Trabajo Fin de Grado
Grado en Ingeniería de Tecnologías de las Telecomunicaciones

Diseño de un transmisor y un receptor digital en Arduino, basados en la modulación de señales digitales

Autor: Laura Pérez Alonso

Tutor: F. Javier Payán Somet

Dep. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Departamento de Teoría de la Señal y Comunicaciones

Trabajo Fin de Grado
Grado en Ingeniería de Tecnologías de las Telecomunicaciones

Diseño de un transmisor y un receptor digital en Arduino, basados en la modulación de señales digitales

Autor:
Laura Pérez Alonso

Tutor:
F. Javier Payán Somet
Profesor Titular

Dep. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo Fin de Grado: Diseño de un transmisor y un receptor digital en Arduino, basados en la modulación de señales digitales

Autor: Laura Pérez Alonso
Tutor: F. Javier Payán Somet

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

Durante la elaboración de este trabajo, ha sido fundamental el apoyo de mi familia. También quiero agradecer el interés de Alejandro Gamero Salas, que me ha ayudado constantemente a devolver mi mente a sus cabales.

Agradezco la ayuda de mis profesores, entre ellos mi tutor F.Javier Payán Somet, la profesora María Jose Madero Ayora, y especialmente quiero agradecer el apoyo del profesor Ignacio Alvarado Aldea, que me ha brindado una valiosa ayuda en esta tecnología.

Laura Pérez Alonso
Sevilla, 2017

Resumen

En el transcurso del grado, se ha estudiado el funcionamiento de los sistemas de comunicaciones desde diferentes perspectivas, pero en ocasiones, este aprendizaje se ha enfocado en un plano puramente teórico. Por ese motivo, me pareció interesante enfocar el asunto desde un punto de vista práctico, para que tanto los lectores como yo, podamos experimentar su implementación en un microcontrolador, y cambiar un poco nuestro punto de vista.

Abstract

During the degree, the operation of communication systems has been studied from different perspectives, but sometimes this learning has been focused on a purely theoretical level. For that reason, I found it interesting to approach the issue from a practical point of view, so that both readers and I can experience its implementation in a microcontroller, and change our point of view a little bit.

Índice Abreviado

Índice

1 Introducción

El objetivo de este trabajo es desarrollar un ejemplo práctico de un sistema de comunicaciones basándose en la plataforma de software libre Arduino, utilizando los conocimientos adquiridos sobre comunicaciones digitales a lo largo de la carrera.

Con este fin, se ha desarrollado el software para un par de placas Arduino, que van a funcionar como transmisor y receptor respectivamente. A través de una explicación didáctica, se pretende familiarizar al lector al uso de esta herramienta, y a combinarla con el software Matlab, para permitirle hacer pruebas a tiempo real, de bajo coste.

El diseño del sistema se basa en la transmisión de señales por radiofrecuencia, pero se ha preferido modificarlo para comunicar las placas por cable, con el fin de reducir costes. Dicho medio se comporta de una forma demasiado idílica para transmisiones de corta distancia, de modo que se ha añadido el ruido por software para hacer las pruebas.

1.1 Los inicios del trabajo

El primer paso antes de comenzar el trabajo fue la documentación sobre Arduino y sus capacidades. Al principio surgió la idea de desarrollar una pareja transmisor – receptor que se comunicasen por radiofrecuencia. Para ponerlo en práctica se encontraron dos alternativas. La primera, buscar por separado cada uno de los elementos necesarios para transmitir la señal en por el canal radio, véanse filtros, osciladores, mezcladores, amplificadores y antenas.

Planteando esta opción, se debe tener en cuenta que la frecuencia de muestreo de la placa de Arduino es baja [introducir frecuencia] debido a que entre muestra y muestra es necesaria una conversión analógica. Siendo éste el caso, se descarta la posibilidad de elevar la frecuencia de la señal por software para introducirla en el canal, y es necesario realizar la conversión en frecuencia con oscilador externo y un mezclador, siendo necesarias dos etapas para introducir la señal en una banda libre como la de 868 MHz con un coste razonable.

Otro aspecto remarcable, es que la interconexión entre elementos a alta frecuencia debe llevarse a cabo utilizando líneas microstrip, y la señal requiere de un nivel de filtrado y amplificación cuyo diseño se escapa a la extensión del trabajo. Por estos motivos se planteó de inmediato una segunda opción, utilizar un trasceptor prediseñado e integrado en una pequeña placa que recibiese como entrada la señal modulada y enviase como salida la señal elevada en frecuencia y dispuesta para el canal.

El problema en este caso fue encontrar un elemento con esa funcionalidad, ya que existen en el mercado dispositivos que reciben señales binarias, las modulan y transmiten pero en raras ocasiones permiten elegir la modulación a usar y menos aún recibir como entrada la señal ya modulada.

Dada la dificultad para encontrar los elementos necesarios se optó por simplificar el diseño, cambiando el medio del aire por un sencillo cable. De esta manera el trabajo podría enfocarse directamente al diseño del sistema de comunicaciones digital, tratando digitalmente la señal como si se enviase por un medio más complejo.

1.2 El desarrollo

Una vez definido el trabajo se comenzó a estudiar cómo realizar una transmisión básica entre dos placas. El primer problema a abordar fue el formato de las salidas de la placa. Este aspecto se explica de nuevo más

adelante, pero de momento es suficiente con entender que para realizar la modulación es necesario disponer de un pin de salida de tensión analógico, y en este caso, lo más cercano a él es una salida PWM.

La modulación de señales por ancho de pulso, conocida como PWM, es una técnica que consiste en la generación de pulsos rectangulares periódicos, modificando su ciclo de trabajo para obtener un valor medio determinado. En este sentido, se denomina ciclo de trabajo al porcentaje de tiempo que la señal permanece a nivel alto en un periodo, respecto al tiempo que dura el periodo completo.

Para convertir la señal PWM a analógica es suficiente con filtrarla, ya que en su valor medio se encuentran los valores analógicos transmitidos. Éste filtrado puede hacerse de forma digital o analógica. En éste caso se decidió hacer un filtrado analógico para quitarle carga computacional al microprocesador.

Por otro lado, es interesante dedicarle especial atención al sincronismo entre placas. Para abordar este tema, fué conveniente esperar a tener el proyecto en una fase más avanzada de desarrollo y contar con más experiencia en el manejo del software. Por ello, se decidió realizar una sincronización inicial provisional, que permitió realizar las pruebas del modulador y demodulador, sin perder tiempo en el cuello de botella que supuso este tema en su momento.

A continuación se procedió a desarrollar el algoritmo de modulación y demodulación para una QPSK, basada en el pulso de raíz de coseno alzado. Durante esta fase, fué necesario implementar manualmente la correlación para el caso particular de la raíz de coseno alzado, debido a que las librerías de Arduino no implementan dicha función, o al menos, no de la forma que interesa en el proyecto. Para ello se realizaron simulaciones en Matlab que fueron muy útiles para la comprobación de resultados.

Una vez validada la primera transmisión, se empezó a utilizar una segunda sincronización, más robusta que la primera, que se explica en detalle más adelante. Finalmente, se desarrollaron dos versiones de la pareja transmisor-receptor, cada cual con una utilidad ligeramente diferente, como se detalla en siguientes apartados.

2 Arduino

Arduino es una plataforma electrónica de software libre, basada en una placa con un microcontrolador y un software que facilita su uso. Surgió en el *Ivrea Interaction Design Institute* con el objetivo de proporcionar una herramienta sencilla que pudiesen usar sus estudiantes para diseñar sus prototipos sin necesidad de tener un amplio conocimiento de electrónica ni programación.

Durante los últimos años la placa *Arduino* se ha seguido modificando dando lugar a una amplia gama de placas que facilitan la interconexión con otras tecnologías. Entre ellas se encuentran las placas convencionales y los *shield*, una placas compatibles con *Arduino* que se pueden colocar en la parte superior de los mismos para extender sus capacidades. Existen shields para facilitar el control de motores, elacceso a Ethernet, Wifi, y otras numerosas aplicaciones.

En resumen, es una herramienta idónea para llevar a cabo proyectos docentes, dado su bajo coste, sencillez y compatibilidad de software y hardware. En este trabajo se utiliza el modelo *Arduino UNO*.

2.1 La placa de Arduino UNO

La placa de *Arduino UNO* utiliza el microcontrolador *ATmega328*, así que toda la documentación relacionada con el uso de los registros dependerá de éste. Cuenta con 14 pines digitales de entrada/salida, 6 de los cuales pueden utilizarse como salida PWM, y 6 pines de entrada analógica. Más adelante se tratan con más detenimiento.

Utiliza un reloj de cuarzo de 16 MHz, ICSP, comunicación serie I2C, SPI y UART, conexión USB (puerto serie) y un puerto Jack para alimentación, aunque la placa también puede alimentarse a través del USB. Además dispone de un conjunto de LEDs para indicar si la placa está alimentada o si hay intercambio de datos por el puerto serie.

2.1.1 Los pines de la placa

Entradas y salidas digitales

Los pines 0-13 pueden escribir y leer los valores HIGH (5V) y LOW(0V) en el bus, si se configuran como pines digitales. Para ello es necesario declarar el modo de dichos pines como INPUT, OUTPUT o INPUT_-PULLUP previamente usando la función pinMode() y para leer o escribir los valores HIGH o LOW se emplean las funciones `digitalRead()` y `digitalWrite()`.

Por defecto, los pines se encuentran en modo INPUT. Se utiliza el modo INPUT_PULLUP para habilitar la resistencia interna de pullup de 20kΩ. Ésta configuración se utiliza para evitar leer valores indeterminados de ruido cuando no se conecta nada al pin, se conecta internamente el pin a 5V de modo que en reposo se lee HIGH y al conectar tierra se lee LOW.

El pin 13 está conectado a un LED de la placa por lo que al conectarlo en el modo INPUT_PULLUP no alcanzará los 5V, así que es más conveniente usarlo en modo INPUT y colocar una resistencia de pullup manualmente si se necesita. Los pines digitales suministran y reciben hasta 40mA, aunque se recomienda trabajar con 20mA, y hasta un total de 200mA.

Salidas analógicas

Los pines 3,5,6,9,10 y 11 pueden funcionar como salida pseudoanalógica PWM (*Pulse Width Modulation*) usando la función `analogWrite()`. En general, tanto las entradas como las salidas analógicas no necesitan declararse como INPUT ó OUTPUT previamente.

Se permite escribir de 0-5V a partir de un rango de valores de 0-255, lo que se traduce en ±9.8mV de precisión. La señal de salida PWM debe ser filtrada posteriormente si se requiere una señal analógica convencional. En este aspecto, conviene tener en cuenta que la velocidad del convertidor influye en el rizado de salida, siendo conveniente aumentar la tasa de conversión del DAC modificando los registros del convertidor.

Entradas analógicas

Los pines A0-A5 leen tensiones de 0 a 5V por defecto, aunque es posible cambiar límite superior indicando la nueva referencia con el pin AREF y usando la función `analogReference()`. La medida se lee con la función `analogRead()`.

El ADC proporciona un máximo de 10 bits de resolución, es decir, genera un valor entero en un rango de 0-1023 para representar los 0-5V de entrada, lo cual se traduce en una precisión de ±2.44mV, en su configuración por defecto. En estas condiciones el convertidor tarda aproximadamente 100µs en proporcionar la medida. Es posible reducir la precisión para aumentar la tasa de conversión, modificando los registros del ADC.

Otros pines

- Interrupciones externas en los pines 2 (INT0) y 3 (INT1).
- LED en el pin 13.
- Comunicación serie UART en los pines 0 (TX) y 1 (RX).
- Comunicación serie SPI en los pines 10 (SS), 11 (MOSI), 12 (MISO) y 13 (SCK).
- Comunicación serie I2C (TWI) en los pines A4 (SDA) y A5 (SCL).
- Botón de reset.

2.1.2 La alimentación

Se recomienda una alimentación externa entre 7 y 12 Voltios. Ésta se introduce desde el puerto USB, por puerto Jack o por el pin VIN. La placa proporciona salidas de 5V, 3.3V y tierra (GND) a través de los pines con dicho nombre. El pin IOREF sirve para que la placa reconozca el tipo de alimentación que necesita cada shield.

2.1.3 La memoria

El *ATmega328* tiene 32KB de memoria FLASH (0.5KB ocupados en el arranque), 2KB de SRAM y 1KB de EEPROM, que puede leerse y escribirse con su propia librería.

2.1.4 Sobre ICSP

In-Circuit Serial Programming es la característica de algunos dispositivos lógicos programables, como microcontroladores, que les permite ser programados una vez instalados en el sistema completo. Es útil si es necesario reemplazar al microcontrolador que viene de fábrica por otro nuevo y vacío, porque sirve para instalarle el gestor de arranque.

2.2 El entorno de trabajo

Arduino proporciona un *Entorno de Desarrollo Integrado* (IDE), llamado *Arduino Software* que nos permite programar desde el mismo, compilar, copiar los ficheros a la placa desde el puerto serie, usando un cable USB, y monitorizar las variables del programa en tiempo real. Existen otras alternativas para conseguir una mejor monitorización, pero en este trabajo será suficiente con usar el IDE básico y complementarlo con *Simulink*, cuando sea necesario.

A continuación se presentan los controles básicos del IDE *Arduino Software*. Éste entorno puede descargarse gratuitamente de la página oficial de *Arduino*, y una vez abierto presenta el siguiente aspecto.

Como puede observarse, la ventana consta principalmente de una barra de herramientas, un editor de código y una consola inferior, que sirve como salida de errores. Al inicio se abre el último proyecto con el que se trabajó.

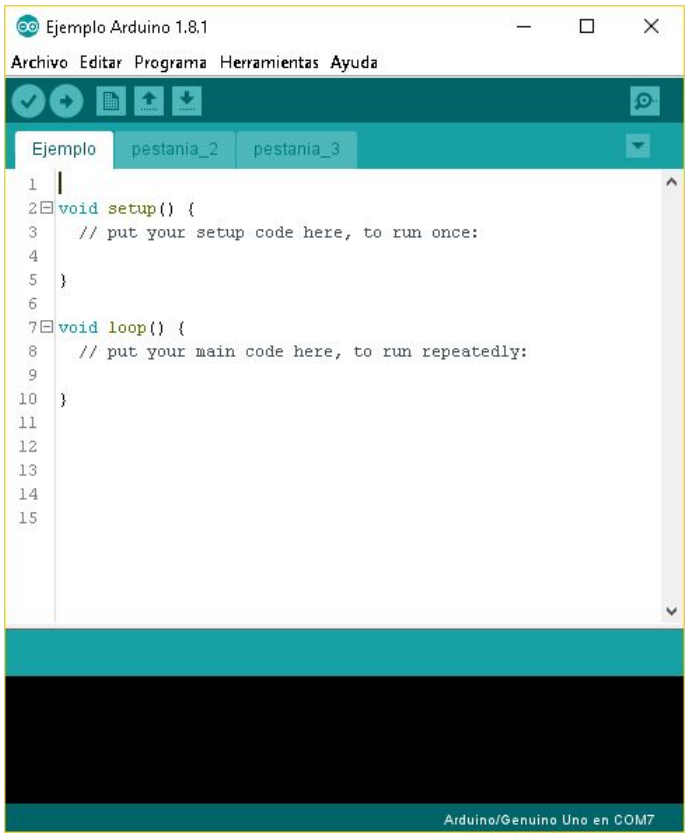

Figura 2.1 IDE de *Arduino Software*.

Un proyecto es una agrupación de archivos que funcionan en conjunto. Es importante recordar que el fichero principal del proyecto debe tener el mismo nombre que este. Los ficheros de código convencionales llevan la extensión `.ino`, mientras que las librerías se crean con otras extensiones. En este IDE los diversos ficheros `.ino` del proyecto se consideran una continuación natural unos de otros, por lo que es posible dividir el código en distintas partes para hacer el proyecto más manejable, sin necesidad de importar en cada fichero los anteriores. Esto se comenta más adelante.

A continuación se comentan las opciones más comunes de la barra de herramientas. En la siguiente imagen se encuentran los botones: `Verificar` (el código del proyecto actual), `Subir` (a la placa el proyecto), `Nuevo` (proyecto y fichero, al mismo tiempo), `Abrir` (un proyecto), `Salvar` (el proyecto actual) y a la derecha *Monitor Serie*. Éste último se explica en breves.

Figura 2.2 Menú del IDE.

Dichas tareas también pueden realizarse mediante los menús `Archivo`, `Programa` y `Herramientas` . Desde `Archivo`, podemos acceder a otras opciones útiles como `Ejemplos` varios de diversa índole o `Guardar Como`.

Por otro lado, en el menú `Herramientas` se puede especificar la placa sobre la que se quiere actuar, seleccionando su `Puerto` asociado y el tipo de `Placa`. También se puede acceder a la información transmitida por el puerto serie, en modo texto, seleccionando *Monitor Serie*, o en modo gráfica, utilizando *Serial Plotter*. En la siguiente imagen se puede observar que la opción `Puerto` permanece deshabilitada cuando no detecta ninguna placa conectada.

En el programa se inicia el puerto serie a una tasa en baudios, normalmente 9600, y es posible mandar información para visualizar varias señales al mismo tiempo con el *Serial Plotter* o leer valores y cadenas de caracteres en el *Monitor Serie*. El monitor sirve a su vez de entrada y de salida por puerto serie, para ello su ventana dispone de entrada y salida de texto.

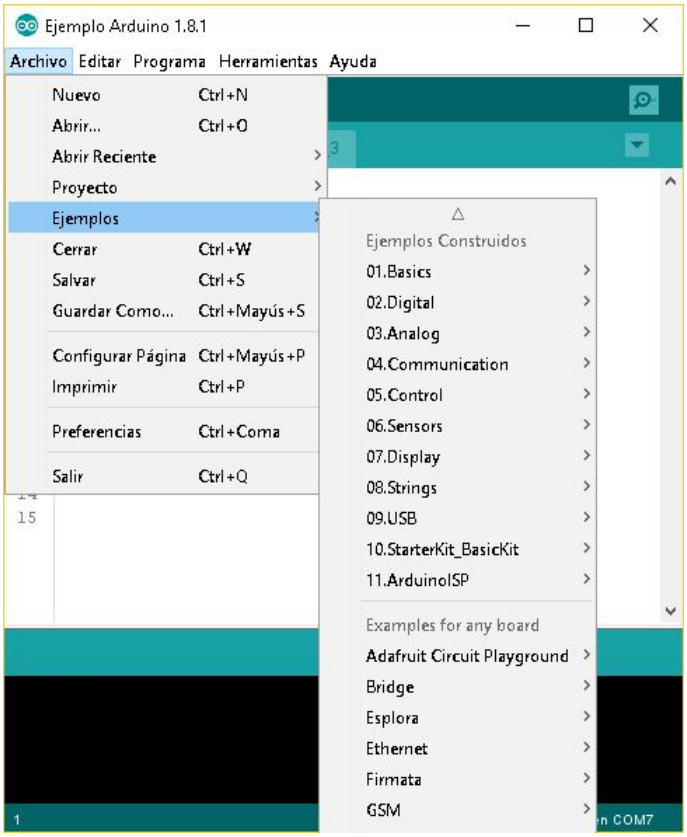

Figura 2.3 Ejemplos del IDE.

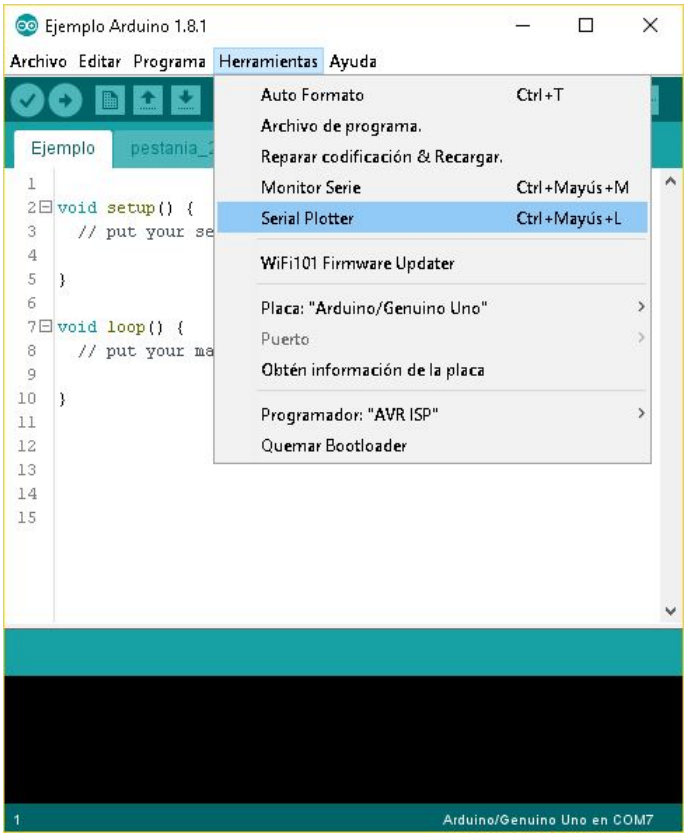

Figura 2.4 Opciones: `Placa`, *Serial Plotter* y *Monitor Serie*.

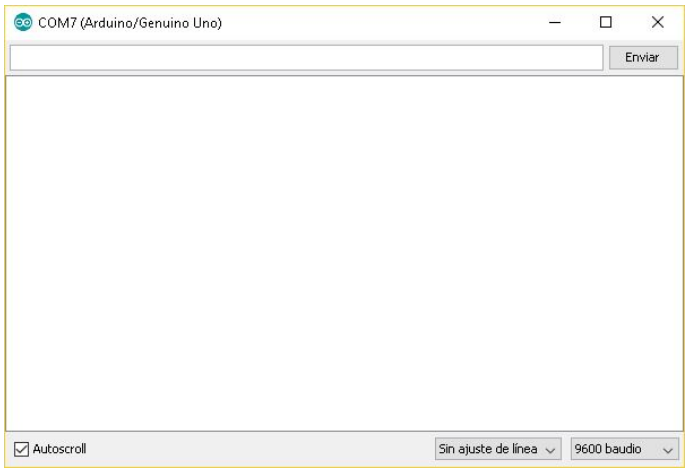

Figura 2.5 *Monitor Serie.*

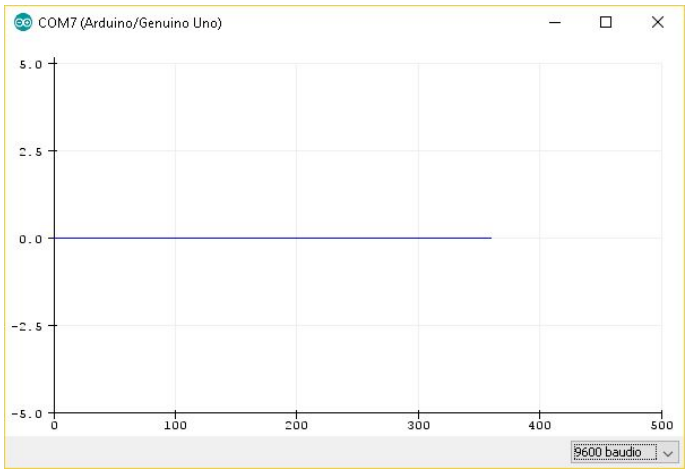

Figura 2.6 *Serial Plotter.*

Para usar el puerto serie, debe ser inicializado previamente, indicando la tasa a usar, con la función `Serial.begin()`. Una vez hecho, para imprimir un valor se utiliza una de las siguientes funciones:

Tabla 2.1 Funciones del puerto serie..

Funciones	Descripción
`Serial.print()`	Es la función básica para imprimir un valor. Además permite cambiar el formato de impresión a binario, ASCII o hexadecimal entre otros.
`Serial.println()`	Incluye un retorno de carro y salto de línea. También permite cambiar el formato de impresión.
`Serial.avaible()`	Devuelve el número de bytes disponibles para leer. El buffer serie puede almacenar como máximo 128 bytes.
`Serial.Read()`	Lee un byte del buffer serie. Devuelve -1 si estaba vacío. Para no eliminar el byte leído del buffer, usar en su lugar la función `peek()`.

Para representar más de una variable en el *Serial Plotter* es necesario imprimir una coma entrecomillada *(",")* separando un valor de otro. Ejemplo:

Código 2.1 Código para imprimir en el *Serial Plotter*.

```
Serial.print(variable1);
Serial.print(",");
Serial.println(variable2);
```

Como se comentó anteriormente, es posible dividir el fichero principal, y crear continuaciones del mismo usando la herramienta `Nueva Pestaña`, a la derecha del menú. La pestaña que contenga el primer fragmento del programa donde se encuentren las funciones principales y las constantes, debe mantener el nombre original.

Figura 2.7 Pestañas del IDE.

Tras éste breve repaso, ya se han presentado las herramientas más usadas del IDE. Desde este punto, comienza una breve descripción de la programación en *Arduino*.

2.3 El software

La estructura del lenguaje de programación de *Arduino* es bastante simple, y el lenguaje mezcla conceptos de Java con una base de C.

Todo proyecto cuenta con dos funciones principales, `setup()` y `loop()` que se definen en el fichero que da nombre al mismo. La primera, es la función de configuración y se ejecuta una sola vez, al iniciar el programa. La segunda, es la función bucle y se repite periódicamente. Por esto se suele utilizar setup() para inicializar procesos, como el puerto serie o los pines digitales.

Al margen de estas dos funciones, uno puede crear otras funciones auxiliares que le ayuden a realizar tareas. Además, si es necesario incluir librerías, declarar funciones, variables globales o definir constantes, esto se realiza al inicio del código.

2.3.1 Las interrupciones

En todo programa de gran extensión, es conveniente automatizar todas las tareas posibles para evitar dedicar un tiempo y capacidad innecesarios a las mismas. Con este propósito se usan las interrupciones.

Una interrupción es una parada temporal de la ejecución del programa principal, para ejecutar una subrutina, llamada *Rutina de Interrupción de Servicio*, o ISR, que requiere atención en ese instante. Una vez finalizada se reanuda al programa principal desde donde se dejó.

En general en las interrupciones se detecta un evento y se reacciona en consecuencia a través de una rutina. Estos eventos pueden producirse tanto por el cambio de valor de uno de los pines como por la actualización de un registro, y según esto se distinguen interrupciones externas e interrupciones temporales o *timers*, que se explican más adelante.

Gracias a este mecanismo se evita permanecer monitorizando continuamente un pin esperando que cambie de valor (a esto se le llama *Polling*), lo que se traduce en una pérdida de tiempo y se soluciona fácilmente utilizando una interrupción externa.

Por poner otro ejemplo, si es necesario tomar muestras cada cierto tiempo, lo más interesante es programar una interrupción temporal. Ya que al usar en su lugar la función de espera `delay()` el procesador permanece detenido durante ese tiempo, ignorando posibles eventos e incapaz de dedicarse a resolver otra tarea. Además esto produce un aumento del consumo de batería, y no garantiza una temporización exacta.

Ya que existen diversas fuentes de interrupción, si varias ocurren al mismo tiempo, el microprocesador las atiende según su prioridad. En la siguiente tabla, obtenida del *datasheet* del *Atmega328*, se puede consultar la prioridad de las diversas fuentes de interrupción.

Tabla 2.2 Prioridad de las interrupciones del *Atmega328*.

Vector Nº	Program ddress	Source	Interrupt Definition
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset and Watchdog System Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	PCINT0	Pin Change Interrupt Request 0
5	0x0008	PCINT1	Pin Change Interrupt Request 1
6	0x000A	PCINT2	Pin Change Interrupt Request 2
7	0x000C	WDT	Watchdog Time-out Interrupt
8	0x000E	TIMER2 COMPA	Timer/Counter2 Compare Match A
9	0x0010	TIMER2 COMPB	Timer/Counter2 Compare Match B
10	0x0012	TIMER2 OVF	Timer/Counter2 Overflow
11	0x0014	TIMER1 CAPT	Timer/Counter1 Capture Event
12	0x0016	TIMER1 COMPA	Timer/Counter1 Compare Match A
13	0x0018	TIMER1 COMPB	Timer/Coutner1 Compare Match B
14	0x001A	TIMER1 OVF	Timer/Counter1 Overflow
15	0x001C	TIMER0 COMPA	Timer/Counter0 Compare Match A
16	0x001E	TIMER0 COMPB	Timer/Counter0 Compare Match B
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI Serial Transfer Complete
19	0x0024	USART, RX	USART Rx Complete
20	0x0026	USART, UDRE	USART, Data Register Empty
21	0x0028	USART, TX	USART, Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator
25	0x0030	TWI	2-wire Serial Interface
26	0x0032	SPM READY	Store Program Memory Ready

En esta tabla se puede observar que en caso de coincidencia se atiende primero a las interrupciones externas (INTx y PCINTx), después a las temporizadas (TIMERx), seguidas de las interrupciones relacionadas con comunicaciones serie (UART, SPI y TWI) entre otras .

De forma resumida, el proceso de atención de una interrupción es el siguiente. Inicialmente se está ejecutando el programa principal hasta que de repente se detecta una interrupción. Tras esto, se completa la instrucción actual y la CPU salva parte del contexto actual, el Registro de Estado. A continuación se detecta la fuente de interrupción activa con mayor prioridad y se atiende su ISR, salvando previamente el resto del contexto. Esta se ejecuta, se restauran los registros, se atiende al resto de fuentes de interrupción si las hay, y

al finalizar se restaura el Registro de Estado para continuar con el programa principal.

Como se puede ir observando, mientras se atiende a una de las interrupciones puede saltar el aviso de otra y, mientras no se permita anidar interrupciones, la segunda permanece a la espera. Por este motivo, conviene que las interrupciones se atiendan de la forma más breve posible, para evitar desatender otras tareas que exijan gran sincronización.

Una vez entendido el concepto de interrupción, para seguir profundizando pueden clasificarse en dos tipos principalmente, interrupciones de hardware y temporizadores o *timers*.

Las interrupciones de hardware y las ISR

Las interrupciones de hardware responden a eventos ocurridos en ciertos pines. Dentro de las interrupciones de hardware se pueden distinguir interrupciones externas e interrupciones por cambio de pin. La placa *Arduino UNO* proporciona dos fuentes de interrupción externa, INT0 en el pin 2 e INT1 en el pin 3. El evento a detectar puede ser uno de los siguientes:

- `RISING`, ocurre en el flanco de subida de nivel bajo LOW a nivel alto HIGH.
- `FALLING`, ocurre en el flanco de bajada de nivel HIGH a LOW.
- `CHANGING`, ocurre cuando el pin cambia de estado. Es decir, se activa en ambos flancos.
- `LOW`, se ejecuta continuamente mientras el pin se encuentra a nivel bajo.

Por otro lado, aunque la familia de microprocesadores *ATmega328* implementa interrupciones por cambio de pin, *Arduino* no las tiene habilitadas por defecto y es necesario recurrir a la librería *PinChangeInt* para acceder a ellas, usando las mismas funciones que se comentan más adelante para las interrupciones externas.

Las ISR pueden crearse de dos formas distintas. Programándolas a bajo nivel o utilizando librerías diseñadas para cada tipo de interrupción. De forma general podemos establecer una ISR utilizando la macro `ISR()` e indicando el tipo de interrupción a usar. Es bastante usado para las interrupciones temporales, ya que sus librerías en ocasiones limitan ciertas funcionalidades.

Utilizando funciones de librería, las rutinas de interrupción externa se escriben como una función aislada y se habilitan con la función `attachInterrupt()` desde la que se indica el nombre de la función, el pin y el evento al que responde la ISR. Así mismo se puede inhabilitar la interrupción usando la función `detachInterrupt()`. Las interrupciones por cambio de pin se usan de la misma manera, junto a la librería que se ha comentado.

Al diseñar cualquier ISR es necesario tener en cuenta ciertos aspectos. Para empezar debe ser una rutina que no reciba ni devuelva nada. Conviene hacerla lo más corta posible, ya que durante su ejecución el programa principal se encuentra detenido, por eso lo más común es actualizar banderas y salir de la interrupción. Por ello se evitan procesos lentos como leer y escribir pines, o usar el puerto serie y se dejan para el programa principal.

También hay que tener en cuenta que las variables que se utilicen en la rutina deben ser del tipo `volatile`. Esto indica al compilador que la variable tiene que ser consultada siempre antes de ser usada, dado que puede haber sido modificada de forma ajena al flujo normal del programa (lo que, precisamente, hace una interrupción). Aun así conviene declarar como `volatile` las variables estrictamente necesarias ya que es un tipo poco eficiente.

Por último pero no menos importante, dentro de la ISR no pueden ejecutarse otras funciones que hagan uso de interrupciones como `millis()` o `delay()` , ya que por defecto no se permiten interrupciones anidadas. Aun así es posible leer su último valor, que permanecerá constante dentro de la rutina.

Las interrupciones temporizadas. Configuración, ISR y librería

Cambiando de cuestión, *Arduino UNO* cuenta con tres fuentes de interrupciones temporales, o *timers*, que son el *timer0*, *timer1* y *timer2*.

Un *timer* o temporizador actúa incrementando un registro llamado registro contador (*counter register*) construido sobre el propio hardware del microcontrolador, que puede ser programado a través de ciertos registros. Las interrupciones se dan cuando dicho registro desborda o se hace cero, por ello los *timers* con registros mayores permiten hacer temporizaciones más largas.

En este proyecto es muy útil el uso de temporizadores ya que son la solución ideal para tomar o enviar muestras de forma periódica automatizando la tarea para dedicar el programa principal al procesamiento de dichas muestras.

La placa de *Arduino UNO* incorpora tres *timers*, como ya se ha comentado, los *timers 0* y *2* cuentan con registros de 8 bits, mientras que el *timer1* usa un registro de 16 bits y por ello permite hacer temporizaciones mayores que el resto y será el escogido en este trabajo.

Cada *timer* es utilizado a su vez por otras funciones por lo que conviene tener cuidado a la hora de usar cada temporizador. El *timer0* es usado por las funciones `delay()`, `millis()` y `micros()`. El *timer1* lo usa la librería *servo* y el *timer2* la función `tone()`. Los *timers* también se utilizan para generar señales PWM, por lo cual no conviene usar temporizaciones y PWM al mismo tiempo con el mismo *timer*. Para evitarlo, a continuación se presenta la relación de pines de PWM y *timers* asociados.

Tabla 2.3 Pines PWM usados por cada *timer*.

Timers	**Pines PWM**
Timer0	6 y 5
Timer1	9 y 10
Timer2	11 y 3

Para empezar, las interrupciones por temporización pueden configurarse según distintos modos de operación. La elección del modo, se realiza modificando el registro `TIMSKx`, para el *timer x*. Existen los siguientes modos:

Timer Overflow: Ocurre por desbordamiento del registro. Se inicializa el registro manualmente tras cada interrupción, y con el paso del tiempo el registro se va incrementando hasta desbordar. Para este caso, la fórmula anterior se adaptaría cambiando el registro de referencia por el valor máximo del registro $(2^{numerodebits} - 1)$ menos el valor de referencia. Se activa usandoel bit `TOIEx`, para el *timer1* se usaría `TIMSK1 |= (1 << TOIE1);`.

Output Compare Match: Este tipo de interrupción se lleva a cabo incrementando un registro hasta alcanzar el mismo valor que la referencia. De este modo, no es necesario inicializar manualmente (es decir, en la propia interrupción) el valor del registro, lo que hace el trabajo más sencillo. Se activa usando el bit `OCIE1A/B`. Para el *timer1* se usaría por ejemplo `TIMSK1 |= (1 << OCIE1A);`.

Ya que en éste proyecto se escoge éste tipo de interrupción temporal, se va a comentar un poco más en profundidad. El valor de referencia se almacena en uno de los Registros *Output Compare* , `OCR1A/B`. De modo que se puede utilizar la interrupción con el registro `OCR1A` o el `OCR1B` , según hayamos seleccionado al activarla en el registro `TIMSK1`. En adelante se usará únicamente el registro `OCR1A`.

Timer Input Capture: Este modo de interrupción salta cada vez que recibe un cambio en su pin asociado. Se diferencia de una interrupción externa en que el modo *Input Capture* mide además el tiempo transcurrido y lo guarda en un registro, lo cual puede ser bastante útil para hacer posteriores cálculos.

Por otro lado, el periodo de la temporización, depende en conjunto del *timer* elegido, la frecuencia de reloj (16 MHz) y el preescalador que se aplique. Éste se usa para conseguir temporizaciones más largas de las que permite el reloj y se configura modificando los bits `CSx2`, `CSx1` y `CSx0` del registro `TCCRxB`, siendo x el número del *timer* a usar.

Los preescaladores disponibles son 1,8, 64,256 y 1024 y se configuran para el *timer1* a través de los bits `CS12`, `CS11` y `CS10` de la siguiente forma:

A su vez, para usar el tipo de interrupción *Output Compare*, será necesario modificar el registro `OCRxA` donde se guarda el valor del *timer* que usará como referencia de tiempo. Dependiendo del tipo de interrupción que se use, la temporización puede realizarse, entre otras opciones, disminuyendo el valor de este registro hasta alcanzar el cero o bien aumentando la cuenta desde cero hasta alcanzar dicho valor. Este registro puede tomar un rango de valores u otro según el número de bits del *timer* en cuestión, por ejemplo, el registro del *timer1* `OCR1A` puede tomar valores de 0 a $2^{16bits} - 1$.

De modo que el cálculo del periodo para el *timer1*, para el modo *Output Compare*, se realiza de la siguiente forma:

$$T = \frac{1}{1610^6 Hz} OCR1A preescalador$$

Concretamente, para un preescalador de 64 se pueden hacer temporizaciones de hasta 0.2621 segundos.

Tabla 2.4 Bits CSxy con los preescaladores.

CS12	CS11	CS10	DESCRIPCIÓN
0	0	0	Timer detenido
0	0	1	Preescalador vale 1
0	1	0	Preescalador vale 8
0	1	1	Preescalador vale 64
1	0	0	Preescalador vale 256
1	0	1	Preescalador vale 1024
1	1	0	Se usa un reloj externo en el pin T1 activo por flanco de bajada.
1	1	1	Se usa un reloj externo en el pin T1 activo por flanco de subida.

$$T = \frac{1}{1610^6 Hz}(2^{16} - 1)64 = 0.2621s$$

Las interrupciones temporales se pueden atender usando la macro `ISR()` o funciones de librería. Actualmente la librería *TimerOne* para controlar el *timer1* está probada y funciona adecuadamente, pero para otros *timers* las librerías son demasiado recientes y dan algunos fallos, por lo que en esta sección se va a explicar el uso de ISR de las dos formas distintas, comenzando con la macro `ISR()`.

La macro es la forma más general de definir interrupciones, válida para los tres *timers*, y en ella se basan las librerías como *timerOne*. En éste trabajo, la configuración de los registros para la temporización se realiza al inicio del programa, en la función `setup\_timer()`, dentro de la función principal `setup()`. Su código es el siguiente:

Código 2.2 Código para configurar la interrupción temporal al inicio del programa.

```
void setup_timer(){

  noInterrupts();

  // Se inicializan los registros de control
  TCCR1A = 0;
  TCCR1B = 0;
  TCNT1  = 0;

  // Se define el periodo de interrupción
  OCR1A = OCR1A_50;

  // Modo output compare, preescalador de 64 y activación del timer
  TCCR1B |= (1 << WGM12);
  TCCR1B |= (1 << CS10);
  TCCR1B |= (1 << CS11);
  TIMSK1 |= (1 << OCIE1A);

  interrupts();
}
```

En éste código se utiliza un par de funciones que permite activar y desactivar todas las interrupciones externas o de temporización a la vez. Se trata de las funciones `Interrupts()` y `noInterrupts()`, que pueden ser útiles para modificar registros sensibles, o para detener toda la temporización.

Una vez visto esto, el código inicializa a cero los registros de control TCCR1A y TCCR1B para porteriormente modificar el segundo de ellos, estableciendo el prescalador en 64 y el modo en Output Compare.

Es importante prestar atención a la modificación del registro OCR1A, para definir el periodo de interrupción, como se comentó anteriormente. En este caso se utiliza el valor 12499, definido en la constante OCR1A_50, que corresponde a una temporización de 50 ms. Esta es la mínima que se ha conseguido implementar en los programas de este trabajo.

Posteriormente, se define la ISR, que debe ser una función muy corta y que solo modifique variables del tipo volatile. Como puede observarse, el modo de interrupción se define en la cabecera de la macro.

Código 2.3 Rutina de interrupción temporal.

```
ISR(TIMER1_COMPA_vect){
  flagINT = 1;
  numMuestras++;
}
```

En este caso, la ISR se utiliza únicamente para aumentar la cuenta de muestras tomadas y levantar la bandera flagINT, que avisa al programa principal de la llegada de una nueva interrupción.

Por otro lado, la otra opción disponible para definir la interrupción, es usar la librería *timerOne*. Ésta librería es bastante útil, y solo permite configurar interrupciones para el *timer1*. A continuación se pueden ver sus funciones.

Tabla 2.5 Funciones de la librería *timerOne*.

Funciones	Descripción
Timer1.initialize(Periodo)	Inicia timerOne, selecciona el periodo en µs.
Timer1.setPeriod(Periodo)	Cambia el periodo, en µs.
Timer1.start()	Inicia el timer con un nuevo periodo.
Timer1.stop()	Detiene el timer.
Timer1.restart()	Reactiva el timer desde un periodo nuevo.
Timer1.resume()	Reactiva el timer desde donde se quedó.
Timer1.pwm(pin, duty)	*-No se usa para temporizar.-*
Timer1.setPwmDuty(pin, duty)	*-No se usa para temporizar.-*
Timer1.disablePwm(pin)	*-No se usa para temporizar.-*
Timer1.attachInterrupt(función)	Ejecuta la función en cada interrupción.
Timer1.detachInterrupt()	Deshabilita la interrupción.

2.3.2 Configuración adicional

Es interesante conocer que se puede controlar el funcionamiento del convertidor analógico a digital (CAD), así como la generación de señal PWM, modificando los registros que determinan su funcionamiento.

En la información del *datasheet* se puede encontrar una configuración más avanzada de estos y otros registros para lograr un mayor rendimiento y ajustar mejor la placa a las necesidades del programa.

3 Los fundamentos de comunicaciones digitales

Para empezar a analizar la comunicación, es conveniente visualizar el sistema completo.

Desde el transmisor, se genera una secuencia de bits, y éstos se agrupan en conjuntos de k bits, representados por $\log_2 k$ símbolos. Cada símbolo se traduce en una forma de onda concreta, y se envía cada periodo de símbolo. Al atravesar el canal, la señal puede sufrir atenuaciones y distorsiones, pero el receptor debe estar preparado para corregirlas y detectar la información que se envió originalmente.

Al transformar una serie de símbolos en la forma de onda en cuestión decimos que la señal se modula. Así bien, existen diversas técnicas de modulación y en este proyecto se escoge la QPSK (*Modulación por desplazamiento de fase en cuadratura*). Se trata de una modulación por desplazamiento de fase, descendiente de la M-PSK, que se ha escogido por ser lineal, de implementación sencilla, y por su buena detección independientemente del nivel de señal recibido, al contrario que las modulaciones con detección de amplitud.

En ésta modulación las señales se construyen a partir de una base formada por dos funciones ortonormales, y una serie de amplitudes asociadas a cada componente de la base. Se entiende como constelación de la señal, la representación en el espacio vectorial de las componentes de energía de la señal asociadas a cada función de la base.

Las modulaciones por desplazamiento de fase se caracterizan por generar símbolos con la misma energía y distinto desfase, lo cual se ve representado en su constelación con símbolos equiespaciados del origen de coordenadas. La QPSK consta de cuatro símbolos que pueden inscribirse en una misma circunferencia formando un cuadrado.

3.1 El transmisor

Cada periodo de símbolo se genera una señal $s_i(t)$, combinación lineal de las funciones de la base ortonormal $\phi_1(t)$ y $\phi_2(t)$.

$$s_i(t) = \phi_1(t)s_{1i} + \phi_2(t)s_{2i}$$

Dichas funciones se definen en relación a la función generadora $g(t)$, de la cual se hablará más adelante.

$$\phi_1(t) = \sqrt{\frac{2}{E_g}}g(t)\cos w_c t$$

$$\phi_2(t) = \sqrt{\frac{2}{E_g}}g(t)\sin w_c t$$

Para generar la $s_i(t)$ se multiplica a la base por las amplitudes s_{1i} y s_{2i} que dependen del símbolo i transmitido.

$$s_{1i} = \cos\left[\frac{2\pi(i-1)}{4} + \frac{\pi}{4}\right] \quad i = 1,2,3,4$$

$$s_{2i} = \sin\left[\frac{2\pi(i-1)}{4} + \frac{\pi}{4}\right] \quad i = 1,2,3,4$$

De modo que tanto s_{1i} como s_{2i} toman los valores $\pm\frac{1}{\sqrt{2}}$, que se aproximan a ±0.707.

3.2 El canal

Por un lado, la señal modulada se transmite por un canal y sufre cierta distorsión. En éste proyecto el canal se puede considerar ideal por lo que se añade un ruido blanco gaussiano para estudiar la robustez de la modulación.

Por otro lado, es conveniente tener en cuenta que la señal a transmitir, en una situación más realista deberá ajustarse a un ancho de banda dado, y esto limita la forma del pulso $g(t)$para cumplir el criterio de Nyquist. Recordando que los pulsos de Nyquist $h(t)$ están limitados en banda y garantizan ISI nula, debe cumplirse la siguiente condición, donde T es el tiempo de símbolo.

$$h(n) = \begin{cases} 1 & \text{si } n = 0 \\ 0 & \text{eoc} \end{cases}$$

Dicha condición se aplica al pulso conformado por el filtro transmisor, el canal y el filtro receptor, y se escoge para representarlo la familia coseno alzado. Por ello se define $g(t)$ como el pulso raíz de coseno alzado.

En el trabajo, se genera la señal transmitida, calculando muestra a muestra en tiempo real, ya que no se disponen de funciones específicas que proporcionen la convolución de cada cadena de símbolos completa con su filtro. Por ello, y debido a que el pulso raíz de coseno alzado tiene una longitud equivalente a más de un símbolo, es importante recordar que la señal transmitida en cada periodo de símbolo, contiene contribuciones de los símbolos anteriores, además del símbolo actual.

Concretamente, se puede entender la señal como la suma de diversas copias de los filtros, retrasadas varios periodos de símbolo, y multiplicadas por la amplitudes de los últimos símbolos transmitidos. Esto se explica en detalle más adelante.

3.3 El receptor

En el receptor, se hace la correlación entre la señal modulada junto al ruido, y las funciones de la base $\phi_1(t)$ y $\phi_2(t)$, que actúan como el banco de correladores. De la operación se obtiene el vector de observación, que se usa posteriormente para tomar una decisión sobre el símbolo transmitido.

Al igual que en el transmisor, conviene desglosar un poco la señal para saber como demodularla. De la fórmula de la correlación, se entiende que para demodular la señal es necesario almacenar las muestras anteriores. Así, desde el momento en que se obtienen todas las muestras anteriores correspondientes a un símbolo concreto, es posible demodularlo. De esta forma se obtiene un símbolo nuevo cada periodo de símbolo, como es natural.

4 La práctica

El diseño de la pareja transmisor-receptor se centra fundamentalmente en la modulación y demodulación de señales para transmitir información. En torno a esta idea surgen otros aspectos que también conviene estudiar, como la necesidad de filtrado, el tipo de sincronización o la información que se transmite. En este sentido, se han desarrollado varios prototipos entre los cuales cabe destacar un par de ellos que se comentan más adelante, los ejemplos 1 y 2.

4.1 El diseño en la práctica

Para empezar, como se comenta en el apartado 1.2, la señal pseudo-analógica que transmite la placa de Arduino necesita ser filtrada para convertirla en una señal analógica aceptable. Este proceso puede llevarse a cabo de forma analógica o digital, pero tras analizar la situación, se decidió que es más conveniente filtrar de forma analógica para reducir la carga computacional y evitar interrupciones demasiado frecuentes, que puedan afectar al funcionamiento normal del programa. Por ello se optó por un filtro paso bajo RC, en el que se tuvo que recurrir a una solución de compromiso, con los valores R=220Ω y C=100µF para garantizar un funcionamiento aceptable respecto a su rizado y su tiempo de respuesta.

Por otro lado, en cada placa se ejecuta un programa basado en la transmisión o recepción de muestras, así que para trabajar a una velocidad constante, conviene hacer una temporización usando una interrupción que se active cuando sea necesario transmitir o recibir cada una de ellas. Pero a pesar de que el modulador y el demodulador trabajan con la misma temporización, no se puede garantizar una recepción correcta si no están sincronizados, ya que un pequeño desfase provoca errores importantes en la detección. Al menos si se usa una modulación coherente, como es el caso de la QPSK.

Además, una vez sincronizados los relojes, como se explica en los siguientes apartados, el receptor debe saber cuándo comenzar a demodular la señal que le llega, porque, como se explica más adelante, es necesario prescindir de los tres primeros símbolos que se demodulan. Así que para lograrlo, se desarrollan los dos ejemplos comentados anteriormente, que tratan este aspecto de formas ligéramente diferentes.

La solución del ejemplo 1, consiste en establecer slots de tiempo y transmitir cuando corresponda. Esta propuesta es considerablemente lenta e ineficiente para un único receptor, pero es interesante de analizar, ya que se parece a un sistema TDMA. Para empezar a contar el primer slot, se espera a que el transmisor termine de enviar la señal de sincronización de relojes y ponga la línea a cero, y se toma este instante como referencia en el transmisor y receptor.

La segunda de las propuestas, desarrollada en el ejemplo 2, se basa en transmitir la información de forma inmediata, avisando al receptor de la inminente transmisión a través de una cabecera o pulso de sincronización que precede al mensaje, y que el receptor detectará a través de la correlación.

Y respecto a la información transmitida, el primero de los ejemplos transmite un conjunto de bits de prueba que se traducen en símbolos, mientras que el segundo ejemplo es lgo más avanzado. En éste, se añade la funcionalidad de enviar caracteres por el puerto serie desde el entorno de Arduino para codificarlos de forma binaria y representarlos en un LCD en el receptor. De esta forma se aprovecha el programa para hacerlo más visual. Además en cada uno de los ejemplos es posible monitorizar las variables de interés en el programa, con Matlab o el IDE de Arduino, a tiempo real, lo cual también ha sido muy útil para la depuración de errores.

4.1.1 Las primeras pruebas de la sincronización

Durante las primeras pruebas, se intentó demostrar que era posible demodular una serie de símbolos. En esa fase del proyecto, los programas no estaban muy avanzados y era preferible no detenerse demasiado tiempo en la sincronización, por lo que se optó por prescindir de una sincronización inicial como tal, y reducirla a una simple detección del nivel de señal en la línea.

De modo que para avisar al receptor de una inminente transmisión, el transmisor se limitó a enviar un pulso a nivel alto antes de cada transmisión y el receptor lo detectaba cuando el nivel de la línea superaba un umbral. Como puede observarse, esta sincronización fue muy rudimentaria pero cumplió la función para la que fue diseñada: permitir probar la demodulación y corregir los fallos que pudieron darse al implementarla desde la teoría.

En la práctica, la detección de un nivel de señal, al igual que otros procedimientos similares como la detección de un flanco de subida o bajada, es poco conveniente para la sincronización en entornos expuestos a ruido y de alta precisión. Aun así, puede ser útil en comunicaciones cableadas de poca distancia, como es el caso de I2C.

4.1.2 La sincronización Early-Late Gate

Más adelante se decidió retomar el desarrollo de la sincronización, y se usaron como referencia las propuestas de los desarrolladores de Matlab, y la detección de los códigos PRN de los satélites de GPS basada en correlación.

A partir de estas ideas se decidió implementar el método de sincronización *Early-Late Gate*, dada su sencillez y efectividad. Además, gracias al uso de la correlación, se trata de un método muy robusto frente al ruido. Dicho método, se ejecuta calculando la correlación entre la señal ruidosa recibida y la señal original para distintos desfases. Si el máximo de la correlación se encuentra en el desfase nulo, las señales se encuentran alineadas.

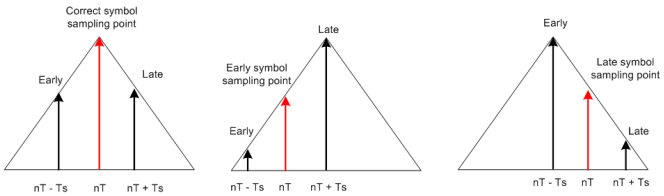

Figura 4.1 Sincronización *Early-Late Gate*.

Para comprobar dónde se da el máximo de la correlación, se calcula para un desfase positivo y para uno negativo, y al comparar resultados se puede saber si el máximo se encuentra adelantado o atrasado respecto al centro con desfase nulo. A continuación el código correspondiente.

Código 4.1 Código para realizar la sincronización *Early-Late Gate*.

```
void corrigeDesfase(){

  corr1 = 0;
  corr2 = 0;
  corr = 0;

  for(int i=0; i<3*L/4; i++){
    corr1 += bufferL[i+L/4]*sincro[i];
    corr2 += bufferL[i]*sincro[i+L/4];
  }
  diferencia = corr2-corr1;

  // si las muestras estan retrasadas respecto al filtro
  if(diferencia < 0){
    if(( - diferencia) > MARGEN)
      // se retrasa el indice que recorre el filtro
```

```
        numMuestras--;
    else{
      for(int i=0; i<L; i++)
        corr += bufferL[i]*sincro[i];
      if(corr < MARGEN*3)
        // si se ha confundido con el desfase PI, se desplaza el índice
        numMuestras += L/2;
      else{
        // fin sincronización
        flagFASE = 2;
        corrMax = corr;
      }
    }

  // si las muestras estan adelantadas respecto al filtro
  }else{
    if(diferencia > MARGEN)
      // se adelanta el indice que recorre el filtro
      numMuestras++;
    else{
      for(int i=0; i<L; i++)
        corr += bufferL[i]*sincro[i];
      if(corr < MARGEN*3)
        // si se ha confundido con el desfase PI, se desplaza el índice
        numMuestras += L/2;
      else{
        flagFASE = 2;
        corrMax = corr;
      }
    }
  }
}
```

De esta manera se ajusta poco a poco el índice de muestras para sincronizarse con el índice del trasmisor y obtener un desfase prácticamente nulo. En esta tarea hay que llegar a una solución de compromiso, porque si se acepta que el desfase es aproximadamente nulo con un margen grande, el método converge rápido pero se comete un error mayor, que afecta a la detección posterior. Por el contrario, un margen muy pequeño produce mejores resultados pero la sincronización puede no llegar a converger.

Además, se detectó un problema en el algoritmo, que se ha corregido desfasando el índice $L/2$ posiciones, como se muestra en el código. Se trata de que cuando dos señales se encuentran desfasadas 180º, el algoritmo lo confunde con el desfase nulo, ya que la correlación de la muestra anterior y la posterior, también coincide. Así que para corregirlo, se detecta el caso erroneo, sabiendo que su correlación es nula. Para evitar el estancamiento, cuando se ha cometido ese error, se desplaza bruscamente la señal, medio periodo, para que retome la sincronización. Por ello, como efecto secundario, al inicio de la misma, mientras no recibe señal, el receptor detecta continuamente correlación nula e imprime un pulso a la mitad, hasta que comienza a recibir señal, y se sincroniza sin problemas.

Este caso, realmente solo se produce cuando se inician los dispositivos con un desfase muy cercano a 180º, dentro del margen de error que se le ha puesto. En el resto de casos, la sincronización se hace de forma realmente rápida y siempre en la dirección más corta. Pero, si por alguna razón, el efecto de los pulsos a la mitad que se imprimen en el visualizador, fuera desagradable, es posible corregirlo sin muchos problemas. Para ello solo sería necesario iniciar la sincronización, justo cuando se detecte que la correlación deje de ser nula, lo que indicaría que ya se está recibiendo señal.

4.2 Las pruebas en Matlab

Para modular y demodular la señal en la placa transmisora, cabría esperar que el procedimiento fuese similar al que se realiza en una simulación de Matlab. Pero en este caso hay que tener en cuenta una serie de

cuestiones. Primero, que no se dispone de funciones para hacer la convolución ni generar los filtros raiz de coseno alzado en Arduino. Segundo, que el cálculo de de la modulación y demodulación de la señal depende de las contribuciones de los símbolos anteriores, porque se trabaja con un coseno alzado que abarca concretamente 4 símbolos por cada periodo del filtro. Y tercero, que en el caso del ejemplo 2, la señal no se genera al completo antes de transmitir porque se va leyendo carácter a carácter a tiempo real para evitar almacenar una señal demasiado grande, ya que el transmisor permite transmitir cadenas de cualquier tamaño entre 1 y 255, como se comenta más adelante.

De modo que para llevar a cabo las pruebas se ha recurrido a Matlab, y se ha representado la comparación entre la señal modulada en el transmisor siguiendo el procedimiento tradicional de simulación en este programa, frente a la señal que se genera en Arduino, para demostrar que son intercambiables.

4.2.1 La modulación en el transmisor

Para empezar, se generan los filtros de QPSK con raíz de coseno alzado, usando el siguiente código, en el que los filtros $\phi_1(t)$ y $\phi_2(t)$ se representan en Matlab como $g1$ y $g2$.

Código 4.2 Cálculo de los filtros transmisores.

```
L = 10;
beta = 1;
wc = 10/L;
N=4;

rrc = rcosdesign(beta,N,L);
E=sum(rrc.*rrc);

n = (-(length(rrc)-1)/2:(length(rrc)-1)/2);
g1 = sqrt(2/E)*rrc.*cos(wc*n);
g2 = -sqrt(2/E)*rrc.*sin(wc*n);
```

Cada símbolo se representa con L muestras, y el filtro se genera con una longitud equivalente a cuatro periodos de símbolo, como se detalla más adelante. Para elegir el número de muestras L, hay que llegar a una solución de compromiso, porque con un mayor número de muestras se puede conseguir mayor precisión en la posterior detección, pero esto relentizaría la transmisión considerablemente.

A continuación se genera la señal modulada de forma convencional, utilizando las siguientes amplitudes para la constelación: $[0.707, -0.707, 0.707, 0.707, -0.707, 0.707, -0.707, -0.707]$. El factor 0.707 es una aproximación de $\frac{1}{\sqrt{2}}$, que corresponde al módulo de la componente de señal en cada base en el espacio vectorial de la QPSK, para energía unidad.

Código 4.3 Generación de la señal completa utilizando las funciones de Matlab.

```
simbolos1 = reshape(kron([1 zeros(1,L-1)]',0.707*[1,1,-1,-1]),1,4*L);
senal1 = conv(g1,simbolos1);
simbolos2 = reshape(kron([1 zeros(1,L-1)]',0.707*[-1,1,1,-1]),1,4*L);
senal2 = conv(g2,simbolos2);
senalQPsk = senal1 +senal2;
```

Puede observarse que la señal se genera por completo en unas líneas de código. A diferencia de esto, en la placa se va a generar la señal muestra a muestra de forma pausada. Para explicar el procedimiento, se va a simular una serie de iteraciones que equivale a la generación de cada una de las muestras en la placa a ritmo de reloj. Como ya se comentó, la señal se genera a partir de las contribuciones de los símbolos anteriores. Concretamente, la señal es la suma de la componente en fase y en cuadratura, y cada cual se genera con un filtro diferente a los que se ha denominado filtro 1 y filtro 2.

Cada uno de ellos es un filtro de tipo raíz de coseno alzado con una longitud equivalente a cuatro periodos de símbolo. Se ha escogido modular cuatro símbolos por cada periodo del filtro porque el filtro de coseno alzado converge bastante bien a cero una vez trascurrido cuatro conjuntos de L muestras, eliminándose las colas. Cada componente de señal está formada por la superposición de cuatro versiones desplazadas del filtro,

moduladas por los símbolos anteriores correspondientes. Este procedimiento se lleva a cabo en el siguiente código, que se ha adaptado de Arduino a Matlab, para la misma secuencia de amplitudes de la constelación:

Código 4.4 Generación de la señal completa utilizando las funciones de Matlab.

```
filtro1 = g1(1:length(g1)-1);
filtro2 = g2(1:length(g2)-1);

simbolo1=[0,0,0, 0.707*[1,1,-1,-1], 0,0,0];
simbolo2=[0,0,0, 0.707*[-1,1,1,-1], 0,0,0];
senalQPsk = [];

L = 10;              % equivalente al tiempo de muestreo
N = 4;               % numero de muestras
M = 3+ N +3;         % tamaño del vector de símbolos

for numSimbolos=N:M   % 4+3 valores
    for numMuestras=1:L
        senial = filtro1(numMuestras)*simbolo1(numSimbolos) +
            filtro2(numMuestras)*simbolo2(numSimbolos) ...
              + filtro1(numMuestras+L)*simbolo1(numSimbolos-1) +
              filtro2(numMuestras+L)*simbolo2(numSimbolos-1) ...
              + filtro1(numMuestras+2*L)*simbolo1(numSimbolos-2) +
              filtro2(numMuestras+2*L)*simbolo2(numSimbolos-2) ...
              + filtro1(numMuestras+3*L)*simbolo1(numSimbolos-3) +
              filtro2(numMuestras+3*L)*simbolo2(numSimbolos-3);

        senalQPsk = [senalQPsk senial];

    end
end
```

Para empezar, se definen los índices o contadores `numSimbolos` y `numMuestras`. El primero recorre cada vector de símbolos como se explica posteriormente. El segundo índice recorre las L muestras que corresponden a cada símbolo.

Como puede apreciarse, se recorta la última muestra de las funciones de la base ortonormal para ajustar el tamaño del filtro a $4L$ muestras.

En las primeras líneas, se define el vector de símbolos asociado a cada componente de señal, que consiste en las cuatro componentes del vector de observación a transmitir, precedidos y sucedidos de ceros. Los primeros 3 ceros se utilizan para acceder a las posiciones `simboloX(numSimbolos-i)` de los primeros símbolos. Estas son nulas porque corresponden a los símbolos anteriores al inicio de la transmisión.

En definitiva, el resultado queda plasmado en la siguiente gráfica:

Por último, como puede observarse, la señal tiene componentes positivas y negativas, pero para transmitir de una placa a otra, el margen de tensiones es de 0 a 5 Voltios. Por este motivo, en el momento de transmitir y recibir cada muestra, se adapta el valor al rango correspondiente. En el caso del transmisor, la señal de salida pertenece al rango de 0 a 255, así que se genera de la siguiente forma, aproximando el valor máximo a transmitir, por 0.6, como se observa en la imagen anterior.

$$muestra_{0a255} = (muestra_{-0.6a0.6} + 0.6) * 255/1.2$$

Mientras que para recibir, se dispone de un rango de 0 a 1023, por lo que se hace la siguiente conversión:

$$muestra_{-0.6a0.6} = muestra_{0a1023} * 1.2/1023 - 0.6$$

4.2.2 La demodulación en el receptor

Así mismo, se ha simulado en Matlab el algoritmo de la demodulación que se ejecuta en la placa, frente a una simulación estándar en Matlab. Primeramente, la demodulación convencional en Matlab se realiza utilizando

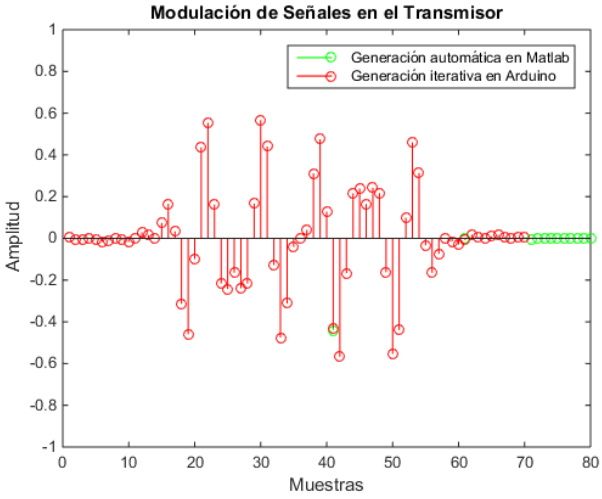

Figura 4.2 Comparación sobre la modulación de señales en el transmisor.

los filtros adaptados, $h1$ y $h2$, de la siguiente forma:

Código 4.5 Demodulación usando las funciones de Matlab.

```
h1 = fliplr(g1);  % se generan los filtros adaptados
h2 = fliplr(g2);

recibido1= conv(senalQPsk,h1);
recibido2= conv(senalQPsk,h2);

vObservacion1MAT = recibido1(4*L+1:L:length(recibido1)-4*L)
vObservacion2MAT = recibido2(4*L+1:L:length(recibido2)-4*L)
```

A continuación, se desarrolla la demodulación muestra a muestra, según como se ejecuta en la placa. Para empezar, es conveniente recordar cómo se crea la señal que se recibe. Cada símbolo se transmite durante $4L$ muestras, modulando cada uno de los dos filtros según las amplitudes de una constelación con codificación de gray. Además, cada L muestras se introduce un símbolo nuevo en la señal. Todo ello tiene ciertas implicaciones. Primero, que las primeras $3*L$ muestras solo contienen información incompleta sobre el primer símbolo, así que es necesario esperar hasta obtener las primeras $4*L$ muestras para comenzar a demodular los símbolos. Segundo, cada muestra interviene en la demodulación de 4 símbolos, así que es importante guardar las muestras anteriores y actualizarlas conforme se obtienen muestras nuevas. Tercero, lo más sencillo es demodular en grupos de L muestras, así que en el programa se cuentan las muestras trascurridas, para hacer la correlación cada vez que se alcancen L muestras y actualizar entonces los registros de muestras guardadas. Para entenderlo mejor, se puede consultar la definición de correlación. Como es conocido, es equivalente el uso de filtros adaptados, y de correladores, y al fin y al cabo el procedimiento es el mismo. Si se define la señal de muestras, como $s(n) = muestras(n)$ y el filtro adaptado i, como $f_{adaptado_i}(n) = filtro_i((4*L-1)-n)$, la señal recibida, sin contar el ruido, puede definirse como la suma de productos de las componentes del filtro y de la señal, recorridas en sentidos opuestos, como se observa a continuación.

$$\sum_{k=0}^{4L-1} muestras(n-k)filtro_i((4*L-1)-k) = muestras(n)*filtro_i(4*L-1)+muestras(n-1)*filtro_i((4*L-1)-1)+mue$$

Una vez entendido esto, se definen los vectores $x1$, $x2$ y $x3$ para almacenar las $3*L$ muestras más antiguas, donde su índice hace referencia a su antigüedad. De modo que el conjunto $[x3,x2,x1]$ equivale a las muestras $[muestras(n-(4*L-1)),\ldots,muestras(n-L)]$. Las muestras más recientes se almacenan en el vector $bufferL$, que recoge las L muestras más recientes, almacenadas como sigue: $[muestras(n-(L-$

1)),...,*muestras*(*n*)]. Se define este vector al margen del resto porque también se utiliza para otras tareas que utilizan las últimas L muestras recibidas, como la sincronización.

Todos estos vectores se han definido con las muestras en orden invertidas para facilitar el cálculo de la suma de productos, que gracias a ello, recorre los filtros y vectores en el mismo sentido. De este modo, el código, adaptado a Matlab, se reduce a lo siguiente:

Código 4.6 Código del demodulador simulado en Matlab.

```
filtro1 = g1(1:length(g1)-1);
filtro2 = g2(1:length(g2)-1);

Len = length(senalQPsk);

numMuestras = 0;
numSimbolos = 0;
r1 = 0;
r2 = 0;
vObservacion1ARD = [0 0 0 0];
vObservacion2ARD = [0 0 0 0];

bufferL = zeros(1,L);
x1 = zeros(1,L);
x2 = zeros(1,L);
x3 = zeros(1,L);

for i=1:Len-L

    muestra = senalQPsk(i);
    numMuestras = numMuestras +1;
    bufferL(numMuestras) = muestra;

    if (numMuestras == L)

        for k=1:L
            r1 = r1 + x1(k)*filtro1(k)...
                 + x2(k)*filtro1(L+k)...
                 + x3(k)*filtro1(2*L+k)...
                 + bufferL(k)*filtro1(3*L+k);

            r2 = r2 + x1(k)*filtro2(k)...
                 + x2(k)*filtro2(L+k)...
                 + x3(k)*filtro2(2*L+k)...
                 + bufferL(k)*filtro2(3*L+k);

            x1(k) = x2(k);
            x2(k) = x3(k);
            x3(k) = bufferL(k);

        end
        numSimbolos = numSimbolos +1;

        if numSimbolos > 3
            vObservacion1ARD(numSimbolos-3) = r1;
            vObservacion2ARD(numSimbolos-3) = r2;
        end
        r1 = 0;
```

```
        r2 = 0;
        numMuestras = 0;
    end
  end

vObservacion1ARD
vObservacion2ARD
```

Los resultados de la demodulación, pueden consultarse en las siguientes tablas:

Tabla 4.1 Vector de observación obtenido al demodular utilizando funciones de Matlab.

componente 1 (r1)	0.7103	0.7092	-0.7092	-0.7103
componente 2 (r2)	-0.7055	0.7056	0.7056	-0.7055

Tabla 4.2 Vector de observación obtenido demodulando con el código nuevo.

componente 1 (r1)	0.7086	0.7076	-0.7092	-0.7103
componente 2 (r2)	-0.7019	0.7092	0.7056	-0.7055

4.3 La detección del símbolo

Para realizar el test de hipótesis, que permite detectar el símbolo transmitido a partir del vector de observación, se establecen fronteras en la constelación, suponiendo los cuatro símbolos equiprobables. En el caso de la QPSK, el código en Arduino se reduce a la comprobación del signo de cada componente, como se aprecia en el siguiente código:

Código 4.7 Detección del símbolo..

```
void detectaSimbolos(){
    if(r1>0){
      if(r2>0)
        detectado = "11"; //s1
      else
        detectado = "10"; //s2
    }
    else{
      if(r2>0)
        detectado = "01"; //s4
      else
        detectado = "00"; //s3
    }
}
```

4.4 El ejemplo 1

En este ejemplo sencillo, se muestra una forma de transmitir en el canal cuando el tiempo se divide en slots para su uso en distintos sentidos (TDM) o por varios usuarios (TDMA). Se realizan cuatro transmisiones de la misma señal, con un ruido blanco diferente en cada caso.

4.4.1 El código

La cadena de bits a transmitir es la siguiente, y a partir de ella se genera la secuencia de amplitudes a transmitir en cada base, agrupando cada pareja de bits como un símbolo con dos componentes, como se observa a continuación:

Código 4.8 Generación de amplitudes de cada símbolo, en el ejemplo 1..

```
// se transmiten 4 símbolos de 2 bits
bool bits[2*N] = {1, 0, 1, 1, 0, 1, 0, 0};

// amplitudes de la primera componente de la base
float simbolo1[M]={0,0,0, (2*bits[0]-1)*0.707, (2*bits[2]-1)*0.707,
    (2*bits[4]-1)*0.707, (2*bits[6]-1)*0.707 , 0,0,0};

// amplitudes de la segunda componente de la base
float simbolo2[M]={0,0,0, (2*bits[1]-1)*0.707, (2*bits[3]-1)*0.707, (2*bits
    [5]-1)*0.707, (2*bits[7]-1)*0.707 , 0,0,0};
```

Al principio del programa, se lleva a cabo la sincronización del reloj del receptor con el del trasmisor. Después se espera a que el transmisor deje de transmitir sus pulsos de sincronización para empezar a contar el tiempo del primer slot. Y por último, una vez situados en el slot correspondiente, comienza la transmisión-recepción. De este modo se pueden distinguir en el transmisor 3 fases, que consisten en enviar suficientes pulsos de sincronización, esperar a que empiece el slot correspondiente, e iniciar la transmisión. Así el programa parte de la primera fase y avanza hasta que alcanza ala última, tras la cual vuelve a la segunda para esperar a la siguiente transmisión.

Por otro lado, en el receptor se distinguen cuatro fases. Se comienza por la sincronización con los pulsos del transmisor, a continuación se espera a que el transmisor termine de enviar pulsos, instante en el que comienza el primer slot en el que no se transmite y durante la tercera fase se espera a que empiece el slot correspondiente, para comenzar la recepción en la cuarta fase. Se hace una distinción en fases porque estas se pueden implementar de forma muy sencilla en el programa, aplicando el concepto de diagrama de bolas y cambio de estado. De esta forma el programa principal se reduce a una llamada al estado correspondiente, dentro del cual, existe posibilidad de actualizar el estado o fase siguiente usando la bandera flagFASE. En la práctica se implementa de la siguiente forma para el código del transmisor:

Código 4.9 Función principal del transmisor, en el ejemplo 1..

```
void loop() {

  if(flagINT == 1){
    flagINT = 0;
    switch(flagFASE) {
    // Fase de sincronización
    case 1: fase1();
      break;
    // Fase de espera al segundo Time Slot
    case 2: fase2();
      break;
    // Fase de transmisión
    case 3: fase3();
      break;
    }
  }
}
```

Como se observa, el inicio del nuevo estado ocurre únicamente cuando la bandera flagINT se encuentra levantada. Dicha bandera se activa en la interrupción que marca el tiempo de muestreo del sistema, como se

comenta en el apartado 2.3.1. Por último, cabe mencionar la fase de transmisión, que es interesante por la introducción del ruido, y por la distinción entre muestras por símbolo, símbolos por envío o slot, y número de envíos totales con diferentes ruidos antes de la detención del programa. Los ruidos fueron generados con Matlab para distintas SNRs. El código de la fase de transmisión es el siguiente:

Código 4.10 Fase 3, de transmisión, del ejemplo 1..

```
void fase3(){

  // se genera la muestra sin ruido
    muestra = filtro1[numMuestras-1]*simbolo1[numSimbolos] +
        filtro2[numMuestras-1]*simbolo2[numSimbolos]
            + filtro1[numMuestras-1 +L]*simbolo1[numSimbolos-1] +
            filtro2[numMuestras-1 +L]*simbolo2[numSimbolos-1]
            + filtro1[numMuestras-1 +2*L]*simbolo1[numSimbolos-2] +
            filtro2[numMuestras-1 +2*L]*simbolo2[numSimbolos-2]
            + filtro1[numMuestras-1 +3*L]*simbolo1[numSimbolos-3] +
            filtro2[numMuestras-1 +3*L]*simbolo2[numSimbolos-3];

  // se introduce el ruido blanco gausiano, con una SNR distinta
  // para cada uno de los cuatro envíos
    switch(cuentaEnvios){
      case 0: break;
      case 1:  muestra += ruido10[numMuestras-1]; // SNR = 10 dB
        break;
      case 2:  muestra += ruido7[numMuestras-1]; // SNR = 7 dB
        break;
      case 3:  muestra += ruido4[numMuestras-1]; // SNR = 4 dB
        break;
    }

    // se adapta la muestra del rango [-0.6, 0.6] a [0, 255]
    muestra = (muestra +0.6)*255/1.2;
    analogWrite(pinSalida, muestra);

    // si ha terminado el símbolo
    if(numMuestras == L){
      numMuestras = 0;
      numSimbolos++;

    // si ha terminado el envío de 4 símbolos
      if(numSimbolos == M){

        // se vuelve a la fase 2, para esperar al siguiente slot
        flagFASE = 2;
        OCR1A = OCR1A_sincro;
        cuentaEnvios++;

        // el programa transmisor se detiene tras los 4 envíos
        if(cuentaEnvios == 4)
          noInterrupts();

        numSimbolos = 3;

        // se pone la señal a cero
        muestra = 0.6*255/1.2;
```

```
        analogWrite(pinSalida, muestra);
      }
    }
}
```

El receptor utiliza un código similar, en el que se demodula cada símbolo una vez almacenadas L muestras del mismo, y se imprime su valor a partir del cuarto símbolo demodulado. Una vez explicado esto, es el momento de mostrar algunas capturas del funcionamiento completo y el montaje de este ejemplo.

4.4.2 Capturas sobre el funcionamiento

Para empezar, en la imagen 4.3, se puede observar el montaje, que es bastante sencillo y consta de las placas, el filtro montado con una resistencia y un condensador sobre una protoboard, y los cables que conectan las entradas y salidas de tensión. El transmisor envía la señal pseudo-analógica desde el pin 5, y tras ser filtrada, el receptor la lee desde el pin analógico A0. Se han conectadolos pines de tierra de ambas placas, y sirven como referencia para el filtro.

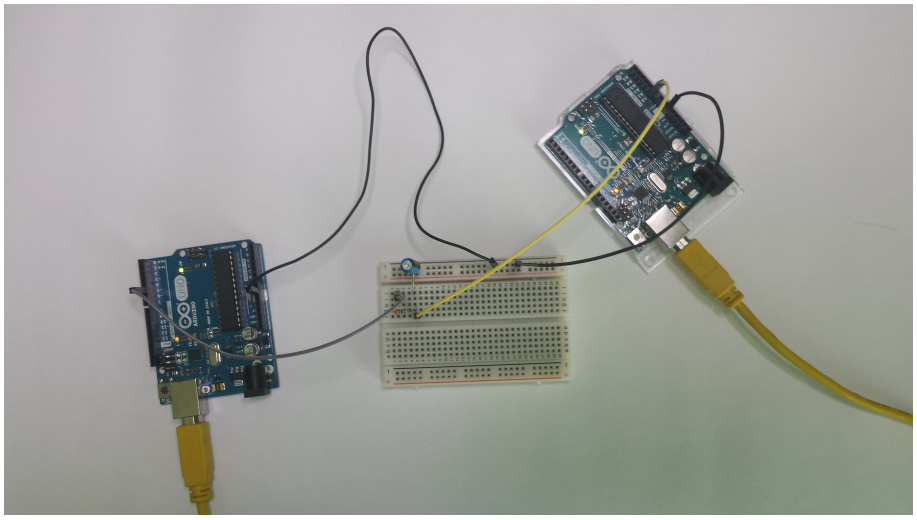

Figura 4.3 Imagen del montaje. De izquierda a derecha: Transmisor, filtro y receptor..

Para observar el transcurso de las señales en el tiempo, se han representado en el *Serial Plotter*, las variables más interesantes del receptor, como se observa en la imagen 4.4. En color verde se imprime la variable `flagINT` para poder saber en que fase se encuentra el sistema. En color rojo se representa la señal de sincronización del receptor, que tiene un periodo de L muestras, y por ello, resulta interesante representarla durante todo el proceso para utilizarla como señal de guía o de reloj, que marca la recogida de cada L muestras. Finalmente, en azul se representa la señal recibida del transmisor. De este modo, se representa el proceso de sincronización de las señales de sincronización que ocurre durante la fase 1, seguido de las fases 2 y 3, y el comienzo de la recepción en la fase 4. Esta fase se ve más detalladamente en otras imágenes.

En la imagen 4.5, se observan las dos primeras transmisiones, correspondientes a un ruido nulo y a uno correspondiente a 10 dB de SNR. A partir de las primeras $4L$ muestras detectadas, aparecen en pantalla otras dos nuevas señales. Éstas corresponden a las componentes del vector de observación, que toman valores que se aproximan a ± 0.707. A través de estas variables, se puede observar cómo afecta el ruido a la demodulación, en comparación con la primera transmisión, que es el caso ideal.

En la imagen 4.6, se representan las dos últimas transmisiones, que corresponden a ruidos de 7 y 4 dB de SNR respectivamente.

Además de la representación visual, también se han impreso los datos del vector de observación y de la detección de cada símbolo en el *Monitor Serie*, para conocer las cifras exactas en cada caso. La información sobre cada transmisión aparece separada por un salto de linea, comenzando por la primera, que es el caso idílico. Se puede ver que con un ruido de 4 de SNR, se deja de detectar el símbolo correctamente.

Una vez visto en acción cómo funciona el primero de los ejemplos, se presenta el segundo de ellos, que tiene un funcionamiento más interesante.

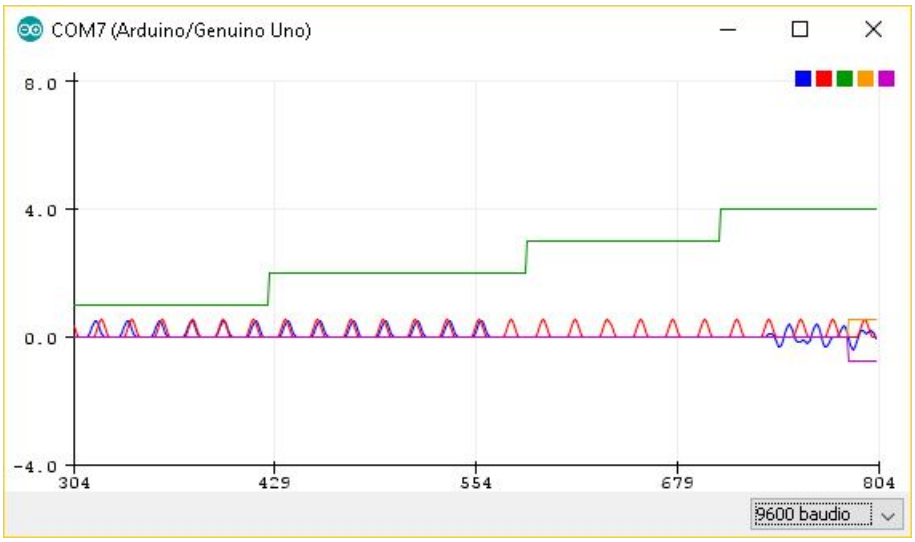

Figura 4.4 Fases 1,2,3 y 4 indicadas por la señal de color verde..

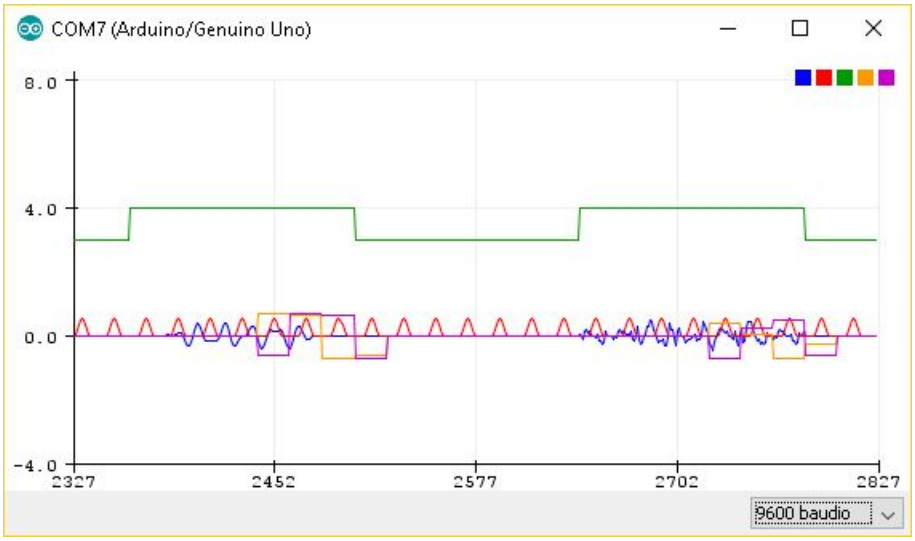

Figura 4.5 Fases 3 y 4. Primeras dos transmisiones..

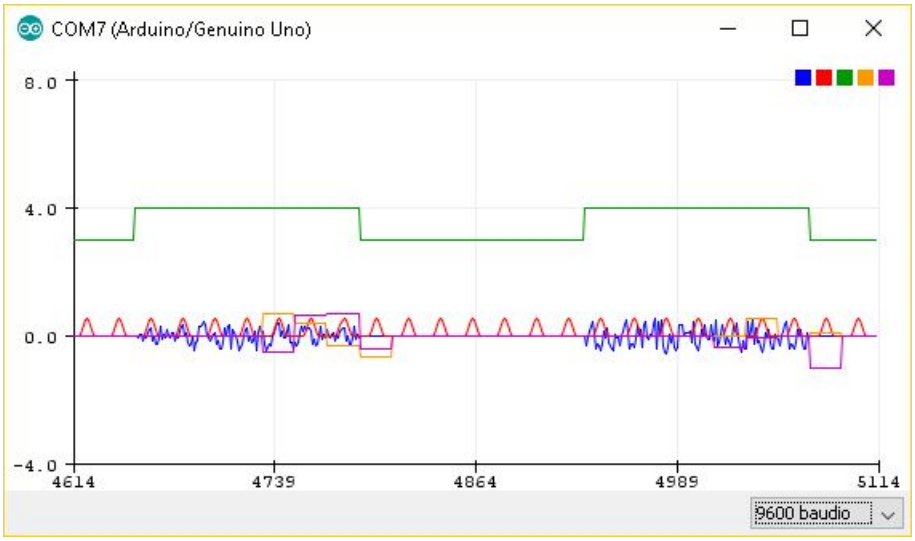

Figura 4.6 Fases 3 y 4. Últimas dos transmisiones..

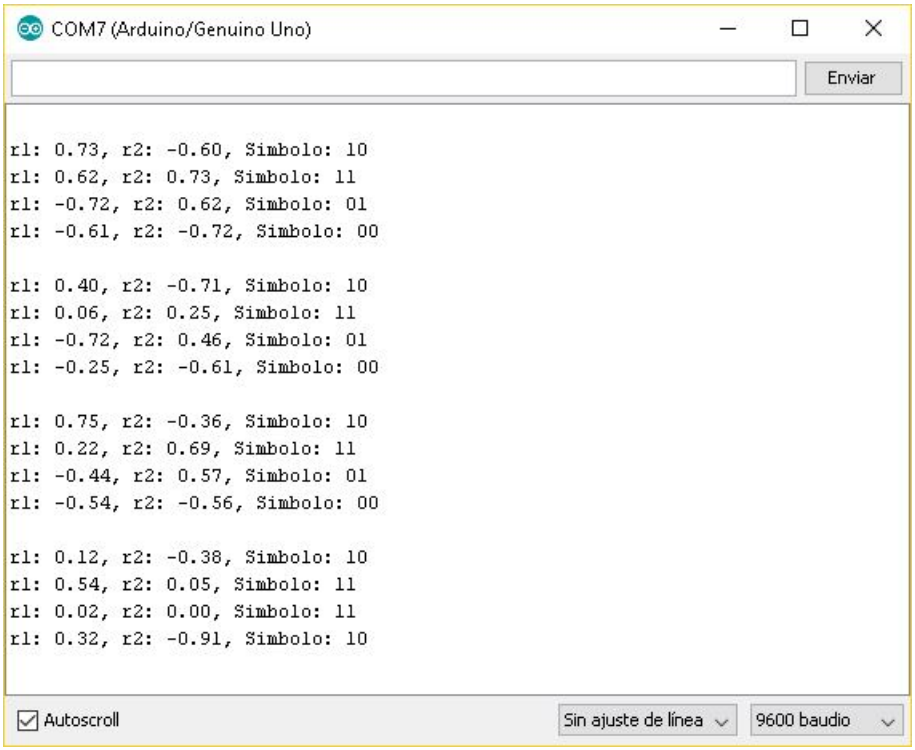

Figura 4.7 Vector de observación recibido..

4.5 El ejemplo 2

Este ejemplo muestra un uso más práctico de la transmisión entre placas, ya que incorpora un LCD para mostrar los datos recibidos, utiliza las cadenas binarias para transmitir información útil, como son los caracteres, y realiza la transmisión de forma inmediata, en cuanto recibe un dato que transmitir, sea cual sea su longitud. Todo ello lo hace un ejemplo más completo que el anterior, pero hay un aspecto en el que sigue siendo interesante haber explicado el ejemplo 1. Se trata de la incorporación de ruido, que en este segundo ejemplo no se ha tenido en cuenta, ya que la información contiene un primer campo más sensible, la longitud de la trama, que provocaría un funcionamiento indeseado en caso de demodularse incorrectamente.

4.5.1 El código

Se ha decidido transmitir directamente toda la cadena de información, carácter tras carácter, porque en caso de dividir en tramas de longitud fija, se perderían tres tiempos de símbolos por cada trama, debido a las tres primeras demodulaciones que no contienen información. De todas formas, ésta podría ser otra variante interesante, y como se puede observar, hay multitud de opciones posibles para realizar la transmisión. Esto da mucho juego a la hora de utilizar este trabajo como base para futuras modificaciones. Respecto al código para este ejemplo, las principales modificaciones frente al anterior son las siguientes. Primero, que la información que se transmite no es siempre la misma, sino que se obtiene de la lectura del Monitor Serie desde el puerto de la placa transmisora.

Segundo, la transmisión se puede iniciar al inicio de cualquier pulso de reloj. De modo que para informar al receptor previamente, se envía una cabecera que consiste en uno de los pulsos de sincronización. Se usa este pulso porque durante la sincronización se almacenó el valor estimado de la correlación entre los pulsos una vez sincronizados, y dicho valor se puede usar en este punto como referencia del valor esperado. Esto se observa más claramente en el siguiente fragmento de código:

Otro cambio que se ha hecho respecto al primer ejemplo, se encuentra en las fases en las que se organiza este programa. En lugar de esperar al slot correspondiente, se espera a la llegada de la cabecera. Para detectarla, se hace la correlación con la copia de la señal que se encuentra en el receptor, como se observa en el siguiente fragmento de código:

Código 4.11 Código para hacer la correlación de la cabecera.

```
void fase3(){

    tomaMuestra();

  // cuando se tienen L muestras, se hace la correlación entre estas
  // y la cabecera, que es el filtro de sincronización
    if(numMuestras == L){
      numMuestras = 0;
      for(int i=0; i<L; i++)
        corr += bufferL[i]*sincro[i];

      if((corr >(corrMax - 2*MARGEN)) && (corr <(corrMax + 2*MARGEN)) )
        flagFASE = 4;
    }
    corr = 0;
    imprimeMuestras();
}


void tomaMuestra(){

  // se recibe una muestra
  muestra = (float)analogRead(pinLectura);
  muestra = muestra*1.2/1023-0.6;

  // se almacena hasta completar L
  bufferL[numMuestras-1] = muestra;
}
```

Tercero, la longitud de los datos es desconocida a priori, así que ésta se transmite al principio de la trama usando 8 bits, que equivalen a 4 símbolos, al igual que el resto de caracteres. De modo que se admiten frases de hasta 255 caracteres.

Cuarto, ya que las tramas tienen una longitud variable, los vectores de símbolos se modifican brevemente, dividiendo la trama principal en distintos segmentos, correspondientes a cada Byte trasmitido, hasta llegar al último de ellos que se rellena con ceros para terminar de transmitir los símbolos incompletos, como se hace en la simulación en Matlab del apartado 4.2.1.

Para entender esto, puede ser interesante observar los siguientes fragmentos de código. Estos pertenecen a la función del transmisor, encargada de la lectura de cada nuevo carácter del buffer del Monitor Serie. El primer paso consiste en leer el tamaño de la cadena, del *Monitor Serie*, como se muestra a continuación.

Código 4.12 Lectura de la longitud de la trama, en binario.

```
// se lee el número de bytes disponibles en el buffer del Monitor Serie
Ncar = Serial.available();

// se guarda en una cadena binaria
cadena = String(Ncar,BIN);
while(cadena.length() < 8)
  cadena = '0' + cadena;
```

En el código se aprecia como la cadena se rellena de ceros hasta tener un tamaño de un byte, porque la función anterior no guarda en la cadena los ceros a la izquierda. A continuación se realiza el mapeo de bits a símbolos, y para ello se convierte la cadena a un vector de enteros previamente. Para entender bien cómo funciona este fragmento de código, es necesario tener en cuenta que la trama completa que va a transmitirse, debe descomponerse en tramas más pequeñas de 8 bits. Entonces, para transmitir la primera trama de 8 bits,

que corresponde a la longitud, se generan los vectores `simbolo1` y `simbolo2`, que están formados por 3 ceros iniciales, seguidas de las 4 amplitudes de los símbolos. De este modo, los siguientes vectores que se generen lo harán de forma similar, con la diferencia de que los 3 símbolos anteriores dejarán de ser cero, así que habrá que incluirlos en los vectores. Y para terminar, la última de las tramas completa la transmisión, y sus vectores contienen 3 ceros al final.

Código 4.13 Mapeo de bits a símbolos y cálculo de las amplitudes de las dos componentes de la constelación.

```
    // se convierte la cadena binaria a un vector de enteros
    for(int i=0; i<8; i++){
         if(cadena[i]=='0')
           bits[i]=0;
         else
           bits[i]=1;
     }

     // se mapean los bits a símbolos. Para ello, primero
     // guarda las amplitudes de los 3 símbolos anteriores,
     // y después añade las de los nuevos 4 símbolos
     for(int i=0; i<3; i++){
         simbolo1[i] = simbolo1[4+i];
         simbolo2[i] = simbolo2[4+i];
     }
     if(numCaracteres != Ncar+1){
        for(int i=0; i<4; i++){
          simbolo1[i+3] = (2*bits[2*i]-1)*0.707;
          simbolo2[i+3] = (2*bits[2*i+1]-1)*0.707;
        }
     // en la última trama se añade la cola de ceros
     }else{
        for(int i=0; i<4; i++){
          simbolo1[i+3] = 0;
          simbolo2[i+3] = 0;
        }
     }
  }
}
```

A continuación, para leer cada caracter se procede de forma similar, alterando simplemente lineas iniciales para que lean un byte en lugar del tamaño.

Código 4.14 Lectura de un caracter en binario.

```
// se extrae 1 byte del buffer del Monitor Serie
cadena = String(Serial.read(),BIN);

// se guarda en una cadena binaria
while(cadena.length() < 8)
  cadena = '0'+cadena;
```

Dejando a un lado el transmisor, es interesante comentar los cambios en el receptor. Una vez que se realiza la detección de los símbolos, el receptor los almacena en grupos de cuatro para decodificar posteriormente el carácter transmitido en cada caso, basándose en la codificación binaria de la tabla ASCII. El siguiente código muestra cómo se realiza el proceso, y para ello se ha incluído el código de representación en el LCD, pero ya que es un elemento más de visualización, y no es importante para entender el resto, se va a omitir la explicación de las funciones de su librería.

Código 4.15 Funciones del receptor del ejemplo 2, para formar cadenas de bits y procesarlas.

```
// esta función agrupa las parejas de bits detectadas en bytes
void agrupaBits(){

  // Se va generando la cadena binaria de 1 byte.
  cuentaBits +=2;
  bitsCaracter = String(bitsCaracter + detectado);

  // Una vez completo el byte, se distingue si representa
  // un caracter más o la longitud de trama.
  if(cuentaBits == 8){
    if(numSimbolos == 4+3)
      // a) Se actualiza la longitud de trama
      actualizaN();
    else
      // b) Se imprime el caracter en el LCD
      imprimeLCD();

    cuentaBits = 0;
    bitsCaracter = String("");
  }
}


// esta función obtiene el número de símbolos que tiene el mensaje
void actualizaN(){

  bitsCaracter.toCharArray(binary_string, 9);

  N = (strtol(binary_string,NULL, 2)+1)*4;

  Ntotal = N+3;
}


// esta función imprime un caracter en el LCD
void imprimeLCD(){

  // obtención del caracter
  bitsCaracter.toCharArray(binary_string, 9);
  caracter = (char)strtol(binary_string, NULL, 2);

  // impresión en el LCD
  indiceLCD++;
  if(indiceLCD < 16)
    lcd.write(caracter);

  else if(indiceLCD == 16){
    lcd.write(caracter);
    lcd.setCursor(0,rowL);
  }
  else if(indiceLCD <= 32){
    lcd.write(caracter);
    LCD_anterior[indiceLCD-17] = caracter;
  }
  else{
```

```
    indiceLCD = 17;
    lcd.clear();
    lcd.setCursor(0,rowH);
    lcd.print(LCD_anterior);
    lcd.setCursor(0,rowL);
    lcd.write(caracter);
    LCD_anterior[indiceLCD-17] = caracter;
  }
}
```

Una vez entendido esto, llega el momento de empezar a visualizar los resultados. Para realizar la monitorización de las señales en el receptor a la vez que se introducen datos en el Monitor Serie del transmisor ha sido necesario recurrir a un pequeño programa de Simulink. Esto se debe a que el IDE de Arduino no permite abrir el Monitor Serie o el Serial Plotter para más de un puerto simultáneamente. Por ello si se desea introducir datos para transmitir, no es posible visualizar el comportamiento del receptor al mismo tiempo.

Tampoco es posible abrir el Monitor para el transmisor, escribir en él, cerrarlo y abrir a continuación el Plotter del receptor, porque al comenzar la lectura del puerto serie de cualquiera de estas formas, el IDE de Arduino reinicia automáticamente la placa conectada a dicho puerto. De modo que la solución más efectiva es utilizar el Monitor Serie para escribir en el transmisor, y el visualizador de Simulink para monitorizar las señales del receptor. Para esto es necesario descargarse unas librerías para Simulink y Matlab, que permiten su conexión por puerto serie con la placa Arduino. El esquemático se muestra al final de la memoria. A continuación, pueden observarse algunas capturas que muestran el funcionamiento completo del ejemplo.

4.5.2 Capturas sobre el funcionamiento

Para empezar, el montaje se muestra en las últimas capturas del trabajo, y es similar al del ejemplo 1, con el añadido del LCD que se conecta a los pines del receptor, a través de la protoboard. La placa transmisora usa de nuevo el pin 5 como salida pseudo-analógica, y comparte la misma conexión de tierra que el resto. La placa receptora sigue utilizando el pin A0 como entrada analógica, y en esta ocasión, reserva los pines 7 al 12 para la conexión con el LCD. Esto se define al configurar sus funciones de librería. Para comenzar con las transmisiones, es necesario esperar a que se sincronicen las placas. El proceso tarda unos segundos y comienza en cuanto se reinician las placas. Durante el mismo, en la imagen 4.13 se observa que el receptor pasa de la fase 1, en la que se sincroniza con los pulsos, a la fase 2, donde espera a que el transmisor deje de enviar pulsos, y una vez en la fase 3, ya se encuentra operativo para comenzar a recibir. La fase se representa en celeste, la señal recibida, en amarillo, y los pulsos del receptor, se representan en rosa, a modo de reloj. Como detalle, al inicio de la linea temporal, se aprecia que los pulsos rosas del receptor son más estrechos. Esto afecta únicamente a la visualización, y se debe únicamente al efecto de los 180º, que se comenta durante la explicación del código.

Una vez sincronizados, cada vez que se desee transmitir alguna frase, se introduce la misma en el `Monitor Serie`, y se pulsa *Enter* o *Enviar*. Los datos introducidos se guardan en el buffer interno del `Monitor Serie`, y el transmisor los recoge durante la fase 3. Es decir, el transmisor consulta el buffer continuamente siempre que no esté transmitiendo y ya se haya sincronizado. Por esto, se pueden enviar frases por el `Monitor Serie`, aunque el transmisor esté ocupado con la sincronización o realizando una transmisión, y él ya consultará el buffer cuando acabe con estas tareas.

Cuando el transmisor, lee el buffer y detecta que hay contenido, se avanza inmediatamente a la fase de transmisión y se envía la cabecera, que es un pulso. Al mismo tiempo, conforme se generan tramas para enviar la longitud total y los caracteres, se imprime en el `Monitor Serie` la información más relevante. Concretamente, en la imagen 4.15 puede observarse que se imprime primero el elemento a transmitir, seguido de su codificación en binario, y a modo de curiosidad, se imprime el vector de amplitudes de cada símbolo, de la primera componente de la base. Es decir, se imprime el vector `simbolo1`, al que se presenta como $S1$, porque es interesante observar cómo se generan las tramas. En dicho vector, las tres primeras posiciones corresponden a copias de los vectores anteriores, como ya se ha explicado, para acceder a ellas durante la generación de muestras. Y las últimas cuatro posiciones corresponden a los cuatro símbolos que se generan por cada nuevo byte. La última trama, donde se genera la cola de ceros, se ha denominado "0".

En la imagen 4.16, se observa como el receptor entra en la fase 4, donde se comienza a recibir el mensaje. Una vez que han transcurrido 4 pulsos en esta fase, se sabe que hay $4L$ muestras recibidas y se va a demodular el primer símbolo. Como curiosidad, para distinguir cada transmisión, se ha impreso una señal en azul, que se pone a nivel alto cada vez que se termina de transmitir un caracter. Cada vez que se demodula un caracter, se

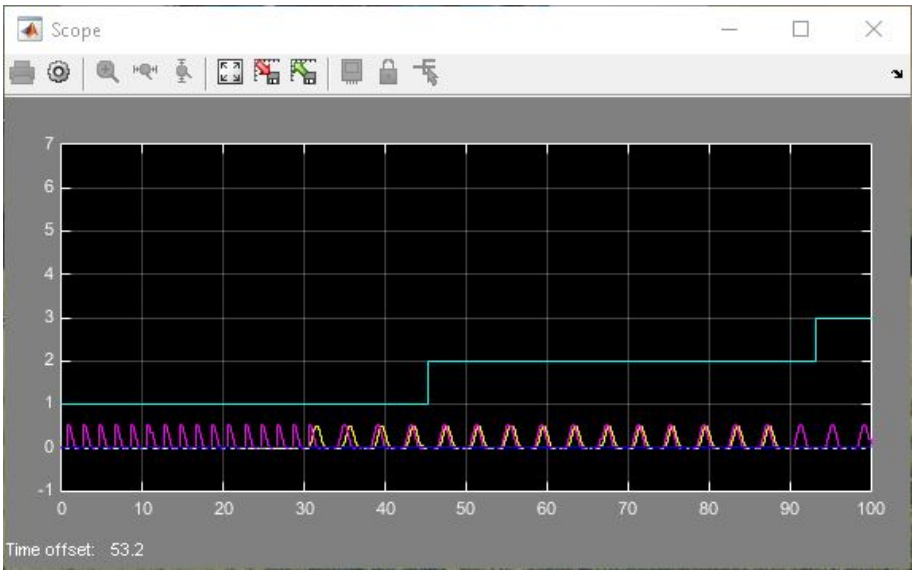

Figura 4.8 Fases 1,2 y 3 del ejemplo 2..

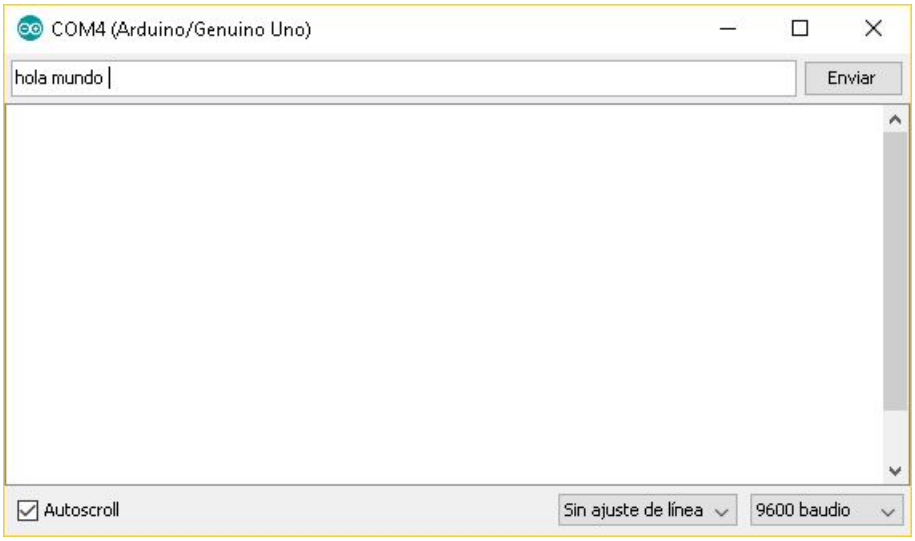

Figura 4.9 Transmisión desde el `Monitor Serie` (1)..

representa en el LCD. Una vez terminada la transmisión, en la imagen 4.17, se retrocede a fase 3, para dejar preparado al receptor ante nuevas transmisiones. Como se puede observar, existen miles de posiblidades para diseñar una comunicación entre microcontroladores, basada en el procesamiento de señales moduladas, y resulta bastante interesante utilizar estos dispositivos para hacer pruebas y demostraciones prácticas.

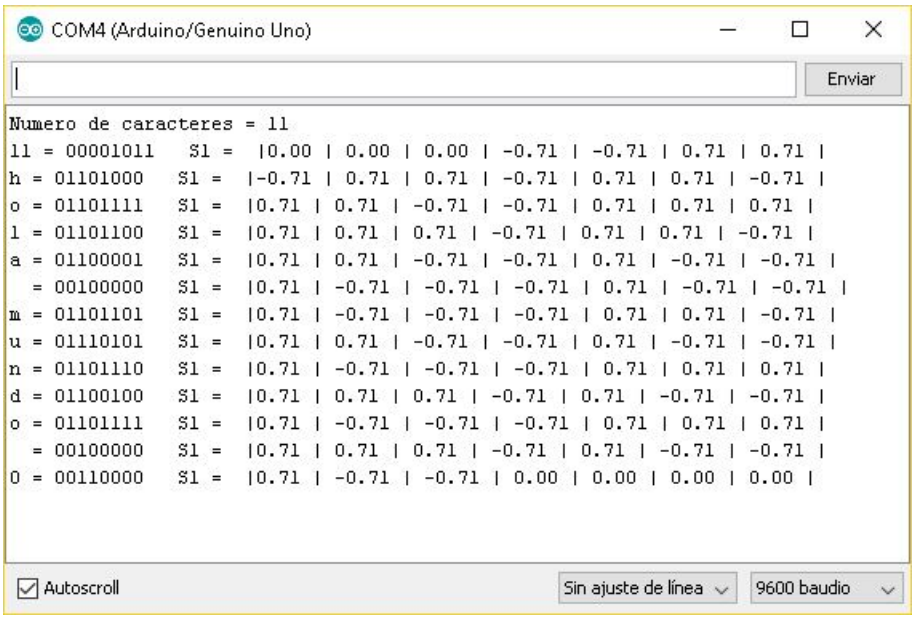

Figura 4.10 Transmisión desde el `Monitor Serie` (2)..

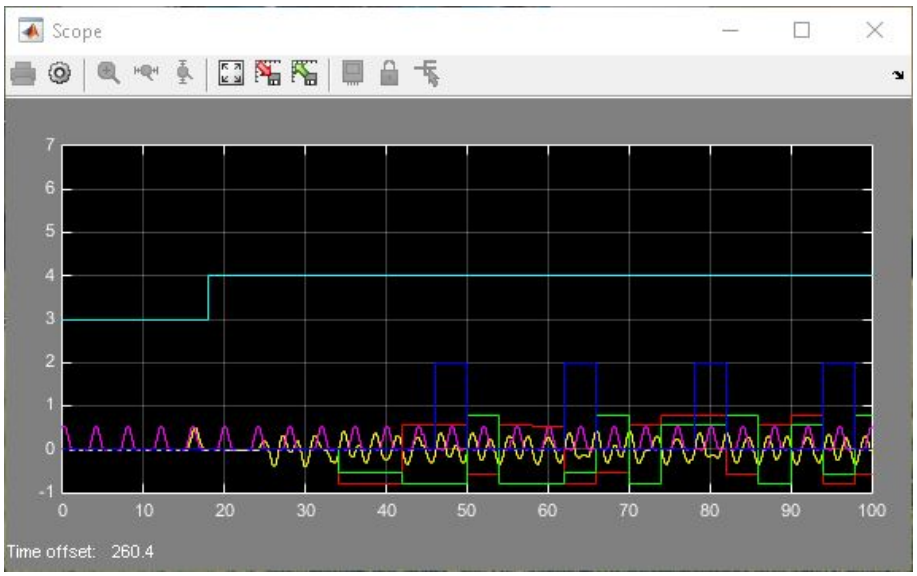

Figura 4.11 Fases 3 a 4 del ejemplo 2..

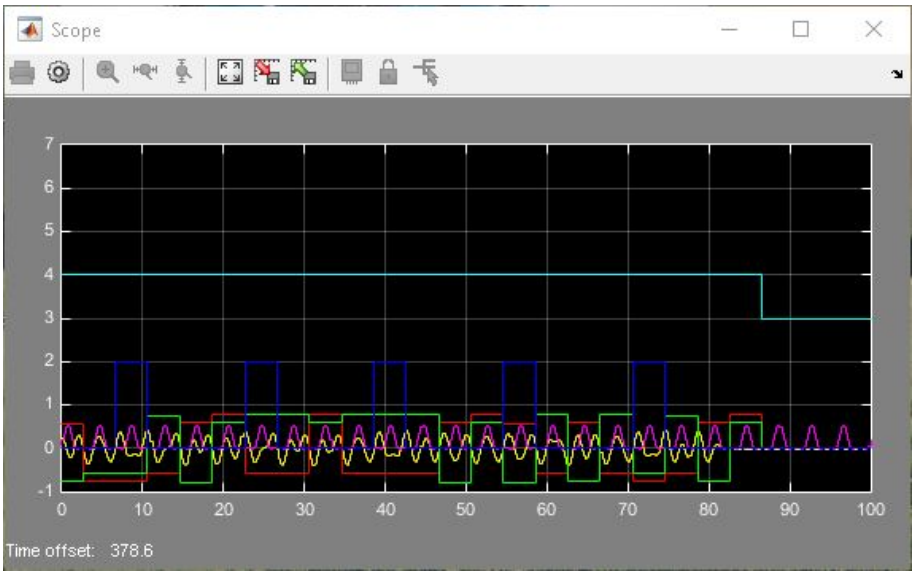

Figura 4.12 Fases 4 a 3 del ejemplo 2..

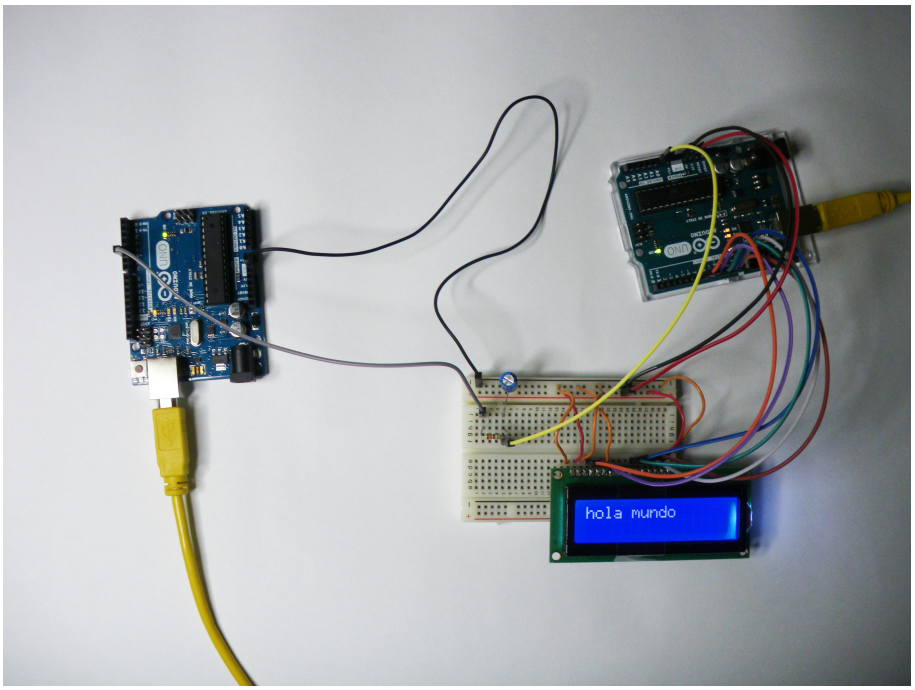

Figura 4.13 Montaje completo. De izquierda a derecha: Transmisor, filtro, LCD y receptor..

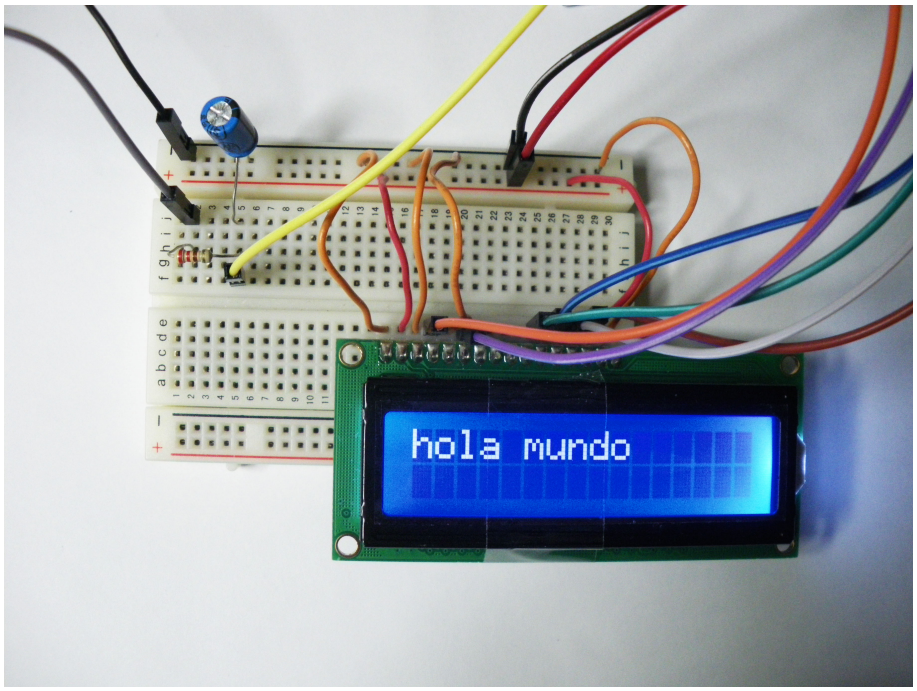

Figura 4.14 Filtro RC, común para ambos ejemplos, y LCD con el mensaje que se ha recibido..

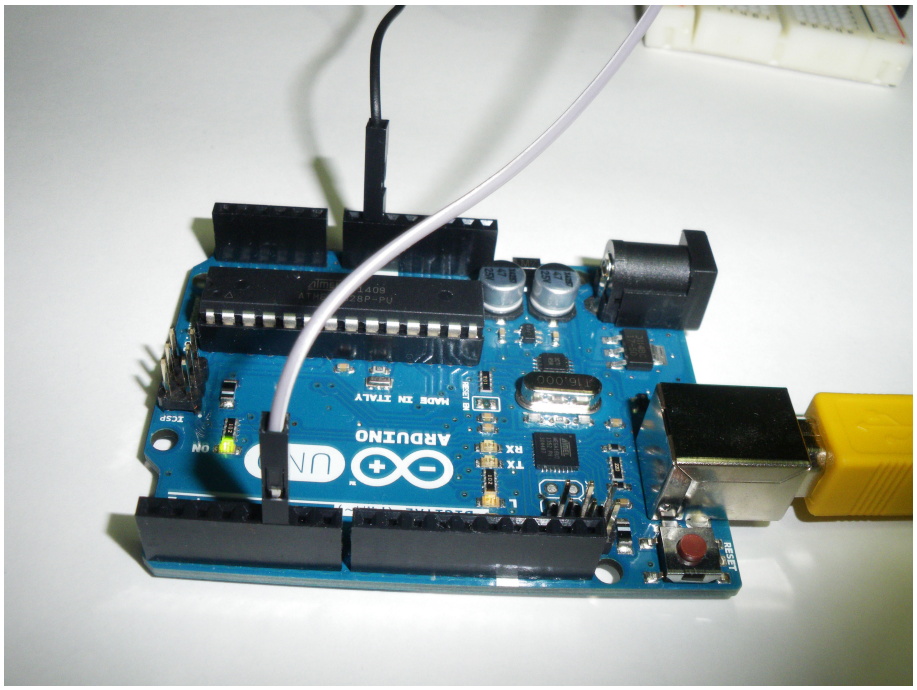

Figura 4.15 Transmisor..

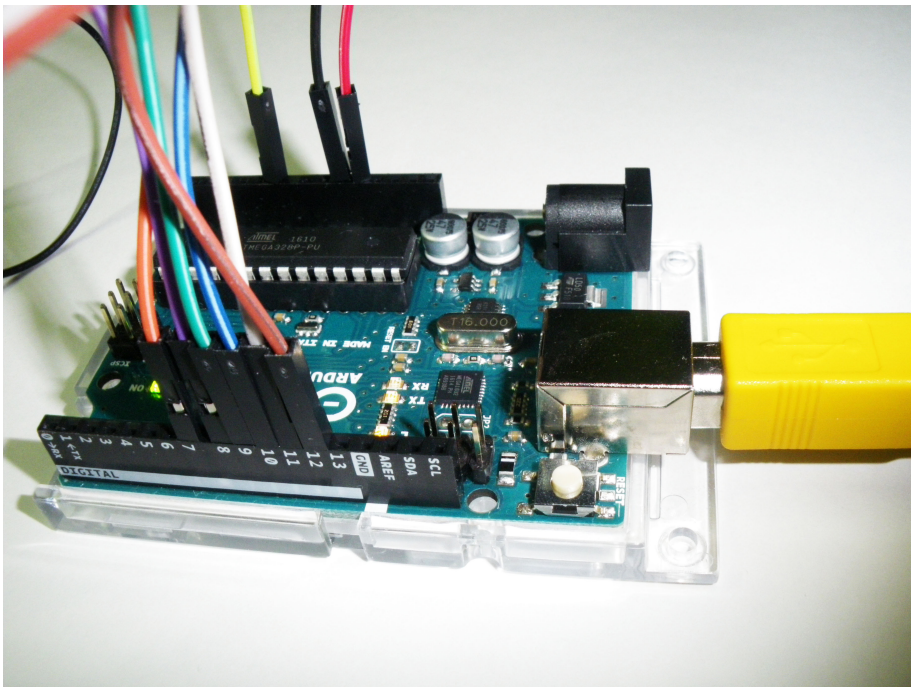

Figura 4.16 Receptor..

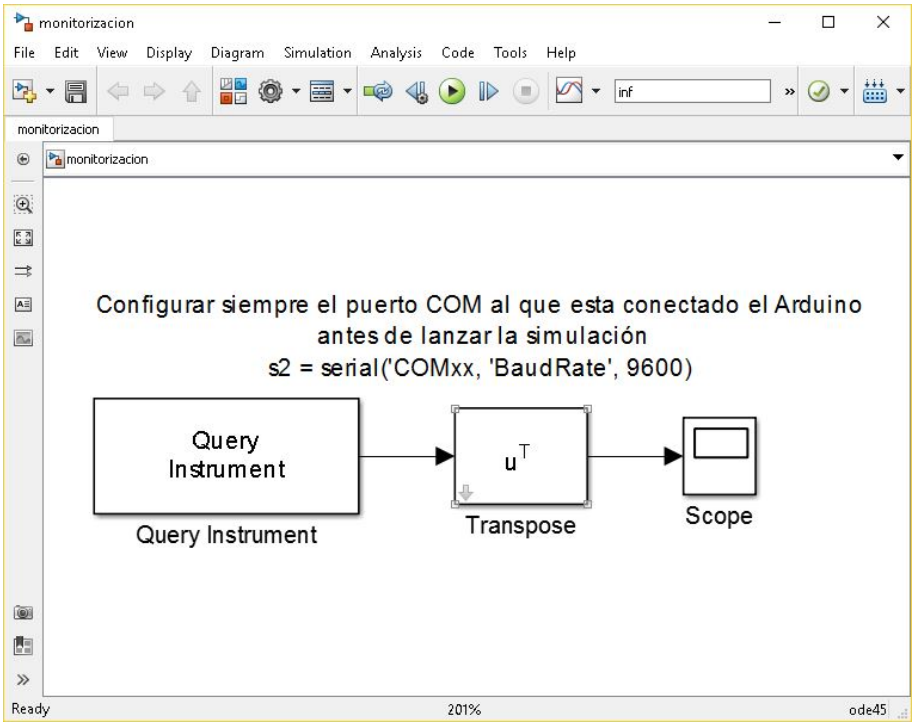

Figura 4.17 Esquemático de Simulink..

5 Conclusiones y futuras lineas de trabajo

La elaboración de este trabajo, ha sido bastante interesante, ya que ha supuesto una implementación práctica de la teoría sobre comunicaciones digitales, utilizando un par de microcontroladores. Para llevar a cabo este desarrollo, fué necesario investigar sobre el entorno de Arduino, y una vez conocida la base, se decidió comenzar a hacer pequeñas pruebas para asentar las ideas y dar los primeros pasos. De esta forma, el aprendizaje ha sido continuo, y se ha estado ampliando la documentación durante todo el proceso, para conseguir solventar los problemas que fueron surgiendo.

Comenzando por un muestreo periódico, se fueron incorporando al programa una serie de funcionalidades, partiendo de la modulación de las señales, hasta llegar a la traducción de caracteres en cadenas binarias y la generación de tramas. La experiencia en general, ha servido para despertar mi interés en áreas como la programación de DSP, desde la que se puede seguir trabajando en esta linea. Una idea atractiva para probar en desarrollos futuros, podría ser diseñar una codificación de fuente y canal para complementar el funcionamiento del transmisor y del receptor.

5.1 Conclusiones

A partir del estudio y aprendizaje que se ha llevado a cabo en este trabajo, se pueden obtener las siguientes conclusiones:

- Es posible trabajar con Arduino para hacer procesamiento de señales. En este sentido, se pueden realizar filtrados, correlaciones, codificaciones y otras operaciones útiles para las comunicaciones digitales de un sistema. Esto es posible gracias a la sencillez de su lenguaje, una variante del C++, y a la posibilidad de crear librerías propias o usar otras ya existentes.
- Resulta una herramienta útil para realizar pruebas y simulaciones prácticas. En concreto, es posible simular algoritmos para comprobar su comportamiento en una situación real, interactuando con los datos recogidos del medio, mediante sensores.
- Puede utilizarse para incorporar componentes electrónicos al proyecto. De este modo, se pueden crear sistemas con un uso más práctico, que interactúen con el usuario mediante teclados, botones, pantallas, motores y otros elementos. Esto puede resultar interesante, en proyectos como la puesta en marcha del sistema de comunicaciones que necesita un dron, y su relación con el resto del hardware.

5.2 Trabajos futuros

Para seguir trabajando en esta línea, se pueden utilizar los resultados del trabajo en distintos ámbitos. Por un lado, Arduino es una herramienta bastante interesante para el aprendizaje a nivel académico. Ciertamente, la visualización de la teoría aplicada a la práctica, resulta muy atractiva para el alumno que estudia la materia. Además, el uso de herramientas como esta, puede ayudar a despertar la creatividad del estudiante, fomentando su interés en el campo. En concreto, puede ser bastante sencillo e interesante, mostrar un caso práctico de filtrado de una señal, utilizando la placa de Arduino, y visualizando los resultados desde el IDE de Arduino, o desde un osciloscopio.

Por otro lado, es posible implementar en Arduino otros diseños más avanzados para el transmisor y el receptor. Estos podrían incorporar codificación de señal y canal, además de utilizar otras modulaciones o

incluso realizar un seguimiento del estado del canal, para contrarrestar sus efectos actualizando los filtros transmisores y receptores.En resumen, la plataforma Arduino resulta muy provechosa como herramienta para hacer pruebas, demostraciones, y para seguir experimentando en el procesamiento de señales.

Índice de Figuras

Índice de Tablas

Índice de Códigos

Bibliografía

[1] *Apuntes de Comunicaciones Digitales*, F. Javier Payán Somet. Departamento de Teoría de la Señal y Comunicaciones. 2016

[2] *ATmega328 datasheet*, http://www.atmel.com/images/Atmel-8271-8-bit-AVR-Microcontroller-ATmega48A-48PA-88A-88PA-168A-168PA-328-328P_datasheet_Complete.pdf

[3] *Arduino Board Uno*, https://www.arduino.cc/en/Main/ArduinoBoardUno

[4] *Arduino Pin Current Limitations* http://playground.arduino.cc/Main/ArduinoPinCurrentLimitations

[5] *Pin Change Int*, http://playground.arduino.cc/Main/PinChangeInt

[6] *Arduino Uno*, https://www.arduino.cc/en/Guide/ArduinoUno#toc4

[7] *Arduino PWM*, https://www.arduino.cc/en/Tutorial/SecretsOfArduinoPWM

[8] *Arduino timer*, https://aprendiendoarduino.wordpress.com/tag/timer/

[9] *Funciones en Arduino*, https://aprendiendoarduino.wordpress.com/tag/funciones/

[10] *Arduino counters*, https://sites.google.com/site/qeewiki/books/avr-guide/counter-atmega328

[11] *Arduino timer tnterrupts*, http://www.instructables.com/id/Arduino-Timer-Interrupts/

[12] *Salidas digitales de Arduino*, https://www.luisllamas.es/salidas-digitales-en-arduino/

[13] *Interrupciones Arduino*, https://www.luisllamas.es/que-son-y-como-usar-interrupciones-en-arduino/