAA 1000DBBD)
    adrH = highByte(instruccion) & 0x03;
    pos = 0; // pos = 0000A00P A: activo/inactivo, P: rojo/verde
    if (bitRead(instruccion,15))
        pos |= 0x01;
    if (activa)
        pos |= 0x08;
    XpressNet.setTrntPos (adrH, lowByte(instruccion), pos);
}
```

Para enviar los accesorios de la cola esperaremos los tiempos necesarios para activación/desactivación y espera a recargar apoyándonos en unas banderas (*flags*) para el orden e iremos sacando de la cola un accesorio cada vez y enviándolo.

```
void enviaColaAccesorios () {
    if (recargando) { // esperando a que se recargue la CDU.
        if (millis() - tiempoAccesorio > ACC_TIEMPO_CDU) {
            recargando = false;
        }
    } else {
        if (esperaAccesorio) {
            if (millis() - tiempoAccesorio > ACC_TIEMPO_ON) {
                enviaAccesorio (accEnviado, false); // Envía desconectar accesorio
                tiempoAccesorio = millis();
                esperaAccesorio = false;
                recargando = true;
            }
        } else {
            if ((enCola > 0) && (csStatus == csNormal)) { // Envía accesorio de la cola en estado normal
                accEnviado = readFIFO();
                enviaAccesorio (accEnviado, true); // Envía conectar accesorio
                tiempoAccesorio = millis();
                esperaAccesorio = true;
            }
        }
    }
}
```

Ya solo nos queda definir el **loop()** que queda así:

```
void loop() {
    char tecla;
    XpressNet.receive(); // llama a la libreria para que vaya recibiendo paquetes
    enviaColaAccesorios(); // va enviando los accesorios de la cola
    tecla = keypad.getKey(); // lee teclado
    if ((tecla != NO_KEY) && (csStatus == csNormal)) // si se pulsa tecla en estado normal
        enviaRuta (tecla); // se envia ruta a la cola
    if (Serial.available() >0) // si el usuario escribe un comando
        entradaComandos(); // se ejecuta el comando
}
```

El programa

```
// Lanza Rutas XpressNet - Paco Cañada 2020
// Arduino Pro Micro (32U4)
#include <EEPROM.h>
#include <Keypad.h>
#include <XpressNet.h>
Micro

#define MAX_ACCESESORIOS 8
#define INICIO_EEPROM 4
#define NUM_ACC_COLA 32
#define ACC TIEMPO_ON 100
#define ACC_TIEMPO_CDU 500
#define FILAS 4
#define COLUMNAS 3

#if ((MAX_ACCESESORIOS * FILAS * COLUMNAS) > 510) // Comprobar que no se exceda capacidad EEPROM (1K)
    #error "Los datos exceden la capacidad de la EEPROM"
#endif

char keys[FILAS][COLUMNAS] = { // array dos dimensiones
    {1,2,3},
    {4,5,6},
    {7,8,9},
    {10,11,12}
};

byte filaPins[FILAS] = {5, 4, 3, 2}; // pins de las filas del teclado
byte colPins[COLUMNAS] = {10, 16, 14}; // pins de las columnas del teclado
Keypad keypad = Keypad( makeKeymap(keys), filaPins, colPins, FILAS, COLUMNAS);

const int totalRutas = FILAS * COLUMNAS; // numero de rutas
unsigned int MemoriaAccesorios[totalRutas][MAX_ACCESESORIOS]; // Memoria para accesos
unsigned int accesoriosFIFO [NUM_ACC_COLA]; // cola FIFO
int ultimoLlegar; // indice cabeza cola
int primeroSalir; // indice final cola
int enCola; // numero de accesos en cola
unsigned long tiempoAccesorio; // envio de accesos
bool recargando, esperaAccesorio;
unsigned int accEnviado;

XpressNetClass XpressNet; // Objeto XpressNet

const int pinTXRX = 9; // pin del MAX485
const int miDireccionXpressnet = 24; // direccion Xpressnet
const int RXLED = 17; // LED RX indica estado de la central (Pro Micro: 17, Mega: 13)

byte csStatus; // estado de la central

void setup() {
    Serial.begin (115200);
    pinMode (RXLED,OUTPUT);
    leeEEPROM();
    borraFIFO();
    XpressNet.start (miDireccionXpressnet, pinTXRX);
    csStatus = csTrackVoltageOff;
    XpressNet.getPower();
}

void loop() {
    char tecla;
    XpressNet.receive(); // llama a la libreria para que vaya recibiendo paquetes
    enviaColaAccesorios(); // va enviando los accesos de la cola
    tecla = keypad.getKey(); // lee teclado
    if ((tecla != NO_KEY) && (csStatus == csNormal)) // si se pulsa tecla en estado normal
        enviaRuta (tecla); // se envia ruta a la cola
    if (Serial.available() >0) // si el usuario escribe un comando
        entradaComandos(); // se ejecuta el comando
}

void entradaComandos () {
    int c = Serial.read(); // lee caracter comando
    switch (c) {
        case 'L': // L: Listado de rutas
            Serial.println (F("\nLanza Rutas Xpressnet by Paco Cañada 2020"));
            listadoRutas();
            break;
        case 'E': // E: Entrada de nuevas rutas
            entradaRutas();
            break;
        case 'G': // G: Guardar rutas en EEPROM
            Serial.println (F("\nGuardar todas las Rutas en EEPROM. Esta seguro? - 0: No, 1: Si"));
            c = entradaSerie (1);
            if (c == 1){
                guardaEEPROM();
            }
            else
                Serial.println (F("Guardar rutas cancelado"));
            break;
    }
}

// ATENCION: Parchear libreria para el Arduino Pro
// añadir: || defined(_AVR_ATmega32U4_)
// Maximo de accesos a enviar por cada tecla
// Direccion a partir de donde se guardan los datos
// Maximo numero de desvios en cola
// tiempo de activacion de accesos en ms
// tiempo de espera para recarga unidad descarga capacitiva (CDU) en ms
// Filas y Columnas de la matriz
```

```

        case '\n':                                // descartar
        case '\r':
            break;
        case '?':                                // ?: Ayuda
        default:
            Serial.println (F("\nIntroduzca uno de estos comandos"));
            Serial.println (F("L: Listado Rutas"));
            Serial.println (F("E: Entrada Rutas"));
            Serial.println (F("G: Guardar Rutas en EEPROM"));
            Serial.println (F("?: Esta ayuda"));
            break;
    }
}

void leeEEPROM () {                                // Leer memoria de rutas de la EEPROM
    int direccionEE, n, i;
    unsigned int dato;
    if ((EEPROM.read(0) != 'O') || (EEPROM.read(1) != 'K')) { // Si las dos primeras posiciones no es 'OK'
        for (n=0; n< totalRutas; n++)           // Borra la memoria de rutas
            for (i=0; i< MAX_ACCESESORIOS; i++)
                MemoriaAccesorios[n][i] = 0xFFFF;
    } else {
        direccionEE = INICIO_EEPROM;
        for (n=0; n< totalRutas; n++)           // Lee la memoria de rutas de la EEPROM
            for (i=0; i< MAX_ACCESESORIOS; i++) {
                dato = (EEPROM.read (direccionEE++) << 8) + EEPROM.read (direccionEE++);
                MemoriaAccesorios[n][i] = dato;
            }
    }
}

void guardaEEPROM () {                            // Guarda la memoria de rutas de la EEPROM
    int direccionEE, n, i;
    direccionEE = INICIO_EEPROM;
    for (n=0; n< totalRutas; n++) {
        for (i=0; i< MAX_ACCESESORIOS; i++) {
            EEPROM.update (direccionEE++, highByte(MemoriaAccesorios[n][i]));
            EEPROM.update (direccionEE++, lowByte(MemoriaAccesorios[n][i]));
        }
    }
    EEPROM.update (0, 'O');                      // Marca 'OK' de datos validos
    EEPROM.update (1, 'K');
    Serial.println (F("Rutas guardadas en EEPROM"));
}

void listadoRutas () {                           // Lista cada ruta
    for (int n=0; n< totalRutas; n++)
        listaUnaRuta (n);
}

void listaUnaRuta (int ruta) {
    int i;
    unsigned int dato;
    Serial.print (F("Ruta "));
    Serial.print (ruta + 1);
    Serial.print (F(": <"));
    for (i=0; i< MAX_ACCESESORIOS; i++) {
        dato = MemoriaAccesorios[ruta][i];
        if (dato != 0xFFFF) {                   // 0xFFFF es accesorio no definido
            Serial.print ((dato & 0x03FF) + 1); // accesorio 1 a 1024
            if (bitRead (dato,14))
                Serial.print (F("C "));          // Cambia posicion si bit 14 activado
            else {
                if (bitRead (dato,15))
                    Serial.print (F("V "));      // posicion Rojo/Verde segun bit 15
                else
                    Serial.print (F("R "));
            }
        }
    }
    Serial.println (F(">"));                  // fin ruta
}

int entradaSerie (int maximo) {
    unsigned int n=0,c=0,valor=0;
    do {
        if (Serial.available()>0) {           // espera a que llegue un caracter procesando Xpressnet
            c = Serial.read();
            if (isDigit(c)) {
                valor = (valor * 10) + (c - '0'); // el digito leido se añade a valor
                n++;
            }
        }
        XpressNet.receive();                  // llama a la libreria para que vaya recibiendo paquetes
    } while ((c != '\r') || (n == 0));       // hasta que se pulse enter
    valor = constrain (valor, 0, maximo);   // lo ajustamos
    return (valor);
}

```

```

void entradaRutas () {
    int ruta, accesorio, posicion, n;
    Serial.print (F("Introduzca Ruta a modificar: (0: Cancelar, 1-"));
    Serial.print (totalRutas);
    Serial.println (F("")));
    ruta = entradaSerie (totalRutas);
    if (ruta==0)
        return;
    Serial.print (F("Defina los "));
    Serial.print (MAX_ACCESESORIOS);
    Serial.print (F(" accesorios de la ruta "));
    Serial.println (ruta);
    for (n=0; n< MAX_ACCESESORIOS; n++) {
        Serial.print (F("Introduzca accesorio "));
        Serial.print (n + 1);
        Serial.println(F(" (0: No definir, 1-1024)"));
        accesorio = entradaSerie (1024);
        if (accesorio > 0) {
            Serial.println (F("Posicion - 0: Rojo, 1: Verde, 2: Cambia"));
            posicion = entradaSerie (2);
            switch (posicion) {
                case 0:
                    MemoriaAccesorios[ruta - 1][n] = accesorio - 1;           // Rojo: bit 15 a 0
                    break;
                case 1:
                    MemoriaAccesorios[ruta - 1][n] = (accesorio - 1) | bit(15); // Verde: bit 15 a 1
                    break;
                case 2:
                    MemoriaAccesorios[ruta - 1][n] = (accesorio - 1) | bit(14); // Cambio: bit 14 a 1
                    break;
            }
        } else
            MemoriaAccesorios[ruta - 1][n] = 0xFFFF;                   // no definido
    }
    listaUnaRuta (ruta - 1);
}

void borraFIFO () {
    ultimoLlegar=0;                                         // indice cabeza cola
    primeroSalir=0;                                         // indice final cola
    enCola=0;                                                 // numero de accesorios en cola
}

unsigned int readFIFO () {
    primeroSalir = (primeroSalir + 1) % NUM_ACC_COLA;      //avanza puntero
    enCola--;
    return (accesoriosFIFO[primeroSalir]);
}

void writeFIFO (unsigned int instruccion) {
    ultimoLlegar = (ultimoLlegar + 1) % NUM_ACC_COLA;      // avanza puntero
    enCola++;
    accesoriosFIFO[ultimoLlegar] = instruccion;           // lo guarda en la cola
}

void enviaRuta (char tecla) {
    unsigned int accesorio;
    for (int n=0; n < MAX_ACCESESORIOS ; n++) {
        while (enCola >= NUM_ACC_COLA - 2)                  // Si la cola esta llena espera a que haya hueco
            enviaColaAccesorios();
        accesorio = MemoriaAccesorios[tecla-1][n];
        if (accesorio != 0xFFFF) {                            // Si esta definido el accesorio, lo pone en la cola
            if (bitRead (accesorio,14)) {                     // Cambio de posicion si bit 14 activado
                accesorio ^= bit(15);
                MemoriaAccesorios[tecla-1][n] = accesorio;
            }
            writeFIFO (accesorio);
        }
    }
}

void enviaColaAccesorios () {
    if (recargando) {                                     // esperando a que se recargue la CDU.
        if (millis() - tiempoAccesorio > ACC_TIEMPO_CDU) {
            recargando = false;
        }
    } else {
        if (esperaAccesorio) {
            if (millis() - tiempoAccesorio > ACC_TIEMPO_ON) {
                enviaAccesorio (accEnviado, false);          // Envia desconectar accesorio
                tiempoAccesorio = millis();
                esperaAccesorio = false;
                recargando = true;
            }
        }
    } else {
        if ((enCola > 0) && (csStatus == csNormal)) {    // Envia accesorio de la cola en estado normal
            accEnviado = readFIFO();
        }
    }
}

```

```

        enviaAccesorio (accEnviado, true); // Envia conectar accesorio
        tiempoAccesorio = millis();
        esperaAccesorio = true;
    }
}

void enviaAccesorio (unsigned int instruccion, bool activa) { // Accessory Decoder operation request (AAAAAAA
1000DBBD)
    byte adrH, pos;
    adrH = highByte(instruccion) & 0x03;
    pos = 0; // pos = 0000A00P A: activo/inactivo, P: rojo/verde
    if (bitRead(instruccion,15))
        pos |= 0x01;
    if (activa)
        pos |= 0x08;
    XpressNet.setTrntPos (adrH, lowByte(instruccion), pos);
}

void notifyXNetPower (uint8_t State) { // La libreria llama a esta funcion cuando cambia estado
de la central
    csStatus = State;
}

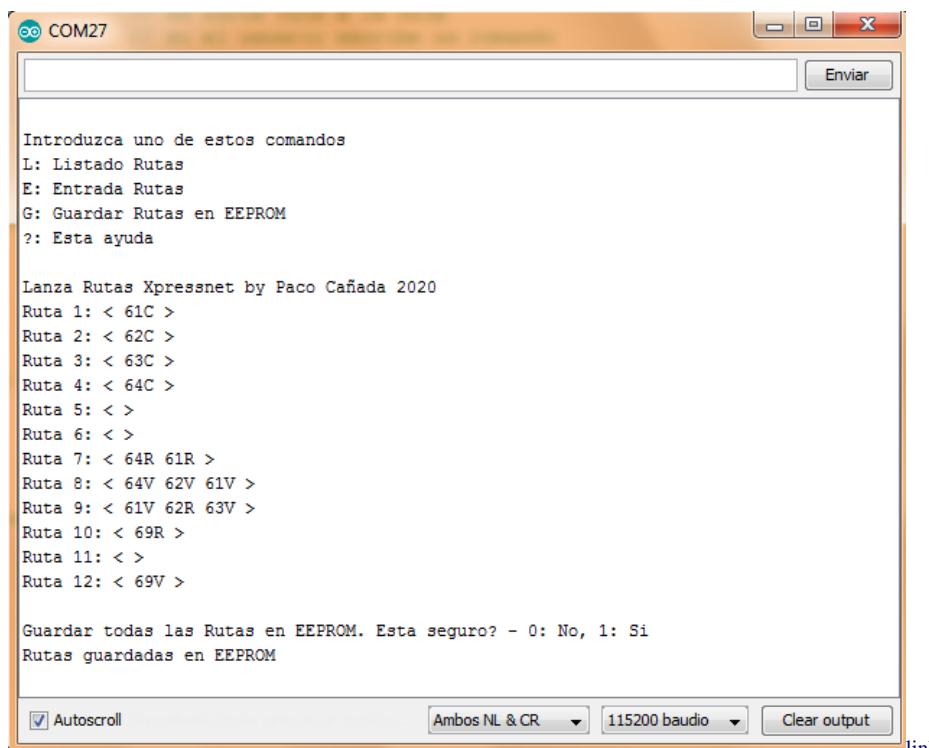
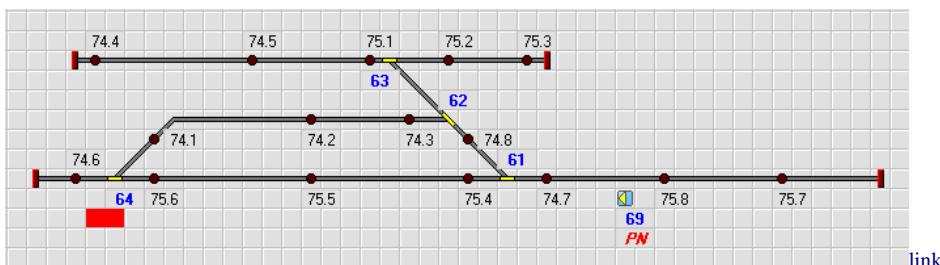
void notifyXNetStatus (uint8_t State) { // Funcion de la libreria para mostrar estado en un LED
digitalWrite(RXLED, State);
}

```

NOTA: Ya que no tomamos nota de la posición de los desvíos cuando se envían a la central y cuando definimos un accesorio como cambio de posición solo se cambia en esa definición, si definimos dos veces ese accesorio no funcionará como se espera ya que cada definición guarda su posición independientemente.

Las definiciones de cambio se deberían usar solas en un botón para cambiar la posición de ese accesorio.

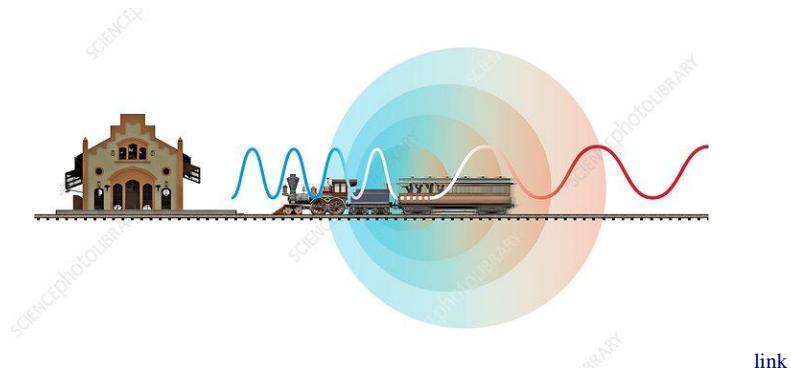
Estas son las rutas para mi módulo CTMS de la estación del Carrilet de Salou:



18. Sketch en varios archivos: Megafonía para la maqueta

Un elemento que da realismo a las maquetas es la inclusión de sonido ambiente o de megafonías de la estación, así podremos tener el ambiente sonoro de una estación, la tranquilidad del campo, la ajetreada ciudad, etc.

La atmósfera de nuestra maqueta cambia si se escucha de fondo un sonido de día lluvioso acorde con la iluminación natural que nos llega desde la ventana o un despertar, con gallo incluido, de la actividad de nuestro pequeño mundo a escala.



También se puede tener locuciones para el anuncio de nuestros trenes en la estación. En algunos decodificadores de sonido para locomotoras ya suelen incluir alguna locución pero son bastante limitadas y muchas veces poco adecuadas para nuestro tren además de que no queda muy real que el mismo tren se anuncie a sí mismo al tiempo que se acerca a una estación, es mejor que el sonido se reproduzca desde la estación.



La idea

Vamos a realizar un decodificador DCC que nos permita reproducir sonidos reales guardados en una tarjeta SD en formato .wav o .mp3

El reto está en no limitar el número de sonidos a unos pocos y que podamos escoger varios para ser reproducidos uno detrás de otro.

Si tenemos 5 trenes y 3 estaciones en nuestra maqueta, las megafonías personalizadas para la llegada y partida de nuestros trenes a cada estación ya suponen 30 locuciones.

Si escogemos que sea como un decodificador de locomotoras con sonido como los que se montan en nuestros trenes nos veremos limitados al número de funciones que se pueden usar (28 o 29), si por el contrario escogemos que sea como un decodificador de accesorios podemos tener muchas más (hasta 999 o 2044 según sistemas) pero tampoco es plan de usar un gran cantidad de direcciones de accesorios porque en caso de megafonías lo ideal sería que cada estación tuviera su decodificador.

Una solución intermedia es hacer un decodificador de accesorios de sólo 16 direcciones lo que nos dará opción a seleccionar 32 sonidos (uno para la posición roja y otro para la verde).

Necesitamos prever una manera de detener el sonido destinando una de esa posiciones al efecto y se podría tener la opción de repetir un sonido, así el sonido ambiente de un día de campo no necesitaría de un archivo de sonido de muchos minutos, con uno más corto repetido nos daría el mismo ambiente.

Para no quedarnos limitados como en el caso de los decodificadores de locomotoras y dado que los archivos residen en una SD, los podríamos repartir en carpetas y destinar alguna dirección a seleccionar una carpeta en concreto así multiplicamos los sonidos a los que podemos acceder.

De esta manera haremos que podamos acceder a 24 sonidos por carpeta distribuidos por 6 carpetas lo que nos permitirá tener [144 sonidos](#) diferentes con posibilidad de parar los sonidos y de repetir el último.

Dir	Pos	Descripción
1	0	Sonido 1
1	1	Sonido 2
2	0	Sonido 3
2	1	Sonido 4
3	0	Sonido 5
3	1	Sonido 6
4	0	Sonido 7
4	1	Sonido 8
5	0	Sonido 9
5	1	Sonido 10
6	0	Sonido 11
6	1	Sonido 12
7	0	Sonido 13
7	1	Sonido 14
8	0	Sonido 15
8	1	Sonido 16
9	0	Sonido 17
9	1	Sonido 18
10	0	Sonido 19
10	1	Sonido 20
11	0	Sonido 21
11	1	Sonido 22
12	0	Sonido 23
12	1	Sonido 24
13	0	Carpeta 1
13	1	Carpeta 2
14	0	Carpeta 3
14	1	Carpeta 4
15	0	Carpeta 5
15	1	Carpeta 6
16	0	Stop
16	1	Repetir

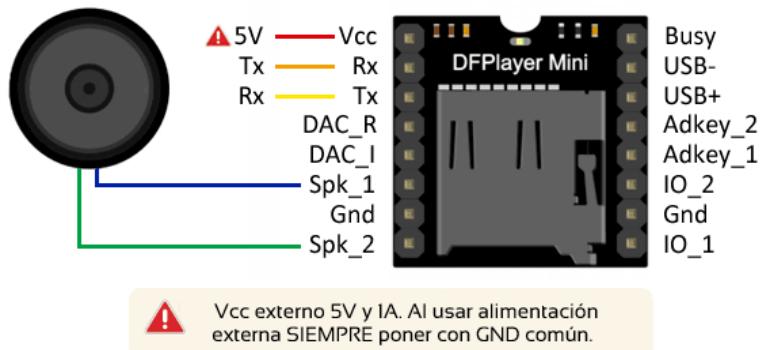
[link](#)

Tendríamos que poder escoger otro sonido aunque no haya acabado de reproducirse el actual así podríamos formar frases con pequeños fragmentos de sonido: "Tren talgo", "procedente de Madrid Atocha", "con destino Barcelona", "viene con retraso"

Si usamos control por ordenador reproducir diferentes locuciones sería tan sencillo como crear una ruta.

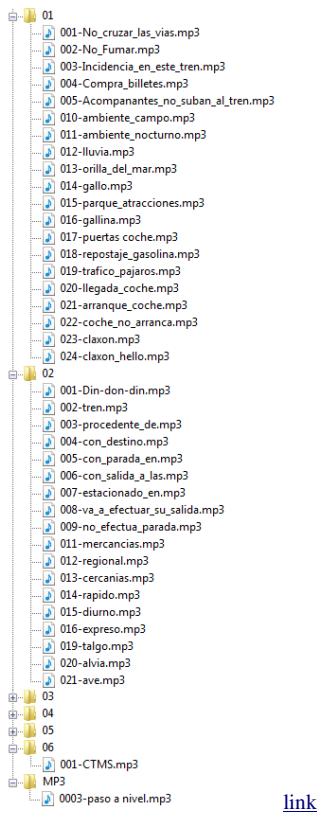
Reproduciendo sonidos

El DFPlayer Mini es un pequeño módulo reproductor de sonido en formato MP3 y wav de bajo precio y con un amplificador de 3W con salida directa al altavoz de 8 ohm. El módulo puede utilizarse de forma autónoma con batería, altavoz y pulsadores conectados, o en combinación con un Arduino a través del puerto serie.



[link](#)

La conexión con el módulo DFPlayer Mini es muy sencilla ya que la comunicación se realiza a través del puerto de serie (9600 bps). El pin RX de Arduino se conecta con el TX del módulo y viceversa. El modulo funciona entre 3.2V y 5V (típico 4.2V) pero el puerto serie funciona a 3.3V por lo que hay que colocar una resistencia de 1kΩ entre el pin RX del DFPlayer y el TX del Arduino para adaptarlo a las tensiones del puerto serie del Arduino ya que trabaja a 5V.



Dispone de un lector de tarjetas micro SD de hasta 32 GB en formato FAT16 y FAT32. Soporta hasta 100 carpetas (01, 02, ... 99) y puede acceder hasta 3000 archivos en la carpeta MP3.

Los nombres de archivos no pueden contener espacios ni otros caracteres que no sean letras, números y guiones bajos. Los archivos se nombran de la forma 001.mp3, 002.mp3... (también es válido 001tren_talgo.mp3, 002_lluvia.mp3,...) en las diferentes carpetas.

En la carpeta mp3 se nombran de la forma 0001.mp3, 0002.mp3, 0004megafonia.mp3,...

Para controlar este módulo se usa la librería `DFRobotDFPlayerMini.h` disponible a través del gestor de librerías:

<https://github.com/DFRobot/DFRobotDFPlayerMini>

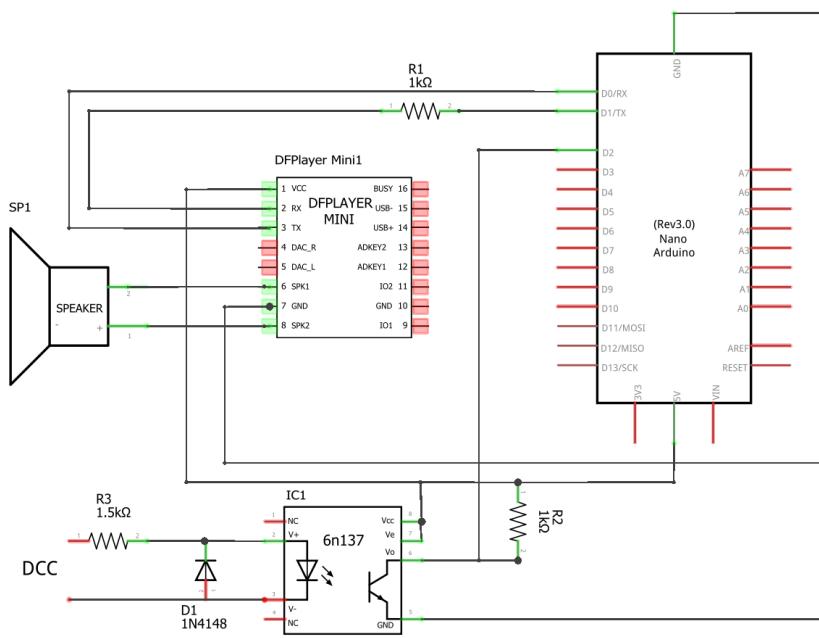
Entre las posibilidades de este módulo está el poder seleccionar el volumen, la ecualización, almacenamiento en SD y memoria USB, salida amplificada o para auriculares, control por puerto serie o manual a través de botones.

El esquema

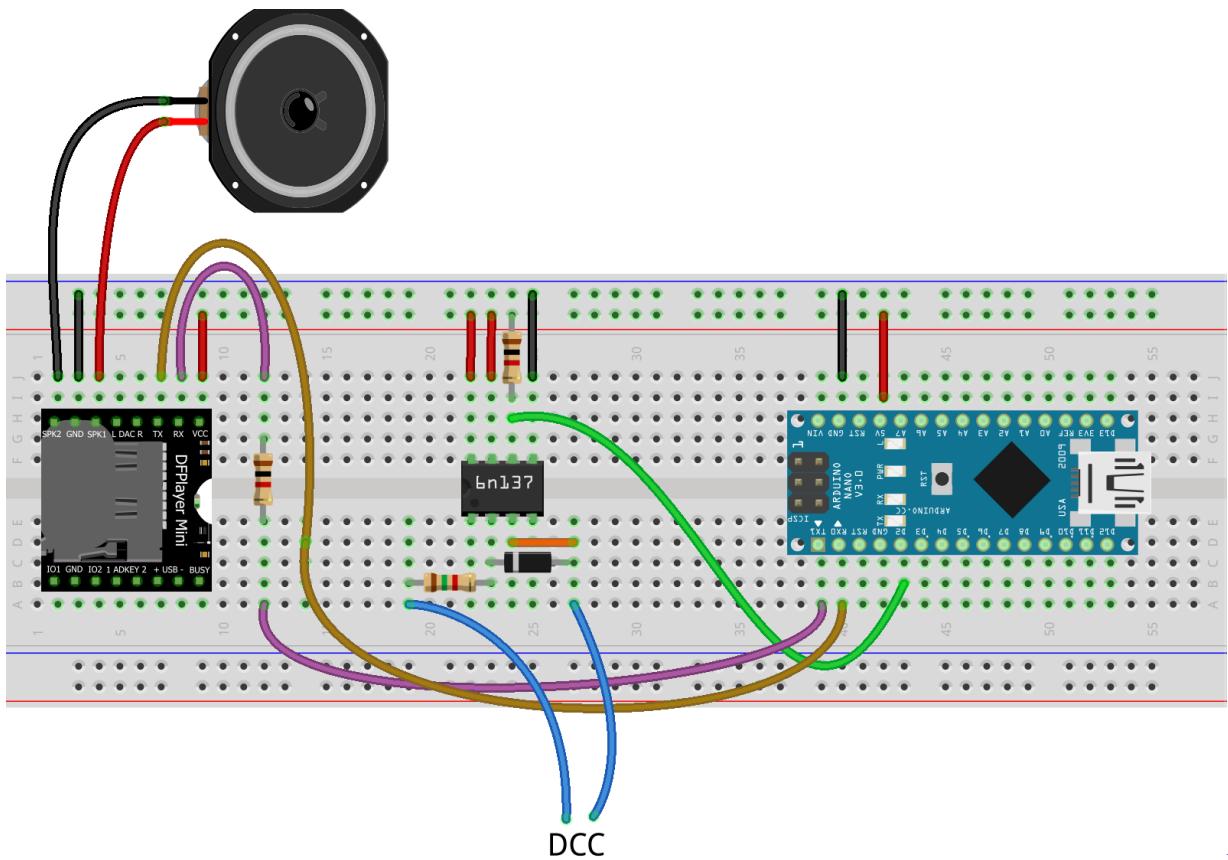
Usaremos el circuito con el optoacoplador 6N137 y la librería [NmraDcc.h](#) ya utilizados en otros proyectos para hacer llegar la señal DCC al pin D2 del Arduino (por comodidad de montaje usaremos un Arduino Nano aunque se puede usar con cualquier otro) y decodificar la señal.

Conectaremos el módulo DFPlayer por el puerto serie al Arduino a través de la resistencias y lo alimentaremos desde los 5V

Todo se alimentará desde un cargador USB de 5V conectado al Arduino Nano, de esta manera tendremos suficiente corriente para alimentar el amplificador ya que no se alimentará por el regulador de 5V incluido en el Arduino que no proporcionaría suficiente corriente (la falta de corriente produce unos ruidos molestos en el altavoz).



link



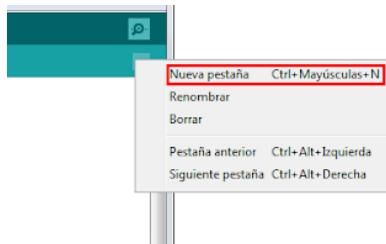
link

El desarrollo

A medida que vamos escribiendo programas para Arduino ocurre que estos son cada vez más largos ya que vamos añadiendo funciones haciendo que el listado ocupe cientos de líneas.

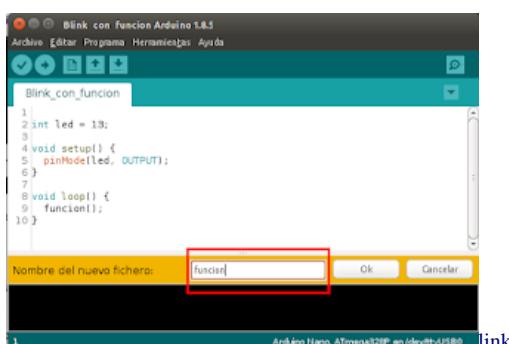
Buscar una línea para cambiar una función concreta puede ser engoroso. Una manera de gestionar esto es dividir el código en varios archivos .ino de forma que acceder a las diferentes partes de un programa sea más sencillo. En el Arduino IDE lo podemos hacer fácilmente añadiendo pestañas dentro del mismo sketch.

Para esto nos tenemos que ir a la flecha situada en la parte superior derecha de la pantalla donde se nos abrirá el siguiente menú:



link

Nos saldrá un recuadro donde introduciremos el nombre del archivo:



• link

Lo bueno que tiene el entorno Arduino IDE es que nos oculta las dificultades de la programación en C++ como la inclusión de librerías y el prototipado de funciones antes de proceder a compilar el código quedando una estructura simple de cara al usuario.

El IDE de Arduino funciona uniendo todos los archivos .ino dentro de la carpeta del programa (comenzando con el archivo que se llama igual que la carpeta seguido de los otros en orden alfabético) en una carpeta temporal y realiza un preprocesamiento para convertir el sketch en un programa C++ para que realmente funcione antes de pasarlo al compilador avr-gcc.

Primero añade `#include "Arduino.h"` al inicio del *sketch*. Esta librería incluye todas las definiciones necesarias para el núcleo Arduino estándar. También añade directivas `#line` con comentarios, avisos o errores.

A continuación, el entorno busca definiciones de funciones dentro del sketch y crea declaraciones (prototipos) para ellas sino los tienen. Estos se insertan después de cualquier comentario o declaración previa al procesador (`#include` o `#define`), pero antes de cualquier otra declaración.

También copia a esa carpeta temporal todos los archivos necesarios de las librerías que incluyamos en el sketch.

Este preprocesamiento no se realiza a otros ficheros con extensiones diferentes a .ino (como los .h o .cpp)

Por ejemplo, si dividimos nuestro *sketch* en tres archivos (pestañas) con este contenido:

The screenshot shows the Arduino IDE interface with three tabs open: 'sketch.ino' (active), 'funcion1.ino', and 'funcion2.ino'. The code in 'sketch.ino' is:

```
1 void setup() {  
2 }  
3  
4 void loop() {  
5     miFuncion();  
6     miOtraFuncion();  
7 }
```

A link button is visible at the bottom right of the code area.

Archivo sketch.ino

```
void setup() {  
}  
  
void loop() {  
    miFuncion();  
    miOtraFuncion();  
}
```

Archivo funcion1.ino

```
void miFuncion (void) {  
}
```

Archivo funcion2.ino

```
void miOtraFuncion (void) {  
}
```

El entorno Arduino lo cambia de esta manera antes de enviarlo al compilador para cumplir con las reglas de la programación en C++:

```
#include <Arduino.h>  
#line 1 "sketch.ino"  
void setup();  
#line 4 "sketch.ino"  
void loop();  
#line 1 "funcion1.ino"  
void miFuncion(void);  
#line 1 "funcion2.ino"  
void miOtraFuncion(void);  
  
#line 1 "sketch.ino"  
void setup() {  
}
```

```

void loop() {
    miFuncion();
    miOtraFuncion();
}

#line 1 " funcion1.ino"
void miFuncion (void) {

}

#line 1 " funcion2.ino"
void miOtraFuncion (void) {
}

```

La cola

Como queremos que mientras se está reproduciendo un sonido se puedan recibir órdenes DCC para añadir sonidos una vez el actual finalice y se reproduzca a continuación necesitamos implementar una cola en nuestro *sketch* como hicimos en un capítulo anterior ([17. Gestor de tarjetas y colas: Lanzador de rutas XpressNet](#))

En la cola guardaremos la orden de sonido o carpeta que recibamos. Ya que hemos decidido que haya 24 sonidos y 6 carpetas al establecer las direcciones del decodificador de accesorios, los números 0 a 23 representarán un sonido específico y del 24 al 29 representarán una carpeta específica. La cola podrá almacenar hasta 64 órdenes así no habrá problemas por que cambiemos de carpeta para seleccionar varios sonidos.

La cola será del tipo FIFO (*First in First Out*) y tendremos funciones para inicializarla, introducir un dato y sacar un dato de la misma. Además la variable `enCola` nos dirá en todo momento cuantos elementos hay en la cola.

```

#define NUM_ACC_COLA  64                                // Numero de accesorios en la cola de envio

byte sonidoFIFO [NUM_ACC_COLA];                      // cola FIFO
int ultimoLlegar;                                     // indice cabeza cola
int primeroSalir;                                    // indice final cola
int enCola;                                         // numero de accesorios en cola

void borraFIFO () {                                 // indice cabeza cola
    ultimoLlegar=0;                                  // indice final cola
    primeroSalir=0;                                 // numero de accesorios en cola
    enCola=0;
}

byte readFIFO () {                                // avanza puntero
    primeroSalir = (primeroSalir + 1 ) % NUM_ACC_COLA;
    enCola--;
    return (sonidoFIFO[primeroSalir]);
}

void writeFIFO (byte sonido) {                     // avanza puntero
    ultimoLlegar = (ultimoLlegar + 1 ) % NUM_ACC_COLA;
    enCola++;
    sonidoFIFO[ultimoLlegar] = sonido;             // lo guarda en la cola
}

```

Reproduciendo sonidos

En el **setup()** inicializaremos la librería para que use el puerto serie del Arduino y seleccione la reproducción de la SD en el módulo DFPlayer.

```
DFRobotDFPlayerMini DFPlayer;  
  
Serial.begin (9600); // Comunicacion con DFPlayer  
if (!DFPlayer.begin (Serial))  
    errorSD(); // en caso de error, comprobar conexion y SD  
DFPlayer.setTimeout(500); // tiempo para responder 500ms  
DFPlayer.setOutputDevice(DFPLAYER_DEVICE_SD); // selecciona reproducir desde tarjeta SD
```

En caso de error en la inicialización (mala conexión o que la tarjeta no esté insertada) lo comunicaremos al usuario haciendo parpadear el LED de la placa del Arduino.

```
void errorSD () {  
    pinMode (LED_BUILTIN, OUTPUT);  
    for (;;) { // Parpadea LED indefinidamente  
        digitalWrite (LED_BUILTIN, HIGH);  
        delay (500);  
        digitalWrite (LED_BUILTIN, LOW);  
        delay (500);  
    }  
}
```

El bucle **for(;;)** sin parametros es un bucle sin fin ya que no hay definida una condición final que marque el final del mismo.

Dedicaremos una pestaña a colocar las funciones relacionadas con el reproductor DFPlayer. El archivo lo llamaremos **MegafoniaDFPlayer.ino**

La función principal será la que controle si hay órdenes para reproducir sonidos en la cola y hará que se reproduzcan una vez finalice el sonido en curso.

Para saber si se está reproduciendo un sonido definiremos la variable **Playing**, cuando sea **false** querrá decir que no se reproduce ningún sonido en el módulo. Cuando finaliza un sonido el modulo DFPlayer necesita unos 100ms en leer la información del siguiente archivo de la SD así que tendremos que dejarle ese tiempo antes de poder enviar una orden para que reproduzca un sonido de la SD.

La librería tiene diferentes funciones para reproducir un sonido:

Comando	Descripción
play (fileNumber)	Reproduce un archivo del directorio principal según el orden en que se grabó en la SD (no tiene en cuenta el nombre)
playFolder (folderNumber,fileNumber)	Reproduce un archivo específico (1-255) de un directorio específico (1-99)
playMp3Folder (fileNumber)	Reproduce un archivo específico del directorio MP3 (1-3000)

Nosotros vamos a usar **playFolder()** y usaremos nuestra variable **ultimaCarpeta** para contener la ultima orden de carpeta recibida, así no necesitamos especificar cada vez la carpeta si el sonido está almacenado en la misma carpeta que el ultimo sonido.

También definiremos la variable **repetir** para la orden de repetir o dejar de repetir el ultimo sonido, para ello usaremos las funciones de la librería **enableLoop()** para repetir y **disableLoop()** para dejar de repetirlo.

```
bool Playing;  
byte ultimaCarpeta = 1;  
bool repetir = false;  
unsigned long tiempoDFP;  
  
void runDFPlayer() {  
    byte sonido;  
  
    if ((! Playing) && (enCola > 0)) { // si no se reproduce sonido y hay sonidos en la cola  
        if ((millis()- tiempoDFP) > 100) { // DFPlayer necesita 100ms  
            sonido = readFIFO();  
            if (sonido > 23) { // cambio de carpeta  
                ultimaCarpeta = sonido - 23;  
            }  
        }  
    }
```

```

        else {                                // sonido
            tiempoDFP = millis();
            DFPlayer.playFolder(ultimaCarpeta, sonido + 1);
            repetir = false;
            Playing = true;
        }
    }
}

void activaBucleSonido () {
    repetir = true;
    DFPlayer.enableLoop();
}

void desactivaBucleSonido () {
    repetir = false;
    DFPlayer.disableLoop();
}

```

Para conocer el estado del reproductor la librería recibe los mensajes de estado del DFPlayer, a nosotros nos interesa saber cuándo ha dejado de reproducir un archivo ya que así podremos hacer que reproduzca el siguiente en la cola, además debemos tomar nota del tiempo para dejar al módulo que finalice sus operaciones.

En el **loop()** comprobaremos si le llegan mensajes y los decodificaremos según los datos de la librería. Si no se reproducen archivos nuestra función **notPlaying()** pondrá la variable **Playing** a **false** y tomara nota del tiempo actual.

```
if (DFPlayer.available()) {                                // recibe mensaje de DFPlayer
    mensajeDFPlayer(DFPlayer.readType(), DFPlayer.read());
```

y en el archivo **MegafoniaDFPlayer.ino** pondremos

```

void notPlaying () {                                // pone estado de no reproduciendo
    Playing=false;
    tiempoDFP = millis();
}

void mensajeDFPlayer(uint8_t tipo, int valor){      // procesa mensaje del DFPlayer
    switch (tipo) {
        case DFPlayerPlayFinished:                //track is finished playing
        case TimeOut:                            //DFPlayer doesn't answer in 500ms
        case WrongStack:                         //SD card is pulled out
        case DFPlayerCardRemoved:                 //SD card is plugged in
        case DFPlayerCardInserted:                //SD card is pulled out
            notPlaying();
            break;
        case DFPlayerUSBInserted:                //USB flash is plugged in
        case DFPlayerUSBRemoved:                 //USB flash is pulled out
        case DFPlayerCardOnline:                  //SD card online
            break;
        case DFPlayerError:
            switch (valor) {
                case Busy:                      //initialization is not done
                case FileIndexOut:              //Specified track is out of current track scope
                case FileMismatch:             //Specified track is not found
                case CheckSumNotMatch:         //Checksum incorrect
                case SerialWrongStack:         //a frame has not been received completely yet
                case Sleeping:                //supports only specified device in sleep mode
                case Advertise:               //a inter-cut operation only can be done when a track is being played)
                    notPlaying();
                    break;
            }
            break;
        default:                                // cualquier otro error
            break;
    }
}

```

Ya solo nos queda definir una función para borrar la lista de reproducción en caso necesario para tenerlo todo listo:

```
void resetPlayList () {                                // borra la lista de reproduccion
    borraFIFO();
    notPlaying();
}
```

Reproductor DCC

Para el control del reproductor desde DCC usaremos la librería `NmraDcc.h` que con la función `notifyDccAccTurnoutOutput()` nos informa de que ha recibido una orden de accesorios.

Además hace la gestión de las CV aunque nosotros solo necesitaremos leerlas al principio para asignar la dirección base de nuestro decodificador de 16 direcciones que guardaremos en la variable `dccBase`.

Necesitaremos una CV para establecer el volumen por defecto para el modulo de sonido (valor entre 1 y 30) y así poderla cambiar cuando nos interese, usaremos la CV33 que está reservada para el fabricante.

En el `setup()` inicializamos la librería `NmraDcc.h` y leeremos la dirección base de nuestro modulo desde las CV con nuestra función `leeDireccionDcc()`, además leeremos la CV que contiene el volumen y se lo pasaremos al DFPlayer.

```
NmraDcc Dcc; // Objetos libreria

const byte DCC_PIN = 2; // pin señal DCC
#define CV_VOLUMEN 33 // CV para volumen por defecto
#define DEF_VOLUMEN 20 // Valor del volumen por defecto (1-30)

int dccBase;
int volumen;

Dcc.pin (digitalPinToInterrupt(DCC_PIN), DCC_PIN, 1); // Configura DCC
Dcc.initAccessoryDecoder (MAN_ID_DIY, 1, FLAGS_OUTPUT_ADDRESS_MODE, 0);
dccBase = leeDireccionDcc(); // Lee direccion base DCC
volumen = Dcc.getCV (CV_VOLUMEN); // Lee volumen por defecto
volumen = constrain (volumen, 1, 30);
DFPlayer.volume(volumen);
```

Tambien dedicaremos otra pestaña a contener todas nuestras funciones relacionadas con el DCC. El archivo lo llamaremos `MegafoniaDCC.ino`

Para no complicarnos usaremos CV1 y CV9 para definir la dirección base de nuestro decodificador DCC de 16 direcciones. Así la dirección inicial será la definida por la fórmula:

$$\text{dirección base} = (\text{CV9} * 256) + (\text{CV1} * 4) - 3$$

Para CV1=1 y CV9=0 (valores por defecto de los decodificadores de accesorios) es la dirección 1, para CV1=2 y CV9=0 la dirección será la 5.

En nuestra función `leeDireccionDcc()` aprovecharemos para inicializar el resto de CV si la CV8 no es igual a 13 (fabricante DIY, Do it Yourself), así tenemos una manera de resetear nuestro decodificador.

```
int leeDireccionDcc() {
    int direccion, cv1, cv9;

    if (Dcc.getCV (CV_MANUFACTURER_ID) != MAN_ID_DIY) { // Si CV8 no es 13, resetea CV
        Dcc.setCV (CV_ACCESSORY_DECODER_ADDRESS_LSB, 1);
        Dcc.setCV (CV_ACCESSORY_DECODER_ADDRESS_MSB, 0);
        Dcc.setCV (CV_VOLUMEN, DEF_VOLUMEN);
        Dcc.setCV (CV_MANUFACTURER_ID, MAN_ID_DIY);
    }
    cv1 = Dcc.getCV (CV_ACCESSORY_DECODER_ADDRESS_LSB);
    cv9 = Dcc.getCV (CV_ACCESSORY_DECODER_ADDRESS_MSB);
    direccion = ((cv9 << 8) + (cv1 << 2)) - 3;
    return (direccion);
}
```

Cuando se reciba una orden de cambio de accesorios por DCC, se ejecutará la función `notifyDccAccTurnoutOutput()` que tendremos que definir.

En esta función deberemos comprobar que la dirección del paquete DCC es para una de las 16 direcciones que maneja nuestro decodificador.

Si lo es, la convertiremos en un valor entre 0 y 31 teniendo en cuenta si la posición recibida es **Rojo** o **Verde**, lo haremos multiplicando la dirección recibida dentro del rango por 2 y añadiéndole la posición.

Ahora solo hay que comparar este valor con los definidos para reproducir uno de los sonidos (0 a 23), establecer una carpeta (24 a 29), detener el sonido (30) o repetirlo (31).

Hay que tener en cuenta que esta función será llamada con cada paquete de accesorios recibido y que las centrales suelen repetir el mismo paquete de accesorios varias veces o mientras se mantiene pulsada la tecla en el mando.

Para evitar introducir en la cola valores repetidos, que no es lo que deseamos, lo introduciremos solo cuando cambie su valor o bien si ha pasado más de un segundo desde la última vez que se recibió el mismo valor.

Necesitaremos algunas variables para controlar esto: `dccSonido` para guardar la ultima orden de selección de sonido, `dccCarpetas` para la ultima orden de carpeta recibida y `dccTiempo` para el temporizador de un segundo.

Si es una carpeta compararemos `dccCarpetas` con la orden recibida y en caso de que sea diferente la pondremos en la cola FIFO y actualizamos la variable.

Si es la orden de detener la reproducción, enviaremos la orden `stop()` al DFPlayer y borraremos la cola FIFO y las variables asociadas a la reproducción. Luego introduciremos en la cola la ultima orden de cambio de carpeta recibida para no confundir al usuario cuando envíe una nueva orden.

En caso de recibir la orden de repetir el ultimo sonido, comprobaremos si ha pasado más de un segundo desde la última orden para activar o desactivar la repetición del último sonido recibido en bucle.

Para la reproducción de sonidos comprobamos si es el mismo sonido recibido la ultima vez. Si lo es, ha de pasar un segundo antes de que sea una orden válida.

Si es válido lo introducimos en la cola y actualizamos la variable `dccSonido`

```
byte dccSonido = -1;
byte dccCarpetas = ultimaCarpeta + 23;
unsigned long dccTiempo = millis();

void notifyDccAccTurnoutOutput( uint16_t Addr, uint8_t Direction, uint8_t OutputPower ) { byte
sonido;
unsigned long tiempoActual;

if ((Addr >= dccBase) && (Addr <= dccBase + 15) && OutputPower) { // comprueba direccion dentro
del rango
    sonido = (((Addr - dccBase) << 1) + Direction); // orden de sonido entre 0 y 31
    switch (sonido) { // actua segun orden
        case 24: // Seleccion de carpeta SD (01 a 06)
        case 25:
        case 26:
        case 27:
        case 28:
        case 29:
            if (dccCarpetas != sonido) { // mete en la cola los cambios de carpeta
                dccCarpetas = sonido;
                writeFIFO (sonido);
            }
            break;
        case 30: // Detiene sonidos
            DFPlayer.stop();
            resetPlayList();
            writeFIFO (dccCarpetas);
            break;
        case 31: // Reintroduce ultima carpeta recibida
            tiempoActual = millis();
            if (tiempoActual - dccTiempo < 1000) // Ha de pasar mas de un segundo para poder repetir
                break;
            dccTiempo = tiempoActual;
            if (repetir)
                desactivaBucleSonido();
            else
                activaBucleSonido();
            break;
        default:
            tiempoActual = millis(); // Seleccion sonido (001.mp3 a 024.mp3)
            if (sonido == dccSonido) { // Evita paquetes repetidos
                if (tiempoActual - dccTiempo < 1000) // Ha de pasar mas de un segundo para reintroducir
el mismo sonido
                    break;
            }
            writeFIFO(sonido);
            dccSonido = sonido;
            dccTiempo = tiempoActual;
            break;
    }
}
}
```

Reproducción manual

Si queremos un control básico por teclas podemos montar en el módulo DFPlayer dos pulsadores que nos permitirán subir y bajar el volumen con una pulsación larga o pasar al siguiente/anterior sonido con una pulsación corta. Y mediante un tercer pulsador podemos tener la función PLAY/PAUSA.



[link](#)

Se pueden poner más botones para el control manual consultando el manual de usuario del módulo DFPlayer:

https://github.com/Arduinolibrary/DFPlayer_Mini_mp3/raw/master/DFPlayer%20Mini%20Manual.pdf

El programa

El programa dividido en los diferentes archivos .ino quedaría así:

Archivo MegafoniaMaqueta.ino

```
#include <NmraDcc.h>
#include <DFRobotDFPlayerMini.h>

NmraDcc Dcc; // Objetos libreria
DFRobotDFPlayerMini DFPlayer;

const byte DCC_PIN = 2; // pin señal DCC
#define CV_VOLUMEN 33 // CV para volumen por defecto
#define DEF_VOLUMEN 20 // Valor del volumen por defecto (1-30)

int volumen;
bool Playing;
byte ultimaCarpeta = 1;
bool repetir = false;

byte dccSonido = -1;
byte dccCarpeta = ultimaCarpeta + 23;
int dccBase;

unsigned long dccTiempo = millis();
unsigned long tiempoDFP; // Numero de accesorios en la cola de envio

#define NUM_ACC_COLA 64

byte sonidoFIFO [NUM_ACC_COLA];
int ultimoLlegar; // indice cabeza cola
int primeroSalir; // indice final cola
int enCola; // numero de accesorios en cola

void setup() {
  Serial.begin (9600); // Comunicacion con DFPlayer
  if (!DFPlayer.begin (Serial)) // en caso de error, comprobar conexion y SD
    errorSD(); // tiempo para responder 500ms
  DFPlayer.setTimeOut(500); // selecciona reproducir desde tarjeta SD
  Dcc.pin (digitalPinToInterrupt (DCC_PIN), DCC_PIN, 1);
  Dcc.initAccessoryDecoder (MAN_ID_DDIY, 1, FLAGS_OUTPUT_ADDRESS_MODE, 0);
  dccBase = leeDireccionDcc(); // Lee direccion base DCC
  volumen = Dcc.getCV (CV_VOLUMEN); // Lee volumen por defecto
  volumen = constrain (volumen, 1, 30);
```

```

DFPlayer.volume(volumen);
resetPlayList();                                     // resetea lista reproduccion
}

void loop() {
  Dcc.process();                                     // procesa paquetes DCC
  runDFPlayer();                                    // procesa estado reproduccion
  if (DFPlayer.available()) {                       // recibe mensaje de DFPlayer
    mensajeDFPlayer(DFPlayer.readType(), DFPlayer.read());
  }
}

void borraFIFO () {
  ultimoLlegar=0;                                  // indice cabeza cola
  primeroSalir=0;                                 // indice final cola
  enCola=0;                                       // numero de accesorios en cola
}

byte readFIFO () {
  primeroSalir = (primeroSalir + 1 ) % NUM_ACC_COLA; //avanza puntero
  enCola--;
  return (sonidoFIFO[primeroSalir]);               // lo guarda en la cola
}

void writeFIFO (byte sonido) {
  ultimoLlegar = (ultimoLlegar + 1 ) % NUM_ACC_COLA; // avanza puntero
  enCola++;
  sonidoFIFO[ultimoLlegar] = sonido;                // lo guarda en la cola
}

```

Archivo MegafoniaDFPlayer.ino

```

void resetPlayList () {                           // borra la lista de reproduccion
  borraFIFO();
  notPlaying();
}

void notPlaying () {                            // pone estado de no reproduciendo
  Playing=false;
  tiempoDFP = millis();
}

void activaBucleSonido () {
  repetir = true;
  DFPlayer.enableLoop();
}

void desactivaBucleSonido () {
  repetir = false;
  DFPlayer.disableLoop();
}

void mensajeDFPlayer(uint8_t tipo, int valor){ // procesa mensaje del DFPlayer
  switch (tipo) {
    case DFPlayerPlayFinished:           //track is finished playing
    case TimeOut:                      //DFPlayer doesn't answer in 500ms
    case WrongStack:                   //SD card is pulled out
    case DFPlayerCardRemoved:          //SD card is plugged in
    case DFPlayerCardInserted:         //notPlaying();
    break;
    case DFPlayerUSBInserted:          //USB flash is plugged in
    case DFPlayerUSBRemoved:           //USB flash is pulled out
    case DFPlayerCardOnline:           //SD card online
    break;
    case DFPlayerError: {
      switch (valor) {
        case Busy:                    //initialization is not done
        case FileIndexOut:            //Specified track is out of current track scope
        case FileMismatch:           //Specified track is not found
        case CheckSumNotMatch:       //Checksum incorrect
        case SerialWrongStack:       //a frame has not been received completely yet
        case Sleeping:               //supports only specified device in sleep mode
        case Advertise:              //a inter-cut operation only can be done when a track is being played
        notPlaying();
        break;
      }
      break;
    default:                         // cualquier otro error
    break;
  }
}

void runDFPlayer() {
  byte sonido;

  if ((! Playing) && (enCola > 0)) {           // si no se reproduce sonido y hay sonidos en la cola
    if ((millis()- tiempoDFP) > 100) {          // DFPlayer necesita 100ms al acabar la reproduccion para
      inicializar la informacion de la proxima
      sonido = readFIFO();
    }
  }
}

```

```

        if (sonido > 23) {                                // cambio de carpeta
            ultimaCarpeta = sonido - 23;
        }
        else {                                         // sonido
            tiempoDFP = millis();
            DFPlayer.playFolder(ultimaCarpeta, sonido + 1);
            repetir = false;
            Playing = true;
        }
    }
}

void errorSD () {
    pinMode (LED_BUILTIN, OUTPUT);
    for (;;) {                                     // Parpadea LED indefinidamente
        digitalWrite (LED_BUILTIN, HIGH);
        delay (500);
        digitalWrite (LED_BUILTIN, LOW);
        delay (500);
    }
}

```

Archivo MegafoniaDcc.ino

```

int leeDireccionDcc() {
    int direccion, cv1, cv9;

    if (Dcc.getCV (CV_MANUFACTURER_ID) != MAN_ID_DIY) {           // Si CV8 no es 13, resetea CV
        Dcc.setCV (CV_ACCESSORY_DECODER_ADDRESS_LSB, 1);
        Dcc.setCV (CV_ACCESSORY_DECODER_ADDRESS_MSB, 0);
        Dcc.setCV (CV_VOLUMEN, DEF_VOLUMEN);
        Dcc.setCV (CV_MANUFACTURER_ID, MAN_ID_DIY);
    }
    cv1 = Dcc.getCV (CV_ACCESSORY_DECODER_ADDRESS_LSB);
    cv9 = Dcc.getCV (CV_ACCESSORY_DECODER_ADDRESS_MSB);
    direccion = ((cv9 << 8) + (cv1 << 2)) - 3;
    return (direccion);
}

void notifyDccAccTurnoutOutput( uint16_t Addr, uint8_t Direction, uint8_t OutputPower ) {
    byte sonido;
    unsigned long tiempoActual;

    if ((Addr >= dccBase) && (Addr <= dccBase + 15) && OutputPower) { // comprueba direccion dentro del rango
        sonido = (((Addr - dccBase) << 1) + Direction);                // orden de sonido entre 0 y 31
        switch (sonido) {                                                 // actua segun orden
            case 24:                                                 // Seleccion de carpeta SD (01 a 06)
            case 25:
            case 26:
            case 27:
            case 28:
            case 29:
                if (dccCarpeta != sonido) {                           // mete en la cola los cambios de carpeta
                    dccCarpeta = sonido;
                    writeFIFO (sonido);
                }
                break;
            case 30:                                                 // Detiene sonidos
                DFPlayer.stop();
                resetPlayList();
                writeFIFO (dccCarpeta);
                break;
            case 31:
                tiempoActual = millis();                            // Reintroduce ultima carpeta recibida
                if (tiempoActual - dccTiempo < 1000)                 // Repite ultimo sonido
                    if (repetir)
                        desactivaBucleSonido();
                    else
                        activaBucleSonido();
                break;
            default:
                tiempoActual = millis();                            // Seleccion sonido (001.mp3 a 024.mp3)
                if (sonido == dccSonido) {                          // Evita paquetes repetidos
                    if (tiempoActual - dccTiempo < 1000)             // Ha de pasar mas de un segundo para
                    reintroducir el mismo sonido
                        break;
                }
                writeFIFO(sonido);
                dccSonido = sonido;
                dccTiempo = tiempoActual;
                break;
        }
    }
}

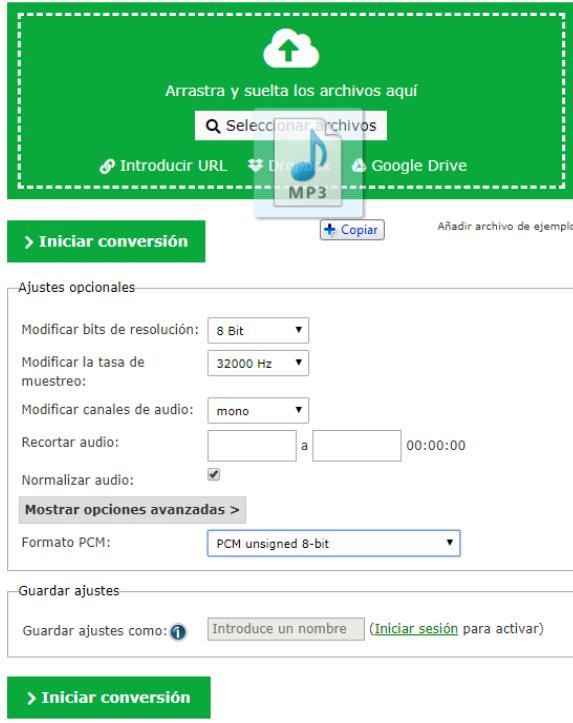
```

Creando Sonidos

El reproductor DFPlayer soporta archivos de sonido .mp3 y .wav con frecuencias de muestreo (sampling rate) de 8kHz, 11.025 kHz, 12 kHz, 16 kHz, 22.05 kHz, 24 kHz, 32 kHz, 44.1 kHz y 48 kHz.

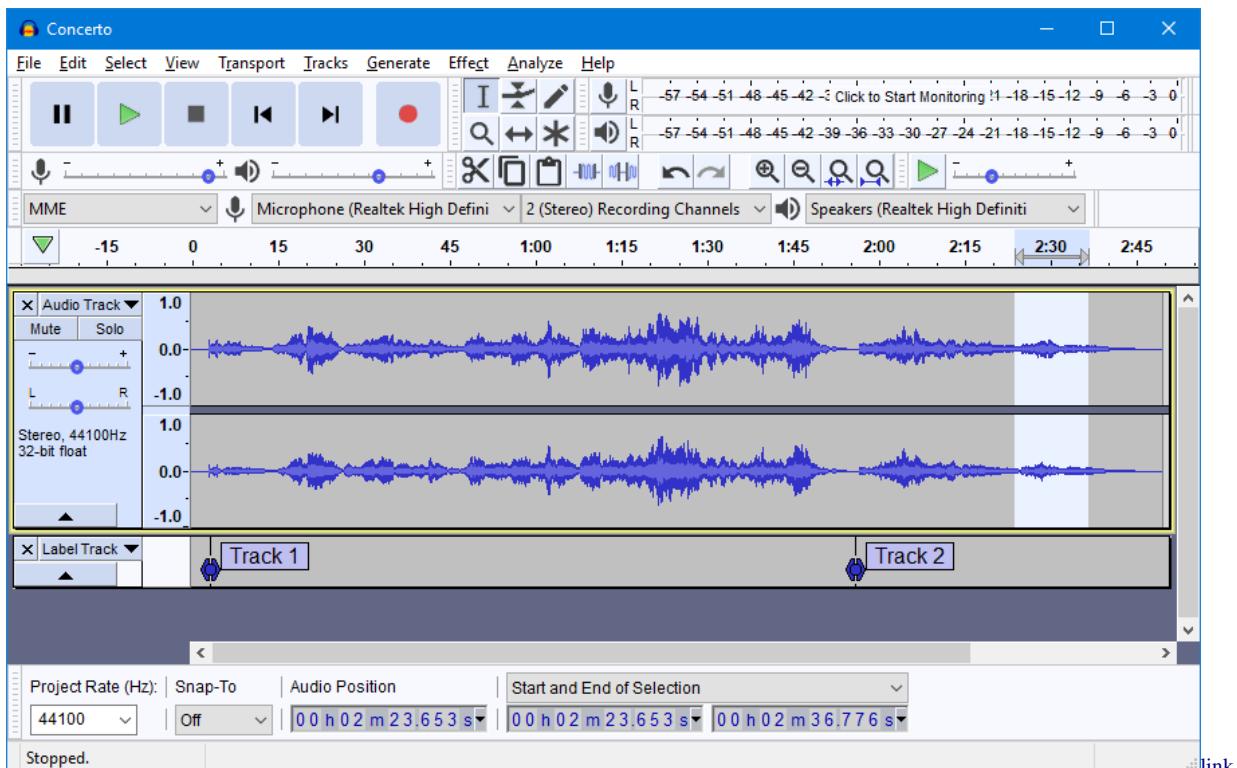
Si necesitáis convertir vuestro archivo de sonido a uno de estos formatos podeis hacerlo online:

<https://audio.online-convert.com/es/convertir-a-wav>



[link](#)

o usar programas como el Audacity (<https://www.audacityteam.org/>) que además os permitirán grabar y editar los archivos de audio.



En este página hay muchos sonidos que nos pueden valer para nuestra maqueta:

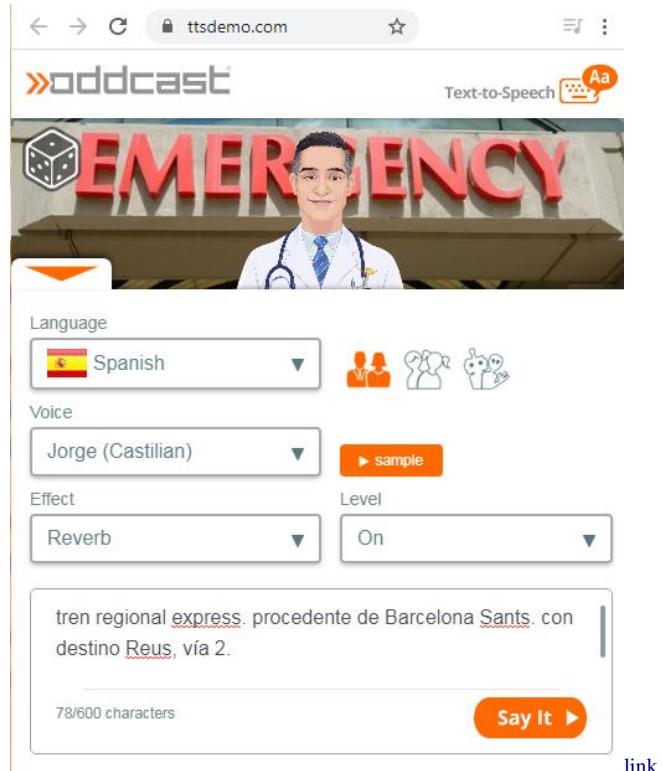
<https://www.itproducts.com/GL.html>

También podéis encontrar diferentes archivos de sonido que se pueden usar en el banco de sonidos del Ministerio de Educación

<http://recursostic.educacion.es/bancoimagenes/web/>

Las megafonías, sino tenemos una grabación original o queremos una frase en concreto, las podemos crear usando convertidores de texto a dicción (Text to Speech) como este online:

<https://ttsdemo.com/>



19. Crear librería: Decodificador DCC para señales RENFE

La circulación de los trenes reales viene condicionada por la señalización, por lo que añadirla a nuestra maqueta proporciona más realismo, pero la sencilla señalización habitual con luces roja y verde no proporciona el mismo sabor a los entendidos que una más acorde con señales de tres luces o cuatro luces habituales de la señalización RENFE.

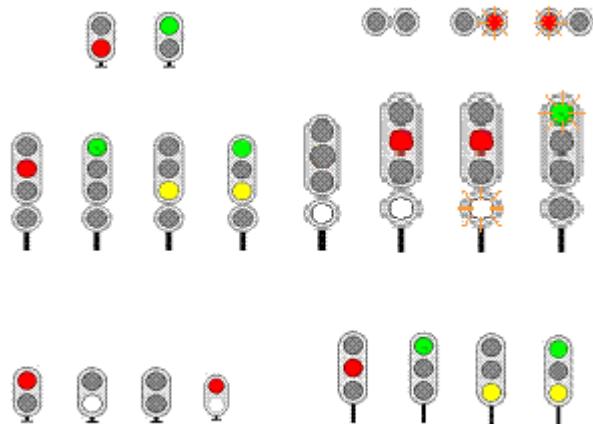
Afortunadamente se dispone en los comercios de señales con varias luces tal como se dispone sobre la vía real por lo que solo queda poder mostrar los diferentes aspectos para sacar todo el provecho a las mismas.



La idea

Hacer un decodificador de accesorios DCC para que pueda mostrar varios aspectos en una o varias señales de varias luces acorde con el Reglamento General de Circulación de Renfe.

Se tiene que tener la posibilidad de adaptar el código para que se puedan usar diferentes números y configuraciones de señales en un Arduino, por lo que habría que desarrollar una librería para que se pueda adaptar fácilmente los diferentes tipos de señales.



Creación de librerías

Las librerías suelen estar formadas por dos archivos con el mismo nombre, un archivo de cabecera .h (header) con las definiciones y prototipos de las funciones de las librerías y un archivo .cpp con el código de las diferentes funciones.

En el archivo .h lo primero es escribir unas directivas para el compilador para que la librería solo se incluya una vez. Para ello comprobamos con `#ifndef` sino esta ya definida una etiqueta. Si está definida lo que haya entre `#ifndef` y `#endif` no se compilará. En caso contrario la definimos. También tenemos que incluir la librería `Arduino.h` para usar las funciones de Arduino en nuestra librería.

miLibreria.h

```
#ifndef MI_LIBRERIA_H
#define MI_LIBRERIA_H

#include <Arduino.h>

// Aqui Estructuras, variables, clases, prototipos de la libreria

#endif
```

En caso de trabajar con objetos se ha de definir una clase (con `class`) y los prototipos de sus funciones (métodos), así como las variables de uso interno tanto públicas como privadas que escribiremos en el archivo .cpp. Las públicas son las que puede llamar el usuario desde el `sketch` y las privadas solo se pueden llamar desde el archivo .cpp de nuestra librería.

Hay que definir el prototipo de una función con el mismo nombre del objeto (con o sin parámetros) que es constructor del objeto (observad que no tiene tipo de datos) seguida de los prototipos de las funciones que creemos para trabajar con ese objeto/clase.

```
class miObjeto {           // creacion una clase llamada miObjeto
public:
  // Constructor. Se llama al crear el objeto
  miObjeto ();

  // Metodos.
  int miMetodo ();
  int miMetodo (int valor);

private:
  int _miVariable;
};
```

El constructor se ejecuta cuando definimos un objeto de esta clase en el sketch, por ejemplo en la librería `Servo.h`

```
Servo miServo;
```

Los métodos que trabajan con ella se llaman con el objeto separados por un `.` como por ejemplo:

```
miServo.attach()
```

Puede haber dos funciones (métodos) con el mismo nombre pero diferentes argumentos, a esto se llama sobrecarga de funciones, el compilador elegirá la adecuada.

También se permite tener valores por defecto para los parámetros. Esto supone que, si no se pasa el parámetro correspondiente, se asume el valor predefinido (han de ser los últimos en la lista de parámetros). La forma de indicarlo es declararlo en el prototipo de la función.

En la parte privada van las funciones o variables que usa nuestra librería internamente pero que no queremos que sean accesibles al usuario.

Por convención los nombres de las variables privadas globales de la clase se suelen preceder de `_` aunque no es algo obligatorio.

En el archivo .cpp (o archivos) se escriben las funciones para la librería. Para objetos se escribe el nombre de la clase `::` y el nombre de la función que trabaja con ese objeto.

Se ha de usar la directiva `#include` para incluir los archivos de cabecera `Arduino.h` y el de nuestra librería. Las `<>` indican al Arduino IDE que busque el archivo .h en el directorio de las librerías instaladas, y las `" "` que lo busque en el directorio donde está el archivo.

La función constructor tiene el mismo nombre que la clase del objeto y sirve para inicializarlo.

Archivo miLibreria.cpp

```
#include <Arduino.h>
#include "miLibreria.h"
miObjeto::miObjeto () {
// Lo necesario para inicializar el objeto
_miVariable = 0;
}
int miObjeto::miMetodo (int valor) {
    _miVariable = valor;
}

int miObjeto::miMetodo () {
    miMetodo (0);
}
```

En el sketch tenemos que incluir nuestra librería, definimos un objeto de la clase y ya podremos usar las funciones definidas en la clase para ese objeto.

Archivo miPrograma.ino

```
#include "miLibreria.h"
miObjeto demo;
void setup() {
}
void loop() {
demo.miMetodo();
}
```

Referencias

Existen dos formas de pasar argumentos a las funciones: por valor y por referencia. El primero es el utilizado con la declaración usual de parámetros.

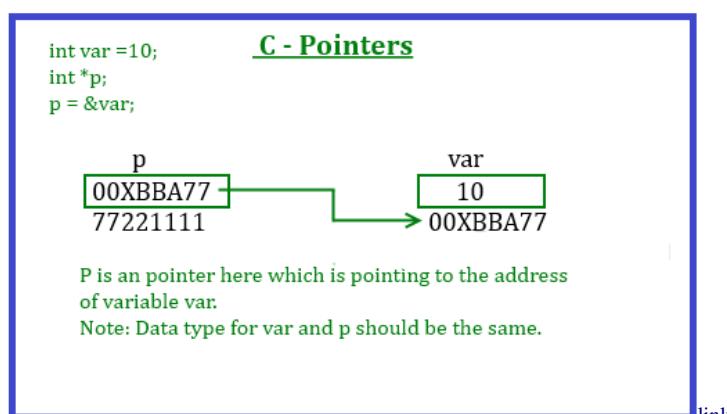
En el paso "por valor", se crean copias de los argumentos para pasarlo a la función por lo que una función no puede alterar ninguna variable pasada como argumento ya que trabaja con las copias.

Las referencias (punteros) permiten a la función modificar los valores de los argumentos.

El operador `&` es usado para pasar la dirección de una variable. Si `var` es una variable `&var` representa la dirección (puntero) de la variable `var`.

De igual modo el operador `*` es usado para acceder al valor de una variable. Si `p` es un puntero, entonces `*p` representa el valor contenido en la dirección apuntada por el.

El tipo de datos de la variable y el puntero han de coincidir. Para definir un puntero se añade `*` al inicio del nombre



El operador `->` permite acceder a los miembros de una estructura cuando es pasada como puntero.

`Objeto.miembro`

`PunteroObjeto->miembro`

Como una función no puede devolver más de un valor usando `return()`, podemos usar una estructura con los valores adecuados que será pasada a la función por referencia, de forma que sus miembros podrán ser modificados desde el cuerpo de la función.



Cuando se pasa por referencia, si `fillCup()` llena la taza (modifica el valor de la variable) también llena la original (porque es una referencia). Cuando se pasa por valor, no lo hace.

El objeto

En esta práctica, nuestro objeto es el Semaforo, que es una clase que tendrá una estructura de datos para describir las características del mismo (luces, pin, estado, brillo,...) y unos métodos para acceder e interactuar con esta estructura.

Partiremos de una estructura que describa cada una de las luces de la señal semáforica con la que crearemos un array genérico con el numero de luces máxima a controlar, dado que serán las señales tipo RENFE elegiremos 4 luces como máximo a controlar.

Para dar realismo a la señal queremos que se encienda y apague progresivamente por lo que definiremos unas variables para poder hacer un control PWM por software y así controlar el brillo de la luz.

```
#define NUMLEDS 4                                     // Numero maximo de LED por señal

struct LED {                                         // Estructura de control de un LED
    int pinLED;                                      // Pin al que esta conectado. -1 para sin conexion
    int estado;                                       // estado del LED
    int currPWM;                                      // Brillo actual
    int finPWM;                                       // Brillo final
    unsigned long interval;                           // Intervalo para el temporizador
    unsigned long timerPWM;                          // Temporizador
    bool fasePWM;                                     // Fase del control PWM
};

LED Leds[NUMLEDS];                                // Array de control de LED
```

El constructor del objeto simplemente tiene que poner valores por defecto al array de esta estructura, como que aún no esté definida la conexión con un pin, si la luz se encuentra encendida, apagada o parpadea, etc.

```
enum estados {OFF, ON, FLASH_A, FLASH_B};           // Estado del LED

// Constructor. Se llama al crear el objeto
Semaforo::Semaforo () {
    for (int i = 0; i < NUMLEDS; i++) {                // Inicializa array de LED
        Leds[i].pinLED = -1;
        Leds[i].estado = OFF;
        Leds[i].currPWM = 0;
        Leds[i].interval = 100;
        Leds[i].timerPWM = millis();
    }
}
```

Los métodos

Necesitamos diferentes funciones para trabajar con el objeto, entre ellas una que nos permita indicar los pines a los que tenemos conectadas las diferentes luces del semáforo. Como podemos definir 4 luces como máximo (luces Roja, Verde, Amarilla y Blanca) usaremos el valor -1 para indicar que una luz no tiene conexión a un pin determinado.

Sabemos que cómo mínimo habrá dos luces por lo que podemos poner los otros parámetros a un valor por defecto de -1 sino los definimos al llamar a la función.

```
enum color { ROJO, VERDE, AMARILLO, BLANCO};           // Indices del array segun color

void Semaforo::init (int pinRojo, int pinVerde, int pinAmarillo = -1, int pinBlanco = -1) {
    Leds[ROJO].pinLED = pinRojo;
    Leds[VERDE].pinLED = pinVerde;
    Leds[AMARILLO].pinLED = pinAmarillo;
    Leds[BLANCO].pinLED = pinBlanco;
    if (pinRojo != -1) pinMode (pinRojo, OUTPUT);
    if (pinVerde != -1) pinMode (pinVerde, OUTPUT);
    if (pinAmarillo != -1) pinMode (pinAmarillo, OUTPUT);
    if (pinBlanco != -1) pinMode (pinBlanco, OUTPUT);
}
```

Por ejemplo, para una señal de dos luces:

```
Semaforo2Luces.init (10, 11, -1, -1);
```

Aunque por sencillez también se podría llamar a nuestro método init para indicar los pines de diferente manera según tenga dos, tres o cuatro luces:

```
Sem2Luces.init (0, 1);
Sem3Luces.init (3, 4, 5);
Sem4Luces.init (6, 7, 8, 9);
```

Para facilitarnos la lectura del código podemos definir un método concreto para definir tipos especiales de señal como la de maniobras o la de paso a nivel.

```
void Semaforo::initManiobra (int pinRojo, int pinBlanco) {
    init (pinRojo, -1, -1, pinBlanco);
}

void Semaforo::initPasoNivelVehiculos (int pinDerecha, int pinIzquierda) {
    init (pinDerecha, pinIzquierda, -1, -1);
}
```

Se podría llamar a estos métodos para definir este tipo de señales así:

```
Sem2LucesManiobras.initManiobra(10, 11);
Sem2LucesPasoNivel.initPasoNivelVehiculos (13, 12);
```

Otro método indispensable es el que nos permite indicar que aspecto queremos mostrar en la señal, que simplemente coloca los valores adecuados en el array según el valor de `aspect` que se le pase como parámetro:

```
// Aspectos RENFE
enum aspectos { PARADA, VIA_LIBRE, ANUNCIO_PRECAUCION, ANUNCIO_PARADA, REBASE_AUTORIZADO,
                REBASE_AUTORIZADO_NO_PARAR, MOVIMIENTO_AUTORIZADO,
                VIA_LIBRE_CONDICIONAL, ANUNCIO_PARADA_INMEDIATA,
                PARADA_SELECTIVA, PASO_NIVEL_ABIERTO, PASO_NIVEL_CERRADO
};

void Semaforo::aspecto (byte aspect) { // Selecciona aspecto de la señal
    switch (aspect) {                                // LED rojo segun aspecto
        case PARADA:
        case REBASE_AUTORIZADO:
        case REBASE_AUTORIZADO_NO_PARAR:
            Leds[ROJO].estado = ON;
            break;
        case PASO_NIVEL_CERRADO:
            Leds[ROJO].estado = FLASH_A;
            break;
        default:
            Leds[ROJO].estado = OFF;
            break;
    }
}
```

```

switch (aspect) {                                     // LED verde segun aspecto
    case VIA_LIBRE:
    case ANUNCIO_PRECAUCION:
        Leds[VERDE].estado = ON;
        break;
    case PASO_NIVEL_CERRADO:
    case VIA_LIBRE_CONDICIONAL:
        Leds[VERDE].estado = FLASH_B;
        break;
    default:
        Leds[VERDE].estado = OFF;
        break;
}
switch (aspect) {                                     // LED amarillo segun aspecto
    case ANUNCIO_PARADA:
    case ANUNCIO_PRECAUCION:
        Leds[AMARILLO].estado = ON;
        break;
    case ANUNCIO_PARADA_INMEDIATA:
        Leds[AMARILLO].estado = FLASH_A;
        break;
    default:
        Leds[AMARILLO].estado = OFF;
        break;
}
switch (aspect) {                                     // LED blanco segun aspecto
    case REBASE_AUTORIZADO:
    case MOVIMIENTO_AUTORIZADO:
        Leds[BLANCO].estado = ON;
        break;
    case REBASE_AUTORIZADO_NO_PARAR:
        Leds[BLANCO].estado = FLASH_A;
        break;
    default:
        Leds[BLANCO].estado = OFF;
        break;
}
}

```

Finalmente necesitamos otro método que nos vaya procesando los datos y actualizando las luces en función de su estado, brillo, parpadeo, etc. según este definido en el array del objeto.

Esta función se llamará en el **loop()** lo más frecuentemente posible para que las luces realicen los efectos necesarios (encendido, brillo, parpadeo) si tienen definido un pin para la luz.

Necesitaremos unas variables para los parpadeos (**fase** y **tiempo**). Cuando pase el intervalo de tiempo definido cambiaremos la fase, posteriormente se actualizará el estado de la luz según la fase.

Para controlar el brillo de una luz usaremos el modo PWM (modulación por ancho de pulso) por lo que se tendrá que actualizar el brillo actual según el brillo final que queramos conseguir entre 0 para luz apagada y un valor máximo de PWM.

Si el brillo actual es mayor que 0 y menor que el valor máximo se actualizará la salida del pin según los datos contenidos en la estructura del objeto para esa luz y el tiempo actual.

```

#define PWM 20 // Periodo PWM. Influye en el tiempo de encendido/apagado
#define FLASH_INTERVAL 1000 // Periodo de tiempo para intermitencias

unsigned long _flashTime;                           // Temporizadores
unsigned long _currTime;
bool _flashFase;
void Semaforo::process () {                         // procesa estado de la señal
    _currTime = millis();
    if (_currTime - _flashTime > FLASH_INTERVAL) { // actualiza temporizador parpadeos
        _flashTime = _currTime;
        _flashFase = ! _flashFase;
    }
    for (int i = 0; i < NUMLEDS; i++) {           // controla los LED
        if (Leds[i].pinLED != -1) {                // si tiene definido un pin
            switch (Leds[i].estado) { // segun su estado actualiza brillo final
                case OFF:
                    Leds[i].finPWM = 0;
                    break;
                case ON:
                    Leds[i].finPWM = PWM;
                    break;
            }
        }
    }
}

```

```

        case FLASH_A:
            if (_flashFase)
                Leds[i].finPWM = 0;
            else
                Leds[i].finPWM = PWM;
            break;
        case FLASH_B:
            if (_flashFase)
                Leds[i].finPWM = PWM;
            else
                Leds[i].finPWM = 0;
            break;
    }
    fadePin (&Leds[i], _currTime);           // Controla brillo
}
}
}

```

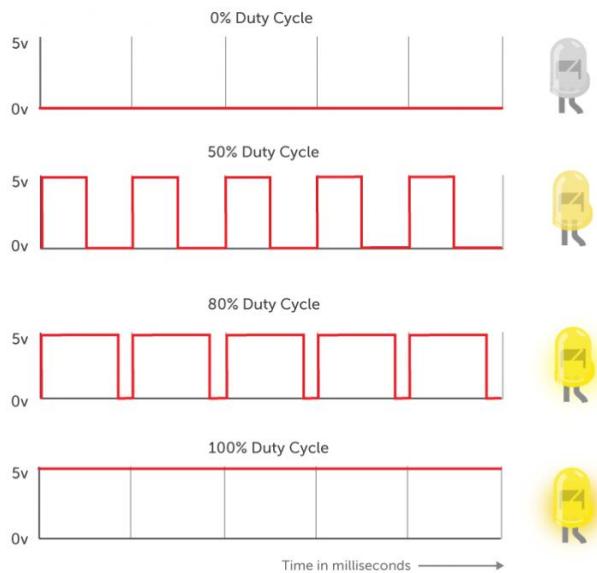
Definiremos una función `fadePin()` será la encargada de controlar el brillo por PWM de acuerdo a los datos contenidos en la estructura. Esta función los actualizará así que no podemos pasar el parámetro por valor ya que los cambios se perderían, pasaremos la estructura por referencia , observad el `&` que indica que se pasa el puntero y no el valor.

Nota: Con el valor 20 en PWM da un buen efecto de encendido/apagado lento, si se aumenta mucho el valor tiende a temblar cuando cambia el brillo

Controlando el brillo

El Arduino Uno posee 6 salidas que se pueden usar con PWM y la instrucción `analogWrite()` para controlar el brillo de las luces, pero son insuficientes para nuestro propósito, así que lo implementaremos por software.

Se trata de generar continuamente una señal en el pin de forma que dentro de un periodo determinado de tiempo la salida este encendida un tiempo y luego se apague, así se consigue que el brillo de la luz varíe en función de la relación entre el tiempo encendido y el apagado (*Duty Cycle*).



[link](#)

Para generar la señal en el pin, tenemos que esperar a que pase el intervalo de tiempo (periodo o el duty cycle) para un cambio en la señal de salida, para ello comparamos los tiempos que tenemos en la estructura para esta luz con el tiempo actual que nos han pasado como parámetro.

Observad que la estructura se pasa por referencia (`*led` en los parámetros) por lo que para acceder a los valores usamos `->`

Según el brillo actual generamos la señal de salida. Si el brillo actual es 0, la luz esta apagada así que ponemos el pin a **APAGADO** (lo hemos definido como **LOW** si usamos señales de LED con ánodo común que funcionen a 5V o usamos un ULN2803, pero lo podemos definir a **HIGH** para señales a 5V con cátodo común)

Actualizamos el intervalo de tiempo y si el brillo final no es 0, aumentamos el valor del brillo actual para el próximo intervalo.

Procedemos de forma similar si el brillo es el máximo, en este caso ponemos el pin a **ENCENDIDO** y si el brillo final es menor, decrementamos el brillo actual para el próximo intervalo.

Si es un brillo intermedio tenemos que generar la señal PWM mediante la función **fadePin()**, para ello comprobamos en qué fase de la señal PWM estamos y si es la primera fase encendemos la luz el intervalo de tiempo de brillo actual, mientras que si es la segunda fase lo apagamos el tiempo restante al periodo, además en esta fase comprobamos si el brillo final es mayor o menor que el actual y actualizamos el brillo actual para el próximo periodo. Finalmente cambiamos la fase.

```
#define ENCENDIDO HIGH // Tension pin. Cambiar segun anodo/catodo comun
#define APAGADO LOW

void Semaforo::fadePin (LED *led, unsigned long currTime) { // Controla brillo de un LED
    if (currTime - led->timerPWM > led->interval) { // espera a que pase el intervalo
        led->timerPWM = currTime;
        switch (led->currPWM) {
            case 0: // Brillo 0: Apaga LED
                digitalWrite (led->pinLED, APAGADO);
                led->interval = 100;
                if (led->finPWM != 0) // Si se tiene que encender cambia brillo
                    led->currPWM++;
                break;
            case PWM: // Brillo maximo. Enciende LED
                digitalWrite (led->pinLED, ENCENDIDO);
                led->interval = 100;
                if (led->finPWM < PWM) // Si se tiene que apagar cambia brillo
                    led->currPWM--;
                break;
            default: // Brillo intermedio. Control PWM del LED
                if (led->fasePWM) {
                    digitalWrite (led->pinLED, ENCENDIDO);
                    led->interval = led->currPWM;
                }
                else {
                    digitalWrite (led->pinLED, APAGADO);
                    led->interval = PWM - led->currPWM;
                    (led->currPWM < led->finPWM) ? led->currPWM++ : led->currPWM--; // Cambia brillo segun
brillo final
                }
                led->fasePWM = !led->fasePWM;
                break;
        }
    }
}
```

La librería

En el archivo .h definimos la clase **Semaforo** con **class** con las partes pública (la que queremos que pueda usar el usuario) y privada con su constructor, con sus métodos y variables en su lugar adecuado.

También ponemos las definiciones generales que podría cambiar el usuario, como el intervalo, periodo y nivel de salida.

Archivo Semaforo.h

```
// Libreria Semaforo.h -- Paco Cañada 2020 -- https://usuaris.tinet.cat/fmco/
#ifndef SEMAFORO_H
#define SEMAFORO_H

#include <Arduino.h>

// Estructuras, variables, clases, prototipos de la libreria

#define FLASH_INTERVAL 1000 // Periodo de tiempo para intermitencias
#define PWM 20 // Periodo PWM. Influye en el tiempo de
encendido/apagado

#define ENCENDIDO HIGH // Tension pin. Cambiar segun anodo/catodo comun
#define APAGADO LOW
```

```

enum aspectos { PARADA, VIA_LIBRE, ANUNCIO_PRECAUCION, // Aspectos RENFE
    ANUNCIO_PARADA, REBASE_AUTORIZADO,
    REBASE_AUTORIZADO_NO_PARAR, MOVIMIENTO_AUTORIZADO,
    VIA_LIBRE_CONDICIONAL, ANUNCIO_PARADA_INMEDIATA,
    PARADA_SELECTIVA, PASO_NIVEL_ABIERTO, PASO_NIVEL_CERRADO
};

#define NUMLEDS 4 // Numero maximo de LED por señal

class Semaforo {
public:
    // Constructor. Se llama al crear el objeto
    Semaforo ();

    // Metodos que puede llamar el usuario

    // Inicializacion de los pines usados por los semaforos
    void init (int pinRojo, int pinVerde, int pinAmarillo = -1, int pinBlanco = -1) ;
    void initManiobra (int pinRojo, int pinBlanco);
    void initPasoNivelVehiculos (int pinDerecha, int pinIzquierda);

    void process (); // Procesa el semaforo. Llamar lo mas a menudo posible
    void aspecto (byte aspect); // Cambia el aspecto mostrado en el semaforo

private:
    // Son invisibles para el usuario
    unsigned long _flashTime; // Temporizadores
    unsigned long _currTime; // Fase actual del parpadeo
    bool _flashFase;

    struct LED { // Estructura de control de un LED
        int pinLED; // Pin al que esta conectado. -1 para sin conexion
        int estado; // estado del LED
        int currPWM; // Brillo actual
        int finPWM; // Brillo final
        unsigned long interval; // Intervalo para el temporizador
        unsigned long timerPWM; // Temporizador
        bool fasePWM; // Fase del control PWM
    };
    LED Leds[NUMLEDS]; // Array de control de LED

    enum color {ROJO, VERDE, AMARILLO, BLANCO}; // Indices del array segun color
    enum estados {OFF, ON, FLASH_A, FLASH_B}; // Estado del LED

    void fadePin (LED *led, unsigned long currTime); // Controla el encendido/apagado progresivo del LED
};

#endif

```

En el archivo .cpp pondremos todos los métodos y funciones de nuestra librería.

Archivo Semaforo.cpp

```

// Libreria Semaforo.h -- Paco Cañada 2020 -- https://usuaris.tinet.cat/fmco/
#ifndef include <Arduino.h>
#include "Semaforo.h"

// Constructor. Se llama al crear el objeto
Semaforo::Semaforo () {
    for (int i = 0; i < NUMLEDS; i++) { // Inicializa array de LED
        Leds[i].pinLED = -1;
        Leds[i].estado = OFF;
        Leds[i].currPWM = 0;
        Leds[i].interval = 100;
        Leds[i].timerPWM = millis();
    }
}

void Semaforo::init (int pinRojo, int pinVerde, int pinAmarillo = -1, int pinBlanco = -1) {
    Leds[ROJO].pinLED = pinRojo;
    Leds[VERDE].pinLED = pinVerde;
    Leds[AMARILLO].pinLED = pinAmarillo;
    Leds[BLANCO].pinLED = pinBlanco;
    if (pinRojo != -1) pinMode (pinRojo, OUTPUT);
    if (pinVerde != -1) pinMode (pinVerde, OUTPUT);
    if (pinAmarillo != -1) pinMode (pinAmarillo, OUTPUT);
    if (pinBlanco != -1) pinMode (pinBlanco, OUTPUT);
}

void Semaforo::initManiobra (int pinRojo, int pinBlanco) {
    init (pinRojo, -1, -1, pinBlanco);
}

```

```

void Semaforo::initPasoNivelVehiculos (int pinDerecha, int pinIzquierda) {
    init (pinDerecha, pinIzquierda, -1, -1);
}

void Semaforo::process () {                                     // procesa estado de la señal
    _currTime = millis();                                     // actualiza temporizador parpadeos
    if (_currTime - _flashTime > FLASH_INTERVAL) {
        _flashTime = _currTime;
        _flashFase = !_flashFase;
    }
    for (int i = 0; i < NUMLEDS; i++) {
        if (Leds[i].pinLED != -1) {                           // controla los LED
            switch (Leds[i].estado) {                         // si tiene definido un pin
                case OFF:                                     // segun su estado actualiza brillo final
                    Leds[i].finPWM = 0;
                    break;
                case ON:
                    Leds[i].finPWM = PWM;
                    break;
                case FLASH_A:
                    if (_flashFase)
                        Leds[i].finPWM = 0;
                    else
                        Leds[i].finPWM = PWM;
                    break;
                case FLASH_B:
                    if (_flashFase)
                        Leds[i].finPWM = PWM;
                    else
                        Leds[i].finPWM = 0;
                    break;
                }
            fadePin (&Leds[i], _currTime);                   // Controla brillo
        }
    }
}

void Semaforo::fadePin (LED *led, unsigned long currTime) { // Controla brillo de un LED
    if (currTime - led->timerPWM > led->interval) {       // espera a que pase el intervalo
        led->timerPWM = currTime;
        switch (led->currPWM) {
            case 0:                                         // Brillo 0: Apaga LED
                digitalWrite (led->pinLED, APAGADO);
                led->interval = 100;
                if (led->finPWM != 0)
                    led->currPWM++;
                break;
            case PWM:                                       // Brillo maximo. Enciende LED
                digitalWrite (led->pinLED, ENCENDIDO);
                led->interval = 100;
                if (led->finPWM < PWM)
                    led->currPWM--;
                break;
            default:                                      // Brillo intermedio. Control PWM del LED
                if (led->fasePWM) {
                    digitalWrite (led->pinLED, ENCENDIDO);
                    led->interval = led->currPWM;
                }
                else {
                    digitalWrite (led->pinLED, APAGADO);
                    led->interval = PWM - led->currPWM;
                    (led->currPWM < led->finPWM) ? led->currPWM++ : led->currPWM--; // Cambia brillo segun brillo final
                }
            led->fasePWM = !led->fasePWM;
            break;
        }
    }
}

void Semaforo::aspecto (byte aspect) {                         // Selecciona aspecto de la señal
    switch (aspect) {                                         // LED rojo segun aspecto
        case PARADA:
        case REBASE_AUTORIZADO:
        case REBASE_AUTORIZADO_NO_PARAR:
            Leds[ROJO].estado = ON;
            break;
        case PASO_NIVEL_CERRADO:
            Leds[ROJO].estado = FLASH_A;
            break;
        default:
            Leds[ROJO].estado = OFF;
            break;
    }
    switch (aspect) {                                         // LED verde segun aspecto
        case VIA_LIBRE:
        case ANUNCIO_PRECAUCION:
            Leds[VERDE].estado = ON;
            break;
    }
}

```

```

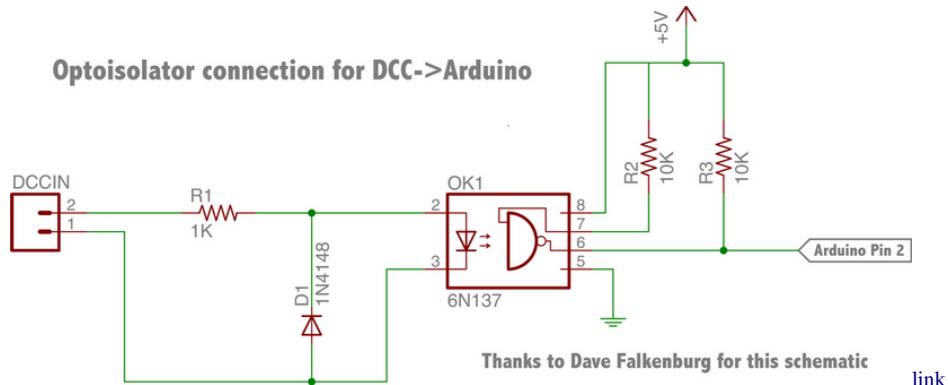
        case PASO_NIVEL_CERRADO:
        case VIA_LIBRE_CONDICIONAL:
            Leds[VERDE].estado = FLASH_B;
            break;
        default:
            Leds[VERDE].estado = OFF;
            break;
    }
    switch (aspect) {
        case ANUNCIO_PARADA:
        case ANUNCIO_PRECAUCION:
            Leds[AMARILLO].estado = ON;
            break;
        case ANUNCIO_PARADA_INMEDIATA:
            Leds[AMARILLO].estado = FLASH_A;
            break;
        default:
            Leds[AMARILLO].estado = OFF;
            break;
    }
    switch (aspect) {
        case REBASE_AUTORIZADO:
        case MOVIMIENTO_AUTORIZADO:
            Leds[BLANCO].estado = ON;
            break;
        case REBASE_AUTORIZADO_NO_PARAR:
            Leds[BLANCO].estado = FLASH_A;
            break;
        default:
            Leds[BLANCO].estado = OFF;
            break;
    }
}

```

Usando la librería

Como ejemplo de uso de nuestra librería vamos a realizar un decodificador de accesorios DCC que controlará una señal de dos luces, una de tres luces, una de cuatro luces, una señal baja de maniobras y una señal de paso a nivel usando varias direcciones de accesorios DCC.

Para ello usaremos el ya conocido circuito con el 6N137 para llevar la señal DCC al pin **D2** del Arduino y la librería **NmraDcc.h**



Lo primero es incluir las librerías y definir los objetos. Definiremos un objeto para cada señal, de esta forma se ejecutará su constructor.

```

#include "Semaforo.h"
#include <NmraDcc.h>

NmraDcc Dcc; // Objetos librerias

Semaforo Sem2Luces;
Semaforo Sem3Luces;
Semaforo Sem4Luces;
Semaforo Sem2LucesManiobras;
Semaforo Sem2LucesPasoNivel;

```

Observad que la librería **Semaforo.h** esta con “ “ indicando que los archivos .h y .cpp se encuentran en el mismo directorio que el sketch (.ino) mientras que la librería **Nmra.h** esta con < > indicando que es una de las librerías instaladas en el Arduino IDE.

En el **setup()** inicializamos los objetos con su método correspondiente. Las señales las inicializaremos indicando los pines a las que están conectadas las diferentes luces y el objeto DCC indicando el pin de señal DCC y el modo de accesorios como en sus ejemplos.

```
const byte DCC_PIN = 2;           // pin señal DCC

Sem2Luces.init (0, 1); // define los semáforos y sus conexiones
Sem3Luces.init (3, 4, 5);
Sem4Luces.init (6, 7, 8, 9);
Sem2LucesManiobras.initManiobra(10, 11);
Sem2LucesPasoNivel.initPasoNivelVehiculos (13, 12);

Dcc.pin (digitalPinToInterrupt (DCC_PIN), DCC_PIN, 1); // Inicializa DCC
Dcc.initAccessoryDecoder (MAN_ID_DIY, 1, FLAGS_OUTPUT_ADDRESS_MODE, 0);
```

Cuando se reciba una orden DCC de accesorios se comprobará que la dirección del accesorio es una de las que controlará nuestro decodificador a partir de una dirección base que esta programada en las CV1 y CV9. Si la CV8 (**CV_MANUFACTURER_ID**) no es 13 (**MAN_ID_DIY**) las inicializaremos a la dirección 1 por defecto como base.

```
int leeDireccionDcc() {
    int direccion, cv1, cv9;

    if (Dcc.getCV (CV_MANUFACTURER_ID) != MAN_ID_DIY) { // Si CV8 no es 13, resetea CV
        Dcc.setCV (CV_ACCESSORY_DECODER_ADDRESS_LSB, 1);
        Dcc.setCV (CV_ACCESSORY_DECODER_ADDRESS_MSB, 0);
        Dcc.setCV (CV_MANUFACTURER_ID, MAN_ID_DIY);
    }
    cv1 = Dcc.getCV (CV_ACCESSORY_DECODER_ADDRESS_LSB);
    cv9 = Dcc.getCV (CV_ACCESSORY_DECODER_ADDRESS_MSB);
    direccion = (cv9 << 8) + cv1;
    return (direccion);
}
```

Esta dirección base la leeremos en el **setup()**

```
int dccBase; // dirección base DCC
dccBase = leeDireccionDcc(); // Lee dirección base DCC
```

y si cambia una CV usando la función **notifyDccCVChange()** de la librería.

```
void notifyDccCVChange( uint16_t CV, uint8_t Value) {
    dccBase = leeDireccionDcc();
}
```

Al recibir una orden DCC de accesorios la librería **NmraDcc.h** ejecutará la función **notifyDccAccTurnoutOutput()** por lo que en ella tendremos que poner que aspecto queremos que muestre cada señal según la dirección DCC recibida si esta dentro del rango que controla nuestro decodificador comprobando la diferencia con la dirección base.

```
void notifyDccAccTurnoutOutput( uint16_t Addr, uint8_t Direction, uint8_t OutputPower ) {
    byte orden;

    if ((Addr >= dccBase) && (Addr <= dccBase + 11) && OutputPower) { // comprueba dirección dentro del rango
        orden = (((Addr - dccBase) << 1) + Direction); // orden de cambio de aspecto entre 0 y 21
        switch (orden) { // actua según orden
            case 0: // Dirección base: Semáforo dos luces
                Sem2Luces.aspecto (PARADA);
                break;
            case 1:
                Sem2Luces.aspecto (VIA_LIBRE);
                break;
            case 2: // Dirección base + 1/2: Semáforo 3 luces
                Sem3Luces.aspecto (PARADA);
                break;
            case 3:
                Sem3Luces.aspecto (VIA_LIBRE);
                break;
            case 4:
                Sem3Luces.aspecto (ANUNCIO_PRECAUCION);
                break;
        }
    }
}
```

```

case 5:
    Sem3Luces.aspecto (ANUNCIO_PARADA);
    break;

case 6: // Direccion base + 3/4/5/6/7: Semaforo 3 luces
    Sem4Luces.aspecto (PARADA);
    break;
case 7:
    Sem4Luces.aspecto (VIA_LIBRE);
    break;
case 8:
    Sem4Luces.aspecto (ANUNCIO_PRECAUCION);
    break;
case 9:
    Sem4Luces.aspecto (ANUNCIO_PARADA);
    break;
case 10:
    Sem4Luces.aspecto (VIA_LIBRE_CONDICIONAL);
    break;
case 11:
    Sem4Luces.aspecto (ANUNCIO_PARADA_INMEDIATA);
    break;
case 12:
    Sem4Luces.aspecto (-1); // apagado de todas las luces
    break;
case 13:
    Sem4Luces.aspecto (MOVIMIENTO_AUTORIZADO);
    break;
case 14:
    Sem4Luces.aspecto (REBASE_AUTORIZADO);
    break;
case 15:
    Sem4Luces.aspecto (REBASE_AUTORIZADO_NO_PARAR);
    break;
case 16: // Direccion base + 8/9: Semaforo 2 luces Maniobras
    Sem2LucesManiobras.aspecto (-1); // apagado de todas las luces
    break;
case 17:
    Sem2LucesManiobras.aspecto (MOVIMIENTO_AUTORIZADO);
    break;
case 18:
    Sem2LucesManiobras.aspecto (REBASE_AUTORIZADO);
    break;
case 19:
    Sem2LucesManiobras.aspecto (REBASE_AUTORIZADO_NO_PARAR);
    break;

case 20: // Direccion base + 10/11: Semaforo 2 paso a nivel
    Sem2LucesPasoNivel.aspecto (PASO_NIVEL_CERRADO);
    break;
case 21:
    Sem2LucesPasoNivel.aspecto (PASO_NIVEL_ABIERTO);
    break;
}

}
}

```

Para que todo se muestre correctamente (encendido/apagado lento, parpadeos, ...) y se vaya decodificando la señal DCC en el **loop()** tenemos que llamar a las funciones **process()** de cada objeto lo más a menudo posible.

```

void loop() {
    Dcc.process(); // Procesa paquetes DCC
    Sem2Luces.process(); // Procesa semaforos
    Sem3Luces.process();
    Sem4Luces.process();
    Sem2LucesManiobras.process();
    Sem2LucesPasoNivel.process();
}

```

Eso es todo, las librerías hacen el resto.

El programa

El programa es bastante sencillo ya que la mayor parte del trabajo la realizan las librerías y queda así.

Archivo decoSemaforos.ino

```
#include "Semaforo.h"           // Libreria Semaforo.h -- Paco Cañada 2020 -- https://usuaris.tinet.cat/fmco/
#include <NmraDcc.h>

NmraDcc Dcc;                      // Objetos librerias

Semaforo Sem2Luces;
Semaforo Sem3Luces;
Semaforo Sem4Luces;
Semaforo Sem2LucesManiobras;
Semaforo Sem2LucesPasoNivel;

const byte DCC_PIN = 2;             // pin señal DCC
int dccBase;                      // direccion base DCC

void setup() {
    Sem2Luces.init (0, 1);          // define los semaforos y sus conexiones
    Sem3Luces.init (3, 4, 5);
    Sem4Luces.init (6, 7, 8, 9);
    Sem2LucesManiobras.initManiobra(10, 11);
    Sem2LucesPasoNivel.initPasoNivelVehiculos (13, 12);

    Dcc.pin (digitalPinToInterrupt (DCC_PIN), DCC_PIN, 1);      // Inicializa DCC
    Dcc.initAccessoryDecoder (MAN_ID_DIY, 1, FLAGS_OUTPUT_ADDRESS_MODE, 0);
    dccBase = leeDireccionDcc();                                     // Lee direccion base DCC
}

void loop() {
    Dcc.process();                // Procesa paquetes DCC
    Sem2Luces.process();          // Procesa semaforos
    Sem3Luces.process();
    Sem4Luces.process();
    Sem2LucesManiobras.process();
    Sem2LucesPasoNivel.process();
}

int leeDireccionDcc() {
    int direccion, cv1, cv9;

    if (Dcc.getCV (CV_MANUFACTURER_ID) != MAN_ID_DIY) {           // Si CV8 no es 13, resetea CV
        Dcc.setCV (CV_ACCESSORY_DECODER_ADDRESS_LSB, 1);
        Dcc.setCV (CV_ACCESSORY_DECODER_ADDRESS_MSB, 0);
        Dcc.setCV (CV_MANUFACTURER_ID, MAN_ID_DIY);
    }
    cv1 = Dcc.getCV (CV_ACCESSORY_DECODER_ADDRESS_LSB);
    cv9 = Dcc.getCV (CV_ACCESSORY_DECODER_ADDRESS_MSB);
    direccion = (cv9 << 8) + cv1;
    return (direccion);
}

void notifyDccCVChange( uint16_t CV, uint8_t Value) {
    dccBase = leeDireccionDcc();
}

void notifyDccAccTurnoutOutput( uint16_t Addr, uint8_t Direction, uint8_t OutputPower ) {
    byte orden;

    if ((Addr >= dccBase) && (Addr <= dccBase + 11) && OutputPower) { // comprueba direccion dentro del rango
        orden = (((Addr - dccBase) << 1) + Direction);                  // orden de cambio de aspecto entre 0 y 21
        switch (orden) {                                                     // actua segun orden
            case 0:
                Sem2Luces.aspecto (PARADA);                                  // Direccion base: Semaforo dos luces
                break;
            case 1:
                Sem2Luces.aspecto (VIA_LIBRE);
                break;
            case 2:
                Sem3Luces.aspecto (PARADA);                                  // Direccion base + 1/2: Semaforo 3 luces
                break;
            case 3:
                Sem3Luces.aspecto (VIA_LIBRE);
                break;
            case 4:
                Sem3Luces.aspecto (ANUNCIO_PRECAUCION);
                break;
            case 5:
                Sem3Luces.aspecto (ANUNCIO_PARADA);
                break;
            case 6:
                Sem4Luces.aspecto (PARADA);                                  // Direccion base + 3/4/5/6/7: Semaforo 3 luces
                break;
        }
    }
}
```

```

        case 7:
            Sem4Luces.aspecto (VIA_LIBRE);
            break;
        case 8:
            Sem4Luces.aspecto (ANUNCIO_PRECAUCION);
            break;
        case 9:
            Sem4Luces.aspecto (ANUNCIO_PARADA);
            break;
        case 10:
            Sem4Luces.aspecto (VIA_LIBRE_CONDICIONAL);
            break;
        case 11:
            Sem4Luces.aspecto (ANUNCIO_PARADA_INMEDIATA);
            break;
        case 12:
            Sem4Luces.aspecto (-1);                                // apagado de todas las luces
            break;
        case 13:
            Sem4Luces.aspecto (MOVIMIENTO_AUTORIZADO);
            break;
        case 14:
            Sem4Luces.aspecto (REBASE_AUTORIZADO);
            break;
        case 15:
            Sem4Luces.aspecto (REBASE_AUTORIZADO_NO_PARAR);
            break;

        case 16:                                              // Direccion base + 8/9: Semaforo 2 luces Maniobras
            Sem2LucesManiobras.aspecto (-1);
            break;
        case 17:
            Sem2LucesManiobras.aspecto (MOVIMIENTO_AUTORIZADO);
            break;
        case 18:
            Sem2LucesManiobras.aspecto (REBASE_AUTORIZADO);
            break;
        case 19:
            Sem2LucesManiobras.aspecto (REBASE_AUTORIZADO_NO_PARAR);
            break;

        case 20:                                              // Direccion base + 10/11: Semaforo 2 paso a nivel
            Sem2LucesPasoNivel.aspecto (PASO_NIVEL_CERRADO);
            break;
        case 21:
            Sem2LucesPasoNivel.aspecto (PASO_NIVEL_ABIERTO);
            break;
    }
}
}

```

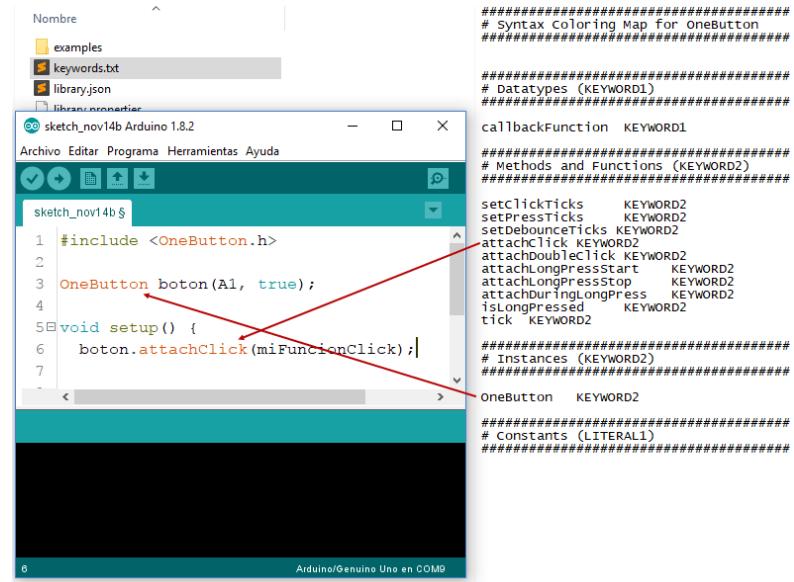
Ubicando

Los archivos de nuestra librería ([Semaforo.h](#) y [Semaforo.cpp](#)) los ponemos en el mismo directorio del sketch pero tendríamos que copiarlos en cada nuevo sketch que realicemos con esta librería. (Para usarla ponemos `#include "Semaforo.h"`)

Es mejor ubicarlos en el directorio de librerías del Arduino IDE, para ello se crea una carpeta llamada igual que nuestra librería ([Semaforo](#)) en el directorio ubicado en [Mis Documentos/Arduino/libraries/](#) y se copian en ella los archivos .h y .cpp (Ahora la usaríamos con `#include <Semaforo.h>`)

Al iniciar el Arduino IDE en el menú **Programa-> Incluir librería** se debería ver la librería [Semaforo](#) y ya se podrá usar como cualquier otra librería.

Coloreando



[link](#)

Para hacer que el Arduino IDE resalte en color las funciones de nuestra librería hay que crear un archivo llamado **keywords.txt** en el directorio de nuestra librería. Debe tener un aspecto como este:

Archivo keywords.txt

```
Semaforo KEYWORD1
init      KEYWORD2
initManiobra      KEYWORD2
initPasoNivelVehiculos  KEYWORD2
process   KEYWORD2
aspecto   KEYWORD2

PARADA    LITERAL1
VIA_LIBRE    LITERAL1
ANUNCIO_PRECAUCION  LITERAL1
ANUNCIO_PARADA  LITERAL1
REBASE_AUTORIZADO  LITERAL1
REBASE_AUTORIZADO_NO_PARAR  LITERAL1
MOVIMIENTO_AUTORIZADO  LITERAL1
VIA_LIBRE_CONDICIONAL  LITERAL1
ANUNCIO_PARADA_INMEDIATA LITERAL1
PARADA_SELECTIVA LITERAL1
PASO_NIVEL_ABIERTO  LITERAL1
PASO_NIVEL_CERRADO  LITERAL1
```

Cada línea tiene el nombre de la palabra clave, seguido de una tabulación (no espacios), seguida del tipo de palabra clave. Las clases deben ser **KEYWORD1**; las funciones deben ser **KEYWORD2** y las constantes deben ser **LITERAL1**. Para poner una linea de comentarios hay que comenzar la linea con **#**

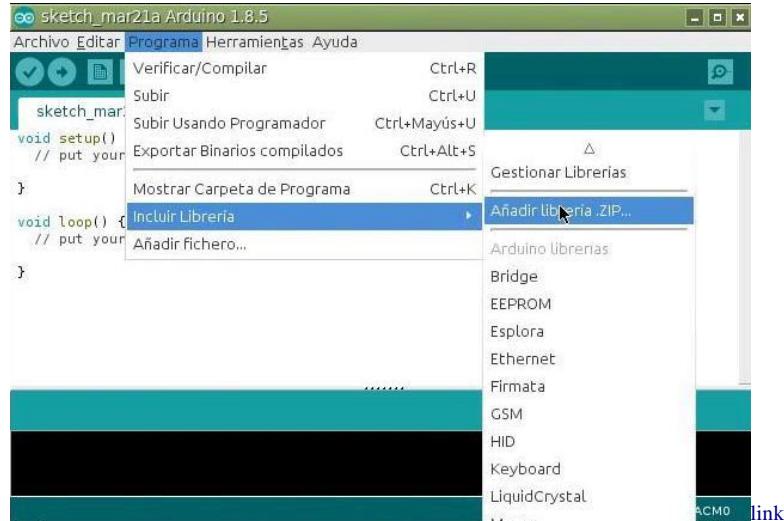
Tendrá que reiniciar el entorno Arduino para que reconozca las nuevas palabras clave.

Compartiendo

Tambien es buena idea incluir algunos ejemplos de uso de nuestra librería, para ello se crea un directorio **examples** en el directorio de nuestra librería y en el se guardan la carpetas que contienen nuestros sketch de ejemplo. Al iniciar el Arduino IDE en el menú Archivo-> Ejemplos encontraremos nuestra librería y los ejemplos que hayamos incluido.

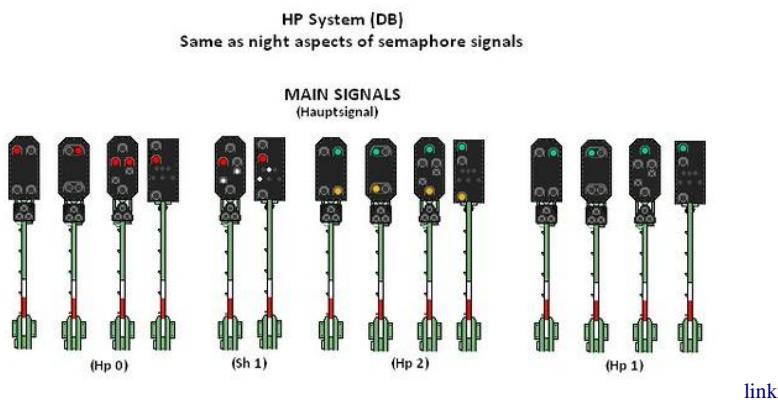
Para distribuir nuestra librería a nuestros compañeros de afición podemos hacer un archivo .zip del directorio de nuestra librería y enviárselo.

Ellos tendrán que importar la librería al Arduino IDE con el menú Programa-> Incluir Libreria -> Añadir libreria .ZIP... con lo que ya podrán usar nuestra librería en sus sketch y compilar los ejemplos que hayamos incluido.



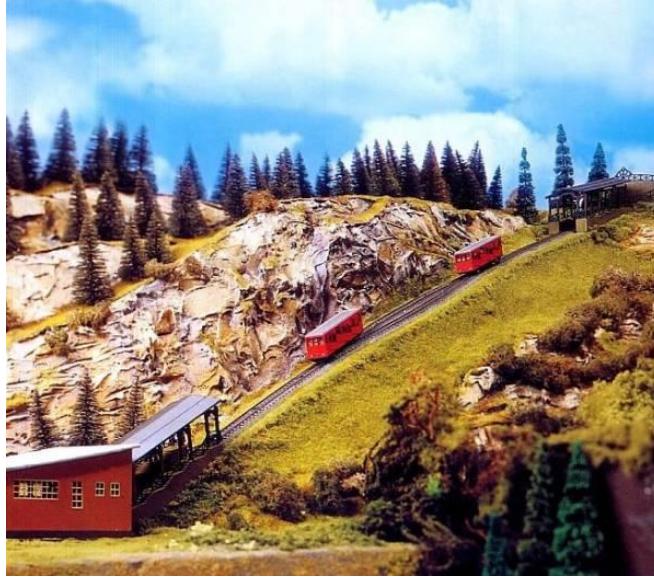
Yo voy a compartir esta con vosotros 😊, a la que he añadido los aspectos básicos de la señalización DB.
La tenéis aquí :

<https://usuaris.tinet.cat/fmco/cgi-bin/download.cgi?file=download/Semaforo.zip>



20. Máquina de estados: Control de un funicular

He conseguido de segunda mano un funicular de BRAWA (Standseilbahn 6410) de escala N lamentablemente no tenía el controlador para el mismo, se debió quedar en la maqueta del anterior dueño, así que necesito montar uno para ponerlo en marcha.



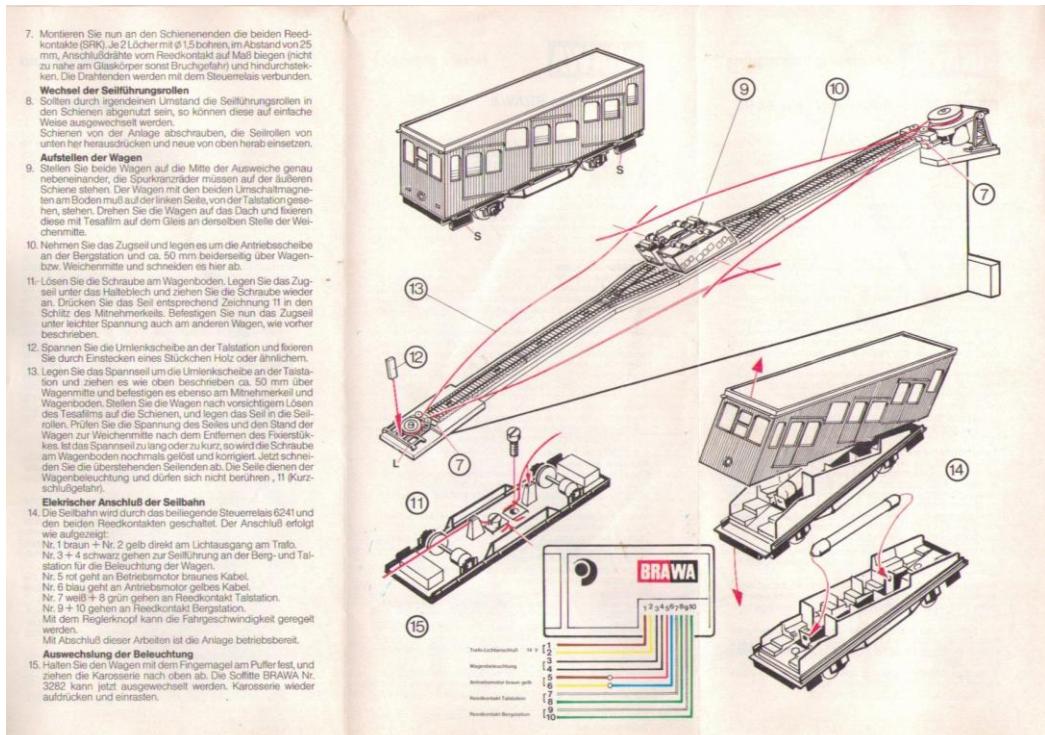
[link](#)

El funicular tiene un motor en una de las estaciones que hace girar una rueda tractora que mueve el cable que arrastra a las cabinas por la vía.

Este cable es conductor (de cobre) por lo que las cabinas reciben corriente para su iluminación, un polo llega de la estación superior y el otro polo de la estación inferior.

Una de las cabinas tiene unos imanes con los que activará una ampolla Reed situada en cada una de las estaciones para detener el movimiento y cambiar el sentido.

El mecanismo del motor lleva una leva interna que al cambiar el sentido de rotación del motor necesita que la rueda tractora de una vuelta para que el movimiento se transmita al cable tractor con lo que se simula el tiempo de parada en la estación.

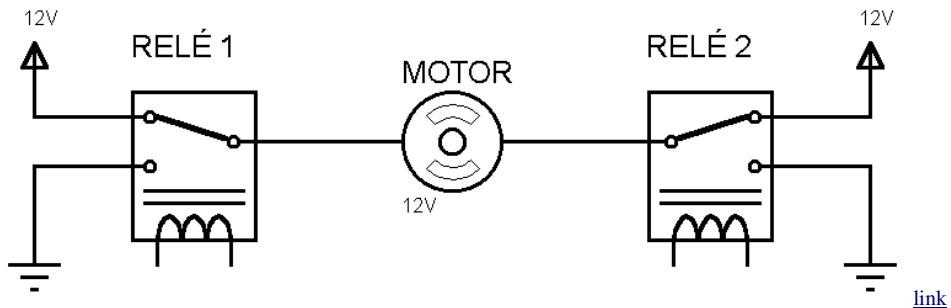


[link](#)

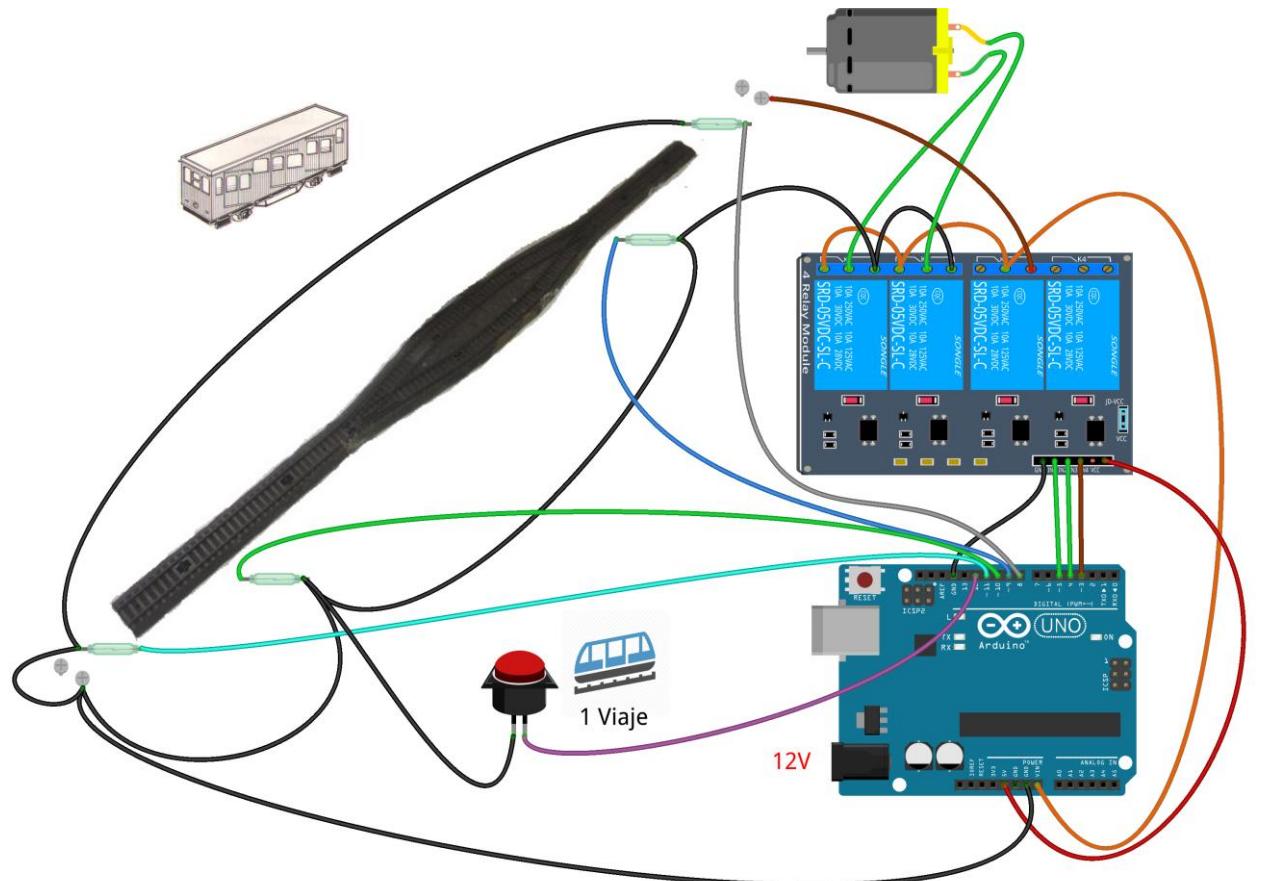
La idea

Voy a diseñar un módulo para montar el funicular con un pulsador en el lado 'Público' para que lo puedan poner en marcha y hacer un viaje de ida y vuelta. La luz de la cabina la controlaré de forma que se encienda antes de llegar a la estación y durante el tiempo que los viajeros *Preisermanes* suben y bajan del funicular. Para ello añadiré un par de Reed en la vía (uno arriba y otro abajo) para detectar cuando la cabina está llegando a la estación además de los que ya se colocan para detectar el final del recorrido.

Para el control del motor y de la luz usaré una placa de 4 relés. Un relé será para el control de la luz de las cabinas. Para el control del sentido de giro del motor se usan dos relés, el motor se conecta al Común de cada relé y los otros terminales de cada relé se conectan de igual forma a la alimentación. Así si los dos relés están activados o desactivados a la vez el motor está parado. Si solo se activa un relé el motor girará en un sentido u otro dependiendo del relé activado.



El circuito a montar queda así:



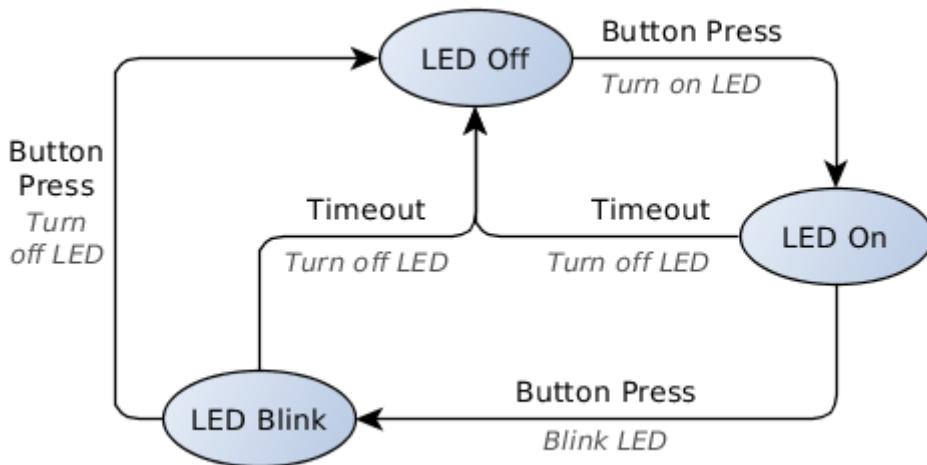
fritzing[link](#)

La máquina de estados finitos

Voy a usar una forma relativamente simple de programar un automatismo como este, la **Máquina de Estados Finitos**. Esto quiere decir que el automatismo estará en cada momento en un estado conocido y cambiará a otro estado conocido en función de algún evento (por una entrada, un temporizador, contador, etc.)

Por ejemplo, si queremos controlar un LED con un pulsador de forma que se encienda o parpadee durante un tiempo, tendremos tres 'Estados' conocidos: LED apagado, LED encendido y LED parpadeando.

La 'Transición' de un estado a otro dependerá de un 'Evento', el accionamiento del pulsador o que haya pasado un tiempo, que hará que se modifiquen las 'Variables' internas y según en el estado en el que nos encontremos harán que ejecuten las 'Acciones' correspondientes para pasar al nuevo estado. Lo podemos representar gráficamente así:



[link](#)

Los estados son estables en tanto no actúa sobre ellos una variable a la que son sensibles, las otras variables se ignoran. Una entrada física puede afectar a varias variables, por ejemplo, un pulsador puede activar dos variables, una para pulsación corta y otra para pulsación larga.

Este conjunto, formado por Estados, Variables de contexto y Acciones es un Autómata de Estados. Informáticamente existen muchas implementaciones y representaciones del mismo, por ejemplo en la forma gráfica descrita, aunque también se puede representar en forma de matriz, en la que se indica, por cada Estado (línea), y para cada Variable de contexto (columna), la dupla [Acción, Estado de llegada]

Estado	Variable	Acción	Nuevo Estado
LED apagado	Botón pulsado	Encender Temporizador 10s LED	LED encendido
LED encendido	Botón pulsado	Inicia parpadeo LED Temporizador 10s	LED parpadea
	Fin Temporizador	Apagar LED	LED apagado
LED parpadea	Botón pulsado	Apagar LED	LED apagado
	Fin Temporizador	Apagar LED	LED apagado

El ciclo comienza resemando **TODAS** las variables que afectan al sistema independientemente del estado y se setean después si han de estar activas en función de las entradas, los temporizadores, contadores, etc.

Este es el elemento básico, a cada vuelta de programa se toma "una foto" de todas las variables del sistema, y se actúa con ellas.

Luego según el estado en que se encuentre el sistema se comprueban las variables a la que ese estado es '**sensible**' y si esta activa se ejecutan las acciones correspondientes para '**transicionar**' al nuevo estado. Y ya está, no hay más, se vuelve al principio del ciclo.

Siempre hay que definir un estado inicial según dejemos el `setup()` a partir del cual empieza a funcionar nuestra maquina de estados. También puede ser necesario en el bucle general o en un estado concreto '**ejecutar**' determinadas funciones para gestionar las entradas o las salidas, por ejemplo, en el estado 'LED parpadea' llamariamos a la función que hace parpadear el LED.

Detectando pulsaciones

Si nos fijamos, en los tres estados, una de las Variables sensibles es la de botón pulsado, si actualizamos la variable en nuestra maquina así:

```
const int pin = 12;
bool botonPulsado ;

botonPulsado = false;
if (digitalRead(pin) == LOW)
    botonPulsado = true;
```

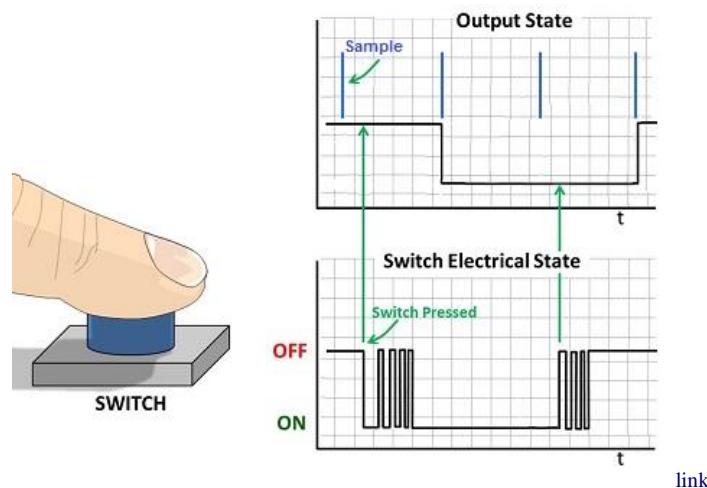
Ocurre que cuando se pulse el botón, en el estado veremos que esta activa y pasaremos al siguiente estado, lo cual es correcto, pero como el bucle se ejecuta tan rápido en el siguiente bucle se volverá a leer y activar la variable y como ese estado también es sensible a esa Variable pasaremos al siguiente estado y así sucesivamente.

Necesitamos detectar cuando se pulsa, pasa de **HIGH** a **LOW** la entrada, no solo que este a **LOW**. Para detectar el flanko de bajada tenemos que escribir el código así:

```
int ultimoValor;
int valorActual;

botonPulsado = false;
valorActual = digitalRead(pin);
if ((valorActual == LOW) && (ultimoValor == HIGH))
    botonPulsado = true;
ultimoValor = valorActual;
```

Ahora solo pasará de un estado a otro en el momento de pulsar el botón, no mientras este pulsado. Como los pulsadores no son perfectos generan rebotes, para tener una lectura correcta es conveniente poner condensador en el botón o hacer un pequeño retraso de algunos milisegundos entre exploraciones del estado del botón mayor que el tiempo que el botón esta rebotando.



[link](#)

```
void loop () {
actualizaVariables();
ejecutaEstado();
delay(10); // retraso para eliminar rebotes en lectura pulsador
}
```

Si quisieramos pasar directamente del estado de LED apagado a LED parpadeante cuando pulsamos más de un segundo y medio el pulsador, sería tan sencillo como definir dos Variables y en función del tiempo pulsado activar una u otra. Detectamos los flancos y si es el de bajada guardamos el tiempo actual, al detectar el flanco de subida comparamos el tiempo actual con el guardado y activamos la variable correspondiente.

Añadiríamos una transición, de forma que la pulsación corta pasaría del estado LED apagado al estado LED encendido y la pulsación larga pasaría al estado LED parpadea.

```
unsigned long tiempoInicial;
bool pulsacionCorta = false;
bool pulsacionLarga = false;
```

```

valorActual = digitalRead(pin);
if (ultimoValor!= valorActual) {
  if (valorActual ==LOW)
    tiempoInicial = millis();
  else {
    if (millis() - tiempoInicial > 1500)
      pulsacionLarga = true;
    else
      pulsacionCorta = true;
  }
}
ultimoValor = valorActual;

```

El Funicular de estados finitos

En el funicular durante su funcionamiento podemos identificar una serie de estados conocidos en los que pasará de uno a otro cuando por ejemplo se active un Reed.

Si empezamos teniendo la cabina con los imanes en la estación superior y todo apagado, para iniciar la secuencia tendremos que esperar a que el público pulse el botón. Una vez esto ocurra pasaremos a otro estado en el que tendremos las luces encendidas durante unos segundos para que los viajeros suban a las cabinas. El siguiente estado será cuando pasado este tiempo se inicie el viaje de bajada, etc.

Podemos describir la matriz de la máquina de estados del funicular así:

Estado	Variable	Acción	Nuevo Estado
Cabina arriba	Pulsador	Enciende Luz Temporizador 3 seg.	Espera viajeros arriba
	Fin Temporizador	Apaga luz	Cabina arriba
Espera viajeros arriba	Fin temporizador	Motor bajar	Baja con luz
Baja con luz	Reed vía arriba	Apaga Luz	Baja sin luz
Baja sin luz	Reed vía abajo	Enciende luz	Cabina abajo
Cabina abajo	Reed estación abajo	Para motor Temporizador 6 seg.	Espera viajeros abajo
Espera viajeros abajo	Fin temporizador	Motor subir	Sube con luz
Sube con luz	Reed vía abajo	Apaga luz	Sube sin luz
Sube sin luz	Reed vía arriba	Enciende luz	Cabina fin trayecto
Cabina fin trayecto	Reed estación arriba	Para motor Temporizador 3 seg	Cabina arriba

Pero como hicimos en el ejemplo del LED anterior también lo podemos describir con sólo cuatro estados: Cabina arriba, Bajando, Cabina abajo y Subiendo.

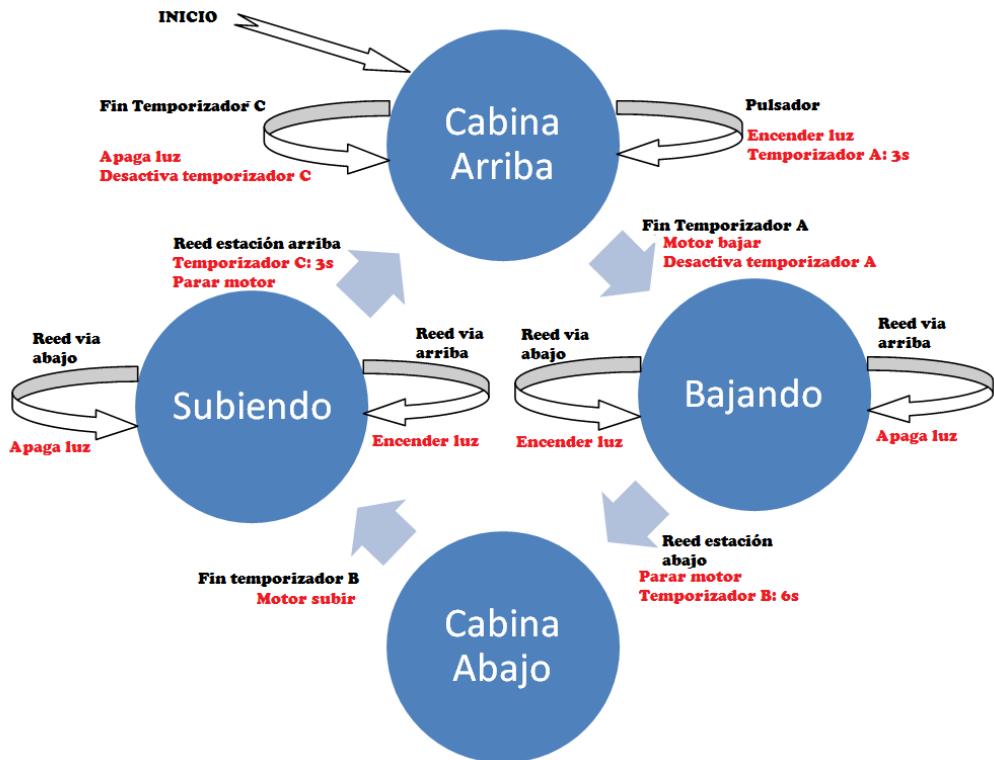
Durante el funcionamiento, dependiendo de las variables que se activen ejecutaremos unas determinadas acciones y permaneceremos en el mismo estado o transicionamos al siguiente.

En este caso nuestra matriz de estados quedaría así:

Estado	Variable	Acción	Nuevo Estado
Cabina arriba	Pulsador	Enciende Luz Temporizador A: 3s	Cabina arriba
	Fin Temporizador A	Motor bajar Desactivar Temporizador A	Bajando
	Fin Temporizador C	Apaga Luz Desactivar Temporizador C	Cabina arriba
Bajando	Reed vía arriba	Apaga Luz	Bajando
	Reed vía abajo	Enciende Luz	Bajando
	Reed Estación abajo	Para Motor Temporizador B: 6s	Cabina abajo
Cabina abajo	Fin Temporizador B	Motor subir	Subiendo
Subiendo	Reed vía abajo	Apaga Luz	Subiendo
	Reed vía arriba	Enciende Luz	Subiendo
	Reed Estación arriba	Para Motor Temporizador C: 3s	Cabina arriba

En este caso vemos que necesitamos usar más temporizadores y desactivarlos si es necesario, para que no estén disparados al entrar en un estado determinado.

De forma gráfica se podría definir así:



[link](#)

Cualquiera de las dos matrices es válida pero vamos a usar esta segunda, además vamos a definir las estructuras de datos de forma genérica de forma que nos puedan servir para otros programas que implementemos como máquina de estados como por ejemplo un tren ida y vuelta con paradas intermedias, una atracción de feria, un paso a nivel, etc. e incluso que lo podamos convertir fácilmente a una librería propia.

La máquina de estados finitos (FSM) es bastante simple de definir, tendremos unos estados que pueden ser el estado actual o un próximo estado al que transicionar:

```

int actualEstado;
int proximoEstado;
enum Estados {CABINA_ARRIBA, BAJANDO, CABINA_ABAJO, SUBIENDO};
    
```

Unas variables de contexto :

```

#define NUM_VARIABLES 8
enum Variables {PULSADOR, REED_ESTACION_ARRIBA, REED_ESTACION_ABAJO, REED_VIA_ARRIBA,
REED_VIA_ABAJO, TIEMPO_A, TIEMPO_B, TIEMPO_C};
    
```

y unas acciones que ejecutar sobre las salidas para llegar al próximo estado (los relés de esta placa se activan con nivel **LOW**):

```

#define PIN_LUZ 3
#define PIN_MOTOR1 4
#define PIN_MOTOR2 5

void luzOn() {digitalWrite(PIN_LUZ, LOW);}
void luzOff() {digitalWrite(PIN_LUZ, HIGH);}
void motorParar() {digitalWrite(PIN_MOTOR1, HIGH); digitalWrite(PIN_MOTOR2, HIGH);}
void motorBajar() {digitalWrite(PIN_MOTOR1, LOW); digitalWrite(PIN_MOTOR2, HIGH);}
void motorSubir() {digitalWrite(PIN_MOTOR1, HIGH); digitalWrite(PIN_MOTOR2, LOW);}
    
```

Y cíclicamente se ejecutará nuestra maquina de estados.

```

void loop() {
    actualizaVariables(); // actualiza variables (entradas y temporizadores)
    ejecutaFSM(); // ejecuta maquina de estados finitos
}
    
```

Las variables de contexto

El cambio de estado se realiza comprobando las variables de contexto que pueden estar activas o no, en caso de que este activa y el estado sea sensible a ella se producirá el cambio.

La activación puede ser porque un pin de entrada este a un nivel determinado, cambie de nivel o se mantenga en nivel determinado tiempo, pero también puede ser porque hayamos establecido un temporizador y haya pasado el tiempo especificado o un contador haya llegado a un número determinado.

En lugar de escoger un tipo de variable o variables diferentes para contener los datos necesarios de cada variable de contexto voy a tratar de generalizar todos estos parámetros en una estructura genérica que me sirva tanto para pines de entrada como para temporizadores.

Por tanto, esta estructura contendrá una variable para ver si su estado es activo o no, en caso de la activación provenga de una entrada guardaremos el pin y su nivel (**HIGH** o **LOW**) incluso la referencia de tiempo por si tenemos que discernir entre pulsación corta y larga. Si la activación proviene de un temporizador tendremos un parámetro para saber si está en marcha o no y las referencias de tiempo necesarias:

```
struct {
    bool activo;                      // variable de contexto activa
    bool marcha;                      // Temporizador en marcha
    int pin;                           // Pin de entrada
    int ultimoNivel;                  // Nivel del pin de entrada
    unsigned long tiempoInicial;      // Tiempo inicial
    unsigned long tiempoEspera;        // Tiempo de espera para activacion
} Variable[NUM_VARIABLES];
```

Para establecer estos parámetros de las variables de contexto nos valdremos de unas funciones que almacenarán los valores adecuados según provengan los datos.

Para el caso de temporizadores, guardaremos el tiempo inicial y el tiempo de espera e indicaremos en **marcha** que el temporizador está contando tiempo:

```
void setTemporizador (int num, unsigned long espera) {           // establece temporizador
    Variable[num].tiempoEspera = espera;
    Variable[num].tiempoInicial = millis();
    Variable[num].marcha = true;
}
```

Para desactivar el temporizador:

```
void stopTemporizador (int num) {                                // desactiva temporizador
    Variable[num].marcha = false;
}
```

Si proviene del estado de un pin, tendremos una función para indicar el pin e inicializarlo como entrada.

```
void iniEntrada (int entrada, int pin) {                         // definir pin entrada
    pinMode(pin, INPUT_PULLUP);
    Variable[entrada].pin = pin;
}
```

Para la evaluación de la variable, si ha de ser por nivel de entrada (el nivel **LOW** es el activo) es tan sencillo como:

```
void detectaActivacion (int entrada) {                            // detecta pin entrada a LOW
    if (digitalRead(Variable[entrada].pin) == LOW)
        Variable[entrada].activo = true;
}
```

Pero si ha de ser por detección del momento de la pulsación o tiempo pulsado tendremos que comprobar cuando se produce un cambio de nivel en el pin comparado con el anterior por lo que necesitaremos de una función que nos indique si ha habido un flanco:

```
bool isFlanco (int entrada) {                                     // detecta flanco de subida o bajada en pin
    if (Variable[entrada].ultimoNivel != digitalRead(Variable[entrada].pin)) {
        if (Variable[entrada].ultimoNivel == HIGH)
            Variable[entrada].ultimoNivel = LOW;
        else
            Variable[entrada].ultimoNivel = HIGH;
        return (true);
    }
    return (false);
}
```

Ahora ya podremos evaluar el pin para detectar la pulsación al pasar de **HIGH** a **LOW**:

```
void detectaPulsacion (int entrada) {                                // detecta flanco de bajada
    if (isFlanco (entrada))
        if (Variable[entrada].ultimoNivel == LOW)
            Variable[entrada].activo = true;
}
```

También podremos evaluar el pin para comprobar si es una pulsación corta inferior a un tiempo o larga si supera ese tiempo (en el caso del funicular no lo utilizamos):

```
// detecta pulsacion corta inferior a tiempo
void detectaPulsacionCorta (int entrada, unsigned long tiempo) {
    if (isFlanco(entrada)) {
        if (Variable[entrada].ultimoNivel == LOW)
            Variable[entrada].tiempoInicial = millis();
        else {
            if (millis() - Variable[entrada].tiempoInicial < tiempo)
                Variable[entrada].activo = true;
        }
    }
}

// detecta pulsacion larga superior a tiempo
void detectaPulsacionLarga (int entrada, unsigned long tiempo) {
    if (isFlanco(entrada)) {
        if (Variable[entrada].ultimoNivel == LOW)
            Variable[entrada].tiempoInicial = millis();
        else {
            if (millis() - Variable[entrada].tiempoInicial > tiempo)
                Variable[entrada].activo = true;
        }
    }
}
```

Actualizando variables

La máquina de estados a cada ciclo resetea las variables de estado y luego evalúa cada una para ver si se ha de activar de forma que en cada ciclo tendremos una versión actualizada del sistema.

Podemos aprovechar el momento de resetear las variables de contexto para actualizar las que correspondan a temporizadores en marcha para ver si se ha superado el tiempo de espera.

```
void resetVariables() {                                         // reseta variables y comprueba temporizadores
    for (int i = 0; i < NUM_VARIABLES; i++) {
        Variable[i].activo = false;
        if (Variable[i].marcha)
            if (millis() - Variable[i].tiempoInicial > Variable[i].tiempoEspera)
                Variable[i].activo = true;
    }
}
```

En función del tipo de variable de contexto llamaremos a una u otra función de evaluación de entradas para que actualice las variables.

La evaluación de los Reed se puede hacer por nivel, la del pulsador también se podría hacer por nivel pero prefiero hacerlo por flanco ya que se me ha ocurrido que podría poner un interruptor en el lado 'Operador' en paralelo con el pulsador del lado 'Público' de forma que yo también pueda poner en marcha el funicular con el mismo y si lo dejo activado no se podrá poner de nuevo en marcha el viaje con lo que tendría un 'interesado' control del flujo de visitantes.

```
void actualizaVariables() {                                     // actualiza variables al inicio del ciclo
    resetVariables();
    detectaActivacion (REED_ESTACION_ARRIBA);
    detectaActivacion (REED_ESTACION_ABAJO);
    detectaActivacion (REED_VIA_ARRIBA);
    detectaActivacion (REED_VIA_ABAJO);
    detectaPulsacion (PULSADOR);
    delay(10);                                              // espera para evitar rebotes
}
```

Los estados y transiciones

Ha de haber un estado inicial conocido desde el que pongamos en marcha nuestra maquina de estados así que definiré una función que nos establecerá este estado y reseteará las variables de contexto.

```
void iniEstado (int estado) {
    for (int i = 0; i < NUM_VARIABLES; i++) {      // resetea variables, temporizadores y estado
        Variable[i].marcha = false;
        Variable[i].activo = false;
    }
    actualEstado = estado;                         // establece estado inicial
}
```

El resto de condiciones iniciales los estableceremos en el **setup()**

```
#define PIN_REED_ESTACION_ARRIBA 8
#define PIN_REED_VIA_ARRIBA 9
#define PIN_REED_VIA_ABAJO 10
#define PIN_REED_ESTACION_ABAJO 11
#define PIN_PULSADOR 12

void setup() {
    iniEntrada(PULSADOR, PIN_PULSADOR);                      // definimos pin entrada
    iniEntrada(REED_ESTACION_ARRIBA, PIN_REED_ESTACION_ARRIBA);
    iniEntrada(REED_VIA_ARRIBA, PIN_REED_VIA_ARRIBA);
    iniEntrada(REED_VIA_ABAJO, PIN_REED_VIA_ABAJO);
    iniEntrada(REED_ESTACION_ABAJO, PIN_REED_ESTACION_ABAJO);
    pinMode(PIN_LUZ, OUTPUT);                                  // definimos pin salidas
    pinMode(PIN_MOTOR1, OUTPUT);
    pinMode(PIN_MOTOR2, OUTPUT);
    motorParar();                                            // inicializar salidas
    luzOff();
    iniEstado(CABINA_ARRIBA);                                // definimos estado inicial
}
```

Ya solo queda programar la máquina de estados que ejecutará las transiciones de estado a estado en función de las variables de contexto. Como escribiremos bastantes veces la comprobación de la transición entre estados comparando de una variable de contexto para ver si esta activa y actualizar el nuevo estado voy a crear una función para mejorar la legibilidad del código que como parámetros tiene la variable de estado, y nuevo estado al que iría, así si esta activa actualiza al estado y devuelve true si esta activa.

```
//comprueba la activacion de una variable hacia un estado
bool isTransicionVariable (int entrada, int estado) {
    if (Variable[entrada].activo == true)
        proximoEstado = estado;
    return (Variable[entrada].activo);
}
```

De esta forma mejora la legibilidad del código de nuestra maquina de estados finitos ya que es un simple **switch()-case**, en el que para cada caso comprobamos para cada variable de contexto sensible si hay una transición y ejecutamos las acciones para llegar al nuevo estado.

```
void ejecutaFSM () {                                         // maquina de estados finitos
    switch (actualEstado) {
        case CABINA_ARRIBA:
            if (isTransicionVariable(PULSADOR, CABINA_ARRIBA)) { // comprueba transicion hacia un estado
                luzOn();                                         // acciones
                setTemporizador (TIEMPO_A, 3000);
            }
            if (isTransicionVariable (TIEMPO_A, BAJANDO)) {
                motorBajar();
                stopTemporizador(TIEMPO_A);
            }
            if (isTransicionVariable (TIEMPO_C, CABINA_ARRIBA)) {
                luzOff();
                stopTemporizador(TIEMPO_C);
            }
            break;
        case BAJANDO:
            if (isTransicionVariable (REED_VIA_ARRIBA, BAJANDO))
                luzOff();
            if (isTransicionVariable (REED_VIA_ABAJO, BAJANDO))
                luzOn();
    }
}
```

```

        if (isTransicionVariable (REED_ESTACION_ABAJO, CABINA_ABAJO)) {
            motorParar();
            setTemporizador (TIEMPO_B, 6000);
        }
        break;
    case CABINA_ABAJO:
        if (isTransicionVariable (TIEMPO_B, SUBIENDO))
            motorSubir();
        break;
    case SUBIENDO:
        if (isTransicionVariable (REED_VIA_ABAJO, SUBIENDO))
            luzOff();
        if (isTransicionVariable (REED_VIA_ARRIBA, SUBIENDO))
            luzOn();
        if (isTransicionVariable (REED_ESTACION_ARRIBA, CABINA_ARRIBA)) {
            motorParar();
            setTemporizador (TIEMPO_C, 3000);
        }
        break;
    }
    actualEstado = proximoEstado; // actualiza nuevo estado de la maquina
}

```

El programa

El programa completo para el funicular queda así:

```

/* BRAWA Standseilbahn 6410 - Maquina de estados - F.Cañada 2020 */

#define PIN_LUZ      3
#define PIN_MOTOR1   4
#define PIN_MOTOR2   5
#define PIN_REED_ESTACION_ARRIBA 8
#define PIN_REED_VIA_ARRIBA 9
#define PIN_REED_VIA_ABAJO 10
#define PIN_REED_ESTACION_ABAJO 11
#define PIN_PULSADOR 12

// FSM - Definicion Maquina de estados finitos

int actualEstado;
int proximoEstado;

// Estados
enum Estados {CABINA_ARRIBA, BAJANDO, CABINA_ABAJO, SUBIENDO};

// Variables de contexto

// Entradas
#define NUM_VARIABLES 8

enum Variables {PULSADOR, REED_ESTACION_ARRIBA, REED_ESTACION_ABAJO, REED_VIA_ARRIBA, REED_VIA_ABAJO, TIEMPO_A,
TIEMPO_B, TIEMPO_C};

struct {
    bool activo; // variable de contexto activa
    bool marcha; // Temporizador en marcha
    int pin; // Pin de entrada
    int ultimoNivel; // Nivel del pin de entrada
    unsigned long tiempoInicial; // Tiempo inicial
    unsigned long tiempoEspera; // Tiempo de espera para activacion
} Variable[NUM_VARIABLES];

// Acciones
void luzOn() {digitalWrite(PIN_LUZ, LOW);}
void luzOff() {digitalWrite(PIN_LUZ, HIGH);}
void motorParar() {digitalWrite(PIN_MOTOR1, HIGH); digitalWrite(PIN_MOTOR2, HIGH);}
void motorBajar() {digitalWrite(PIN_MOTOR1, LOW); digitalWrite(PIN_MOTOR2, HIGH);}
void motorSubir() {digitalWrite(PIN_MOTOR1, HIGH); digitalWrite(PIN_MOTOR2, LOW);}

void setup() {
    iniEntrada(PULSADOR, PIN_PULSADOR); // definimos pin entrada
    iniEntrada(REED_ESTACION_ARRIBA, PIN_REED_ESTACION_ARRIBA);
    iniEntrada(REED_VIA_ARRIBA, PIN_REED_VIA_ARRIBA);
    iniEntrada(REED_VIA_ABAJO, PIN_REED_VIA_ABAJO);
    iniEntrada(REED_ESTACION_ABAJO, PIN_REED_ESTACION_ABAJO);
    pinMode(PIN_LUZ, OUTPUT); // definimos pin salidas
    pinMode(PIN_MOTOR1, OUTPUT);
    pinMode(PIN_MOTOR2, OUTPUT);
    motorParar(); // inicializar salidas
    luzOff(); // definimos estado inicial
    iniEstado(CABINA_ARRIBA);
}

```

```

void loop() {
    actualizaVariables(); // actualiza variables (entradas y temporizadores)
    ejecutaFSM(); // ejecuta maquina de estados finitos
}

void iniEstado (int estado) {
    for (int i = 0; i < NUM_VARIABLES; i++) { // resetea temporizadores y estado
        Variable[i].marcha = false;
        Variable[i].activo = false;
    }
    actualEstado = estado; // establece estado inicial
}

void iniEntrada (int entrada, int pin) { // definir pin entrada
    pinMode(pin, INPUT_PULLUP);
    Variable[entrada].pin = pin;
}

void detectaActivacion (int entrada) { // detecta pin entrada a LOW
    if (digitalRead(Variable[entrada].pin) == LOW)
        Variable[entrada].activo = true;
}

bool isFlanco (int entrada) { // detecta flanco de subida o bajada en pin
    if (Variable[entrada].ultimoNivel != digitalRead(Variable[entrada].pin)) {
        if (Variable[entrada].ultimoNivel == HIGH)
            Variable[entrada].ultimoNivel = LOW;
        else
            Variable[entrada].ultimoNivel = HIGH;
        return (true);
    }
    return (false);
}

void detectaPulsacion (int entrada) { // detecta flanco de bajada
    if (isFlanco (entrada))
        if (Variable[entrada].ultimoNivel == LOW)
            Variable[entrada].activo = true;
}

void stopTemporizador (int num) { // desactiva temporizador
    Variable[num].marcha = false;
}

void setTemporizador (int num, unsigned long espera) { // establece temporizador
    Variable[num].tiempoEspera = espera;
    Variable[num].tiempoInicial = millis();
    Variable[num].marcha = true;
}

void resetVariables() { // reseta variables y comprueba temporizadores
    for (int i = 0; i < NUM_VARIABLES; i++) {
        Variable[i].activo = false;
        if (Variable[i].marcha)
            if (millis() - Variable[i].tiempoInicial > Variable[i].tiempoEspera)
                Variable[i].activo = true;
    }
}

bool isTransicionVariable (int entrada, int estado) { // comprueba la activacion de una variable
    hacia un estado
    if (Variable[entrada].activo == true) {
        proximoEstado = estado;
    }
    return (Variable[entrada].activo);
}

//-----

void actualizaVariables() { // actualiza variables al inicio del ciclo
    resetVariables();
    detectaActivacion (REED_ESTACION_ARRIBA);
    detectaActivacion (REED_ESTACION_ABAJO);
    detectaActivacion (REED_VIA_ARRIBA);
    detectaActivacion (REED_VIA_ABAJO);
    detectaPulsacion (PULSADOR);
    delay(10); // espera para evitar rebotes
}

void ejecutaFSM () { // maquina de estados finitos
    switch (actualEstado) {
        case CABINA_ARRIBA:
            if (isTransicionVariable(PULSADOR, CABINA_ARRIBA)) { // comprueba transicion hacia un estado
                luzOn(); // acciones
                setTemporizador (TIEMPO_A, 3000);
            }
            if (isTransicionVariable (TIEMPO_A, BAJANDO)) {
                motorBajar();
                stopTemporizador(TIEMPO_A);
            }
    }
}

```

```

if (isTransicionVariable (TIEMPO_C, CABINA_ARRIBA)) {
    luzOff();
    stopTemporizador(TIEMPO_C);
}
break;
case BAJANDO:
    if (isTransicionVariable (REED_VIA_ARRIBA, BAJANDO))
        luzOff();
    if (isTransicionVariable (REED_VIA_ABAJO, BAJANDO))
        luzOn();
    if (isTransicionVariable (REED_ESTACION_ABAJO, CABINA_ABAJO)) {
        motorParar();
        setTemporizador (TIEMPO_B, 6000);
    }
    break;
case CABINA_ABAJO:
    if (isTransicionVariable (TIEMPO_B, SUBIENDO))
        motorSubir();
    break;
case SUBIENDO:
    if (isTransicionVariable (REED_VIA_ABAJO, SUBIENDO))
        luzOff();
    if (isTransicionVariable (REED_VIA_ARRIBA, SUBIENDO))
        luzOn();
    if (isTransicionVariable (REED_ESTACION_ARRIBA, CABINA_ARRIBA)) {
        motorParar();
        setTemporizador (TIEMPO_C, 3000);
    }
    break;
default: // No deberiamos haber llegado hasta aqui
    iniEstado(CABINA_ARRIBA);
    motorParar();
    break;
}
actualEstado = proximoEstado; // actualiza nuevo estado de la maquina
}

```

21. Baterías y Neopixels: Diorama en un tarro de cristal

Leí un artículo en la revista [Model Rail](#) sobre cómo hacer un pequeño diorama dentro de un tarro de cristal. Siempre había visto el típico barco velero en una botella pero nunca se me había ocurrido hacer algo parecido con un tren a escala. Dentro del tarro además de una pequeña locomotora habían colocado una farola que se alimentaba con unas pilas situadas en la parte inferior y mediante un interruptor colocado taladrando la tapa podían encenderla y apagarla.

Dándole vueltas al asunto y pensando en cómo mejorar la iluminación se me ocurrió hacer un efecto noche/día con unos LED blanco, rojo y azul y no agujerear la tapa para encender y apagar el efecto por lo que la incorporación de un Arduino tiene su sentido. Esto que parece simple se empieza a complicar cuando quieras exponer tu tarro de cristal en un encuentro modular, las pilas han de aguantar las horas que dura la exposición abierta al público por lo que hay que mirar cuánto se consume y qué capacidad tienen las pilas o baterías que se instalen.

La idea

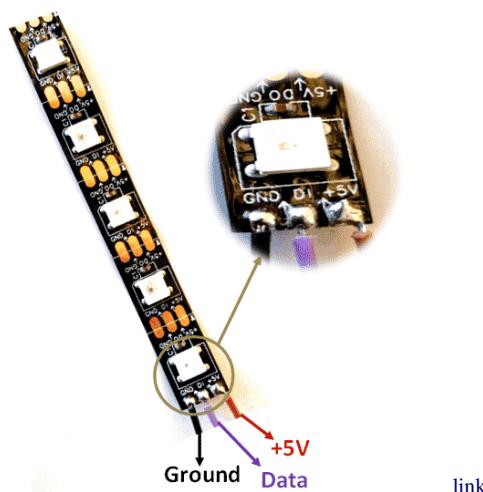
He decidido usar una batería recargable LiPo en lugar de pilas para ser más ecológico y voy a usar un Neopixel en lugar de tres LED de colores, además del LED de la farola y un Reed para encender y apagar la iluminación.



[link](#)

Neopixels

Los Neopixels son unos LED RGB direccionables individualmente basados en un controlador WS2812 incluido en el propio encapsulado que se conectan encadenados, y se presentan normalmente en tiras de LED. Cuentan con 4 pines (+5V, GND, DIN, DOUT), reciben la información en un protocolo específico por DIN y se la pasan al siguiente Neopixel por DOUT por lo que requieren de un micro para funcionar.



[link](#)

Los WS2812 usan una velocidad de transmisión de 800kHz, lo que es bastante elevada para un Arduino, y requiere de temporizaciones muy precisas, afortunadamente para Arduino hay dos librerías que pueden controlar este tipo de LED respetando sus temporizaciones teniendo en cuenta la frecuencia a la que funciona el Arduino, la [FastLED.h](#) y la [Adafruit_Neopixel.h](#), esta última más simple.

<https://github.com/FastLED/FastLED>

https://github.com/adafruit/Adafruit_NeoPixel

Con la [Adafruit_Neopixel.h](#) para controlar la tira de Neopixels se define un objeto pixel al que le pasamos la cantidad de LED, el pin al que está conectada la tira (señal DIN) y unas constantes que definen el tipo de Neopixel, [NEO_KHZ800](#) para la velocidad de transmisión y [NEO_GRB](#) para el orden de los datos (Green, Red, Blue) este puede ser que varíe de un fabricante a otro del Neopixel.

```
#define NUMPIXELS      1
#define NEOPIXEL_PIN    6
Adafruit_NeoPixel pixels(NUMPIXELS, NEOPIXEL_PIN, NEO_GRB + NEO_KHZ800);
```

para usarlo en el `setup()` ponemos.

```
pixels.begin();                                // inicializa pin Neopixels
```

Para definir el color de un pixel se puede usar `setPixelColor()` y finalmente para enviar los datos a la tira de Neopixels se usa `show()`

Ahormando

Además del consumo del Neopixel (hasta 20mA por color, o sea 60mA mostrando el color blanco a máxima intensidad), en el Arduino hay varios elementos en la placa que van sumando consumo: LED, chip USB, reguladores de tensión... Por ejemplo, si cargamos este *sketch* vacío y medimos el consumo sólo de la placa Arduino:

```
void setup () {}
void loop () {}
```

Alimentando con una pila de 9V por VIN mi Arduino Uno consume 25,7 mA, el Arduino Nano 19,8 mA y el Arduino Pro Mini sólo 15,1 mA al no tener chip USB, etc. Si alimentamos a 5V por el pin 5V desciende algunas décimas de mA.

Hay muchos momentos en que el Arduino no hace nada, como en ese *sketch* de prueba o cuando está esperando que pase un tiempo determinado con `delay()` o `millis()`. Para estos momentos contemplativos el procesador se puede poner en un modo de bajo consumo desconectando diferentes módulos internos e incluso parando el procesador y que vuelva a funcionar normalmente tras un tiempo vigilado por el *watchdog timer* WDT (temporizador perro guardián) o al saltar una interrupción.

Lo más simple es usar la librería [Low-Power.h](#) que permite gestionar los diferentes modos de ahorro de energía.

<https://github.com/rocketscream/Low-Power>

Para conseguir el mínimo consumo posible hacemos que el micro entre en un modo de suspensión en el que detiene su reloj y los periféricos un máximo de 8 segundos con [SLEEP_8S](#) (también podemos escoger entre 15ms, 30ms, 60ms hasta los 8 segundos).

```
LowPower.powerDown(SLEEP_8S, ADC_OFF, BOD_OFF);
```

Incluso podemos suspenderlo indefinidamente con [SLEEP_FOREVER](#) hasta que se reciba una interrupción, por ejemplo en el pin D2 conectado al reed:

```
#define REED_PIN      2          // pin de interrupcion D2 o D3

void dormirProfundo() {
  attachInterrupt(digitalPinToInterrupt(REED_PIN), despierta, LOW);
  LowPower.powerDown(SLEEP_FOREVER, ADC_OFF, BOD_OFF);
  detachInterrupt(digitalPinToInterrupt(REED_PIN));
}

void despierta () {           // rutina de interrupcion externa
}                            // no necesita hacer nada
```

El chip se congelará con `LowPower.powerDown()` hasta que se active el reed, entonces saltará a la interrupción que no necesita hacer nada especial y al retornar seguirá por `detachInterrupt()` que desconecta la interrupción.

En este modo el Arduino Uno consume 8,6mA, el Nano 4,8mA y el Pro Mini sólo 3,1mA y casi todo debido al LED indicador de alimentación de la placa, así que usaré esta placa.

Como pulsador de encendido uso una ampolla reed, podría haber usado un sensor Hall pero este consume algo de corriente, el reed literalmente no consume nada.

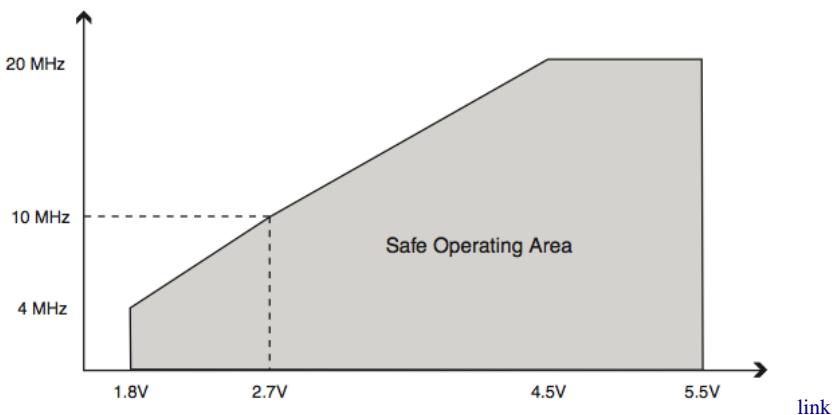
Un modo intermedio en el que aún están disponibles algunos periféricos es el modo *Idle*, con la librería podemos escoger los que dejamos operativos aunque aquí no lo voy a necesitar.

```
LowPower.idle(SLEEP_8S, ADC_OFF, TIMER2_OFF, TIMER1_OFF, TIMERO_OFF, SPI_OFF, USART0_OFF, TWI_OFF);
```

Trabajando

El procesador Atmel 328P puede trabajar desde un mínimo de 1,8V hasta un máximo de 5,5V por lo que se puede alimentar desde pilas o baterías pero si se alimenta por debajo de 4,5V tiene limitada la frecuencia del reloj:

Figure 28-1. Maximum Frequency vs. V_{CC}



[link](#)

Mi Arduino Pro Mini tiene un cristal de 16MHz por lo que la tensión mínima es de 3,8V aproximadamente. Hay placas con cristales de 8MHz con lo que la tensión mínima serían unos 2,4V. En el Arduino IDE se pueden elegir los dos modelos.

Las baterías LiPo dan 3,7V lo que está por debajo de las especificaciones de funcionamiento del Atmel 328P a 16MHz, afortunadamente cuando están cargadas dan 4,2V y en el procesador podemos cambiar la frecuencia de funcionamiento por software para trabajar a 8MHz aunque nos alterará **TODAS** las temporizaciones, incluidas las de las comunicaciones serie y la de los Neopixels, ya que están calculadas para la frecuencia del cristal por un **#define** del compilador llamado **F_CPU** que pone el Arduino IDE al seleccionar la placa.

Una forma simple de cambiar la frecuencia de reloj es modificando el *prescaler* (divisor) del reloj, podemos hacerlo usando una de las librerías del propio compilador avr, la **power.h**, así podemos cambiar su frecuencia de funcionamiento usando el divisor por 2 para trabajar a 8MHz en lugar de a 16MHz. Además funcionando a una frecuencia inferior consume menos lo que nos va bien.

```
#include <avr/power.h>
clock_prescale_set(clock_div_2); // Permite dividir por 1, 2, 4, 8, 16, 32, 64, 128 y 256
```

Al modificar el reloj los Neopixels no reciben la señal con la temporización correcta, ya que la librería se ha compilado teniendo en cuenta **F_CPU** que tiene el valor 16000000. Podemos probar a cambiarlo a 8000000 antes de compilar la librería usando la directiva **#undef** para borrarla y definirla de nuevo con el valor que vamos a usar:

```
#undef F_CPU
#define F_CPU 8000000
#include <Adafruit_NeoPixel.h>
```

Desgraciadamente no funciona, ya que el compilador primero precompila las librerías usadas en el *sketch* y luego el propio *sketch* antes de compilarlo todo por lo que la librería toma el valor **F_CPU** seleccionado en el IDE y no el del *sketch*. **Serial** también se ve afectado y la velocidad de comunicación también sería la mitad.

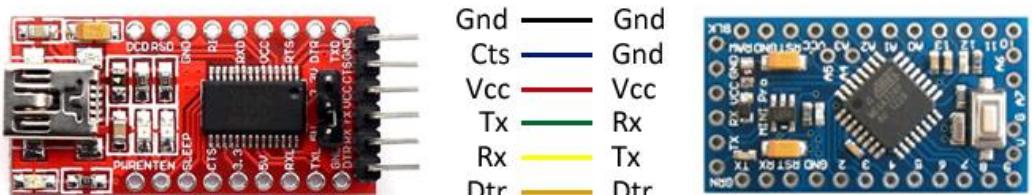
La solución alternativa (*work around*) que he encontrado para mantener las temporizaciones precisas de los Neopixels sin modificar la librería es seleccionar en el Arduino IDE el Arduino Pro Mini el "ATmega 328P (3.3V, 8MHz)" aunque use una placa a 16MHz y en el *sketch* compilar el cambio de frecuencia en función de un **#define**, así las temporizaciones serán correctas para 8MHz.

```
// Vamos a trabajar a 8MHz
// Compilar siempre el sketch seleccionando Arduino Pro Mini - ATmega 328P (3.3V, 8MHz)
// Comentar o borrar la siguiente linea si usamos un Arduino Pro Mini a 8MHz
#define XTAL_16MHz
```

En el **setup()** pondríamos lo siguiente:

```
#ifdef XTAL_16MHz
    clock_prescale_set(clock_div_2);
#endif
```

Para programar el Arduino Pro Micro, ya que no tiene chip USB en la placa, se necesita un convertidor USB-serie externo. Yo uso este con el chip FTDI en que la conexión es directa y se puede seleccionar entre alimentar a 5V o 3,3V, si usáis otro conectad correctamente las señales entre el convertidor USB y el Arduino.



[link](#)

Midiendo

Las baterías de LiPo tienen una tensión nominal de 3,7V, están completamente cargadas cuando alcanzan los 4,2V y están descargadas cuando bajan de 3,5V. Hay que evitar que bajen de 3,0V para que no sufran daños irreversibles.

Es posible conocer la tensión a la que está alimentado el Arduino usando el convertidor analógico-digital (ADC). Normalmente lo usamos para conocer la tensión analógica en uno de los pines A0 a A7 con **analogRead()** pero configurado adecuadamente nos puede servir para conocer la tensión de alimentación (Vcc) así podremos avisar cuando la tensión de la batería sea insuficiente.

Para ello configuraremos los registros del ADC para comparar la tensión Vcc respecto a la referencia interna de 1,1V del convertidor (*Bandgap*). La siguiente rutina nos devuelve el valor de la tensión de alimentación en milivoltios.

```
#define AREF_mV      1100          // VBG referencia interna 1100 mV
const unsigned long BAND_GAP = 1024UL * AREF_mV;

unsigned int readVcc() {
    unsigned int result;
    // REFS0 : Selecciona AVcc referencia externa
    // MUX3 MUX2 MUX1 : Selecciona 1.1V (VBG)
    ADMUX = bit(REFS0) | bit(MUX3) | bit(MUX2) | bit(MUX1);    // Referencia:AVCC,Canal:1.1V (VBG)
    delay(2);                                                 // Espera a estabilizar Vref
    ADCSRA |= bit(ADSC);                                    // inicia conversion
    while (ADCSRA & bit(ADSC));                            // espera a acabar
    result = ADC;                                         // lee resultado impreciso
    ADCSRA |= bit(ADSC);                                    // inicia conversion, la segunda vez mejor
    while (ADCSRA & bit(ADSC));                            // espera a acabar
    result = BAND_GAP / ADC;                               // Recalcula AVcc en mV   Vcc (in mV) = 1024 × 1100 / ADC
    return result;                                         // Vcc en millivoltios
}
```

Desde luego hay un pequeño error, ya que según especificaciones el *Bandgap* puede estar entre 1,0V y 1,2V según el procesador y la temperatura, pero nos puede servir para predecir cuándo se agota la batería. Para tener un resultado más preciso podemos medir con un voltmetro la tensión de referencia interna en el pin **AREF** de nuestra placa con el siguiente *sketch* y definirlo para **AREF_mV** de la rutina anterior.

```
void setup () {
    analogReference (INTERNAL);
    analogRead (A0); // fuerza voltaje referencia a encenderse
}

void loop () { }
```

El parámetro de `analogReference()` puede ser **INTERNAL** (1,1V), **EXTERNAL** (la tensión del pin AREF) o **DEFAULT** (la tensión de alimentación Vcc) para el ATmega328P y es la tensión máxima analógica a medir.

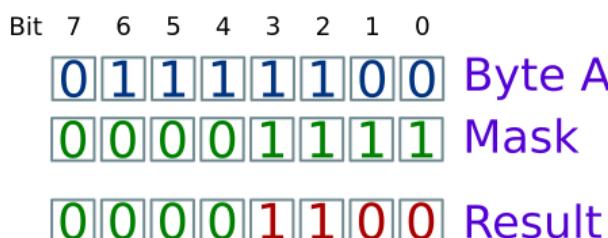
<https://www.arduino.cc/reference/en/language/functions/analog-io/analogreference/>

Avisando

Para avisar de que la batería esta baja encenderemos el LED de la placa (el definido como **LED_BUILTIN** en el compilador) pero si lo dejamos encendido tendremos más consumo y se acabará antes la batería, por lo que solo lo encenderemos un momento cada segundo.

La función `checkBattery()` la llamaré cada 60ms aproximadamente así que solo iluminaré el LED durante 60ms cada 16 llamadas a la función (60ms x 16 = 960ms).

Contar hasta 16 es sencillo usando las máscaras, estas son unos valores que haciendo la operación AND binaria (`&`) con otro valor acotan el resultado. Por ejemplo, con el valor 15 (**B1111** en binario, **0x0F** en hexadecimal) el valor queda acotado entre 0 y 15:



[link](#)

Los valores de máscara habituales son 1, 3, 5, 7, 15, 31, 63 y 127 para bytes. Haciendo la máscara podemos comparar el valor de un contador sin tener que ponerlo a cero.

```
#define BATT_MIN      3500      // Tension minima bateria para aviso
byte batteryFlash;

void checkBattery() {           // cada 60ms durante transiciones
    batteryFlash++;
    if ((readVcc() < BATT_MIN) && ((batteryFlash & 0x0F)==0))// flash cada segundo (60ms * 16 = 960ms)
        digitalWrite (LED_BUILTIN, HIGH);
    else
        digitalWrite (LED_BUILTIN, LOW);
}
```

Amanece que no es poco

Voy a tener cuatro escenas de iluminación:

Escena	Iluminación
Día	Luz blanca durante un periodo largo (máxima luz en LED R, G y B)
Atardecer	Un tono rojo-violáceo cuando se esconde el Sol (R más brillo que B y que G)
Noche	Luz azul durante un periodo largo (máxima luz sólo en LED B)
Amanecer	Un tono rojizo cuando aparece el Sol (medio brillo en R y poco en G)

Estos colores RGB los voy a tener en una tabla e iré calculando los colores intermedios para enviarlos al Neopixel y así pasar de una escena a otra de forma progresiva.

```
enum leds      {ROJO, VERDE, AZUL};
enum escenas   {DIA, ATARDECER, NOCHE, AMANECER};

byte colorEscena[][3] = {          // colores iniciales de la transicion
    {160, 50, 0},                // amanecer -> dia
    {255, 255, 255},              // dia -> atardecer
    {200, 50, 110},               // atardecer -> noche
    {0, 0, 255}                  // noche -> amanecer
};
```

Para la transición entre una y otra escena voy a enviar cada 60ms al Neopixel un valor calculado, por una regla de tres, entre el valor de la escena inicial que pasará como parámetro y el valor de siguiente escena en la tabla de cada color en 255 pasos por lo que tardaré 15s en cambiar de escena de iluminación.

La regla de tres la haremos de forma sencilla con la función `map()` y la pausa de 60ms con `LowPower.powerDown()`. La librería Neopixel tiene una función llamada `gamma8()` para el ajuste del brillo del pixel que suaviza las transiciones.

Este brillo ajustado por cada color del pixel se actualiza en el buffer de la librería con `setPixelColor()` que tiene como parámetros el número de pixel y los valores RGB.

Después de enviar el valor RGB calculado con la función de la librería `show()` dormiré el Arduino para reducir su consumo. También comprobaremos la batería y si hemos despertado porque se usó el imán para apagarlo reiniciamos la escena.

Observad que el parámetro de la función es del tipo `escenas` tal como hemos definido `escenaActual` ya que tomará los valores de la enumeración, podríamos haber usado otro tipo compatible como `byte` o `int` pero así no mezclamos tipos.

```
escenas escenaActual;

void transicion(escenas escena) {
    byte r, g, b, i;
    escenas proxEscena;
    escenaActual = escena;
    proxEscena = (escena == AMANECER) ? DIA : escena + 1;
    for (i = 0; i < 255; i++) {
        r = map(i, 0, 255, colorEscena[escenaActual][ROJO], colorEscena[proxEscena][ROJO]);
        g = map(i, 0, 255, colorEscena[escenaActual][VERDE], colorEscena[proxEscena][VERDE]);
        b = map(i, 0, 255, colorEscena[escenaActual][AZUL], colorEscena[proxEscena][AZUL]);
        pixels.setPixelColor(0, pixels.Color(pixels.gamma8 (r), pixels.gamma8 (g), pixels.gamma8 (b)));
        pixels.show();
        checkBattery(); // comprueba bateria
        LowPower.powerDown(SLEEP_60MS, ADC_OFF, BOD_OFF); // pausa 60ms
        if (imanDetectado()) // reiniciar transicion al reactivar
            i = 0;
    }
    digitalWrite (LED_BUILTIN, LOW); // apaga LED fuera de la transicion
}
```

Durmiendo

Al detectar un imán en el Reed entraremos en un sueño profundo en que apagaremos todos los LED, los Neopixels con `clear()` y los otros con `digitalWrite()`, y después de una pequeña espera para que dé tiempo a retirar el imán, con `dormirProfundo()` pondremos al micro en bajo consumo y activará la interrupción para que una nueva detección nos despierte.

Al despertar, recuperamos la iluminación según la escena actual y esperamos un poco a retirar el imán.

```
bool imanDetectado() {
    byte r, g, b;
    if (digitalRead(REED_PIN) == LOW) { // comprueba reed
        pixels.clear(); // apaga Neopixel y LED
        pixels.show();
        digitalWrite(FAROLA_PIN, LOW);
        digitalWrite (LED_BUILTIN, LOW);
        LowPower.powerDown(SLEEP_4S, ADC_OFF, BOD_OFF); // espera a retirar iman
        dormirProfundo();
        if ((escenaActual == NOCHE) || (escenaActual == AMANECER))
            digitalWrite(FAROLA_PIN, HIGH);
        r = colorEscena[escenaActual][ROJO];
        g = colorEscena[escenaActual][VERDE];
        b = colorEscena[escenaActual][AZUL];
        pixels.setPixelColor(0, pixels.Color(pixels.gamma8 (r), pixels.gamma8 (g), pixels.gamma8 (b)));
        pixels.show();
        LowPower.powerDown(SLEEP_4S, ADC_OFF, BOD_OFF); // espera a retirar iman
        return true;
    }
    else
        return false;
}
```

Siesta

Las pausas largas como la duración del día o de la noche en lugar de con `delay()` las voy a hacer con `LowPower.powerDown()` para dormir el micro y gastar menos. Como la pausa más larga que podemos obtener es de 8 segundos hay que ir dando cabezaditas hasta llegar al tiempo de pausa total. Para afinar un poco más haremos las pausas restantes de menos de 8 segundos, no es imprescindible pero así puedo ajustar un poco más.

Cuando despertemos comprobamos el reed para finalizar la pausa, luego en el cambio de escena ya entraremos en sueño profundo. Tiene el inconveniente de que podemos estar hasta 8 segundos con el imán en el reed antes de que finalice pero el consumo es menor con pausas largas.

```
void dormirSiesta (unsigned int segundos, escenas escena) {
    escenaActual = escena;
    while (segundos > 8) {
        LowPower.powerDown(SLEEP_8S, ADC_OFF, BOD_OFF); // da cabezaditas de 8 segundos
        segundos = (imanDetectado()) ? 0 : segundos - 8;
    }
    if (segundos > 4) {
        LowPower.powerDown(SLEEP_4S, ADC_OFF, BOD_OFF); // resto de cabezaditas hasta tiempo de espera
        segundos = (imanDetectado()) ? 0 : segundos - 4;
    }
    if (segundos > 2) {
        LowPower.powerDown(SLEEP_2S, ADC_OFF, BOD_OFF);
        segundos = (imanDetectado()) ? 0 : segundos - 2;
    }
    if (segundos > 1)
        LowPower.powerDown(SLEEP_1S, ADC_OFF, BOD_OFF);
}
```

Ciclo diario

Ya sólo queda ejecutar las transiciones en el orden adecuado para mostrar la iluminación del ciclo diario.

En nuestra latitud hay luz durante el día entre 9 y 15 horas (solsticios de invierno y verano) lo que representa entre un 40% y un 60% de la duración total del día así que no necesitamos establecer la misma duración para el día y la noche.

Podemos variar los valores de duración entre esos márgenes según queramos representar una época del año en nuestro tarro de cristal.

```
const unsigned int tiempoDia = 80; // duracion del dia en segundos
const unsigned int tiempoNoche = 112; // duracion de la noche en segundos

void loop() {
    digitalWrite (FAROLA_PIN, LOW); // Apagar farola al alba
    transicion(DIA); // El Sol va saliendo
    dormirSiesta(tiempoDia, ATARDECER); // Esperar hasta el atardecer
    transicion(ATARDECER); // El Sol va bajando
    digitalWrite (FAROLA_PIN, HIGH); // Encender farola al atardecer
    transicion(NOCHE); // El Sol se esconde
    dormirSiesta(tiempoNoche, AMANECER); // Esperar hasta el amanecer
    transicion(AMANECER); // Los primeros rayos del Sol
}
```

El programa

El programa completo queda así:

```
// Diorama en un tarro de cristal - Scene in a jar -- Paco Cañada 2022 -- https://usuaris.tinet.cat/fmco/
// Usa un Arduino Pro Mini
// Vamos a trabajar a 8MHz
// Compilar siempre el sketch seleccionando Arduino Pro Mini - ATmega 328P (3.3V, 8MHz)
// Comentar o borrar la siguiente linea si usamos un Arduino Pro Mini a 8MHz
#define XTAL_16MHz

#include <Adafruit_NeoPixel.h>
#include <LowPower.h>
#include <avr/power.h>

#define NUMPIXELS      1
#define NEOPIXEL_PIN   6
#define FAROLA_PIN     7
#define REED_PIN        2           // pin de interrupcion D2 o D3

#define AREF_mV         1100        // VBG referencia interna 1100 mV
#define BATT_MIN        3500        // Tension minima bateria para aviso

const unsigned int tiempoDia    = 80; // duracion del dia en segundos
const unsigned int tiempoNoche = 112; // duracion de la noche en segundos

const unsigned long BAND_GAP = 1024UL * AREF_mV;

enum leds      {ROJO, VERDE, AZUL};
enum escenas   {DIA, ATARDECER, NOCHE, AMANECER};

escenas escenaActual;

byte colorEscena[][3] = {           // colores iniciales de la transicion
{160, 50, 0},                      // amanecer -> dia
{255, 255, 255},                   // dia -> atardecer
{200, 50, 110},                    // atardecer -> noche
{0, 0, 255}                        // noche -> amanecer
};

byte batteryFlash;

Adafruit_NeoPixel pixels(NUMPIXELS, NEOPIXEL_PIN, NEO_GRB + NEO_KHZ800);

void setup() {
#ifndef XTAL_16MHz
  clock_prescale_set(clock_div_2);    // Permite dividir por 1, 2, 4, 8, 16, 32, 64, 128 y 256
#endif
  pinMode(LED_BUILTIN, OUTPUT);       // LED aviso bateria baja
  pinMode(FAROLA_PIN, OUTPUT);
  pinMode(REED_PIN, INPUT_PULLUP);
  digitalWrite(LED_BUILTIN, LOW);
  digitalWrite(FAROLA_PIN, LOW);
  pixels.begin();                  // inicializa pin Neopixels
}

void loop() {
  digitalWrite(FAROLA_PIN, LOW);      // Apagar farola al alba
  transicion(DIA);                 // El Sol va saliendo
  dormirSiesta(tiempoDia, ATARDECER); // Esperar hasta el atardecer
  transicion(ATARDECER);           // El Sol va bajando
  digitalWrite(FAROLA_PIN, HIGH);    // Encender farola al atardecer
  transicion(NOCHE);               // El Sol se esconde
  dormirSiesta(tiempoNoche, AMANECER); // Esperar hasta el amanecer
  transicion(AMANECER);            // Los primeros rayos del Sol
}

void transicion(escenas escena) {
  byte r, g, b, i;
  escenas proxEscena;
  escenaActual = escena;
  proxEscena = (escena == AMANECER) ? DIA : escena + 1;
  for (i = 0; i < 255; i++) {
    r = map(i, 0, 255, colorEscena[escenaActual][ROJO], colorEscena[proxEscena][ROJO]);
    g = map(i, 0, 255, colorEscena[escenaActual][VERDE], colorEscena[proxEscena][VERDE]);
    b = map(i, 0, 255, colorEscena[escenaActual][AZUL], colorEscena[proxEscena][AZUL]);
    pixels.setPixelColor(0, pixels.Color(pixels.gamma8(r), pixels.gamma8(g), pixels.gamma8(b)));
    pixels.show();
    checkBattery();                // comprueba bateria
    LowPower.powerDown(SLEEP_60MS, ADC_OFF, BOD_OFF);    // pausa 60ms
    if (imanDetectado())          // reiniciar transicion al reactivar
      i = 0;
  }
  digitalWrite(LED_BUILTIN, LOW);    // apaga LED fuera de la transicion
}
```

```

void dormirSiesta (unsigned int segundos, escenas escena) {
    escenaActual = escena;
    while (segundos > 8) {
        LowPower.powerDown(SLEEP_8S, ADC_OFF, BOD_OFF); // da cabezaditas de 8 segundos
        segundos = (imanDetectado()) ? 0 : segundos - 8;
    }
    if (segundos > 4) {
        LowPower.powerDown(SLEEP_4S, ADC_OFF, BOD_OFF); // resto de cabezaditas hasta tiempo de espera
        segundos = (imanDetectado()) ? 0 : segundos - 4;
    }
    if (segundos > 2) {
        LowPower.powerDown(SLEEP_2S, ADC_OFF, BOD_OFF);
        segundos = (imanDetectado()) ? 0 : segundos - 2;
    }
    if (segundos > 1)
        LowPower.powerDown(SLEEP_1S, ADC_OFF, BOD_OFF);
}

void dormirProfundo() {
    attachInterrupt(digitalPinToInterrupt(REED_PIN), despierta, LOW);
    LowPower.powerDown(SLEEP_FOREVER, ADC_OFF, BOD_OFF);
    detachInterrupt(digitalPinToInterrupt(REED_PIN));
}

void despierta () { // rutina de interrupcion externa
} // no necesita hacer nada

bool imanDetectado() {
    byte r, g, b;
    if (digitalRead(REED_PIN) == LOW) { // comprueba reed
        pixels.clear(); // apaga Neopixel y LED
        pixels.show();
        digitalWrite(FAROLA_PIN, LOW);
        digitalWrite(LED_BUILTIN, LOW);
        LowPower.powerDown(SLEEP_4S, ADC_OFF, BOD_OFF); // espera a retirar iman
        dormirProfundo();
        if ((escenaActual == NOCHE) || (escenaActual == AMANECER))
            digitalWrite(FAROLA_PIN, HIGH);
        r = colorEscena[escenaActual][ROJO];
        g = colorEscena[escenaActual][VERDE];
        b = colorEscena[escenaActual][AZUL];
        pixels.setPixelColor(0, pixels.Color(pixels.gamma8 (r), pixels.gamma8 (g), pixels.gamma8 (b)));
        pixels.show();
        LowPower.powerDown(SLEEP_4S, ADC_OFF, BOD_OFF); // espera a retirar iman
        return true;
    }
    else
        return false;
}

unsigned int readVcc() {
    unsigned int result;
    // REFS0 : Selects AVcc external reference
    // MUX3 MUX2 MUX1 : Selects 1.1V (VBG)
    ADMUX = bit (REFS0) | bit (MUX3) | bit (MUX2) | bit (MUX1); // Voltage Reference: AVCC, Input Channel: 1.1V(VBG)
    delay(2); // Wait for Vref to settle
    ADCSRA |= bit( ADSC ); // start conversion
    while (ADCSRA & bit (ADSC)); // wait until done
    result = ADC; // read inaccurate result
    ADCSRA |= bit( ADSC ); // Start conversion, second time is a charm
    while (ADCSRA & bit (ADSC)); // wait until done
    result = BAND_GAP / ADC; // Back-calculate AVcc in mV; Vcc(in mV)=1024×1100/ADC
    return result;
}

void checkBattery() { // cada 60ms durante transiciones
    batteryFlash++;
    if ((readVcc() < BATT_MIN) && ((batteryFlash & 0x0F) == 0)) // flash cada segundo (60ms * 16 = 960ms)
        digitalWrite (LED_BUILTIN, HIGH);
    else
        digitalWrite (LED_BUILTIN, LOW);
}

```

Consumos

Estas son las mediciones de consumo de las diferentes placas de Arduino que tengo cuando ejecutan un sketch vacío:

	Arduino Uno			Arduino Nano			Arduino Pro Mini		
	9V (VIN)	5V	3,7V	9V (VIN)	5V	3,7V	9V (VIN)	5V	3,7V
Normal	25,7 mA	25,0 mA	13,3 mA	19,8 mA	19,5 mA	10,8 mA	15,1 mA	14,9 mA	7,9 mA
8MHz	20,8 mA	20,3 mA	10,2 mA	14,8 mA	14,3 mA	7,8 mA	12,1 mA	12,1 mA	6,0 mA
Idle	16,5 mA	16,0 mA	7,2 mA	11,3 mA	10,8 mA	5,4 mA	8,2mA	8,2 mA	3,3 mA
Power Down	8,7 mA	8,6 mA	5,1 mA	4,9 mA	4,8 mA	3,1 mA	3,1 mA	3,1 mA	1,8 mA

Para ver si ha valido la pena aplicar todas estas medidas para aumentar la autonomía en mi diorama he comparado el consumo entre hacer las pausas con `delay()` y utilizando la librería `LowPower` como en el programa.

	delay()	LowPower.powerDown()	LowPower.powerDown() Sin LED alimentación
Consumo noche	29,6mA	25,2mA	21,6mA
Consumo día	44,3mA	39,4mA	36,1mA
Duracion batería (300mAh)	5h13'	6h22'	7h12'

NOTA: He quitado el LED que indica la alimentación de la placa (en realidad, su resistencia que ha resultado más fácil) para tener más autonomía.

Aunque para esta aplicación el aumento de la duración de la batería no es espectacular debido a que siempre tenemos un consumo elevado por los LED, para aplicaciones con sensores que solo se activen un tiempo para enviar los datos de los mismos la duración se puede ver aumentada significativamente.

Video

Video: <https://www.youtube.com/watch?v=YnBe4HN7AjQ>



22. Pantalla táctil: TCO Xpressnet táctil

La construcción de un Tablero de Control Óptico (TCO) siempre es una tarea complicada:

- En los montajes analógicos hay que cablear una cantidad importante de hilos desde el panel a los desvíos. Para señalizar la posición del desvío si utilizamos pulsadores además de la conexión de los LED correspondientes hay que tener los desvíos de bobina con final de carrera o bien cablear relés biestables. Si utilizamos interruptores de palanca para indicar la posición seguramente tendremos que incorporar unidades CDU (Capacitor Discharge Unit /Unidad de descarga de condensador) para accionar los desvíos.
- En digital el cableado se simplifica algo ya que los desvíos se conectan a los decodificadores y los pulsadores e indicadores se conectan a retromódulos y decodificadores o a aparatos especiales comerciales o autoconstruidos (como el [Lanza Rutas del capítulo 17](#)) para que la central envíe las oportunas órdenes DCC para los decodificadores.
- La simplificación máxima la tenemos cuando usamos una central, normalmente cara, con pantalla gráfica táctil dónde dibujar nuestro TCO y accionarlo (ESU ECoS, Viessmann Commander, CS3, etc.) o bien lo hacemos a través de la pantalla del móvil o tableta con una aplicación compatible con nuestra central (Z21, etc.)

La idea

Voy a usar una pantalla gráfica táctil a color donde dibujaré el TCO de mis módulos, al estilo de la pantalla de la ECoS, para accionarlos 'digitalmente' y controlarlos desde mi sencilla central Xpressnet (NanoX, Multimaus, LZV100, DR5000, etc.).

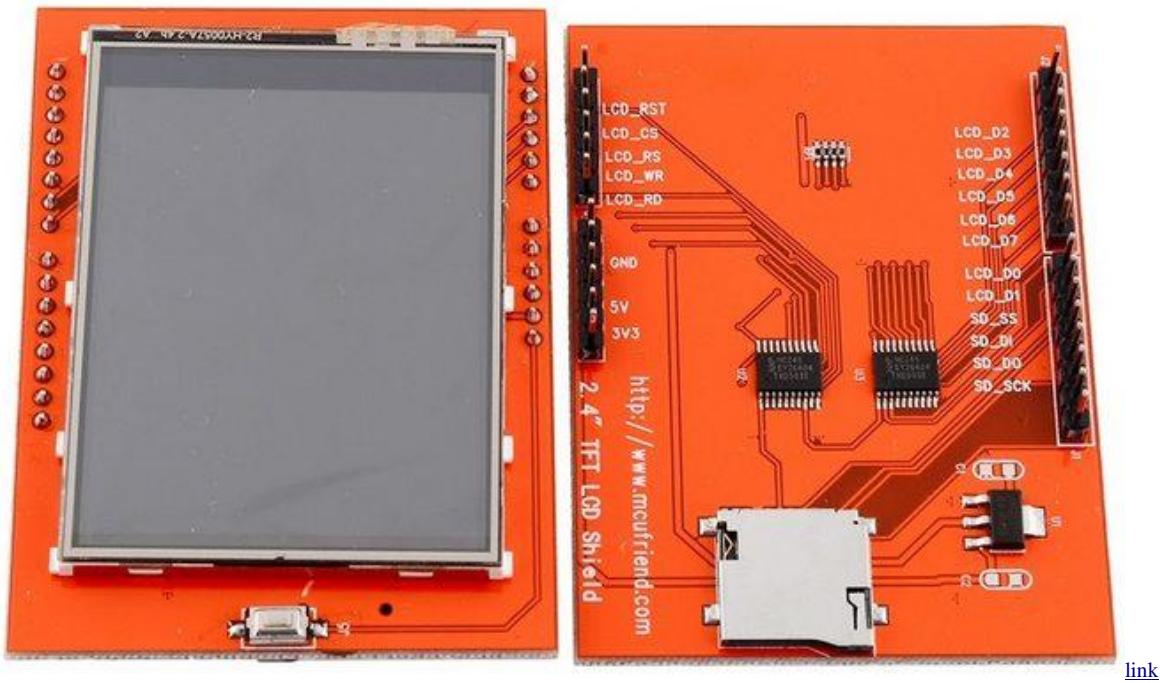


El Hardware

Para nuestro proyecto:

- Como pantalla gráfica usaremos una pantalla TFT en color de las que hay disponibles en versiones de pequeño formato alrededor 2,4" o 3,5" que sería el tamaño de un Arduino Uno. Se suelen conectar por el bus SPI y tener una resolución de 320x240 pixel o 480x320 como los primeros Smartphone.
- Hay versiones de pantalla TFT táctil y no táctil, para este proyecto la elegiré táctil. Hay modelos con y sin chip especializado.
- Para el control de los desvíos usaré mi central DCC conectada por Xpressnet y los decodificadores de accesorios instalados en mis módulos.

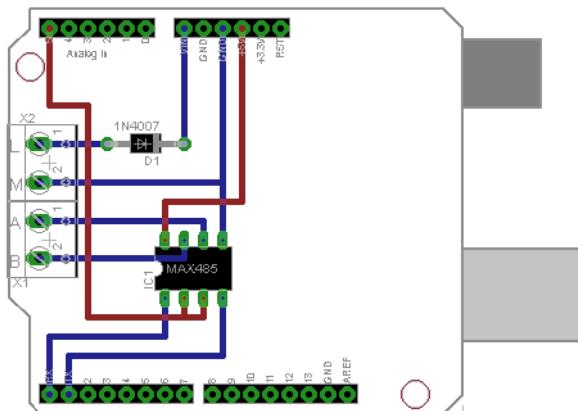
Las pantallas TFT funcionan a 3.3V por lo que se necesita instalar resistencias o adaptadores de tensión para usarlo con Arduino, afortunadamente he encontrado una shield para Arduino con una pantalla TFT táctil de 2,4" (320x240 pixel) con el chip ILI9341 que además tiene un conector para SD que me simplifica la conexión:



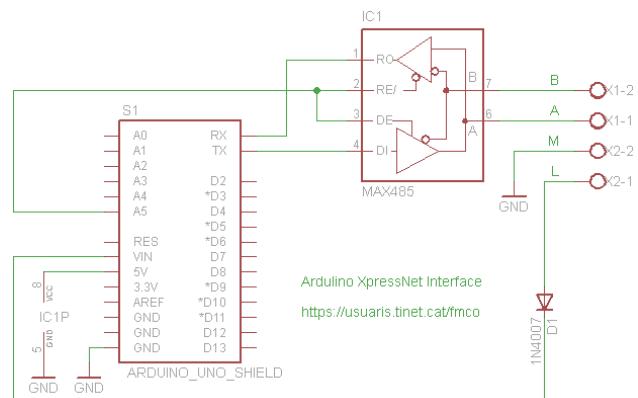
[link](#)

La desventaja es que el bus SPI solo está conectado a la SD y la pantalla usa el modo paralelo para comunicarse a más velocidad lo que hace que ocupe prácticamente todos los pines de un Arduino Uno, si usamos la SD solo deja libres D0, D1 y A5 los mínimos para el interface Xpressnet.

Para el interface Xpressnet recurriremos al circuito con el MAX485 que ya hemos usado en otros montajes.



[link](#)



[link](#)

Librerías

Para controlar estas pantallas que no se comunican por SPI sino en modo 8-bit paralelo hay dos librerías:

- La [Adafruit_TFTLCD.h](https://github.com/adafruit/TFTLCD.h) (<https://github.com/adafruit/TFTLCD-Library>) que soporta los chips ILI9325, ILI9328, ILI7575, ILI9341 y HX8357D
- La [MCUFRIEND_kbv.h](https://github.com/prenticedavid/MCUFRIEND_kbv.h) (https://github.com/prenticedavid/MCUFRIEND_kbv) que soporta muchos más chips como el ILI9341, ILI9488, HX8357D, R6105, etc. Además trae ejemplos para averiguar el chip de la pantalla y los pines a los que se conecta el panel táctil.

Ambas necesitan de la librería [Adafruit_GFX.h](https://github.com/adafruit/Adafruit-GFX-Library) (<https://github.com/adafruit/Adafruit-GFX-Library>) para funcionar.

Ya que mi chip es el ILI9341 y la [Adafruit_TFTLCD.h](https://github.com/adafruit/TFTLCD.h) es más ligera voy a utilizar preferentemente esta.

```
#include <Adafruit_GFX.h>
#include <Adafruit_TFTLCD.h>

#define LCD_CS A3
#define LCD_CD A2
#define LCD_WR A1
#define LCD_RD A0
#define LCD_RESET A4
Adafruit_TFTLCD tft(LCD_CS, LCD_CD, LCD_WR, LCD_RD, LCD_RESET); // Usar Adafruit que es mas ligera
```

pero si nuestro controlador no está soportado por esta o lo desconocemos usaremos la otra.

```
#include <Adafruit_GFX.h>
#include <MCUFRIEND_kbv.h>
MCUFRIEND_kbv tft; // Libreria de graficos v1.5.0
// Libreria especifica para el Hardware v2.9.9
// MCUFRIEND para pantallas no soportadas por Adafruit
```

Para inicializarla en el `setup()` tenemos que llamar a la función `begin()` con el tipo de pantalla, para la ILI9341 es el valor 0x9341 pero la función `readID()` se encargará de ello.

```
unsigned int ID; // ID de la pantalla
ID = tft.readID(); // Iniciamos el LCD especificando el controlador ILI9341 (0x9341).
```

El panel táctil de mi shield es resistivo y se controla con 4 pines, dos analógicos y dos digitales. La librería que usaremos ya que no tiene chip especializado es la `TouchScreen.h` (https://github.com/adafruit/Adafruit_TouchScreen)

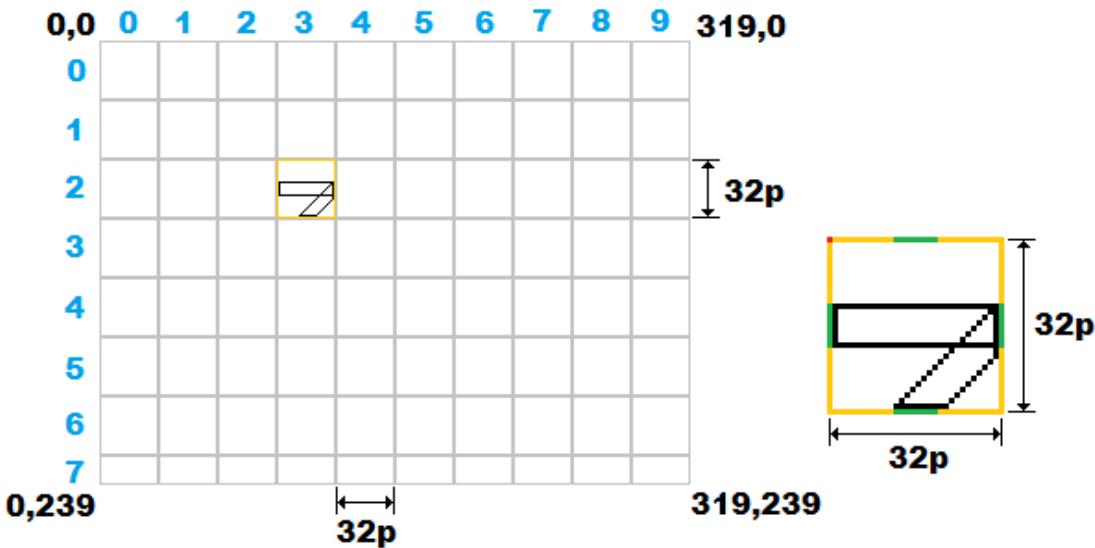
Parece ser que en este tipo de shields cada fabricante usa diferentes conexiones y pueden estar conectados a D8, D9, A2 y A3 (como la mía) o bien a D6, D7, A1 y A2. Además no siempre coinciden las coordenadas del panel táctil con las de la pantalla y pueden estar giradas o invertidas unas respecto a las otras por lo que habrá que tenerlo en cuenta.

Para el interface Xpressnet que montaré en una shield de conexiones usaremos la librería `Xpressnet.h`

Las coordenadas

Nuestra pantalla TFT es de 320x240 pixels con el origen de coordenadas (0,0) en la esquina superior izquierda en formato vertical pero para posicionar los objetos en pantalla de nuestro TCO nos va mejor en formato apaisado, la función `setRotation()` con valores 1 o 3 nos pondrá la pantalla en formato apaisado.

Además como serán cuadrículas trabajaremos mejor usando columnas y filas en lugar de pixels. Para el tamaño de nuestra pantalla he pensado que con unas cuadrículas de 32x32 pixels nos permiten dibujar un pequeño y sencillo TCO de 10 columnas y 7 filas y aún queda la altura de media fila para poder escribir el título de nuestra pantalla.



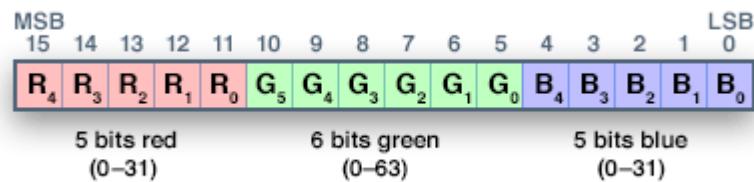
[link](#)

La cuadrícula básica de 32x32 será la que contenga un ícono representando los elementos de vía. Para que las vías encajen, los dibujos deben conectar en las mismas zonas (en verde).

Observad que para la curva del desvío he tenido que ampliar un poco el ancho para que ópticamente resulte del mismo grosor que la parte recta.

Los colores

Nuestra pantalla permite dibujar cada pixel individual de un color diferente con valores RGB pero a diferencia de lo habitual (un byte por componente de color) solo usa dos bytes para almacenar los componentes de color en formato 565:



link

Por ejemplo, algunos colores para nuestra pantalla:

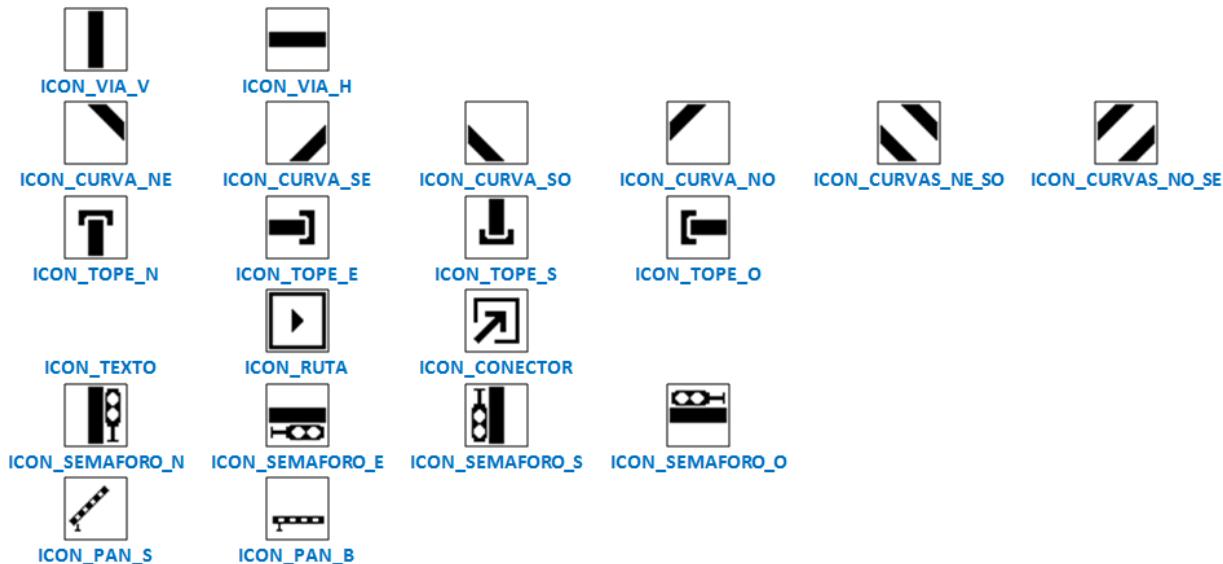
```
#define BLACK 0x0000
#define BLUE 0x001F
#define RED 0xF800
#define GREEN 0x07E0
#define CYAN 0x07FF
#define MAGENTA 0xF81F
#define VIOLET 0x9199
#define BROWN 0x8200
#define PINK 0xF97F
#define YELLOW 0xFFE0
#define WHITE 0xFFFF
#define ORANGE 0xFD20
#define LIME 0x87E0
#define GREENYELLOW 0xAFE5
#define AQUA 0x5D1C
#define NAVY 0x000F
#define DARKGREEN 0x03E0
#define DARKCYAN 0x03EF
#define MAROON 0x7800
#define PURPLE 0x780F
#define OLIVE 0x7BE0
#define SILVER 0xA510
#define GOLD 0xA508
#define LIGHTGREY 0xC618
#define DARKGREY 0x7BEF
```

Esto significa que cada ícono a todo color necesita 2048 bytes ($32 \times 32 \times 2$) para almacenarse en la memoria, si la memoria de nuestro Arduino Uno es de 32768 bytes y tienen que entrar los íconos y el programa vemos que es una misión imposible.

Realmente no necesitamos iconos a todo color, si miramos el TCO en una ESU ECoS vemos que son bastante monocromáticos por lo que si usamos un solo color, un byte puede contener información de 8 pixels (encendido/apagado) y un ícono sólo ocupará 128 bytes ($32 \times 32 / 8$). La función `drawBitmap()` de la librería [Adafruit_GFX.h](#) nos permitirá dibujarlos de una forma sencilla.

Los iconos

Con el Paint elegimos un tamaño de 32x32 pixels, dibujamos los iconos, sólo con blanco y negro, y guardamos cada uno en un archivo con formato .bmp de 24bits:



[link](#)

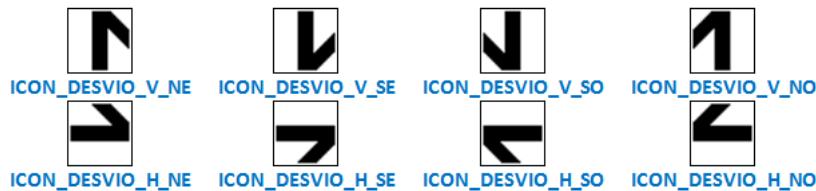
El orden no es casual, los he dispuesto según la orientación en sentido de las agujas del reloj, y primero los que no cambian el aspecto al tocarlos (vías, curvas, topes y auxiliares) y luego los que cambian (semáforos, paso a nivel y desvíos).

El texto no tiene icono. La ruta nos moverá varios accesorios a la vez y el conector nos servirá para cambiar de pantallas ya que una solo contiene un número limitado de símbolos.

Unas enumeraciones me ayudarán luego a seleccionar la orientación de un ícono.

```
enum posViaPaN { VERTICAL, HORIZONTAL };
enum orientacion { NORTE, ESTE, SUR, OESTE };
enum iconoOrientacion { NORDESTE, SUDESTE, SUDOESTE, NOROESTE, NE_SO, NO_SE };
```

Si os dais cuenta no están los iconos para los desvíos ya que necesitan dos colores, para la posición activa y la no activa. Estos los dibujaré usando los iconos de vía recta y los de curva pintando de un color la posición no activa y luego de otro la posición activa.



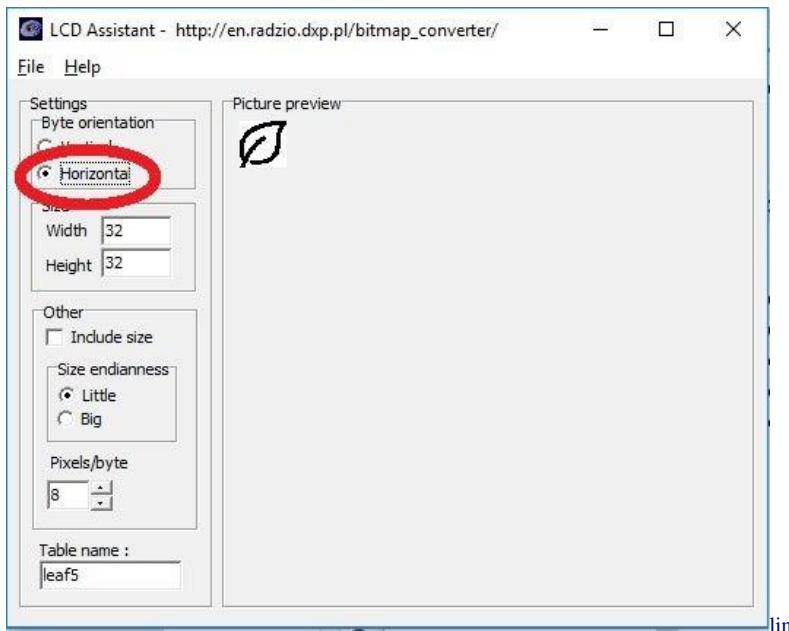
[link](#)

```
enum tipoSimbolo {
    ICON_VIA_V, ICON_VIA_H, // con ícono, sin dirección
    ICON_CURVA_NE, ICON_CURVA_SE, ICON_CURVA_SO, ICON_CURVA_NO, ICON_CURVAS_NE_SO, ICON_CURVAS_NO_SE,
    ICON_TOPE_N, ICON_TOPE_E, ICON_TOPE_S, ICON_TOPE_O, // sin ícono, sin dirección
    ICON_TEXTO, // con ícono,
    ICON_RUTA,
    ICON_CONECTOR,
    ICON_SEMAFORO_N, ICON_SEMAFORO_E, ICON_SEMAFORO_S, ICON_SEMAFORO_O, // con ícono, con dirección
    ICON_PAN_S, ICON_PAN_B,
    ICON_DESVIO_V_NE, ICON_DESVIO_V_SE, ICON_DESVIO_V_SO, ICON_DESVIO_V_NO, //sin ícono, con dirección
    ICON_DESVIO_H_NE, ICON_DESVIO_H_SE, ICON_DESVIO_H_SO, ICON_DESVIO_H_NO,
};
```

La función `drawBitmap()` necesita que los datos de los iconos sean una array de caracteres almacenados en la memoria Flash (la ROM) de nuestro Arduino. Cuando queremos almacenar una constante o variable en la memoria Flash añadimos después del nombre la palabra `PROGMEM`

Para convertir nuestros archivos .bmp en un array de caracteres tenemos el programa LCD Assistant que nos genera un archivo de texto con los datos necesarios:

[https://en.radzio.dxp.pl\(bitmap_converter/](https://en.radzio.dxp.pl(bitmap_converter/)



[link](#)

Y más simple, la pagina web image2cpp (<http://javl.github.io/image2cpp/>) con más opciones (múltiples archivos, escalado, centrado, etc..) que nos permite subir el archivo y obtener el código necesario:

```
// 'semaforoN' 32x32px
const unsigned char icon_semaforoN [] PROGMEM = {
    0x00, 0x00, 0x00, 0x00, 0xf0, 0x00, 0x00, 0xf3, 0xf0, 0x00, 0x0f, 0xf3, 0xf0,
    0x00, 0x0f, 0xf7, 0x38, 0x00, 0xf0, 0xf6, 0x18, 0x00, 0x0f, 0x4, 0x08, 0x00, 0x0f, 0xf4, 0x08,
    0x00, 0x0f, 0xf4, 0x08, 0x00, 0xf0, 0xf6, 0x18, 0x00, 0x0f, 0xf7, 0x38, 0x00, 0x0f, 0xf7, 0x08,
    0x00, 0x0f, 0xf7, 0x38, 0x00, 0xf0, 0xf6, 0x18, 0x00, 0x0f, 0xf4, 0x08, 0x00, 0x0f, 0xf4, 0x08,
    0x00, 0x0f, 0xf4, 0x08, 0x00, 0xf0, 0xf6, 0x18, 0x00, 0x0f, 0xf7, 0x38, 0x00, 0x0f, 0xf7, 0x08,
    0x00, 0x0f, 0xf3, 0xf8, 0x00, 0x0f, 0xf3, 0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00,
    0x00, 0x0f, 0xf0, 0xc0, 0x00, 0x0f, 0xf3, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00
};

// Array of all bitmaps for convenience. (Total bytes used to store images in PROGMEM = 144)
```

[link](#)

Estos son los datos de nuestro ícono aptos para la función `drawBitmap()`:

```
const unsigned char icon_semaforoN [] PROGMEM = {
  0x00, 0x00, 0x00, 0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf3, 0xf0, 0x00, 0x0f, 0xf3, 0xf0,
  0x00, 0x0f, 0xf7, 0x38, 0x00, 0x0f, 0xf6, 0x18, 0x00, 0x0f, 0xf4, 0x08, 0x00, 0x0f, 0xf4, 0x08,
  0x00, 0x0f, 0xf4, 0x08, 0x00, 0x0f, 0xf6, 0x18, 0x00, 0x0f, 0xf7, 0x38, 0x00, 0x0f, 0xf7, 0xf8,
  0x00, 0x0f, 0xf7, 0x38, 0x00, 0x0f, 0xf6, 0x18, 0x00, 0x0f, 0xf4, 0x08, 0x00, 0x0f, 0xf4, 0x08,
  0x00, 0x0f, 0xf4, 0x08, 0x00, 0x0f, 0xf6, 0x18, 0x00, 0x0f, 0xf7, 0x38, 0x00, 0x0f, 0xf7, 0xf8,
  0x00, 0x0f, 0xf3, 0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf3, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0xc0,
  0x00, 0x0f, 0xf0, 0xc0, 0x00, 0x0f, 0xf0, 0xc0, 0x00, 0x0f, 0xf0, 0xc0, 0x00, 0x0f, 0xf0, 0xc0,
  0x00, 0x0f, 0xf0, 0xc0, 0x00, 0x0f, 0xf3, 0xf0, 0x00, 0x0f, 0xf0, 0xc0, 0x00, 0x0f, 0xf0, 0xc0,
  0x00, 0x0f, 0xf0, 0xc0, 0x00, 0x0f, 0xf0, 0xc0, 0x00, 0x0f, 0xf0, 0xc0, 0x00, 0x0f, 0xf0, 0xc0
};
```

El ícono para el texto es un ícono vacío:

```
const unsigned char icon_vacio [] PROGMEM = {
```

Para incorporar los datos de todos los íconos al programa voy a ubicarlos en un archivo `iconos.h` así no tendrá tanta información en el archivo .ino y sólo necesitaré incluirlo de forma parecida a una librería:

```
#include "iconos.h" // iconos 32x32
```

Haciendo un array con las referencias (punteros) a los array de datos de cada ícono será sencillo pintar un ícono con la función `drawBitmap()` escogiendo un elemento del array.

```
const unsigned char* iconos[] = {
  icon_viaV, icon_viaH, icon_curvaNE, icon_curvaSE, icon_curvaSO, icon_curvaNO, icon_curvaNE_SO,
  icon_curvaNO_SE,
  icon_topeN, icon_topeE, icon_topeS, icon_topeO, icon_vacio, icon_ruta, icon_conector,
  icon_semaforoN, icon_semaforoE, icon_semaforoS, icon_semaforoO, icon_PaNsub, icon_PaNbaj,
};
```

Observad que están en el mismo orden que la enumeración `tipoSimbolo` y están sólo los que tienen ícono, ya que los desvíos son en realidad dos íconos por eso los dispuse al final. Así para seleccionar el ícono del semáforo con orientación Norte lo haríamos así:

```
iconos[ICON_SEMAFORO_N]
```

para el ícono semáforo con orientación Sur podríamos hacerlo de estas dos maneras:

```
iconos[ICON_SEMAFORO_S] // indice ícono
iconos[ICON_SEMAFORO_N + SUR] // indice ícono base + orientacion
```

Definiendo...

Como queremos tener varias pantallas pasando de una a otra al pinchar el ícono conector podemos pintarlas de varias maneras.

La más simple, nivel principiante, es tener una función por pantalla o una función con un `switch()` según el número de pantalla e ir usando `drawBitmap()` por cada ícono de la pantalla para ubicarlos. La función `fillScreen()` llena toda la pantalla de un color, nos sirve para borrar la pantalla. Los textos se posicionan con `setCursor()` y se imprimen con `print()`, se puede elegir el tamaño con `setTextSize()` y el color con `setTextColor()`:

```
void pintaPantalla0 () {
tft.fillScreen(WHITE);
// tft.drawBitmap(x, y, ícono, ancho, alto, color);
tft.drawBitmap(64, 96, iconos[ICON_SEMAFORO_N], 32,32, BLACK);
tft.drawBitmap(96, 96, iconos[ICON_VIA_H], 32,32, BLACK);
tft.drawBitmap(128, 96, iconos[ICON_CONECTOR], 32,32, DARKCYAN);
tft.setTextSize(2);
tft.setTextColor(BLUE);
tft.setCursor (100, 224);
tft.print ("PANTALLA 0");
}
```

Es bastante engorroso de escribir y mantener al trabajar con pixel y tener que ir introduciendo todos los datos. Además cuando pinchemos en un desvío o semáforo para cambiar su aspecto, con otra función, seguramente compleja, tendremos que saber todos los datos de su posición y aspecto para pintarlo correctamente, algo que aquí no hemos tenido en cuenta.

Otra forma es tener una estructura para contener los datos de cada ícono (tipo, posición y dirección DCC del mismo) para que luego sea más fácil trabajar con ellos.

```
struct Simbolo {  
    byte tipo; // ícono  
    byte posX; // columna: 0..9  
    byte posY; // fila: 0..6  
    unsigned int direccion; // dirección DCC: 1..1024 (0..1023 Xpressnet)  
};
```

Así el circuito no es más que un array de estas estructuras y para pintar todos los íconos de la pantalla podemos hacer una función genérica:

```
#define NO_ASIGNADO 0x0F00  
  
const Simbolo pantalla0[] = {  
{ICON_SEMAFORO_N, 2, 3, 10},  
{ICON_VIA_H, 3, 3, NO_ASIGNADO},  
{ICON_CONECTOR, 4, 3, NO_ASIGNADO + 1},  
};  
  
#define NUM_SIMBOLOS 3  
  
void pintaPantallaSimbolos (Simbolo *pantalla, int numSimbolos) {  
    int x, y, icono, n;  
    for (n=0; n < numSimbolos; n++) {  
        x = pantalla[n].posX * 32;  
        y = pantalla[n].posY * 32;  
        icono = pantalla[n].tipo;  
        tft.drawBitmap (x, y, iconos[icono], 32, 32, BLACK);  
    }  
}  
  
pintaPantallaSimbolos (pantalla0, NUM_SIMBOLOS);
```

Esto es más fácil de mantener y modificar y nos permite hacer búsquedas de la posición que ocupa cada símbolo y la dirección DCC que tiene. Usamos un valor imposible de dirección (mayor de 1024) para los íconos que no tienen dirección DCC (vía, conector, etc.) para que las búsquedas no lo interpreten erróneamente: byte alto como 0x0F y el byte bajo para almacenar un parámetro como en el caso de [CONECTOR](#). Queda por resolver el tema de los textos, posiciones y los colores pero vemos la idea.

El número de símbolos a pintar para pasarlo a la función podemos contarlos manualmente o dejar al compilador que lo haga.

La función [`sizeof\(\)`](#) devuelve el tamaño de la variable que se le pase como parámetro, en el caso del array, el tamaño total del array. Pero también podemos pasárle un elemento del array para que nos diga su tamaño. Si dividimos el tamaño del array por el tamaño de un elemento, en este caso será la longitud de la estructura [Simbolo](#), obtendremos el número de elementos del array.

```
const int numSimbolosPantalla0 = sizeof(pantalla0) / sizeof(pantalla0[0]);  
  
pintaPantallaSimbolos(pantalla0, numSimbolosPantalla0);
```

Hasta ahora hemos utilizado la directiva del compilador [`#define`](#) para definir unos valores constantes como si se hubieran hecho con [`const`](#). En realidad son cosas diferentes, [`const`](#) es del lenguaje y va ligado a un tipo de datos, [`#define`](#) es del compilador y hace una sustitución de una palabra por lo que viene a continuación en la línea sea lo que sea, aquí no hay comprobación de tipos y pueden darse errores difíciles de ver por un uso inadecuado de tipo de datos.

```
#define numSimbolosPantalla0 sizeof(pantalla0) / sizeof(pantalla0[0])  
  
pintaPantallaSimbolos(pantalla0, numSimbolosPantalla0);
```

Los [`#define`](#) también pueden tener parámetros y hacen la sustitución de esos parámetros en el resto de la línea. Cuando encuentre la palabra en el resto del programa lo expandirá con esa línea modificada. Observad que no hay espacios entre el [`\(`](#) y el nombre del [`#define`](#).

```
#define LONG_PANTALLA(x) sizeof(x)/sizeof(x[0])  
  
pintaPantallaSimbolos(pantalla0, LONG_PANTALLA(pantalla0));
```

Usando adecuadamente este uso de **#define** nos puede servir para llenar la estructura **Simbolo** dejando un código mas legible:

```
#define NO_EXISTE      0xFF

#define VIA(o, x, y)      {ICON_VIA_V+o, x, y, NO_ASIGNADO}
#define CURVA(o, x, y)     {ICON_CURVA_NE+o, x, y, NO_ASIGNADO}
#define TOPE(o, x, y)       {ICON_TOPE_N+o, x, y, NO_ASIGNADO}
#define SEMAFORO(o, x, y, d) {ICON_SEMAFORO_N+o, x, y, d-1}
#define PAN(o, x, y, d)     {ICON_PAN_S+o, x, y, d-1}
#define DESVIO_V(o, x, y, d) {ICON_DESVIO_V_NE+o, x, y, d-1}
#define DESVIO_H(o, x, y, d) {ICON_DESVIO_H_NE+o, x, y, d-1}
#define CONECTOR(x, y, d)   {ICON_CONECTOR, x, y, NO_ASIGNADO+d}
#define RUTA(x, y, d)        {ICON_RUTA, x, y, NO_ASIGNADO+d}
#define TEXTO(d)             {ICON_TEXTO, NO_EXISTE, NO_EXISTE, NO_ASIGNADO+d}
```

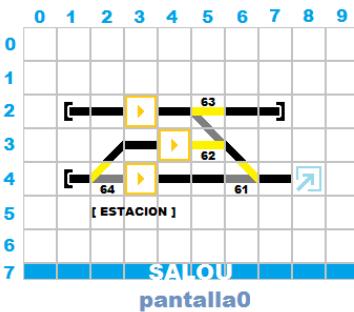
Hemos utilizado el parámetro **o** para indicar la orientación, sumándolo al ícono base de ese tipo. El parámetro **d** lo guardamos como **d-1** ya que la librería Xpressnet empieza a contar desde 0 y nosotros desde 1, así no nos liamos entre la dirección humana y la de Xpressnet. Para el **CONECTOR**, **RUTA** y **TEXTO** en lugar de una dirección DCC, el byte bajo es simplemente el numero de pantalla o el índice de un array que contendrá los datos de rutas o textos.

Definiremos nuestras pantallas usando estos **#define** con lo que quedarán de este modo:

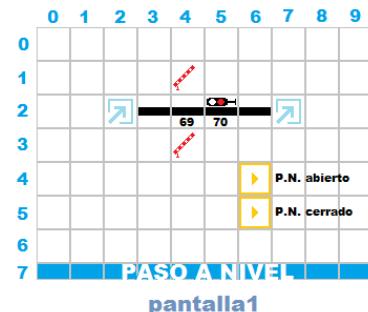
```
const Simbolo pantalla0[] = {
    TOPE(OESTE, 1, 2), VIA(HORIZONTAL, 2, 2), RUTA(3, 2, 2), VIA(HORIZONTAL, 4, 2), DESVIO_H(SUDOESTE, 5, 2, 63),
    VIA(HORIZONTAL, 6, 2), TOPE(ESTE, 7, 2),
    CURVA(SUDESTE, 2, 3), VIA(HORIZONTAL, 3, 3), RUTA(4, 3, 1), DESVIO_H(NORDESTE, 5, 3, 62),
    CURVA(SUDOESTE, 6, 3),
    TOPE(OESTE, 1, 4), DESVIO_H(NOROESTE, 2, 4, 64), RUTA(3, 4, 0), VIA(HORIZONTAL, 4, 4),
    VIA(HORIZONTAL, 5, 4), DESVIO_H(NORDESTE, 6, 4, 61), VIA(HORIZONTAL, 7, 4), CONECTOR(8, 4, 1),
    TEXTO(0), TEXTO(1),
};

const Simbolo pantalla1[] = {
    PAN(VERTICAL, 4, 1, 69), PAN(VERTICAL, 4, 3, 69),
    CONECTOR(2, 2, 0), VIA(HORIZONTAL, 3, 2), VIA(HORIZONTAL, 4, 2), SEMAFORO(OESTE, 5, 2, 70),
    VIA(HORIZONTAL, 6, 2), CONECTOR(7, 2, 2),
    RUTA(6, 4, 3), RUTA(6, 5, 4),
    TEXTO(2),
};

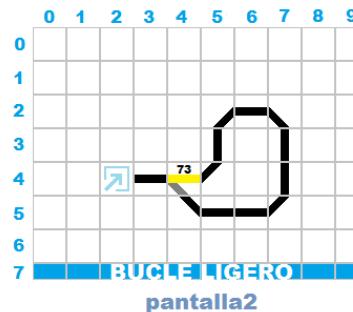
const Simbolo pantalla2[] = {
    CURVA(SUDESTE, 5, 2), VIA(HORIZONTAL, 6, 2), CURVA(SUDOESTE, 7, 2),
    VIA(VERTICAL, 5, 3), VIA(VERTICAL, 7, 3),
    CONECTOR(2, 4, 1), VIA(HORIZONTAL, 3, 4), DESVIO_H(SUDOESTE, 4, 4, 73), CURVA(NOROESTE, 5, 4),
    VIA(VERTICAL, 7, 4),
    CURVA(NORDESTE, 4, 5), VIA(HORIZONTAL, 5, 5), VIA(HORIZONTAL, 6, 5), CURVA(NOROESTE, 7, 5),
    TEXTO(3),
};
```



PANTALLA0



PANTALLA1



[link](#)

Ahora que las tenemos definidas podemos crear un array para poder direccionarlas fácilmente y otro array para saber cuántos íconos hemos colocado en cada pantalla:

```
#define NUM_PANTALLAS 3

const Simbolo *Pantalla[NUM_PANTALLAS] = {pantalla0, pantalla1, pantalla2};
const int numIconosPantalla[NUM_PANTALLAS] = {LONG_PANTALLA(pantalla0), LONG_PANTALLA(pantalla1),
                                             LONG_PANTALLA(pantalla2)};
```

Los **#define** también pueden servir para decirle al compilador que código tiene que compilar en función de si está definida una etiqueta, aquí lo podemos usar para elegir las librerías que tiene que usar en función de si definimos una etiqueta determinada o no:

```
// Descomentar la siguiente linea sino funciona con la libreria Adafruit_TFTLCD.h o se desconoce el
chip
#ifndef TIPO_DESCONOCIDO

#endif TIPO_DESCONOCIDO
#include <MCUFRIEND_kbv.h> // Libreria especifica para el Hardware v2.9.9
MCUFRIEND_kbv tft;
#else
#include <Adafruit_TFTLCD.h> // Libreria especifica para el Hardware v1.0.3
Adafruit_TFTLCD tft(LCD_CS, LCD_CD, LCD_WR, LCD_RD, LCD_RESET#endif
#endif
```

Los Textos

Voy a tener dos tipos de textos, el título de la pantalla que imprimiremos en la última media fila y los textos del circuito que trataremos como un tipo de ícono.

El título de la pantalla actual voy a imprimirlo sobre una barra de color para que resalte, tendrá un color diferente según la central este en modo de operaciones o no (Stop, programación,...) para informar al usuario.

```
#define COLOR_FONDO_TITULO AQUA
#define COLOR_FONDO_TIT_STOP PINK
#define COLOR_TITULO WHITE
```

Los nombres de las pantallas serán arrays de caracteres, con estos arrays haré otro array de forma que usando como índice el numero de pantalla actual pueda imprimir uno u otro.

```
byte pantallaActual;

const char nombrePantalla0[] = {"SALOU"};
const char nombrePantalla1[] = {"PASO A NIVEL"};
const char nombrePantalla2[] = {"BUCLE LIGERO"};

const char *nombrePantallas[NUM_PANTALLAS] = {nombrePantalla0, nombrePantalla1, nombrePantalla2};
```

Para posicionar el rectángulo de color usaré las dimensiones de anchura y altura de la pantalla que me proporciona la librería con las funciones **width()** y **height()**, así a la altura le restaré media fila (16 pixels) para tener el origen del rectángulo que dibujaré con **fillRect()** con el color que dependerá del estado de la central que tendrá almacenado en **csStatus**

Para centrar el título tendrá que restar del ancho de la pantalla el ancho del texto y dividirlo entre 2 para tener el origen de donde imprimirlo. El ancho del texto depende del tamaño con que se escriba, voy a usar un tamaño 2 en el que cada letra tiene un ancho de 12 pixels, para saber cuántas letras tiene el título uso la función **strlen()**

```
byte csStatus; // Estado de la central

void pintaTitulo() {
    int n, x, y;
    y = tft.height() - 16;
    tft.fillRect(0,y,tft.width(),16,(csStatus==csNormal) ? COLOR_FONDO_TITULO : COLOR_FONDO_TIT_STOP);
    n = strlen(nombrePantallas[pantallaActual]);
    x = (tft.width() - (n * 12)) / 2;
    tft.setCursor(x, y);
    tft.setTextSize(2);
    tft.setTextColor(COLOR_TITULO);
    tft.print(nombrePantallas[pantallaActual]);
}
```

La librería Xpressnet nos informa del cambio de estado de la central con la función **callback notifyXNetPower()** así que tomaremos nota del cambio y pintaremos el título para que muestre el color adecuado.

```
void notifyXNetPower (uint8_t State) { // La libreria llama a esta funcion cuando cambia
    estado de la central
    csStatus = State;
    pintaTitulo(); // guardamos el estado
    // Cambiamos el color de fondo del titulo segun el estado
}
```

Para los textos de la pantalla que tratamos como iconos podemos hacer algo parecido, un array de los arrays de textos y según un índice imprimir uno u otro.

Uno de los textos posibles es imprimir el número de desvío justo debajo de él en su misma cuadricula, pero si por la disposición del desvío es mejor imprimirlo encima del mismo o al lado damos lugar a muchas posibles variaciones.

Voy a escoger un método más flexible en el que el mismo texto se especifique las coordenadas en pixels usando un carácter especial que no se utilice normalmente así podremos incluso escribir en varios lugares de la pantalla diferentes textos.

El carácter @ indicará la posición horizontal, el carácter , la posición vertical y el carácter ; servirá de separador de coordenadas y texto numérico.

```
const char texto0[] = {"@64,172[ ESTACION ]"};
const char texto1[] = {"@74,150;64@202;61@170,118;62,66;63"}; // Y = +2,+12,+22 X= +10
const char texto2[] = {"@138,86;69@170;70@230,140P.N. abierto@230,172P.N. cerrado"};
const char texto3[] = {"@138,130;73"};
```

```
const char *textos[] = {texto0, texto1, texto2, texto3};
```

La función `pintaTexto()` nos imprimirá carácter a carácter el texto del array a tamaño 1 y con el color que hayamos definido en `COLOR_TEXTO`. El compilador hace que las cadenas de caracteres acaben con el valor 0, por lo que el `while()` finalizará tras alcanzar el último carácter de la cadena.

Cuando encontremos el carácter especial @ o el , debemos extraer el valor de la coordenada, incluso podemos limitarla a un máximo por si nos hemos equivocado al escribir el texto. El carácter ; no hará nada ya que es un simple separador de dígitos.

La función `extraeNumero()` seguirá recorriendo el texto y mientras el carácter sea un número irá añadiéndolo a la variable `valor`. La función `isDigit()` nos dirá si el carácter es un dígito, en ese caso el valor anterior lo multiplicamos por 10 y le añadimos el valor del dígito (restando al carácter leído el carácter 0).

```
#define COLOR_TEXTO BLACK

int posTexto;

int extraeNumero (char *texto, int maximo) {
    int valor;
    valor = 0;
    while (isDigit (texto[posTexto])) // mientras sea un numero
        valor = (valor * 10) + (texto[posTexto] - '0'); // lo añadimos al valor
    valor = min (valor, maximo); // no ha de ser superior al maximo
    return (valor);
}

void pintaTexto (int numTexto) {
    int x, y;
    char caracter;
    x = 0;
    y = 0;
    posTexto = 0;
    tft.setTextColor(COLOR_TEXTO);
    tft.setTextSize(1);
    while (caracter = textos[numTexto][posTexto++]) {
        switch (caracter) { // caracteres poco usados: ! # $ % & ; , @ ^ _ | ~
            case '@': // coordenada X
                x = extraeNumero(textos[numTexto], tft.width() - 1);
                tft.setCursor(x, y);
                break;
            case ',': // coordenada Y
                y = extraeNumero(textos[numTexto], tft.height() - 1);
                tft.setCursor(x, y);
                break;
            case ';': // separador
                break;
            default: // texto
                tft.print(caracter);
                break;
        }
    }
}
```

Las posiciones

Para pintar una pantalla no necesitamos más que borrarla, imprimir el título y los iconos. En realidad no es tan sencillo, debemos pintar los iconos que pueden tener varios aspectos en la posición correcta.

En Xpressnet un accesorio puede tener cuatro posiciones: No movido, desviado/rojo, recto/verde e inválido. La función *callback* de la librería Xpressnet [notifyTrnt\(\)](#) que ya hemos utilizado otras veces nos informa de la posición de un accesorio cuando la cambiamos nosotros o nos informa la central de un cambio.

También podemos pedir información de la posición actual de un accesorio con la función [getTrntInfo\(\)](#) por lo que al inicio de nuestro programa pediremos a la central que nos informe de todos los accesorios que tienen una dirección válida de nuestro circuito. Para no colapsar el bus y dar tiempo a pintarlos lo haremos de uno en uno cada cierto tiempo.

La función [refrescaInformacion\(\)](#) inicializará los contadores y activará el *flag* [pideInformacion](#) indicando que se están pidiendo datos. En el [loop\(\)](#) si esta activado ese *flag* se llamará a la función [buscaInformacion\(\)](#) que irá pidiendo datos de accesorios cuando la dirección de un ícono sea válida (1 a 1024) recorriendo todos los íconos de cada una de las pantallas definidas.

```
const int tiempoPosicionInfo = 300; // Tiempo entre petición información posición

unsigned long tiempoInfo; // tiempo para pedir información
byte pantallaInfo, iconInfo; // contadores para pedir información
bool pideInformacion; // pidiendo información íconos

void refrescaInformacion() {
    tiempoInfo = millis(); // inicializamos variables para pedir datos
    pantallaInfo = 0;
    iconInfo = 0;
    pideInformacion = true; // hay información pendiente de recibir
}

void buscaInformacion() {
    unsigned int dirección;

    if (millis() - tiempoInfo > tiempoPosicionInfo) { // espera un tiempo entre peticiones
        dirección = Pantalla[pantallaInfo][iconInfo].dirección & 0xFF;
        if (dirección < 1024) { // solo accesorios
            tiempoInfo = millis(); // resetea tiempo espera
            XpressNet.getTrntInfo (highByte(dirección), lowByte(dirección)); // pide información a la central
        }
        iconInfo++; // siguiente ícono de la pantalla
        if (iconInfo == numIconosPantalla[pantallaInfo]) { // información pantalla completa
            iconInfo = 0; // íconos próxima pantalla
            pantallaInfo++;
            if (pantallaInfo == NUM_PANTALLAS) { // todas las pantallas
                pantallaInfo = 0;
                pideInformacion = false; // ya hemos recibido toda la información
            }
        }
    }
}
```

Cuando la central nos informe de la posición se ejecutará la función [notifyTrnt\(\)](#), en ella tomaremos nota de la posición para cuando tengamos que repintar la pantalla y podríamos pintar el ícono o los íconos con esa dirección buscándolos en la pantalla actual (y también en el resto de pantallas), pero como esto es muy lento, especialmente el pintar el ícono, y se podría perder algún paquete Xpressnet si nos informa de varios cambios a la vez vamos a meterlos en una cola para pintarlos luego.

La cola como vimos en un capítulo anterior será FIFO (First In First Out) y es fácil de implementar con máscaras si usamos longitudes que sean potencias de 2, en este caso 32.

```
unsigned int cambiosFIFO[32]; // cola FIFO para cambios en pantalla
byte cambioIn; // índice cabeza cola
byte cambioOut; // índice final cola
byte enColaCambios; // número de accesorios en cola

void borraCambiosFIFO () {
    cambioIn = 0; // índice cabeza cola
    cambioOut = 0; // índice final cola
    enColaCambios = 0; // número de accesorios en cola
}
```

```

unsigned int readCambiosFIFO () {
    cambioOut = (cambioOut + 1) & 0x1F; //avanza puntero circular
    enColaCambios--;
    return (cambiosFIFO[cambioOut]);
}

void writeCambiosFIFO (unsigned int direccion) {
    cambioIn = (cambioIn + 1) & 0x1F; // avanza puntero circular
    enColaCambios++;
    cambiosFIFO[cambioIn] = direccion; // lo guarda en la cola
}

```

La función `notifyTrnt()` guardará la posición informada en `Pos` en un array para que después se pueda consultar y meterá en la cola la dirección informada si la posición es diferente de la que teníamos guardada, así nos ahorraremos volver a pintar un ícono si ya está en la posición informada.

Para tomar nota de la posición y no ocupar mucha memoria ya que `Pos` son sólo dos bits para las cuatro posiciones, vamos a empaquetar 4 direcciones en un byte (8 bits) así los 1024 accesorios solo ocuparán 128 bytes. Esto se corresponde con el formato de los módulos de retroseñalización RS de Lenz para Xpressnet.

Módulo							
Nibble alto				Nible bajo			
Z1	Z0	Z1	Z0	Z1	Z0	Z1	Z0
Dirección n+3		Dirección n+2		Dirección n+1		Dirección n	

```

enum posDesvio {NO_MOVIDO, DESVIADO, RECTO, INVALIDO}; // DESVIADO: ROJO - > links      RECTO:
VERDE + < rechts

byte RS[128]; // Datos de los RS

for (int i = 0; i < 128; i++) // borra estado contactos
    RS[i] = 0;

```

Usando máscaras (`B11` para dos bits) y desplazamientos podemos comparar y colocar los dos bits de `Pos` en el lugar correcto dentro del byte de un módulo. Si al hacer la operación XOR el resultado no es cero es que hay un cambio en alguno de los dos bits que estamos comprobando y tenemos que borrar el resultado antiguo y poner el actual además de meter la dirección en la cola.

```

void notifyTrnt(uint8_t Adr_High, uint8_t Adr_Low, uint8_t Pos) { // Llamado por la libreria
cuando hay cambios en accesorios
    byte modulo, mascara, desplazar, nuevaPos;
    unsigned int direccion;

    direccion = (Adr_High << 8) + Adr_Low; // calculamos modulo RS en el que esta la direccion
    modulo = direccion >> 2; // la mascara ver sus bits de estado
    desplazar = (direccion & 0x03) * 2; // los datos de la nueva posicion
    mascara = B11 << desplazar; // si cambia la posicion respecto a la guardada
    nuevaPos = (Pos & B11) << desplazar; // borramos estado actual
    if ((RS[modulo] ^ nuevaPos) & mascara) { // ponemos nuevo estado
        RS[modulo] &= ~mascara;
        RS[modulo] |= nuevaPos;
        writeCambiosFIFO (direccion); // guardamos direccion en la FIFO para actualizar pantalla
    }
}

```

Ahora si queremos saber la posición actual de un ícono sólo tenemos que consultar la tabla que estará siempre actualizada:

```

byte buscaPosicionActual(unsigned int direccion) { // busca posicion actual de un accesorio
    byte modulo, desplazar, posicion;
    modulo = direccion >> 2; // calculamos modulo RS en el que esta la direccion
    desplazar = (direccion & B11) * 2; // y la posicion de sus bits de estado
    posicion = (RS[modulo] >> desplazar) & B11; // leemos posicion actual
    return posicion;
}

```

Pintando iconos

Después de todo este trabajo previo ya estamos en disposición de poder pintar un ícono en cualquier cuadrícula de la pantalla mostrando su posición correcta con el color adecuado.

Con `#define` al ser una sustitución podemos hacer que en lugar de una constante sea una función, por ejemplo, `color565()` que transforma los parámetros RGB al formato 565 que necesita nuestra pantalla. Lo voy a hacer así con el color de fondo ya que el blanco es demasiado blanco y lo voy a oscurecer un poco:

```
#define COLOR_FONDO      tft.color565(235, 235, 235)
#define COLOR_VIA          BLACK
#define COLOR_PAN           RED
#define COLOR_RUTA          ORANGE
#define COLOR_CONECTOR      DARKCYAN
#define COLOR_DESVIO_NO_POS DARKGREY
#define COLOR_DESVIO_POS    YELLOW
```

Tenemos diferentes tipos de íconos:

- Sin posición con un ícono: VIA, CURVA, TOPE, CONECTOR
- Con posición con un ícono: SEMAFOROS, PAN, RUTA
- Con posición con dos íconos: DESVIOS
- Otros: TEXTOS

La función `pintaIcono()` solo necesita que le pasemos la estructura de datos del ícono y la posición actual para pintarlo correctamente ya que calculará los datos necesarios para la función `drawBitmap()`.

Para los íconos sin posición, simplemente calculamos las coordenadas x e y en pixels y seleccionamos el color adecuado.

Para los íconos con dos posiciones y dos íconos, los desvíos, además buscamos qué ícono corresponde al tramo recto y cuál al tramo desviado y según la posición pintamos primero uno u otro en su color correspondiente. Ya que los hemos puesto al final en la enumeración y siguiendo un orden con la orientación, es fácil averiguar si el tramo recto es horizontal o vertical y que orientación tiene la curva con una simple operación matemática:

```
if (icono < ICON_DESVIO_H_NE) {
    iconoR = ICON_VIA_V;                                // Tramo recto vertical
    iconoD = icono - ICON_DESVIO_V_NE + ICON_CURVA_NE; // Tramo orientación desviado (0..3) + base
}
else {
    iconoR = ICON_VIA_H;                                // Tramo recto horizontal
    iconoD = icono - ICON_DESVIO_H_NE + ICON_CURVA_NE; // Tramo orientación desviado (0..3) + base
}
```

Para los textos sólo necesitamos llamar a `pintaTexto()` ya que no tiene ícono que pintar.

El ícono de la ruta sólo hace un cambio de color brevemente cuando se pulsa así que según se indique en posición lo pintaremos de un color u otro.

El paso a nivel mostrará las barreras subidas o bajadas según la posición, así que borraremos primero el ícono anterior pintando un rectángulo de color de fondo y elegiremos el ícono correspondiente según la posición. Para hacerlo más rápido, el rectángulo será algo menor de 32x32 pixels ya que borraremos sólo lo que se sobreimprime.

El color dependerá de la posición, si a `posicion` le restamos 1, para `RECTO` o `DESVIADO` el bit 1 será 0 por lo que lo pintaremos de su color, pero si es `NO_MOVIDO` o `INVALIDO` el bit 1 será 1 y lo pintaremos del color de no estar en posición.

Para los semáforos procederemos de forma parecida al paso a nivel, calcularemos el desplazamiento (`offset`) de cada una de las luces para pintar un rectángulo, del tamaño de la luz, de color de fondo o de la luz según la posición actual misma y luego pintaremos el ícono del semáforo. No pintamos un círculo ya que el rectángulo se dibuja más rápido y al final queda tapado por el dibujo del semáforo.

Como dibujar es lento, añadiremos llamadas a `XpressNet.receive()` para no perder posibles datos del bus.

```
void pintaIcono (Simbolo simbolo, byte posicion) {
    unsigned int color, colorLuz, x, y, offsetOn, offsetOff;
    byte icono, iconoR, iconoD;
    x = simbolo.posX * 32;
    y = simbolo.posY * 32;
    icono = simbolo.tipo;
```

```

if (icono < ICON_DESVIO_V_NE) {                                // No son desvios
    color = COLOR_VIA;                                         // color por defecto
    switch (posicion) {                                         // calculamos luz en caso semaforos
        case DESVIADO:
            offsetOn = 8;
            offsetOff = 0;
            colorLuz = RED;
            break;
        case RECTO:
            offsetOn = 0;
            offsetOff = 8;
            colorLuz = GREEN;
            break;
        default:
            offsetOn = 0;
            offsetOff = 8;
            colorLuz = COLOR_FONDO;
            break;
    }
}                                                               // casos especiales, RUTA, CONECTOR, SEMAFOROS
switch (icono) {
    case ICON_PAN_S:
        color = bitRead(posicion - 1, 1) ? COLOR_DESVIO_NO_POS : COLOR_PAN;
        tft.fillRect(x + 2, y + 4, 28, 21, COLOR_FONDO); // borra posicion anterior
        if (posicion == DESVIADO)
            icono = ICON_PAN_B;
        break;
    case ICON_PAN_B:
        color = bitRead(posicion - 1, 1) ? COLOR_DESVIO_NO_POS : COLOR_PAN;
        tft.fillRect(x + 2, y + 4, 28, 21, COLOR_FONDO); // borra posicion anterior
        if (posicion == DESVIADO)
            icono = ICON_PAN_S;
        break;
    case ICON_TEXTO:
        pintaTexto(lowByte(simbolo.direccion)); // pintamos texto y salimos
        return;
    break;
    case ICON_RUTA:
        color = (posicion == NO_MOVIDO) ? COLOR_RUTA : COLOR_VIA;
        break;
    case ICON_CONECTOR:
        color = COLOR_CONECTOR;
        break;
    case ICON_SEMAFORO_N:
        tft.fillRect(x + 22, y + 4 + offsetOff, 6, 7, COLOR_FONDO); // borra luz
        XpressNet.receive(); // recibimos paquetes por
Xpressnet
        tft.fillRect(x + 22, y + 4 + offsetOn, 6, 7, colorLuz); // pinta Luz
        break;
    case ICON_SEMAFORO_E:
        tft.fillRect(x + 21 - offsetOff, y + 22, 7, 6, COLOR_FONDO); // recibimos paquetes por
        XpressNet.receive();
Xpressnet
        tft.fillRect(x + 21 - offsetOn, y + 22, 7, 6, colorLuz);
        break;
    case ICON_SEMAFORO_S:
        tft.fillRect(x + 4, y + 21 - offsetOff, 6, 7, COLOR_FONDO); // recibimos paquetes por
        XpressNet.receive();
Xpressnet
        tft.fillRect(x + 4, y + 21 - offsetOn, 6, 7, colorLuz);
        break;
    case ICON_SEMAFORO_O:
        tft.fillRect(x + 4 + offsetOff, y + 4, 7, 6, COLOR_FONDO); // recibimos paquetes por
        XpressNet.receive();
Xpressnet
        tft.fillRect(x + 4 + offsetOn, y + 4, 7, 6, colorLuz);
        break;
    }
    XpressNet.receive(); // recibimos paquetes por Xpressnet
    tft.drawBitmap(x, y, iconos[icono], 32, 32, color);
}
else { // Son desvios
    if (icono < ICON_DESVIO_H_NE) {
        iconoR = ICON_VIA_V; // Tramo recto vertical
        iconoD = icono - ICON_DESVIO_V_NE + ICON_CURVA_NE; // Tramo orientacion desviado (0..3) + base
    }
    else {
        iconoR = ICON_VIA_H; // Tramo recto horizontal
        iconoD = icono - ICON_DESVIO_H_NE + ICON_CURVA_NE; // Tramo orientacion desviado (0..3) + base
    }
}

```

Pintar la pantalla además de poner el título ahora no es más que buscar la posición actual de cada ícono si es uno de los que cambia y pintar el ícono correspondiente.

Puede ocurrir que la posición dibujada este invertida con la realidad debido a que hemos conectado los cables del desvío o semáforo a la inversa. Para solucionar esto y que se pinte correctamente vamos a usar un bit libre en la dirección del ícono para indicar esta circunstancia.

La dirección imposible era `0x0F00` y la máscara que hemos usado para comprobar si era una dirección válida (menor de 1024) en `buscalInformacion()` era `0x0FFF` lo que nos deja los cuatro bits altos libres. Usaremos el bit 14 para indicar que se tiene que pintar invertido, haremos un `#define` para poder usarlo en la definición del circuito:

```
#define INVERTIDO(x)    (x | bit(14))
```

Ahora si queremos invertir un icono, por ejemplo el desvío con dirección 2, escribiremos:

DESVIO H(NORDESTE, 5, 3, INVERTIDO(2)),

Antes de llamar a `pintarIcono()` comprobaremos si el bit 14 está activo e invertiremos el estado con una operación XOR de los dos bits.

```

void pintaPantalla() {
    unsigned int direccion, n;
    byte posicion;
    tft.fillScreen(COLOR_FONDO);
    pintaTitulo();
    XpressNet.receive(); // recibimos paquetes por Xpressnet
    for (n = 0; n < numIconosPantalla[pantallaActual]; n++) {
        direccion = Pantalla[pantallaActual][n].direccion;
        if (Pantalla[pantallaActual][n].tipo < ICON_SEMAFORO_N)
            posicion = NO_MOVIDO;
        else
            posicion = buscaPosicionActual(direccion);
        if (bitRead(direccion, 14)) // comprobamos si hay que invertirlo
            posicion ^= B11;
        pintaIcono (Pantalla[pantallaActual][n], posicion);
        XpressNet.receive(); // recibimos paquetes por Xpressnet
    }
}

```

Los iconos que cambian y que habíamos dejado en la cola para imprimirlos luego se dibujaran de igual forma. Buscaremos en la pantalla actual todos los iconos con la dirección que hemos sacado de la cola, buscando la posición actual e invirtiéndola en caso necesario.

```
void actualizaCambiosIconos() {
    unsigned int direccion;
    byte posicion, n;
    if (enColaCambios > 0) {
        direccion = readCambiosFIFO();
        posicion = buscaPosicionActual(direccion);
        for (n = 0; n < numIconosPantalla[pantallaActual]; n++){ // Puede haber varios con la misma
```

```

direccion
    if (direccion == (Pantalla[pantallaActual][n].direccion & 0x0FFF)) {      // 0..1023
        if (bitRead(Pantalla[pantallaActual][n].direccion, 14))
            pintaIcono (Pantalla[pantallaActual][n], posicion ^ B11);
        else
            pintaIcono (Pantalla[pantallaActual][n], posicion);
        XpressNet.receive();                                // recibimos paquetes por Xpressnet
    }
}
}
}

```

Moviendo accesorios

Con lo que tenemos hasta ahora podemos hacer un panel con una pantalla para visualizar nuestro circuito:

```

const int pinTXRX = A5;                      // pin del MAX485
const int miDireccionXpressnet = 23;           // direccion Xpressnet

void setup() {
    ID = tft.readID();
    tft.begin(ID);                            // Iniciamos el LCD
    tft.setRotation(1);                      // Establecemos la pantalla Horizontal
    for (int i = 0; i < 128; i++)           // borra estado contactos
        RS[i] = 0;
    csStatus = csTrackVoltageOff;             // Suponemos que la central esta sin tension en via
    pantallaActual = 0;                     // pinta pantalla inicial
    pintaPantalla();
    borraCambiosFIFO();                    // borraCambiosFIFO()
    XpressNet.start(miDireccionXpressnet, pinTXRX); // Inicializamos bus Xpressnet
    XpressNet.getPower();                  // Pedimos el estado actual de la central
    refrescaInformacion();                // Pedimos informacion de la posicion de los iconos
}

void loop() {
    XpressNet.receive();                   // recibimos paquetes por Xpressnet
    actualizaCambiosIconos();            // actualiza cambio en iconos de la pantalla actual
    XpressNet.receive();                 // recibimos paquetes por Xpressnet
    if (pideInformacion)                // comprueba si hay informacion pendiente
        buscaInformacion();            // pide informacion posicion iconos
}

```

Pero lo realmente interesante y por eso se llama TCO (Tablero de Control Óptico) es que podamos mover los accesorios desde la pantalla.

Para mover un accesorio en Xpressnet hay que enviar la orden de activarlo y después de un tiempo desactivarlo. Si tenemos que mover varios como en el caso de una ruta para simplificar el código es buena idea implementar una cola FIFO para almacenar los accesorios mientras movemos un accesorio, esperamos el tiempo para desactivarlo e incluso luego tener un tiempo de espera para dar tiempo a la CDU a recargarse antes de enviar el siguiente accesorio.

```

unsigned int accesoriosFIFO[32];               // cola FIFO para mover accesorios
byte accesorioIn;                             // indice cabeza cola
byte accesorioOut;                           // indice final cola
byte enColaAccesorios;                      // numero de accesorios en cola
bool recargando, esperaAccesorio;

void borraAccesoriosFIFO () {
    accesorioIn = 0;                          // indice cabeza cola
    accesorioOut = 0;                         // indice final cola
    enColaAccesorios = 0;                     // numero de accesorios en cola
    recargando = false;                      // CDU cargada
    esperaAccesorio = false;                  // ningun accesorio activado enviado
}

unsigned int readAccesoriosFIFO () {
    accesorioOut = (accesorioOut + 1) & 0x1F; //avanza puntero circular
    enColaAccesorios--;
    return (accesoriosFIFO[accesorioOut]);
}

void writeAccesoriosFIFO (unsigned int accesorio) {
    accesorioIn = (accesorioIn + 1) & 0x1F; // avanza puntero circular
    enColaAccesorios++;
    accesoriosFIFO[accesorioIn] = accesorio; // lo guarda en la cola
}

```

En la cola mediante `writeAccesoriosFIFO()` meteremos el número de accesorio y la posición a la que lo queremos mover. Ya que el número máximo es 1024, usaremos el bit 15 de la dirección para indicar la posición, Recto desactivado y Desviado activado.

La función `enviaColaAccesorios()` la iremos llamando continuamente. Si hay un accesorio en la cola, enviamos la activación e inicializamos el tiempo, con ayuda de un *flag* indicaremos que esperamos a que pase el tiempo de activación. Al pasar ese tiempo enviamos la desactivación y con otro *flag* indicaremos que estamos en tiempo de espera de recarga de la CDU. Al pasar este último tiempo volveremos al estado inicial para enviar el siguiente accesorio.

```
const int tiempoAccesorioON    = 150;           // Tiempo de activacion accesorios
const int tiempoCDU          = 250;           // Tiempo espera recarga CDU

unsigned long tiempoAccesorio;                 // envio de accesorios
unsigned int accEnviado;

void enviaColaAccesorios() {
    byte adrH, adrL, pos;
    if (recargando) {                                // esperando a que se recargue la CDU.
        if (millis() - tiempoAccesorio > (tiempoCDU)) {
            recargando = false;
        }
    }
    else {
        if (esperaAccesorio) {
            if (millis() - tiempoAccesorio > (tiempoAccesorioON)) {
                adrH = highByte (accEnviado) & 0x03;
                adrL = lowByte (accEnviado);
                pos = bitRead(accEnviado, 15) ? B0000 : B0001; // bit 15 activo indica posicion Desviado
                XpressNet.setTrntPos (adrH, adrL, pos);      // A00P Envia desconectar accesorio
                tiempoAccesorio = millis();
                esperaAccesorio = false;
                recargando = true;
            }
        }
        else {
            if (enColaAccesorios > 0) {
                accEnviado = readAccesoriosFIFO();
                adrH = highByte (accEnviado) & 0x03;
                adrL = lowByte (accEnviado);
                pos = bitRead(accEnviado, 15) ? B1000 : B1001; // bit 15 activo indica posicion Desviado
                XpressNet.setTrntPos (adrH, adrL, pos);      // A00P
                tiempoAccesorio = millis();
                esperaAccesorio = true;
            }
        }
    }
}
```

Para las rutas solo hay que meter en la cola los accesorios que la componen con su posición. Para que sea fácil definirlas voy a hacer de forma similar a los textos, una cadena con las direcciones y un carácter **R** para indicar la posición recto o un carácter **D** para indicar posición desviado:

```
const char ruta0[] = {"64R61R"};           // RECTO: VERDE + <
const char ruta1[] = {"64D62D61D"};         // DESVIADO: ROJO - >
const char ruta2[] = {"61D62R63D"};
const char ruta3[] = {"70D69R"};
const char ruta4[] = {"70R69D"};

const char *rutas[] = {ruta0, ruta1, ruta2, ruta3, ruta4};
```

Cuando se tenga que enviar una ruta leeremos carácter a carácter la cadena, si no es una **D** o una **R** se tomará el valor numérico del carácter como hicimos con `extraeNumero()` y lo añadiremos a `direccion`.

Si es una **R**, restaremos 1 a `direccion` para pasar de dirección humana a Xpressnet y meteremos la dirección en la cola, reseñando luego la dirección para el próximo accesorio. Si es una **D** activaremos el bit 15 de `direccion` y procedemos igual que con **R**.

Finalmente tomaremos nota del tiempo para luego evitar repetir rutas si mantenemos pulsado el icono.

```

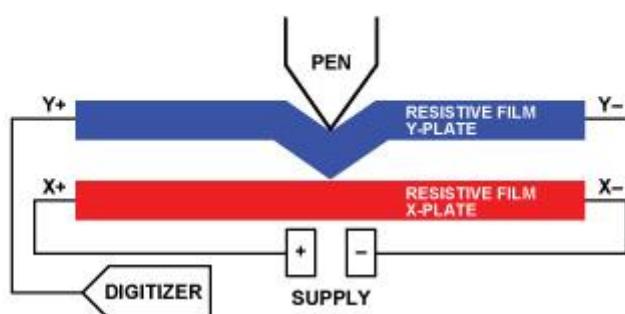
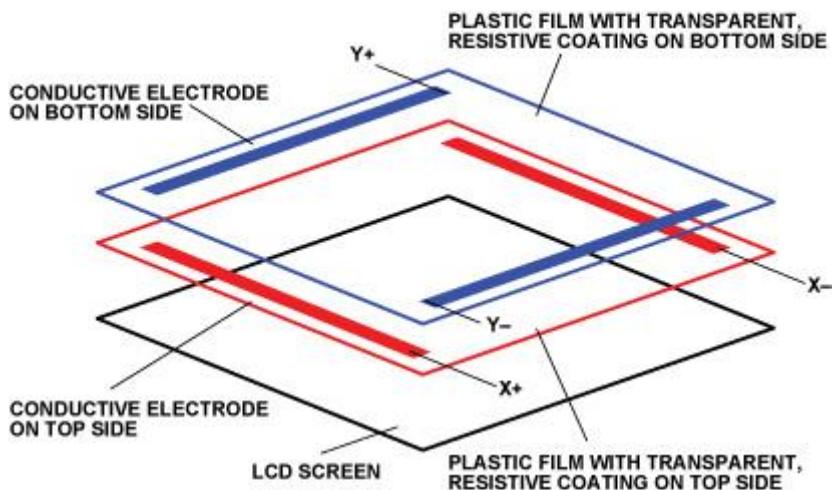
unsigned long tiempoRuta = millis(); // envio de rutas

void enviaRuta(unsigned int numRuta) {
    unsigned int pos, direccion;
    char caracter;
    pos = 0;
    direccion = 0;
    while (caracter = rutas[numRuta][pos]) {
        switch (caracter) {
            case 'D':
                bitSet(direccion, 15); // desviado bit 15 activo
            case 'R':
                direccion--;
                writeAccesoriosFIFO(direccion);
                direccion = 0;
                pos++;
                break;
            default:
                direccion = (direccion * 10) + (rutas[numRuta][pos] - '0'); // añade valor a direccion
                break;
        }
    }
    tiempoRuta = millis(); // timeout para rutas
}

```

El panel táctil

Una pantalla táctil resistiva consiste principalmente en dos capas separadas de material plástico conductor con una determinada resistencia a la corriente eléctrica, que al pulsar en un punto determinado de la capa exterior, ésta hace contacto con la capa interior y midiendo la resistencia (en realidad la tensión en el divisor resistivo que se forma) calcula el punto exacto donde se ha pulsado en un eje de coordenadas.



[link](#)

La librería **TouchScreen.h** se encarga poner las señales adecuadas y de proporcionarnos los valores leídos en los pines analógicos en una estructura con los datos de los ejes X, Y y Z (presión).

ATENCIÓN: Es MUY importante calibrar la pantalla táctil que tenemos conectada de lo contrario no funcionará o no producirá el resultado esperado. Para ello la librería **MCUFRIEND_kbV.h** tiene un programa de ejemplo, **TouchScreen_Calibr_native** (Archivo -> Ejemplos -> **MCUFRIEND_kbV** -> **TouchScreen_Calibr_native**), que nos proporciona los datos necesarios para configurar adecuadamente nuestra pantalla táctil en el Monitor Serie configurado a 9600b.

Con el lápiz de la pantalla táctil pulsaremos lo más cerca del centro de cada una de las 8 cruces blancas que nos muestra, manteniéndolo pulsado hasta que la cruz se vuelva roja. En la pantalla y en el Monitor Serie nos mostrará los datos que necesitamos de nuestra pantalla y del panel táctil.

Hay que copiar estas líneas desde el Monitor Serie a nuestro programa (los valores pueden ser diferentes con vuestra pantalla):

```
const int XP = 8, XM = A2, YP = A3, YM = 9; //240x320 ID=0x9341
const int TS_LEFT=239, TS_RT=902, TS_TOP=177, TS_BOT=909;
```

Una vez tenemos los datos de los pines analógicos y digitales que usa así como los límites leídos para cada uno de los extremos inicializamos la librería:

```
TouchScreen ts = TouchScreen(XP, YP, XM, YM, 300);
```

NOTA: El valor 300 es un valor genérico de la resistencia de nuestro panel. Podemos afinarlo un poco más poniendo el valor de la medida de resistencia entre los pines XP e XM de nuestro panel aunque con ese valor genérico funciona razonablemente bien.

La función de la librería **getPoint()** nos dará los datos leídos por los pines analógicos en una estructura **TSPoint**. Los pines quedan como analógicos pero son los mismos que usa nuestra pantalla por lo que tendremos que volver a ponerlos como digitales para que funcione nuestra pantalla.

Los datos tenemos que convertirlos a pixel según los límites de nuestro panel táctil y el tamaño de nuestra pantalla de ahí la importancia de calibrar bien nuestro panel. La función **map()** nos dará los valores adecuados para los ejes X e Y.

El eje Z nos sirve para ver la presión así que le definiremos unos límites de presión mínima y máxima para aceptar los valores.

```
#define MINPRESSURE 10
#define MAXPRESSURE 1000

int X; // Variables que almacenaran la coordenada
int Y; // X, Y donde presionemos y la variable Z
int Z; // almacenara la presion realizada

TSPoint p = ts.getPoint(); // Realizamos lectura de las coordenadas panel tactil
pinMode(XM, OUTPUT); // La librería utiliza estos pines como entrada y salida
pinMode(YP, OUTPUT); // por lo que es necesario declararlos como salida justo despues de
realizar una lectura de coordenadas.

X = map(p.y, TS_TOP, TS_BOT, 0, tft.width()); // Mapeamos los valores analogicos leidos del panel
tactil (0-1023)
Y = map(p.x, TS_RT, TS_LEFT, 0, tft.height());
Z = p.z;
```

A nosotros más que la coordenada nos interesa saber la cuadrícula de 32x32 pixel que se ha pulsado (columna y fila) así que convertimos los valores de coordenadas a cuadrículas. Si la calibración no es adecuada puede dar lugar a valores fuera de rango por lo que con **constrain()** los restringiremos al ancho y alto de la pantalla.

```
int col, fila;

col = constrain (X, 0, tft.width() - 1) / 32;
fila = constrain (Y, 0, tft.height() - 1) / 32;
```

Sabiendo la columna y la fila solo tenemos que buscar en la pantalla actual el icono que ocupa esa posición:

```
byte buscaIconoPulsado (int col, int fila) {      // busca icono pulsado en la pantalla actual
    byte x, y, n;
    for (n = 0; n < numIconosPantalla[pantallaActual]; n++) {
        x = Pantalla[pantallaActual][n].posX;
        y = Pantalla[pantallaActual][n].posY;
        if ((x == col) && (y == fila))
            return n;
    }
    return NO_EXISTE;                                // no encontrado
}
```

En el **loop()** no tenemos más que añadir que se envíe la cola de accesorios y leer el panel táctil. Si el eje Z tiene un valor válido es que se ha pulsado la pantalla. Buscaremos si se ha pulsado sobre un ícono y comprobaremos de qué tipo es:

- Si es un conector tomamos el valor que tiene y lo asignamos a **pantallaActual** y pintaremos la nueva pantalla.
- Si es una ruta, miramos si ha pasado un tiempo prudencial entre rutas y si así, enviamos la ruta a la cola de accesorios. Haremos un pequeño efecto con el ícono pintándolo de otro color brevemente para que el usuario sepa que se ha detectado la pulsación.
- Si es un accesorio (desvío, semáforo o paso a nivel) buscaremos la posición actual del mismo y meteremos en la cola la posición contraria. Indicaremos que se ha pulsado el ícono y tomaremos nota del tiempo para evitar rebotes.

```
bool pulsacionDetectada = false;                      // pulsacion detectada en panel tactil
unsigned long tiempoRebote;                          // tiempo de espera para evitar rebotes

byte iconoPulsado, tipo, posicion;
unsigned int direccion;

if (pulsacionDetectada) {
    if (millis() - tiempoRebote > 200)
        pulsacionDetectada = false;
}
else {
    if (Z > MINPRESSURE && Z < MAXPRESSURE) {          // si se pulsa la pantalla
        col = constrain (X, 0, tft.width() - 1) / 32;
        fila = constrain (Y, 0, tft.height() - 1) / 32;

        iconoPulsado = buscaIconoPulsado (col, fila);
        if (iconoPulsado != NO_EXISTE) {                  // comprueba si hemos pulsado sobre un ícono
            tipo = Pantalla[pantallaActual][iconoPulsado].tipo;
            if (tipo < ICON_SEMAFORO_N) {                // via, ruta o conector
                if (tipo == ICON_CONECTOR) {              // si es un conector cambiamos de pantalla
                    pantallaActual = lowByte(Pantalla[pantallaActual][iconoPulsado].direccion);
                    pintaPantalla();
                }
                if (tipo == ICON_RUTA) {                  // si es una ruta enviamos la ruta
                    if (millis() - tiempoRuta > ((tiempoAccesorioON + tiempoCDU) * 3)) {  // Esperar un
                        tiempo prudencial entre rutas
                        pintaIcono(Pantalla[pantallaActual][iconoPulsado], RECTO);
                        direccion = lowByte(Pantalla[pantallaActual][iconoPulsado].direccion);
                        enviaRuta(direccion);
                        pintaIcono(Pantalla[pantallaActual][iconoPulsado], NO_MOVIDO);
                    }
                }
            }
            else {                                    // semáforo o desvío
                direccion = Pantalla[pantallaActual][iconoPulsado].direccion;
                posicion = buscaPosicionActual (direccion); // busca posición actual
                if (posicion == RECTO)                   // cambiar a la otra posición
                    bitSet(direccion, 15);             // bit 15 activo indica posición Desviado
                writeAccesoriosFIFO(direccion);
                pulsacionDetectada = true;
                tiempoRebote = millis();
            }
        }
    }
}
```

El programa

Para un fácil mantenimiento he dividido el programa en tres archivos:

- **iconos.h** contiene los datos gráficos de los iconos

- **circuito.h** contiene los #defines y datos para que el usuario/usuaria adapte el TCO a su circuito (iconos, colores, tiempos).

- **TFT_TCO.ino** contiene el programa principal

iconos.h

```
// TFT_TCO - Paco Cañada 2022 -- https://usuaris.tinet.cat/fmco/  
  
// 'viaV', 32x32px  
const unsigned char icon_viaV [] PROGMEM = {  
    0x00, 0x00, 0x00, 0x00, 0x0f, 0xf0, 0x00, 0x00, 0x0f, 0xf0, 0x00, 0x00, 0x0f, 0xf0, 0x00,  
    0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x00, 0x0f, 0xf0, 0x00,  
    0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x00, 0x0f, 0xf0, 0x00,  
    0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x00, 0x0f, 0xf0, 0x00,  
    0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x00, 0x0f, 0xf0, 0x00,  
    0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x00, 0x0f, 0xf0, 0x00,  
    0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x00, 0x0f, 0xf0, 0x00,  
    0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x00, 0x0f, 0xf0, 0x00,  
    0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x00, 0x0f, 0xf0, 0x00,  
};  
// 'viaH', 32x32px  
const unsigned char icon_viaH [] PROGMEM = {  
    0x00,  
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
    0x7f, 0xff, 0xff, 0xfe, 0x7f, 0xff, 0xff, 0xfe, 0x7f, 0xff, 0xff, 0xfe, 0x7f, 0xff, 0xff, 0xfe,  
    0x7f, 0xff, 0xff, 0xfe, 0x7f, 0xff, 0xff, 0xfe, 0x7f, 0xff, 0xff, 0xfe, 0x7f, 0xff, 0xff, 0xfe,  
    0x00,  
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
};  
// 'curvaNE', 32x32px  
const unsigned char icon_curvaNE [] PROGMEM = {  
    0x00, 0x00, 0x00, 0x00, 0x0f, 0xfc, 0x00, 0x00, 0x07, 0xfe, 0x00, 0x00, 0x03, 0xff, 0x00,  
    0x00, 0x01, 0xff, 0x80, 0x00, 0x00, 0xff, 0xc0, 0x00, 0x00, 0x7f, 0xe0, 0x00, 0x00, 0x3f, 0xf0,  
    0x00, 0x00, 0x1f, 0xf8, 0x00, 0x00, 0x0f, 0xfc, 0x00, 0x00, 0x07, 0xfe, 0x00, 0x00, 0x03, 0xfe,  
    0x00, 0x00, 0x01, 0xfe, 0x00, 0x00, 0x0f, 0xe0, 0x00, 0x00, 0x00, 0x7e, 0x00, 0x00, 0x00, 0x3e,  
    0x00, 0x00, 0x00, 0x1e, 0x00, 0x00, 0x00, 0x0e, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x02,  
    0x00,  
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
};  
// 'curvaSE', 32x32px  
const unsigned char icon_curvaSE [] PROGMEM = {  
    0x00,  
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
    0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x0e, 0x00, 0x00, 0x00, 0x00, 0x1e,  
    0x00, 0x00, 0x00, 0x3e, 0x00, 0x00, 0x00, 0x7e, 0x00, 0x00, 0x00, 0x0e, 0x00, 0x00, 0x00, 0x01, 0xfe,  
    0x00, 0x00, 0x03, 0xfe, 0x00, 0x00, 0x07, 0xfe, 0x00, 0x00, 0x0f, 0xfc, 0x00, 0x00, 0x1f, 0xf8,  
    0x00, 0x00, 0x3f, 0xf0, 0x00, 0x00, 0x7f, 0xe0, 0x00, 0x00, 0xff, 0xc0, 0x00, 0x01, 0xff, 0x80,  
    0x00, 0x03, 0xff, 0x00, 0x00, 0x07, 0xfe, 0x00, 0x00, 0x0f, 0xfc, 0x00, 0x00, 0x00, 0x00, 0x00,  
};  
// 'curvaSO', 32x32px  
const unsigned char icon_curvaSO [] PROGMEM = {  
    0x00,  
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
    0x40, 0x00, 0x00, 0x00, 0x60, 0x00, 0x00, 0x00, 0x70, 0x00, 0x00, 0x00, 0x78, 0x00, 0x00, 0x00,  
    0x7c, 0x00, 0x00, 0x00, 0x7e, 0x00, 0x00, 0x00, 0x7f, 0x00, 0x00, 0x00, 0x7f, 0x80, 0x00, 0x00,  
    0x7f, 0xc0, 0x00, 0x00, 0x7f, 0xe0, 0x00, 0x00, 0x3f, 0xf0, 0x00, 0x00, 0x1f, 0xf8, 0x00, 0x00,  
    0x0f, 0xfc, 0x00, 0x00, 0x07, 0xfe, 0x00, 0x00, 0x03, 0xff, 0x00, 0x00, 0x01, 0xff, 0x80, 0x00,  
    0x00, 0xff, 0xc0, 0x00, 0x00, 0x7f, 0xe0, 0x00, 0x00, 0x3f, 0xf0, 0x00, 0x00, 0x00, 0x00, 0x00,  
};  
// 'curvaNO', 32x32px  
const unsigned char icon_curvaNO [] PROGMEM = {  
    0x00, 0x00, 0x00, 0x00, 0x3f, 0xf0, 0x00, 0x00, 0x7f, 0xe0, 0x00, 0x00, 0x0f, 0xff, 0xc0, 0x00,  
    0x01, 0xff, 0x80, 0x00, 0x03, 0xff, 0x00, 0x00, 0x07, 0xfe, 0x00, 0x00, 0x0f, 0xfc, 0x00, 0x00,  
    0x1f, 0xf8, 0x00, 0x00, 0x3f, 0xf0, 0x00, 0x00, 0x7f, 0xe0, 0x00, 0x00, 0x7f, 0xc0, 0x00, 0x00,  
    0x7f, 0x80, 0x00, 0x00, 0x7f, 0xe0, 0x00, 0x00, 0x00, 0x7e, 0x00, 0x00, 0x00, 0x7c, 0x00, 0x00, 0x00,  
    0x78, 0x00, 0x00, 0x00, 0x70, 0x00, 0x00, 0x00, 0x60, 0x00, 0x00, 0x00, 0x40, 0x00, 0x00, 0x00,  
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
};
```



```

// 'semaforoS', 32x32px
const unsigned char icon_semaforoS [] PROGMEM = {
  0x00, 0x00, 0x00, 0x00, 0xf0, 0x00, 0x0f, 0xcf, 0xf0, 0x00, 0x03, 0x0f, 0xf0, 0x00,
  0x03, 0x0f, 0xf0, 0x00, 0x03, 0x0f, 0xf0, 0x00, 0x03, 0x0f, 0xf0, 0x00, 0x03,
  0x0f, 0xf0, 0x00, 0x03, 0x0f, 0xf0, 0x00, 0x0f, 0xcf, 0xf0, 0x00, 0x0f, 0xf0, 0x00,
  0x1f, 0xef, 0xf0, 0x00, 0x1c, 0xef, 0xf0, 0x00, 0x18, 0x6f, 0xf0, 0x00, 0x10, 0x2f,
  0xf0, 0x00, 0x10, 0x2f, 0xf0, 0x00, 0x18, 0x6f, 0xf0, 0x00, 0x1c, 0xef, 0xf0, 0x00,
  0x1f, 0xef, 0xf0, 0x00, 0x1c, 0xef, 0xf0, 0x00, 0x18, 0x6f, 0xf0, 0x00, 0x10, 0x2f,
  0xf0, 0x00, 0x10, 0x2f, 0xf0, 0x00, 0x18, 0x6f, 0xf0, 0x00, 0x1c, 0xef, 0xf0, 0x00,
  0x10, 0x2f, 0xf0, 0x00, 0x0f, 0xcf, 0xf0, 0x00, 0x0f, 0xf0, 0x00, 0x0f, 0x00, 0x00
};

// 'semaforoO', 32x32px
const unsigned char icon_semaforoO [] PROGMEM = {
  0x00, 0x0f, 0xff, 0xf0, 0x00,
  0x3c, 0x7c, 0x7c, 0x04, 0x38, 0x38, 0x3c, 0x04, 0x30, 0x10, 0x1f, 0xfc, 0x30, 0x10, 0x1f, 0xfc,
  0x38, 0x38, 0x3c, 0x04, 0x3c, 0x7c, 0x7c, 0x04, 0x0f, 0xff, 0xf0, 0x00, 0x00, 0x00, 0x00,
  0x7f, 0xff, 0xff, 0xfe, 0x7f, 0xff, 0xfe, 0x7f, 0xff, 0x7f, 0xff, 0x7f, 0xff, 0x7f, 0xff, 0x7f,
  0x7f, 0xff, 0xff, 0xfe, 0x7f, 0xff, 0x7f, 0xff, 0x7f, 0xff, 0x7f, 0xff, 0x7f, 0xff, 0x7f, 0xff,
  0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

// 'ruta', 32x32px
const unsigned char icon_ruta [] PROGMEM = {
  0x00, 0x00, 0x00, 0x00, 0x7f, 0xff, 0xfe, 0x7f, 0xff, 0x7f, 0xfe, 0x60, 0x00, 0x00, 0x00,
  0x60, 0x00, 0x00, 0x06, 0x60, 0x00, 0x06, 0x60, 0x00, 0x06, 0x60, 0x00, 0x06, 0x60, 0x00,
  0x60, 0x00, 0x00, 0x06, 0x60, 0x00, 0x06, 0x60, 0x00, 0x04, 0x00, 0x06, 0x60, 0x00, 0x06,
  0x60, 0x07, 0x00, 0x06, 0x60, 0x07, 0x80, 0x06, 0x60, 0x07, 0xc0, 0x06, 0x60, 0x07, 0xe0,
  0x60, 0x07, 0xe0, 0x06, 0x60, 0x07, 0xc0, 0x06, 0x60, 0x07, 0x80, 0x06, 0x60, 0x07, 0x00,
  0x60, 0x06, 0x00, 0x06, 0x60, 0x04, 0x00, 0x06, 0x60, 0x00, 0x06, 0x60, 0x00, 0x06, 0x60,
  0x60, 0x00, 0x06, 0x60, 0x00, 0x06, 0x60, 0x00, 0x06, 0x60, 0x00, 0x06, 0x60, 0x00, 0x06,
  0x60, 0x00, 0x00, 0x06, 0x60, 0x00, 0x06, 0x60, 0x00, 0x06, 0x60, 0x00, 0x06, 0x60, 0x00
};

// 'conector', 32x32px
const unsigned char icon_conector [] PROGMEM = {
  0x00, 0x00,
  0x0f, 0xff, 0xf0, 0x0f, 0xff, 0xf0, 0x0c, 0x00, 0x00, 0x30, 0x0c, 0x00, 0x00, 0x30,
  0x0c, 0x00, 0x30, 0x0c, 0x00, 0x00, 0x30, 0x0c, 0x00, 0x00, 0x30, 0x0c, 0x3f, 0xf8, 0x30,
  0x0c, 0x3f, 0xf8, 0x30, 0x0c, 0x3f, 0xf8, 0x30, 0x0c, 0x01, 0xf8, 0x30, 0x0c, 0x03, 0xf8, 0x30,
  0x0c, 0x07, 0xf8, 0x30, 0x00, 0x0f, 0xb8, 0x30, 0x00, 0x1f, 0x38, 0x30, 0x00, 0x3e, 0x38, 0x30,
  0x00, 0x7c, 0x38, 0x30, 0x00, 0x0f, 0xf8, 0x38, 0x30, 0x01, 0xf0, 0x00, 0x30, 0x03, 0xe0,
  0x00, 0x07, 0xc0, 0x00, 0x30, 0x0f, 0x80, 0x00, 0x30, 0x07, 0x01, 0xff, 0xf0, 0x02, 0x01, 0xff, 0xf0,
  0x00, 0x00
};

// 'PaNdwn', 32x32px
const unsigned char icon_PaNbjaj [] PROGMEM = {
  0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

// 'PaNup', 32x32px
const unsigned char icon_PaNsub [] PROGMEM = {
  0x00, 0x00,
  0x00, 0x00, 0x08, 0x00, 0x00, 0x00, 0x1c, 0x00, 0x00, 0x00, 0x3e, 0x00, 0x00, 0x00, 0x5c, 0x00,
  0x00, 0x00, 0x88, 0x00, 0x00, 0x01, 0xd0, 0x00, 0x00, 0x03, 0xe0, 0x00, 0x00, 0x05, 0xc0, 0x00,
  0x00, 0x08, 0x80, 0x00, 0x00, 0x01, 0x1d, 0x00, 0x00, 0x3e, 0x00, 0x00, 0x00, 0x5c, 0x00, 0x00,
  0x00, 0x88, 0x00, 0x00, 0x01, 0xd0, 0x00, 0x00, 0x03, 0xe0, 0x00, 0x00, 0x05, 0xc0, 0x00, 0x00,
  0x08, 0x80, 0x00, 0x00, 0x01, 0x1d, 0x00, 0x00, 0x3e, 0x00, 0x00, 0x00, 0x1c, 0x00, 0x00, 0x00,
  0xa0, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x07, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

const unsigned char icon_vacio [] PROGMEM = {
};

```

circuito.h

```
// TFT_TCO - Paco Cañada 2022 -- https://usuaris.tinet.cat/fmco/  
  
===== Macros y datos =====  
  
struct Simbolo {  
    byte tipo; // icono  
    byte posX; // columna 0..9 con 320x240  
    byte posY; // fila 0..6 con 320x240  
    unsigned int direccion; // direccion DCC 1..1024 (0..1023 Xpressnet) (Bit 14 activo:  
    mostrar posicion actual invertida)  
};  
  
enum posViaPaN { VERTICAL, HORIZONTAL };  
enum orientacion { NORTE, ESTE, SUR, OESTE };  
enum iconoOrientacion { NORDESTE, SUDESTE, SUDOESTE, NOROESTE, NE_SO, NO_SE };  
  
enum tipoSimbolo {  
    ICON_VIA_V, ICON_VIA_H, // con icono, sin direccion  
    ICON_CURVA_NE, ICON_CURVA_SE, ICON_CURVA_SO, ICON_CURVA_NO, ICON_CURVAS_NE_SO, ICON_CURVAS_NO_SE,  
    ICON_TOPE_N, ICON_TOPE_E, ICON_TOPE_S, ICON_TOPE_O, // sin icono, sin direccion  
    ICON_TEXTO, // con icono,  
    ICON_RUTA, // con icono,  
    ICON_CONECTOR, // con icono, con direccion  
    ICON_SEMAFORO_N, ICON_SEMAFORO_E, ICON_SEMAFORO_S, ICON_SEMAFORO_O, // sin icono, con direccion  
    ICON_PAN_S, ICON_PAN_B, // con icono, con direccion  
    ICON_DESVIO_V_NE, ICON_DESVIO_V_SE, ICON_DESVIO_V_SO, ICON_DESVIO_V_NO, // sin icono, con direccion  
    ICON_DESVIO_H_NE, ICON_DESVIO_H_SE, ICON_DESVIO_H_SO, ICON_DESVIO_H_NO,  
};  
  
#define NO_ASIGNADO 0x0FO0  
#define NO_EXISTE 0xFF  
#define INVERTIDO(x) (x | bit(14))  
  
#define VIA(o,x,y) {ICON_VIA_V+o,x,y,NO_ASIGNADO}  
#define CURVA(o, x, y) {ICON_CURVA_NE+o,x,y,NO_ASIGNADO}  
#define TOPE(o,x,y) {ICON_TOPE_N+o,x,y,NO_ASIGNADO}  
#define PAN(o,x,y,d) {ICON_PAN_S+o,x,y,d-1}  
#define RUTA(x,y,d) {ICON_RUTA,x,y,NO_ASIGNADO+d}  
#define CONECTOR(x,y,d) {ICON_CONECTOR,x,y,NO_ASIGNADO+d}  
#define SEMAFORO(o,x,y,d) {ICON_SEMAFORO_N+o, x,y,d-1}  
#define DESVIO_V(o,x,y,d) {ICON_DESVIO_V_NE+o, x,y,d-1}  
#define DESVIO_H(o,x,y,d) {ICON_DESVIO_H_NE+o, x,y,d-1}  
#define TEXTO(d) {ICON_TEXTO,NO_EXISTE,NO_EXISTE,NO_ASIGNADO+d}  
  
===== Colores del circuito =====  
  
#define COLOR_FONDO tft.color565(235, 235, 235)  
#define COLOR_FONDO_TITULO AQUA  
#define COLOR_FONDO_TIT_STOP PINK  
#define COLOR_TITULO WHITE  
#define COLOR_TEXTO BLACK  
#define COLOR_VIA BLACK  
#define COLOR_PAN RED  
#define COLOR_RUTA ORANGE  
#define COLOR_CONECTOR DARKCYAN  
#define COLOR_DESVIO_NO_POS DARKGREY  
#define COLOR_DESVIO_POS YELLOW  
  
===== Parametros del circuito =====  
  
const int tiempoAccesorioON = 150; // Tiempo de activacion accesorios  
const int tiempoCDU = 250; // Tiempo espera recarga CDU  
const int tiempoPosicionInfo = 300; // Tiempo entre peticion informacion posicion  
  
===== Definicion del circuito =====  
  
#define NUM_PANTALLAS 3  
  
#define LONG_PANTALLA(x) sizeof(x)/sizeof(Simbolo)  
  
const char texto0[] = {"@64,172[ ESTACION ]"};  
const char texto1[] = {"@74,150;64@202;61@170,118;62,66;63"}; // Y = +2,+12,+22 X= +10  
const char texto2[] = {"@138,86;69@170;70@230,140P.N. abierto@230,172P.N. cerrado"};  
const char texto3[] = {"@138,130;73"};  
  
const char ruta0[] = {"64R61R"}; // DESVIADO: ROJO - >  
const char ruta1[] = {"64D62D61D"}; // RECTO: VERDE + <  
const char ruta2[] = {"61D62R63D"};  
const char ruta3[] = {"70D69R"};  
const char ruta4[] = {"70R69D"};  
  
const char nombrePantalla0[] = {"SALOU"};  
const Simbolo pantalla0[] = {
```

```

TOPE(OESTE, 1, 2), VIA(HORIZONTAL, 2, 2), RUTA(3, 2, 2), VIA(HORIZONTAL, 4, 2), DESVIO_H(SUDOESTE, 5, 2, 63),
VIA(HORIZONTAL, 6, 2), TOPE(ESTE, 7, 2),
CURVA(SUDESTE, 2, 3), VIA(HORIZONTAL, 3, 3), RUTA(4, 3, 1), DESVIO_H(NORDESTE, 5, 3, INVERTIDO(62)),
CURVA(SUDOESTE, 6, 3),
TOPE(OESTE, 1, 4), DESVIO_H(NOROESTE, 2, 4, 64), RUTA(3, 4, 0), VIA(HORIZONTAL, 4, 4), VIA(HORIZONTAL, 5, 4),
DESVIO_H(NORDESTE, 6, 4, 61), VIA(HORIZONTAL, 7, 4), CONECTOR(8, 4, 1),
TEXTO(0), TEXTO(1),
};

const char nombrePantalla1[] = {"PASO A NIVEL"};
const Simbolo pantalla1[] = {
    PAN(VERTICAL, 4, 1, 69), PAN(VERTICAL, 4, 3, 69),
    CONECTOR(2, 2, 0), VIA(HORIZONTAL, 3, 2), VIA(HORIZONTAL, 4, 2), SEMAFORO(OESTE, 5, 2, 70), VIA(HORIZONTAL, 6,
2), CONECTOR(7, 2, 2),
    RUTA(6, 4, 3), RUTA(6, 5, 4),
    TEXTO(2),
};

const char nombrePantalla2[] = {"BUCLE LIGERO"};
const Simbolo pantalla2[] = {
    CURVA(SUDESTE, 5, 2), VIA(HORIZONTAL, 6, 2), CURVA(SUDOESTE, 7, 2),
    VIA(VERTICAL, 5, 3), VIA(VERTICAL, 7, 3),
    CONECTOR(2, 4, 1), VIA(HORIZONTAL, 3, 4), DESVIO_H(SUDOESTE, 4, 4, 73), CURVA(NOROESTE, 5, 4), VIA(VERTICAL, 7,
4),
    CURVA(NORDESTE, 4, 5), VIA(HORIZONTAL, 5, 5), VIA(HORIZONTAL, 6, 5), CURVA(NOROESTE, 7, 5),
    TEXTO(3),
};

// Inicializar los siguientes arrays con los valores adecuados a nuestro circuito

const char *textos[] = {texto0, texto1, texto2, texto3};
const char *rutas[] = {ruta0, ruta1, ruta2, ruta3, ruta4};
const Simbolo *Pantalla[NUM_PANTALLAS] = {pantalla0, pantalla1, pantalla2};
const int numIconosPantalla[NUM_PANTALLAS] = {LONG_PANTALLA(pantalla0), LONG_PANTALLA(pantalla1),
LONG_PANTALLA(pantalla2)};
const char *nombrePantallas[NUM_PANTALLAS] = {nombrePantalla0, nombrePantalla1, nombrePantalla2};

```

TFT_TCO.ino

```

// TFT_TCO - Paco Cañada 2022 -- https://usuaris.tinet.cat/fmco/
// Descomentar la siguiente linea sino funciona con la libreria Adafruit_TFTLCD.h o se desconoce el chip
#ifndef TIPO_DESCONOCIDO

#define TIPO_DESCONOCIDO
#define ROTACION_PANTALLA 1      // MCUFRIEND
#else
#define ROTACION_PANTALLA 3      // Adafruit
#endif

//===== Librerias y variables =====
#include <Adafruit_GFX.h>          // Libreria de graficos v1.5.0
#ifdef TIPO_DESCONOCIDO
#include <MCUFRIEND_kbv.h>          // Libreria especifica para el Hardware v2.9.9
#else
#include <Adafruit_TFTLCD.h>          // Libreria especifica para el Hardware v1.0.3
#endif
#include <TouchScreen.h>             // Libreria para panel tactil v1.1.3
#include <XpressNet.h>               // Libreria Xpressnet v2.1.0
#include "iconos.h"                  // iconos 32x32
#include "circuito.h"                // Definicion del circuito

// Definicion de las conexiones de la pantalla

#define LCD_CS    A3           // Chip Select
#define LCD_CD    A2           // Command/Data
#define LCD_WR    A1           // TFT Write
#define LCD_RD    A0           // TFT Read
#define LCD_RESET A4           // TFT Reset

// When using the BREAKOUT BOARD only, use these 8 data lines to the LCD:
// For the Arduino Uno, Duemilanove, Diecimila, etc.:
// D0 connects to digital pin 8 (Notice these are
// D1 connects to digital pin 9 NOT in order!)
// D2 connects to digital pin 2
// D3 connects to digital pin 3
// D4 connects to digital pin 4
// D5 connects to digital pin 5
// D6 connects to digital pin 6
// D7 connects to digital pin 7
// For the Arduino Mega, use digital pins 22 through 29
// (on the 2-row header at the end of the board).

```

```

// Colour definitions for 64K colour mode
// Bits 0..4 -> Blue 0..4
// Bits 5..10 -> Green 0..5
// Bits 11..15 -> Red 0..4
// Assign human-readable names to some common 16-bit color values:

#define BLACK      0x0000
#define BLUE       0x001F
#define RED        0x8000
#define GREEN      0x07E0
#define CYAN       0x07FF
#define MAGENTA    0xF81F
#define VIOLET     0x9199
#define BROWN      0x8200
#define PINK        0xF97F
#define YELLOW     0xFFE0
#define WHITE      0xFFFF
#define ORANGE     0xFD20
#define LIME        0x87E0
#define GREENYELLOW 0xAFE5
#define AQUA        0x5D1C
#define NAVY        0x000F
#define DARKGREEN   0x03E0
#define DARKCYAN    0x03EF
#define MAROON     0x7800
#define PURPLE     0x780F
#define OLIVE       0x7BE0
#define SILVER      0xA510
#define GOLD        0xA508
#define LIGHTGREY   0xC618
#define DARKGREY    0x7BEF

#ifndef TIPO_DESCONOCIDO
MCUFRIEND_kbv tft;                                // usar MCUFRIEND para pantallas no soportadas por Adafruit
#else
Adafruit_TFTLCD tft(LCD_CS, LCD_CD, LCD_WR, LCD_RD, LCD_RESET); // Usar Adafruit que es mas ligera
#endif

unsigned int ID;                                     // ID de la pantalla
// Usar TouchScreen_Calibr_native para obtener los valores. Libreria MCUFRIEND

const int XP = 8, YM = A2, YM = 9;                  //240x320 ILI9341
const int TS_LEFT = 250, TS_RT = 905, TS_TOP = 175, TS_BOT = 907;

// For better pressure precision, we need to know the resistance
// between X+ and X- Use any multimeter to read it
// For the one we're using, its 300 ohms across the X plate

TouchScreen ts = TouchScreen(XP, YM, XM, YM, 300);

//Minimum and maximum pressure to sense the touch
#define MINPRESSURE 10
#define MAXPRESSURE 1000

int X;                                              // Variables que almacenaran la coordenada
int Y;                                              // X, Y donde presionemos y la variable Z
int Z;                                              // almacenara la presion realizada

bool pulsacionDetectada = false;                    // pulsacion detectada en panel tactil
unsigned long tiempoRebote;                         // tiempo de espera para evitar rebotes

byte pasoConfiguracion;                            // paso configuracion panel tactil

const int pinTXRX = A5;                            // pin del MAX485
const int miDireccionXpressnet = 23;                // direccion Xpressnet

byte csStatus;                                     // Estado de la central
byte RS[128];                                      // Datos de los RS

XpressNetClass XpressNet;

const unsigned char* iconos[] = {
    icon_viaV, icon_viaH, icon_curvaNE, icon_curvaSE, icon_curvaSO, icon_curvaNO, icon_curvaNE_SO, icon_curvaNO_SE,
    icon_topeN, icon_topeE, icon_topeS, icon_topeO, icon_vacio, icon_ruta, icon_conector,
    icon_semaforoN, icon_semaforoE, icon_semaforoS, icon_semaforoO, icon_PaNsub, icon_PaNbaj,
};

enum posDesvio {NO_MOVIDO, DESVIADO, RECTO, INVALIDO}; // DESVIADO: ROJO - > links      RECTO:      VERDE + <
                                                       // rechts

byte pantallaActual;
int posTexto;

unsigned int cambiosFIFO[32];                        // cola FIFO para cambios en pantalla
byte cambioIn;                                     // indice cabeza cola
byte cambioOut;                                    // indice final cola
byte enColaCambios;                                // numero de accesorios en cola

```

```

unsigned int accesoriosFIFO[32];
byte accesorioIn;
byte accesorioOut;
byte enColaAccesorios;                                // cola FIFO para mover accesorios
                                                       // indice cabeza cola
                                                       // indice final cola
                                                       // numero de accesorios en cola

unsigned long tiempoAccesorio;                         // envio de accesorios
unsigned int accEnviado;
bool recargando, esperaAccesorio;                     // envio de rutas
unsigned long tiempoRuta = millis();                  // tiempo para pedir informacion
                                                       // contadores para pedir informacion
                                                       // pidiendo informacion iconos

//==== Programa principal =====

void setup() {
  ID = tft.readID();
  // Controladores disponibles: 0x9325, 0x9328, 0x7575, 0x9341, 0x8357
  tft.begin(ID);                                       // Iniciamos el LCD especificando el controlador ILI9341
  (0x9341).
  // Rotation: 0: portrait-USB top right, 2: portrait-USB bottom left, 1:landscape-USB bottom right, 3: landscape-USB top left.
  tft.setRotation(ROTACION_PANTALLA);                  // Establecemos la pantalla Horizontal
  for (int i = 0; i < 128; i++)                      // borra estado contactos
    RS[i] = 0;
  csStatus = csTrackVoltageOff;                        // Suponemos que la central esta sin tension en via
  pantallaActual = 0;                                  // pinta pantalla inicial
  borraAccesoriosFIFO();                             // borra colas
  borraCambiosFIFO();
  XpressNet.start(miDireccionXpressnet, pinTXRX);   // Inicializamos bus Xpressnet
  XpressNet.getPower();                               // Pedimos el estado actual de la central
  refrescaInformacion();                            // Pedimos informacion de la posicion de los iconos
}

void loop() {
  byte iconoPulsado, tipo, posicion;
  unsigned int direccion;
  int col, fila;

  XpressNet.receive();                                // recibimos paquetes por Xpressnet
  actualizaCambiosIconos();                          // actualiza cambio en iconos de la pantalla actual
  XpressNet.receive();                                // recibimos paquetes por Xpressnet
  enviaColaAccesorios();                            // envia accesorios de la cola
  XpressNet.receive();                                // recibimos paquetes por Xpressnet
  if (pideInformacion)                             // comprueba si hay informacion pendiente
    buscaInformacion();                            // pide informacion posicion iconos
  XpressNet.receive();                                // recibimos paquetes por Xpressnet

  TSPoint p = ts.getPoint();                         // Realizamos lectura de las coordenadas panel tactil
  pinMode(XM, OUTPUT);                             // La libreria utiliza estos pines como entrada y salida
  pinMode(YP, OUTPUT);                             // por lo que es necesario declararlos como salida justo despues
  de realizar una lectura de coordenadas.

  X = map(p.y, TS_TOP, TS_BOT, 0, tft.width());    // Mapeamos los valores analogicos leidos del panel tactil (0-1023)
  Y = map(p.x, TS_RT, TS_LEFT, 0, tft.height());
  Z = p.z;

  if (pulsacionDetectada) {                         // si se pulsa la pantalla
    if (millis() - tiempoRebote > 200)
      pulsacionDetectada = false;
  } else {
    if (Z > MINPRESSURE && Z < MAXPRESSURE) {      // si se pulsa la pantalla
      col = constrain (X, 0, tft.width() - 1) / 32;
      fila = constrain (Y, 0, tft.height() - 1) / 32;

      iconoPulsado = buscaIconoPulsado (col, fila);
      if (iconoPulsado != NO_EXISTE) {                // comprueba si hemos pulsado sobre un icono
        tipo = Pantalla[pantallaActual][iconoPulsado].tipo;
        if (tipo < ICON_SEMAFORO_N) {                 // via, ruta o conector
          if (tipo == ICON_CONECTOR) {                  // si es un conector cambiamos de pantalla
            pantallaActual = lowByte(Pantalla[pantallaActual][iconoPulsado].direccion);
            pintaPantalla();
          }
          if (tipo == ICON_RUTA) {                      // si es una ruta enviamos la ruta
            if (millis() - tiempoRuta > ((tiempoAccesorioON + tiempoCDU) * 3)) { // Esperar un tiempo prudencial
entre rutas
              pintaIcono(Pantalla[pantallaActual][iconoPulsado], RECTO);
              direccion = lowByte(Pantalla[pantallaActual][iconoPulsado].direccion);
              enviaRuta(direccion);
              pintaIcono(Pantalla[pantallaActual][iconoPulsado], NO_MOVIDO);
            }
          }
        }
      }
    }
  }
}

```

```

        else { // semaforo o desvio
            direccion = Pantalla[pantallaActual][iconoPulsado].direccion;
            posicion = buscaPosicionActual (direccion); // busca posicion actual
            if (posicion == RECTO) // cambiar a la otra posicion
                bitSet(direccion, 15); // bit 15 activo indica posicion Desviado
            writeAccesoriosFIFO(direccion);
            pulsacionDetectada = true;
            tiempoRebote = millis();
        }
    }
}

//==== Iconos y pantallas =====
void pintaIcono (Simbolo simbolo, byte posicion) {
    unsigned int color, colorLuz, x, y, offsetOn, offsetOff;
    byte icono, iconoR, iconoD;
    x = simbolo.posX * 32;
    y = simbolo.posY * 32;
    icono = simbolo.tipo;
    if (icono < ICON_DESVIO_V_NE) { // No son desvios
        color = COLOR_VIA; // color por defecto
        switch (posicion) { // calculamos luz en caso semaforos
            case DESVIADO:
                offsetOn = 8;
                offsetOff = 0;
                colorLuz = RED;
                break;
            case RECTO:
                offsetOn = 0;
                offsetOff = 8;
                colorLuz = GREEN;
                break;
            default:
                offsetOn = 0;
                offsetOff = 8;
                colorLuz = COLOR_FONDO;
                break;
        }
        switch (icono) { // casos especiales, RUTA, CONECTOR, SEMAFOROS
            case ICON_PAN_S:
                color = bitRead(posicion - 1, 1) ? COLOR_DESVIO_NO_POS : COLOR_PAN;
                tft.fillRect(x + 2, y + 4, 28, 21, COLOR_FONDO); // borra posicion anterior
                if (posicion == DESVIADO)
                    icono = ICON_PAN_B;
                break;
            case ICON_PAN_B:
                color = bitRead(posicion - 1, 1) ? COLOR_DESVIO_NO_POS : COLOR_PAN;
                tft.fillRect(x + 2, y + 4, 28, 21, COLOR_FONDO); // borra posicion anterior
                if (posicion == DESVIADO)
                    icono = ICON_PAN_S;
                break;
            case ICON_TEXTO:
                pintaTexto(lowByte(simbolo.direccion)); // pintamos texto y salimos
                return;
            case ICON_RUTA:
                color = (posicion == NO_MOVIDO) ? COLOR_RUTA : COLOR_VIA;
                break;
            case ICON_CONECTOR:
                color = COLOR_CONECTOR;
                break;
            case ICON_SEMAFORO_N:
                tft.fillRect(x + 22, y + 4 + offsetOff, 6, 7, COLOR_FONDO); // borra luz
                XpressNet.receive(); // recibimos paquetes por Xpressnet
                tft.fillRect(x + 22, y + 4 + offsetOn, 6, 7, colorLuz); // pinta Luz
                break;
            case ICON_SEMAFORO_E:
                tft.fillRect(x + 21 - offsetOff, y + 22, 7, 6, COLOR_FONDO); // recibimos paquetes por Xpressnet
                XpressNet.receive();
                tft.fillRect(x + 21 - offsetOn, y + 22, 7, 6, colorLuz);
                break;
            case ICON_SEMAFORO_S:
                tft.fillRect(x + 4, y + 21 - offsetOff, 6, 7, COLOR_FONDO); // recibimos paquetes por Xpressnet
                XpressNet.receive();
                tft.fillRect(x + 4, y + 21 - offsetOn, 6, 7, colorLuz);
                break;
            case ICON_SEMAFORO_O:
                tft.fillRect(x + 4 + offsetOff, y + 4, 7, 6, COLOR_FONDO); // recibimos paquetes por Xpressnet
                XpressNet.receive();
                tft.fillRect(x + 4 + offsetOn, y + 4, 7, 6, colorLuz);
                break;
        }
        XpressNet.receive(); // recibimos paquetes por Xpressnet
        tft.drawBitmap(x, y, iconos[icono], 32, 32, color);
    }
}

```

```

else {                                     // Son desvios
    if (icono < ICON_DESVIO_H_NE) {
        iconoR = ICON_VIA_V;           // Tramo recto vertical
        iconoD = icono - ICON_DESVIO_V_NE + ICON_CURVA_NE; // Tramo orientacion desviado (0..3) + base
    }
    else {
        iconoR = ICON_VIA_H;          // Tramo recto horizontal
        iconoD = icono - ICON_DESVIO_H_NE + ICON_CURVA_NE; // Tramo orientacion desviado (0..3) + base
    }
    switch (posicion) {
        case RECTO:
            tft.drawBitmap(x, y, iconos[iconoD], 32, 32, COLOR_DESVIO_NO_POS);      // recto
            XpressNet.receive();                                         // recibimos paquetes por Xpressnet
            tft.drawBitmap(x, y, iconos[iconoR], 32, 32, COLOR_DESVIO_POS);
            break;
        case DESVIADO:
            tft.drawBitmap(x, y, iconos[iconoR], 32, 32, COLOR_DESVIO_NO_POS);      // desviado
            XpressNet.receive();                                         // recibimos paquetes por Xpressnet
            tft.drawBitmap(x, y, iconos[iconoD], 32, 32, COLOR_DESVIO_POS);
            break;
        default:
            tft.drawBitmap(x, y, iconos[iconoR], 32, 32, COLOR_DESVIO_NO_POS);      // no movido o invalido
            XpressNet.receive();                                         // recibimos paquetes por Xpressnet
            tft.drawBitmap(x, y, iconos[iconoD], 32, 32, COLOR_DESVIO_NO_POS);
            break;
    }
}
}

void pintaTitulo() {
    int n, x, y;
    y = tft.height() - 16;
    tft.fillRect(0, y, tft.width(), 16, (csStatus == csNormal) ? COLOR_FONDO_TITULO : COLOR_FONDO_TIT_STOP);
    n = strlen(nombrePantallas[pantallaActual]);
    x = (tft.width() - (n * 12)) / 2;
    tft.setCursor(x, y);
    tft.setTextSize(2);
    tft.setTextColor(COLOR_TITULO);
    tft.print(nombrePantallas[pantallaActual]);
}

void pintaPantalla() {
    unsigned int direccion, n;
    byte posicion;
    tft.fillScreen(COLOR_FONDO);
    pintaTitulo();
    XpressNet.receive();                      // recibimos paquetes por Xpressnet
    for (n = 0; n < numIconosPantalla[pantallaActual]; n++) {
        direccion = Pantalla[pantallaActual][n].direccion;
        if (Pantalla[pantallaActual][n].tipo < ICON_SEMAFORO_N)
            posicion = NO_MOVIDO;
        else
            posicion = buscaPosicionActual(direccion);
        if (bitRead(direccion, 14))           // comprobamos si hay que invertirlo
            posicion ^= B11;
        pintaIcono (Pantalla[pantallaActual][n], posicion);
        XpressNet.receive();                  // recibimos paquetes por Xpressnet
    }
}

int extraeNumero (char *texto, int maximo) {
    int valor;
    valor = 0;
    while (isDigit (texto[posTexto]))           // mientras sea un numero
        valor = (valor * 10) + (texto[posTexto++]- '0'); // lo añadimos al valor
    valor = min (valor, maximo);               // no ha de ser superior al maximo
    return (valor);
}

void pintaTexto (int numTexto) {
    int x, y;
    char caracter;
    x = 0;
    y = 0;
    posTexto = 0;
    tft.setTextColor(COLOR_TEXTO);
    tft.setTextSize(1);
    while (caracter = textos[numTexto][posTexto++]) {
        switch (caracter) {                // caracteres poco usados: ! # $ % & ; , @ ^ _ | ~
            case '@':                   // coordenada X
                x = extraeNumero(textos[numTexto], tft.width() - 1);
                tft.setCursor(x, y);
                break;
            case ',':                   // coordenada Y
                y = extraeNumero(textos[numTexto], tft.height() - 1);
                tft.setCursor(x, y);
        }
    }
}

```

```

        case ';':                                // separador
            break;
        default:                                 // texto
            tft.print(caracter);
            break;
    }
}

//==== Busqueda datos =====
byte buscaIconoPulsado (int col, int fila) {      // busca icono pulsado en la pantalla actual
    byte x, y, n;
    for (n = 0; n < numIconosPantalla[pantallaActual]; n++) {
        x = Pantalla[pantallaActual][n]. posX;
        y = Pantalla[pantallaActual][n]. posY;
        if ((x == col) && (y == fila))
            return n;
    }
    return NO_EXISTE;                         // no encontrado
}

byte buscaPosicionActual(unsigned int direccion) { // busca posicion actual de un accesorio
    byte modulo, desplazar, posicion;
    modulo = direccion >> 2;                  // calculamos modulo RS en el que esta la direccion
    desplazar = (direccion & B11) * 2;          // y la posicion de sus bits de estado
    posicion = (RS[modulo] >> desplazar) & B11; // leemos posicion actual
    return posicion;
}

void refrescaInformacion() {
    tiempoInfo = millis();                    // inicializamos variables para pedir datos
    pantallaInfo = 0;
    iconInfo = 0;
    pideInformacion = true;                  // hay informacion pendiente de recibir
}

void buscaInformacion() {
    unsigned int direccion;

    if (millis() - tiempoInfo > tiempoPosicionInfo) { // espera un tiempo entre peticiones
        direccion = Pantalla[pantallaInfo][iconInfo].direccion & 0xFFFF;
        if (direccion < 1024) {                      // solo accesorios
            tiempoInfo = millis();                  // resetea tiempo espera
            XpressNet.getTrntInfo (highByte(direccion), lowByte(direccion)); // pide informacion a la central
        }
        iconInfo++;                                // siguiente icono de la pantalla
        if (iconInfo == numIconosPantalla[pantallaInfo]) { // informacion pantalla completa
            iconInfo = 0;                           // iconos proxima pantalla
            pantallaInfo++;
            if (pantallaInfo == NUM_PANTALLAS) {     // todas las pantallas
                pantallaInfo = 0;
                pideInformacion = false;             // ya hemos recibido toda la informacion
            }
        }
    }
}

//==== Callback Xpressnet =====
void notifyTrnt(uint8_t Adr_High, uint8_t Adr_Low, uint8_t Pos) { // Llamado por la libreria cuando hay cambios
en accesorios
    byte modulo, mascara, desplazar, nuevaPos;
    unsigned int direccion;

    direccion = (Adr_High << 8) + Adr_Low;           // calculamos modulo RS en el que esta la direccion
    modulo = direccion >> 2;                          // la mascara ver sus bits de estado
    desplazar = (direccion & 0x03) * 2;               // los datos de la nueva posicion
    mascara = B11 << desplazar;                      // si cambia la posicion respecto a la guardada
    nuevaPos = (Pos & B11) << desplazar;              // borramos estado actual
    if ((RS[modulo] ^ nuevaPos) & mascara) {          // ponemos nuevo estado
        RS[modulo] &= ~mascara;
        RS[modulo] |= nuevaPos;
        writeCambiosFIFO (direccion);                  // guardamos direccion en la FIFO para actualizar pantalla
    }
}

void notifyXNetPower (uint8_t State) {                // La libreria llama a esta funcion cuando cambia estado de la
central
    csStatus = State;
    pintaTitulo();                                     // guardamos el estado
                                                        // Cambiamos el color de fondo del titulo segun el estado
}

```

```

//==== Cola accesorios =====

void borraAccesoriosFIFO () {
    accesorioIn = 0;                                // indice cabeza cola
    accesorioOut = 0;                               // indice final cola
    enColaAccesorios = 0;                           // numero de accesorios en cola
    recargando = false;                            // CDU cargada
    esperaAccesorio = false;                         // ningun accesorio activado enviado
}

unsigned int readAccesoriosFIFO () {
    accesorioOut = (accesorioOut + 1) & 0x1F;        // avanza puntero circular
    enColaAccesorios--;
    return (accesoriosFIFO[accesorioOut]);
}

void writeAccesoriosFIFO (unsigned int accesorio) {
    accesorioIn = (accesorioIn + 1) & 0x1F;        // avanza puntero circular
    enColaAccesorios++;
    accesoriosFIFO[accesorioIn] = accesorio;        // lo guarda en la cola
}

void enviaColaAccesorios() {
    byte adrH, adrL, pos;
    if (recargando) {                                // esperando a que se recargue la CDU.
        if (millis() - tiempoAccesorio > (tiempoCDU)) {
            recargando = false;
        }
    } else {
        if (esperaAccesorio) {
            if (millis() - tiempoAccesorio > (tiempoAccesorioON)) {
                adrH = highByte (accEnviado) & 0x03;
                adrL = lowByte (accEnviado);
                pos = bitRead(accEnviado, 15) ? B0000 : B0001; // bit 15 activo indica posicion Desviado
                XpressNet.setTrntPos (adrH, adrL, pos);      // A00P Envia desconectar accesorio
                tiempoAccesorio = millis();
                esperaAccesorio = false;
                recargando = true;
            }
        } else {
            if (enColaAccesorios > 0) {
                accEnviado = readAccesoriosFIFO();
                adrH = highByte (accEnviado) & 0x03;
                adrL = lowByte (accEnviado);
                pos = bitRead(accEnviado, 15) ? B1000 : B1001; // bit 15 activo indica posicion Desviado
                XpressNet.setTrntPos (adrH, adrL, pos);      // A00P
                tiempoAccesorio = millis();
                esperaAccesorio = true;
            }
        }
    }
}

void enviaRuta(unsigned int numRuta) {
    unsigned int pos, direccion;
    char caracter;
    pos = 0;
    direccion = 0;
    while (caracter = rutas[numRuta][pos]) {
        switch (caracter) {
            case 'D':
                bitSet(direccion, 15);           // desviado bit 15 activo
            case 'R':
                direccion--;                   // Las direcciones empiezan en 0
                writeAccesoriosFIFO(direccion); // guarda en cola
                direccion = 0;                 // resetea calculo direccion
                pos++;
                break;
            default:
                direccion = (direccion * 10) + (rutas[numRuta][pos] - '0'); // añade valor a direccion
                break;
        }
    }
    tiempoRuta = millis();                      // timeout para rutas
}

//==== Cola cambios iconos =====

void borraCambiosFIFO () {
    cambioIn = 0;                                // indice cabeza cola
    cambioOut = 0;                               // indice final cola
    enColaCambios = 0;                           // numero de accesorios en cola
}

```

```

unsigned int readCambiosFIFO () {
    cambioOut = (cambioOut + 1 ) & 0x1F;           //avanza puntero circular
    enColaCambios--;
    return (cambiosFIFO[cambioOut]);
}

void writeCambiosFIFO (unsigned int direccion) {
    cambioIn = (cambioIn + 1 ) & 0x1F;           // avanza puntero circular
    enColaCambios++;
    cambiosFIFO[cambioIn] = direccion;          // lo guarda en la cola
}

void actualizaCambiosIconos() {
    unsigned int direccion;
    byte posicion, n;
    if (enColaCambios > 0) {
        direccion = readCambiosFIFO();
        posicion = buscaPosicionActual(direccion);
        for (n = 0; n < numIconosPantalla[pantallaActual]; n++) { // Puede haber varios con la misma direccion
            if (direccion == (Pantalla[pantallaActual][n].direccion & 0xFFFF)) { // 0..1023
                if (bitRead(Pantalla[pantallaActual][n].direccion, 14))
                    pintaIcono (Pantalla[pantallaActual][n], posicion ^ B11);
                else
                    pintaIcono (Pantalla[pantallaActual][n], posicion);
                XpressNet.receive(); // recibimos paquetes por Xpressnet
            }
        }
    }
}

```

Video

Video: <https://www.youtube.com/watch?v=Nh4lX98AqSI>

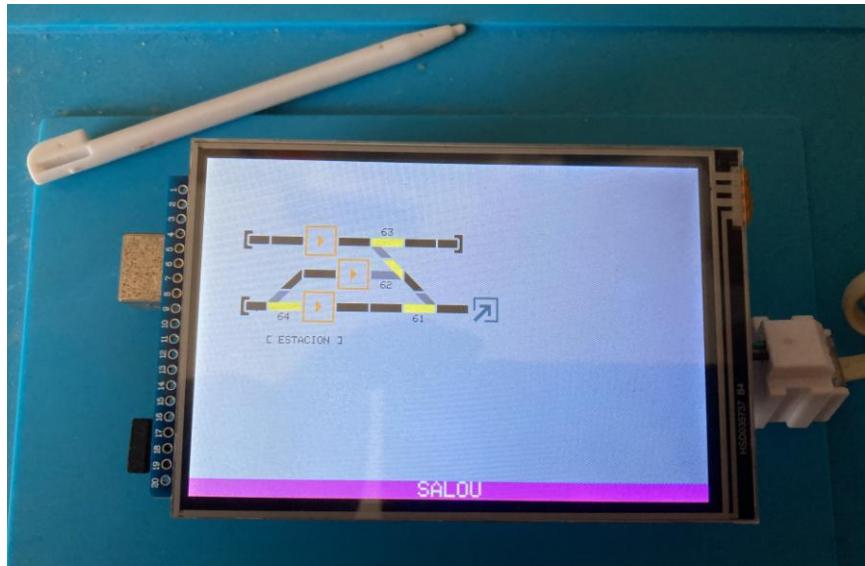


Variaciones

He comprado otra *shield TFT* de 480x320 pixel, esta tenía el chip ILI9488, así que lo único que he cambiado en el programa ha sido indicar que use la librería `MCUFRIEND_kbv.h` descomentando el `#define` y calibrar la pantalla con el programa `TouchScreen_Calibr_native.ino` que me ha proporcionado estos datos para copiarlos en el programa:

```
const int XP = 9, XM = A3, YP = A2, YM = 8; // ILI9488 480x320
const int TS_LEFT = 145, TS_RT = 913, TS_TOP = 72, TS_BOT = 956;
```

Como hemos usado parámetros genéricos y usado `width()` y `height()` se ha adaptado correctamente, ahora dispongo de 15 columnas y 10 filas (en realidad 9,5) para dibujar mi TCO. 😊



[link](#)

Mi elección de colores quizás no sea la mejor o no este al día con el diseño gráfico actual por lo que podéis probar otras combinaciones de colores, por ejemplo para el moderno 'dark mode' podéis cambiar estos colores:

```
#define COLOR_FONDO      BLACK           // Dark mode
#define COLOR_TEXTO        WHITE
#define COLOR_VIA          LIGHTGREY
#define COLOR_CONECTOR     LIME
```



[link](#)

NOTA: El consumo de estas pantallas es alto del orden de 100mA a 150mA por lo que si tenéis varios dispositivos en el bus Xpressnet tenedlo en cuenta.

Anexo I. Referencia al Lenguaje

El lenguaje de programación Arduino se puede dividir en tres partes principales: funciones, valores (variables y constantes) y estructura.

<https://www.arduino.cc/reference/en/>

NOTA: Se indica el capítulo en el que se ha tratado o usado por primera vez.

FUNCIONES

Para controlar la placa Arduino y realizar cálculos.

E/S Digitales		Matemáticas	Números Aleatorios
<code>digitalRead()</code>	3	<code>abs()</code> (valor absoluto)	<code>random()</code> 7
<code>digitalWrite()</code>	2	<code>constrain()</code> (limita)	<code>randomSeed()</code> 7
<code>pinMode()</code>	2	<code>map()</code> (cambia rango)	
		<code>max()</code> (máximo)	
		<code>min()</code> (mínimo)	
E/S Analógicas		<code>pow()</code> (eleva a un número)	
<code>analogRead()</code>	5, 12	<code>sq()</code> (eleva al cuadrado)	
<code>analogReference()</code>	21	<code>sqrt()</code> (raíz cuadrada)	
<code>analogWrite()</code> (PWM)	12		
Zero, Due & MKR Family		Trigonometría	
<code>analogReadResolution()</code>		<code>cos()</code> (coseno)	
<code>analogWriteResolution()</code>		<code>sin()</code> (seno)	
		<code>tan()</code> (tangente)	
E/S Avanzadas		Caracteres	Interrupciones Externas
<code>noTone()</code>	13	<code>isAlpha()</code>	<code>attachInterrupt()</code> 16
<code>pulseIn()</code>		<code>isAlphaNumeric()</code>	<code>detachInterrupt()</code> 21
<code>pulseInLong()</code>		<code>isAscii()</code>	<code>digitalPinToInterrupt()</code> 13, 16
<code>shiftIn()</code>	14	<code>isControl()</code>	
<code>shiftOut()</code>	16	<code>isDigit()</code>	
<code>tone()</code>	13	<code>isGraph()</code>	
Tiempo		<code>isHexadecimalDigit()</code>	
<code>delay()</code>	2	<code>isLowerCase()</code>	Interrupts
<code>delayMicroseconds()</code>	14	<code>isPrintable()</code>	<code>interrupts()</code> 16
<code>micros()</code>		<code>isPunct()</code>	<code>noInterrupts()</code> 16
<code>millis()</code>	7	<code>isSpace()</code>	
		<code>isUpperCase()</code>	
		<code>isWhitespace()</code>	
			Communicaciones
			<code>Serial</code>
			<code>Stream</code> 11, 14
			USB
			<code>Keyboard</code>
			<code>Mouse</code>

VARIABLES

Tipos de datos Arduino y constantes.

Constantes		Tipo de datos		Alcance y Calificadores	
Floating Point Constants		String()	10	const	7
Integer Constants	11,12	array (cadena)	7	scope	
HIGH - LOW	2	bool	13	static	
INPUT OUTPUT INPUT_PULLUP	2	boolean		volatile	16
LED_BUILTIN	16	byte	11		
true false	10	char (carácter)	11	Utilidades	
		double		PROGMEM	22
Conversión		float (en coma flotante)		sizeof()	10
(unsigned int)		int (entero16b)	3		
(unsigned long)		long (entero 32b)	7	Otros	
byte()		short		enum	12,15
char()		size_t		struct	9,15
float()		string		union	16
int()	11	unsigned char			
long()		unsigned int			
word()		unsigned long	7		
		void			
		word	1,4		

ESTRUCTURA

Los elementos del código Arduino (C ++).

Sketch		Operadores aritméticos		Operador acceso puntero	
setup () (inicialización)	1	% (resto)	16	& (referencia)	11
loop () (bucle)	1	* (multiplicación)	11	*	11
		+ (suma)	4		
Estructuras de control		- (resta)			
? : (condicional ternario)	11	/ (división)	9	Operadores bit a bit	
break	8	= (asignación)	3	& (y)	12
continue				<< (desplazam. izquierda)	12,14
do...while	9	Operadores comparación		>> (desplazam. derecha)	12
else	3	!= (distinto de)	9	^ (xor)	17
for	4	< (menor que)	7	(o)	12
goto-	-	<= (menor o igual que)	4	- (no)	
if	3	== (igual a)	3	Operadores compuestos	
return (devuelve valor)	7	> (mayor que)	10	%= (resto compuesto)	
switch...case	8	>= (mayor o igual que)	10	&= (compuesto bit a bit y)	
while	9			*= (multiplicación comp.)	
Más sintaxis		Operadores booleanos		+= (incremento)	4,10
#define (define)	1,5	! (negación)	13	+= (adición compuesta)	4
#include (incluir librería)	1,5	&& (y)	6	-- (decremento)	10
#if	17	(o)	10	--= (resta compuesta)	
#endif	17			/= (división compuesta)	
#error	17			^= (compuesto bitor xor)	17
/* */ (comentario bloque)	1			= (compuesto bit a bit o)	14
// (comentario linea)	1				
;	1				
{ } (curly braces)	1				

Anexo II. Librerías

Estas son las librerías a las que se hace uso o referencia en los diferentes capítulos o que son útiles para aplicarlos al modelismo ferroviario.

Aquí tenéis la referencia de las incluidas en el Arduino IDE:

<https://www.arduino.cc/en/Reference/Libraries>

NOTA: Se indica el capítulo en el que se ha usado por primera vez.

NmraDcc.h	https://github.com/mrrwa/NmraDcc	DCC	6
DCC_Decoder.h	https://github.com/MynaBay/DCC_Decoder	DCC	
DCCpp.h	https://github.com/Locoduino/DCCpp	DCC (central)	
MaerklinMotorola.h	https://github.com/Laserlicht/MaerklinMotorola	MM	
Wire.h	https://www.arduino.cc/en/Reference/Wire	I2C	11
LocoNet.h	https://github.com/mrrwa/LocoNet	Loconet	
RSbus.h	https://github.com/aikopras/RSbus/tree/master/src	RS	
RCN600.h	https://github.com/TheFidax/RCN600	SUSI	
Stepper.h	https://www.arduino.cc/en/Reference/Stepper	Motor paso a paso	
AccelStepper.h	https://www.airspayce.com/mikem/arduino/AccelStepper	Motor paso a paso	9
FastLED	https://github.com/FastLED/FastLED	NeoPixel (WS2812)	
Adafruit_NeoPixel.h	https://github.com/adafruit/Adafruit_NeoPixel	NeoPixel (WS2812)	21
Adafruit_SSD1306.h	https://github.com/adafruit/Adafruit_SSD1306	OLED	
Adafruit-GFX-Library.h	https://github.com/adafruit/Adafruit-GFX-Library	OLED/TFT	22
Adafruit_TFTLCD.h	https://github.com/adafruit/TFTLCD-Library	TFT	22
u8glib.h	https://github.com/olikraus/u8glib	OLED	
u8g2.h	https://github.com/olikraus/u8g2	OLED	
SSD1306Ascii.h	https://github.com/greiman/SSD1306Ascii	OLED	11
MCUFRIEND_kbv.h	https://github.com/prenticedavid/MCUFRIEND_kbv	TFT	22
TouchScreen.h	https://github.com/adafruit/Adafruit_TouchScreen	Pantalla táctil	22
uRTCLib.h	https://github.com/Naguissa/uRTCLib	Reloj (DS3231)	11
SD.h	https://www.arduino.cc/en/reference/SD	SD	8
SoftwareSerial.h	https://www.arduino.cc/en/Reference/SoftwareSerial	Serial	
Servo.h	https://www.arduino.cc/en/Reference/Servo	Servo	5
ServoTimer2.h	https://github.com/nabontra/ServoTimer2	Servo	
PWMServo.h	https://github.com/PaulStoffregen/PWMServo	Servo	
VarSpeedServo.h	https://github.com/netlabtoolkit/VarSpeedServo	Servo	10
DFRobotDFPlayerMini.h	https://github.com/DFRobot/DFRobotDFPlayerMini	Sonido (DFPlayer)	18
TMRpcm.h	https://github.com/TMRh20/TMRpcm	Sonido (SD)	8
SPI.h	https://www.arduino.cc/en/reference/SPI	SPI	
Keypad.h	https://github.com/Chris--A/Keypad	Teclado	17
TimerOne.h	https://github.com/PaulStoffregen/TimerOne	Timer	16
XpressCommand.h	https://github.com/nzin/xpressnet_arduino	Xpressnet	
XpressNet.h	http://sourceforge.net/projects/pgahtow	Xpressnet	16
XpressNetMaster.h	http://sourceforge.net/projects/pgahtow	Xpressnet (central)	
Semaforo.h	https://usuaris.tinet.cat/fmco/Semaforo.zip	Semaforo	19
efsm.h	https://github.com/cubiwan/easy-finite-state-machine-arduino	FSM	
LowPower.h	https://github.com/rocketscream/Low-Power	Baja Potencia	21