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

Transmisión Li-Fi sobre microprocesador Cortex M4

Autora: MªDolores Tristán del Barrio
Tutor: Vicente Baena Lecuyer

Dep. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2015

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

Transmisión Li-Fi sobre microprocesador Cortex M4

Autor:
MªDolores Tristán del Barrio

Tutor:
Vicente Baena Lecuyer
Profesor titular

Dep. de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2015

Proyecto Fin de Carrera: *Transmisión Li-Fi sobre microprocesador Cortex M4*

Autor: MªDolores Tristán del Barrio

Tutor: Vicente Baena Lecuyer

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2015

El Secretario del Tribunal

Resumen

Debido al incremento exponencial de los dispositivos portátiles que utilizan redes inalámbricas, la explotación del espectro de luz visible para transmitir información se ha convertido en un importante campo de investigación en los últimos años. Como ejemplo más sobresaliente que postula a ser futuro referente en las comunicaciones por luz visible se encuentra la técnica genéricamente conocida como VLC (Visual Light Communication) o, de forma más comercial, Li-Fi (Light Fidelity).

En este proyecto se diseñará un transmisor y un receptor capaces de comunicarse a través de Li-Fi, y se implementará en un microcontrolador ARM Cortex-M4 de 32 bits, integrado en el kit de bajo coste STM32F4Discovery. Se tomará como referencia en la medida de lo posible el estándar PRIME de comunicación a través de línea eléctrica, que incluye toda la cadena de bloques necesaria para transmitir usando OFDM: codificador, entrelazador, barajador, mapper, FFT...

Para ello, en primer lugar tendremos que modelar todo el sistema en Matlab con un doble objetivo: comprender en su totalidad la función de cada uno de los bloques necesarios para transmitir utilizando, en este caso DCO-OFDM; y por otra parte, contribuir a la depuración del sistema hardware. Una vez esté dicho modelo totalmente operativo, será necesario pasar a lenguaje C todos los bloques que realizan el procesamiento digital, además de configurar y utilizar los convertidores digital/analógico y analógico/digital en transmisor y receptor respectivamente. Consecuentemente, tendremos que programar tanto a nivel de bit y registros como trabajar con la FPU utilizando las librerías específicas de ARM para realizar el procesado de señal de la forma más eficiente posible.

Tras esto se realizará la verificación del correcto funcionamiento de cada uno de los bloques contrastando el sistema de Matlab y el implementado en el ARM, pasando a través del puerto serie las salidas del sistema hardware a unos ficheros que manipularemos con un script en Matlab para realizar las comprobaciones. Después de asegurarnos de que la configuración de los convertidores es correcta representando las muestras transmitidas y recibidas en Matlab y con el osciloscopio, finalmente, conectaremos las STM32F4Discovery de transmisor y receptor a través de los mismos. Para ello se deberá tener en cuenta tanto el ajuste de los datos que pasamos y nos devuelven los convertidores, así cómo la sincronización entre transmisor y receptor.

Todo esto demostrará la posibilidad de integrar un sistema DCO-OFDM válido para transmitir a través de luz visible, en una plataforma tan ampliamente utilizada y accesible actualmente, pero a la vez potente, como es un ARM Cortex-M4.

Índice

ÍNDICE DE FIGURAS Y TABLAS

GLOSARIO Y ABREVIATURAS

Wi-Fi:Wireless Fidelity

LED: Light-emitting diode

Li-Fi: Light fidelity

RF: Radiofrecuencia

VLC: Comunicación a través de luz visible

OFDM: Multiplexión por división ortogonal de frecuencia

COFDM: Multiplexión por división ortogonal de frecuencia codificado

ISI: Interferencia entre símbolos

ICI: Interferencia entre portadoras

FFT: Transformada rápida de Fourier

PRIME: PoweRline Intelligent Metering Evolution

IFFT: Transformada inversa rápida de Fourier

DFT: Transformada discreta de Fourier

DVB-T: Digital video broadcasting – Terrestrial

LTE: Long term evolution

ADSL: Asymmetric Digital Subscriber Line

PLC*:* Power Line Communications

DMT: Discrete multitone

bps: Bits por segundo

DSP: Procesador digital de señales

FPGA: Field programmable gate array

BER: Tasa de error de bit

FEC: Forward error correction

IM/DD: Intensity-modulated direct-detection

ACO-OFDM: Asymmetrically clipped optical OFDM

DCO-OFDM: DC- biased optical OFDM

PAPR: Peak-to-Average Power Ratio

DQPSK: Modulación diferencial por desplazamiento de fase (en cuadratura)

DBPSK: Modulación diferencial por desplazamiento de fase (binaria)

D8PSK: : Modulación diferencial por desplazamiento de fase (de orden 8)

CMSIS: Cortex® Microcontroller Software Interface Standard

SPI: Serial peripheral interface

I2C: Inter-Integrated Circuit

RCC: Reset an clock control

UART: Universal Asynchronous Receiver-Transmitter

USART: Universal Synchronous/Asynchronous Receiver-Transmitter

ADC: Convertidor analógico digital

DAC: Convertidor digital analógico

DMA: Direct memory access

GPIO: Entrada/Salida de propósito general

FPU: Floating-point unit

1 INTRODUCCIÓN

1.1 Motivación del proyecto

En los últimos años, el incremento de los aparatos que utilizan redes inalámbricas, sobre todo smartphones y demás dispositivos portátiles conectados a Internet vía Wi-Fi, han impulsado la investigación de la explotación de otras bandas del espectro electromagnético. En este punto se encuentra la que pretende dar uso a la iluminación con diodos emisores de luz (LED) como medio de transporte de información. Esta técnica es conocida genéricamente como VLC (Visual Light Communication) o, de forma más comercial, Li-Fi (Light Fidelity)

Este proyecto tiene como objetivo el diseño e implementación hardware de un transmisor y un receptor capaces de comunicarse a través de Li-Fi. Esto ayudará a identificar los posibles problemas y las limitaciones que pueden presentarse al llevar al terreno real dicho sistema.

1.2 Alcance y objetivos

Como ya se ha introducido, el principal objetivo de este proyecto es la implementación de un sistema de comunicaciones Li-Fi. En primera instancia, y con el objetivo de asegurar una depuración correcta del sistema hardware, modelaremos el sistema en Matlab. Se creará un script que llame a las distintas funciones correspondientes a cada bloque del sistema, y que devuelva los resultados de cada uno en ficheros y variables que contengan la información resultante.

Tras esto, pasaremos a implementar nuestro sistema en un microprocesador ARM Cortex-M4, incluido en una placa de desarrollo Discovery de STMicroelectronics. Para ello, tendremos que, aparte de implementar los distintos bloques, contemplar la necesidad de interactuar con la placa vía puerto serie así cómo la configuración de los convertidores analógico/digital y digital/analógico. Esto nos permitirá obtener los archivos de datos que el sistema hardware vaya procesando en cada bloque, con el objetivo de compararlos con los que nos devuelva el sistema en Matlab. De esta forma podremos detectar más fácilmente los errores que estemos cometiendo en la programación del microprocesador o los que puedan surgir del diseño inicial del sistema.

1.3 Estructura del documento

En la memoria presente se intentará presentar una explicación descriptiva del trabajo realizado, dejándose para los anexos los desarrollos más específicos del proyecto, como podría ser el código implementado. Además, se intentará apoyar al texto con esquemas, gráficas y diagramas de flujo siempre que sea conveniente.

En este primer capítulo se introduce de forma superficial el proyecto, sin entrar en demasiados detalles, dejando las explicaciones teóricas y los conceptos más específicos sobre la transmisión Li-Fi que se han considerado necesarios mostrar para el segundo capítulo.

En el tercer y cuarto capítulo se explicará con más detalle el diseño de transmisor y receptor respectivamente, tanto en el hardware como en Matlab.

En el quinto capítulo se mostrarán los recursos hardware y software que han sido necesarios a la hora de implementar el proyecto, así como la configuración de los mismos, las pruebas a las que se ha sometido y sus respectivos resultados.

En el sexto capítulo, para finalizar, se comentarán los resultados generales y conclusiones que se pueden sacar del desarrollo completo del proyecto, así cómo las posibles mejoras y las futuras líneas de trabajo que origina.

Además del contenido en sí, se ha incluido un índice de figuras y de abreviaturas para facilitar la comprensión del documento. En cuanto a las referencias, hay que puntualizar que no se especifican en el texto si no hay correspondencia directa. Es decir, cuándo el texto escrito es resultado del conjunto de varias fuentes, éstas se incluyen al final aunque no se dispongan los enlaces en el párrafo correspondientes. Si es necesario una mención directa a cualquiera de ellas se indicará en la notación ya establecida: [número de referencia] Esto se debe a que hay ocasiones a lo largo del texto en las que muchas fuentes contribuyen de forma homogénea y no merece la pena añadir referencias específicas que más bien emborronan y hacen más ilegible el texto escrito.

2 COMUNICACIÓN LI-FI Y OFDM

2.1 Comunicación Li-Fi

El término Li-Fi fue nombrado en 2011 por primera vez por el profesor Harald Haas, de la Universidad de Edimburgo, durante una de sus presentaciones de la serie de conferencias *Ted Talks* [1]. En ella, lo describía como la tecnología de comunicación por luz visible que proporcionaría unas comunicaciones móviles bidireccionales y de alta velocidad, tal y cómo posibilita Wi-Fi actualmente. Ese mismo año, se añadió al estándar del IEEE para redes locales y de área metropolitanas un apartado que contemplaba la comunicación de corto alcance a través de comunicación por luz visible (VLC) aportando un modelo de capa física y de enlace pensados para ser capaces de transmitir información multimedia hasta en tres escenarios distintos [2].

Inspirado en VLC, Li-Fi basa su funcionamiento en el control del brillo de uno o varios diodos led, que se apagan y se encienden para transmitir información. La frecuencia a la que lo hacen es lo suficientemente alta como para que lo sigamos percibiendo como una iluminación continua, gracias a la propiedad de persistencia del ojo humano. En el receptor, la intensidad de la luz del led del transmisor se detecta por un fotodiodo sensible a la frecuencia elegida.

Una de las principales motivaciones es aprovechar el hecho de que según la tendencia actual, la iluminación mediantes leds irá reemplazando a la tradicional cada vez en más instalaciones, gracias en gran medida al amplio desarrollo en las tecnologías de semiconductores de los últimos años. Esto nos permite encontrar leds de luminosidad suficiente y capaces de alcanzar altas velocidades de conmutación, a un precio razonable. Gracias a eso es realizable el estudio práctico de VLC a través de prototipos como los descritos en [8].

Además de todo esto, al investigar entre algunos de los diversos estudios que ya se han realizado en torno a Li-Fi [3-10], encontramos las siguientes ventajas adicionales que empujan el estudio de dicha tecnología:

- Utilización del espectro de luz visible, en el intervalo de longitudes de onda desde los 380nm hasta 780nm aproximadamente. Esta parte del espectro no está regulada, por lo tanto puede utilizarse sin licencia, lo que abarata y hace más accesible su uso.

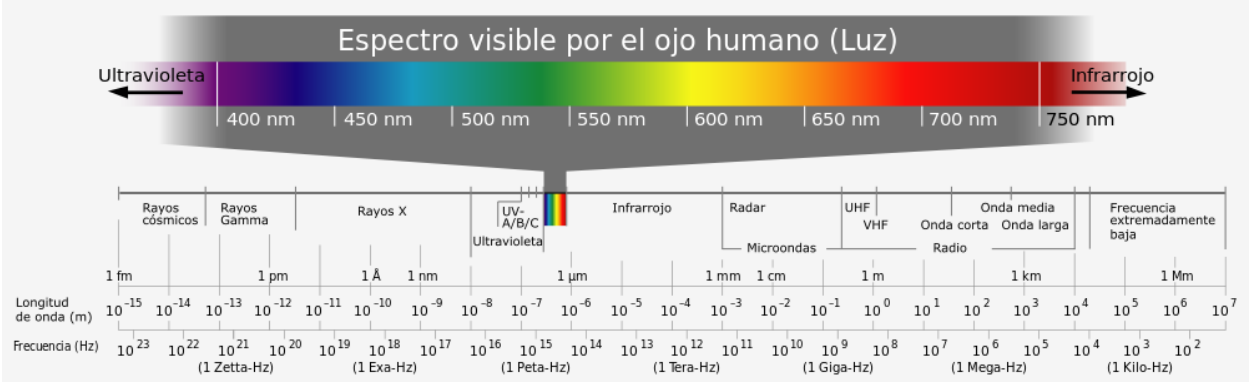


Figura 2-1: Espectro electromagnético y detalle de la ocupación de la banda de luz visible

(Fuente: Wikipedia)

- Aprovechamiento de la iluminación led presente en interiores, señalizaciones de tráfico,

luminarias,… Se presenta la posibilidad incluso de controlar la atenuación de las fuentes de luz a la vez que transmitimos la información.

- Ofrece una comunicación segura ya que cualquier pared opaca bloquearía la señal, y es posible ver en todo momento dónde está confinada. A la vez que potencia la seguridad, este aspecto de la comunicación Li-Fi facilita la reutilización del espectro.
- Es ideal para entornos sensibles a interferencias electromagnéticas, como hospitales, aeropuertos, centrales nucleares,… ya que las señales ópticas no las causan. Incluso se plantea la posibilidad de utilizarlo para reemplazar las conexiones cableadas submarinas.
- Existen aplicaciones en las que la demanda de ancho de banda de bajada es muy grande respecto a la del de subida, como son las descargas de vídeo, audio, retransmisiones en directo… Por lo tanto se baraja que Li-Fi podría ser el apoyo idóneo en los enlaces de bajada de Wi-Fi.

A pesar de todas las ventajas nombradas, aún quedan aspectos que deben resolverse, y que constituyen un amplio campo de investigación actualmente tal y como señala [3]:

- La influencia de la luz visible en los seres humanos debe ser tratada con sumo cuidado, ya que una desviación accidental en parámetros como la frecuencia o la intensidad puede afectarnos, provocando desde naúseas hasta, en el peor de los casos, epilepsia. Aparte de esto, tenemos que tener en cuenta que la transmisión de la información no debe influir en la calidad de la iluminación así cómo permitir en muchos la variación del brillo de la fuente de luz.
- La misma ventaja que aparecía antes en cuanto a seguridad y confinamiento de la señal luminosa se nos puede volver en nuestra contra si no tenemos en cuenta que pueden aparecer obstáculos y sombras entre receptor y transmisor. El escenario se complica aún más si hablamos de comunicaciones entre dispositivos que se mueven, y tendremos que procurar que nuestro sistema funcione aunque no dispongamos de una trayectoria de la luz directa y saber cómo tratar las reflexiones.

La solución que se propone en este proyecto para implementar este sistema se basa en la técnica de modulación y multiplexión conocida como OFDM (Multiplexión por división ortogonal de frecuencia), en la que mandamos la información por diversas portadoras. Esto es debido a que nos garantiza la robustez necesaria ante el efecto multitrayecto del canal.

2.2 La técnica OFDM

El origen de OFDM se remonta a 1966, cuándo Robert W. Chang publicó su artículo sobre la síntesis de señales limitadas en banda ortogonales para transmisión de datos multicanal [11]. La técnica que en ese artículo se proponía posibilitaba transmitir múltiples mensajes a través de un canal lineal limitado en banda evitando, por primera vez, la interferencia entre símbolos (ISI). Más tarde en 1971 S.B.Weinstein y Paul M.Ebert propusieron en un artículo [12] el uso de la Transformada discreta de Fourier (DFT) para realizar la demodulación y modulación en paralelo, sustituyendo a los costosos bancos de demoduladores y moduladores coherentes que hacían impracticable el uso de la técnica propuesta por Chang.

Tras esto siguieron múltiples desarrollos que añadieron diversas mejoras a OFDM, cómo la inclusión del prefijo cíclico o la utilización de portadoras piloto para la sincronización. Sin embargo, no sería hasta los años 90 cuando, impulsada por el amplio avance de la electrónica digital y la capacidad de los circuitos integrados de realizar procesamiento de señal avanzado, que OFDM se incluyera en los estándares de comunicaciones, perdurando hasta la actualidad.

Aunque sufra las convenientes adaptaciones a cada tipo de transmisión, la utilización de OFDM se encuentra en comunicaciones inalámbricas tan comunes como Wi-Fi, la televisión digital terrestre (DVB-T) o la cuarta generación de comunicaciones móviles (LTE); así cómo en medios de transmisión guiado, dónde se denomina modulación por multitono discreto (DMT). Por ejemplo, está presente en la línea de abonado digital asimétrica (ADSL) o la comunicación a través de línea eléctrica (PLC). Esto demuestra la madurez que ha alcanzado la técnica OFDM y que gracias a sus ventajas, puede constituir la base para los próximos avances en telecomunicaciones.

Además, es conveniente puntualizar que a pesar de que se repita el término OFDM a lo largo de la memoria, realmente utilizaremos la técnica COFDM, que añade a OFDM codificación contra errores y entrelazado, haciéndola una técnica aún más robusta. Para ello se añadirán una serie de bloques que se detallarán más adelante. A pesar de que se utilicen ambos términos indistintivamente, es necesario remarcar la diferencia entre ellos.

2.2.1 Ventajas e inconvenientes de OFDM

La principal ventaja de utilizar OFDM es su robustez ante los problemas originados por el efecto multitrayecto del canal, como ya se ha comentado. Esto se debe a que en general, debido a la presencia de obstáculos entre transmisor y receptor se producen múltiples reflexiones de la onda transmitida, provocando que al receptor lleguen copias retardadas y atenuadas de la señal origen: los llamados ecos del canal.

Los problemas que se presentan en este tipo de canal si utilizamos una modulación monoportadora, y contra los que es robusta la técnica OFDM, son:

- **Interferencia entre símbolos (ISI):** un mismo símbolo llega por caminos diferentes en distintos momentos, provocando interferencias con sus propias copias retrasadas además de con los símbolos contiguos.
- **Dispersión temporal de los símbolos:** la duración del símbolo s_n recibido aumenta respecto al tiempo de símbolo T_s inicial debido a la recepción de sus ecos.
- **Desvanecimientos selectivos en frecuencia**: a determinadas frecuencias el canal presenta una atenuación muy acusada.

Sin embargo, también existen varios aspectos que, de no tenerlos en cuenta, podrían ocasionarnos diversos problemas al utilizar OFDM:

- **Interferencia entre portadoras (ICI):** una de las principales características de OFDM es que las subportadoras que conforman la señal sean ortogonales entre sí, pero esto es algo que no siempre se cumple. Debido a que la referencia de frecuencia de transmisor y receptor puede no ser la misma, esta propiedad se pierde, por lo que se producen interferencias entre las distintas portadoras. Para solucionar este problema, tendremos que incluir mecanismos de sincronización entre transmisor y receptor.
- **Alta PAPR:** la PAPR se define como la relación potencia de pico a potencia media de la señal trasmitida, y se calcula como se describe en 2-1. Lo que viene a representar es lo grandes que son los picos de nuestra señal respecto a la potencia media. La alta PAPR de OFDM se debe a que la señal es el resultado de la superposición de un gran número de subportadoras que al sumarse pueden generar dichos picos. Controlar que no saturen es esencial para evitar distorsión, además de que tendremos que tenerlo en cuenta a la hora de elegir el rango y punto de funcionamiento de componentes como el led.

$$PAPR = 10 \cdot \log\left(\frac{|s|^2_{pico}}{s^2_{rms}}\right) \tag{2–1}$$

Aunque no constituya un inconveniente como tal, hay que tener en cuenta que la necesidad de incluir información redundante en COFDM en fases como la codificación así como el uso del período de guarda/prefijo cíclico hace perder parte de la capacidad de transmisión.

2.2.2 La señal OFDM

La modulación OFDM es una modulación multiportadora donde cada portadora viene dada una señal $\phi_k(t)$ definida como:

$$\phi_k(t) = e^{j2\pi f_k t} \cdot u(t) = \begin{Bmatrix} e^{j2\pi f_k t} & 0 < t < T_s \\ 0 & e.o.c \end{Bmatrix} \quad (2\text{–}2)$$

Y que cumplirá que cada $f_k = h/T_s$, con $0 < h < N-1$, haciendo que las el máximo de la portadora k coincida con un cero en las demás, esto hace que las subportadoras sean señales ortogonales entre sí:

$$\int_0^{T_s} \phi_j(t) \cdot \phi_i^*(t) dt = \begin{Bmatrix} T_s & si\ i = j \\ 0 & si\ i \neq j \end{Bmatrix} \quad (2\text{–}3)$$

Si representamos la parte imaginaria respecto al tiempo podemos observar la superposición de las distintas portadoras sinusoidales (Figura 2-2) y cómo en el espectro se refleja la ortogonalidad en frecuencia de cada una (Figura 2-3):

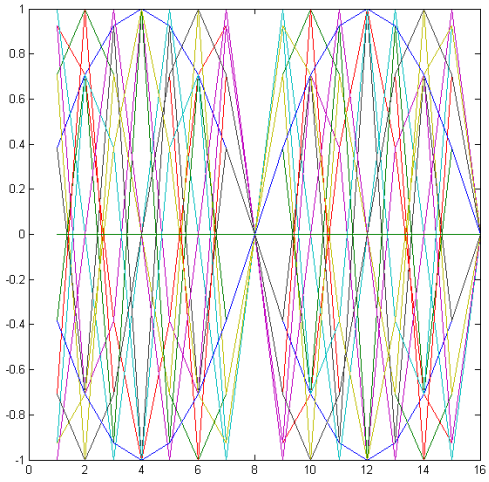


Figura 2-2: Parte imaginaria de la señal OFDM (16 portadoras)

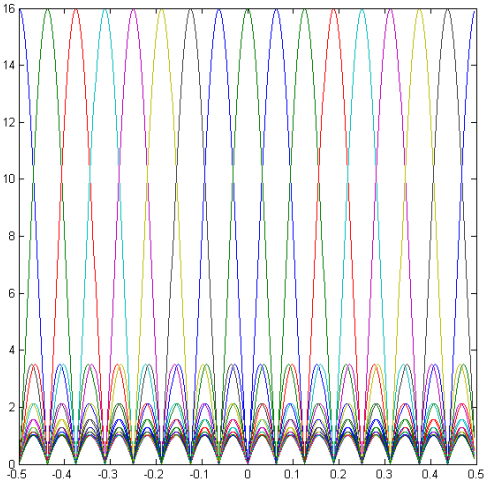


Figura 2-3: Espectro de la señal OFDM (16 portadoras)

La función descrita será la que multiplicaremos por la salida del mapper, constituida por los distintos C_k complejos resultantes de la modulación elegida. Por lo tanto, la señal modulada OFDM quedará:

$$s(t) = \sum_{k=0}^{N-1} c_k \cdot \phi_k(t) = \sum_{k=0}^{N-1} c_k \cdot e^{j2\pi f_k t} \cdot u(t) \quad (2\text{–}4)$$

Para implementar esta modulación, tal y cómo se muestra en la Figura 2-4, serían necesarios N osciladores que mantuviesen con suficiente precisión las frecuencias impuestas para cumplir la ortogonalidad entre las portadoras. Esto en la práctica es algo muy costoso, por lo que se prefiere trabajar en el dominio digital, aprovechando las propiedades de la DFT (Transformada Discreta de Fourier).

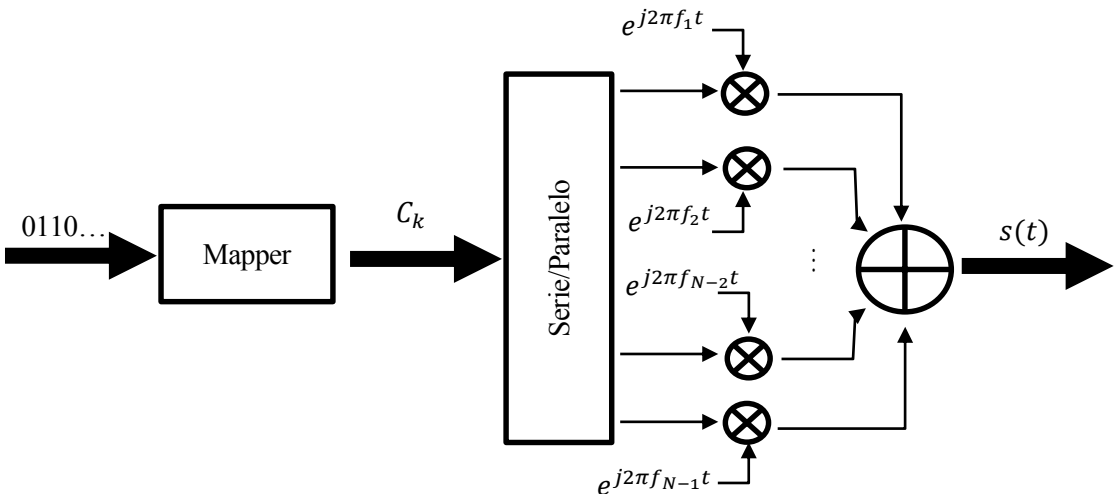


Figura 2-4: Diagrama de bloques de la modulación OFDM

Si muestreamos el símbolo con N muestras cada $T_m = T_s/N$ y N es potencia de 2, podemos sustituir el banco de osciladores por un bloque que ejecute el algoritmo de cálculo rápido de la DFT, conocido como FFT (transformada rápida de Fourier). Numerosos microprocesadores, DSP y FPGA disponen de bloques digitales especialmente diseñados para ejecutar el algoritmo de forma eficiente y sencilla. Por eso, en la práctica, lo que haremos será elegir una IFFT/FFT de M número de puntos, siendo M una potencia de 2 mayor que N, y rellenaremos con ceros las portadoras no utilizadas. Tras esto, y considerando que la salida de la IFFT está en el dominio del tiempo, podemos pasar dichas muestras temporales en serie por un convertidor digital/analógico para generar la señal a transmitir. En el otro lado del demodulador haremos lo análogo con la FFT y un convertidor analógico/digital tal y como indica la Figura 2-5.

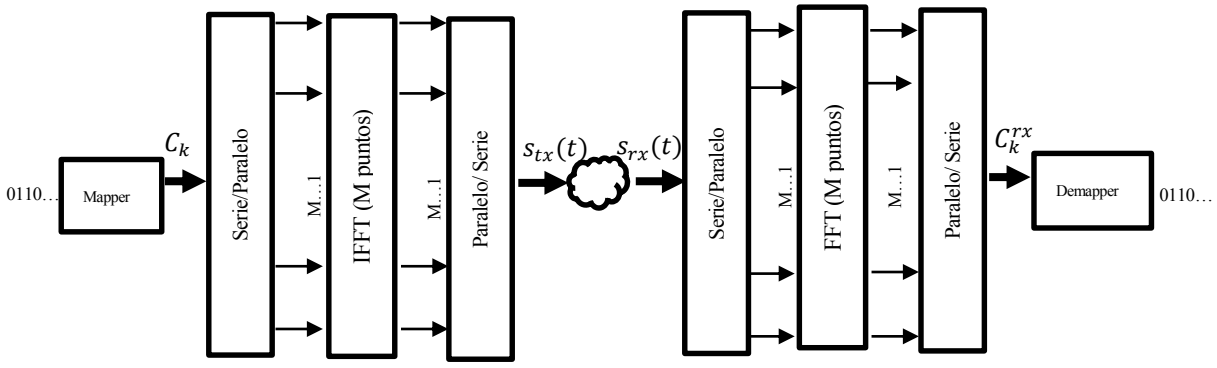


Figura 2-5: Sustitución del banco de osciladores por la IFFT/FFT

2.2.3 El prefijo cíclico

Aunque el haber utilizado la multiplexión en frecuencia para hacer que los símbolos duren más disminuye el problema de la ISI, no hace que éste desaparezca del todo. Para hacer aún más robusto el sistema ante dichas interferencias, evitando el ensanchamiento del espectro, utilizaremos un prefijo cíclico.

El prefijo cíclico consiste en coger un número de muestras determinado del final de nuestro símbolo (NCP muestras) y colocarlas al principio. Esto es posible gracias a que la IFFT/FFT se aplica y da como resultado señales periódicas, por lo que podemos garantizar que el símbolo no se interrumpirá de forma abrupta como resultado de la propiedad de ciclicidad. Aunque con prefijo cíclico tiramos parte de la capacidad de transmisión, evitaremos aumentar el espectro transmitido a la vez que eliminamos la ISI.

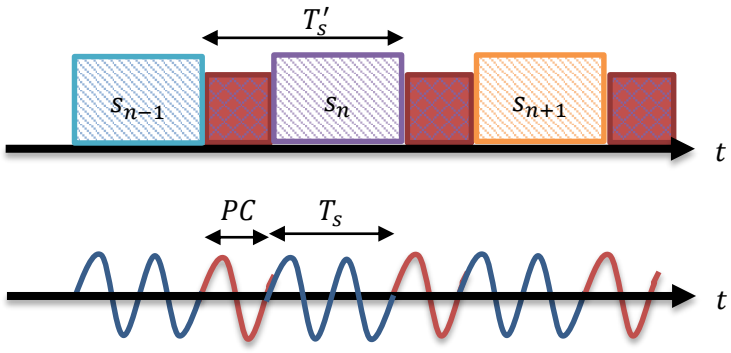


Figura 2-6: Utilización del prefijo cíclico

2.2.4 Codificador convolucional e interleaver

Como ya se ha comentado, la técnica COFDM incluye codificación contra errores, así que, teniendo como objetivo el mejorar la tasa de error (BER) en recepción incluiremos en el transmisor un bloque encargado de añadir información redundante a los bits transmitidos: el codificador de canal o FEC. En nuestro caso utilizaremos un codificador convolucional, que trabaja de forma continua sobre un flujo de datos de cualquier tamaño. Tomando como ejemplo el codificador convolucional que utilizaremos en nuestro sistema (ver Figura 2-7) los parámetros que lo definen serían:

- n = 2. Número de bits de la palabra codificada a la salida.
- k = 1. Número de bits de los datos a la entrada.
- m = 7. Longitud del código. Incluye los bits que se van almacenando en cada registro de desplazamiento así como el bit actual de la entrada.
- Polinomios generadores = 1111001 y 1011011 respectivamente. Indican en cada posición si se contribuye a la XOR de cada salida (1=sí se contribuye, 0=no).

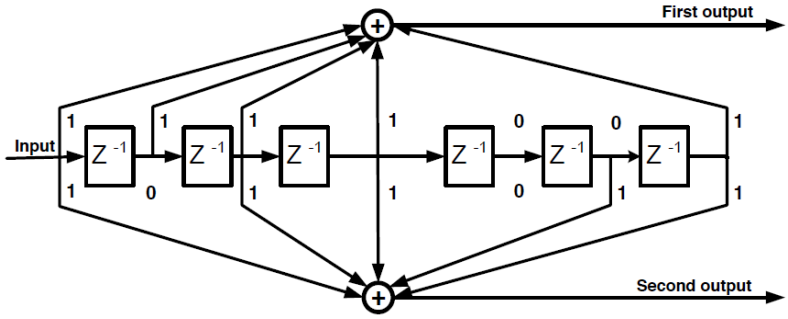


Figura 2-7: Codificador convolucional a implementar.

(Fuente: [18])

Es usual encontrar en la documentación que el codificador puede representarse también como una estructura de Trellis, debido a que es una forma más sencilla de visualizar cómo toman decisiones tanto el codificador como el decodificador. Si tenemos en cuenta que contamos con m-1 registros, podemos considerar al codificador como una máquina de 2^{m-1} estados. Los estados representan los bits almacenados en los registros, la entrada del sistema d_k es el estímulo que hace cambiar de estado, y las salidas del codificador x_k también son las sálidas de la máquina de estados. El diagrama de Trellis (ver ejemplo en la Figura 2-8) nos muestra la evolución de esta máquina de estados a lo largo del tiempo, ante los d_k que vengan por la entrada.

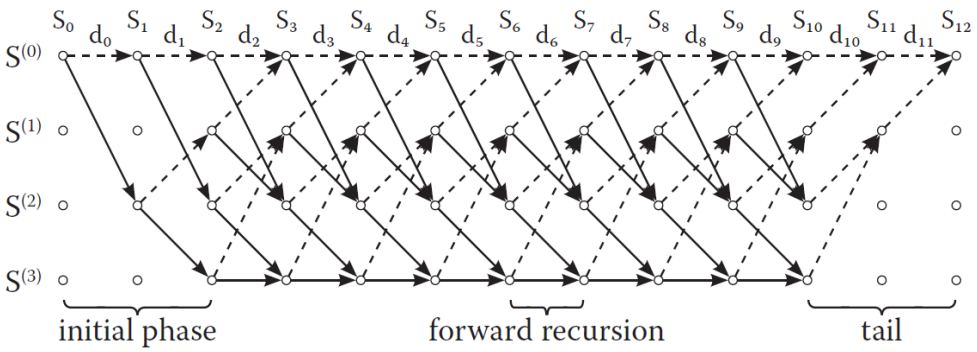


Figura 2-8: Diagrama de Trellis de un codificador de dos registros

(Fuente: [17])

En este caso, los estados vienen definidos por cada fila, y las líneas punteadas indican la ruta a seguir si $d_k = 0$ mientras las que no están punteadas corresponden a $d_k = 1$. También se incluyen los ceros necesarios al final (*tail*) para que los registros queden como al principio y el estado inicial sea siempre 00.

Para decodificar el resultado de una codificación convolucional suele utilizarse un bloque que implemente el algoritmo de Viterbi ya que resulta ser el más eficiente cuando m es menor que 10. En general, para decodificar tendremos que comparar los bits recibidos con todas las posibles secuencias que podrían obtenerse con nuestro codificador, seleccionando la más parecida. La ventaja de utilizar el algoritmo de Viterbi es que, si recibimos L bits, en vez de tener que comparar con las 2^L posibles secuencias recibidas, se irán descartando las secuencias no válidas a lo largo de la ejecución del algoritmo, quedando sólo las válidas para comparar. Se basa en el principio de optimalidad: el mejor camino a través del diagrama de Trellis que pasa por un determinado nodo, necesariamente incluye el mejor camino desde el principio del diagrama de Trellis hasta este nodo.

Es conveniente que el codificador de canal vaya acompañado de un interleaver o entrelazador. Este bloque se encarga de entremezclar los datos de tal forma que en caso de que existiera un error que afecte a muchos bits contiguos en la transmisión, o error en ráfaga, al FEC le llegaran los bits erróneos de forma distribuida. Esto ayuda a que se consigan tasas de error mucho más bajas ya que el decodificador de Viterbi se aprovecha de ello en su funcionamiento.

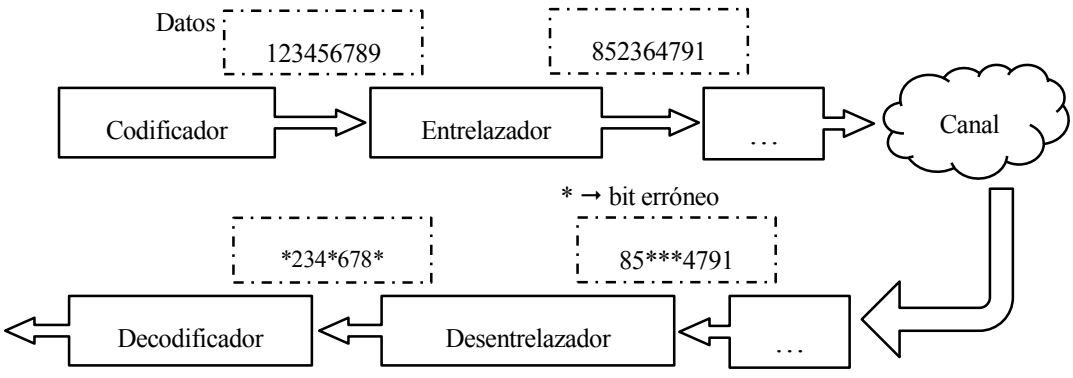


Figura 2-9: Sistema con codificador y entrelazador

2.2.5 Scrambler

El scrambler o barajador es el bloque encargado de evitar que aparezcan largas cadenas de de unos o ceros, así como de que se repitan periódicamente ciertas secuencias. Esto es importante ya podría provocar picos indeseados de amplitud a la salida de la IFFT, o dificultar la sincronización.

Este elemento irá entre el codificador y el entrelazador en el transmisor, y de forma análoga, un descrambler, en el receptor. Para aleatorizar el flujo de datos, haremos la operación XOR con una secuencia generada por un conjunto de registros de desplazamiento, como por ejemplo el de la Figura 2-10:

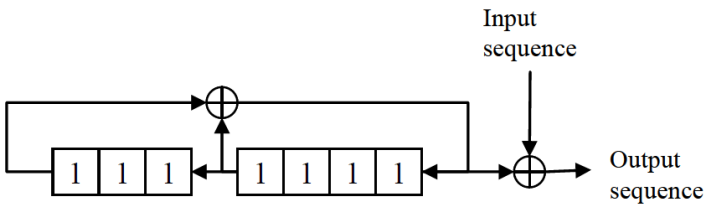


Figura 2-10: Scrambler

(Fuente: [18])

Hay que tener en cuenta que al principio de cada trama los registros deberán estar todos a uno y que cada bit a la entrada generará un bit a la salida, algo similar a lo que pasaría con un codificador convolucional de una entrada y una salida. En el receptor, tendremos que implementar un descrambler como el descrito en la Figura 2-11 para deshacer el barajado de los datos. La estructura de los registros es la misma, y lo que cambia es que ahora no hay autorealimentación como en el scrambler, si no que los bits de entrada son directamente los que se desplazan a lo largo de los registros.

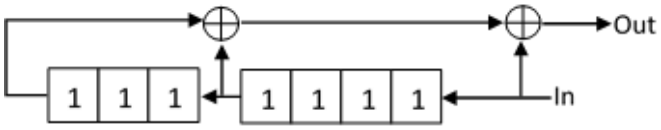


Figura 2-11: Descrambler

2.2.6 OFDM en el canal óptico

Como ya se ha dicho existe un apartado del estándar del IEEE para redes locales y de área metropolitanas dónde se contempla la comunicación de corto alcance a través de comunicación por luz visible [2], pero utilizando una modulación monoportadora. Por ello resulta indispensable, observando las peculiaridades del canal óptico, adaptar la técnica OFDM, tema que está siendo estudiado desde hace relativamente poco tiempo.

Lo primero que debemos tener en cuenta son las diferencias entre los sistemas dónde se ha venido utilizando OFDM y el sistema óptico que vamos a tratar en Li-Fi. Lo definiremos como un sistema IM/DD (intensidad-modulada detección-directa) que básicamente consiste en que la señal OFDM irá modulada en la intensidad que recorre el led del transmisor. La detección directa mediante un fotodiodo o similar generará una corriente proporcional a la potencia recibida. Debido a esto, nos encontramos con el problema de que la señal modulada en intensidad deberá ser real y positiva, mientras que normalmente en OFDM tratamos con señales bipolares y complejas.

El problema de que la señal sea real se soluciona haciendo que el vector de los símbolos resultantes de la modulación a la entrada de la IFFT en el transmisor posea simetría hermítica.

Para conseguir que la señal transmitida sea unipolar, se suele elegir entre las dos técnicas siguientes de OFDM [19-20]:

- **DCO-OFDM**: se añade a la señal un nivel de continua, haciendo que la parte negativa pase a ser positiva. El punto a favor de DCO-OFDM es que permite controlar la intensidad del brillo del led y esto puede ser muy conveniente en aplicaciones de iluminación inteligente que se adaptan a las condiciones atmosféricas. Además, utilizamos todas las portadoras posibles para transmitir información. Como contraparte, requiere mayor potencia que la otra alternativa. Si llamamos $\boldsymbol{s}$ al

vector dónde colocaremos todas las portadoras que se modularán mediante la IFFT, utilizando DCO-OFDM nos quedaría como en 2-5 para cumplir la condición de simetría hermítica. La señal temporal modulada ya en OFDM, para N=16 se muestra en la Figura 2-12, dónde podemos observar cómo es necesario tras modular la aplicación del offset para que se elimine la parte negativa. La elección de dicho nivel de continua afectará al rendimiento del sistema y dependerá de las características del led elegido, de la envolvente de la señal OFDM y del número de subportadoras.

$$\boldsymbol{s} = \left[c_0 \;\; c_1 \;\; \dots \;\; c_{\frac{N}{2}-1} \;\; 0 \;\; c^*_{\frac{N}{2}-1} \;\; \dots \;\; c^*_1 \;\; c^*_0\right]^T \tag{2–5}$$

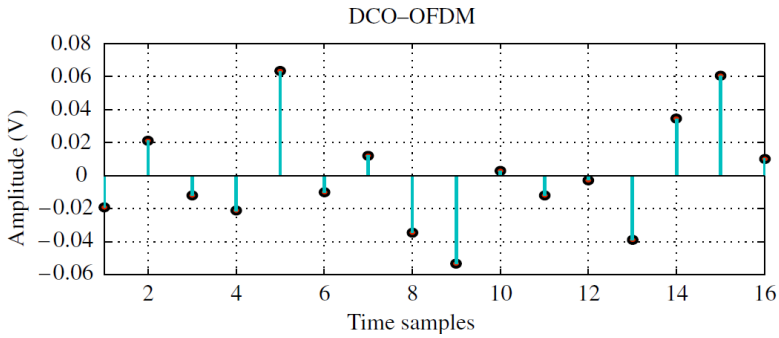


Figura 2-12: DCO-OFDM para N=16

(Fuente: [20])

- **ACO-OFDM**: en este caso se recortan todas las amplitudes negativas de la señal. Si sólo las frecuencias impares son las no nulas en la entrada de la IFFT, la distorsión debido al recortamiento de la señal sólo afectará a las portadoras pares, por lo que la información no se verá afectada, aunque reducirá la amplitud de la señal. El vector $\boldsymbol{s}$ en este caso quedaría como en 2-6, ya teniendo en cuenta la necesidad de simetría hermítica. Si vemos la señal OFDM generada en la Figura 2-13 podemos observar como a partir de la muestra $\frac{N}{2}+1$ la información de la primera mitad se repite con simetría impar, por lo que podemos recortar la parte negativa de la señal OFDM sin que se pierda información, generando así una señal unipolar utilizando menor potencia que en DCO-OFDM .

$$\boldsymbol{s} = \left[0 \;\; c_0 \;\; 0 \;\; c_1 \;\; \dots \;\; 0 \;\; c_{\frac{N}{4}-1} \;\; 0 \;\; c^*_{\frac{N}{4}-1} \;\; 0 \;\; \dots \;\; c^*_1 \;\; 0 \;\; c^*_0\right]^T \tag{2–6}$$

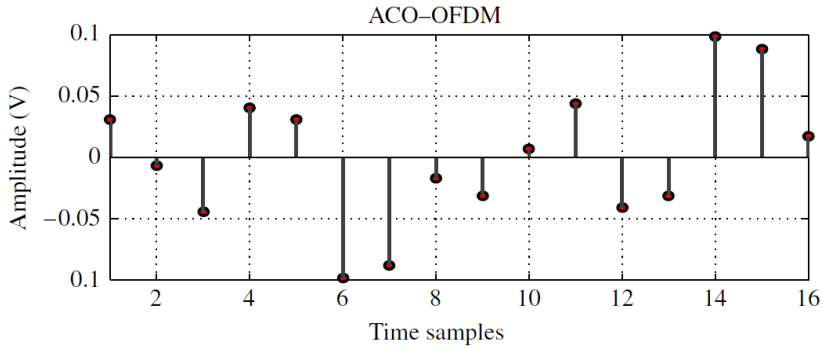


Figura 2-13: ACO-OFDM para N=16

(Fuente: [20])

Por las especificaciones iniciales del proyecto, en este caso se utilizará DCO-OFDM.

3 DISEÑO DEL TRANSMISOR

En este proyecto se partirá del sistema definido para la capa física del estándar PRIME (v1.3.6) [18] para la transmisión de símbolos OFDM. En él se incluye toda la serie de bloques adicionales que se describieron en el Capítulo 2, por lo que servirá de punto de partida para utilizarlo en la transmisión Li-Fi:

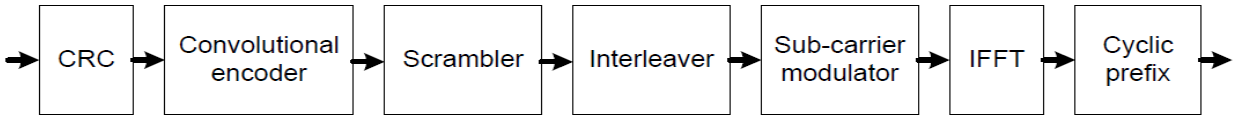


Figura 3-1: Diagrama de bloques descrito en las especificaciones de PRIME

(Fuente: [18])

3.1 Modelado del transmisor en Matlab

Antes de implementar el sistema en el microcontrolador se modelaron los principales bloques por los que pasan los datos en Matlab, con dos objetivos: por una parte, poder comparar la salida bloque a bloque entre ambos sistemas para comprobar su funcionamiento. Por otro lado, el empezar implementando en Matlab el sistema ayuda a comprender el funcionamiento de cada bloque, facilitando así el paso a la aplicación hardware.

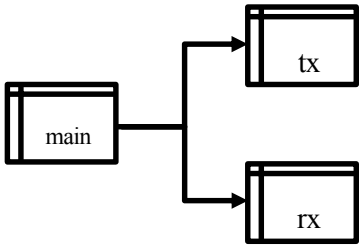


Figura 3-2: Scripts principales

De esta forma el proyecto en Matlab consta de un script **main** que llama a los scripts de transmisor (**tx**) y receptor (**rx**). En él se determina la cadena de caracteres de entrada así como si los datos recibidos son los mismos que los transmitidos o si queremos modelar algún tipo de problema en el canal.

El script del transmisor (**tx**) toma la cadena de datos definida en main y la va haciendo pasar por los distintos bloques, llamando a la función correspondiente. Además, se encarga de transformar las matrices de datos para que se corresponda con lo que espera cada función. Por último, la inclusión del prefijo cíclico se hace directamente en el script, debido a la simplicidad de la operación.

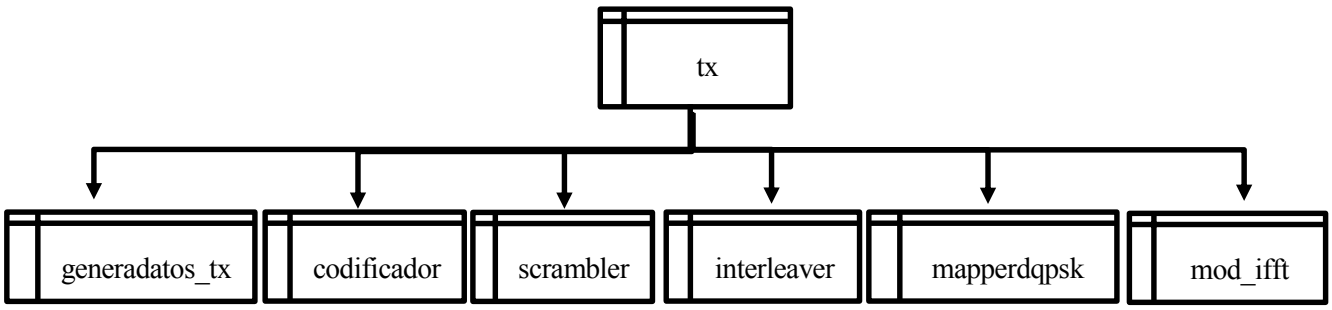


Figura 3-3: Funciones a las que llama el script del transmisor

A continuación se describe el funcionamiento de cada uno de los bloques del transmisor de forma sucinta, cuyo código comentado puede consultarse en los anexos correspondientes de la memoria:

- **generadatos_tx**: en esta función tomaremos la cadena de caracteres que mandamos, la convertimos a bytes de datos binarios (unos y ceros) para luego insertarla en la trama, tal y cómo se muestra en los comentarios del código expuesto. El tamaño de la trama se calculará dependiendo del tamaño de los datos de entrada:

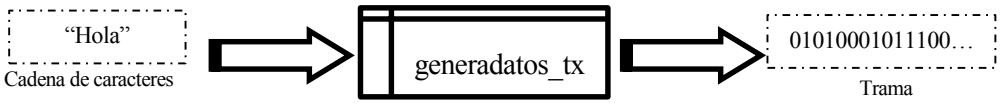


Figura 3-4: Entrada y salida de generadatos_tx

- **codificador**: en esta función sólo definimos la estructura de trellis que le pasaremos a la función de Matlab **convenc**. Para ello haremos uso de la función **poly2trellis,** pasándole la longitud del código y los polinomios generadores en octal de nuestro codificador convolucional. En nuestro caso, ya que utilizamos el codificador descrito en el estándar PRIME [18], dichos datos serán los indicados en la sección 2.2.4 de la memoria, tal y cómo se muestra en el código de la función. A la salida del codificador se sabe que se obtendrán el doble de los datos de entrada más 16 ceros que son los que se encargan de devolver los registros a su estado inicial.

Figura 3-5: Entrada y salida de codificador

- **scrambler**: aquí tomaremos todo el flujo de datos de entrada y lo dividiremos en vectores de 127 elementos para poder hacer la XOR con la secuencia pseudoaleatoria que se obtiene al utilizar el scrambler definido en el estándar PRIME (véase su definición en la sección 2.2.5 de la memoria). Dicha secuencia tiene 127 elementos y está definida directamente para evitar el tener que utilizar bucles para generarla. El único punto a tener en cuenta de esta función, y por lo que aumenta un poco su complejidad es que también tiene en cuenta el trozo sobrante de bits que quedan al dividir en vectores de 127 elementos, haciendo la XOR al final y añadiéndola a la cola de los datos de salida.

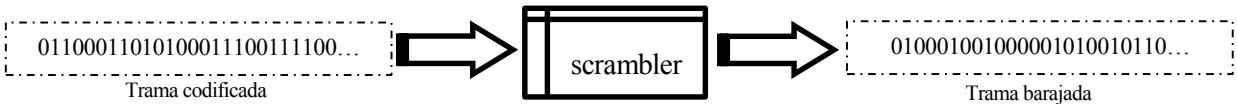


Figura 3-6: Entrada y salida de scrambler

- **interleaver:** antes de llamar a esta función, se reordena el flujo de datos, ya que a partir de ahora dejaremos de trabajar sobre el vector completo de información para trabajar en bloques de 192 bits, según se define el interleaver del estándar PRIME para DQPSK. De esta forma llamaremos a esta función tantas veces como bloques de 192 bits obtengamos al dividir el vector de datos. Se utilizará la función **reshape** para reordenar los datos según la fórmula que se define en el estándar.

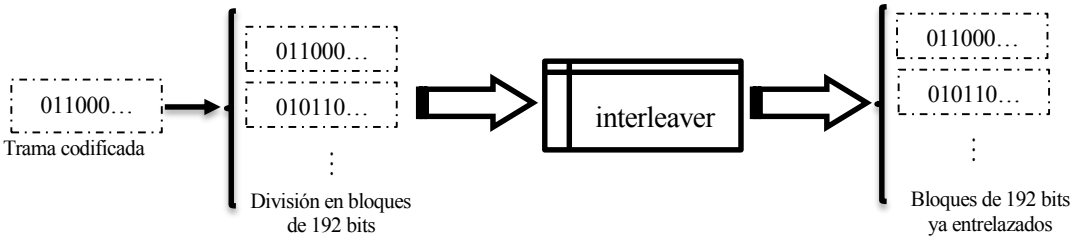


Figura 3-7: Entrada y salida de interleaver

- **mapperdqpsk**: el mapper tomará los bits de entrada de dos en dos para obtener la portadora correspondiente a la modulación diferencial. Para ello, convertiremos cada par de bits a un número

que representa el "número de saltos", o lo que es lo mismo, la cantidad de veces que tenemos que añadir el ángulo $\pi/2$ a la fase actual. Tras esto, en un bucle vamos obteniendo la fase acumulada partiendo de una portadora de referencia que se sitúa en $1 + 0j$. Hay que tener en cuenta, que según el requerimiento del símbolo OFDM de mantener una simetría hermítica, y contando con sólo 128 puntos en la IFFT, al mapper le llegarán sólo 96 bits, obteniendo a su salida 49 portadoras ya que cada portadora representa dos bits y le añadimos la de referencia. Para ello, el bloque de 192 bits que sale del interleaver debe ser nuevamente dividido por la mitad, transformación que se realiza en el mismo script de **tx** antes con la ayuda de la función de Matlab **reshape**.

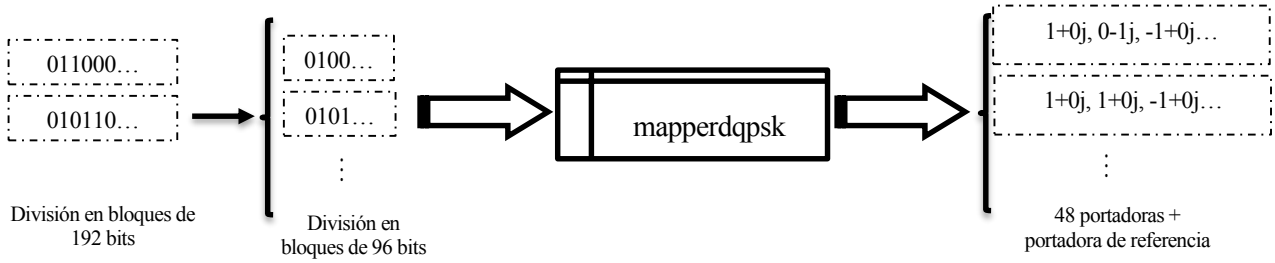


Figura 3-8: Entrada y salida de mapperdqpsk

- **mod_ifft:** en esta función preparamos el vector de entrada para la IFFT, teniendo en cuenta la condición de simetría hermítica y que tendremos que utilizar la función **ifftshift** de Matlab para que la entrada sea correcta. Esto es debido a que la función **ifft** presupone que la entrada tiene los cuadrantes de frecuencias desplazados (la frecuencia cero no está en el centro) En la misma función hacemos la llamada a la **IFFT** con un número de puntos determinado de 128 muestras, por lo que a la salida tendremos un símbolo OFDM de 128 elementos por cada 49 portadoras a la entrada.

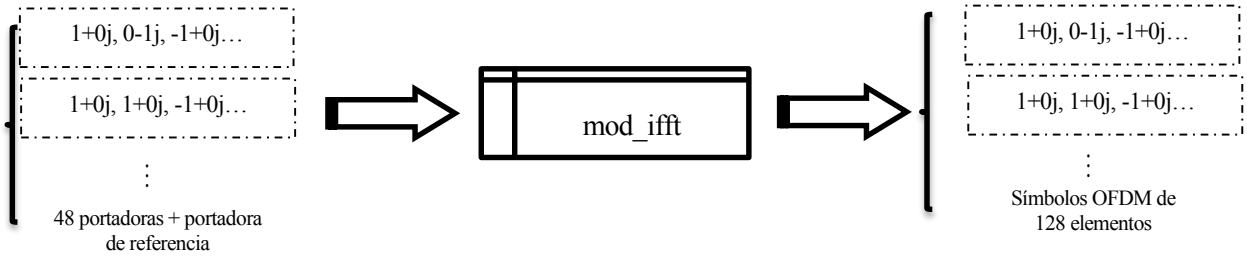


Figura 3-9: Entrada y salida de mod_ifft

Tras el paso por todos los bloques, se añade el prefijo cíclico y se pasa la matriz de símbolos OFDM a un vector con todos los símbolos en serie que serán finalmente los datos transmitidos.

3.2 Implementación del transmisor en el microcontrolador

Tras comprobar que el modelo de Matlab funciona según lo esperado se pasa a realizar el programa en C que cargaremos en la STM32F4Discovery. Debido a que transmisor y receptor estarán en placas distintas, se ha realizado una distribución de ficheros análoga a la de Matlab, con el código del transmisor en **tx.c** y el código del receptor en **rx.c**. Las constantes y macros para ambos estarán definidas en **main.h.**

3.3 Visión general

En el main se implementa tanto la parte del transmisor como la del receptor, por lo que para distinguir entre ambos modos de funcionamiento se comprueba una de las entradas digitales: si se encuentra a nivel bajo pasamos a la parte del transmisor y en caso contrario, a la del receptor.

La parte de **main.c** que gestiona el transmisor, aparte de llamar a la función **transmisor** con los parámetros correspondientes se encarga de ajustar los datos para luego pasarlos al convertidor digital analógico. Para ello hace uso de las siguientes funciones propias:

Función	**Descripción**
`Void ajusta_rango_DAC (float32_t *buffer_entrada, uint32_t tam_datos, uint16_t *buffer_salida)`	Ajusta los datos de *buffer_entrada* de forma conveniente para el DAC, dejando el resultado en *buffer_salida.*
`void DAC1_DMA_Timer6Trigg ()`	Configura el convertidor digital analógico, listo para usar la DMA y el Timer 6 como disparo.

Tabla 3-10: Funciones propias utilizadas en el transmisor

Por lo tanto, el diagrama de bloques general del transmisor queda de la siguiente forma:

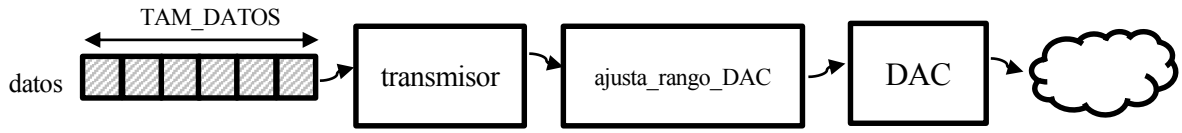


Figura 3-11: Diagrama general del transmisor

3.3.1 Transmisor

```
 void transmisor(uint8_t *datos, uint8_t tam_datos,float32_t
*salida_trama);
```

La función del transmisor recibe del main sendos punteros a los datos de entrada y salida, así cómo el tamaño del vector de entrada, cuyo valor estará definido en **main.h** en la siguiente constante:

TAM_DATOS	(Depende de los datos de entrada)	Tamaño del vector de datos de entrada.

Tabla 3-12: Constantes definidas en **main.h** referentes al tamaño de los datos de entrada

Además, en ella se declaran las variables de entrada y salida de cada uno de los bloques intermedios, cada una del tamaño pertinente según las constantes de **main.h**. Sólo destacar que el resultado del interleaver incluye todos los bloques procesados mientras que el mapper, dentro del bucle ***while*** en el que es llamado, da como salida un solo símbolo cada vez, con objeto de que sea la entrada de ***ifft_prefijo*** sin necesidad de guardar todos los símbolos mapeados.

De esta forma, el funcionamiento es análogo al de la función ***tx*** realizada en Matlab, tal y cómo se detalla en la siguiente Figura:

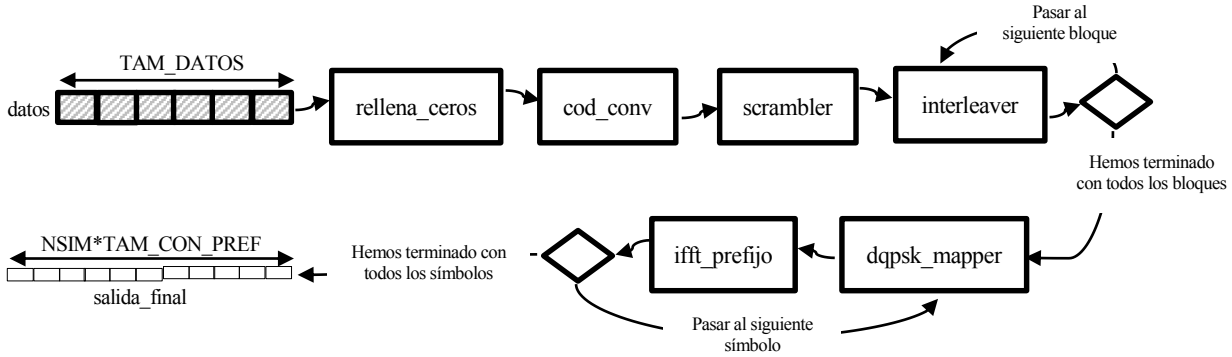


Figura 3-13: Funcionamiento del transmisor

3.3.2 Formación de la trama

```
void rellena_ceros(uint8_t *entrada, uint8_t tam_datos,uint8_t
*osalida_tram);
```

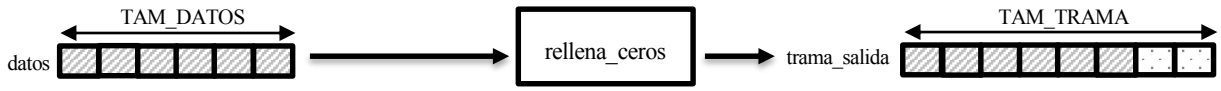


Figura 3-14: Diagrama de la función rellena_ceros

Esta función se encarga de formar la trama de manera que se ajuste a un número de bits totales múltiplo de 192, que es el número de bits con el que trabaja el interleaver. Para ello debe tener en cuenta que al pasar por el codificador convolucional se duplican los datos y los N bits de partida pasan a ser (N*2)+16 ya que también añadiremos un byte al final para que los registros del codificador vuelvan a su estado inicial.

Para realizar esto, el compilador realiza los cálculos necesarios en las macros que se encuentran en **main.h** para definir las constantes que determinan el tamaño de la trama total:

NBYTESPBLOQ	24	Número de bytes por bloque del interleaver.
TAM_DATOS_COD	((TAM_DATOS*2)+2)	Número de bytes que se generarían tras codificar.
NBLOQ	((TAM_DATOS_COD/NBYTESPBLOQ)+1)	Número de bloques que recibe el interleaver.
TAM_TRAMA	(NBLOQ*12)	Tamaño de la trama en bytes.

Tabla 3-15: Constantes definidas en **main.h** referentes al tamaño de la trama

De esta forma, el microcontrolador solo tendrá que copiar mediante un bucle ***while*** los datos de entrada iniciales al principio del espacio reservado para la trama, dónde los bytes sobrantes estarán ya inicializados a cero.

3.3.3 Codificador convolucional

```
void cod_conv(uint8_t *entrada, uint8_t tam_datos,uint8_t*
salida_cod);
```

La función recibe punteros a los vectores de salida y de entrada, así cómo el tamaño de los datos de entrada para poder saber cuándo debe terminar de codificar. Además, el tamaño del vector de salida está calculado en la siguiente constante del archivo **main.h**:

TAM_TRAMA_COD	(TAM_TRAMA*2)	Tamaño de la trama tras codificar en bytes.

Tabla 3-16: Constantes definidas en **main.h** referentes al tamaño de la trama codificada.

Se implementará el codificador convolucional tal y cómo se describe en el estándar, que se muestra en la Figura 3-17:

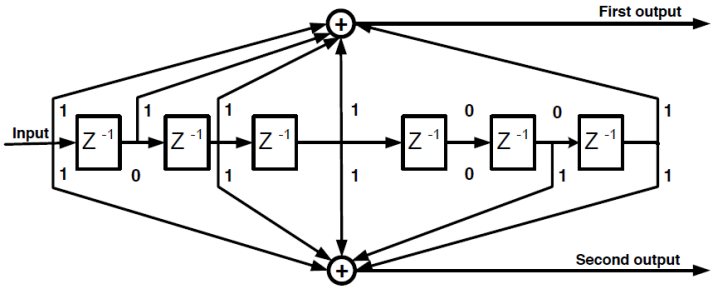


Figura 3-17: Codificador convolucional a implementar

(Fuente: [18])

Para ello, reservamos memoria para un uint16 (dos bytes) que trataremos a partir de ahora como si fuera un registro de desplazamiento que incluye los registros del codificador, tal y como se muestra en el siguiente esquema:

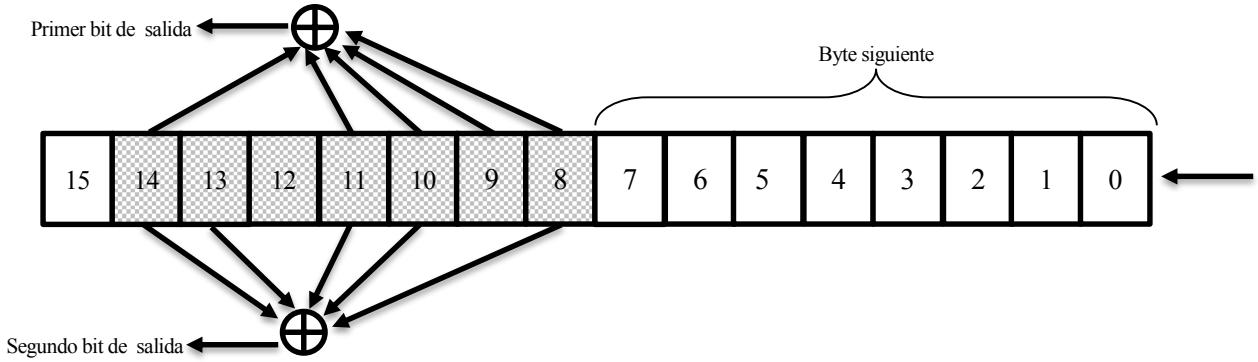


Figura 3-18: Registro de desplazamiento implementado para el codificador.

Por lo tanto, la función realiza las siguientes operaciones:

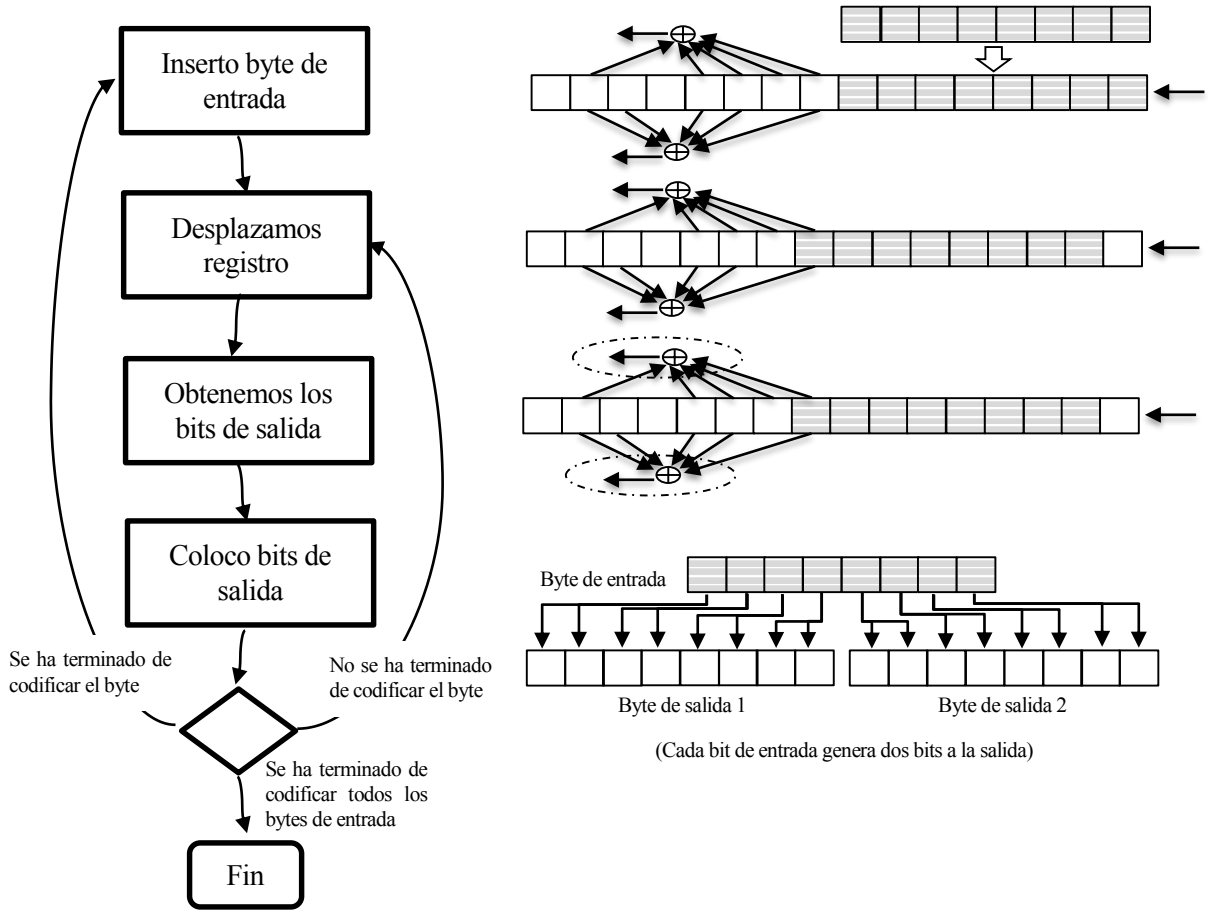


Figura 3-19: Funcionamiento del codificador convolucional

Al finalizar, obtenemos en el vector de salida la codificación pertinente del vector de los datos de entrada, ordenado de izquierda a derecha.

3.3.4 Scrambler

```
void scrambler(uint8_t *entrada, uint8_t tam_datos,uint8_t* salida_scr);
```

Atendiendo a su definición, el scrambler recibe punteros a los vectores de salida y de entrada, así cómo el tamaño de los datos de entrada para poder saber cuándo debe terminar. Este tamaño será la misma constante de **main.h** que define el número de elementos de la salida del codificador convolucional (ver Tabla 3-16).

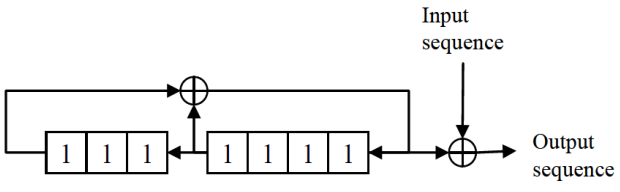


Figura 3-20: Registros necesarios para obtener la secuencia pseudoaleatoria del scrambler

(Fuente: [18])

A la hora de implementar el scrambler definido en PRIME podríamos haber optado por dos alternativas: la primera, obtener la secuencia pseudoaleatoria a través del registro de desplazamiento descrito en la Figura 3-20, o simplemente guardar dicha secuencia de 127 bits en memoria para ir haciendo la XOR byte a byte.

En este proyecto se ha optado por la segunda opción al ser la de mayor rendimiento en velocidad a priori, ya que contamos con memoria suficiente. Por lo tanto, el primer paso es realizar la extensión cíclica de los 127 bits de la secuencia facilitada en el estándar (ver Figura 3-21) para guardarla en un vector de bytes.

The scrambler block performs a xor of the input bit stream by a pseudo noise sequence pn, obtained by cyclic extension of the 127-element sequence given by:

$Pref_{0..126}$=
{0,0,0,0,1,1,1,0,1,1,1,1,0,0,1,0,1,1,0,0,1,0,0,1,0,0,0,0,0,0,1,0,0,0,1,0,0,1,1,0,0,0,1,0,1,1,1,0,1,0,1,1,0,1,1,0,0,
0,0,0,1,1,0,0,1,1,0,1,0,1,0,0,1,1,1,0,0,1,1,1,1,0,1,1,0,1,0,0,0,0,1,0,1,0,1,0,1,1,1,1,1,0,1,0,0,1,0,1,0,0,0,1,1,0,1
,1,1,0,0,0,1,1,1,1,1,1,1}

Figura 3-21: Extracto del estándar dónde se define la secuencia de bits pseudoaleatoria

(Fuente: [18])

Para conservar la ciclicidad de la secuencia al pasarla a un vector de bytes, hay que concatenarla repetida ocho veces como mínimo, obteniendo así un vector de 127 bytes. Teniendo este vector en memoria, la función del scrambler simplemente tendrá que recorrerlo tantas veces sea necesario para hacer la XOR con el vector de bytes de entrada. Esto se realiza a través de dos bucles ***while***, según se ilustra en el siguiente diagrama:

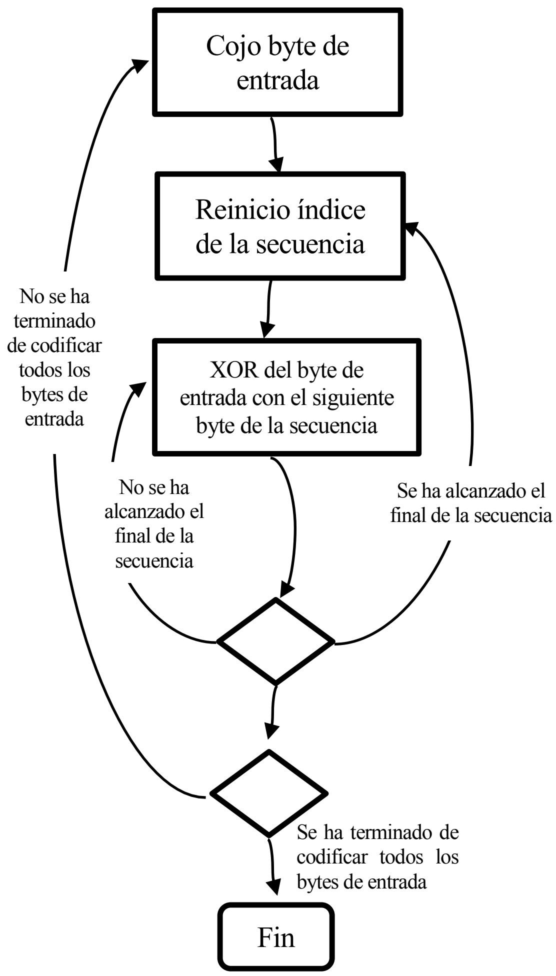


Figura 3-22: Funcionamiento del scrambler

3.3.5 Interleaver

```
void interleaver (uint8_t *datos,uint8_t *salida );
```

Como en las funciones anteriores, este bloque recibe punteros a los vectores de entrada y salida. Ya que lo hemos definido como un interleaver de bloque, se conoce de forma exacta el número de bits que debe procesar y no es necesario que se le pase como parámetro, si no que directamente utilizará la siguiente constante definida en **main.h**:

NBPBLOQ	192	Número de bits por bloque del interleaver.

Tabla 3-23: Constantes definidas en **main.h** referentes al tamaño de bloque del interleaver

El entrelazado en PRIME se realiza, según se define en el estándar, utilizando un interleaver de bloque que actúa sobre un grupo de N_{BPS} bits (número de bits por símbolo OFDM). Dicho número depende de las portadoras ocupadas y del número de bits por portadora. Tal y cómo se detallará en el siguiente apartado, nuestro mapper utilizará una modulación DQPSK, por lo que el número de bits por portadora será dos. Además, en cada símbolo OFDM utilizaremos 96 portadoras ocupadas, por lo que el interleaver trabajará sobre bloques de $N_{BPS} = 96 \cdot 2 = 192$ bits.

En el estándar se define la fórmula mediante la cuál obtenemos los índices de los bits que se van transmutando a la salida (ver Figura 3-24) Aunque podríamos hacer que el microcontrolador calculase los valores obtenidos en dicha fórmula, se ha optado por calcularlos anteriormente y guardarlos en un vector. De esta forma, se evita tener que hacer ese procesamiento aprovechando que disponemos de memoria suficiente.

Let $N_{CBPS} = 2 \times_{NBPS}$ be the number of coded bits per OFDM symbol in the cases convolutional coding is used. All coded bits must be interleaved by a block interleaver with a block size corresponding to N_{CBPS}. The interleaver ensures that adjacent coded bits are mapped onto non-adjacent data subcarriers. Let v(k), with k = 0,1,..., N_{CBPS} –1, be the coded bits vector at the interleaver input. v(k) is transformed into an interleaved vector w(i), with i = 0,1,..., N_{CBPS} –1, by the block interleaver as follows:

$$w((N_{CBPS}/s) \times (k \bmod s) + \text{floor}(k/s)) = v(k) \qquad k = 0,1,..., N_{CBPS} - 1$$

The value of s is determined by the number of coded bits per subcarrier, $N_{CBPSC} = 2 \times N_{BPSC}$. N_{CBPSC} is related to N_{CBPS} such that $N_{CBPS} = 96 \times N_{CBPSC}$ (payload) and $N_{CBPS} = 84 \times N_{CBPSC}$ (header)

$s = 8 \times (1 + \text{floor}(N_{CBPSC}/2))$ for the payload and
$s = 7$ for the header.

R1.3.6 page 34 PRIME Alliance TWG

Figura 3-24: Extracto del estándar dónde se define la fórmula del interleaver

(Fuente: [18])

Así, el funcionamiento de este bloque es el siguiente:

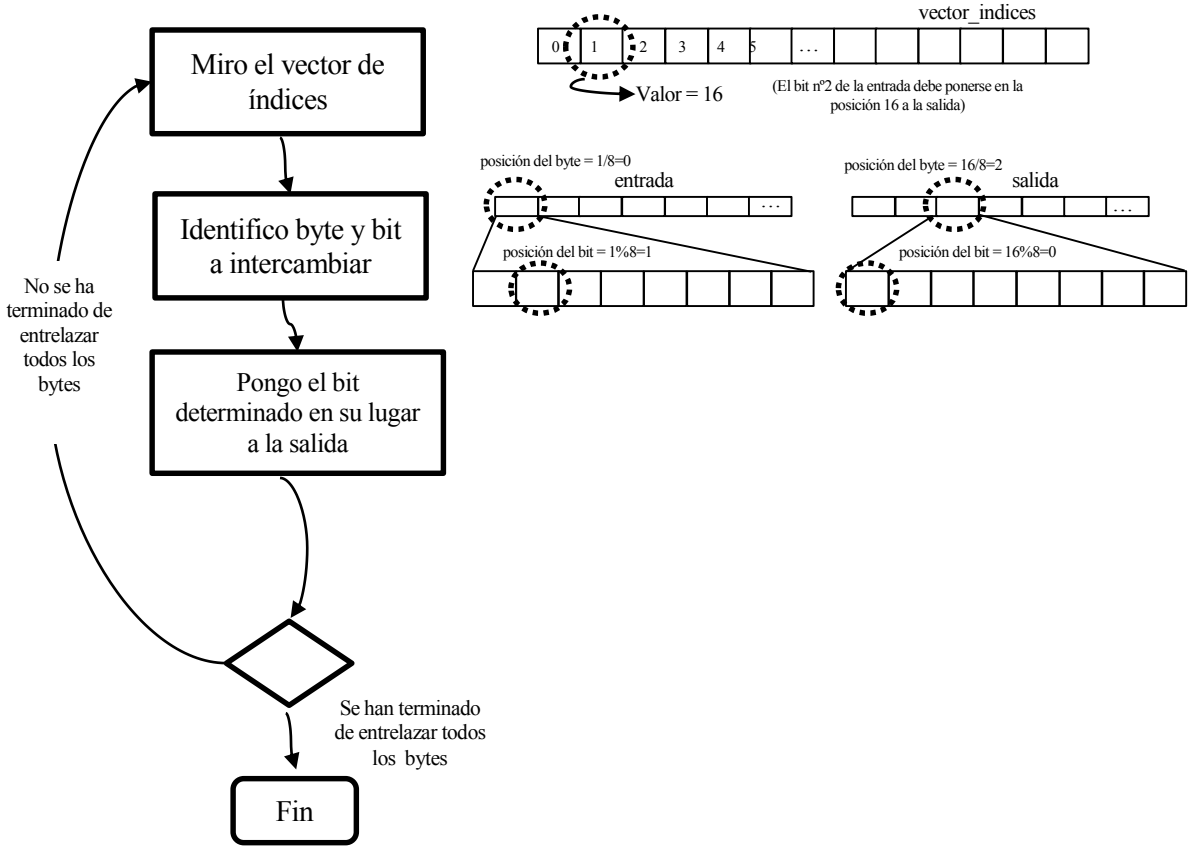


Figura 3-25: Funcionamiento del interleaver

3.3.6 Mapper

```
  void dqpsk_mapper(uint8_t *bits_entrada,float32_t *portadoras_salida,
uint8_t n_elem_portadoras);
```

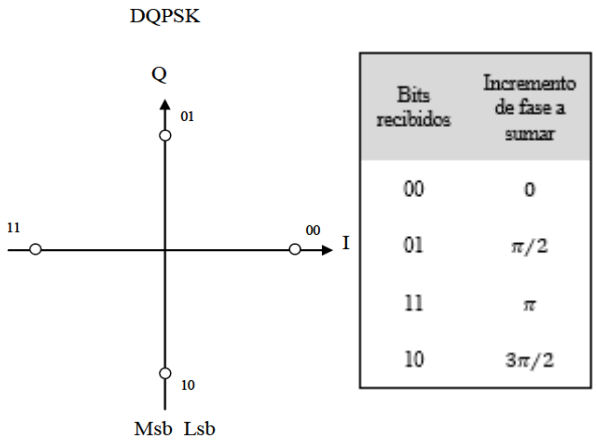
A esta función tendremos que pasar como parámetro el puntero al vector de los datos de entrada, que contendrá los 98 bits resultantes de dividir en símbolos los datos procesados por el interleaver. Además, el puntero al vector de los datos de salida que, en este caso, almacenará un único símbolo ya mapeado. Por último, el número de portadoras que deben procesarse, teniendo en cuenta la portadora de referencia que añade el mapper para la modulación diferencial. Tanto el tamaño de los datos de entrada como el de los de salida están definidos mediante constantes del fichero **main.h**, así como el número de portadoras que el mapper debe procesar por símbolo, y la potencia transmitida de cada portadora:

NSIM	(NBLOQ*2)	Número de simbolos ofdm con N_PORT portadoras que pasarán al mapper
NBYTESMAPPER	12	Número de bytes que se mapean por símbolo
POT	1	Potencia transmitida en cada portadora
N_PORT_2	98	Número de portadoras teniendo en cuenta parte real e imaginaria

Tabla 3-26: Constantes definidas en **main.h** referentes al mapper

A pesar de que en el estándar PRIME se define un mapper capaz de realizar tanto DBPSK como DQPSK y D8PSK, en este proyecto se ha preferido implementar sólo DQPSK (ver Figura 3-27). Por lo tanto, tendremos que tomar la salida del interleaver de dos en dos bits, ver qué posición de la constelación representa, e ir sumando la fase correspondiente a la fase acumulada de los bits anteriores para realizar la modulación diferencial. En el caso de DQPSK, el incremento de fase mínimo es de $\frac{\pi}{2}$, y la fase a sumar dependerá de los bits a transmitir según se ve en el cuadro de la Figura 3-27.

El proceso que sigue esta función se describe en la Figura 3-28:



Bits recibidos	Incremento de fase a sumar
00	0
01	$\pi/2$
11	π
10	$3\pi/2$

Figura 3-27: Mapeado definido para DQPSK en PRIME

(Fuente: [18])

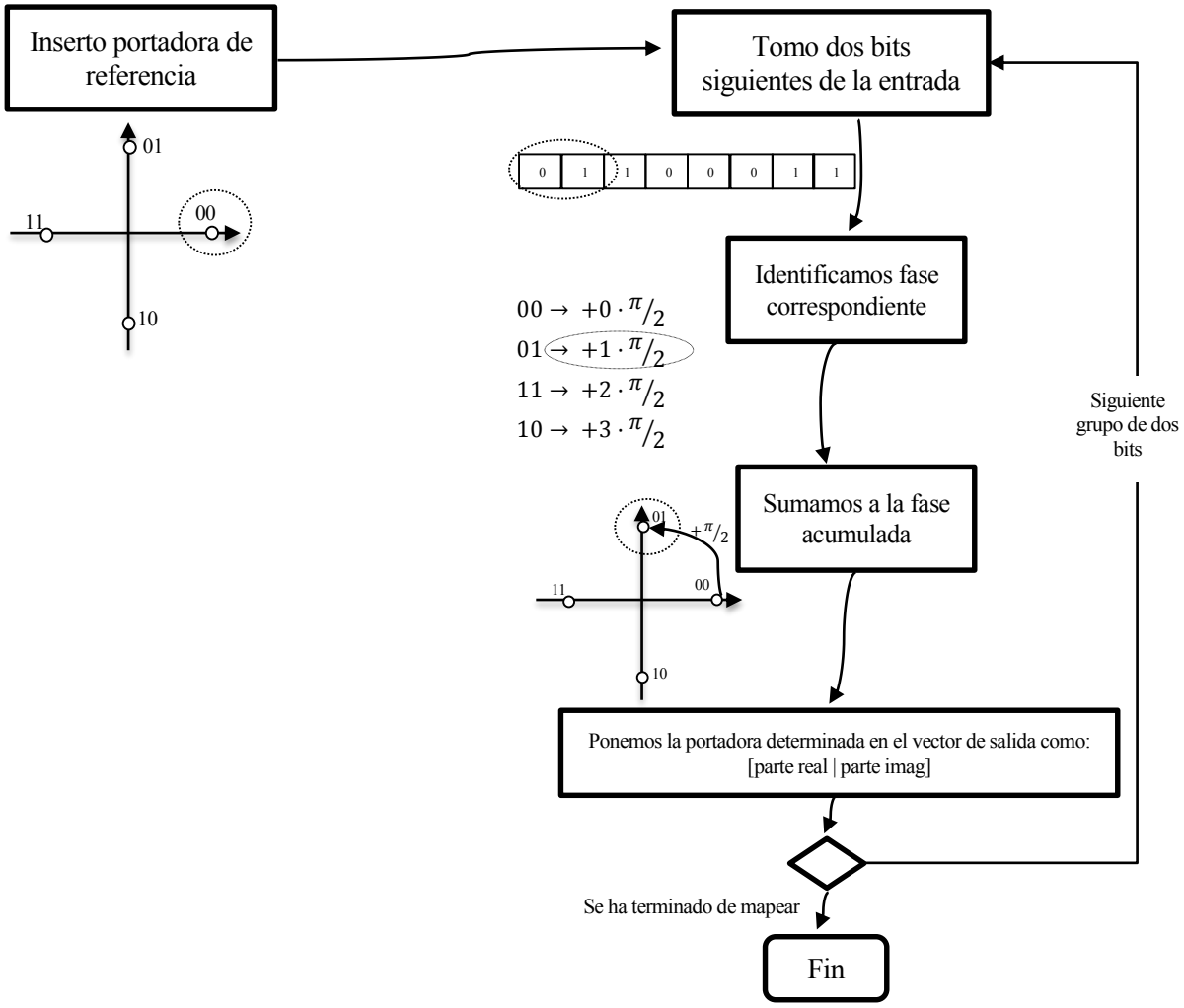


Figura 3-28: Funcionamiento del mapper

El vector de salida debe estar organizado de forma [parte real| parte imaginaria] por portadora debido a que la función que realizará la IFFT necesita que los datos de entrada estén dispuestos de esa forma (se detallará en el siguiente apartado).

3.3.7 IFFT + Prefijo cíclico

```
 void     ifft_prefijo(float32_t     *portadoras_mapper,     float32_t
*salida_ifft_pref);
```

Esta función es la última en la cadena de bloques por lo que, aparte de recibir como todas las demás el puntero al vector de datos de entrada resultantes del mapper, recibe el puntero al vector de la salida final, definido en el main y pasado como parámetro a la función ***transmisor***.

El tamaño de los datos de entrada esta definido por la constante que se definió en el apartado del mapper, **N_PORT_2** (ver Tabla 3-26) Por otra parte, en este bloque también haremos uso de las siguientes constantes del fichero **main.h**:

TAM_CON_PREF	280	Tamaño de cada símbolo completo con prefijo cíclico.
IND_PORT	2	Índice desde donde metemos las portadoras.
IND_PORT_M	254	Índice donde va la ultima parte real de las portadoras espejadas.
N_PREF_CIC	24	Número de portadoras teniendo en cuenta parte real e imaginaria
INICIO_MUESTRAS_PREF	((TAM_CON_PREF)-(N_PREF_CIC))	Índice a partir del cual tomamos las muestras del final que irán en el prefijo cíclico.

Tabla 3-29: Constantes definidas en **main.h** referentes a la IFFT y el prefijo cíclico

Ya que la salida de la IFFT debe ser real, a la entrada nuestras portadoras deben estar dispuestas en un vector con simetría Hermítica, tal y cómo se muestra en la siguiente Figura:

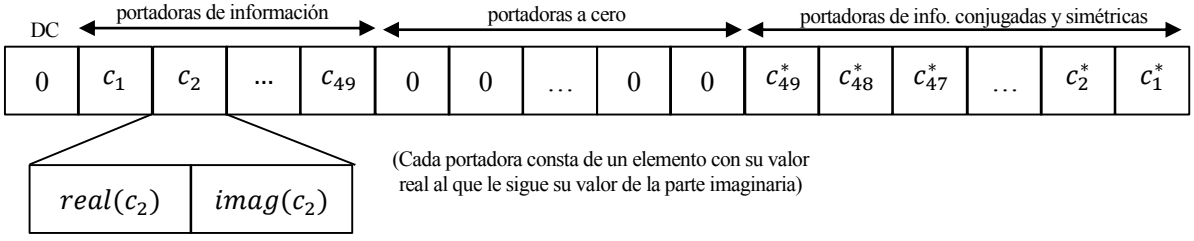


Figura 3-30: Colocación de las portadoras a la entrada de la IFFT

Como el resultado del mapper contiene las portadoras ordenadas incluyendo parte real y parte imaginaria, sólo tendremos que copiarlas en el lugar correspondiente del vector que será la entrada de la IFFT. A partir de aquí, al trabajar con variables en punto flotante, podremos hacer uso de las funciones de la librería ***arm_math.h***. Incluyendo esta librería se pueden utilizar todas las funciones especializadas de procesamiento digital de la señal que incluye la CMSIS-DSP Software Library [25]. Las que utilizamos en este bloque son las siguientes:

Función	Descripción
`void arm_copy_f32 (float32_t *pSrc, float32_t *pDst, uint32_t blockSize)`	Copia el vector al que apunta *pSrc* en *pDst*, ambos de tamaño *blockSize*.
`void arm_cfft_f32 (const arm_cfft_instance_f32 *S, float32_t *p1, uint8_t ifftFlag, uint8_t bitReverseFlag)`	Realiza la FFT compleja sobre el vector apuntado por *p1*.

Tabla 3-31: Funciones de la librería CMSIS-DSP utilizadas.

Además, se ha diseñado la siguiente función propia:

```
 void copia_conj_decre (float32_t *entrada_c,float32_t *salida_c, uint32_t tam);
```

Ésta realiza la copia de los tam/2 números complejos apuntados por *entrada_c* en la dirección *salida_c,* pero conjugando y trasponiendo los datos de forma que la salida es simétrica, tal y cómo hace falta en las portadoras de frecuencias negativas según la Figura 3-30 para cumplir los requisitos de simetría Hermítica. Todo suponiendo que tanto la entrada como la salida está dispuesta de la forma [parte real | parte imaginaria] tal y cómo hemos representado los números complejos a lo largo de toda la aplicación.

3.3.8 Ajuste de los datos para el DAC

Con el objetivo de que los datos que resultan de toda la cadena de bloques del transmisor conformen una señal adecuada para transmitir a través del DAC se hacen necesaria la siguiente secuencia de transformaciones de las que se encarga la función propia ***ajusta_rango_DAC***:

1. Tomar sólo las partes reales, ya que para ello hemos dispuesto la entrada de la IFFT con simetría hermítica (las partes imaginarias resultan ser cero).
2. Saturamos los valores al rango {-0.2, 0.2} para luego multiplicar por una constante para aumentar el rango que queremos a la salida, teniendo cuidado de no sobrepasar el rango del DAC y del ADC.
3. Multiplicamos por $2^{bits\ de\ resolución-1} - 1 = 2047$ al tener configurado el DAC con 12 bits de resolución.
4. Sumamos el valor medio (+2047).
5. Pasamos de variables tipo float a uint16.

Para realizar todo esto, se utilizan las siguientes funciones propias y de la librería CMSIS-DSP:

Función	Descripción
`void arm_scale_f32 (float32_t *pSrc, float32_t scale, float32_t *pDst, uint32_t blockSize)`	Escala el vector al que apunta *pSrc,* de tamaño *blockSize*, por el número de *scale*. El resultado lo deja en *pDst.*
`void arm_offset_f32 (float32_t *pSrc, float32_t offset, float32_t *pDst, uint32_t blockSize)`	Añade el valor de *offset* al vector al que apunta *pSrc,* de tamaño *blockSize*. El resultado lo deja en *pDst.*

Tabla 3-32: Funciones de la librería CMSIS-DSP utilizadas

Función	Descripción
`void arm_copy_f32_reales(float32_t * pSrc, float32_t * pDst, uint32_t blockSize);`	Copia las partes reales del vector de números complejos al que apunta *pSrc,* de tamaño *blockSize*. El resultado lo deja en *pDst.*
`void arm_copy_f32_cast_uint16(float32_t * pSrc, uint16_t * pDst, uint32_t blockSize);`	Copia el vector al que apunta *pSrc,* de tamaño *blockSize*. El resultado lo deja en *pDst,* haciéndole cast a uint32.

Tabla 3-33: Funciones propias utilizadas

4 DISEÑO DEL RECEPTOR

Tras haber diseñado el transmisor siguiendo el definido para la capa física del estándar PRIME (v1.3.6) [18], se diseñará la cadena de bloques análoga en el receptor, haciendo que sean capaces de deshacer el procesado de los datos recibidos. En este aspecto, el estándar deja cierta libertad, por lo que las elecciones que se han ido tomando en este proyecto no son las únicas que podrían funcionar. Aún así, la mayoría de los bloques realizan simplemente el procesado inverso al de su equivalente en el receptor, y cuando no sea el caso, se detallará más minuciosamente su funcionamiento.

4.1 Modelado del receptor en Matlab

El script del receptor toma la variable de los datos recibidos determinada en el main y la hace pasar por los distintos bloques, llamando a cada función correspondiente. Además, se encarga de transformar las matrices de datos para que se corresponda con lo que espera cada función, así como de simular la sincronización de la trama de datos y descartar el prefijo cíclico directamente antes de pasar a llamar a **demod_fft**.

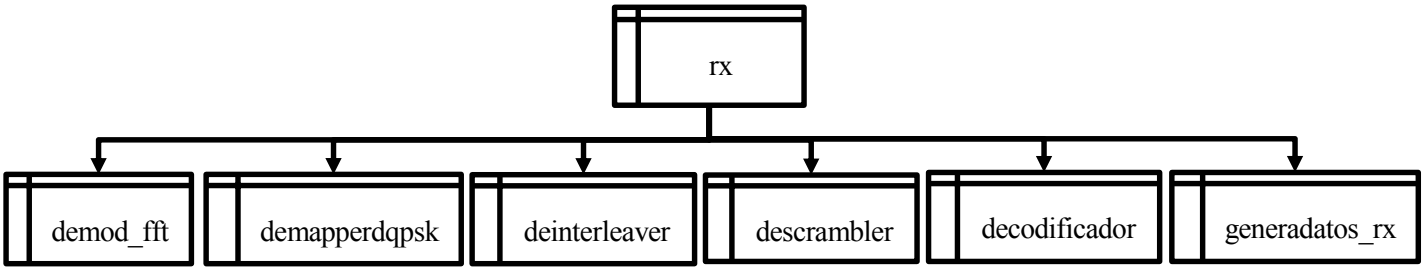


Figura 4-1: Funciones a las que llama el script del receptor

- **demod_fft:** antes de llamarla se descarta el prefijo cíclico y se divide en símbolos OFDM de 128 elementos por lo que lo único que se hace en esta función es pasar la entrada por la función **fft** de Matlab. Tras eso, volvemos a reordenar las frecuencias para que la frecuencia cero sea la central utilizando **fftshift**. Finalmente, tomamos sólo las 49 portadoras de origen para devolverlas a la salida de la función.

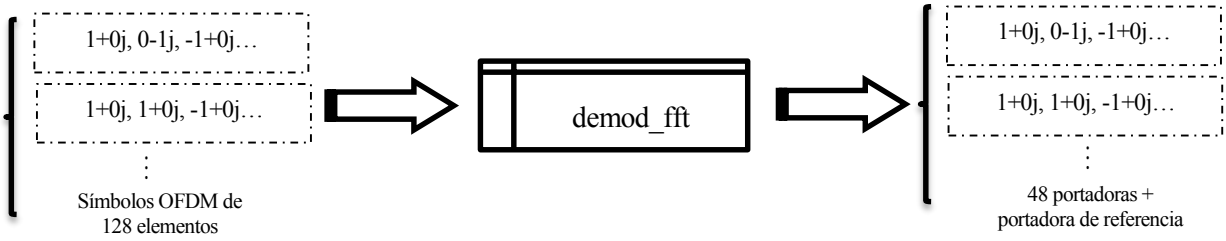


Figura 4-2: Entrada y salida de demod_fft

- **demapperdqpsk**: el demapper toma las 49 portadoras recibidas con el objetivo de obtener la máxima información sobre los bits recibidos. Para ello realizamos las siguientes operaciones:

1. Obtenemos la fase de origen multiplicando el vector de portadoras por sí mismo, conjugado y desplazado en una muestra (ver la ecuación 4-1). Esto lo hacemos ya que, si definimos la fase de la información de origen como $[\ \theta_1\ \theta_2\ \theta_3\ \theta_4\ \theta_5\ \dots]$ y la fase que resulta de la modulación diferencial como $[\ \varphi_0\ \varphi_1\ \varphi_2\ \varphi_3\ \varphi_4\ \varphi_5\ \dots]$, sabemos que $\varphi_n = \varphi_{n-1} + \theta_n\ (con\ n = 1,2,3\dots.)$ siendo φ_0 la portadora de referencia. Por lo tanto, para recuperar la fase de origen, tendremos que obtener $\theta_n = \varphi_n - \varphi_{n-1}$

Vector de portadoras recibido, conjugado y desplazado	$[e^{-j\varphi_0}\ e^{-j\varphi_1}\ e^{-j\varphi_2}\ e^{-j\varphi_3}\ e^{-j\varphi_4}\ e^{-j\varphi_5}\ \dots]$	(4-1)
Vector de portadoras recibido	$[e^{j\varphi_1}\ e^{j\varphi_2}\ e^{j\varphi_3}\ e^{j\varphi_4}\ e^{j\varphi_5}\ e^{j\varphi_6}\ \dots]$	
Resultado de la multiplicación	$[e^{j\varphi_1-\varphi_0}\ e^{j\varphi_2-\varphi_1}\ e^{j\varphi_3-\varphi_2}\ e^{j\varphi_4-\varphi_3}\ \dots]$	

2. Para facilitar la demodulación, a continuación giramos la constelación recibida, multiplicando el vector obtenido en el paso 1 por $e^{j\pi/4}$.

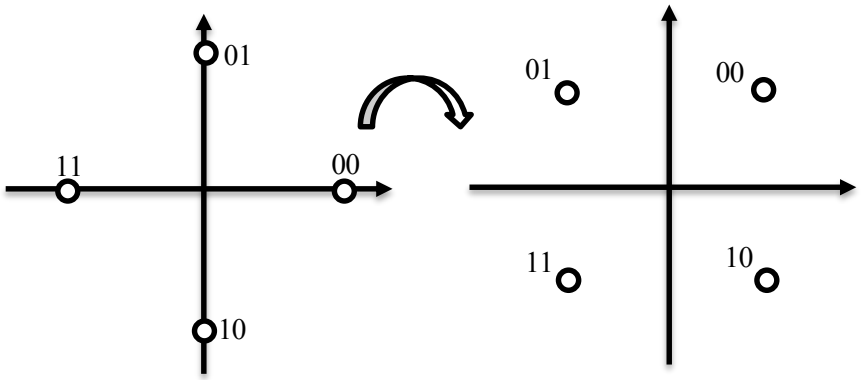


Figura 4-3: Giro de la constelación recibida

3. De esta forma, sólo queda obtener la parte imaginaria de cada portadora que contendrá la información referente al primer bit de información, mientras que la parte real contiene la información sobre el segundo bit. Finalmente, recomponemos el vector de salida, colocando parte real y parte imaginaria de cada portadora alternativamente.

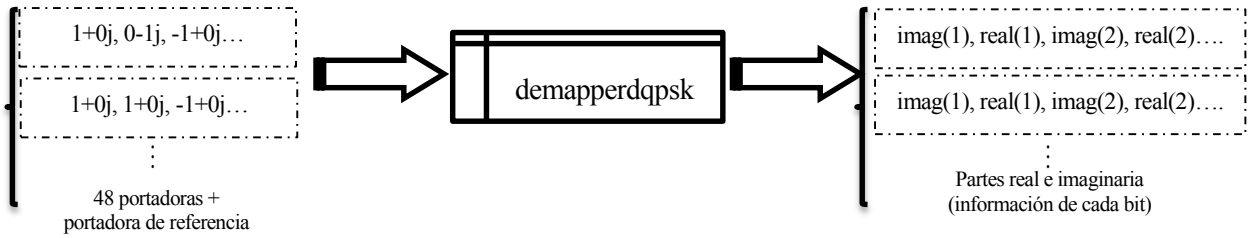


Figura 4-4: Entrada y salida de demapperdqpsk

- **deinterleaver:** de forma análoga a cómo se hace en el transmisor, antes de llamar a esta función, se reordena la matriz de datos, ya que a partir de ahora dejaremos de trabajar sobre los símbolos OFDM para trabajar en bloques de 192 bits. Para ello, tendremos que tomar los datos de cada dos símbolos OFDM para formar un bloque del deinterleaver. El deinterleaver, al igual que el interleaver, simplemente utiliza la función **reshape** de Matlab para reordenar los datos conforme a las matrices

definidas en el estándar PRIME.

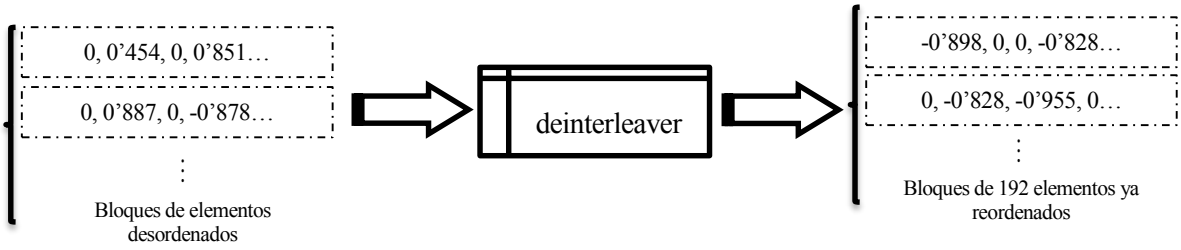


Figura 4-5: Entrada y salida de deinterleaver

- **descrambler**: a esta función le llega un vector continuo formado por los bloques que saca el deinterleaver en serie. Por eso, al igual que se hacía en el scrambler, lo primer que hacemos es dividir el vector completo en trozos de 127 elementos para poder multiplicarlo elemento a elemento con la secuencia pseudoaleatoria definida. En este caso, la XOR con ceros y unos del scrambler se sustituye por un cambio de signo, es decir, por una multiplicación por -1 o 1 según haya un uno o un cero en la secuencia pseudoaleatoria original.

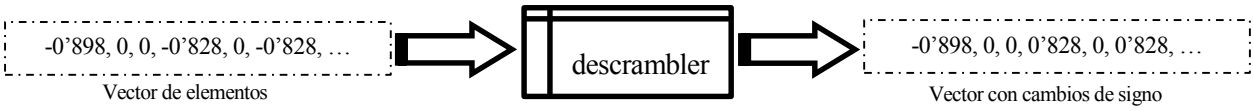


Figura 4-6: Entrada y salida del descrambler

- **decodificador**: en esta función sólo definimos los parámetros que le pasaremos a la función de Matlab **vitdec**. Esta función ya implementa el algoritmo de Viterbi para la decodificación, y con el parámetro **unquant** le indicamos que la entrada no son bits si no valores reales: mientras más cercano sea a 1 más probabilidad de que sea un cero lógico, y de forma análoga con -1 y el 1 lógico. Además, hay que tener en cuenta que con el parámetro **trunc** se le dice al decodificador que devuelva los registros al estado inicial (todo ceros). El decodificador utiliza toda la información redundante para sacar directamente los bits que determina que han sido los transmitidos.

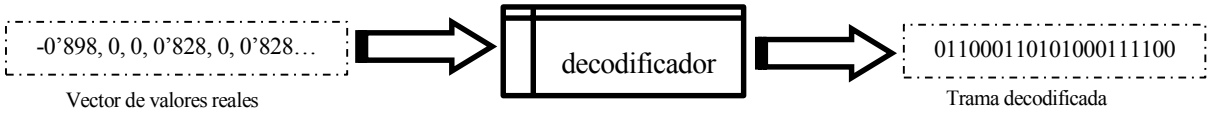


Figura 4-7: Entrada y salida de decodificador

- **generadatos_rx**: finalmente, sólo queda tomar los bytes de información dentro de la trama y transformarlos de nuevo en caracteres, mostrándolos en pantalla para comprobar que es lo mismo que se transmitió.

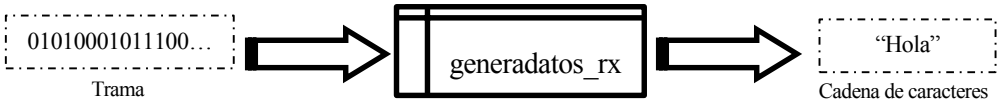


Figura 4-8: Entrada y salida de generadatos_rx

4.2 Implementación del receptor en el microcontrolador

El código de las funciones del receptor se encuentra en **rx.c**. Como ya se ha explicado en la parte del transmisor, ejecutaremos las instrucciones correspondientes a este modo de funcionamiento cuándo tengamos la entrada digital PC6 a nivel alto.

4.3 Visión general

La parte de **main.c** que gestiona el receptor, aparte de llamar a la función **receptor** con los parámetros correspondientes se encarga de ajustar los datos que se reciben a través del convertidor analógico digital. También está preparado para realizar la tarea de sincronizar los datos, así cómo pasar por puerto serie lo que se ha recibido. Para ello hace uso de las siguientes funciones propias:

Función	**Descripción**
`void ajusta_rango_ADC (uint16_t *buffer_entrada, float32_t *buffer_aux, uint32_t tam_datos, float32_t *buffer_salida)`	Ajusta los datos de *buffer_entrada* de forma conveniente para el ADC, dejando el resultado en *buffer_salida.*
`void ADC1_DMA_Timer2Trigger_Init ()`	Configura el convertidor analógico digital, listo para usar la DMA y el Timer 2 como disparo.

Tabla 4-9: Funciones propias utilizadas en el receptor

Por lo tanto, el esquema del receptor queda así:

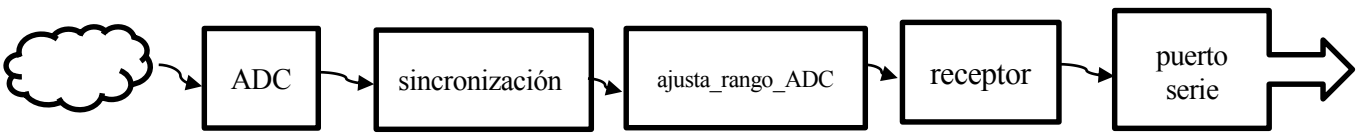


Figura 4-10: Diagrama general del receptor

4.3.1 Sincronización y ajuste de los datos recibidos por el ADC

Para que el receptor pueda identificar el principio de la trama añadiremos un preámbulo compuesto de 32 muestras de un símbolo OFDM elegido de forma aleatoria. El transmisor mandará dicho preámbulo antes de los datos, mientras que el receptor lo tendrá almacenado en memoria, e irá comparándolo con el buffer de sincronización de los datos recibidos por el ADC.

Para ello utilizaremos realizaremos el producto escalar entre nuestro buffer de sincronización dónde va entrando cada muestra que recibiremos por el ADC y el preámbulo guardado en memoria. El resultado de dicho producto escalar alcanzará su máximo cuándo ambas señales sean iguales. Se establecerá un umbral obtenido de forma experimental a partir del cuál, habremos recogido el preámbulo en el buffer de sincronización, y por tanto podremos comenzar a recoger los datos de entrada en el buffer de datos que pasará a la función **receptor**.

Además, se realizarán los pasos intermedios de ajuste de los datos en el orden contrario al que se realizan en el transmisor, con el objetivo de obtener la señal original.

4.3.2 Receptor (receptor)

```
void receptor(float32_t *datos_rx, uint32_t n_simb,float32_t
*caracteres_tx)
```

La función del receptor recibe del main sendos punteros a los datos de entrada y salida, así cómo el tamaño del vector de entrada, cuyo valor estará definido en **main.h** en la siguiente constante:

TAM_CON_PREF_REAL	140	Tamaño de cada símbolo completo con prefijo cíclico, contando solo la parte real.
NSIM	(Ver Tabla 3-26)	Número de simbolos ofdm con N_PORT portadoras.
TAM_BUFFER_DATOS_RX	((NSIM)*(TAM_CON_PREF_REAL))	Tamaño de los datos recibidos sin preámbulo.

Tabla 4-11: Constantes definidas en **main.h** referentes al receptor

Además, en ella se declaran las variables de entrada y salida de cada uno de los bloques intermedios, cada una del tamaño pertinente según las constantes de **main.h**. Sólo destacar que el resultado del deinterleaver incluye todos los bloques procesados, al igual que el del demapper, mientras que que la función ***prefijo_fft***, dentro del bucle ***while*** en el que es llamado, da como salida un solo símbolo cada vez, con objeto de que sea la entrada de ***dqpsk_demap*** sin necesidad de guardar todos los símbolos recibidos.

De esta forma, el funcionamiento es análogo al de la función ***rx*** realizada en Matlab, tal y cómo se detalla en la siguiente Figura:

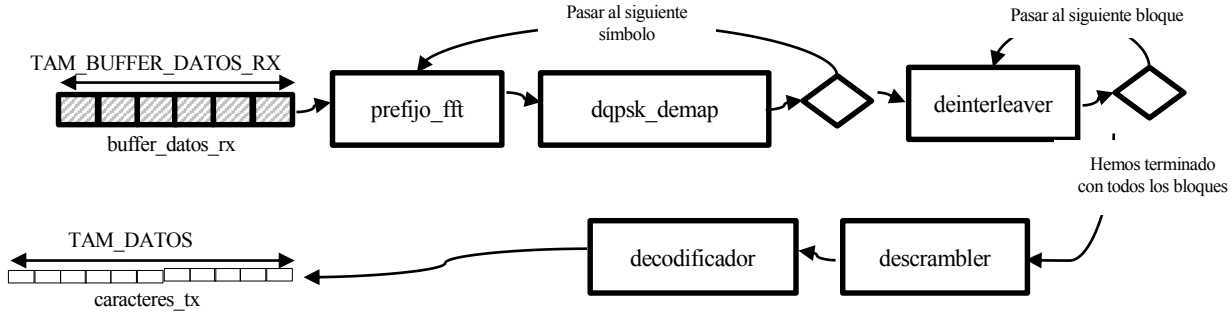


Figura 4-12: Funcionamiento del receptor

4.3.3 Exclusión del prefijo cíclico + FFT

```
void prefijo_fft(float32_t *recibido,float32_t *datos_fft_out);
```

Esta función es el primero en la cadena de bloques por lo que, aparte de recibir como todas las demás el puntero al vector de datos de salida resultante de sus operaciones, recibe el puntero al vector de la entrada al receptor, definido en el main y pasado como parámetro a la función ***transmisor***.

Ignoraremos el prefijo cíclico simplemente tomando el símbolo de entrada a partir de la N_PREF_CIC muestra. A partir de ahí, podemos hacer uso de las función de la librería CMSIS-DSP que realizará la FFT, y

finalmente tomaremos las portadoras de origen que se encontrarán a partir de la muestra IND_PORT. Dichas constantes se encuentran definidas de la siguiente forma en **main.h**.

N_PREF_CIC	24	Número de muestras que contiene el prefijo cíclico, contando parte real e imaginaria.
NFFT_2	256	Número de puntos de la FFT*2, al contar la parte real e imaginaria.
IND_PORT	2	Índice desde donde metemos las portadoras.

Tabla 4-13: Constantes definidas en **main.h** referentes a la FFT y el prefijo cíclico

La función que realiza la FFT es la misma que la de la IFFT, sólo cambiando el parámetro de entrada llamado ifftFlag para ponerlo a 0. De esta forma las funciones de la librería CMSIS-DSP que utilizamos son las siguientes:

Función	**Descripción**
`void arm_copy_f32 (float32_t *pSrc, float32_t *pDst, uint32_t blockSize)`	Copia el vector al que apunta *pSrc* en *pDst*, ambos de tamaño *blockSize*.
`void arm_cfft_f32 (const arm_cfft_instance_f32 *S, float32_t *p1, uint8_t ifftFlag, uint8_t bitReverseFlag)`	Realiza la FFT compleja sobre el vector apuntado por *p1*.

Tabla 4-14: Funciones de la librería CMSIS-DSP utilizadas

De forma inversa a lo que sucede en el transmisor en este caso, al tener a la entrada de la FFT una señal real, obtendremos a la salida un vector de portadoras original con simetría hermítica (ver Figura 3-30). Los números complejos resultantes también estarán dispuestos en la forma [parte real | parte imaginaria].

4.3.4 Demapper

```
void dqpsk_demap(float32_t *portadoras_fft,float32_t *salida_dem);
```

Tras la FFT obtendremos un vector de números complejos que contiene las 48 portadoras de datos y la de referencia. A partir de ellas tendremos que obtener los bits que representan, según el mapeado que hemos realizado en el transmisor utilizando DQPSK. Sin embargo, teniendo en cuenta que al final de la cadena el decodificador de Viterbi realizará una "*soft decision*" tendremos que pasarle un valor real con la información sobre en qué lugar de la constelación está el bit en vez de el bit decidido en sí (0 ó 1).

Aprovechando las funciones de la librería CMSIS-DSP, es posible realizar el procesado análogo al que se realizó en el modelo del receptor de Matlab de la siguiente forma:

1. Sacamos la fase de origen multiplicando el vector de portadoras por sí mismo, conjugado y desplazado en una muestra (ver la ecuación 4-1). Para ello, primero utilizaremos la función ***arm_cmplx_conj_f32*** para obtener el vector de portadoras recibidas conjugado. Justo después, ya podemos multiplicar ambos con ***arm_cmplx_mult_cmplx_f32,*** tomando el vector original desde la muestra número 2.

Función	**Descripción**
`Void arm_cmplx_conj_f32 (float32_t *pSrc, float32_t *pDst, uint32_t numSamples)`	Copia el vector *pSrc* conjugado en *pDst*, ambos de tamaño *blockSize*.
`void arm_cmplx_mult_cmplx_f32 (float32_t *pSrcA, float32_t *pSrcB, float32_t *pDst, uint32_t numSamples)`	Multiplica los vectores de números complejos *pScrA* y *pScrB*, ambos de tamaño *numSamples*, dejando el resultado en *pDst*.

Tabla 4-15: Funciones de la librería CMSIS-DSP utilizadas

2. Para facilitar la demodulación, a continuación giramos la constelación recibida, multiplicando el vector obtenido en el paso 1 por $e^{j\pi/4}$, o lo que es lo mismo, por $\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}j \approx 0.707 + 0.707j$. Para realizar esto se ha modificado la función ***arm_cmplx_mult_cmplx_f32*** dando lugar a ***vectorxnumero_complejo_f32*** que toma uno de los vectores de entrada como un único número complejo, en este caso el ya calculado $0.707 + 0.707j$.

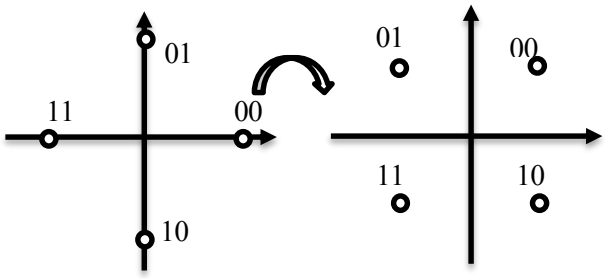


Figura 4-16: Giro de la constelación recibida

3. Tras el giro de la constelación, la información del primer bit de cada portadora queda en la parte imaginaria, mientras que la información del segundo bit queda en la parte real, por lo que debemos reordenar la salida para que al interleaver la información bien ordenada: [info 1er bit | info 2o bit |info 1er bit | info 2o bit |...]:. Para ello, recomponemos el vector de salida, usando la función propia ***intercambia_real_im*** **.**

Función	**Descripción**
`void vectorxnumero_complejo_f32(float32_t * pSrcA, float32_t * pSrcB, float32_t * pDst, uint32_t numSamples)`	Función auxiliar basada en la función ***arm_cmplx_mult_cmplx_f32*** de la CMSIS-DSP Library que multiplica un vector de numeros complejos (*pSrcA*) de tamaño *numSamples* por un único número complejo (*pSrcB*), dejando el resultado en *pDst.*
`void intercambia_real_im (float32_t *entrada_c, float32_t *salida_c, uint32_t tam);`	Función auxiliar que copia el vector entrada_c de números complejos en el vector *salida_c* de forma que la copia tenga los valores de la parte real e imaginaria intercambiados.

Tabla 4-17: Funciones propias utilizadas

Tanto el tamaño de los datos de entrada como el de los de salida están definidos mediante constantes del fichero **main.h**, así como el número de elementos que el demapper debe procesar por símbolo:

N_ELEM_DEMAP	96	Número de elementos que tiene la salida del demapper por símbolo
N_PORT	49	Número de portadoras en un símbolo contando la de referencia
N_PORT_2	98	Número de portadoras teniendo en cuenta parte real e imaginaria

Tabla 4-18: Constantes definidas en **main.h** referentes al demapper

4.3.5 Deinterleaver

```
void deinterleaver (float32_t *datos_deint,float32_t *salida_deint );
```

Como en las funciones anteriores, este bloque recibe punteros a los vectores de entrada y salida. A partir del demapper trabajaremos con variables tipo *float* que almacena la información de cada bit recibido, por lo que el deinterleaver se implementará de manera aún más sencilla que su análogo ya que no tendremos que trabajar a nivel de bit. Guardando los índices que se obtienen de la fórmula definida en el estándar PRIME, sólo vamos intercambiando los elementos del vector *datos_deint* dentro de un bucle ***while.***

Tal y cómo ocurre en el interleaver de bloque del transmisor, se conoce de forma exacta el número de bits que debe procesar y no es necesario que se le pase como parámetro, si no que directamente utilizará la siguiente constante definida en **main.h**:

NBPBLOQ	192	Número de bits por bloque del interleaver.
NBLOQ	(Ver Tabla 4-15)	Número de bloques que recibe el interleaver.

Tabla 4-19: Constantes definidas en **main.h** referentes al tamaño de bloque del deinterleaver

Así, el funcionamiento de este bloque es el siguiente:

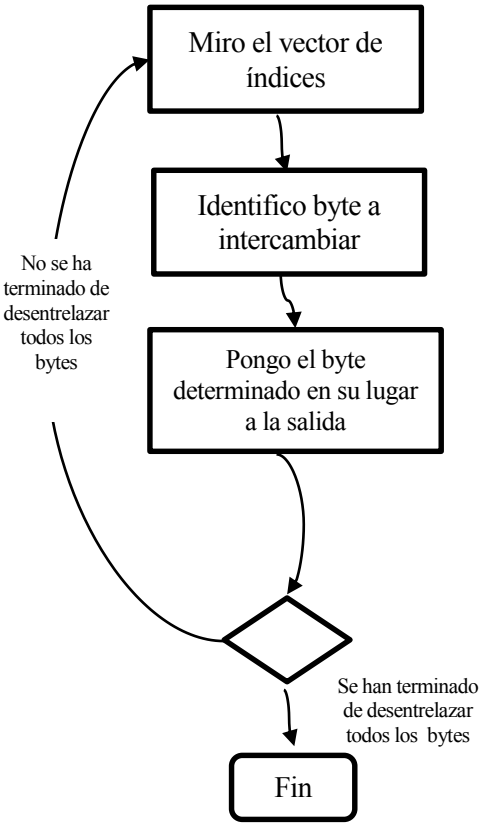


Figura 4-20: Funcionamiento del deinterleaver

4.3.6 Descrambler

```
void descrambler(float32_t *entrada, uint32_t tam_datos, float32_t*
salida_d);
```

Atendiendo a su definición, el descrambler recibe punteros a los vectores de salida y de entrada, así cómo el tamaño de los datos de entrada para poder saber cuándo debe terminar. Este tamaño será la misma constante de **main.h** que define el número de elementos de la salida del deinterleaver (NBPBLOQ*NBLOQ).

De forma análoga a lo que se hace en el transmisor, utilizaremos la secuencia pseudoaleatoria definida en el estándar y la guardaremos previamente en memoria. En este caso, ya que trabajaremos con variables flotantes y no con bits, tendremos que interpretar cada 1 en la secuencia como un cambio de signo, y en cada 0 mantener el signo original. De esta forma, transformamos la XOR por los bits de la secuencia aleatoria por un producto con un vector de 1 y -1.

Tampoco necesitamos concatenarla repetida para conservar la ciclicidad de la secuencia, si no que directamente dispondremos un vector de 127 variables flotantes de valor 1 si en la secuencia original había un 0 y -1 en el caso contrario.

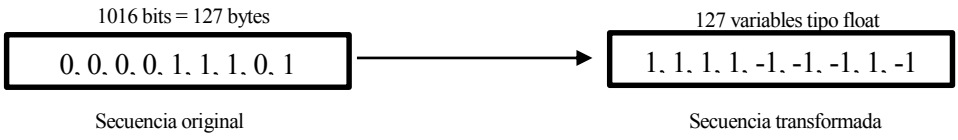


Figura 4-21: Transformación de la secuencia pseudoaleatoria original

Tal y cómo se hace en el descrambler del modelo de Matlab, dividiremos la secuencia de entrada en grupo de 127 elementos para multiplicarlos directamente con ayuda de la función ***arm_mult_f32*** dentro de un bucle ***while***. Luego calculamos los elementos restantes que sobran para multiplicarlos también por la parte correspondiente de la secuencia.

Función	**Descripción**
`void arm_mult_f32 (float32_t *pSrcA, float32_t *pSrcB, float32_t *pDst, uint32_t blockSize)`	Función que multiplica el vector de *pSrcA* por el de *pSrcB*, elemento a elemento dejando el resultado en *pDst*. Todos tienen tamaño *numSamples*

Tabla 4-22: Funciones de la librería CMSIS-DSP utilizadas

4.3.7 Decodificador de Viterbi

Debido a su complejidad, para este último bloque se tomará una función cedida por el GIT del departamento de Ingeniería Electrónica de la ESI que por motivos de confidencialidad no puede ser incluida en este proyecto.

5 DESARROLLO DE LA IMPLEMENTACIÓN DEL PROYECTO

5.1 Plataforma de desarrollo

El componente central del proyecto será el microcontrolador STMF407VG, cuyo core es un ARM Cortex-M4 de 32 bits. Además de contar con las interfaces típicas (I2C, GPIO, SPI, USART…), convertidores analógico-digital y digital-analógico, suficiente memoria (1MB de Flash, 112KB de SRAM,..) y alcanzar hasta 168MHz, ofrece una serie de instrucciones específicas para el procesamiento digital de señales, muy útiles para implementar algunos de los distintos bloques del transmisor y receptor [21-23]

Para trabajar con él, se ha elegido utilizar el kit STM32F4Discovery, una placa de desarrollo que además de incluir dicho microcontrolador, añade un programador y depurador propios (ST-LINK) que permiten cargar los programas fácilmente a través de un mini-USB, y que aparte sirve como suministro de energía. También, en la misma placa contamos con multitud de conexiones con el microcontrolador y los periféricos a través de pines fácilmente accesibles.

Otras de las motivaciones para elegir esta plataforma son que es muy fácil hacerse con ella, ya que está a la venta en la mayoría de tiendas especializadas a un precio ajustado y que, debido a su popularidad, cuenta con una amplia comunidad de soporte en Internet, así como software de desarrollo que facilita la programación y depuración de las aplicaciones. Para hacer más sencillo el diseño del proyecto, se ha optado por implementar de forma separada transmisor y receptor, cada uno en una placa de desarrollo.

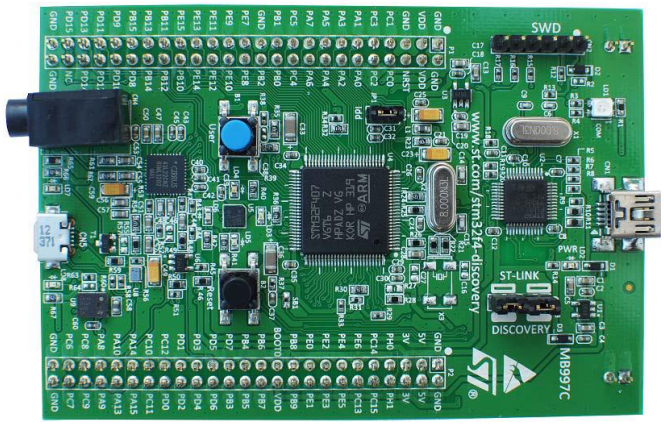


Figura 5-1: Placa de desarrollo STM32F4Discovery.

(Fuente: [21])

Para este proyecto se utilizará principalmente el entorno de desarrollo Keil uVision5, que nos servirá tanto como para editar los ficheros fuente en lenguaje C y compilarlos, así como para cargar y depurar la aplicación:

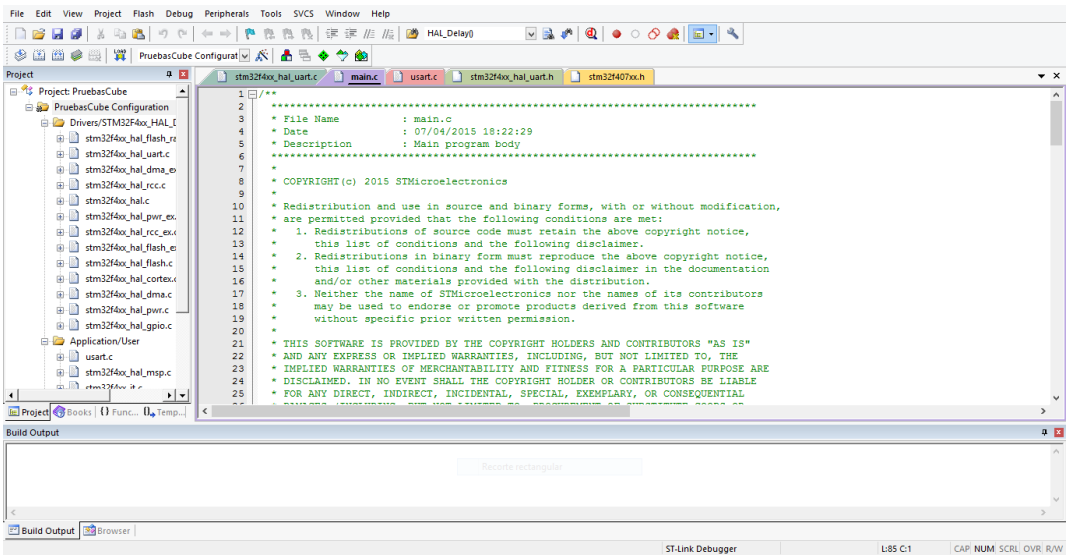


Figura 5-2: Captura del Keil uVision5.

Cabe decir que aunque se esté familiarizado con el lenguaje de programación utilizado, la amplia funcionalidad del ARM y la necesidad de configurarlo todo correctamente y organizar bien los ficheros para que la aplicación funcione, dificulta bastante el primer contacto con la plataforma. Por ello, STM proporciona una herramienta de software con interfaz gráfica llamada STM32CubeMX que automatiza la tarea de crear una estructura de directorios adecuada y ofrece plantillas de los ficheros sobre los que escribir nuestra aplicación. Dicha herramienta ha servido de apoyo al comienzo de la etapa de programación del proyecto, sobre todo en lo que respecta a la configuración y uso de los periféricos.

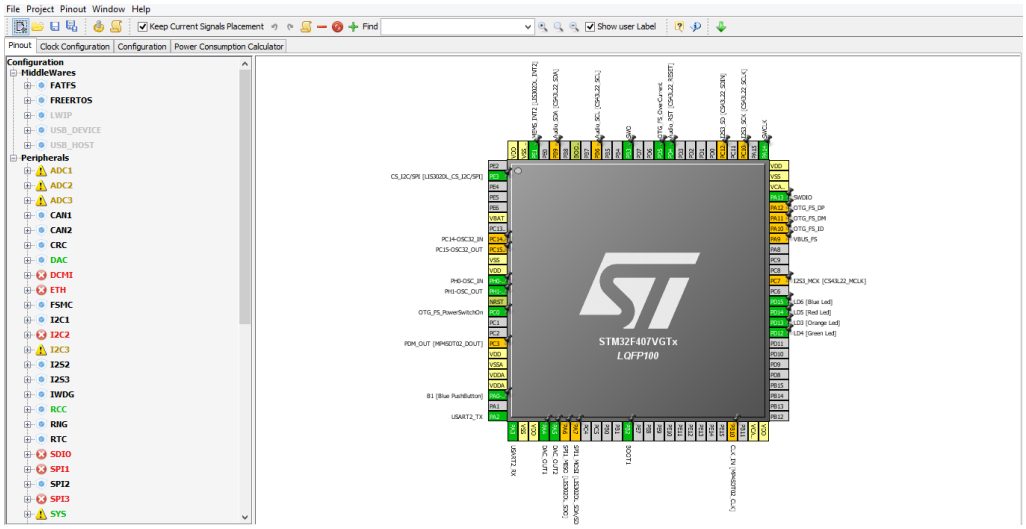


Figura 5-3: Captura del STM32CubeMX.

5.1.1 Frecuencia de reloj del sistema

El primer paso antes de comenzar a configurar los periféricos es controlar la frecuencia de reloj del sistema, ya que si no lo tenemos en cuenta, e ignoramos el esquema de reparto del reloj del sistema entre los distintos periféricos, será imposible determinar con exactitud parámetros tan importantes como la frecuencia de muestreo de los convertidores o la tasa de baudios de la UART.

Para obtener la máxima frecuencia de reloj del sistema tendremos que configurar convenientemente los

registros del RCC, de lo que se hará cargo la función ***config_clk_HSE168().*** En ellos se determinan los valores de todos los prescalers, así cómo el oscilador principal del sistema, que en nuestro caso será el oscilador a 8MHz que incluye la STM32F4Discovery. Sobre todo será importante fijarse en los relojes de los buses de los periféricos, para que luego al configurar cada uno de ellos se conozca a qué frecuencia funcionan.

La función ***config_clk_HSE168()*** modifica los registros de RCC para que el árbol de reloj del sistema quede de la siguiente forma:

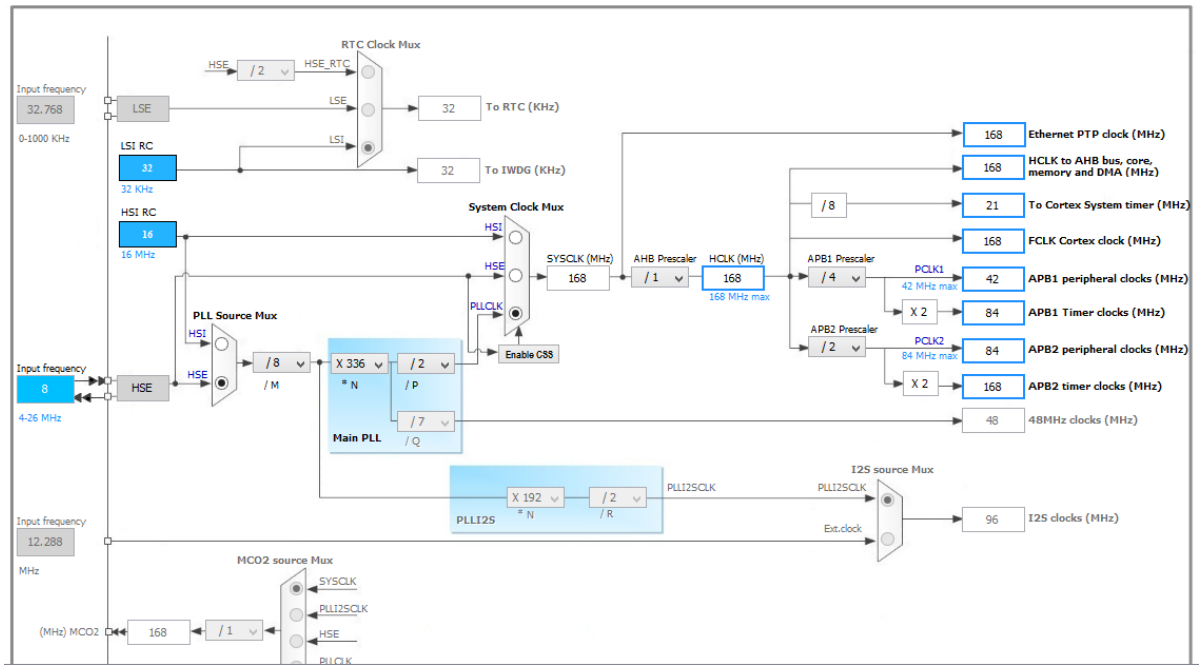
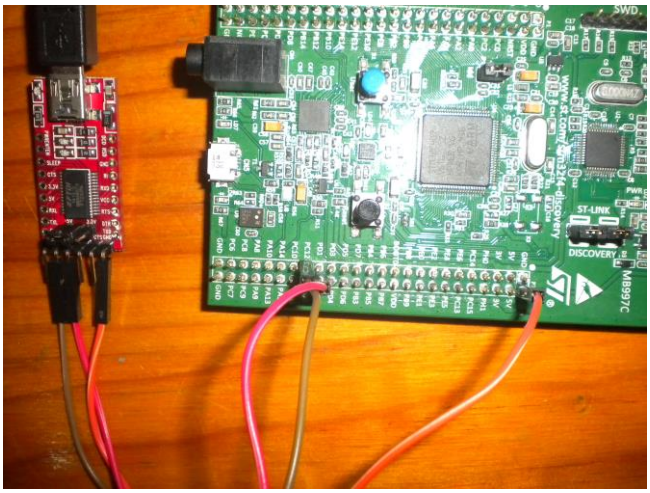


Figura 5-4: Representación del reparto del reloj del sistema que proporciona el STM32CubeMX

5.1.2 UART y comunicación por puerto serie

Antes de integrar la aplicación en el microcontrolador, se ha realizado un modelo del sistema en Matlab, por lo que se hace necesaria una interfaz con el PC para poder comparar el comportamiento entre los dos sistemas y así comprobar que todo funcione correctamente. Para ello se ha utilizado la comunicación vía puerto serie, usando la UART5 del micro y un conversor UART-miniUSB (FTDI232), realizando las conexiones pertinentes tal y cómo se muestra a continuación:



STM32F4	FTDI232
PC12 (UART5 TX)	RX
PD2 (UART5 RX)	TX
GND	GND

Figura 5-5: Conexionado entre la placa de desarrollo y el conversor FTDI232

La configuración del puerto serie en este caso es la siguiente:

- 9600 baudios
- Sin bit de paridad
- 8 bits de datos
- 1 bit de parada
- Sin control de flujo

A través de una función de librería propia configuramos estos parámetros en la UART5 del microcontrolador. Además, también se disponen las funciones que mandarán por el puerto serie los datos.

Para observar lo que se recibe por puerto serie se ha optado por utilizar la terminal del programa Realterm [24] ya que permite guardar en fichero de forma muy cómoda los datos. Además de eso, también se ha implementado un script en Matlab (**pruebas_bloques_fichero.m**) que lee dichos ficheros y manipula lo leído de forma que podamos comparar con las salidas generadas por Matlab de forma automática.

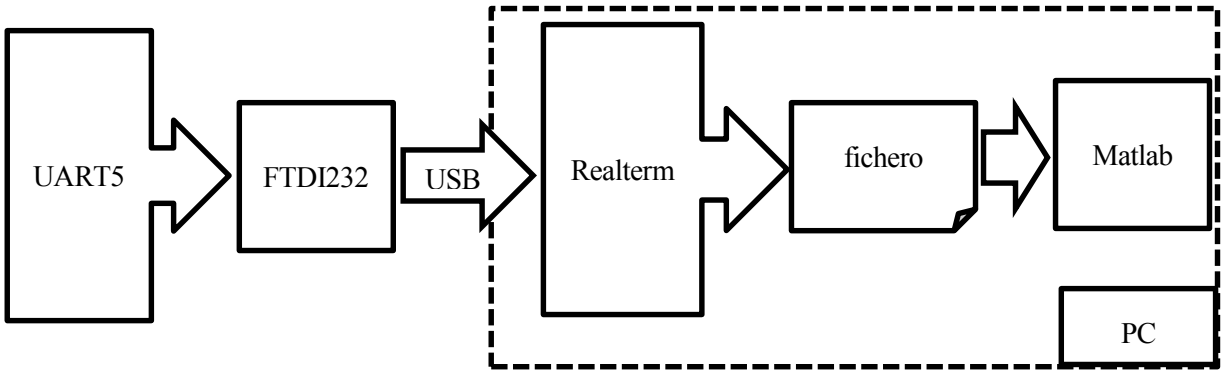


Figura 5-6: Diagrama de bloques de la transmisión por puerto serie y la comprobación de los archivos.

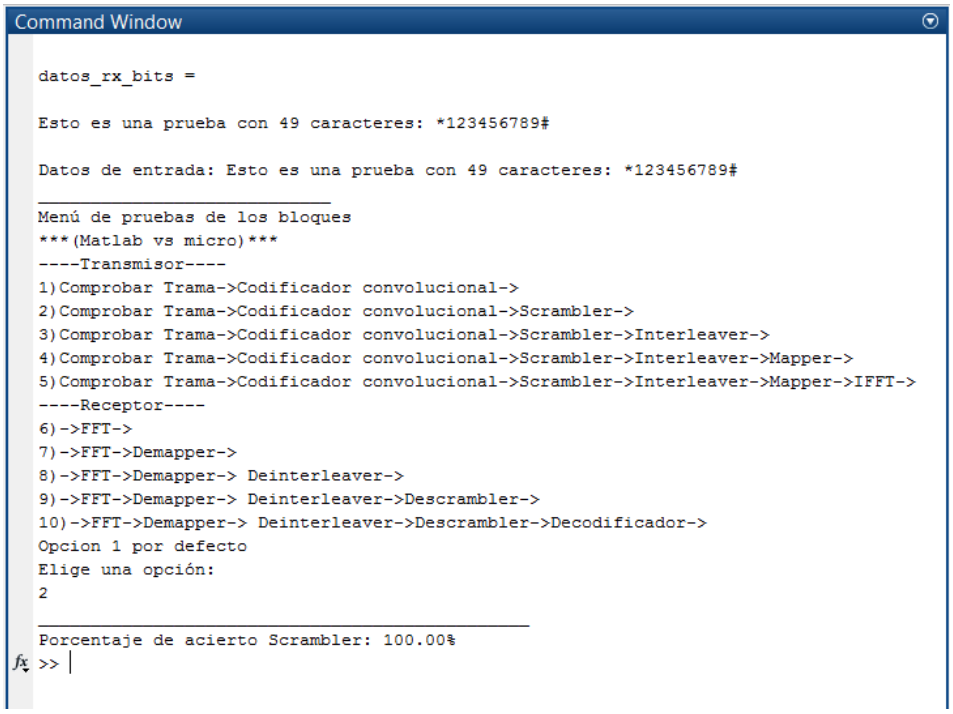


Figura 5-7: Captura de la ejecución del script que comprueba los ficheros de salida

5.1.3 Convertidores analógico/digital y digital/analógico

Se separará transmisor y receptor en placas distintas, y se utilizará como medio de comunicación entre ambos los convertidores digital/analógico (DAC) y analógico/digital (ADC) respectivamente. De esta forma, el transmisor, tras procesar los datos de entrada, los mandará a través del DAC para que lleguen directamente (estarán conectados a través de un cable) al ADC del receptor, que tras sincronizar la trama, recogerá la información recibida para procesarla a través de su cadena de bloques.

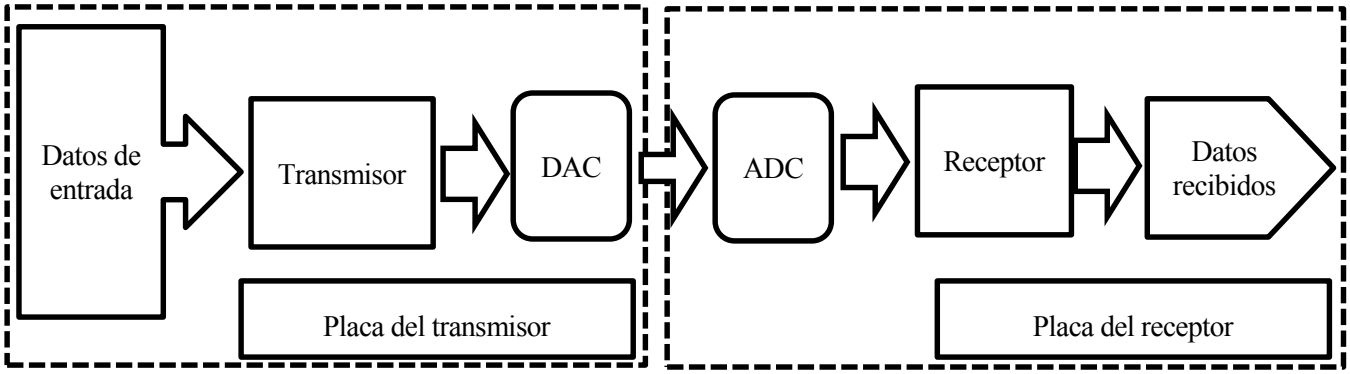


Figura 5-8: Diagrama completo del sistema

Configuraremos los convertidores para que utilicen la DMA para pasar directamente los datos a memoria, y así no tener que encargar esa tarea al procesador. Además, estableceremos en ambos el disparo a través de los temporizadores disponibles, siendo posible de esta forma controlar la frecuencia de muestreo configurando el valor máximo que cuentan. De forma más detallada, la configuración y el funcionamiento del DAC es el siguiente:

Periférico	**Descripción**
DAC1	• Salida en pin 4 del puerto A • Canal 1 • Timer 6 como disparo (en el flanco de subida) • Uso de la DMA • Resolución de 12 bits • Alineación de los datos a la derecha
Timer 6	• Cuenta desde 0 hasta 1344 (la frecuencia de muestreo será de 62.5KHz) • Sin prescalado del reloj • Disparo del DAC1
DMA	• DMA1, Canal 7, Stream5. • Dirección de los datos: desde la memoria al periférico • Dirección de memoria: vector de datos resultado de la salida del transmisor. • Dirección del periférico: dirección del DAC1. • Puntero de la dirección a memoria que se incrementa

	• Puntero a la dirección del periférico que no se incrementa • Modo circular (al terminar de recorrer el vector de la memoria vuelve al principio y comienza de nuevo)

Tabla 5-9: Resumen de la configuración de los periféricos en el transmisor.

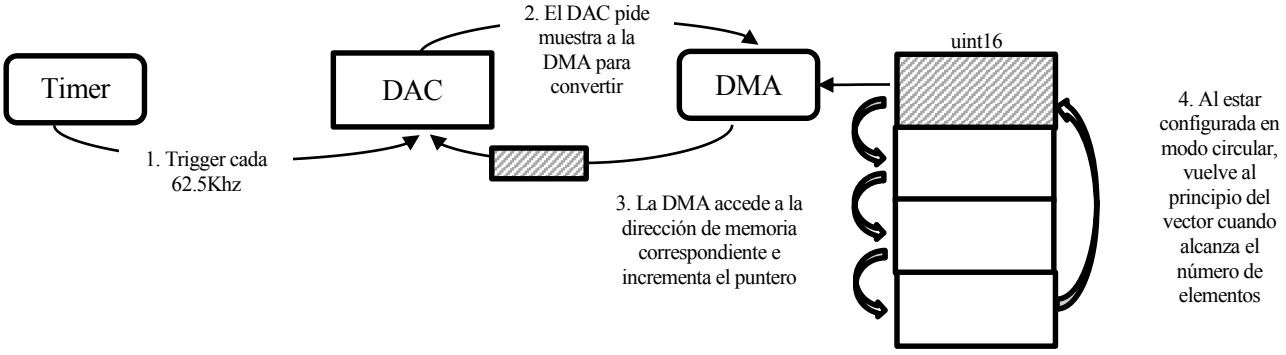


Figura 5-10: Esquema de funcionamiento del DAC según la configuración

Por otro lado, la configuración y el funcionamiento del DAC es el siguiente:

Periférico	**Descripción**
ADC1	• Salida en pin 1 del puerto A • Canal 1 • Timer 2 como disparo (en el flanco de subida) • Uso de la DMA • Prescalado de /2 del reloj • Resolución de 12 bits • Alineación de los datos a la derecha • Peticiones continuas de la DMA activadas
Timer 2	• Cuenta desde 0 hasta 1343 (la frecuencia de muestreo será de 62.5KHz) • Sin prescalado del reloj • Disparo del ADC1
DMA	• DMA2, Canal 0, Stream0. • Dirección de los datos: desde el periférico a la memoria • Interrupción de transferencia completa activada. • Dirección de memoria: variable de entrada (después se copiará desde ahí a los datos para sincronizar o a los de entrada al receptor, según toque)

	• Dirección del periférico: dirección del ADC1. • Puntero de la dirección a memoria que no se incrementa • Puntero a la dirección del periférico que no se incrementa • Modo circular DESACTIVADO

5-11: Resumen de la configuración de los periféricos en el receptor

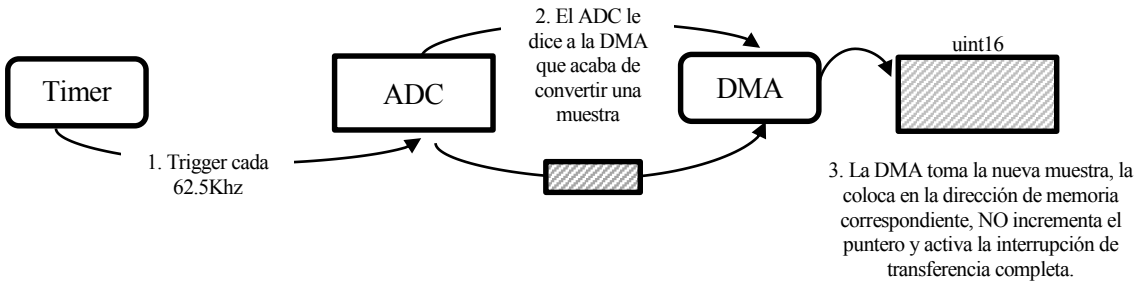


Figura 5-12: Esquema de funcionamiento del ADC según la configuración

Cuándo detectemos la interrupción de transferencia completa de la DMA en el **main,** decidiremos en qué buffer insertamos la nueva muestra dependiendo de si estamos intentando sincronizar o no.

5.2 Depuración y pruebas de funcionamiento

5.2.1 Comprobación del modelo de Matlab

La primera prueba que se ha realizado es la de coherencia entre transmisor y receptor en el modelo de Matlab. Esto significa que cada bloque del receptor debe realizar la función opuesta a su homólogo en el transmisor, por lo que si todo está bien, deberemos obtener a la salida del receptor los datos que recibe de entrada el transmisor. Para ello, siendo conocida la secuencia de datos de entrada y tratándose en este caso de caracteres ASCII, basta con imprimir en pantalla el resultado del receptor y comprobar que es la misma:

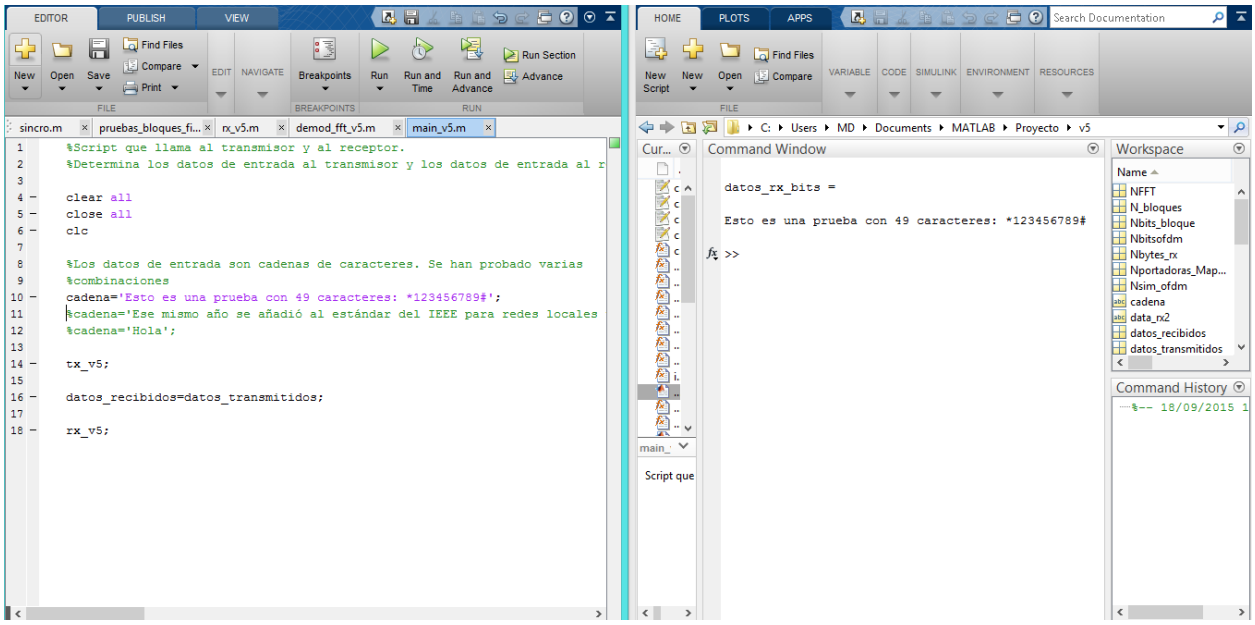


Figura 5-13: Prueba de funcionamiento del modelo de Matlab

5.2.2 Comprobación del modelo de Matlab vs modelo hardware

Para realizar las comprobaciones entre los bloques de Matlab y el microcontrolador, sin tener en cuenta la sincronización y los convertidores, se ha seguido el procedimiento siguiente:

1. Definir una misma entrada para la función en C y en Matlab. En este caso, se mantienen transmisor y receptor en la misma placa, con el objetivo de que la misma variable de salida del transmisor sea la variable de entrada del receptor.
2. Guardar la salida binaria que genera el bloque hardware en un fichero de texto utilizando el programa Realterm.

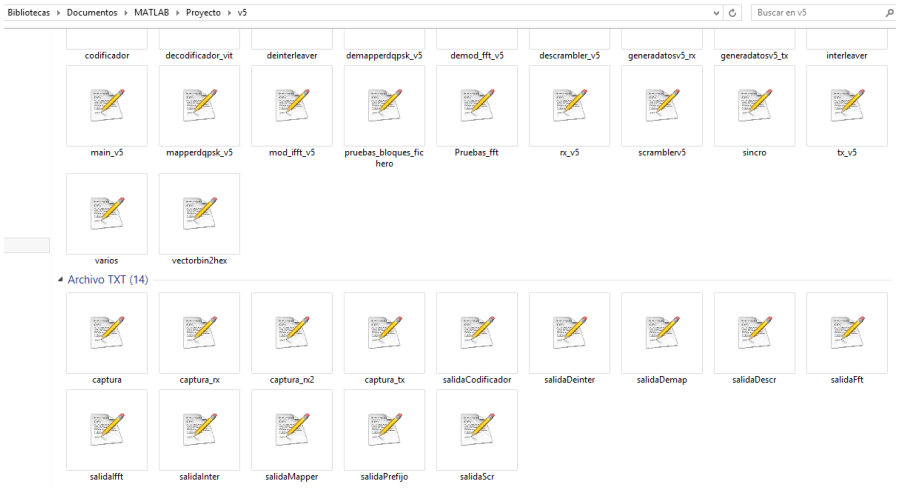


Figura 5-14: Ficheros de salida de cada bloque

3. Con ayuda del script de Matlab **pruebas_bloques_fichero.m**, se ejecuta el modelo en Matlab con la entrada definida, así cómo se manipulan los datos binarios del fichero para poder compararlos con las variables de salida de Matlab. Para ello, entre otras cosas, se hace uso de las funciones que manejan ficheros (**fopen**, **fclose**) así cómo de la que los lee, tomando el formato según le indiquemos (**fread, fscanf**). Dependiendo del tipo de variable con el que tratásemos en el modelo hardware, se ha preferido pasar los datos desde hexadecimal en un paso intermedio, o simplemente leerlos con formato float, uint8…

4. Comparar las salidas de ambos sistemas y representar si es necesario. Cabe destacar que la verificación en este caso es acumulativa: al comprobar un bloque, tenemos en cuenta toda la cadena de bloques anterior, ya que la entrada elegida es sólo parámetro del primero. Así, además de asegurar el funcionamiento individual, ha sido posible detectar fallos en la interconexión y la transformación que requieren los datos al pasar de uno a otro. A continuación se muestran dos posibles ejemplos de la comprobación del bloque Mapper y Descrambler. Este último es el final de la cadena, por lo que verifica todo el conjunto de bloques:

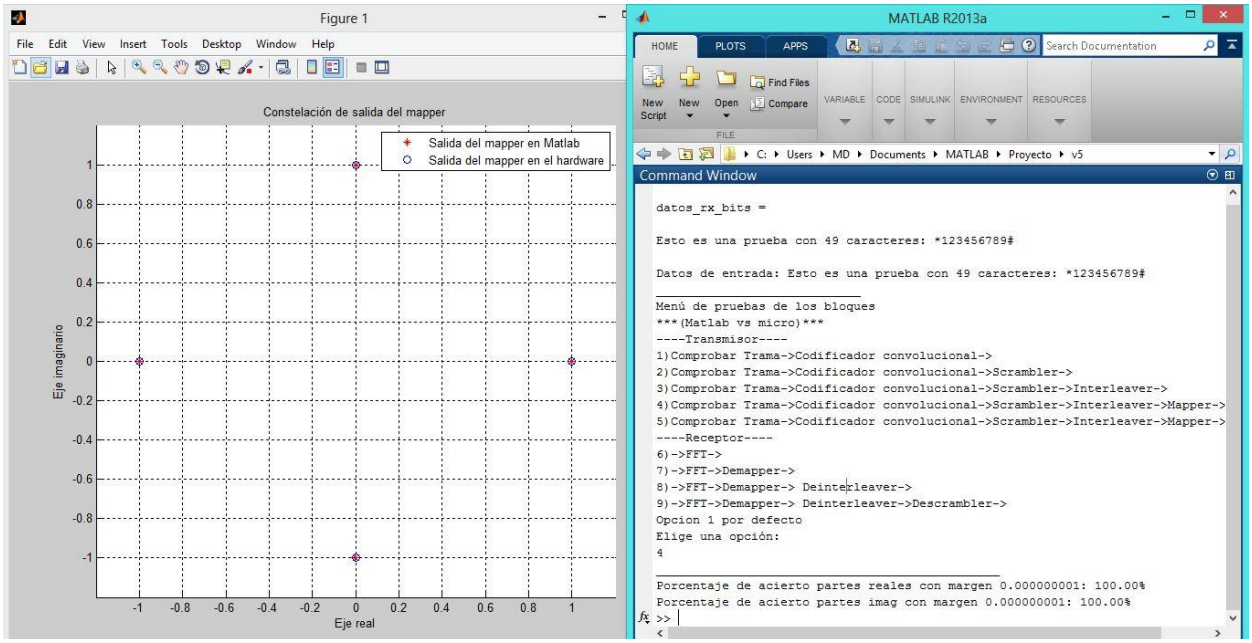


Figura 5-15: Comprobación de la salida del mapper en Matlab vs hardware

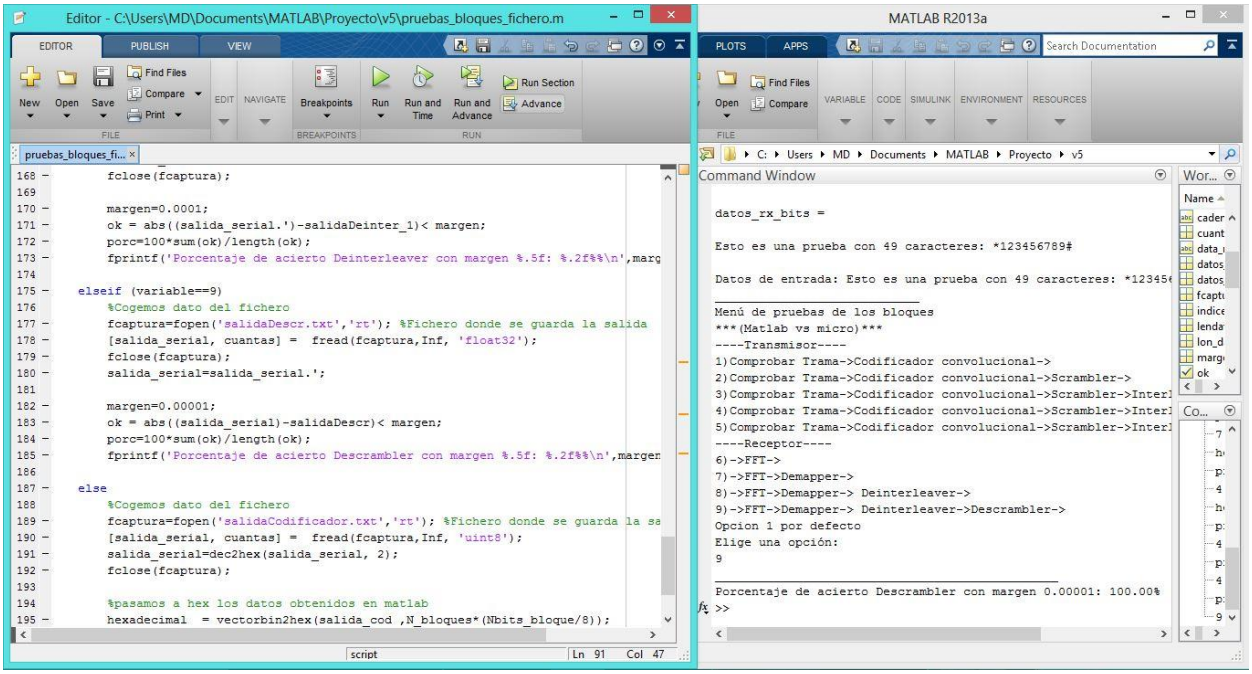


Figura 5-16: Comprobación de la salida del descrambler en Matlab vs hardware

5.2.3 Comprobación de la sincronización y los convertidores

Por último, con el objetivo de verificar el buen funcionamiento de la sincronización y los convertidores, realizamos las siguientes comprobaciones:

1. Capturamos el buffer de salida del DAC en el transmisor y comprobamos efectivamente a través de Matlab y con el osciloscopio que se trata de una señal OFDM:

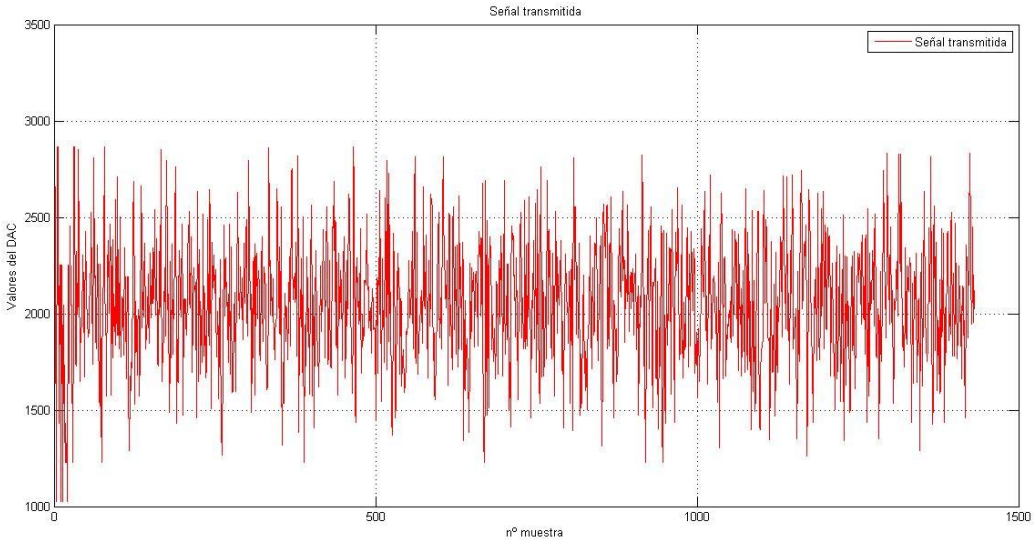


Figura 5-17: Buffer de salida del DAC

2. Hacemos pasar la señal que hemos sacado del DAC por la FFT del receptor para comprobar que la constelación que transmitimos es la correcta:

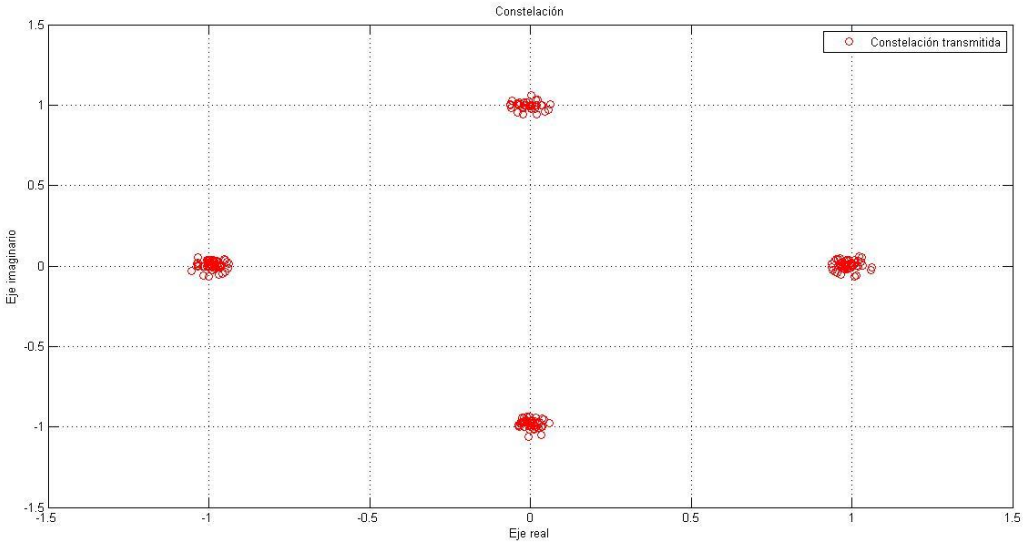


Figura 5-18: Resultado de la FFT de los datos que se transmiten por el DAC

3. Simulamos la sincronización del receptor a través de Matlab: tomamos los datos recibidos a través del ADC, le hacemos la correlación cruzada con el preámbulo elegido y buscamos el máximo, que será el número de muestra dónde ambas señales coinciden:

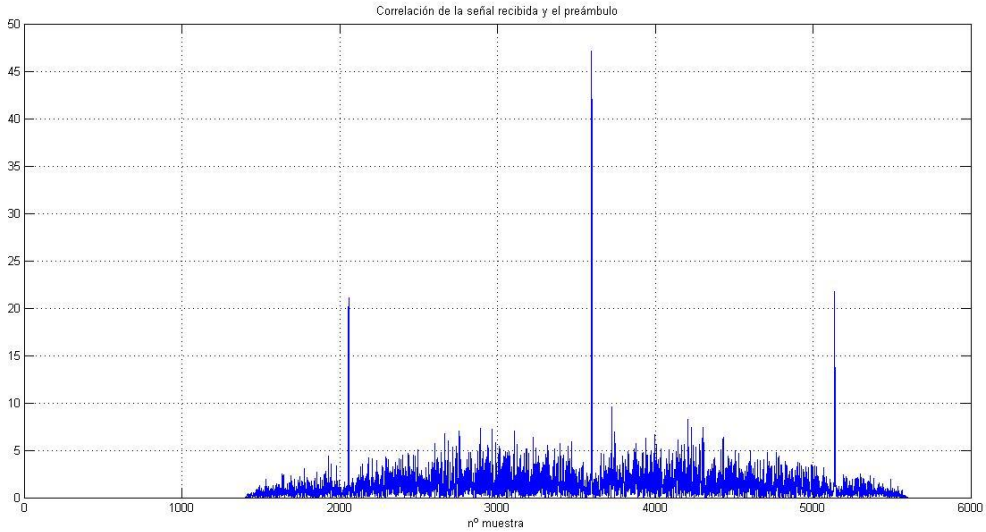


Figura 5-19: Ejemplo de autocorrelación de la señal que se toma a través del ADC con un preámbulo predefinido

4. Sincronizamos las señales ajustándolas según nos ha indicado la correlación en Matlab. Esto indica que el funcionamiento de los convertidores es el esperado, y podemos llegar a sincronizar la señal. Al igual que antes, con la señal recogida del DAC, comprobamos que la constelación que recibiría el modelo de Matlab utilizando la señal que recoge el ADC es correcta. Como se puede ver, existe un pequeño offset entre el DAC y el ADC:

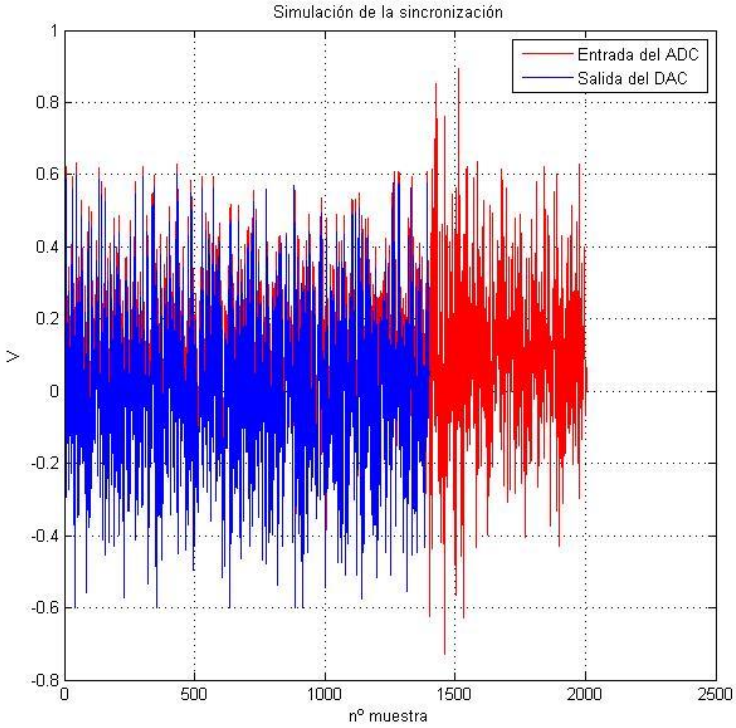


Figura 5-20: Sincronización manual de las señales correladas en la figura anterior

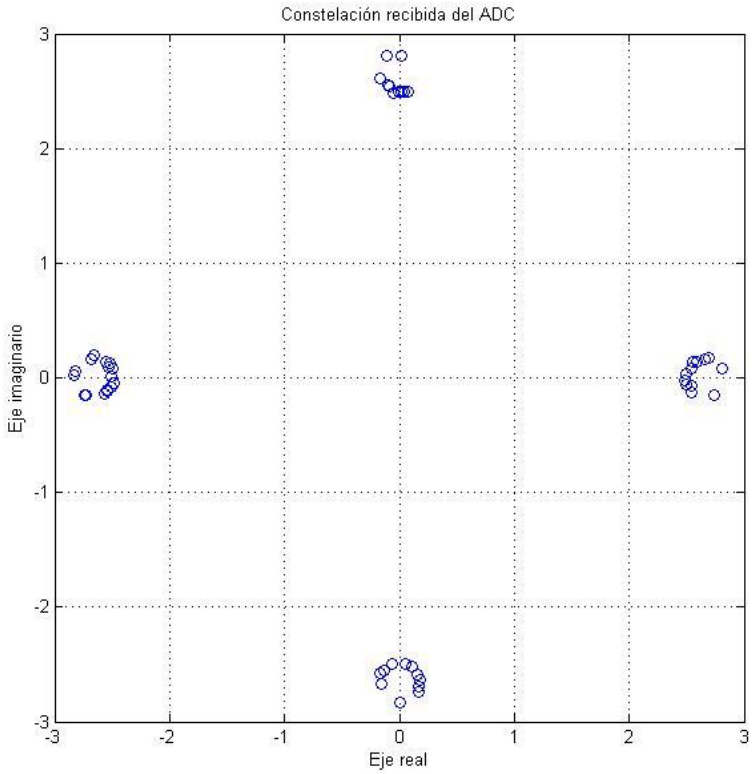
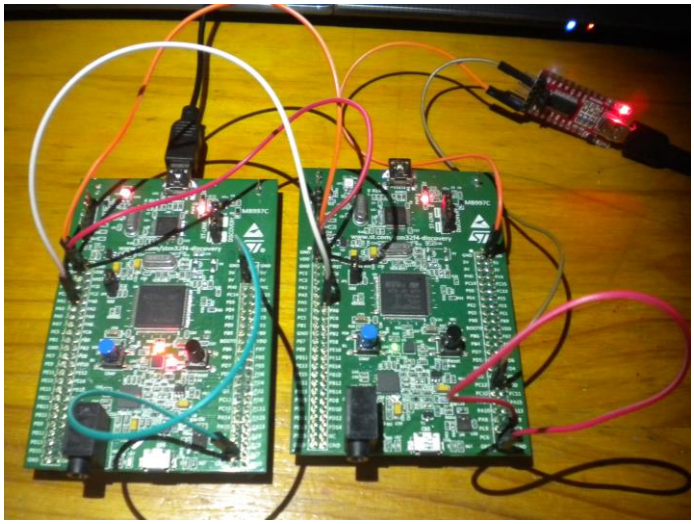


Figura 5-21: Resultado de la FFT de la señal sincronizada manualmente

5. Tras confirmar que con la configuración de los convertidores es posible sincronizar la señal, finalmente, conectamos transmisor y receptor hardware a través de sus respectivos pines para comprobar que la sincronización implementada sea correcta:



Transmisor	Receptor
PA4 (DAC1 OUT)	PA1 (ADC1 IN)
VDD	VDD
GND	GND

Figura 5-22: Conexión entre transmisor y receptor

6. Se comprueba que cuando el receptor haya sincronizado, en el buffer dónde realizamos la correlación se encuentre el preámbulo transmitido, en este caso de 32 muestras:

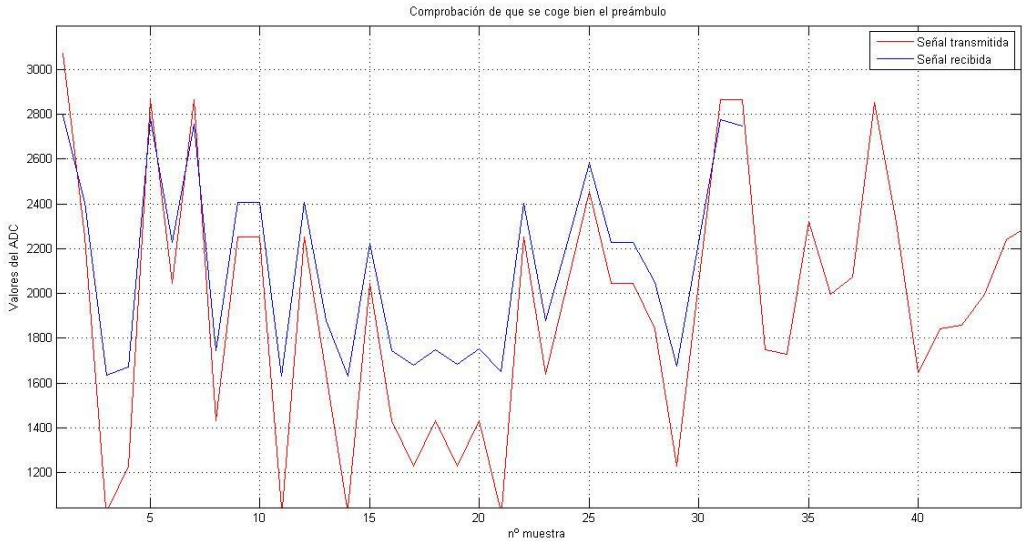


Figura 5-23: Comprobación de la correcta sincronización del preámbulo

7. Tras captar el preámbulo, se pasa a guardar los datos en otro buffer distinto. Comprobamos la similitud entre el buffer de entrada al DAC y los datos recibidos en el ADC al sincronizar:

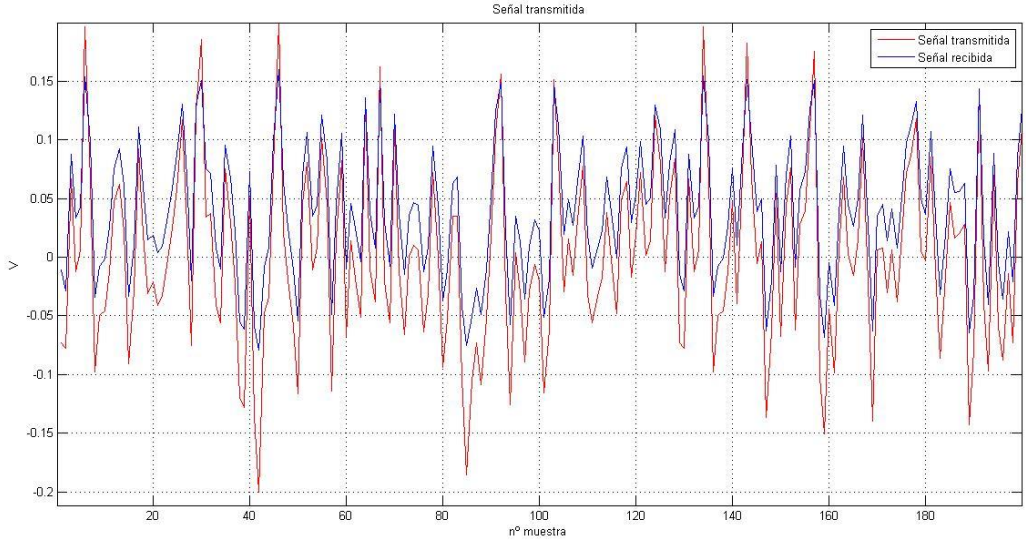


Figura 5-24: Detalle de la sincronización temporal de las 200 primeras muestras de datos

8. Sabiendo que la entrada del receptor hardware está totalmente sincronizada, se puede comparar la salida de cada bloque con el modelo en Matlab. En primer lugar, se comprobará la constelación de salida de ambos demapper, resultando de la siguiente forma:

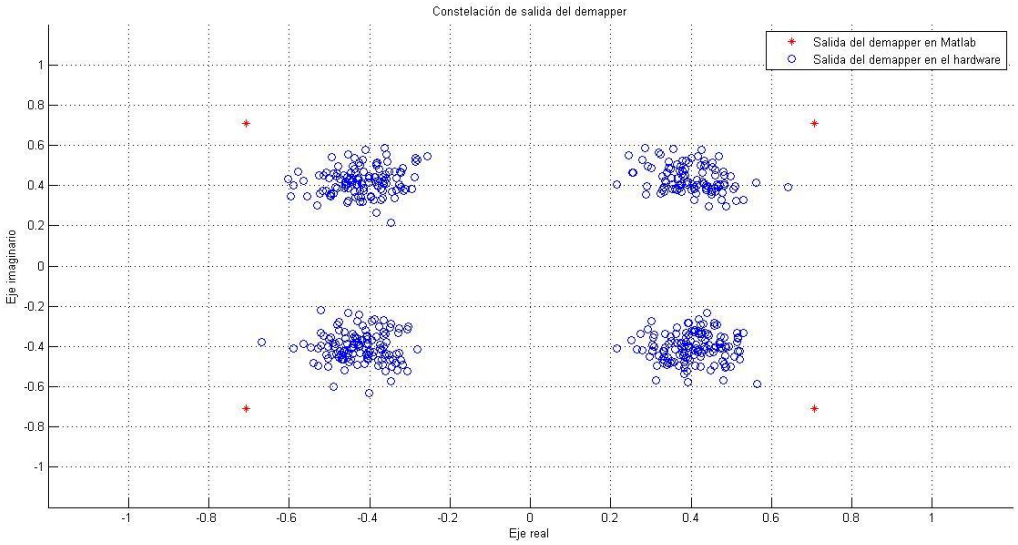


Figura 5-25: Constelación de salida de ambos demapper

Al observar esta figura hay que tener en cuenta el procesado al que hemos sometido a la señal en transmisor y receptor hardware para que se ajustara al rango de los convertidores, y que en Matlab no se ha modelado, de ahí la diferencia de potencia entre ambas constelaciones.

También es posible contrastar la salida del deinterleaver y del descrambler. Aunque la señal entrante esté atenuada y un poco degradada debido al paso por los convertidores, se puede apreciar que cada bloque realiza su función perfectamente al barajar y entrelazar las muestras de la señal:

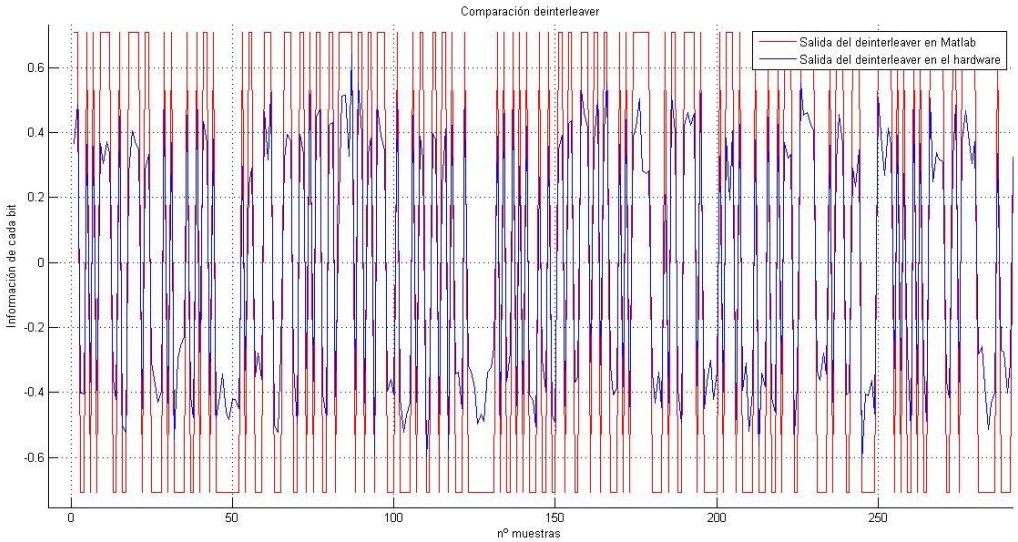


Figura 5-26: Comparación de la salida de los deinterleaver

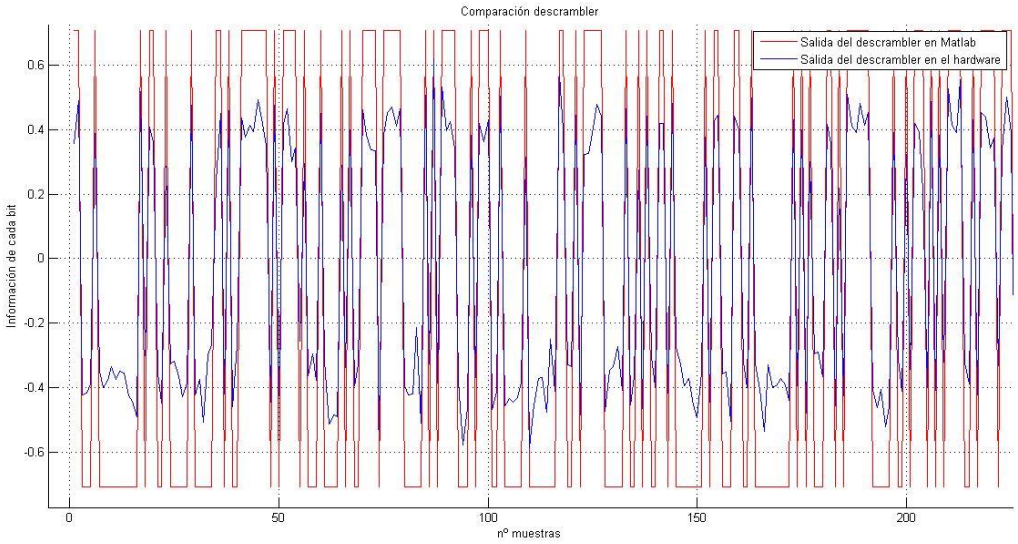


Figura 5-27: Comparación de la salida de los descrambler

De todas estas pruebas se concluye que el sistema hardware funciona de la misma forma que el sistema en Matlab, pero hay que tener en cuenta cómo afecta el uso de los convertidores al rango de la señal para la comparación entre ambos. Aunque la señal real recibida se ha degradado un poco, es posible demodular la señal correctamente. Además, hay que tener en cuenta que el último bloque que aquí no se ha incluído es el decodificador, que utilizará toda la información redundante que hemos incluido para corregir los errores que se produzcan.

6 RESULTADOS Y CONCLUSIONES

De forma resumida, y tal cómo se ha ido detallando en la memoria, las tareas y objetivos que se han ido ejecutando a lo largo de este proyecto son los siguientes:

- Diseño a alto nivel y modelado en Matlab de un sistema completo OFDM, basado en el estándar PRIME y comprendiendo toda la cadena de bloques de transmisor y receptor: codificador convolucional, scrambler y descrambler, interleaver y deinterleaver, mapper y demapper DQPSK, FFT/IFFT,… Para ello, se han usado en la mayoría de los bloques funciones totalmente propias, y en alguno determinado (decodificador y codificador por ejemplo) se ha aprendido a utilizar las funciones del DSP System Toolbox de Matlab.
- Preparación de un procedimiento de verificación a través de un script de Matlab del modelo hardware utilizando como referencia el sistema ya implementado en Matlab. Para ello, se ha tenido que configurar la UART del microcontrolador para poder recibir las salidas de cada bloque a través de ficheros con ayuda del programa Realterm. También ha sido de gran importancia el procesado de los datos que se imprimían dependiendo del formato de las variables con el que se trataba para poder comparar ambos sistemas.
- Finalmente, se ha llevado a cabo la implementación de todos los bloques descritos (excepto el decodificador de Viterbi) en el microcontrolador, comprobando su correcto funcionamiento a través de la UART. Para ello se ha intentado en la medida de lo posible hacer uso de las funciones específicas que proporciona el fabricante para mejorar el rendimiento y realizar procesado de señal. Además, se han configurado y utilizado los convertidores analógico/digital y digital/analógico, comprobando su funcionamiento real a través del osciloscopio y Matlab.

Cabe destacar que a nivel personal la realización de este proyecto ha contribuido a consolidar y ampliar los conocimientos adquiridos sobre programación de microcontroladores, procesamiento de señal y OFDM. Sobre todo, es notable la oportunidad de tomar contacto con una plataforma tan ampliamente utilizada en la actualidad y de tanta potencia como los microcontroladores Cortex de ARM. El haber tenido que abarcar desde la programación a nivel de bit, hasta utilizar las funciones específicas de las librerías CMSIS para mejorar el rendimiento, así cómo la obligación de profundizar en el funcionamiento de cada uno de los periféricos (UART, ADC, DAC, DMA, temporizadores…) para poder configurarlos y usarlos ha resultado ser la parte más instructiva y también de las más complejas.

Aunque ya en la carrera se había tomado contacto con el sistema OFDM basado en PRIME, el haber tenido que diseñar e implementar tanto receptor y transmisor de un sistema completo en Matlab y en hardware, ha ayudado a comprender mejor el funcionamiento y objetivo de cada uno de los bloques que lo componen. Concretamente, el analizar las diferencias entre el sistema modelado en Matlab y el real en la fase de comprobación, representando y verificando las salidas de cada bloque y de los convertidores, ha servido de demostración de los posibles problemas a los que hay que enfrentarse cuándo se implementa un sistema como éste en la realidad. También ha sido muy ilustrativo el poder visualizar de forma totalmente práctica la transmisión de OFDM a través del osciloscopio, demostrando la validez de los conocimientos teóricos que se habían aplicado.

6.1 Futuras líneas de trabajo

Como posible ampliación y continuación del trabajo realizado en este proyecto, se plantea principalmente el culminar el sistema de transmisión Li-Fi DCO-OFDM diseñando y fabricando las etapas de potencia que servirán de interfaces con el led del transmisor y el fotodiodo del receptor, así cómo analizando la

problemática y complejidad que añaden a nuestro sistema dichos elementos y la intervención del canal óptico.

En cuanto a las mejoras que pueden realizarse respecto al diseño realizado, sería conveniente analizar las distintas técnicas posibles de sincronización del receptor para implementar la más adecuada a nuestro canal. También sería interesante incluir estimación de canal o comparar el rendimiento del uso de otras constelaciones. Además, se podría mejorar el rendimiento del código y del diseño para poder así plantear un sistema que en lugar de trabajar a ráfagas se ejecute de forma continua en tiempo real, ayudando a adquirir conocimientos avanzados sobre la programación del microcontrolador Cortex-M4.

ANEXO I: CÓDIGO IMPLEMENTADO EN MATLAB

main_v5.m

```
%Script que llama al transmisor y al receptor.
%Determina los datos de entrada al transmisor y los datos de entrada al
receptor.

clear all
close all
clc

%Los datos de entrada son cadenas de caracteres. Se han probado varias
%combinaciones
cadena='Esto es una prueba con 49 caracteres: *123456789#';
%cadena='Hola';

tx_v5;

datos_recibidos=datos_transmitidos;

rx_v5;
```

tx_v5.m

```
%Script que llama a todas las funciones de los bloques del transmisor
%Consta de:
%--Definición de los parámetros comunes al transmisor y receptor
%1-Generacion de stream de datos binarios a partir de cadena de caracteres
%2-Codificador convolucional
%3-Scrambler
%4-Interleaver
%5-Mapper DQPSK
%6-IFFT
%7-Prefijo cíclico

%Parametros comunes a transmisor y receptor
Nbits_bloque=192;               %Número de bits con el que trabaja el
interleaver

Nportadoras_Mapper=48;          %numero de portadoras que sacará el Mapper
                                %(a las que sumará la portadora de
referencia)
                                %y que irán en cada simbolo ofdm
                                %Es la mitad de las portadoras de inicio
                                %para que quepa en un vector con simetría
                                %Hermítica de 128  puntos

NFFT=128;                       %n° de puntos de la IFFT
Nbitsofdm=96;                   %número de bits en un símbolo ofdm:
                                %al usar dqpsk cada portadora representa a
dos bits
                                %y en un símbolo hay 48 portadoras + la de
referencia
```

```
%1-Generacion de stream de datos binarios a partir de cadena de caracteres
% -Entradas-
% cadena: cadena de entrada definida en main
% Nbits_bloque: Número de bits con el que trabaja el interleaver
% -Salidas-
% salida trama: fila de binarios con la trama formada
% N_bloques: numero de bloques de 192 bits
% lon_data: longitud de los datos de información
[salida_trama,N_bloques, lon_data ] = generadatosv5_tx(cadena,Nbits_bloque);

%2-Codificador convolucional
% -Entradas-
% salida_trama: fila de binarios con la trama formada
% -Salidas-
% trellis: estructura de trellis que se utiliza también en el decodificador
% de Viterbi del receptor
% salida_cod: salida del codificador
[trellis,salida_cod]=codificador(salida_trama);

%3-Scrambler
% -Entradas-
% salida_cod: salida del codificador
% -Salidas-
% salidaScr: salida del scrambler
[salidaScr] = scramblerv5( salida_cod );

%Transformamos en matriz con un un bloque para el interleaver en cada fila
salidaScr2=reshape(salidaScr.',Nbits_bloque,N_bloques).';

salidaInter=ones(N_bloques,Nbits_bloque);                %salida del
Interleaver
for indice=1:N_bloques
    % Aquí se deja de trabajar con stream de datos y se trabaja
    % en bloques de Nbits_bloque bits

    %4-Interleaver
    % -Entradas-
    % salidaScr2: salida del scrambler. Se va cogiendo en cada fila un
    % bloque de largo Nbits_bloque
    % -Salidas-
    % salidaInter: salida del interleaver. En cada fila habrá un bloque de
    % Nbits_bloque bits
     salidaInter(indice,:) = interleaver( salidaScr2(indice,:) );
end
% Ponemos cada 48*2 bits=96 bits del Interleaver que conformarán
% un símbolo OFDM en cada fila
Nsim_ofdm=N_bloques*2;           %número de símbolos ofdm resultantes al
final
salidaInter2=reshape(salidaInter.',Nportadoras_Mapper*2,Nsim_ofdm).';

salidaMapper=ones(Nsim_ofdm,Nportadoras_Mapper+1);     %salida del Mapper
salidaifft=ones(Nsim_ofdm,NFFT);                       %salida de la IFFT
for indice=1:Nsim_ofdm
    %Aquí se trabajará por símbolo OFDM, que corresponde con cada 48
    %portadoras que salgan del Interleaver (2 símbolos por cada bloque de
    %192 bits)
    % Mapper DQPSK: Realiza una modulación DQPSK a partir de los bits de
entrada
    % -Entradas-
    % salidaInter2: salida del interleaver. Se va cogiendo en cada fila un
    % bloque de largo Nbits_bloque
    % Nportadoras_Mapper: número de portadoras que se incluyen en un
```

```
    % símbolo ofdm, sin contar la de referencia
    % -Salidas-
    % salidaMapper: salida del mapper. En cada fila habrá
Nportadoras_Mapper+1
    % portadoras complejas
    salidaMapper(indice,:) =
mapperdqpsk_v5(salidaInter2(indice,:),Nportadoras_Mapper);

    %6-IFFT_Modulacion OFDM
    % -Entradas-
    % salidaMapper: salida del mapper. Se va cogiendo en cada fila
    % Nportadoras_Mapper+1 portadoras complejas
    % NFFT: número de puntos de la IFFT
    % -Salidas-
    % salidaifft: salida de la modulación.En cada fila a la salida habrá un
    % símbolo de NFFT puntos
    salidaifft(indice,:)=mod_ifft_v5(salidaMapper(indice,:),NFFT);
end

%7-Prefijo cíclico: las 12 muestras del final las ponemos al principio
salidaPrefijo=[salidaifft(:,117:end) salidaifft];

%Transformamos en un vector de una única fila con todos los simbolos
datos_transmitidos=reshape(salidaPrefijo.', 140*Nsim_ofdm, 1).';
```

rx_v5.m

```
%Script que llama a todas las funciones de los bloques del receptor
%Consta de:
%1-Quitar prefijo cíclico
%2-FFT
%3-Demapper DQPSK
%4-Deinterleaver
%5-Descrambler
%6-Decodificador de Viterbi
%7-Generación de los datos recibidos

%Transformamos en matriz con un simbolo ofdm en cada fila
rx_ordenado=reshape(datos_recibidos.',140,Nsim_ofdm).';

%1-Quitamos prefijo cíclico
rx_sinpref=rx_ordenado(:,13:end);

salidafft=ones(N_bloques,Nportadoras_Mapper+1);   %salida de la demodulación
OFDM
salidaDemapper=ones(Nsim_ofdm,Nbitsofdm);      %salida del Demapper

for indice=1:Nsim_ofdm
    %Aquí se trabajará por símbolo OFDM, que corresponde con cada 48
    %portadoras
    %2-FFT_Demodulación OFDM: Obtenemos las portadoras recibidas en el
    %símbolo OFDM
    % -Entradas-
    % rx_sinpref: datos recibidos sin prefijo cíclico.
    % NFFT: número de puntos de la IFFT
    % -Salidas-
    % salidafft: salida de la demodulación.En cada fila a la salida habrá un
    % símbolo de NFFT puntos
    salidafft(indice,:)=demod_fft_v5(rx_sinpref(indice,:),NFFT);
```

```
    %3-Demapper DQPSK: Realiza una demodulación DQPSK a partir de las
    %portadoras recibidas (soft decision).
    %Obtiene la distancia a la frontera de cada bit recibido.
    % -Entradas-
    % salidafft: salida de la demodulación.
    % Nbitsofdm: número de bits en un símbolo ofdm
    % -Salidas-
    % salidaDemapper: salida del demapper. Incluye las distancias a la
    % frontera del primer y segundo bit de cada portadora.
    [salidaDemapper(indice,:)] =
demapperdqpsk_v5(salidafft(indice,:),Nbitsofdm);
end

% Unimos dos a dos cada 96 bits de salida del demapper para formar un bloque
% de 192 bits con los que trabajará el deinterleaver
salidaDemapper2=reshape(salidaDemapper.',Nbits_bloque,N_bloques).';

salidaDeinter=ones(N_bloques,Nbits_bloque);     %salida del Deinterleaver
for indice=1:N_bloques
    %4-Deinterleaver: desentrelaza los elementos según la matriz definida en
el
    %estándar Prime
    % -Entradas-
    % salidaDemapper2: salida del demapper.
    % -Salidas-
    % salidaDeinter: salida del deinterleaver. En cada fila habrá un bloque
    % de Nbits_bloque bits.
    salidaDeinter(indice,:)=deinterleaver( salidaDemapper2(indice,:) );
end

%Pasamos a formar un vector único con todos los bloques del deinterleaver
%en serie
salidaDeinter_1=reshape(salidaDeinter.',N_bloques*Nbits_bloque,1).';

%5-Descrambler: desbaraja los elementos en base a una secuencia
pseudoaletoria
%predefinida.
 % -Entradas-
% salidaDeinter_1: salida del deinterleaver en serie.
% -Salidas-
% salidaDescr: salida del descrambler. En cada fila habrá un bloque
% de Nblock*Nbits_bloque elementos.
[ salidaDescr ] = descrambler_v5( salidaDeinter_1 );

%6-Decodificador de Viterbi: decodifica los datos de entrada en base al
%algoritmo de Viterbi y a la estructura de trellis que se le pasa (que
%deberá ser la misma que la del codificador en el transmisor)
% -Entradas-
% salidaDescr: salida del descrambler.
% trellis: estructura de trellis que se utiliza también en el decodificador
% de Viterbi del receptor
% -Salidas-
% salida_Decod: salida del decodificador
salidaDecod = decodificador_vit( salidaDescr,trellis);

%7-Generación de los datos recibidos, eliminando los campos añadidos por el
%entramado
[lendata_rx,Nbytes_rx, data_rx2] = generadatosv5_rx(salidaDecod, lon_data);
```

generadatosv5_tx.m

```
function [datos, Nbloques, lon_datos] = generadatosv5_tx(cadena,Nbits)

%Generacion de trama a partir de la cadena de datos introducida
% ------------------------------------------------------------------
%|MSDU - 8*M simb ofdm |    Flushing - 8    |     PADDING - 8  |
% ------------------------------------------------------------------
%-Entradas-
% cadena: cadena de caracteres a transmitir
% Nbits: numero de bits por simbolo
%-Salidas-
% datos:caracteres en binario en un vector
% Nbloques: numero de simbolos ofdm (1 simb=192 bits al usar dqpsk)
% lon_datos: longitud de los datos binarios de información

%Pasamos la cadena a una fila de binarios
[ filabits, lon_datos ] = ascii2bin(cadena);

% Calculamos el número de bits que se generarian tras el codificador
lon_datac=(lon_datos*2)+16;

%Comprobamos:
if lon_datac<=Nbits
    %Si solo hay un simbolo justo o menos de un simbolo
    lon_falta=((Nbits-lon_datac)/2);
    Nbloques=1;
else
    %Si tenemos mas de un simbolo
    Nbloques=ceil((lon_datac)/Nbits);
    len_res=mod(lon_datac,Nbits);  %bits que tiene el último simb
    lon_falta=((Nbits-len_res)/2);
end

%formamos la trama
flushing=zeros(1,8);
padding=zeros(1,lon_falta);
datos=[filabits flushing padding];

end
```

codificador.m

```
function [trellis,salida_codificador] = codificador( entrada_codificador )
%Codificador convolucional: encargado de añadir información redundante para
%proteger contra posibles errores
%-Entradas-
% entrada_codificador: datos binarios que se codifican
%-Salidas-
% trellis:Estructura de trellis que necesita la función de Matlab. Se
% define como la que se utiliza en el estándar PRIME.
% salida_codificador: datos binarios de salida del codificador

%Definición de la estructura trellis según estándar
trellis=poly2trellis(7,[171 133]);
%Hacemos pasar por el codificador (funcion de Matlab)
salida_codificador=convenc(entrada_codificador,trellis);

end
```

scramblerv5.m

```
function [ salScram ] = scramblerv5( entScram )
%Scrambler:baraja los bits en base a una secuencia pseudoaletoria descrita
%en el estándar
% -Entradas-
% entScram: entrada del scrambler
% -Salidas-
% salScram: salida del scrambler

lent=length(entScram);                  %longitud de los datos de entrada
salScram=zeros(1,lent);                 %variable de salida
bits_sobrante=mod(lent,127);            %número de bits que sobrarán al
dividir en trozos de 127 bits
nfilas_llenas=floor(lent/127);          %número de filas de 127 que irán
llenas
resul_llenas=zeros(nfilas_llenas,127);  %variable donde irán las filas de
127 que van llenas

%Secuencia pseudoaleatoria
pseudo=[0 0 0 0 1 1 1 0 1 1 1 1 0 0 1 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 ...
    0 1 0 0 1 1 0 0 0 1 0 1 1 ...
    1 0 1 0 1 1 0 1 1 0 0 0 0 0 1 1 0 0 1 1 0 1 0 1 0 0 1 1 1 0 0 1 1 1 1 ...
    0 1 1 0 1 0 0 0 0 1 0 ...
    1 0 1 0 1 1 1 1 1 0 1 0 0 1 0 1 0 0 0 1 1 0 1 1 1 0 0 0 1 1 1 1 1 1 1];

%Cogemos las filas que quedarán enteras al dividir entre 127
filas_llenas=reshape(entScram(1:(end-bits_sobrante)).',127,nfilas_llenas).';

%Hacemos la xor de todas esas filas por la secuencia pseudoaleatoria
for indice=1:nfilas_llenas
    resul_llenas(indice,:)=xor(pseudo,filas_llenas(indice,:));
end
%Las ponemos en la salida
salScram(1:127*nfilas_llenas)=reshape(resul_llenas.',127*nfilas_llenas,1).';
%Añadimos la xor de los bits que quedan de sobra al final
salScram((127*nfilas_llenas)+1:end)=xor(entScram((127*nfilas_llenas)+1:end),p
seudo(1:bits_sobrante));
end
```

interleaver.m

```
function [ vectorw ] = interleaver( vectorv )
    %Interleaver: Entrelaza los bits según la especificación del estándar
Prime
    %Trabaja por bloques de 192 bits
    % -Entradas-
    %vector v:vector de 1x192 bits
    % -Salidas-
    %vector w:vector de 1x192 bits ya entrelazado

        prueba1=reshape(vectorv,16,12);
        vectorw=reshape(prueba1.',1,192);
end
```

mapperdqpsk_v5.m

```
function [portadoras] = mapperdqpsk_v5(entrada_mapper,nportadoras)
    %Mapper: Realiza una modulación DQPSK a partir de los bits de entrada
    % -Entradas-
```

```
    % entrada_mapper: entrada del mapper.
    % nportadoras: número de portadoras que se incluyen en un
    % símbolo ofdm, sin contar la de referencia
    % -Salidas-
    % portadoras: salida del mapper. En cada fila habrá nportadoras+1
    % portadoras complejas

    %Sacamos la informacion de los datos binarios y la
    %guardamos en un vector fila de nportadoras numeros.
    binarios = (reshape(entrada_mapper, 2, nportadoras)).';
    %Incremento de fase correspondiente a la dqpsk
    incrFase=pi/2;
    %Obtenemos el incremento de fase que aportará cada par de bits en la
    %modulación diferencial:
    %0*pi/2 si es 00
    %1*pi/2 si es 01
    %2*pi/2 si es 11
    %3*pi/2 si es 10
    InfoFase=bin2gray(bin2dec(int2str(binarios)),'dpsk',4);

    %Obtención de las fases de la modulación diferencial
    faseCarr=zeros(nportadoras+1,1);          %variable de salida con todas
las portadoras
    faseCarr(1)=0;                            %portadora de referencia
    for k=2:nportadoras+1
        faseCarr(k)=mod(faseCarr(k-1)+incrFase*InfoFase(k-1),2*pi);
    end
    %Hacemos el vector de exponenciales con la fase ya determinada
    portadoras=exp(1i*faseCarr);

end
```

mod_ifft_v5.m

```
function [ salidamod ] = mod_ifft_v5( portadoras,NFFT)
%MOD_IFFT: Modulacion OFDM utilizando la IFFT
% -Entradas-
% portadoras:vector del mapper de 1xNportadoras_Mapper+1 números complejos,
% uno por portadora.
% NFFT: numero de puntos de la IFFT
% -Salidas-
% salidamod: vector 1xNFFT con la salida de la ifft

%Generamos el vector con simetría Hermítica para obtener a la salida de la
%IFFT una señal totalmente real:
% ------------------------------------------------------------------------
-------------------------------
%| Portadoras a cero | 49 portadoras | DC a cero | 49 portadoras conjugadas y
espejadas | Portadoras a cero |
% ------------------------------------------------------------------------
-------------------------------
%Tenemos en cuenta al organizar así las muestras de entrada como recibe los
%datos la función ifft de Matlab.
    Entifft = zeros(1 ,NFFT);           %Entrada para la IFFT
    Entifft(: ,2:50) = portadoras;
    Entifft(: ,80:end) = conj(portadoras(:,end:-1:1));
%Usamos ifftshift para que los datos estén correctamente ordenados a la
entrada
    salidamod=ifft(ifftshift(Entifft),NFFT);
```

```
end
```

demod_fft_v5

```
function [ portadoras ] = demod_fft_v5( recibido,NFFT )
%Demodulador OFDM utilizando la FFT
% -Entradas-
% recibido:muestras recibidas sin prefijo cíclico en un vector 1xNFFT
% NFFT: numero de puntos de la FFT
% -Salidas-
% portadoras: vector 1xNportadoras_Mapper+1 con las portadoras recibidas
salidafft=fftshift((fft(recibido,NFFT)),2);
%cogemos las Nportadoras_Mapper+1 portadoras de origen
portadoras=salidafft(1,2:50);

end
```

demapperdqpsk_v5.m

```
function [ demapeado ] = demapperdqpsk_v5( complejos,Nbitsofdm )
%Demapper DQPSK: Realiza una demodulación DQPSK a partir de las
%portadoras recibidas (soft decision).
%Obtiene la distancia a la frontera de cada bit recibido.
% -Entradas-
%complejos: vector 1xNportadoras_Mapper+1 de números complejos
% Nbitsofdm: número de bits en un símbolo ofdm
% -Salidas-
% demapeado: vector de Nbitsofdm elementos. Incluye las distancias a la
% frontera del primer y segundo bit de cada portadora.

%Sacamos las portadoras transmitidas.
cdem=complejos(2:end).*conj((complejos(1:end-1)));
%Para facilitar la demodulación, giramos la constelación 45 grados
cdemdesf=cdem.*exp(1i*(pi/4));

bit1=imag(cdemdesf);     %la distancia a la frontera del primer bit es la
                         %parte imaginaria de la portadora demodulada
bit2=real(cdemdesf);     %la distancia a la frontera del segundo bit es la
                         %parte real de la portadora demodulada

%Ahora hay que recomponer el vector poniendo bit1-bit2 alternativamente
demapeado=zeros(1,Nbitsofdm);
demapeado(1:2:end-1)=bit1;
demapeado(2:2:end)=bit2;

end
```

deinterleaver.m

```
function [ vectorv ] = deinterleaver( vectorw )
    %Deinterleaver: desentrelaza los elementos según la matriz definida en el
    %estándar Prime
    % -Entradas-
    % vectorw: salida del demapper.
    % -Salidas-
    % vectorv: salida reordenada. En cada fila habrá un bloque
    % de Nbits_bloque bits.
```

```
    prueba=reshape(vectorw.',12,16);
    vectorv=reshape(prueba.',1,192);

end
```

descrambler_v5.m

```
function [ salida_descrambler ] = descrambler_v5( entrada_descrambler )
%Descrambler: desbaraja los elementos en base a una secuencia pseudoaletoria
%predefinida.
 % -Entradas-
% entrada_descrambler: stream de datos de entrada para desbarajar.
% -Salidas-
% salida_descrambler: salida desbarajada.

lent=length(entrada_descrambler);       %longitud de los datos de entrada
salida_descrambler=zeros(1,lent);       %variable de salida
sobrante=mod(lent,127);                 %numero de bits que sobraran al
dividir en trozos de 127 bits
nfilas_full=floor(lent/127);            %numero de filas de 127 que iran
llenas
resul_full=zeros(nfilas_full,127);      %variable donde iran las filas de 127
que van llenas

%Secuencia pseudoaleatoria (pasada a -1 dónde hay un 1 y hay que cambiar el
%signo y a 1 dónde había un 0 y no hay que cambiar el signo)
pseudo=(-2*[0 0 0 0 1 1 1 0 1 1 1 1 0 0 1 0 1 1 0 0 1 0 0 1 0 0 0 0 0 0 1 0
0 ...
    0 1 0 0 1 1 0 0 0 1 0 1 1 ...
    1 0 1 0 1 1 0 1 1 0 0 0 0 0 1 1 0 0 1 1 0 1 0 1 0 0 1 1 1 0 0 1 1 1 1 ...
    0 1 1 0 1 0 0 0 0 1 0 ...
    1 0 1 0 1 1 1 1 1 0 1 0 0 1 0 1 0 0 0 1 1 0 1 1 1 0 0 0 1 1 1 1 1 1
1])+1;

%Cogemos las filas que quedaran enteras al dividir entre 127
filas_full=reshape(entrada_descrambler(1:(end-
sobrante)).',127,nfilas_full).';

%Multiplicamos por la secuencia pseudoaleatoria
for indice=1:nfilas_full
    resul_full(indice,:)=pseudo.*filas_full(indice,:);
end
%Las ponemos en la salida de la funcion
salida_descrambler(1:127*nfilas_full)=reshape(resul_full.',127*nfilas_full,1)
.';
%Añadimos la multiplicación de los bits que quedan de sobra al final
salida_descrambler((127*nfilas_full)+1:end)=entrada_descrambler((127*nfilas_f
ull)+1:end).*pseudo(1:sobrante);
end
```

decodificador_vit.m

```
function [ salida_Decod ] = decodificador_vit( entrada_decod,trellis )
%Decodificador de Viterbi: decodifica los datos de entrada en base al
%algoritmo de Viterbi y a la estructura de trellis que se le pasa (que
%deberá ser la misma que la del codificador en el transmisor)
% -Entradas-
% entrada_decod: fila de distancias a la frontera de cada bit a decodificar
% trellis: estructura de trellis que se utiliza también en el decodificador
```

```
% de Viterbi del receptor
% -Salidas-
% salida_Decod: salida del decodificador

tblen=35;               %según se recomienda, utilizar un número en tblen de
                        %5*número de registros de la estructura trellis (7
                        %en nuestro caso)
opmode='trunc';         %el decodificador empieza en cero y no se hacen
llamadas continuas
dectype='unquant';      %a la entrada tenemos valores reales, un 1 representa
un 0 lógico y un -1 un 1 lógico
%Utilizamos la función de Matlab que implementa el decodificador de Viterbi
%con las opciones elegidas
salida_Decod= vitdec(entrada_decod,trellis,tblen,opmode,dectype);

end
```

generadatosv5_rx

```
function [lendatos_rx,Nbytes_rx,datos_rx_bits ] = generadatosv5_rx( datos_rx,
lon datos )

%Desentrama los datos recibidos y muestra en pantalla la cadena de datos
%recibida quitando toda la información sobrante
% ------------------------------------------------------------
%|MSDU - 8*M simb ofdm |    Flushing - 8    |     PADDING - 8  |
% ------------------------------------------------------------
%-Entradas-
% datos_rx: bits recibidos
% lon_datos: longitud de los datos binarios de información
%-Salidas-
%lendatos_rx: número de bits totales recibidos (toda la trama)
%Nbytes_rx: número de bytes totales recibidos (toda la trama)
% datos_rx_bits: caracteres recibidos

lendatos_rx=numel(datos_rx);      %número de bits totales recibidos (toda la
trama)
Nbytes_rx=(lendatos_rx)/8;        %numero de bytes totales recibidos (toda la
trama)
datos_rx_bits=reshape(datos_rx',8,Nbytes_rx)';  %ponemos cada bye en una fila
%Tiramos los bytes sobrantes
datos_rx_bits=datos_rx_bits(1:(lon_datos/8),:);
%Pasamos a caracteres
datos_rx_bits=bin2dec(int2str(datos_rx_bits))';
datos_rx_bits=char(datos_rx_bits);
datos_rx_bits

end
```

pruebas_bloques_fichero.m

```
clc
clear all
close all
%Script de pruebas
main_v5
%____Datos de prueba____%
```

```
fprintf('Datos de entrada: %s\n', cadena);

%imprimimos menú
disp('_____________________________');
disp('Menú de pruebas de los bloques');
disp('***(Matlab vs micro)***');
disp('----Transmisor----');
disp('1)Comprobar Trama->Codificador convolucional->');
disp('2)Comprobar Trama->Codificador convolucional->Scrambler-> ');
disp('3)Comprobar Trama->Codificador convolucional->Scrambler->Interleaver->
');
disp('4)Comprobar Trama->Codificador convolucional->Scrambler->Interleaver-
>Mapper-> ');
disp('5)Comprobar Trama->Codificador convolucional->Scrambler->Interleaver-
>Mapper->IFFT-> ');
disp('----Receptor----');
disp('6)->FFT-> ');
disp('7)->FFT->Demapper-> ');
disp('8)->FFT->Demapper-> Deinterleaver->');
disp('9)->FFT->Demapper-> Deinterleaver->Descrambler-> ');

disp('Opcion 1 por defecto');
variable = input('Elige una opción: \n');
disp('_____________________________________________')

%miramos opcion elegida
if(variable==2)
 %Cogemos dato del fichero
    fcaptura=fopen('salidaScr.txt','rt'); %Fichero donde se guarda la salida
    [salida_serial, cuantas] =  fread(fcaptura,Inf, 'uint8');
    salida_serial=dec2hex(salida_serial, 2);
    fclose(fcaptura);

    %pasamos a hex los datos obtenidos en matlab
    hexadecimal  = vectorbin2hex(salidaScr ,N_bloques*(Nbits_bloque/8));

    ok=hexadecimal==salida_serial;
    porc=sum(100*sum(ok)/length(ok))/2;
    fprintf('Porcentaje de acierto Scrambler: %.2f%%\n', porc);

 elseif (variable==3)

    %Cogemos dato del fichero
    fcaptura=fopen('salidaInter.txt','rt'); %Fichero donde se guarda la
salida
    [salida_serial, cuantas] =  fread(fcaptura,Inf, 'uint8');
    salida_serial=dec2hex(salida_serial, 2);
    fclose(fcaptura);

    %Transformamos en un vector de una única fila con todos los simbolos
    salida_aux=reshape(salidaInter.', Nbits_bloque*N_bloques, 1).';

    %Pasamos a hex los datos obtenidos en Matlab
    hexadecimal  = vectorbin2hex( salida_aux ,N_bloques*(Nbits_bloque/8));
    ok=hexadecimal==salida_serial;
    porc=sum(100*sum(ok)/length(ok))/2;
    fprintf('Porcentaje de acierto Interleaver: %.2f%%\n', porc);

elseif (variable == 4)

    %Cogemos dato del fichero
```

```
    fcaptura=fopen('salidaMapper.txt','rt'); %Fichero donde se guarda la
salida
    [salida_serial, cuantas] =  fread(fcaptura,Inf, 'float32');
    fclose(fcaptura);
    salida_serial=(reshape(salida_serial,  cuantas/2,2));
    real_ps=reshape(salida_serial(1:2:end-
1,:),(Nportadoras_Mapper+1)*Nsim_ofdm,1);

imag_ps=reshape(salida_serial(2:2:end,:),(Nportadoras_Mapper+1)*Nsim_ofdm,1);

    %tranformamos en un vector de una única fila con todos los simbolos
    real_matlab=reshape(real(salidaMapper).',
(Nportadoras_Mapper+1)*Nsim_ofdm, 1);
    imag_matlab=reshape(imag(salidaMapper).',
(Nportadoras_Mapper+1)*Nsim_ofdm, 1);

    margen=10e-10;
    ok = abs(real_matlab-real_ps)< margen;
    porc=100*sum(ok)/length(ok);
    fprintf('Porcentaje de acierto partes reales con
margen %.9f: %.2f%%\n',margen, porc);

    ok = abs(imag_matlab-imag_ps)< margen;
    porc=100*sum(ok)/length(ok);
    fprintf('Porcentaje de acierto partes imag con
margen %.9f: %.2f%%\n',margen, porc);

    figure
    hold on
    plot(salidaMapper(:),'*r')
    complejos_ps=real_ps+(imag_ps*i);
    plot(complejos_ps(:),'ob')
    axis([-1.2 1.2 -1.2 1.2])
    legend('Salida del mapper en Matlab','Salida del mapper en el hardware');
    title('Constelación de salida del mapper')
    xlabel('Eje real')
    ylabel('Eje imaginario')
    grid
    hold off
elseif (variable==5)
    %Cogemos dato del fichero
    fcaptura=fopen('salidaPrefijo.txt','rt'); %Fichero donde se guarda la
salida
    [salida_serial, cuantas] =  fread(fcaptura,Inf, 'float32');
    fclose(fcaptura);
    salida_serial=(reshape(salida_serial, 2, cuantas/2)).';
    real_ps=salida_serial(:,1);
    imag_ps=salida_serial(:,2);

    real_matlab=(real(datos_transmitidos).' );
    imag_matlab=(imag(datos_transmitidos).');

    pru=[real_matlab real_ps imag_matlab imag_ps]

    plot(real_matlab)
    figure
    plot(real_ps)

elseif (variable==6)
    %Cogemos dato del fichero
    fcaptura=fopen('salidaFft.txt','rt'); %Fichero donde se guarda la salida
    [salida_serial, cuantas] =  fread(fcaptura,Inf, 'float32');
```

```
    fclose(fcaptura);
    salida_serial=(reshape(salida_serial, 2, cuantas/2)).';
    real_ps=salida_serial(:,1);
    imag_ps=salida_serial(:,2);

    reshape(real([salidaifft(:,117:end) salidaifft]).', 140*Nsim_ofdm, 1);

real_matlab=reshape((real(salidafft).' ),(Nportadoras_Mapper+1)*Nsim_ofdm,1);

imag_matlab=reshape((imag(salidafft).' ),(Nportadoras_Mapper+1)*Nsim_ofdm,1);

    pru=[real_matlab real_ps imag_matlab imag_ps]

    margen=10e-5;
    ok = abs(real_matlab-real_ps)< margen;
    porc=100*sum(ok)/length(ok);
    fprintf('Porcentaje de acierto partes reales con
margen %.6f: %.2f%%\n',margen, porc);

    ok = abs(imag_matlab-imag_ps)< margen;
    porc=100*sum(ok)/length(ok);
    fprintf('Porcentaje de acierto partes imag con
margen %.6f: %.2f%%\n',margen, porc);


elseif (variable==7)
    %Cogemos dato del fichero
    fcaptura=fopen('salidaDemap.txt','rt'); %Fichero donde se guarda la
salida
    [salida_serial, cuantas] =  fread(fcaptura,Inf, 'float32');
    fclose(fcaptura);
    bit1_ps=salida_serial(1:2:end-1,:);
    bit2_ps=salida_serial(2:2:end,:);

    bit1_matlab=reshape((salidaDemapper(:,1:2:end-1)).',
N_bloques*(Nbits_bloque/2), 1);
    bit2_matlab=reshape((salidaDemapper(:,2:2:end)).',
N_bloques*(Nbits_bloque/2), 1);

    margen=0.00001;
    ok = abs(bit1_matlab-bit1_ps)< margen;
    porc=100*sum(ok)/length(ok);
    fprintf('Porcentaje de acierto bit 1 con margen %.5f: %.2f%%\n',margen,
porc);

    ok = abs(bit2_matlab-bit2_ps)< margen;
    porc=100*sum(ok)/length(ok);
    fprintf('Porcentaje de acierto bit 2 con margen %.5f: %.2f%%\n', margen,
porc);
 %
%     figure
%     hold on
%     complejos_Matlab=bit2_matlab+(bit1_matlab*i);
%     plot(complejos_Matlab(:),'*r')
%     complejos_ps=bit2_ps+(bit1_ps*i);
%     plot(complejos_ps(:),'ob')
%     axis([-1.2 1.2 -1.2 1.2])
%     legend('Salida del demapper en Matlab','Salida del demapper en el
hardware');
%     title('Constelación de salida del demapper')
%     xlabel('Eje real')
%     ylabel('Eje imaginario')
```

```
%     grid
%     hold off
%

elseif (variable==8)
    %Cogemos dato del fichero
    fcaptura=fopen('salidaDeinter.txt','rt'); %Fichero donde se guarda la
salida
    [salida_serial, cuantas] =  fread(fcaptura,Inf, 'float32');
    fclose(fcaptura);

    margen=0.01;
    ok = abs((salida_serial.')-salidaDeinter_1)< margen;
    porc=100*sum(ok)/length(ok);
    fprintf('Porcentaje de acierto Deinterleaver con
margen %.5f: %.2f%%\n',margen, porc);

%     figure
%     hold on
%     plot(salidaDeinter_1(:),'r')
%     plot(salida_serial(:),'b')
%     legend('Salida del deinterleaver en Matlab','Salida del deinterleaver
en el hardware');
%     title('Comparación deinterleaver')
%     xlabel('n° muestras')
%     ylabel('Información de cada bit')
%     grid
%     hold off
%

elseif (variable==9)
    %Cogemos dato del fichero
    fcaptura=fopen('salidaDescr.txt','rt'); %Fichero donde se guarda la
salida
    [salida_serial, cuantas] =  fread(fcaptura,Inf, 'float32');
    fclose(fcaptura);
    salida_serial=salida_serial.';

    margen=0.00001;
    ok = abs((salida_serial)-salidaDescr)< margen;
    porc=100*sum(ok)/length(ok);
    fprintf('Porcentaje de acierto Descrambler con
margen %.5f: %.2f%%\n',margen, porc);

%     figure
%     hold on
%     plot(salidaDescr,'r')
%     plot(salida_serial,'b')
%     legend('Salida del descrambler en Matlab','Salida del descrambler en el
hardware');
%     title('Comparación descrambler')
%     xlabel('n° muestras')
%     ylabel('Información de cada bit')
%     grid
%     hold off

else
    %Cogemos dato del fichero
    fcaptura=fopen('salidaCodificador.txt','rt'); %Fichero donde se guarda la
salida
    [salida_serial, cuantas] =  fread(fcaptura,Inf, 'uint8');
    salida_serial=dec2hex(salida_serial, 2);
```

```
    fclose(fcaptura);

    %pasamos a hex los datos obtenidos en matlab
    hexadecimal  = vectorbin2hex(salida_cod ,N_bloques*(Nbits_bloque/8));

    ok=hexadecimal==salida serial;
    porc=sum(100*sum(ok)/length(ok))/2;
    fprintf('Porcentaje de acierto Codificador Convolucional: %.2f%%\n',
porc);


end
```

vectorbin2hex.m

```
function [ hexadecimal ] = vectorbin2hex( binarios,Ncar )
    bits8=reshape(binarios.',8,Ncar).';   %binarios por filas
    hexadecimal=dec2hex(bin2dec(char(bits8+48)),2);
end
```

ANEXO II: CÓDIGO IMPLEMENTADO EN C

Nota: Se incluyen en este anexo sólo las funciones principales, obviándose algunas funciones de librerías propias menos relevantes así cómo las cabeceras que no se consideren importantes. Todo se incluirá como ficheros de código adjuntos en la versión digital de la memoria.

main.c

```
/* Librerías ------------------------------------------------------------
--*/
#include "stm32f4xx.h"                  // Device header
#include "arm_math.h"                   // ARM::CMSIS:DSP
#include "main.h"
#include "tx.h"
#include "rx.h"
#include "leds.h"
#include "uart.h"
#include "rcc.h"
#include "adc.h"
#include "timer.h"
#include "dac.h"
#include "config.h"
#include "gpio.h"
#include "dma.h"
#include "varios.h"
#include "stdlib.h"

/* ---------------------------------------------------------------------
Variables globales
(Las constantes están definidas en main.h)
---------------------------------------------------------------------*/
uint8_t lectura_PC6=0;                  //Resultado de la lectura de PC6
volatile uint16_t espera=0;    //Variable para la interrupción del botón de
usuario

uint8_t datos[TAM_DATOS] = "Esto es una prueba con 49 caracteres:
*123456789#";                   //Datos de entrada.

float32_t datos_transmitidos[NSIM*TAM_CON_PREF_2];    //Vector de datos de
salida del transmisor
//Buffer de entrada al DAC (lo usa la DMA)
uint16_t
buffer_DAC[TAM_BUFFER_DAC]={3071,2252,1024,1228,2866,2047,2866,1433,2252,2252
,1024,2252,1638,1024,2047,1433,1228,1433,1228,1433,1024,2252,1638,2047,2456,2
047,2047,1842,1228,2047,2866,2866};

// Variable modificada por la interrupción del ADC que indica que se ha
tomado una muestra
volatile uint8_t DMA_ADC_fin = 0;
uint8_t sinc_ok=0;

volatile uint16_t buffer_ADC[1]={0};  //Buffer de entrada del ADC (lo usa la
DMA)
int16_t buffer_sinc[TAM_PREAMB]={0};  //Buffer para sincronizar el preámbulo
//Símbolo para comparar en la sincronización
int16_t simb_sinc[TAM_PREAMB]={1024,205,-1024,-819,819,0,819,-614,205,205,-
```

```
1024,205,-409,-1024,0,-614,-819,-614,-819,-614,-1024,205,-409,0,409,0,0,-
205,-819,0,819,819};
//Punteros para realizar los productos con la función SMLAD
uint32_t * p_simb_sinc=NULL;
uint32_t * p_buffer_sinc=NULL;
uint32_t contador_acum=0;      //Indice para realizar los productos
uint32_t acum=0;               //Resultado de los productos
uint32_t indice_sinc=0;//Índice para desplazar el vector de sincronización
int umbral_sinc= 7000000;
uint32_t indice_buff=0;
uint16_t simb_recibido_ADC[TAM_BUFFER_DATOS_RX]; //Entrada del receptor sin
manipular
float32_t simb_recibido[NSIM*TAM_CON_PREF_2]; //Buffer auxiliar con los datos
ajustados del ADC
float32_t datos_recibidos[NBLOQ*NBPBLOQ];     //Vector de datos de salida del
receptor

/*-----------------------------------------------------------------------------
Function that initializes and configures DAC1:
-Timer6 Trigger
-DMA
 *-----------------------------------------------------------------------------*/
void DAC1_DMA_Timer6Trigg (){
        //enables clocks
         __RCC_GPIOA_CLK_ENABLE();
        __RCC_DAC_CLK_ENABLE();
          RCC_DMA1_CLK_ENABLE();
        __RCC_TIM6_CLK_ENABLE();

        //GPIO config
        config_gpio_mode(GPIOA, 4, GPIO_MODE_ANALOG);
        config_gpio_pull(GPIOA, 4, GPIO_NOPULL);

        //Timer config
        TIM_CounterUp_Init(TIM6,(uint32_t)DAC_TIM_PERIOD, 0, 0);
         TIM_SelectOutputTrigger(TIM6, TIM_TRGOSource_Update);
         TIM_Cmd(TIM6, ENABLE);

        //DMA config
         DMA_DeInit(DMA1_Stream5);
        DMA_Init_Def(DMA1_Stream5);

  /*------------------------- DMAy Streamx NDTR Configuration -------------*/
  /* Write to DMAy Streamx NDTR register */
         DMA1_Stream5->NDTR =TAM_BUFFER_DAC;

  /*------------------------- DMAy Streamx PAR Configuration -------------*/
  /* Write to DMAy Streamx PAR */
         DMA1_Stream5->PAR = (uint32_t)DAC_DHR12R1_ADDR;

  /*------------------------- DMAy Streamx M0AR Configuration -------------*/
  /* Write to DMAy Streamx M0AR */
         DMA1_Stream5->M0AR = (uint32_t)&buffer_DAC;


/*Enable the transfer complete interrupt, the Half transfer complete
interrupt, the transfer Error interrupt,the FIFO Error interrupt,the direct
mode Error interrupt  */
         DMA1_Stream5->CR  |= DMA_SxCR_TCIE | DMA_SxCR_HTIE| DMA_SxCR_TEIE |
DMA_SxCR_DMEIE ;
```

```
        //DAC config
        DAC_Init_Def();

        DMA_Cmd(DMA1_Stream5, ENABLE);
       DAC_Cmd(DAC_Channel_1, ENABLE);
       DAC_DMACmd(DAC_Channel_1, ENABLE);
}

/*-----------------------------------------------------------------------
Function that initializes and configures ADC1:
-Timer2 Trigger
-DMA
 *-----------------------------------------------------------------------*/
void ADC1_DMA_Timer2Trigger_Init () {

                //enables clocks
        __RCC_GPIOA_CLK_ENABLE();
       __RCC_ADC1_CLK_ENABLE();
       __RCC_DMA2_CLK_ENABLE();
       __RCC_TIM2_CLK_ENABLE();

        //GPIO config
        config_gpio_mode(GPIOA, 1, GPIO_MODE_ANALOG);
        config_gpio_pull(GPIOA, 1, GPIO_NOPULL);

        /* Timer config
        Only works correctly if SYSCLK=168MHz=HCLK
        */

        TIM_CounterUp_Init(TIM2,(uint32_t)ADC_TIM_PERIOD, 0, 0);
       TIM_SelectOutputTrigger(TIM2, TIM_TRGOSource_Update);
       TIM_Cmd(TIM2, ENABLE);

/* DMA2 Channel1 configuration ----------------------------------------------*/
        DMA_DeInit(DMA2_Stream0);
        DMA_Init_Def(DMA2_Stream0);
        //Channel 0
        DMA2_Stream0->CR &= ~(DMA_SxCR_CHSEL_0 |DMA_SxCR_CHSEL_1
|DMA_SxCR_CHSEL_2);
        //periph to memory dir
        DMA2_Stream0->CR &= ~(DMA_SxCR_DIR_0 |DMA_SxCR_DIR_1 );

  /*------------------------- DMAy Streamx NDTR Configuration ---------------
-*/
  /* Write to DMAy Streamx NDTR register */
  DMA2_Stream0->NDTR =TAM_BUFFER_ADC;

  /*------------------------- DMAy Streamx PAR Configuration ----------------
-*/
  /* Write to DMAy Streamx PAR */
  DMA2_Stream0->PAR = (uint32_t)ADC1_RDR;

  /*------------------------- DMAy Streamx M0AR Configuration ---------------
-*/
  /* Write to DMAy Streamx M0AR */
  DMA2_Stream0->M0AR = (uint32_t)buffer_ADC;

/*Enable the transfer complete interrupt, the Half transfer complete
interrupt, the transfer Error interrupt,the FIFO Error interrupt,the direct
mode Error interrupt  */
  DMA2_Stream0->CR  |= DMA_SxCR_TCIE | DMA_SxCR_HTIE| DMA_SxCR_TEIE |
DMA_SxCR_DMEIE ;
```

```
        NVIC_SetPriority(DMA2_Stream0_IRQn,1);
        NVIC_EnableIRQ(DMA2_Stream0_IRQn);

        DMA_Cmd(DMA2_Stream0, ENABLE);

        /* Setup and initialize ADC converter
        */
        ADC_Init_Def();
        ADC1->CR2     |= ADC_CR2_ADON;            /* ADC enable
*/

}



/*-------------------------------------------------------------------------
-

Main

*-------------------------------------------------------------------------
*/
int main(void){

SystemInit();
   config_clk_HSE168();
   //Configura el reloj del sistema
   led_ini();
   uart5_setup();
   PC6_choose_mode_init();
   //Inicializa PC6 para determinar modo
   led_off(all_leds);
   //Apaga los leds
   lectura_PC6=gpio_read(GPIOC, 6);    //Lee PC6 para determinar modo

        if(lectura_PC6==0){
                boton_int_ini();
                while(1){
                //MODO TRANSMISOR
                led_on(green_led);
                //Pasamos los datos por el transmisor
                transmisor(datos,TAM_DATOS,datos_transmitidos);
                //Ajustamos el resultado para poder transmitirlo por el DAC
        ajusta_rango_DAC(datos_transmitidos,NSIM*TAM_CON_PREF_2,buffer_DAC+TA
M_PREAMB);
                //Inicializamos DAC
                DAC1_DMA_Timer6Trigg();
                //Mandamos por la UART para depuración
                Send_Uint16_String_UART5 (buffer_DAC, TAM_BUFFER_DAC);
                led_on(blue_led);
                espera=1;
                //Esperamos hasta que se pulse el botón de usuario
                while(espera==1);
                }
        }
        else{
                //MODO RECEPTOR
                led_on(red_led);
                ADC1_DMA_Timer2Trigger_Init();
```

```
        //Inicialiazamos ADC
                while(sinc_ok==0){
                        //Si no se ha sincronizado la señal
                        if(DMA_ADC_fin!=0){
                                //Cuándo se recibe muestra del ADC
                                //gpio_toggle(0x0800);
                                //Ponemos última muestra recogida
                                buffer_sinc[TAM_PREAMB-
1]=(int16_t)(buffer_ADC[0]-0x7FF);
                                p_buffer_sinc=(uint32_t *)&buffer_sinc;
                                p_simb_sinc=(uint32_t *)&simb_sinc;
                                contador_acum=0;
                                acum=0;
        //Realizamos los productos para comprobar que la señal sea la misma
                                while (contador_acum<MAX_CONT_SINC){

        acum=__SMLAD((*p_buffer_sinc++),(*p_simb_sinc++), acum);
                                                contador_acum++;
                                        }
                                acum=abs((int32_t)(acum));
                                //Se comprueba el umbral de sincronización
                                if(acum>=umbral_sinc){
                                        sinc_ok=1;
                                }
                                //Se desplaza el buffer de sincronizacion
                                indice_sinc=0;
                                while(indice_sinc<TAM_PREAMB-1){

buffer_sinc[indice_sinc]=buffer_sinc[indice_sinc+1];
                                                indice_sinc++;
                                }
                                //gpio_toggle(0x0800);
                                DMA_ADC_fin=0;
                        }
                }
        //Si se ha sincronizado comenzamos a guardar en el buffer de datos
                indice_buff=0;
                while(indice_buff<TAM_BUFFER_DATOS_RX){
                        if(DMA_ADC_fin!=0){
                           simb_recibido_ADC[indice_buff++]=buffer_ADC[0];
                                DMA_ADC_fin=0;
                        }
                }

ajusta_rango_ADC(simb_recibido_ADC,TAM_BUFFER_DATOS_RX,simb_recibido);
        receptor(simb_recibido, TAM_BUFFER_DATOS_RX,datos_recibidos);
        SendFloatString_UART5(datos_recibidos,NBLOQ*NBPBLOQ);
//si he terminado, reinicializo el índice e indico que hay que volver a
sincronizar
        led_on(orange_led);
        }

}
/*-----------------------------------------------------------------------

Interrupciones

*-----------------------------------------------------------------------
*/
/*-----------------------------------------------------------------------
-
  Button IRQ: Se ejecuta al pulsar el botón de usuario
```

```
 *-----------------------------------------------------------------------------
-*/
void EXTI0_IRQHandler(void){
  if((EXTI->PR &((uint32_t)0x00001)) != RESET)
  {
                if(espera==1){
                        espera=0;
                }
    //Clear the EXTI line 0 pending bit
    EXTI->PR = (uint32_t)0x00001;
  }
}
/*-----------------------------------------------------------------------------
-
  DMA IRQ: Se ejecuta cuando una transferencia del ADC se ha completado
 *-----------------------------------------------------------------------------
-*/
void DMA2_Stream0_IRQHandler(void) {
                led_toggle(blue_led);
                if( (DMA2->LISR & (DMA_LISR_TCIF0)) | (DMA2->LISR &
(DMA_LISR_HTIF0))){
                        DMA2->LIFCR =DMA_LIFCR_CTCIF0|DMA_LIFCR_CHTIF0; //
Poner a 0 la Transfer Complete Flag
                        DMA_ADC_fin=1;
                }
}
```

main.h

```
#ifndef MAIN_H
#define MAIN_H

//DAC defines
#define   DAC_TRIGG_FREQ        62500    // Frecuencia de disparo del DAC
#define   DAC_CNT_FREQ          84000000 // Reloj de TIM6 (prescaled APB1)
// Valor de Autorreload
#define   DAC_TIM_PERIOD        ((DAC_CNT_FREQ)/(DAC_TRIGG_FREQ))
// Dirección de escritura del DAC para la DMA
#define   DAC_DHR12R1_ADDR                    0x40007408
// Tamaño del buffer del DAC para la DMA
#define   TAM_BUFFER_DAC        ((NSIM*TAM_CON_PREF_REAL)+32)

//ADC defines
#define   ADC_TRIGGER_FREQ      (62500)    // Frecuencia de disparo del ADC
#define   ADC_PRESC                       0   // ADC PRESCALER (/2)
#define   ADC_CNT_FREQ          84000000    // Reloj de TIM2 (prescaled APB1)
// Valor de Autorreload
#define   ADC_TIM_PERIOD        (((ADC_CNT_FREQ)/(ADC_TRIGGER_FREQ)))
#define         ADC1_RDR 0x4001204C
#define         TAM_BUFFER_ADC 1

/* Definición de macros  ------------------------------------------------*/
#define MOD8(VAR)  (VAR&(0x07))                 //VAR%8
#define MOD127(VAR)  (VAR&(0x7E))               //VAR%127
#define MOD4(VAR)  (VAR&(0x03))                 //VAR%4

#define TAM_DATOS 49
```

```
#define TAM_DATOS_COD ((TAM_DATOS*2)+2)       //Número de bits que se
generarian tras cod=(TAM_DATOS*16)+16

#define NBYTESBLOQ 24
#define NBPBLOQ 192

#define NBLOQ ((TAM_DATOS_COD/NBYTESBLOQ)+1)  //Número de bloques que recibe
el interleaver
#define TAM_TRAMA  (NBLOQ*12)
#define TAM_TRAMA_COD (TAM_TRAMA*2)   //Tamaño de la trama tras codificar en
bytes

#define NBYTESMAPPER 12
#define POT 1
#define NSIM (NBLOQ*2)
//Número de portadoras en un símbolo (contando la de referencia)
#define N_PORT 49
//Número de portadoras en un símbolo (contando la de referencia)teniendo en
cuenta parte real e imaginaria
#define N_PORT_2 98
#define N_ELEM_DEMAP   96
/*En el vector de entrada de la IFFT colocamos portadoras desde 2 hasta 99
incluidos*/
#define IND_PORT 2
/*En el vector de entrada de la IFFT colocamos las portadoras conjugadas y
espejadas desde 158 hasta 255 incluidos*/
#define IND_PORT_M 254

#define NFFT_2 256
#define N_PREF_CIC 24
#define TAM_CON_PREF_2 280
/* Índice a partir del cual tomamos las muestras del final que irán en el
prefijo cíclico */
#define INICIO_MUESTRAS_PREF ((TAM_CON_PREF_2)-(N_PREF_CIC))

#define TAM_CON_PREF_REAL 140  //Tamaño de cada símbolo completo con prefijo
cíclico, contando solo la parte real
#define TAM_PREAMB 32  //Tamaño del preámbulo
#define TAM_BUFFER_DATOS_RX ((NSIM)*(TAM_CON_PREF_REAL))     //Tamaño de los
datos recibidos sin preámbulo
#define NIVEL_MIN_RX 1100
#define MAX_CONT_SINC (TAM_PREAMB/2)
#endif
```

tx.c

```
#include "tx.h"
#include "uart.h"
#include "leds.h"
#include "main.h"
#include "varios.h"


/*      Variables de salida de los bloques del Transmisor --------------------
                (las constantes están definidas en main.h)

Datos -> Formar trama -> Cod.conv -> Scrambler -> Interleaver -> Mapper ->
Modulación OFDM(IFFT) + Prefijo cíclico ->
        */
```

```
uint8_t salida_trama[TAM_TRAMA]={0};  //Salida de la función que forma la
trama.
uint8_t salida_cod[TAM_TRAMA_COD]={0};//Salida del codificador convolucional.
uint8_t salida_scr[TAM_TRAMA_COD]={0};//Salida del scrambler.
uint8_t * entrada_int=NULL;    //Puntero auxiliar para pasarle cada bloque de
datos al interleaver.
uint8_t salida_int[NBLOQ*NBYTESBLOQ]={0};     //Salida del interleaver.
uint8_t *entrada_map=NULL;             //Puntero auxiliar para pasarle cada
símbolo al mapper.
float32_t salida_map[N_PORT_2]={0};           //Salida del mapper.

/*____TRANSMISOR_____/
-Recibe-
datos: puntero a la cadena de caracteres de datos de entrada.
tam_datos: tamaño de la cadena datos en bytes.
-------------------------------------------------------------
salida_final: puntero al vector de los símbolos OFDM que obtenemos
tras pasar toda la cadena de bloques del transmisor.
*/
void transmisor(uint8_t *datos, uint32_t tam_datos,float32_t *salida_final)
{
uint32_t ind_aux=0;                   //Índice auxiliar

/* Bloques del Transmisor ------------------------------------------*/

        //____1)Formar trama completa_____///
        rellena_ceros(datos, tam_datos, salida_trama);
        //____2)Codificador convolucional_____///
        cod_conv(salida_trama, TAM_TRAMA,salida_cod);
        //____3)Scrambler_____///
        scrambler(salida_cod, TAM_TRAMA_COD,salida_scr);

        /*      Dividimos en bloques de NBYTESBLOQ para el interleaver.
        */

        while (ind_aux<NBLOQ){
                entrada_int=&(salida_scr[NBYTESBLOQ*ind_aux]);
                //____4)Interleaver_____///
                interleaver (entrada_int,&(salida_int[ind_aux*NBYTESBLOQ]) );
                ind_aux++;
        }

        /*      Al finalizar este bucle obtendremos un vector con todos los
bloques procesados por el interleaver. Ahora dividimos los bloques del
interleaver en símbolos, cada uno de NBYTESMAPPER.En el siguiente bucle se
irá pasando cada símbolo de uno en uno por el mapper y por la función
ifft_prefijo,  colocando cada símbolo OFDM que se obtiene a la salida de la
IFFT (con el prefijo cíclico incluido) en serie en el vector salida_final.*/
        ind_aux=0;
        while (ind_aux<NSIM){
                entrada_map=&(salida_int[NBYTESMAPPER*ind_aux]);
                //____5)Mapper_____///
                dqpsk_mapper(entrada_map, salida_map,N_PORT_2);
                //____6)Mod_IFFT_____///

        ifft_prefijo(salida_map,salida_final+(ind_aux*TAM_CON_PREF_2));
                ind_aux++;
        }
}

/*____1)Formar trama completa_____/
-Recibe-
```

```
datos_entrada: puntero a la cadena de caracteres de datos de entrada.
tam_datos: tamaño de la cadena datos_entrada en bytes.
-------------------------------------------------------------
trama_salida: puntero a la cadena de caracteres de la trama obtenida a la
salida.
*/
void rellena_ceros(uint8_t *datos_entrada, uint8_t tam_datos,uint8_t
*trama_salida){
        uint32_t indice=0;      //Índice auxiliar para el bucle while

        /*      Copiamos los datos al principio de la trama, que ya será del
tamaño adecuado según las constantes de tx.h.
        El resto de la trama estará inicializado ya a cero.
        */
        while(tam_datos>indice){
                *(trama_salida+indice)=*(datos_entrada+indice);
                indice++;
        }
}


/*____2)Codificador convolucional_____/
-Recibe-
entrada: puntero al vector que contiene la trama a codificar a la entrada.
tam_datos: tamaño de entrada en bytes.
-------------------------------------------------------------
salida_cod: puntero al vector codificado obtenido a la salida.
*/
void cod_conv(uint8_t *entrada, uint8_t tam_datos,uint8_t *salida_cod){

        uint16_t registros = 0x0000;            /*      Buffer del codificador
(bits ordenados de izq a derecha: salida<-registros<-entrada) el byte
izquierdo [14:8] contiene los registros de los que obtendremos las salidas el
byte derecho [7:0] contendrá el siguiente byte de datos a codificar. */

        uint16_t out1=0x0000;  //Variable donde obtenemos el primer bit de la
salida.
        uint16_t out2=0x0000;  //Variable donde obtenemos el segundo bit de la
salida.
        uint32_t indice_entrada=0;      //Índice que recorre los datos de
entrada.
        uint32_t indice_salida=16;      //Índice que recorre los datos de
salida.

        for(indice_entrada=0;indice_entrada<tam_datos;indice_entrada++){
                registros&=0xFF00;      //Ponemos a cero los bits [7:0] de
registros para insertar nuevo dato.
                //Insertamos nuevo byte de datos a las posiciones [7:0] de
registros.
                registros|=(entrada[indice_entrada]&0x00FF);

                /*      indice_salida se inicializa en 16 ya que el orden de
los bits de salida va de izquierda a derecha  y tenemos que pasar out 1 y out2
desde la posición más a la derecha hacia la izquierda. */
                indice_salida=16;
                while(indice_salida>0){
                        indice_salida--;
                        registros = registros <<1;

                        //Obtenemos bit de la salida 1.
```

```
        out1=((registros)^(registros>>1)^(registros>>2)^(registros>>3)^(regist
ros>>6));
                        //Lo ponemos en la posicion más a la derecha.
                        out1=(1UL&((out1)>>8));

                        //Obtenemos bit de la salida 2.
        out2=((registros)^(registros>>2)^(registros>>3)^(registros>>5)^(regist
ros>>6));
                        //Lo ponemos en la posicion más a la derecha.
                        out2=(1UL&((out2)>>8));

                        /*Coloco out1 y out 2 alternativamente, de izquierda a
derecha.Como cada byte de entrada genera dos bytes de salida distingo dos
casos. */
                        if(indice_salida>7){
                        /*Resultado en el primer byte de salida.
                                                Colocamos bit de la salida 1. */

        *(salida_cod+(indice_entrada<<1))|=(out1&1UL)<<(MOD8(indice_salida));
                                indice_salida--;
                                //A continuación colocamos bit de la salida 2.

        *(salida_cod+(indice_entrada<<1))|=(out2&1UL)<<(MOD8(indice_salida));
                        }
                        else{
                                /*Resultado en el segundo byte de salida.
                                                Colocamos bit de la salida 1. */

        *(salida_cod+((indice_entrada<<1)+1))|=(out1&1UL)<<(indice_salida);
                                indice_salida--;
                                //A continuación colocamos bit de la salida 2.

        *(salida_cod+((indice_entrada<<1)+1))|=(out2&1UL)<<(indice_salida);
                        }
                }
        }
}

/*____3)Scrambler_____/
-Recibe-
entrada: puntero al vector que contiene la trama a modificar a la entrada.
tam_datos: tamaño de entrada en bytes.
-------------------------------------------------------------
salida_scr: puntero al vector ya aleatorizado obtenido a la salida.
        */
void scrambler(uint8_t *entrada, uint8_t tam_datos,uint8_t* salida_scr){
        uint32_t indice=0;      //Índice para el vector de entrada y el de
salida.
        uint8_t indice_pseudo=0;        //Índice para el vector de la secuencia
pseudoaleatoria.

//Secuencia pseudoaletoria en bytes según el estándar.
uint8_t pseudo[127]={0X0E, 0XF2, 0XC9, 0X02, 0X26, 0X2E, 0XB6, 0X0C, 0XD4,
0XE7, 0XB4, 0X2A, 0XFA, 0X51, 0XB8, 0XFE, 0X1D, 0XE5, 0X92, 0X04, 0X4C, 0X5D,
0X6C, 0X19, 0XA9, 0XCF, 0X68, 0X55, 0XF4, 0XA3, 0X71, 0XFC, 0X3B, 0XCB, 0X24,
0X08, 0X98, 0XBA, 0XD8, 0X33, 0X53, 0X9E, 0XD0, 0XAB, 0XE9, 0X46, 0XE3, 0XF8,
0X77, 0X96, 0X48, 0X11, 0X31, 0X75, 0XB0, 0X66, 0XA7, 0X3D, 0XA1, 0X57, 0XD2,
0X8D, 0XC7, 0XF0, 0XEF, 0X2C, 0X90, 0X22, 0X62, 0XEB, 0X60, 0XCD, 0X4E, 0X7B,
0X42, 0XAF, 0XA5, 0X1B, 0X8F, 0XE1, 0XDE, 0X59, 0X20, 0X44, 0XC5, 0XD6, 0XC1,
0X9A, 0X9C, 0XF6, 0X85, 0X5F, 0X4A, 0X37, 0X1F, 0XC3, 0XBC, 0XB2, 0X40, 0X89,
0X8B, 0XAD, 0X83, 0X35, 0X39, 0XED, 0X0A, 0XBE, 0X94, 0X6E, 0X3F, 0X87, 0X79,
```

```
0X64, 0X81, 0X13, 0X17, 0X5B, 0X06, 0X6A, 0X73, 0XDA, 0X15, 0X7D, 0X28, 0XDC,
0X7F};

                while (indice<tam_datos){
                        indice_pseudo=0;
                        while ((indice_pseudo<127)&(indice<tam_datos)){
//Vamos recorriendo el vector de entrada y haciendo XOR con la secuencia
definida.

        *(salida_scr+indice)=(*(entrada+indice))^(*(pseudo+indice_pseudo++));
                                indice++;
                        }
        }
}

/*____4)Interleaver_____/
-Recibe-
entrada: puntero al vector que contiene los datos a barajar a la entrada.
-------------------------------------------------------------
salida_int: puntero al vector ya barajado obtenido a la salida.
*/

void interleaver (uint8_t *entrada,uint8_t *salida_int ){

        uint8_t indice_aux = 0;
        uint8_t nbyte_i = 0;
        uint8_t nbyte_o = 0;
        uint8_t mask_o=0x00;
        uint8_t mask_i=0x00;
        uint8_t bit_entrada=0x00;
        uint8_t vector_indices [NBPBLOQ]= {0X00, 0X10, 0X20, 0X30, 0X40, 0X50,
0X60, 0X70, 0X80, 0X90, 0XA0, 0XB0, 0X01, 0X11, 0X21, 0X31, 0X41, 0X51, 0X61,
0X71, 0X81, 0X91, 0XA1, 0XB1, 0X02, 0X12, 0X22, 0X32, 0X42, 0X52, 0X62, 0X72,
0X82, 0X92, 0XA2, 0XB2, 0X03, 0X13, 0X23, 0X33, 0X43, 0X53, 0X63, 0X73, 0X83,
0X93, 0XA3, 0XB3, 0X04, 0X14, 0X24, 0X34, 0X44, 0X54, 0X64, 0X74, 0X84, 0X94,
0XA4, 0XB4, 0X05, 0X15, 0X25, 0X35, 0X45, 0X55, 0X65, 0X75, 0X85, 0X95, 0XA5,
0XB5, 0X06, 0X16, 0X26, 0X36, 0X46, 0X56, 0X66, 0X76, 0X86, 0X96, 0XA6, 0XB6,
0X07, 0X17, 0X27, 0X37, 0X47, 0X57, 0X67, 0X77, 0X87, 0X97, 0XA7, 0XB7, 0X08,
0X18, 0X28, 0X38, 0X48, 0X58, 0X68, 0X78, 0X88, 0X98, 0XA8, 0XB8, 0X09, 0X19,
0X29, 0X39, 0X49, 0X59, 0X69, 0X79, 0X89, 0X99, 0XA9, 0XB9, 0X0A, 0X1A, 0X2A,
0X3A, 0X4A, 0X5A, 0X6A, 0X7A, 0X8A, 0X9A, 0XAA, 0XBA, 0X0B, 0X1B, 0X2B, 0X3B,
0X4B, 0X5B, 0X6B, 0X7B, 0X8B, 0X9B, 0XAB, 0XBB, 0X0C, 0X1C, 0X2C, 0X3C, 0X4C,
0X5C, 0X6C, 0X7C, 0X8C, 0X9C, 0XAC, 0XBC, 0X0D, 0X1D, 0X2D, 0X3D, 0X4D, 0X5D,
0X6D, 0X7D, 0X8D, 0X9D, 0XAD, 0XBD, 0X0E, 0X1E, 0X2E, 0X3E, 0X4E, 0X5E, 0X6E,
0X7E, 0X8E, 0X9E, 0XAE, 0XBE, 0X0F, 0X1F, 0X2F, 0X3F, 0X4F, 0X5F, 0X6F, 0X7F,
0X8F, 0X9F, 0XAF, 0XBF};

        indice_aux=0;
        while (indice_aux<NBPBLOQ){
                //Sacamos a partir del vector de índices la posicion del byte
de entrada = indice/8.
                nbyte_i=(vector_indices[indice_aux])>>3;
                //Sacamos a partir del vector de índices la posicion del byte
de salida = indice/8.
                nbyte_o=indice_aux>>3;
                //Sacamos a partir del vector de índices la posicion del bit
de entrada = indice%8.
                mask_i=(0x80>>((vector_indices[indice_aux])&(0x07)));
                //Sacamos a partir del vector de índices la posicion del bit
de salida = indice%8.
                mask_o=(0x80>>((indice_aux)&(0x07)));
```

```
                //Hacemos reset en ese bit a la salida.
                (*(salida_int+nbyte_o))&=(~(mask_o));
                //Obtenemos valor de ese bit a la entrada.
                bit_entrada=(*(entrada+nbyte_i))&(mask_i);
                //Comprobamos valor del bit.
                if(bit_entrada!=0){
                        /*Si es cero ya hemos hecho reset antes, si no,
ponemos ese bit en la salida a uno. */
                        (*(salida_int+nbyte_o))|=(mask_o);
                }
                indice_aux++;
        }
}


/*____5)Mapper_____/
-Recibe-
bits_entrada: puntero al vector que contiene los bits que tendremos que
mapear.
n_elem_portadoras: número de portadoras x 2 (cada portadora tiene parte real
e imaginaria).
-------------------------------------------------------------
portadoras_salida: puntero al vector de portadoras resultantes, cada una
compuesta de parte real e imaginaria.
*/

void dqpsk_mapper(uint8_t *bits_entrada,float32_t *portadoras_salida,uint8_t
n_elem_portadoras){
        uint16_t indice_bits = 0;       //Índice de los datos de entrada
(bits).
        uint16_t indice_portadoras = 0; //Índice de los datos de salida
        (portadoras: parte real e imaginaria).
        uint8_t bits_aux = 0x00;        //Variable auxiliar para guardar los 2
bits que vamos a mapear.
        uint8_t mask=0xC0;
        uint8_t port_act=0;

        /*      Insertamos la portadora inicial
                        POT+0j.
        */
                        *(portadoras_salida+indice_portadoras++)=POT;
                        *(portadoras_salida+indice_portadoras++)=0;

        while (indice_portadoras<n_elem_portadoras){
                //Cogemos los dos bits siguientes para mapear.
                bits_aux=(*(bits_entrada+(indice_bits>>3)));
        //Primero cogemos el byte de entrada correspondiente al índice.
                bits_aux=bits_aux&(mask>>(MOD8(indice_bits)));
        //Ahora tomamos los dos bits que toquen.

/*      Ponemos los dos bits al final (posiciones 0 y 1 desde la derecha) y
todo lo demás a cero.
                */
                bits_aux=(bits_aux>>(6-(MOD8(indice_bits))))&(mask>>6);

                //Dependiendo de los dos bits que hemos cogido sumamos el
incremento de fase correspondiente.
                switch ( bits_aux ) {
                        //Recibido 00-> Gray -> +0*pi/2
                case 0:
                        port_act= MOD4(port_act);
                break;
```

```
                        //Recibido 01   -> Gray -> +1*pi/2
                case 1:
                        port_act= MOD4(port_act+1);
                break;
                        //Recibido 10   -> Gray -> +3*pi/2
                case 2:
                        port_act= MOD4(port_act+3);
                break;
                        //Recibido 11 -> Gray -> +2*pi/2
                case 3:
                        port_act= MOD4(port_act+2);
                break;
                default:
                        //En caso de error encendemos los leds.
                        led_on(all_leds);
                break;
        }
                indice_bits+=2;

                switch (port_act) {

                case 0:
                        //Portadora = POT+0j
                        *(portadoras_salida+indice_portadoras++)=POT;
                        *(portadoras_salida+indice_portadoras++)=0;
                break;

                case 1:
                        //Portadora = 0+POTj
                        *(portadoras_salida+indice_portadoras++)=0;
                        *(portadoras_salida+indice_portadoras++)=POT;
                break;

                case 2:
                        //Portadora = -POT+0j
                        *(portadoras_salida+indice_portadoras++)=-POT;
                        *(portadoras_salida+indice_portadoras++)=0;
                break;

                case 3:
                        //Portadora = 0-POTj
                        *(portadoras_salida+indice_portadoras++)=0;
                        *(portadoras_salida+indice_portadoras++)=-POT;
                break;

                default:
                        //En caso de error encendemos los leds.
                        led_on(all_leds);
                break;
                }
        }
}

/*____6)IFFT + Prefijo cíclico_____/
-Recibe-
portadoras_mapper: puntero al vector de portadoras resultantes del mapper,
cada una compuesta de parte real e imaginaria.
-----------------------------------------------------------
salida_ifft_pref: puntero al vector de la salida de la IFFT, incluyendo el
prefijo cíclico.*/
void ifft_prefijo(float32_t *portadoras_mapper,float32_t *salida_ifft_pref){
```

```
/*Colocamos las portadoras de forma que el vector resultante posea simetría
hermítica para que la salida de la IFFT sea real. También tenemos que tener
en cuenta que el vector de salida es de TAM_CON_PREF_2 elementos para que
quepa el prefijo cíclico.
               Se utilizarán las funciones de la librería arm_math.h.*/
//Colocamos las primeras 49 portadoras.
        arm_copy_f32 (portadoras_mapper, salida_ifft_pref+IND_PORT+N_PREF_CIC,
N_PORT_2);
//Colocamos las portadoras conjugadas y espejadas.
        copia_conj_decre (portadoras_mapper,
salida_ifft_pref+IND_PORT_M+N_PREF_CIC, N_PORT_2);
//Calculamos IFFT de 128 puntos, con bit reversal activado.
        arm_cfft_f32(&arm_cfft_sR_f32_len128, salida_ifft_pref+N_PREF_CIC, 1,
1);

//Añadimos prefijo cíclico.
        arm_copy_f32 (salida_ifft_pref+INICIO_MUESTRAS_PREF,
salida_ifft_pref,N_PREF_CIC );

}
```

rx.c

```
#include "rx.h"
#include "uart.h"
#include "leds.h"
#include "main.h"
#include "varios.h"

/*      Variables de salida de los bloques del Receptor ----------------------
               (las constantes están definidas en main.h)

->Datos recibidos -> Quitar pref.cíclico -> Demodulación (FFT) -> Demapper ->
Deinterleaver -> Descrambler -> Decodificador Viterbi->
               ->Deshacer trama       */

float32_t salida_fft[N_PORT_2]={0};
float32_t salida_demap[N_ELEM_DEMAP*NSIM]={0};        //Salida del demapper
float32_t salida_deint[NBPBLOQ*NBLOQ]={0};    //Salida del deinterleaver
float32_t salida_descr[NBPBLOQ*NBLOQ];        //Salida del descrambler

/*____RECEPTOR_____/
-Recibe-
datos_rx: puntero al vector de datos recibidos.
n_simb: número de símbolos de datos_rx.
-----------------------------------------------------------
caracteres_tx: puntero al vector de caracteres que obtenemos
tras pasar toda la cadena de bloques del receptor.
*/
//void receptor(float32_t *datos_rx, uint8_t n_elem,uint8_t *caracteres_tx)
void receptor(float32_t *datos_rx, uint32_t n_simb,float32_t *caracteres_tx)
{

uint32_t ind_aux=0;                    //Índice auxiliar
uint32_t ind_aux2=0;                   //Índice auxiliar

/* Bloques del Receptor ------------------------------------------------*/
```

```
        while (ind_aux<NSIM){
                //____1)Quitar prefijo ciclico + Demod_FFT_____///
                prefijo_fft(datos_rx+(ind_aux*TAM_CON_PREF_2),salida_fft);

                //____2)Demapper_____///
                dqpsk demap(salida fft,salida demap+(ind aux*N ELEM DEMAP));
                ind_aux++;
        }


        /*      Tomamos bloques de NBYTESBLOQ para el interleaver.
        */

        while (ind_aux2<NBLOQ){
                //____3)Deinterleaver_____///
                deinterleaver
(salida_demap+(ind_aux2*NBPBLOQ),salida_deint+(ind_aux2*NBPBLOQ));
                ind_aux2++;
        }


        //____4)Descrambler_____///
        descrambler(salida_deint, NBPBLOQ*NBLOQ,salida_descr);

        arm_copy_f32 (salida_descr, caracteres_tx, NBPBLOQ*NBLOQ);
}

/*____1)Quitar prefijo ciclico + FFT_____/
-Recibe-
recibido: puntero a los datos recibidos.
-------------------------------------------------------------
datos_fft_out: puntero al vector de portadoras recibidas obtenido a la salida
de la FFT.
*/

void prefijo_fft(float32_t *recibido,float32_t *datos_fft_out){
        float32_t vauxiliar[NFFT_2]={0};              //Variable auxiliar para
los cálculos.

        //Ignoramos prefijo ciclico
        arm_copy_f32 (recibido+N_PREF_CIC, vauxiliar, NFFT_2);
        //Calculamos FFT de 128 puntos, con bit reversal activado.
  arm_cfft_f32(&arm_cfft_sR_f32_len128, vauxiliar, 0, 1);

/*      Tomamos las portadoras del vector de entrada, colocadas dónde indica
IND_PORT, ignorando el prefijo cíclico:*/
        arm_copy_f32 (vauxiliar+IND_PORT, datos_fft_out, N_PORT_2);

}



/*____2)Demapper_____/
-Recibe-
portadoras_fft: puntero al vector de portadoras recibidas como entrada,
obtenido a la salida de la FFT.
-------------------------------------------------------------
salida_dem: puntero al vector que contendrá la información sobre cada bit
transmitido obtenido tras demapear.
*/

void dqpsk_demap(float32_t *portadoras_fft,float32_t *salida_dem){
```

```
        float32_t variable_aux[N_PORT_2];                       //variable
auxiliar para los cálculos

        //Variable dónde guardamos el valor de [sqrt(2)/2]+[sqrt(2)/2]i para
girar la constelación
        float32_t raiz_2[2]={0.707106781186548,0.707106781186548};
        /*Para obtener las portadoras origen de la modulación diferencial
tenemos que obtener:-> cdem=c(2:end).*conj((c(1:end-1))) (escrito en
Matlab)*/
        //Guardamos conj((c(1:end-1)) en variable_aux
        arm_cmplx_conj_f32 (portadoras_fft, variable_aux, N_PORT);
        //Multiplicamos por c(2:end)
        arm_cmplx_mult_cmplx_f32 (portadoras_fft+2, variable_aux, salida_dem,
N_PORT);
        /*      Obtenidas las portadoras recibidas, giramos la constelación un
ángulo de pi/4. Para ello, se usa una función de la CMSIS-DSP modificada:
        */
        vectorxnumero_complejo_f32(salida_dem,raiz_2,variable_aux,N_PORT);
        salida_dem++;
        /*      Tras el giro de la constelación, la información del primer bit
de cada portadora queda en la parte imaginaria, mientras que la información
del segundo bit queda en la parte real, por lo que debemos reordenar la
salida para que al interleaver le llegue:[info 1er bit | info 2o bit |info
1er bit | info 2o bit |...]: */
        intercambia_real_im (variable_aux,salida_dem, N_PORT_2);

}

/*____3)Deinterleaver_____/
-Recibe-
datos: puntero al bloque de elementos a la entrada.
-----------------------------------------------------------
salida: puntero al bloque de elementos a la salida tras desentrelazar.
*/

void deinterleaver (float32_t *datos_deint,float32_t *salida_deint ){
        uint8_t indice = 0;
        //Tabla de índices definida según el estándar
        const uint8_t tabla_indices[NBPBLOQ]={0X00, 0X0C, 0X18, 0X24, 0X30,
0X3C, 0X48, 0X54, 0X60, 0X6C, 0X78, 0X84, 0X90, 0X9C, 0XA8, 0XB4, 0X01, 0X0D,
0X19, 0X25, 0X31, 0X3D, 0X49, 0X55, 0X61, 0X6D, 0X79, 0X85, 0X91, 0X9D, 0XA9,
0XB5, 0X02, 0X0E, 0X1A, 0X26, 0X32, 0X3E, 0X4A, 0X56, 0X62, 0X6E, 0X7A, 0X86,
0X92, 0X9E, 0XAA, 0XB6, 0X03, 0X0F, 0X1B, 0X27, 0X33, 0X3F, 0X4B, 0X57, 0X63,
0X6F, 0X7B, 0X87, 0X93, 0X9F, 0XAB, 0XB7, 0X04, 0X10, 0X1C, 0X28, 0X34, 0X40,
0X4C, 0X58, 0X64, 0X70, 0X7C, 0X88, 0X94, 0XA0, 0XAC, 0XB8, 0X05, 0X11, 0X1D,
0X29, 0X35, 0X41, 0X4D, 0X59, 0X65, 0X71, 0X7D, 0X89, 0X95, 0XA1, 0XAD, 0XB9,
0X06, 0X12, 0X1E, 0X2A, 0X36, 0X42, 0X4E, 0X5A, 0X66, 0X72, 0X7E, 0X8A, 0X96,
0XA2, 0XAE, 0XBA, 0X07, 0X13, 0X1F, 0X2B, 0X37, 0X43, 0X4F, 0X5B, 0X67, 0X73,
0X7F, 0X8B, 0X97, 0XA3, 0XAF, 0XBB, 0X08, 0X14, 0X20, 0X2C, 0X38, 0X44, 0X50,
0X5C, 0X68, 0X74, 0X80, 0X8C, 0X98, 0XA4, 0XB0, 0XBC, 0X09, 0X15, 0X21, 0X2D,
0X39, 0X45, 0X51, 0X5D, 0X69, 0X75, 0X81, 0X8D, 0X99, 0XA5, 0XB1, 0XBD, 0X0A,
0X16, 0X22, 0X2E, 0X3A, 0X46, 0X52, 0X5E, 0X6A, 0X76, 0X82, 0X8E, 0X9A, 0XA6,
0XB2, 0XBE, 0X0B, 0X17, 0X23, 0X2F, 0X3B, 0X47, 0X53, 0X5F, 0X6B, 0X77, 0X83,
0X8F, 0X9B, 0XA7, 0XB3, 0XBF};

        //Vamos intercambiando la salida con la entrada:
        while (indice<NBPBLOQ){
                *(salida_deint+indice)=*(datos_deint+tabla_indices[indice]);
                indice++;
        }
}
```

```
/*____4)Descrambler_____/
-Recibe-
entrada: puntero al bloque de elementos a la entrada.
tam_datos: tamaño de entrada.
-------------------------------------------------------------
salida_d: puntero al bloque de elementos a la salida tras desbarajar.
*/

void descrambler(float32_t *entrada, uint32_t tam_datos,float32_t* salida_d){

	uint32_t contador=0;
	uint32_t cociente_127=0; //Número de grupos de 127 bits que hay a la
entrada.
	uint32_t n_restante=0; //Número de bits sobrantes al dividir en grupos
de 127 bits.
	uint32_t i_restante=0; //Índice dónde comienzan los n_restante bits
restantes.

	/*Esta es la secuencia pseudoaleatoria definida en el estándar, pero
	sustituimos cada 1 de la secuencia original por un cambio de signo,
	mientras que si hay un 0, mantenemos el mismo valor:
	*/
	float32_t pseudo[127]={1, 1, 1, 1, -1, -1, -1, 1, -1, -1, -1, -1, 1,
1, -1, 1, -1, -1, 1, 1, -1, 1, 1, -1, 1, 1, 1, 1, 1, 1, -1, 1, 1, 1, -1, 1,
1, -1, -1, 1, 1, 1, -1, 1, -1, -1, -1, 1, -1, 1, -1, -1, 1, -1, -1, 1, 1, 1,
1, 1, -1, -1, 1, 1,-1, -1, 1, -1, 1, -1, 1, 1, -1, -1, -1, 1, 1, -1, -1, -1,
-1, 1, -1, -1, 1, -1, 1, 1, 1, 1, -1, 1, -1, 1, -1, 1, -1, -1, -1, -1, -1, 1,
-1, 1, 1, -1, 1, -1, 1, 1, 1, -1, -1, 1, -1, -1, -1, 1, 1, 1, -1, -1, -1, -1,
-1, -1, -1};

	//Calculamos cuántos grupos de 127 bits hay en la entrada.
	cociente_127=tam_datos/127;

	//Multiplicamos por la secuencia cada grupo de 127 bits.
	while (contador<cociente_127){
		arm_mult_f32 (entrada+(contador*127), pseudo,
salida_d+(contador*127), 127);
		contador++;
	}
	//Índice dónde comienzan los n_restante bits restantes.
	i_restante=cociente_127*127;
	//Ahora calculamos los bits restantes.
	n_restante=tam_datos-i_restante;
	//Multiplicamos los bits restantes por el trozo conveniente de la
secuencia.
	arm_mult_f32 (entrada+i_restante, pseudo, salida_d+i_restante,
n_restante);
}
```

REFERENCIAS

[1] H. Haas, “Wireless data from every light bulb”, video del TEDGlobal Aug 2011.

[2] IEEE Standard for Local and Metropolitan Area Networks--Part 15.7: Short-Range Wireless Optical Communication Using Visible Light", *IEEE Std 802.15.7-2011*, vol., no., pp.1, 309, Sept. 6 2011.

[3] X. Bao, G. Yu, J. Dai, and X. Zhu, "*Li-Fi*: Light fidelity-a survey," Wireless Networks, p. 11, 01/ 18/2015.

[4] C. W. Chow, C. H. Yeh, Y. F. Liu, P. Y. Huang, and Y. Liu, "Adaptive scheme for maintaining the performance of the in-home white-LED visible light wireless communications using OFDM," *Optics Communications*, vol. 292, pp. 49-52, 4/1/2013.

[5] M. Frishberg, "Li-Fi Lighting the Way," *Research Technology Management*, vol. 58, pp. 7-8, Jan/Feb, 2015.

[6] Sowjanya Gottumukkala, Kumari Pasupuleti, "Li-Fi (Light Fidelity) - The Future Technology in Wireless Communication," *Discovery Daily*, vol. 28, nº103, pp. 29-33, 2015.

[7] H.Haas, “Li-Fi: High Speed Wireless Communications via Light Bulbs”, *IEEE Webinar*, 19 June 2013.

[8] Afgani, M.Z.; Haas, H.; Elgala, H.; Knipp, D., "Visible light communication using OFDM," *2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, pp.6, pp.134, 2006.

[9] Web de “Li-Fi Consortium”

[10] Web de “PureLifi”

[11] Chang, R. W., "Synthesis of band-limited orthogonal signals for multi-channel data transmission", *Bell System Technical Journal*, vol 45, issue 10, pp. 1775 – 1796, 1966.

[12] Weinstein, S.; Ebert, P., "Data Transmission by Frequency-Division Multiplexing Using the Discrete Fourier Transform," *IEEE Transactions on Communication Technology*, vol.19, no.5, pp.628-634, 1971.

[13] Vicente Baena, “Introducción a OFDM”, *Apuntes de la asignatura Sistemas Electrónicos de Comunicaciones de GITT de la ETSI de Sevilla.*

[14] Juan Eloy De los Ángeles Barriga, “Estudio y simulación de un sistema ACO-OFDM para comunicaciones ópticas inalámbricas”, *Proyecto Fin de Carrera de Ingeniería de Telecomunicación de la Escuela Técnica Superior de Ingeniería de la Universidad de Sevilla*, 2014.

[15] Web de Mathworks, “Documentación sobre el *Communications System Toolbox de Matlab*”.

[16] Charan Langton, “Tutorial 12 – Convolutional Coding and Decoding Made Easy”, *Tutoriales de la web “Complex to Real”.*

[17] Prof. Dr.-Ing. N. When, práctica de un laboratorio de microelectrónica: "Versuch 7: Implementing Viterbi Algorithm in DLX", *Technische Universität Kaiserslautern.*

[18] PRIME Alliance Technical Working Group, "Draft Specification for PoweRline Intelligent Metering Evolution", *versión 1.3.6.*

[19] Armstrong, J., "OFDM for Optical Communications", *Journal of Lightwave Technology*, vol.27, no.3, pp.189,204, Feb.1, 2009.

[20] Mesleh, R.; Elgala, H.; Haas, H., "On the Performance of Different OFDM Based Optical Wireless Communication Systems", *Journal of Optical Communications and Networking*, IEEE/OSA, vol.3, no.8, pp.620,628, August 2011.

[21] UM1472: Manual de usuario, "Discovery kit for STM32F407/417 lines"

[22] RM0090: Manual de referencia, "STM32F40xxx, STM32F41xxx, STM32F42xxx, STM32F43xxx advanced ARM-based 32-bit MCUs"

[23] Web oficial, "STM32F4DISCOVERY"-

[24] Web oficial, "Realterm"

[25] Web oficial, "CMSIS-DSP Software Library Reference"

[26] Martin, Trevor (2013). *The designer's guide to the Cortex-M processor family: a tutorial approach.* Waltham, MA: Newnes.

[27] Yiu, Joseph (2014). *The definitive guide to ARM Cortex-M3 and Cortex-M4 processors.* Waltham, MA: Newnes.

[28] AN3116: Nota de aplicación, "STM32's ADC modes and their applications"