Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA ELECTRONÍCA DE COMUNICACIONES (GIEC)

**Trabajo Fin de Grado**

# Sistema de sincronización de disparo de cámaras y LiDAR basado en tarjeta STM32F767 para un vehículo autónomo



**Autor:** Sergio Manzano Pasabados

**Tutor:** Ignacio Parra Alonso

**Cotutor:** Álvaro Quintanar Pascual

2021

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

**Grado en Ingeniería Electrónica de Comunicaciones**

Trabajo Fin de Grado
# Sistema de sincronización de disparo de cámaras y LiDAR basado en tarjeta STM32F767 para un vehículo autónomo

**Autor:** Sergio Manzano Pasabados

**Tutor:** Ignacio Parra Alonso

**Cotutor:** Álvaro Quintanar Pascual

**TRIBUNAL:**

**Presidente:** Melquiades Carbajo Martín

**Vocal 1º:** Noelia Hernández Parra

**Vocal 2º:** Ignacio Parra Alonso

**FECHA**: 14 de Mayo del 2021

# Índice general

# Índice de Figuras

# Índice de tablas

## Resumen

En el presente TFG se realiza un sistema de sincronización del paso de un Velodyne LiDAR (*Laser Imaging Detection and Ranging*) con el disparo de varias cámaras, todo ello montado sobre un vehículo autónomo. El LiDAR proporciona un escenario 3D, pero no es suficiente para conocer con exactitud el terreno, por lo que se incorporan cámaras externas que son de utilidad para conseguir mayor información del entorno. El sistema sincroniza el disparo de las cámaras con la posición del LiDAR, de forma que cuando este se encuentre en ciertos ángulos las cámaras realizan una captura.

Se utiliza un microcontrolador para gestionar el sincronismo y generar la señal de disparo de las cámaras, así como para extraer y gestionar la información que envía el Velodyne por paquetes UDP (*User Datagram Protocol*) a través de Ethernet.

## Palabras clave

LiDAR, sincronización, cámaras, microcontrolador, ángulos de disparo.

## Summary

In the present Bachelor's thesis, a synchronization system of the passage of a Velodyne LiDAR (*Laser Imaging Detection and Ranging*) is carried out with the firing of several cameras, all of it is mounted on an autonomous vehicle. The LiDAR provides a 3D scene, but it is not enough to know exactly the terrain, so external cameras are incorporated that are useful to obtain more information about the environment. The system synchronizes the shooting of the cameras with the position of the LiDAR, so that when it is at certain angles, the cameras are triggered, recording a frame.

A microcontroller is used to manage the synchronism and generate the trigger signal from the cameras, as well as to extract and manage the information sent by the Velodyne by UDP (*User Datagram Protocol*) packets over Ethernet.

## Glosario de acrónimos y abreviaturas

| | |
|---|---|
| ADC | Analog to Digital Converter |
| AHB | Advanced High-perfomance Bus |
| APB | Advanced Peripheral Bus |
| CAN | Controller Area Network |
| CCD | Charge Coupled Device |
| CMOS | Complementary Metal Oxide Semiconductor |
| CNN | Convolutional neural network |
| DAC | Digital to Analog Converter |
| DHCP | Dynamic Host Configuration Protocol |
| DSP | Digital Signal Processor |
| FMC | Control de Memoria Flexible |
| FOV | Field Of View |
| FPGA | Field Programmable Gate Array |
| FPU | Floating Point Unit |
| GPS | Global Positioning System |
| HDMI-CEC | High Definition Multimedia Interface-Consumer Electronics Control |
| HVAC | Heating Ventilating and Air Conditioning |
| I2C | Inter-Integrated Circuit |
| I2S | Integrated Interchip Sound |
| IDE | Integrated Development Environment |
| IoT | Internet of Things |
| IP | Internet Protocol |
| IR | Infared Radiation |
| Laser | Light Amplification by Stimulated Emission of Radiation |
| LED | Light Emitting Diode |
| LiDAR | Laser Imaging Detection and Ranging |
| LwIP | Lightweight IP |
| MPU | Unidad de Protección de Memoria |
| PLC | Programmable Logic Controller |
| PLL | Phase Locked Loop) |
| PWM | Pulse Width Modulation |
| RISC | Reduced Instruction Set Computer |
| RTC | Real Time Clock |
| SAI | Interfaces de Audio en Serie |
| SDMMC | Secure Digital and MultiMediaCard |
| SPI | Serial Peripheral Interface |
| TCM | Tightly-Coupled Memory |
| TCP | Transmission Control Protocol |
| TFT-LCD | Thin Film Transistor-Liquid Crystal Display |
| ToF | Time of flight |
| UART | Universal Asynchronous Receiver-Transmitter |
| UDP | User Datagram Protocol |
| USART | Universal Synchronous/Asynchronous Receiver Transmitter |
| USB | Universal Serial Bus |
| UTC | Tiempo Universal Coordinado |

# 1. Introducción al trabajo

Este TFG trata sobre la realización de un sistema de sincronización entre el disparo de varias cámaras Point Grey y el paso de un LiDAR por varias posiciones. El fin del proyecto es formar parte de un vehículo autónomo, por lo que irá instalado en él. La idea de la conducción autónoma ha existido desde el origen del vehículo, pero esa idea futurista en la actualidad se ha convertido en realidad. Hay una gran cantidad de investigaciones sobre el vehículo autónomo con unos resultados muy positivos, llevadas a cabo por grandes empresas del sector automovilístico y gigantes tecnológicos.

Aunque a esta idea aún le queda camino por recorrer, se están dando los primeros pasos para que sea posible. Para ello se ha de dotar al vehículo de elementos software y hardware, como el LiDAR, que le permitan conocer su entorno e interpretarlo en tiempo real, además de ser capaz de reaccionar frente a los distintos sucesos que puedan ocurrir durante su trayecto.

El LiDAR proporciona una imagen 3D del entorno en tiempo real, pero a causa del movimiento del vehículo y la propia precisión del Velodyne, para este proyecto, se ha requerido el uso de las cámaras externas para conseguir mejores resultados y conocer el entorno de una forma más aproximada.

Un dispositivo LiDAR es un sistema de medición y detección de objetos mediante el empleo de láseres. Este tipo de sensor es uno de los más relevantes sobre los que se puede esperar que evolucione la conducción autónoma de los vehículos. Este dispositivo consiste de manera muy básica, en un foco emisor de haces de rayos láser infrarrojos, y de una lente receptora infrarroja capaz de detectar esos rayos. Los rayos láser que se emiten impactan sobre los objetos reflejándose, siendo detectados posteriormente por la lente. Para determinar la distancia con un objeto, el LiDAR mide el tiempo de retraso entre la emisión y la recepción de los rayos.

Aunque algunos LiDAR son fijos, lo típico es que sea un dispositivo que gire 360 grados sobre sí mismo para cubrir todo el entorno, como es el caso de este sistema. Además, con el uso de varios láseres, el Velodyne es capaz de realizar un barrido en columna para abarcar un terreno mayor. De esta manera el LiDAR obtiene una nube de puntos del entorno pudiendo generar una imagen tridimensional en tiempo real, que se actualiza permanentemente. Dentro de esta nube se conoce la posición en el espacio y la distancia hasta cada uno de sus puntos.

Las cámaras externas no van a realizar una captura del terreno en los 360 grados, sino que lo van a hacer en puntos concretos. El LiDAR estará en movimiento recogiendo datos del entorno y, cuando el barrido del láser pase por los ángulos configurados, se deberá generar la señal de disparo de la cámara correspondiente para tomar una captura. Para conseguir sincronizar el sistema es necesario conocer la posición en la que se encuentra el Velodyne en cada momento, información que manda a través de Ethernet junto con el resto de datos. Conociendo los ángulos en los que se quiere generar el disparo y el ángulo en el que se encuentra el LiDAR se puede sincronizar el sistema.

Se ha seleccionado la tarjeta STM32 Nucleo-144 (F767ZI) de la familia STMicroelectronics para llevar a cabo las tareas de gestión del sincronismo y generación de la señal de disparo de las cámaras, así como para la extracción y manejo de la información que envía el Velodyne por paquetes UDP a través de Ethernet. Por lo tanto, se ahondará en el estudio de una tarjeta diferente a la utilizada durante el grado.

En cuanto a las herramientas, se han empleado Keil uVision, STM32CubeMX, Wireshark y las aplicaciones Packet Sender y PReplay. El primer programa es un compilador que se utiliza para la programación en C, depuración y carga del código en el microcontrolador. El segundo, permite una fácil configuración de los microcontroladores y microprocesadores STM32 a través de un entorno gráfico, así como la generación del código C de inicialización correspondiente.

La herramienta Wireshark es de gran utilidad, ya que permite monitorizar el tráfico que pasa a través de la red, y de esta forma ver todos los paquetes UDP que el Velodyne entrega y el formato que tienen.

Las dos aplicaciones restantes se han utilizado como recurso para trabajar desde casa sin tener que acudir al laboratorio para trabajar directamente con el LiDAR. Con PReplay se ha replicado una emisión de paquetes UDP reales del LiDAR, gracias a un archivo pcap grabado previamente. Packet Sender envía paquetes UDP y TCP (*Transmission Control Protocol*) pudiendo seleccionar el contenido, el puerto y la dirección de destino.

El informe tendrá un desarrollo escalonado. Primero se describirán los diferentes dispositivos que se utilizan, para luego seguir con el desarrollo del trabajo: desde que el tutor me dio el microcontrolador, hasta escribir el inicio de este informe, incluyendo los problemas que me he encontrado durante su realización. También se incluirá el uso que se ha dado a todas las herramientas que se han empleado y varias imágenes para describir cada avance.

# 2. Estado del arte

La idea de los vehículos autónomos no es nueva, pero hasta la actualidad no se habían dado los pasos necesarios para realizarla, ya sea por la falta de tecnología o por otras circunstancias. Las mayores empresas tecnológicas del mundo están apostando por llevar a cabo esta idea, aunque algunas de ellas han abandonado el intento, y otras se han centrado en el software. Si hay que destacar una empresa que encabeza estas investigaciones es Tesla.

Un vehículo autónomo es un sistema complejo donde se involucran muchas tecnologías y entran en juego diferentes disciplinas como la mecánica, visión artificial o geoposicionamiento.

En diferentes estudios sobre los vehículos autónomos como «*Camera-LiDAR sensor fusion in real time*» [6] y «*Fusing vision and LiDAR - Synchronization, correction and occlusion reasoning*» [7], entre otros, se muestran unos problemas comunes en todos los proyectos relacionados con este tema.

Entre esos problemas se encuentra la sincronización de los datos de los diferentes sensores. En estos proyectos se emplean uno o varios sensores LiDAR y varias cámaras. Estos dispositivos por sí solos proporcionan una información limitada, pero al combinarlos se crea un sistema similar a un ojo, distinguiendo colores, características y distancias.

Entre los modelos de LiDAR utilizados se encuentra el Velodyne VLP-16 y el Velodyne HDL-64E. Ambos dispositivos tienen una aplicación similar, únicamente varían sus características como la calidad de los datos recogidos, o el número de láseres incorporados en cada LiDAR. Sin ir más lejos, el modelo Velodyne HDL-64E es una versión mejorada del HDL-32E que se va a emplear en este TFG. En cuanto a las cámaras encontramos un escenario similar, distintos modelos entre los que hay pequeñas variaciones en sus características, pero se emplean para un mismo objetivo.

La fusión de los datos de estos elementos requiere que sean capturados y procesados dentro de un mismo corto periodo de tiempo, para que el vehículo pueda adaptarse a su entorno. Para ello resulta crucial la sincronización de estos datos, cuyo problema es común en todo tipo de sistemas de vehículos autónomos.

En cada estudio se utilizan diferentes herramientas para gestionar y conseguir el sincronismo entre los dispositivos. En este caso se va a utilizar una tarjeta STM32 para realizar estas tareas.

Además del uso del LiDAR y las cámaras, las diferentes líneas de estudio de los vehículos autónomos llevan a estos proyectos a incorporar otras herramientas para solucionar los problemas que encuentran o mejorar los resultados obtenidos como las técnicas de deep learning.

Otros estudios más específicos se centran en entornos concretos donde puede circular un vehículo autónomo, como rotondas o terrenos no estructurados, ya sean caminos rurales, bosques, etc.

En la Figura 1 que se muestra a continuación se aprecia el esquema de un vehículo autónomo equipado con varios sensores para cubrir las necesidades que se requieran.

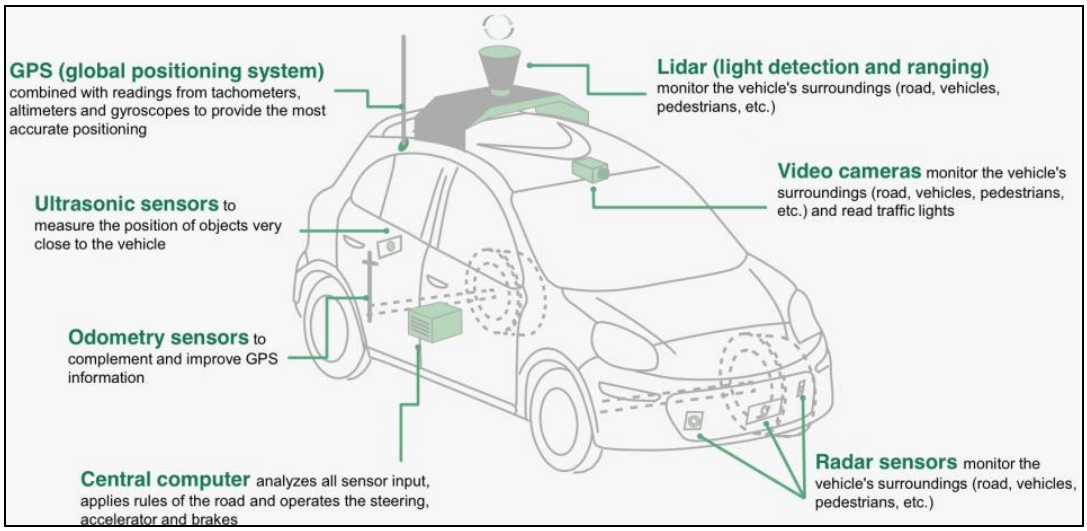



**Figura 1. Esquema general vehículo autónomo. [8]**

En este proyecto se van a tratar los temas de sincronismo entre el disparo de las cámaras y el paso del Velodyne, dejando abierta la posibilidad de en un futuro seguir ampliando el proyecto complementando el resto de necesidades o ampliando las capacidades del sistema creado.

Los datos y resultados que se van a obtener pueden servir como complemento y ayuda para otros proyectos como en «*CMRNet: Camera to LiDAR-Map Registration*» [9] donde se va a emplear una CNN (*Convolutional neural network*) que aprende a hacer coincidir las imágenes tomadas por una cámara con el mapa en 3D generado por un LiDAR.

# 3. Descripción del trabajo desarrollado

En este capítulo se llevará a cabo una descripción general del trabajo y de cada uno de los dispositivos utilizados, explicando la parte teórica relacionada con este TFG. A continuación, se desarrollarán los conceptos teóricos utilizados para la realización del trabajo, las pruebas que se han llevado a cabo y los problemas que han aparecido.

En primer lugar, se va a realizar una breve explicación sobre cada parte del proyecto y cómo están interconectadas. Uno de los dispositivos principales del sistema es el LiDAR, que se ubicará en el vehículo autónomo. Utilizando como apoyo el funcionamiento general de un LiDAR de la Figura 2, se puede observar que la forma en la que realiza el barrido láser del entorno, por características de ratio de frames y distancia, es vertical. Este dispositivo gira sobre sí mismo 360 grados generando una nube de puntos de su entorno.

Mediante el envío de paquetes UDP a través de Ethernet el Velodyne manda los datos que recoge, además de otra información útil como su posición a la tarjeta STM32 conectada al otro extremo del Ethernet, con la que se gestionan los paquetes UDP que recibe y extrae la información que se necesita, además de coordinar el sincronismo entre las cámaras del sistema y el LiDAR.

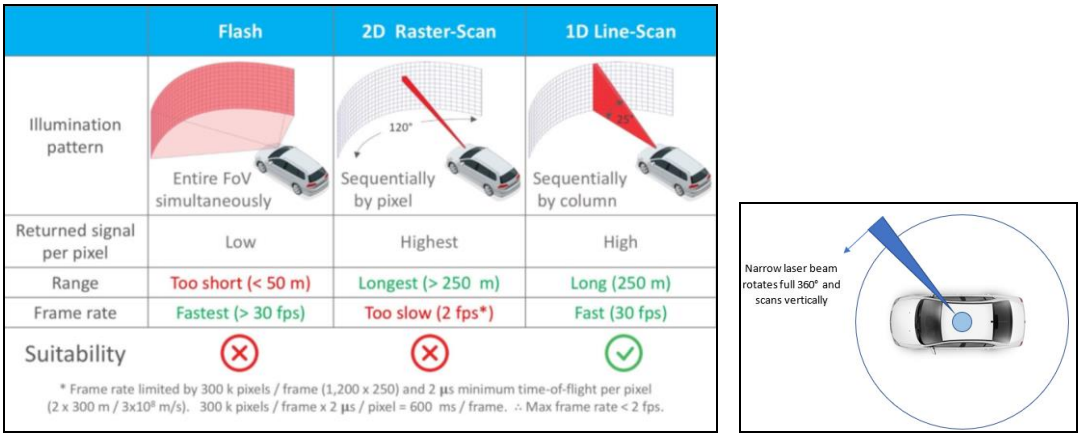



**Figura 2. Barrido láser del LiDAR y rotación de 360 grados alrededor del vehículo. [10]**

Las cámaras externas son otro de los dispositivos principales del sistema junto con la tarjeta STM32 y el Velodyne. Se sitúan en el vehículo orientadas en unos ángulos concretos, tomando como referencia el LiDAR. Estas cámaras realizan una captura del entorno, mejorando y complementando la información generada por la nube de puntos del LiDAR, cuando este se encuentra en el ángulo configurado para cada cámara.

Para activar las cámaras, se necesita disponer de las señales de disparo correspondientes. Con el fin de generar dichas señales, se han empleado los Timers del microcontrolador, y se han utilizado como salidas varios de los pines de la tarjeta STM32.

Además, para realizar la conexión entre las cámaras y sus señales de disparo, se han fabricado unos cables específicos donde se interconectan cada una de ellas con el pin correspondiente de la tarjeta. Para ello se han utilizado los conectores de disparo Hirose, de los que se ha cableado su entrada optoacoplada y su masa para sacarlos en un cable RJ45, que se conecta al RJ45 hembra fijado en la placa. Desde los pines de la tarjeta se han sacado mediante cables las señales de disparo y la masa, y se conectarán en los pines del conector RJ45 hembra. En la Figura 3 se muestra uno de estos cables que ofrece la posibilidad de conectar hasta 3 cámaras.

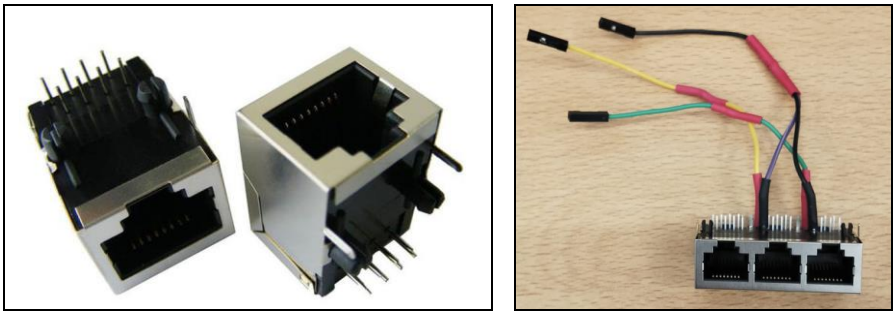


**Figura 3. Base conector hembra RJ45 y cables de conexión fabricados.**

Una vez planteado el funcionamiento global del sistema, dentro de este capítulo se va a realizar una explicación más extensa de cada uno de los 3 dispositivos principales del proyecto, el Velodyne HDL-32E, las cámaras Point Grey Grasshopper3 GS3-U3-23S6C y la placa STM32 Nucleo-144.

## 3.1. Velodyne LiDAR HDL-32E

El sensor LiDAR HDL-32E de Velodyne ha sido diseñado para superar las demandas de las aplicaciones industriales del mundo real como pueden ser el control y la operación autónoma de vehículos, mapeo terrestre móvil, mapeo aéreo en 3D y vigilancia.

### 3.1.1. Características Velodyne LiDAR HDL-32E

El HDL-32E es pequeño y robusto, mide 144 mm por 85 mm, y pesa 1 Kg que aumenta a 1,3 Kg por el cableado. Estas características le hacen ideal para todo tipo de aplicaciones LiDAR, en particular aquellas con factores de forma restringidos, pero que aun así exigen un alto rendimiento. En la Figura 4 que aparece más abajo, se puede ver el LiDAR con sus medidas.

La innovadora matriz multicanal del HDL-32E permite que los sistemas de navegación y cartografía observen más de su entorno que otros sensores LiDAR del mercado. El dispositivo utiliza 32 canales alineados desde 10,67º hasta -30,67º para proporcionar un buen campo de visión vertical, y su diseño de cabezal giratorio ofrece un campo de visión horizontal del 360º en tiempo real. Este LiDAR genera una nube de puntos de hasta 695.000 puntos por segundo con un alcance de hasta 100 m con una precisión de ± 2 cm. Cada canal láser es un par emisor receptor IR (*Infared Radiation*) trabajando a una longitud de onda de 903 nm. Además, cada uno, se fija en un ángulo respecto al plano horizontal del sensor y se le asigna su propio número de identificación.

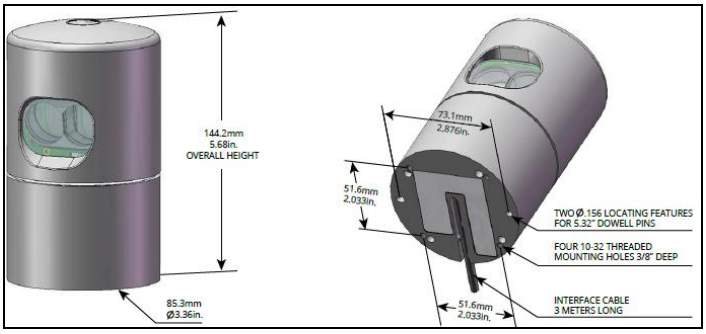



**Figura 4. Dimensiones HDL32-E. [2]**

El sensor HDL-32E utiliza 32 láseres infrarrojos emparejados con detectores IR para crear imágenes 3D de 360º y medir distancias a objetos. El dispositivo está montado de forma segura dentro de una carcasa compacta y resistente a la intemperie que gira rápidamente para escanear el entorno que le rodea, disparando los láseres aproximadamente 18.000 veces por segundo, proporcionando de esta forma, en tiempo real, una amplia nube de puntos 3D. La frecuencia de rotación se puede configurar con un valor o en modo automático.

Los sensores Velodyne LiDAR utilizan la metodología ToF (*Time of flight*). Cuando cada laser IR emite un pulso, se registra su tiempo de disparo y su dirección. El pulso viaja a través del aire hasta que choca con un obstáculo que refleja parte de la energía. Esta energía reflejada es recibida por el detector IR emparejado, que registra el tiempo de adquisición y la potencia recibida.

El procesamiento avanzado de señales digitales y el análisis de formas de onda proporcionan una detección de largo alcance altamente precisa, así como datos de reflectividad calibrados, lo que permite una fácil detección de señales de tráfico, placas de matrícula y marcas viales.

Utiliza un sistema de motor de transmisión directa, sin correas ni cadenas en el tren de transmisión, para mejorar la seguridad y reducir el mantenimiento. La unidad proporciona:

- Un campo de visión horizontal (FOV, *Field Of View*) de 360º.
- Un campo de visión vertical de 41’3º.

El HDL-32E no necesita configuración, calibración u otro tipo de setup para comenzar a producir datos útiles. Una vez se monta y cablea la unidad, al suministrarle energía comenzará a escanear, producir y enviar paquetes de datos a través de Ethernet.

En la Figura 5 que se muestra a continuación se puede ver un resumen de las características del LiDAR HDL-32E.

| | **Specifications:** |
|---|---|
| **Sensor:** | • 32 Channels<br>• Measurement Range: Up to 100 m<br>• Range Accuracy: Up to ±2 cm (Typical)[1]<br>• Single and Dual Returns (Strongest, Last)<br>• Field of View (Vertical): +10.67° to -30.67° (41.33°)<br>• Angular Resolution (Vertical): 1.33°<br>• Field of View (Horizontal): 360°<br>• Angular Resolution (Horizontal/Azimuth): 0.08° - 0.33°<br>• Rotation Rate: 5 Hz – 20 Hz<br>• Integrated Web Server for Easy Monitoring and Configuration |
| **Laser:** | • Laser Product Classification: Class 1 Eye-safe per IEC 60825-1:2007 & 2014<br>• Wavelength: 903 nm |
| **Mechanical/ Electrical/ Operational** | • Power Consumption: 12 W (Typical)[2]<br>• Operating Voltage: 9 V – 18 V (with Interface Box and Regulated Power Supply)<br>• Weight: ~1.0 kg (without Cabling and Interface Box)<br>• Dimensions: See diagram on previous page<br>• Environmental Protection: IP67<br>• Operating Temperature: -10°C to +60°C[3]<br>• Storage Temperature: -40°C to +105°C |

| Output: | • 3D Lidar Data Points Generated:<br>- Single Return Mode: ~695,000 points per second<br>- Dual Return Mode: ~1,390,000 points per second<br>• 100 Mbps Ethernet Connection<br>• UDP Packets Contain:<br>- Time of Flight Distance Measurement<br>- Calibrated Reflectivity Measurement<br>- Rotation Angles<br>- Synchronized Time Stamps (µs resolution)<br>• Orientation: 6DoF Inertial Sensor Measurements<br>• GPS: $GPRMC and $GPGGA NMEA Sentences from GPS Receiver (GPS not included) |
|---|---|

**Figura 5. Resumen de características del HDL-32E. [2]**

### 3.1.2. Formato del paquete de datos

El HDL-32E emite dos paquetes de datos UDP a través de Ethernet: el primer paquete contiene los datos del disparo láser y se envía por el puerto 2368, el segundo paquete contiene los datos de posicionamiento GPS (*Global Positioning System*), y en este caso, se envía por el puerto 8308. El primer paquete está formado por: una cabecera, datos de disparo y datos de estado.

Este paquete se forma con la recopilación de todos los datos obtenidos para 12 secuencias de disparo láser. Los datos de distancia e intensidad del láser se recopilan en el mismo orden escalonado en el que se disparan los láseres, transmitiendo primero el byte menos significativo. A continuación, estos datos se combinan con los datos de estado y la cabecera formando un paquete UDP que es transmitido a través de Ethernet.

Cada paquete de datos UDP que se envía por el puerto 2368 tiene un tamaño de 1.248 bytes, que se divide en: 42 bytes de cabecera, 1.200 bytes con los datos de disparo distribuidos en 12 registros de 100 bytes, y por último 6 bytes con los datos de estado, de los cuales 4 bytes son para la marca de tiempo GPS, que representa los milisegundos desde la hora de inicio, y los otros 2 bytes están reservados por el fabricante. Cada registro de 100 bytes contiene primero un identificador de inicio de 2 bytes, luego un valor de rotación de 2 bytes, seguido de 32 combinaciones de 3 bytes que informan sobre cada láser disparado. De estos 3 bytes, 2 informan de la distancia, y el restante informa de la intensidad en una escala de 0 a 255.

En las Figuras 6 y 7 extraídas de la hoja de características del dispositivo se pueden observar dos esquemas que resumen cómo está estructurado el paquete UDP.

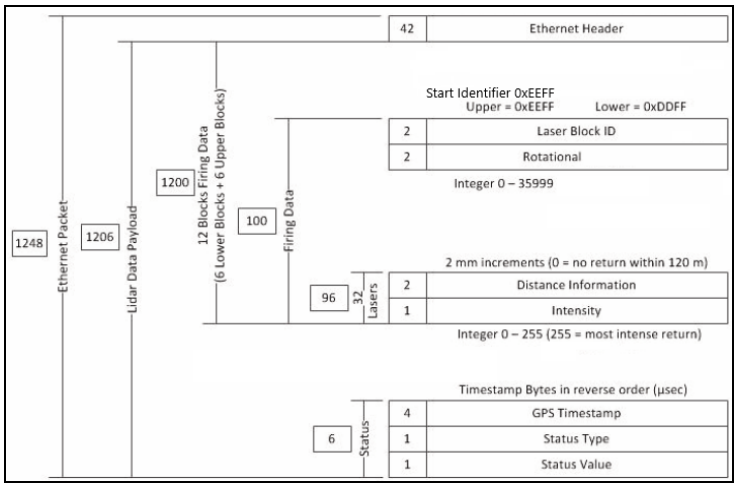



**Figura 6. Estructura del paquete UDP para el puerto 2368. [2]**

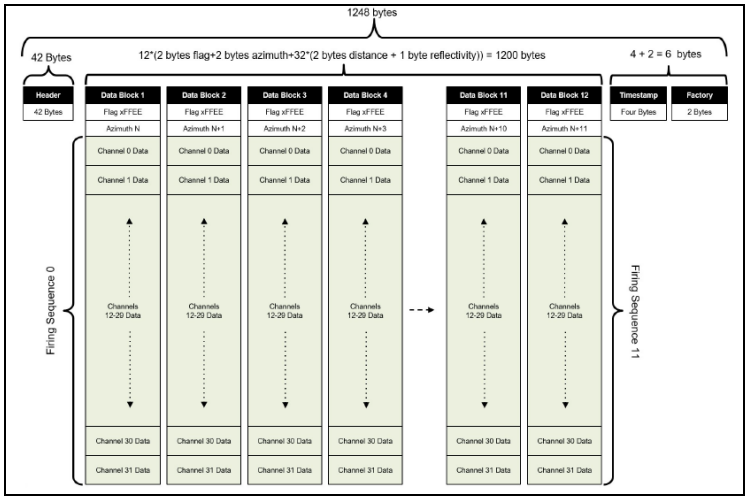



**Figura 7. Esquema del paquete UDP para el puerto 2368. [2]**

En este proyecto el paquete de interés es el que contiene los datos del disparo láser, el que se transmite por el puerto 2368. Estos paquetes UDP son los que la tarjeta STM32 tiene que recibir y gestionar para, una vez extraídos los datos, realizar las acciones correspondientes con la información que poseen.

### 3.1.3. Secuencia de disparo de los láseres

El orden de disparo de los láseres es el mismo que en el que se almacenan los datos en el paquete UDP. Primero dispara el láser que se encuentra en la posición inferior, seguido de disparos intercalados de los bancos superior e inferior de láseres (0, 16, 1, 17 … 14, 30, 15, 31). Como el número de identificación de cada láser no cambia y su posición en el paquete de datos tampoco, se puede deducir el ángulo de elevación de cada láser por su ubicación en el paquete UDP.

El patrón de disparo intercalado está diseñado para evitar posibles imágenes fantasma causadas principalmente por retrorreflectores. En la Figura 8 se puede ver el orden de la secuencia de disparo.

| Firing order | DSR # | Vertical angle |
|---|---|---|
| 1 | 0 | -30.67 |
| 2 | 1 | -9.33 |
| 3 | 2 | -29.33 |
| 4 | 3 | -8.00 |
| 5 | 4 | -28.00 |
| 6 | 5 | -6.66 |
| 7 | 6 | -26.66 |
| 8 | 7 | -5.33 |
| 9 | 8 | -25.33 |
| 10 | 9 | -4.00 |
| 11 | 10 | -24.00 |
| 12 | 11 | -2.67 |
| 13 | 12 | -22.67 |
| 14 | 13 | -1.33 |
| 15 | 14 | -21.33 |
| 16 | 15 | 0.00 |
| 17 | 16 | -20.00 |
| 18 | 17 | 1.33 |
| 19 | 18 | -18.67 |
| 20 | 19 | 2.67 |
| 21 | 20 | -17.33 |
| 22 | 21 | 4.00 |
| 23 | 22 | -16.00 |
| 24 | 23 | 5.33 |
| 25 | 24 | -14.67 |
| 26 | 25 | 6.67 |
| 27 | 26 | -13.33 |
| 28 | 27 | 8.00 |
| 29 | 28 | -12.00 |
| 30 | 29 | 9.33 |
| 31 | 30 | -10.67 |
| 32 | 31 | 10.67 |

**Figura 8. Secuencia de disparo de los láseres. [2]**

### 3.1.4. Tiempo de disparo

Los paquetes UDP se ensamblan en tiempo real, de forma que, el primer disparo láser de cada secuencia de 32 disparos marca el ángulo que se le asocia (un ángulo para cada uno de los 12 bloques del paquete). Por otro lado, la marca de tiempo que aparece en los 6 bytes finales del paquete UDP se relaciona con el ultimo disparo del último de los 12 bloques de datos (una marca de tiempo por paquete).

Los láseres se disparan cuando marca un reloj que funciona con un tiempo de ciclo de 1,152 µseg. Para cada secuencia de 32 disparos hay 40 de estos ciclos, lo que hace que el tiempo total para disparar cada secuencia sea de 46,08 µseg. El tiempo restante, 8 ciclos, se utiliza para recargar los láseres. Como en cada paquete hay 12 bloques de estas secuencias de 32 disparos, el tiempo total por paquete es de 552,96 µseg. En un segundo hay aproximadamente unos 1.808 paquetes, o alrededor de 694,292 disparos por segundo.

Cuando el LiDAR se enciende, comienza a contar microsegundos utilizando una referencia de tiempo interna, desde la media noche del 1 de enero del año 2000. Sin embargo, se pueden sincronizar sus datos con la hora UTC (*Tiempo Universal Coordinado*) y así determinar la hora exacta de disparo de cada laser en cualquier paquete. Para poder sincronizar los datos con la hora UTC se necesita utilizar el GPS proporcionado por Velodyne o uno diferente que habrá que configurar.

La marca de tiempo GPS es útil para determinar el tiempo exacto de disparo de cada láser y de cada punto de la nube generada y así alinear estos datos con los de otras fuentes.

## 3.2. Cámaras Point Grey Grasshopper3 GS3-U3-23S6C

La línea de cámaras Grasshopper3 de FLIR aporta un alto rendimiento y una imagen de alta calidad combinando la tecnología más novedosa de CCD (*Charge Coupled Device*) y CMOS (*Complementary Metal Oxide Semiconductor*), con la asequibilidad y el rendimiento de datos de USB (*Universal Serial Bus*) 3.0. Su arquitectura está basada en FPGA (*Field Programmable Gate Array*) y búfer de fotogramas proporcionando fiabilidad, un rico conjunto de características y un *pipeline* de procesamiento de imágenes completo. Estas cámaras ofrecen una alternativa potente, fácil de usar y rentable a soluciones Camera Link y dual GigE LAG.

Las características clave de estas cámaras son:

- USB 3.0 para ancho de banda, facilidad de uso y rentabilidad.
- Variedad de sensores CCD y CMOS de alta resolución.
- Arquitectura basada en búfer de fotogramas y FPGA para una fiabilidad óptima.

El resto de características para el modelo GS3-U3-23S6C aparecen resumidas en la Figura 9 que se muestra a continuación.

| GS3-U3-23S6C-C | |
|---|---|
| Firmware | 2.22.3.0 |
| Resolution | 1920 x 1200 |
| Frame Rate | 163 FPS |
| Megapixels | 2.3 MP |
| Chroma | Color |
| Sensor | Sony IMX174, CMOS, 1/1.2" |
| Readout Method | Global shutter |
| Pixel Size | 5.86 µm |
| Lens Mount | C-mount |
| ADC | 10-bit / 12-bit in Mode 7 |
| Gain Range | 0 dB to 29 dB |
| Exposure Range | 0.005 ms to 31.9 seconds |
| Trigger Modes | Standard, bulb, overlapped*, multi-shot (*available with firmware 2.9.3.0 and later) |
| Partial Image Modes | Pixel binning, ROI |
| Image Processing | Gamma, lookup table, hue, saturation, and sharpness |
| HDR Sequencing | 4 x exposure, 4 x gain |
| Image Buffer | 128 MB |

| | |
|---|---|
| User Sets | 2 memory channels for custom camera settings |
| Flash Memory | 2 MB non-volatile memory |
| Opto-isolated I/O Ports | 1 input, 1 output |
| Non-isolated I/O | 2 bi-directional |
| Serial Port | 1 (over non-isolated I/O) |
| Auxiliary Output | 3.3 V, 150 mA maximum |
| Interface | USB 3.1 |
| Power Requirements | 5 V via USB 3.1 or 8-24 V via GPIO (external power is recommended for this model) |
| Power Consumption | 4.5 W maximum |
| Dimensions/Mass | 44 mm x 29 mm x 58 mm/90 g |
| Machine Vision Standard | USB3 Vision v1.0 |
| Compliance | CE, FCC, KCC, RoHS. The ECCN for this product is: EAR099. |
| Temperature | Operating: 0° to 50°C<br>Storage: -30° to 60°C |
| Humidity | Operating: 20% to 80% (no condensation)<br>Storage: 20% to 95% (no condensation) |
| Warranty | 3 years |

**Figura 9. Resumen de características cámara Grasshopper3 GS3-U3-23S6C. [3]**

En este proyecto se parte con las cámaras ya preparadas para su uso, por lo que se va a comentar cómo están realizadas las conexiones y en caso de necesitar más información sobre la instalación y configuración de las cámaras se puede consultar el manual de usuario de las mismas. Además de las cámaras, se utilizan como conectores de vídeo y alimentación cables USB 3.0 tipo A-micro B, y como conectores de disparo los conectores circulares GPIO Hirose HR25 que se pueden observar junto con las cámaras en la Figura 10.

Sobre las cámaras se necesita conocer que señal utilizan para generar un disparo. Esta puede ser un pulso a nivel alto o a nivel bajo, dependiendo de la configuración de las cámaras (en este caso a nivel bajo), de 50 ms. La tarjeta STM32 es la encargada de generar por varios de sus pines, uno por cada cámara, la señal anteriormente mencionada.


**Figura 10. Cámaras Point Grey Grasshopper3 GS3-U3-23S6C y conector Hirose HR25. [4]**

## 3.3. Placa STM32 Nucleo-144

La placa STM32 Nucleo-144 proporciona una forma asequible y flexible con la que los usuarios pueden probar nuevos conceptos y crear prototipos eligiendo entre las diversas combinaciones de rendimiento, consumo de energía y funciones que ofrece el microcontrolador STM32.

El conector ST Zio, que amplía la conectividad con Arduino Uno V3, y los conectores morpho ST proporcionan un medio sencillo para expandir la funcionalidad de la plataforma de desarrollo abierta Nucleo con una gran variedad de protectores especializados.

Esta placa no requiere de ninguna sonda externa para las tareas de programación y depuración, ya que lleva integrado el depurador/programador ST_LINK/V2-1. Además, incluye las extensas bibliotecas de software gratuitas HAL y los ejemplos disponibles con el paquete STM32Cube MCU, así como un acceso directo a los recursos en línea de Arm Mbed Enabled.

### 3.3.1. Características placa STM32 Nucleo-144

Las características principales de esta placa son las siguientes:

- Microcontrolador con encapsulado LQFP144.
- 3 LEDs (*Light Emitting Diode)* utilizables por el usuario.
- 2 pulsadores, el de reset y otro de libre uso para el usuario.
- Un cristal oscilador de 32.768 KHz.
- Conectores de la tarjeta: SWD, ST Zio, ST morpho, USB Micro A-B y Ethernet RJ45.
- Varias opciones flexibles de alimentación: ST-LINK, USB o fuentes externas.
- Soporta una amplia variedad de IDEs (*Integrated Development Environment*), incluidos IAR, Keil y IDE basados en GCC.
- Bibliotecas HAL y ejemplos de software gratuitos con el paquete STM32Cube MCU.
- ST-LINK programador/depurador integrado con capacidad de reenumeración USB: almacenamiento masivo, puerto COM virtual y puerto de depuración.
- Ethernet compatible con IEE-802.3-2002.
- USB OTG FS/HS.

A continuación, se muestra la Figura 11 en la que se puede ver la placa vista desde arriba y desde abajo.

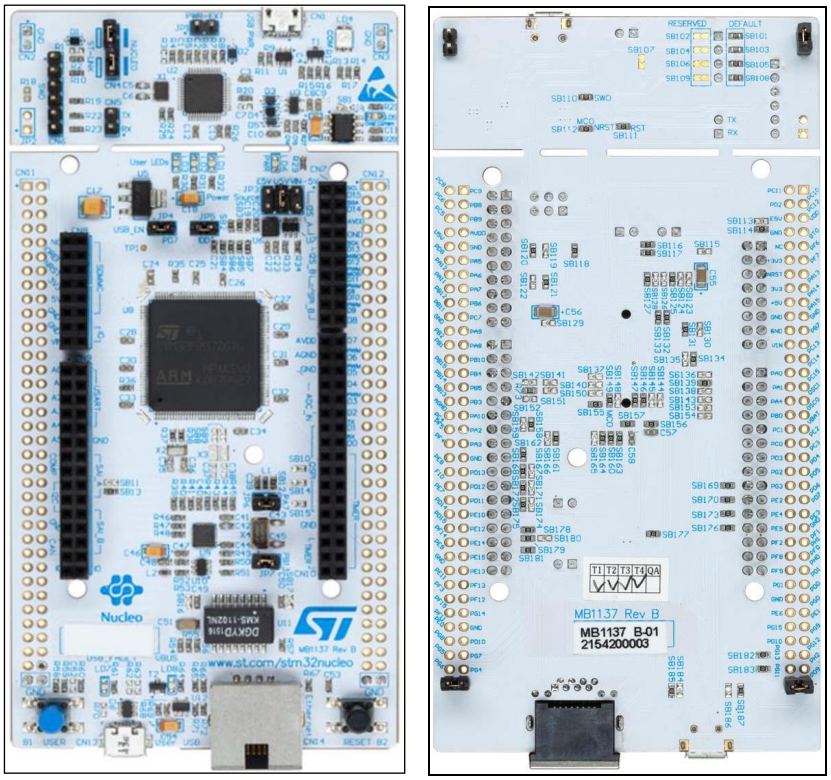



**Figura 11. Parte superior e inferior de la tarjeta STM32 Nucleo-144. [1]**

La ubicación de cada elemento hardware de la placa se muestra en la Figura 12. En ella se ven situados, en su cara superior e inferior, los diferentes elementos de los que dispone, como son los conectores (Ethernet y USB entre otros), LEDs, pulsadores, pines, etc.

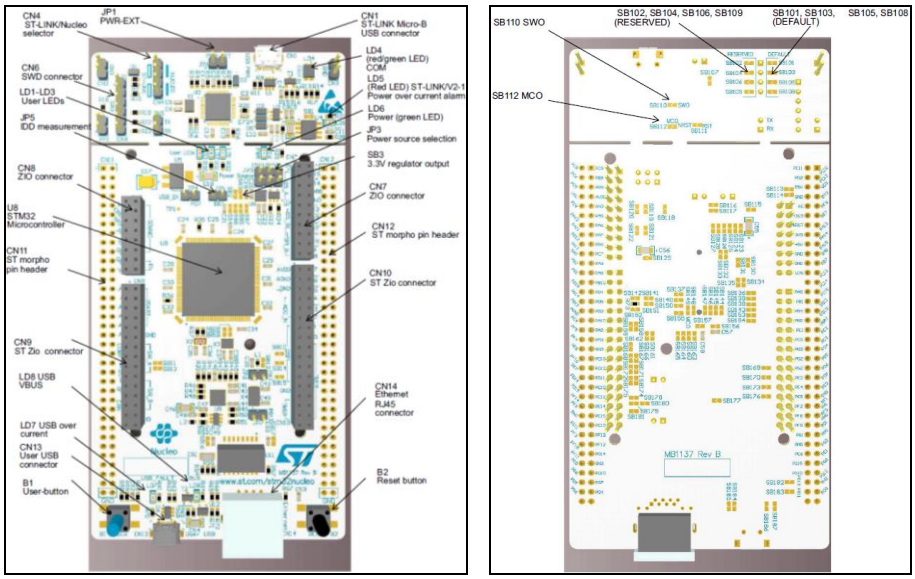



**Figura 12. Disposición hardware superior e inferior de la tarjeta STM32 Nucleo-144. [1]**

### 3.3.2. Microcontrolador

El microcontrolador STM32F767ZI se basa en el núcleo Arm Cortex-M7 RISC (*Reduced Instruction Set Computer*) de 32 bits de alto rendimiento que funciona a una frecuencia de hasta 216 MHz. El núcleo Cortex-M7 cuenta con una unidad de punto flotante (FPU, *Floating Point Unit*) que admite instrucciones de procesamiento y tipos de datos de precisión single (32 bits) y double (64 bits). También implementa un conjunto completo de instrucciones DSP (*Digital Signal Processor*) y una MPU (*unidad de protección de memoria*) que mejora la seguridad de la aplicación.

El dispositivo STM32F767ZI incorpora una memoria Flash integrada de alta velocidad de 2 Mbytes y una memoria SRAM de 512 Kbytes, incluidos 128 Kbytes de datos TCM RAM (para datos críticos en tiempo real), 16 Kbytes más de instrucción TCM (*Tightly-Coupled Memory*) RAM (para rutinas críticas en tiempo real) y 4 Kbytes de backup SRAM en los modos de energía más bajos. También cuenta con un flexible controlador de memoria externa con un bus de datos de hasta 32 bits para memorias: SRAM, PSRAM, SDRAM/LPSDR SDRAM, NOR/NAND.

Además, presenta una amplia gama de E/S mejoradas y periféricos conectados a: dos buses APB (*Advanced Peripheral Bus*), dos buses AHB (*Advanced High-perfomance Bus*), un bus matriz multi-AHB de 32 bits, y una interconexión AXI multicapa que admite el acceso a memorias internas y externas.

El microcontrolador incluye tres ADC (*Analog to Digital Converter*) de 12 bits, dos DAC (*Digital to Analog Converter*), un RTC (*Real Time Clock*) de baja potencia, doce Timers de 16 bits de uso general, dos Timers de 32 bits de uso general y un *true* RNG (*Random Number Generator*) de 32 bits. Cuenta con las siguientes interfaces de comunicación estándar y avanzadas:

- Cuatro I2C (*Inter-Integrated Circuit*).
- Seis SPI (*Serial Peripheral Interface*), tres I2S (*Integrated Interchip Sound*) en modo semidúplex. Para lograr la precisión de la clase de audio, los periféricos I2S se pueden sincronizar a través de un PLL (*Phase Locked Loop*) de audio interno para permitir la sincronización.
- Cuatro USART (*Universal Synchronous/Asynchronous Receiver Transmitter*) y cuatro UART (*Universal Asynchronous Receiver-Transmitter*).
- Un USB OTG de alta velocidad y otro con capacidad de velocidad completa.
- Tres CAN (*Controller Area Network*).
- Dos SAI (*Interfaces de Audio en Serie*).
- Dos interfaces SDMMC (*Secure Digital and MultiMediaCard*).
- Interfaces cámara y Ethernet.
- Controlador de pantalla TFT-LCD (*Thin Film Transistor-Liquid Crystal Display*).
- Acelerador Chrom-ART.
- Interfaz SPDIFRX.
- HDMI-CEC (*High Definition Multimedia Interface-Consumer Electronics Control*).

En cuanto a los periféricos avanzados incluye 2 interfaces SDMMC, una interfaz FMC (*Control de Memoria Flexible*), una interfaz de memoria Flash Quad-SPI y una interfaz de cámara para sensores CMOS.

El dispositivo funciona en el rango de temperatura de -40 a +105 ºC con una fuente de alimentación de 1,7 a 3,6 V. Un conjunto completo de modos de ahorro de energía permite el diseño de aplicaciones de bajo consumo.

Estas características hacen que el microcontrolador STM32F767ZI sea adecuado para una amplia gama de aplicaciones:

- Control de aplicaciones.
- Equipo médico.
- Aplicaciones industriales: PLC (*Programmable Logic Controller*), inversores, disyuntores.
- Impresoras y escáneres.
- Sistemas de alarma, video y HVAC (*Heating Ventilating and Air Conditioning*).
- Aparatos de audio.
- Aplicaciones móviles, IoT (*Internet of Things*).
- Dispositivos portátiles.

Para realizar la señal de disparo de las cámaras se van a utilizar, como herramienta principal, los Timers del microcontrolador, de forma que se va a realizar una breve explicación sobre ellos comentando sus características y usos.

#### 3.3.2.1. Timers

El microcontrolador dispone de 14 Timers divididos clasificados en varios grupos:

- Timers de control avanzado TIM1 y TIM8.
- Timers de propósito general: TIM2, TIM3, TIM4, TIM5, TIM9, TIM10, TIM11, TIM12, TIM13 y TIM14.
- Timers básicos TIM6 y TIM7.

Los Timers pueden emplearse para una gran variedad de finalidades: medir el pulso de señales de entrada (input capture) o generar señales (output compare, PWM (*Pulse Width Modulation*),

PWM con tiempo muerto añadido, y el modo one pulse output). El ancho de los pulsos y el periodo de las señales se puede modular desde unos pocos microsegundos hasta varios milisegundos dependiendo de la configuración escogida.

Según el tipo de Timer, sus características varían, lo que hace que algunos de ellos no puedan realizar todas las tareas anteriormente mencionadas, quedando su uso limitado a unas pocas. Las características principales que diferencian unos Timers de otros quedan resumidas en la Figura 13.

| Timer type | Timer | Counter resolution | Counter type | Prescaler factor | DMA request generation | Capture/ compare channels | Complementary output | Max interface clock (MHz) | Max timer clock (MHz)(1) |
|---|---|---|---|---|---|---|---|---|---|
| Advanced -control | TIM1, TIM8 | 16-bit | Up, Down, Up/down | Any integer between 1 and 65536 | Yes | 4 | Yes | 108 | 216 |
| General purpose | TIM2, TIM5 | 32-bit | Up, Down, Up/down | Any integer between 1 and 65536 | Yes | 4 | No | 54 | 108/216 |
| | TIM3, TIM4 | 16-bit | Up, Down, Up/down | Any integer between 1 and 65536 | Yes | 4 | No | 54 | 108/216 |
| | TIM9 | 16-bit | Up | Any integer between 1 and 65536 | No | 2 | No | 108 | 216 |
| | TIM10, TIM11 | 16-bit | Up | Any integer between 1 and 65536 | No | 1 | No | 108 | 216 |
| | TIM12 | 16-bit | Up | Any integer between 1 and 65536 | No | 2 | No | 54 | 108/216 |
| | TIM13, TIM14 | 16-bit | Up | Any integer between 1 and 65536 | No | 1 | No | 54 | 108/216 |
| Basic | TIM6, TIM7 | 16-bit | Up | Any integer between 1 and 65536 | Yes | 0 | No | 54 | 108/216 |

**Figura 13. Resumen de características de los diferentes Timers. [1]**

# 4. Desarrollo del proyecto

Una vez se conocen los dispositivos con los que se va a contar, se van a describir los objetivos prácticos y los pasos a seguir para llevarlos a cabo, así como los problemas y las pruebas que se han realizado durante este proceso. Hay que mencionar que no es un trabajo orientado a la enseñanza, pero si se podrá utilizar como apoyo en temas concretos, ya que se cuentan los problemas que se han tenido con los diferentes dispositivos al realizarlo. Hay dos bloques principales en los que se divide el trabajo:

- Generación de las señales de disparo para cada una de las cámaras FLIR por los pines de la tarjeta STM32.
- Recepción y lectura con la tarjeta STM32 de los paquetes UDP que manda el LiDAR a través de Ethernet.

Anteriormente se ha hecho una breve introducción al proyecto, pero se puede decir que este bloque es el corazón del TFG, ya que engloba la mayor parte. Tras las primeras reuniones con mi tutor y cotutor se decidió que el primer objetivo que debía realizar era la puesta en marcha de la tarjeta STM32, y adquirir los conocimientos necesarios para manejarla y generar las señales de disparo de las cámaras FLIR, por lo que ese fue el punto de partida.

## 4.1. Generación de la señal de disparo

En primer lugar, realicé un estudio de la tarjeta viendo su estructura y los periféricos que me pudieran servir en este apartado, como son los Timers, los pines y los LEDs.

Como se ha mencionado anteriormente en las características de la tarjeta, esta cuenta con 14 Timers, aunque hay algunas diferencias entre ellos, 3 LEDs para el usuario y una gran cantidad de pines disponibles. También tiene varias librerías HAL para facilitar el uso de los periféricos.

Como herramientas para iniciar la tarjeta STM32, generar el código, depurarlo y cargarlo en la placa se me propusieron Keil uVision y STM32CubeMX. Dado que el compilador Keil uVision ya lo conocía decidí comenzar indagando sobre el otro programa, con el fin de que pudiera facilitarme el trabajo.

STM32CubeMX se trata de una herramienta de generación de código a partir de las especificaciones de las placas de la familia STM32. Es una interfaz sencilla, fácil de utilizar, que permite configurar de forma gráfica cualquier periférico del núcleo que se está programando. Este programa genera el código, pero necesita la ayuda de un entorno de desarrollo integrado, para lo cual nos da varias opciones, entre las que se encuentra MDK-ARM basado en uVision.

En este punto entendí que la utilización de las dos herramientas es compatible, y su uso conjunto facilita mucho el trabajo. Empleando únicamente Keil uVision, iniciar la tarjeta y generar un código desde 0 para esta placa, sin la ayuda de STM32CubeMX, se iba a convertir en un proceso mucho más largo y tedioso, por lo que se decidió usar ambos. Para aprender a usar este programa se ha utilizado el manual de usuario STM32CubeMX para STM32, y varios videos de la plataforma Youtube a modo de tutorial.

Para poner en marcha la tarjeta se ha decidido realizar un primer proyecto simple, en el que un LED de la placa alterne su estado entre encendido y apagado. Con este primer ejemplo se ha querido comprobar que tanto los dos programas como la tarjeta están listos para usarse, y paso a paso, conocer cómo se configuran los diferentes periféricos y elementos de la tarjeta para su posterior uso en el TFG.

Nada más iniciar STM32CubeMX se ha de elegir que microcontrolador o que tarjeta se va a utilizar, en este caso la placa NUCLEO-F767ZI, como se puede ver en la Figura 14. También se da la opción de iniciar los periféricos por defecto como así se ha seleccionado.

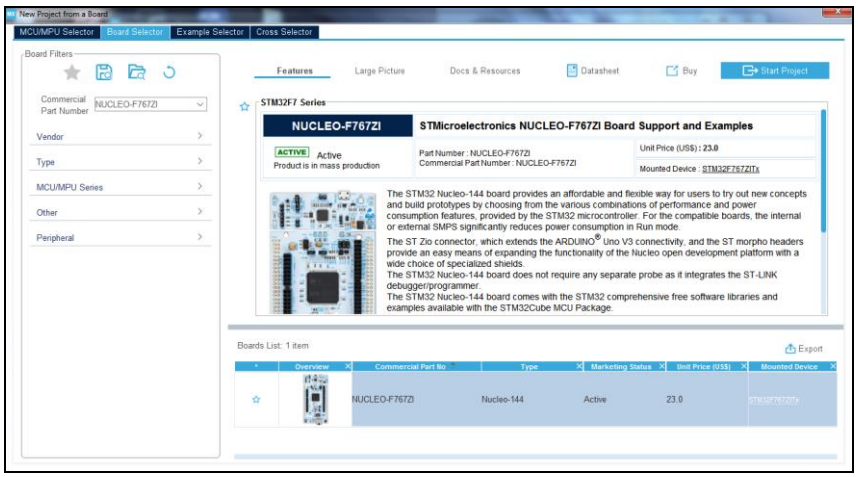



**Figura 14. Selección de tarjeta en STM32CubeMX.**

Una vez iniciado el proyecto aparece en pantalla el microcontrolador gráficamente con todos sus pines. Es una interfaz bastante intuitiva, de forma que se puede configurar cada uno de los pines simplemente con seleccionar uno de ellos y escoger entre las diferentes opciones disponibles: GPIO_Input, GPIO_Output, UART, ADC, etc.

También es fácil visualizar que configuración tienen, ya que aparece la descripción al lado de cada pin y se marcan con diferentes colores dependiendo de su función. Además de los pines se pueden configurar los periféricos y el resto de parámetros de la tarjeta. En la Figura 15 se puede observar cómo se ve el microcontrolador con los periféricos iniciados por defecto.

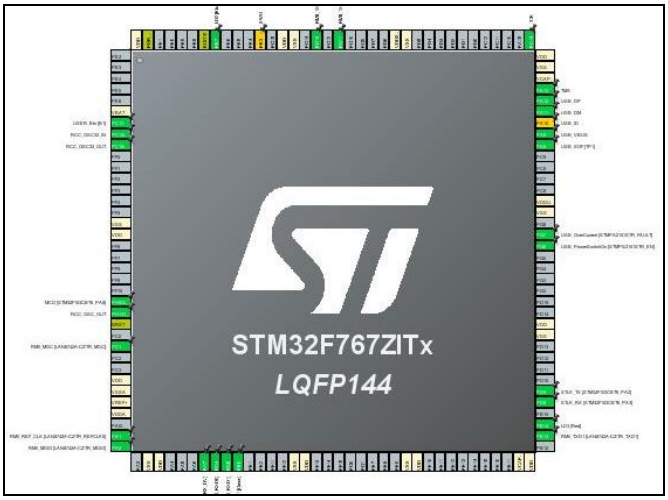



**Figura 15. Microcontrolador inicializado por defecto en STM32CubeMX.**

Para este primer proyecto se parte de la configuración por defecto, en la que los 3 LEDs de la placa están configurados como GPIO_Output (PB0-Verde, PB7-Azul y PB14-Rojo) por lo que no necesitan ser modificados. A continuación, se ha configurado el reloj del sistema con los valores que se pueden ver en la Figura 16. En este caso, como no se utilizan Timers, ADC u otros periféricos para los que el reloj sea importante, se han seleccionado unos valores de reloj estándar.

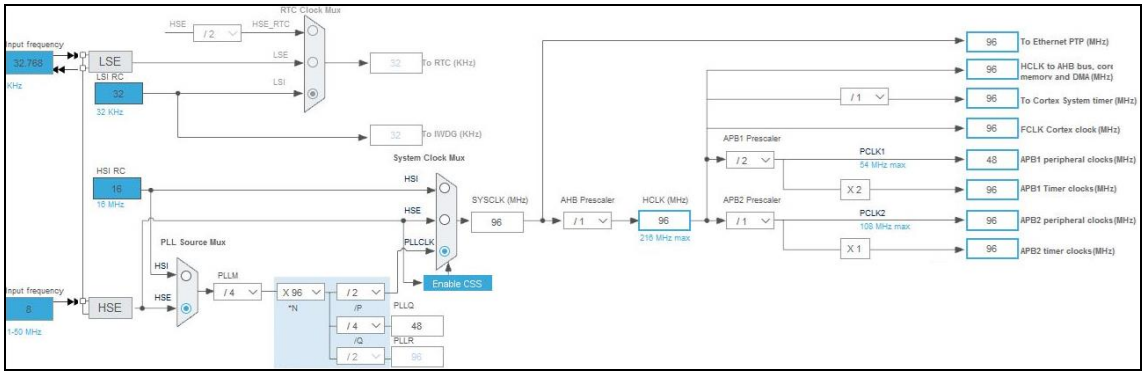



**Figura 16. Configuración del reloj del sistema para el primer proyecto.**

Posteriormente, queda seleccionar el IDE y la versión que se va a utilizar. En mi PC disponía de Keil uVision en versión 4 y 5.27 de forma que podía escoger entre las dos, pero finalmente me decanté por el IDE MDK-ARM en su versión 5.27. En la Figura 17 que se muestra a continuación aparece esta selección.

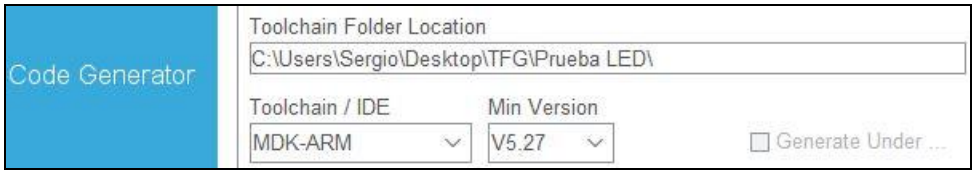



**Figura 17. Selección IDE y su versión.**

Para terminar con STM32CubeMX, se ha marcado que se añadan al proyecto las librerías necesarias, entre ellas las librerías HAL. Una vez se ha finalizado la configuración, la herramienta STM32CubeMX genera el código C que se abre con el IDE seleccionado. El código que se ha generado inicializa la tarjeta y todos los periféricos teniendo en cuenta la configuración previamente establecida.

En la Figura 18 aparecen las librerías HAL que se han añadido al proyecto. Cada uno de los archivos contiene funciones para el periférico con el que están relacionadas. Estas funciones son de gran utilidad y facilitan mucho el trabajo.

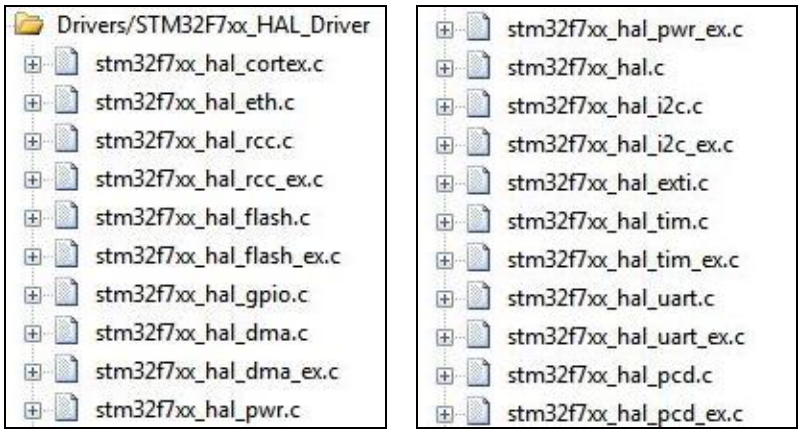



**Figura 18. Librerías HAL.**

Para este primer proyecto son de interés las funciones que se encuentran en el archivo stm32f7xx_hal_gpio.c, ya que se necesitan para modificar el estado del pin PB7 que se corresponde con el LED azul. En este archivo existe la función HAL_GPIO_TogglePin () que permite cambiar el estado de un pin cada vez que se utilice. Dado que el LED azul es el pin 7 del puerto B, y está definido como LD2_PIN, la función queda de la siguiente forma: HAL_GPIO_TogglePin (GPIOB, LD2_PIN).

El bucle while del proyecto se observa en la Figura 19. Se ha añadido la función HAL_Delay () con 200 ms para que se pueda observar el cambio de estado en la luz del LED.

```
102   /* Infinite loop */
103   /* USER CODE BEGIN WHILE */
104   while (1)
105   {
106     /* USER CODE END WHILE */
107     HAL_GPIO_TogglePin(GPIOB, LD2_Pin);
108     HAL_Delay(200);
109     /* USER CODE BEGIN 3 */
110   }
111   /* USER CODE END 3 */
```

**Figura 19. Bucle while del primer proyecto creado.**

La tarjeta STM32 lleva incorporado el depurador/programador ST_LINK/V2-1, de forma que para cargar el programa en la placa únicamente se ha de conectar un cable USB desde la tarjeta hasta el PC, y a través del Keil uVision se selecciona la opción de descarga del código en la tarjeta.

Después de probar con éxito este primer proyecto se han realizado varios programas más similares a este, en los que se prueban pines sin conexión a los LEDs de la tarjeta, el pulsador y su interrupción, otras configuraciones del reloj y por ultimo los Timers y sus diferentes modos.

Inicialmente se tenían que crear 2 señales de disparo debido a que solo se disponía de dos cámaras, pero tras reuniones con mi tutor y cotutor se decidió ampliar hasta 6 el número de señales de disparo por si en un futuro se necesitaban.

La señal que se ha de producir es un pulso de 50 ms a nivel bajo. Las primeras pruebas se han realizado desde casa donde no se disponía de un osciloscopio, dificultando así realizar medidas de la señal. Para ello, se han conectado varios LEDs con sus correspondientes resistencias a los pines de la tarjeta donde la señal es generada, de forma que se puede visualizar cómo se encienden y se apagan cuando se produce el disparo. Aún con los LEDs, medir el pulso de 50 ms sin herramientas es complicado, por ello en lugar de configurarlo con 50 ms se ha hecho con 5 s, para así poder medir el tiempo aproximado de manera visual.

Utilizando de nuevo STM32CubeMX, lo primero que hay que configurar es el reloj del sistema, ya que es quien gestiona la frecuencia a la que funcionan los Timers. En la Figura 20 se puede ver que una vez configurados los prescaler, multiplicadores y divisores, los Timers funcionan a 100 y 200 MHz dependiendo de a que bus APB pertenezcan, siendo 100 MHz los del bus APB1 y 200 MHz los del APB2. Los Timers 2, 3, 4, 5, 6, 7, 12, 13 y 14 pertenecen al bus APB1 y los Timers 1, 8, 9, 10 y 11 al bus APB2 [1].

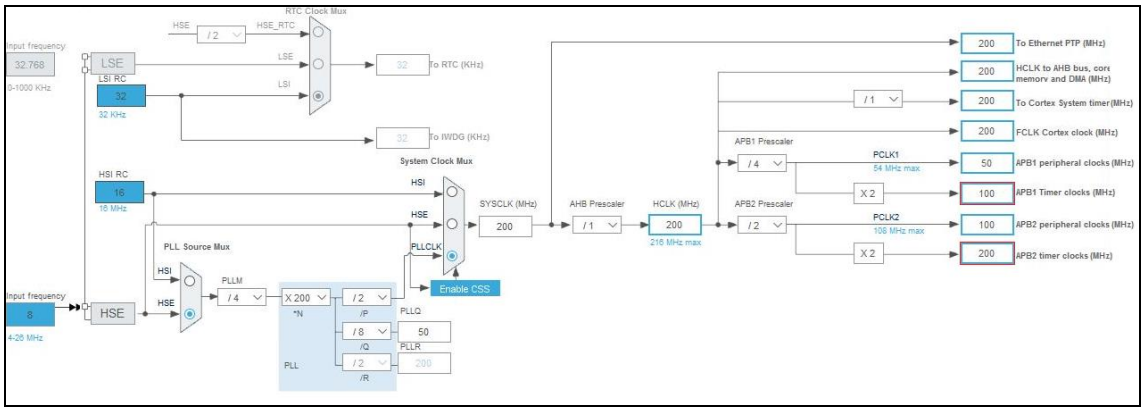



**Figura 20. Configuración del reloj del sistema y frecuencia de los Timers.**

Para generar la señal los Timers brindan varias opciones. Una de ellas es utilizar los canales que tiene asociados cada uno, entre 1 y 4, y configurarlos como PWM de salida. En la Figura 21 se puede ver la configuración de los cuatro canales del Timer como PWM de salida. Además, el pulso de cada PWM se puede configurar de forma independiente, solo comparten el periodo que es común para todos los canales de un mismo Timer. También en esta Figura se puede apreciar que se ha seleccionado como fuente de reloj el reloj interno de la tarjeta.

De igual forma que el Timer 3 se ha realizado la misma configuración para el Timer 9, aunque en este caso solo para 2 canales ya que no dispone de más. Se ha de tener en cuenta para la configuración de periodo y pulso que los Timers pertenecen a buses APB diferentes.

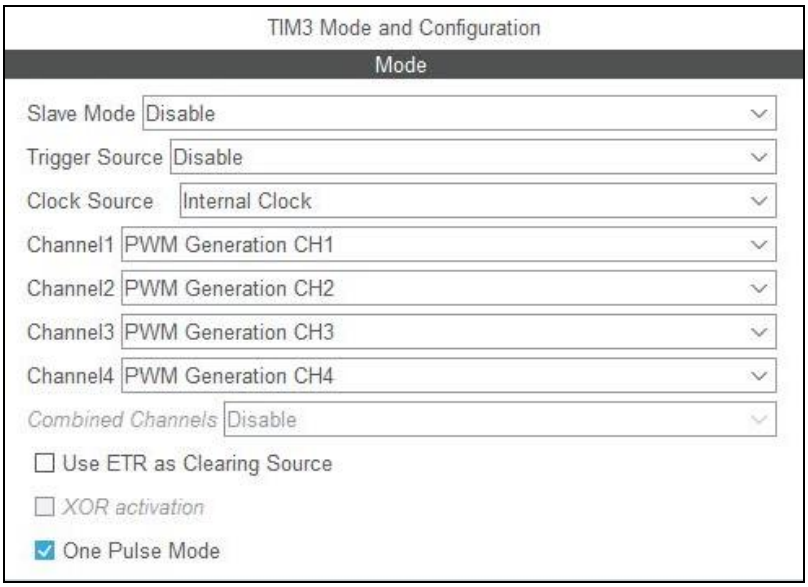



**Figura 21. Configuración Timer 3 y sus canales.**

Automáticamente al realizar esta configuración en los Timers, se han activado los pines asociados a cada canal en el microcontrolador, visualizándose de forma gráfica en la herramienta STM32CubeMX.

A continuación, se ha llevado a cabo la configuración del periodo y las señales PWM. El periodo, común para todos los canales de un mismo Timer, se ha establecido en 5s.

Para el Timer 3, que funciona a una frecuencia de 100MHz, se han realizado los siguientes cálculos en relación al periodo:

$$F_{tick-T3} = \frac{F_{Timer\ 3}}{Prescaler} = \frac{100MHz}{10.000} = 1MHz$$

$$Periodo_{T3}(s) \equiv T = \frac{Número\ de\ cuentas}{F_{tick-T3}} = \frac{50.000}{10.000MHz} = 5s$$

En cambio para el Timer 2:

$$F_{tick-T9} = \frac{F_{Timer\ 9}}{Prescaler} = \frac{200MHz}{20.000} = 1MHz$$

$$Periodo_{T9}(s) \equiv T = \frac{Número\ de\ cuentas}{F_{tick-T9}} = \frac{50.000}{10.000MHz} = 5s$$

Una vez definido el periodo en 5s, se ha de configurar la PWM. Los parámetros que se tienen que configurar son: Mode, Pulse y CH Polarity.

En cuanto al modo hay dos opciones, PWM mode 1 y PWM mode 2. En el modo 1, el canal permanece activo siempre que la cuenta del Timer sea menor que el valor de la variable Pulse.

Se considera activo con un nivel alto si la polaridad es High, y a nivel bajo si es Low. En el modo 2, el canal se encuentra inactivo siempre que la cuenta del Timer sea menor que el valor de la variable Pulse.

La variable Pulse es de 16 bits y define el ancho del pulso de la PWM. Su valor es el que se compara con la cuenta del Timer, y así se determina cuando la señal se ha de activar o no. Por último, CH Polarity indica si el canal activado se corresponde con un nivel alto o bajo dependiendo de si se selecciona High o Low respectivamente.

Vistos estos parámetros, en la Figura 22 se aprecia la configuración seleccionada. Para la señal de disparo se necesita un pulso a nivel bajo, por lo que se ha seleccionado PWM mode 1. Con este modo, el canal permanece activo desde que se inicia la cuenta del Timer hasta que esta llega al valor de Pulse, también se volverá a activar cuando la cuenta llegue al valor del periodo y finalice. El canal únicamente permanecerá inactivo desde el momento en el que la cuenta del Timer se iguala a Pulse y continúa así hasta que finaliza el periodo. Teniendo esto en cuenta, CH Polarity se ha configurado como High, de modo que el canal va a estar siempre a nivel alto excepto cuando se encuentre inactivo. Por último, el valor de la variable Pulse se ha establecido en 1, no se permite 0, de esta forma el canal estará a nivel alto hasta que el Timer empiece a contar, superando inmediatamente el valor de la cuenta al de Pulse, y estableciendo un nivel bajo hasta que finalice el periodo creándose así un pulso de nivel bajo de igual duración al periodo del Timer.

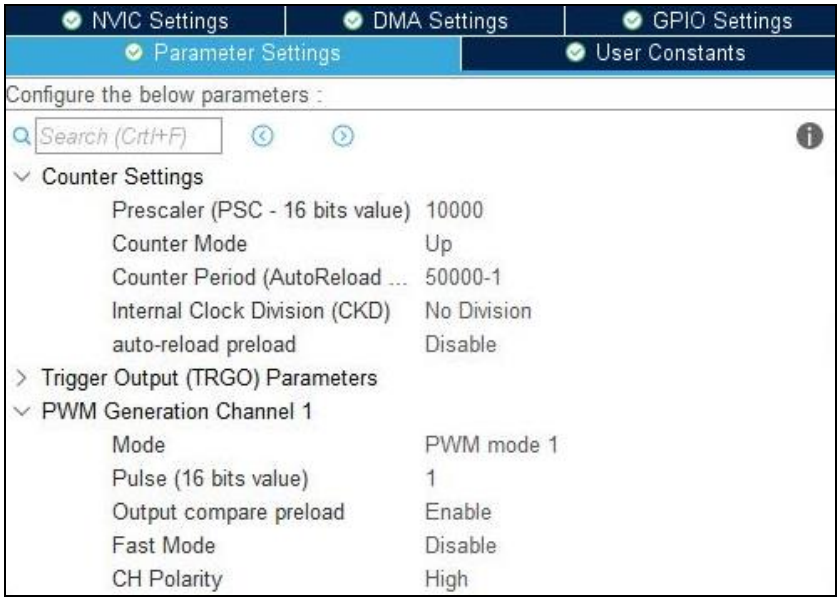



**Figura 22. Configuración periodo Timer 3 y PWM de su canal 1.**

La Figura 23 muestra un cronograma del funcionamiento del Timer 3 y su canal 1 configurado como PWM. Para poder apreciar mejor como se genera el pulso en el cronograma, la variable Pulse no tiene el valor 1. El funcionamiento real del Timer se ve reflejado en la Figura 24.

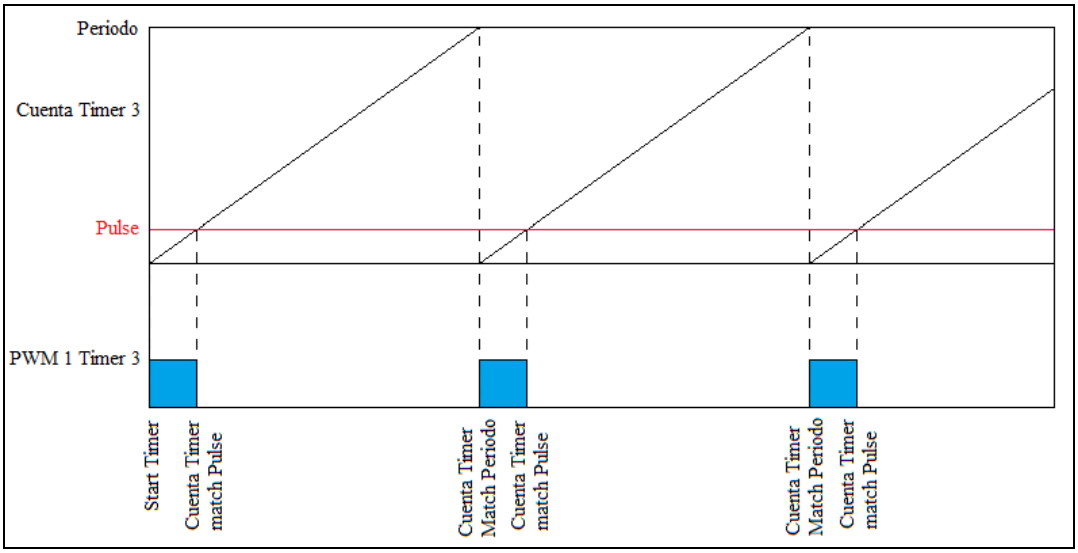



**Figura 23. Cronograma funcionamiento teórico Timer 3.**

Una vez se ha configurado todo en STM32CubeMX, se genera el código y se inicia con Keil uVision. En el código generado aparecen inicializados los dos Timers con toda la configuración establecida, pero hace falta completarlo con otras instrucciones como, por ejemplo, el inicio de cuenta de los Timers. Para ello se ha utilizado la función HAL_TIM_PWM_Start (), de la librería stm32f7xx_hal_tim.c, que permite iniciar la señal PWM configurada en uno de los canales de un Timer. Para iniciar la PWM del canal 1 del Timer 3, la función queda de la siguiente forma: HAL_TIM_PWM_Start (&htim3, TIM_CHANNEL1). Con el resto de canales se procede de igual manera.

A continuación, se ha cargado el programa en la tarjeta y conectado los LEDs anteriormente mencionados en los pines que actúan como PWM. El resultado ha sido el de la Figura 24, pequeños destellos de luz cada 5 segundos. El funcionamiento es correcto, pero aún hay algunos problemas que corregir, el más importante, que solo se genere un pulso cada vez que se necesite disparar una cámara, ya que, en esta prueba una vez comienza a funcionar la PWM no deja de generar pulsos continuamente, lo que produciría disparos indeseados de las cámaras.

Para solucionarlo, se puede utilizar la función HAL_TIM_PWM_Stop (), que para la PWM. Esta función se debería de aplicar cada vez que se produzca un periodo completo de la PWM, y posteriormente utilizar de nuevo HAL_TIM_PWM_Start () cuando se necesite. Además, se debería de emplear la interrupción del Timer o leer constantemente el estado del pin para poder detener la PWM justo cuando se complete un periodo, pero hay otro método más sencillo, el modo One-Pulse mode.

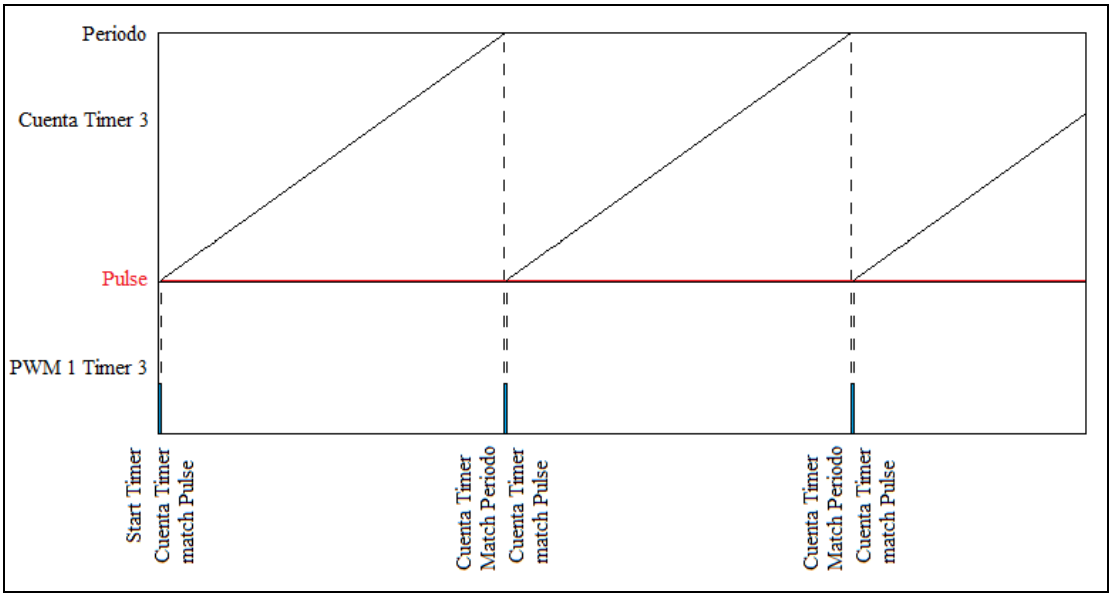



**Figura 24. Cronograma funcionamiento real Timer 3.**

Si se vuelve a revisar la Figura 21 se observa marcado el modo One-Pulse en STM32CubeMX. Este modo, permite parar el contador del Timer de forma automática en el próximo evento programado, es decir cuando se complete el periodo. Con este cambio, el programa funciona teóricamente acorde a lo que se muestra en la Figura 25.

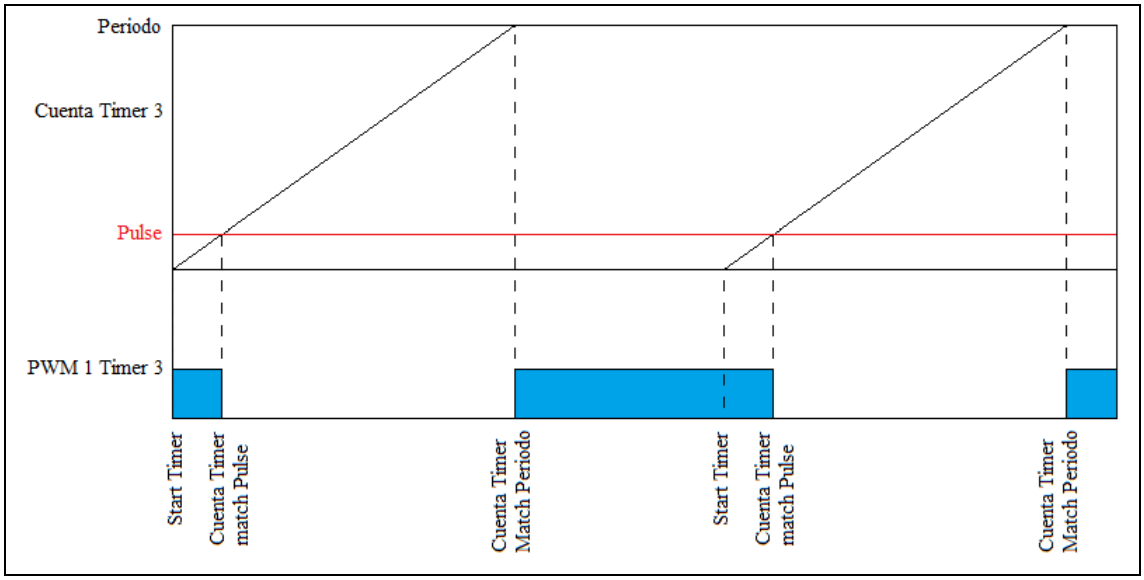



**Figura 25. Cronograma funcionamiento teórico Timer 3 (One-Pulse mode).**

En la Figura 26 se ve el funcionamiento real, cada vez que se llama a la función HAL_TIM_PWM_Start () se genera un pulso a nivel bajo de 5 segundos. Una vez acaba, el canal permanece en nivel alto hasta que se vuelve a llamar a la función generando un nuevo pulso.

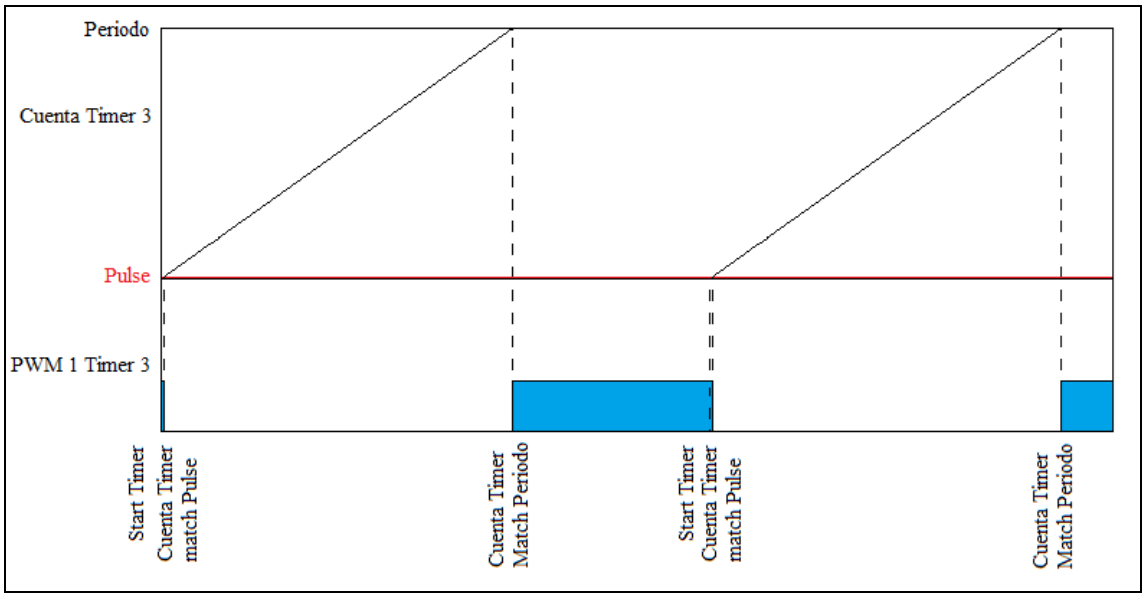



**Figura 26. Cronograma funcionamiento real Timer 3 (One-Pulse mode).**

Este método es totalmente válido, pero tiene la desventaja de no tener agrupados los pines que se corresponden con las señales PWM, sino que están fijos y distribuidos por la tarjeta. Para facilitar las conexiones se ha preferido que los pines se encuentren todos juntos, para lo cual se ha empleado otro procedimiento.

Se han seleccionado 6 pines de la tarjeta que se encuentran en la misma zona, parte inferior derecha visto desde arriba: PE10, PE11, PE12, PE13, PE14 y PE15. En estos pines se van a generar las señales de disparo de las 6 cámaras. En la Figura 27 se ven los pines agrupados.


**Figura 27. Pines seleccionados para generar las señales de disparo. [1]**

Para generar las señales se ha empleado un Timer por cada uno de los pines, cada Timer se encarga de medir el tiempo que el pin debe mantenerse a nivel bajo. Se han seleccionado el TIM2, TIM3, TIM4, TIM5, TIM9 y TIM12.

El funcionamiento teórico es el siguiente. Al inicio del programa los pines se configuran como GPIO_Output y se establecen a nivel alto hasta que se requiera generar una señal de disparo. En el momento de producir el pulso, el nivel del pin correspondiente se fija a nivel bajo y se inicia el Timer para que cuente hasta un valor programado de 50 ms (5 s para probar con los LEDs). Cuando el Timer termina la cuenta, activa su interrupción y en ella se vuelve a establecer un nivel alto en el pin. De esta forma el Timer cuenta exactamente el tiempo que necesita estar el pin a nivel bajo. En este programa también se va a utilizar el modo One-Pulse para que los Timer solo realicen un pulso.

Para realizar estas configuraciones se utiliza de nuevo STM32CubeMX y Keil uVision. Lo primero que se ha realizado ha sido la configuración del reloj del sistema, que es exactamente la misma del programa anterior, Figura 20. A continuación, de forma gráfica, se han configurado los pines que se han enumerado como salidas, GPIO_Output. En la Figura 28 se ve una captura de estos pines ya programados.

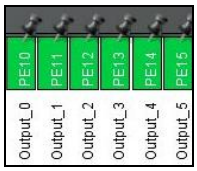



**Figura 28. Pines programados como GPIO_Output en STM32CubeMX.**

Después, se han configurado los 6 Timers, pero a diferencia del programa anterior, en este caso no se han configurado sus canales ya que no se necesitan. Como se ve en la Figura 29, para el Timer 4 se ha marcado de nuevo el modo One-Pulse, sus canales permanecen deshabilitados y se ha seleccionado una fuente de reloj interna. Con los Timers restantes se procede de la misma forma.

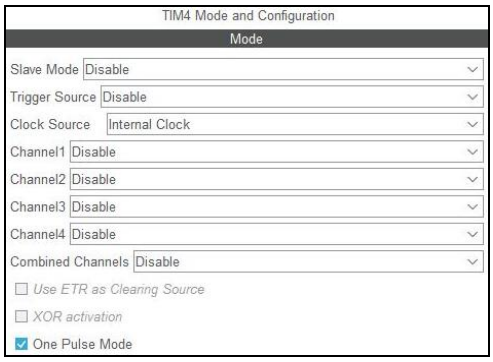



**Figura 29. Configuración Timer 4.**

Igual que ocurría antes, el pulso a nivel bajo se tiene que fijar en 50 ms, pero para poder trabajar desde casa con los LEDs se configura en 5 s. Los cálculos son los mismos que se han mostrado anteriormente. Se ha de remarcar que todos los Timers salvo el 9 pertenecen al bus APB1, al principio en mi caso no lo tuve en cuenta y los traté todos como si perteneciesen al mismo bus, de forma que en las pruebas el Timer 9 realizaba un pulso igual a la mitad de los demás, ya que la frecuencia de su bus APB2 es el doble que la del resto. La configuración para el periodo de los Timers se muestra en la Figura 30.

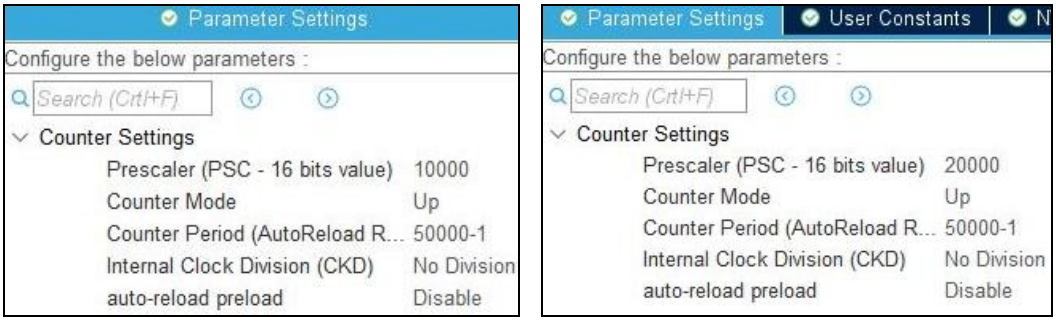



**Figura 30. Configuración periodo Timers 2, 3, 4, 5 y 12 a la izquierda y Timer 9 a la derecha.**

Además, se han activado las interrupciones de los 6 Timers, ya que son necesarias para cambiar desde ellas el estado de los pines. En la Figura 31 se muestra cómo se han activado en STM32CubeMX.

| NVIC Interrupt Table | Enabled | Preemption Priority | Sub Priority |
|---|---|---|---|
| TIM4 global interrupt | ☑ | 0 | 0 |

**Figura 31. Configuración interrupción Timer 4 en STM32CubeMX.**

Cuando se ha terminado de configurar estos parámetros se genera el código C y se abre con Keil uVision. En el código producido aparecen las librerías HAL, los Timers y pines iniciados y configurados, etc. En la Figura 32 se ven los pines iniciados y configurados.

```
201   /*Configure GPIO pin Output Level */
202   HAL_GPIO_WritePin(GPIOE, Output_0_Pin|Output_1_Pin|Output_2_Pin|Output_3_Pin
203                            |Output_4_Pin|Output_5_Pin, GPIO_PIN_SET);
204
205   /*Configure GPIO pins : Output_0_Pin Output_1_Pin Output_2_Pin Output_3_Pin
206                            Output_4_Pin Output_5_Pin */
207   GPIO_InitStructure.Pin = Output_0_Pin|Output_1_Pin|Output_2_Pin|Output_3_Pin
208                            |Output_4_Pin|Output_5_Pin;
209   GPIO_InitStructure.Mode = GPIO_MODE_OUTPUT_PP;
210   GPIO_InitStructure.Pull = GPIO_NOPULL;
211   GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_LOW;
212   HAL_GPIO_Init(GPIOE, &GPIO_InitStructure);
```

**Figura 32. Pines iniciados y configurados.**

En la Figura anterior en lugar del nombre del pin GPIO_PIN_10, aparece Output_0_Pin, ya que se han definido con otro nombre en el programa. Esto se debe a la etiqueta que se ha introducido en los pines en STM32CubeMX. Si se ve de nuevo la Figura 28 se puede apreciar que la etiqueta que tiene cada pin es la misma que se utiliza en el código.

A continuación, se ha de completar el código para que el programa funcione como se espera. Los pines ya se encuentran a nivel alto, la función HAL_GPIO_WritePin () de la Figura 32 permite modificar el estado de los pines seleccionados de un puerto. En este caso, los 6 pines pertenecen al puerto E y todos se establecen a un nivel alto.

El siguiente paso es iniciar el Timer que se corresponde con el pin por el cual se quiere generar la señal de disparo. Los Timer 2, 3, 4, 5, 9 y 12 se corresponden con los pines PE10, PE11, PE12, PE13, PE14 y PE15 respectivamente. Para iniciar la cuenta de los Timer y activar su interrupción al finalizar, se ha utilizado la función HAL_TIM_Base_Start_IT () de la librería stm32f7xx_hal_tim.c. Esta función inicia la cuenta del Timer indicado y activa su interrupción. Antes de iniciar el Timer se ha de cambiar el estado del pin con la función HAL_GPIO_WritePin (). Si se quiere generar una señal de disparo en el pin PE11 las líneas de código que se han usado son las que se ven en la Figura 33.

```
408         HAL_GPIO_WritePin(GPIOE, Output_1_Pin, GPIO_PIN_RESET);
409         HAL_TIM_Base_Start_IT(&htim3);
```

**Figura 33. Cambio de estado del pin PE11 y activación del Timer 3 y su interrupción.**

Cuando la cuenta del Timer se complete, es decir pasen 5 segundos desde que se ha iniciado, se produce su interrupción. Lo primero que hay que hacer en ella es limpiar el flag de la interrupción para que cuando se vuelva a solicitar una señal de disparo funcione. Después se ha de cambiar de nuevo el estado del pin a nivel alto. En la Figura 34 se muestra la rutina de atención a la interrupción.

```
165  void TIM3_IRQHandler(void)
166  {
167    HAL_TIM_IRQHandler(&htim3);
168    HAL_GPIO_WritePin(GPIOE, Output_1_Pin, GPIO_PIN_SET);
169  }
```

**Figura 34. Rutina de atención a la interrupción del Timer 3**

Este procedimiento se repite cada vez que se quiera generar una señal de disparo en cualquiera de los pines. Las pruebas que se han llevado a cabo en casa midiendo el tiempo que los LEDs permanecen apagados confirma que el funcionamiento es correcto, pero es necesario cambiar el periodo a 50 ms y comprobarlo con un osciloscopio, ya que este será el funcionamiento real. En la Figura 35 se aprecia una fotografía de la tarjeta y el montaje que se ha empleado para realizar las pruebas desde casa.

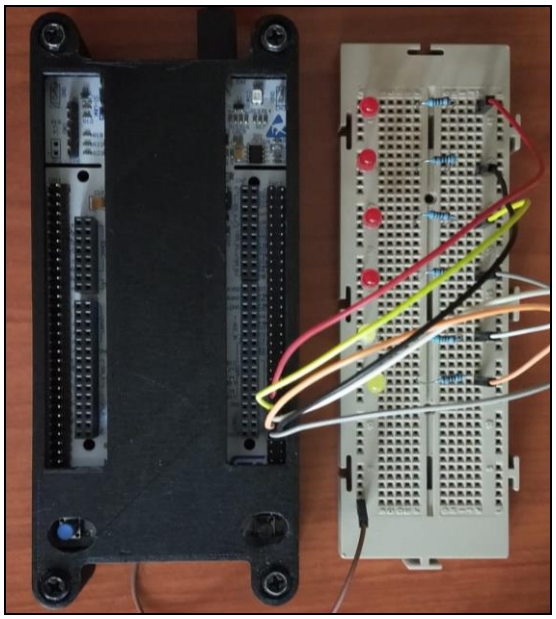


**Figura 35. Tarjeta STM32F676ZI conectada a los LEDs para la realización de pruebas.**

Se ha modificado la configuración de los Timers para que el periodo sea de 50 ms en lugar de 5 s. Únicamente se ha cambiado el número de cuentas, o lo que es lo mismo, el parámetro Counter Period de la Figura 30. Para conseguirlo se ha modificado por un valor de 500 en lugar de 50.000. Antes de realizar pruebas en el laboratorio se decidió comenzar a trabajar con la recepción de paquetes UDP a través de Ethernet, y cuando este objetivo se cumpliese, acudir a probar el conjunto completo.

## 4.2. Recepción a través de Ethernet de paquetes UDP generados por el LiDAR

Este bloque trata sobre la recepción de los paquetes UDP que el LiDAR manda a través de Ethernet, y la extracción de la información útil que contienen. Previo al inicio de este apartado, mi tutor y cotutor me guiaron y facilitaron información y ejemplos que me podían ser de utilidad.

Entre esta información se encuentra la documentacion sobre LwIP (*Lightweight IP* (*Internet Protocol*)) de STM32, de la que interesan las funciones de bajo nivel, y varios ejemplos de programas que envían y reciben paquetes UDP, como un echo server o echo client. El primer objetivo que se me propuso fue establecer la conexión necesaria para recibir los paquetes UDP, e indicar con un LED la llegada de cada uno de ellos.

En primer lugar, se ha comenzado a trabajar con el estudio del documento LwIP para conocer su funcionamiento y utilidad. LwIP es una pila TCP/IP gratuita y enfocada a reducir el uso de RAM, lo que hace que sea adecuada para su empleo en sistemas integrados. Entre los protocolos disponibles se encuentran: IPv4 e IPv6, ICMP, IGMP, UDP, TCP, DNS, SNMP, DHCP, PPP y ARP. En este proyecto se va a necesitar el protocolo UDP para recibir los paquetes del LiDAR.

LwIP cumple con el modelo TCP/IP que especifica como se deben formatear, transmitir, enrutar y recibir los datos para proporcionar comunicaciones de extremo a extremo. Este modelo incluye cuatro capas que se utilizan para ordenar todos los protocolos. La Figura 36 muestra esta organización.

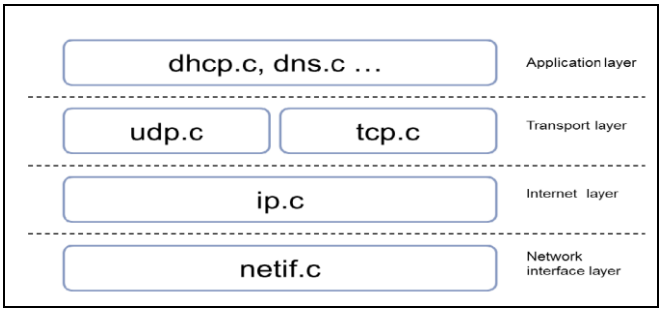



**Figura 36. Capas del modelo LwIP. [5]**

Para administrar los paquetes, LwIP utiliza una estructura de datos llamada pbuf. Esta estructura permite la asignación de memoria dinámica para guardar el contenido de un paquete, y permite que los paquetes residan en memoria estática. Los pbuf se pueden unir en una cadena, lo que permite que los paquetes se distribuyan en varias pbuf. En la Figura 37 se ve esta estructura.

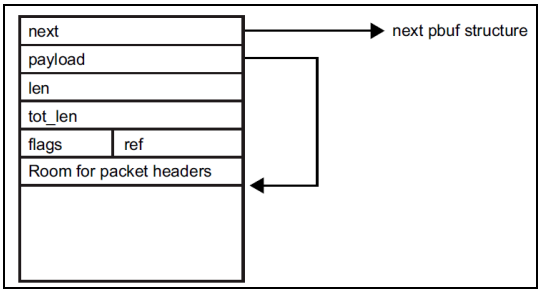



**Figura 37. Estructura pbuf. [5]**

Los elementos que forman la estructura son los siguientes:

- next: puntero al siguiente pbuf en una cadena de pbuf.
- payload: puntero a la carga útil de datos del paquete actual.
- len: tamaño del contenido de datos del pbuf.
- tot_len: es la suma de len del pbuf actual más todos los len de los siguientes pbuf de una cadena.
- ref: 4 bits que indican el número de punteros que apuntan al pbuf actual. Un pbuf se puede liberar de la memoria solo cuando este valor es 0.
- flag: 4 bits que indican el tipo de pbuf (pbuf_pool, pbuf_ram o pbuf_rom).

En la recepción de paquetes se utiliza el tipo pbuf_pool. Este permite asignar memoria rápidamente para el paquete recibido de un grupo de pbufs. Dependiendo del tamaño del paquete, se asignan una o varias pbufs encadenadas. El pbuf_ram no es adecuado para la recepción de paquetes porque utiliza asignación dinámica y requiere cierto tiempo. Para la transmisión de paquetes, se puede elegir el tipo de pbuf que mejor se ajuste según los datos a transmitir.

LwIP se puede implementar con sistema operativo o sin sistema operativo (standalone), eligiendo para este proyecto la segunda implementación. En cuanto a las funciones para Ethernet de LwIP en la figura 38 se ven resumidas.

| Function | Description |
|---|---|
| **low_level_init** | Calls the Ethernet driver functions to initialize the STM32F4xx Ethernet peripheral |
| **low_level_output** | Calls the Ethernet driver functions to send an Ethernet packet |
| **low_level_input** | Calls the Ethernet driver functions to receive an Ethernet packet. |
| **ethernetif_init** | Initializes the network interface structure (netif) and calls low_level_init to initialize the Ethernet peripheral |
| **ethernet_input** | Calls low_level_input to receive a packet then provide it to the LwIP stack |

**Figura 38. Resumen de funciones LwIP para Ethernet y su descripción. [5]**

En el resto del documento aparecen varios ejemplos LwIP, pasos para su configuración, modos de funcionamiento, etc. Como resumen LwIP es una pila TCP/IP que dispone del protocolo UDP necesario para este proyecto, y que con ayuda de las funciones HAL de STM32 facilitan la configuración.

Después de conocer lo que ofrece LwIP se han revisado los ejemplos que se me han proporcionado, estos programas se encuentran en la plataforma Github. Además, se han encontrado ejemplos para la tarjeta STM32F767ZI-Nucleo.

Mi profesor y cotutor me indicaron que mirase primero los ejemplos de LwIP echo server UDP y echo client UDP, con el objetivo de ponerlos en marcha. El primer problema que se ha encontrado es que estos programas no son compatibles con la tarjeta que se dispone, sino con el modelo STM32756G-Eval, aunque hay partes del código C de estos programas que si han sido de utilidad. Para la placa de este TFG solo se ha encontrado un único ejemplo con LwIP, un servidor con página web y sistema operativo.

Para comprobar si la conexión funciona se ha compilado y cargado el programa compatible con la tarjeta que se está utilizando. Previamente en el código se ha establecido una dirección fija IP para la tarjeta, 192.168.1.10, y se ha deshabilitado el DHCP (*Dynamic Host Configuration Protocol*) ya que se utiliza una IP estática. La IP de la conexión Ethernet del PC no se ha modificado ya que es válida. En la Figura 39 se pueden ver las configuraciones.

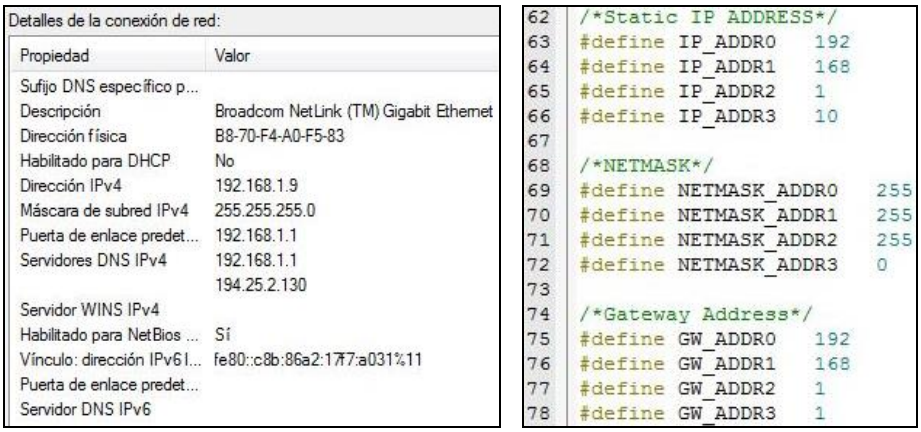



**Figura 39. Configuración inicial área local PC y configuración de red de la tarjeta.**

En la primera prueba al acceder a la web esta no cargaba. Tras varios intentos más sin resultados, en los que se han realizado pequeños cambios, se optó por cambiar la dirección IP por si entraba en conflicto con la conexión Wifi. La nueva dirección IP que se ha seleccionado para la tarjeta es 172.29.21.10. Al realizar esta modificación también se ha tenido que cambiar la IP de la conexión Ethernet del PC como se puede ver en la figura 40.

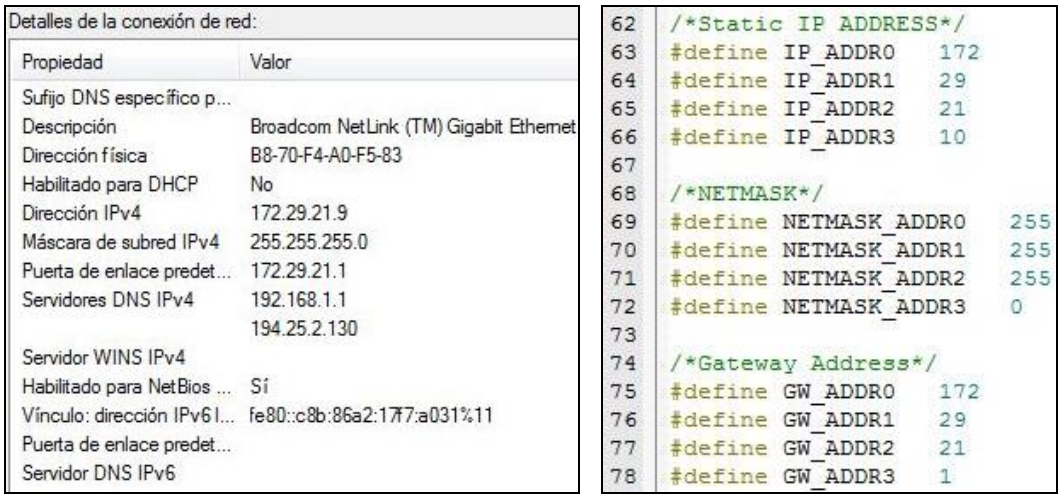



**Figura 40, Configuración final área local PC y configuración de red de la tarjeta.**

Al compilar y cargar de nuevo en la placa esta vez la página web sí se ha iniciado. También se ha realizado un ping a la dirección IP de la tarjeta para comprobar la conexión mediante otro método como se puede ver en la Figura 41. En esta prueba se puede apreciar que los 4 paquetes que se envían llegan a su destino y no se pierde ninguno.

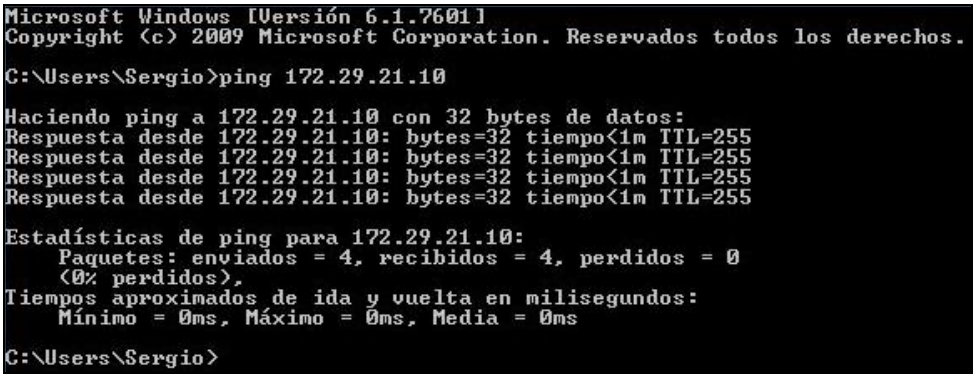



**Figura 41. Ping a la dirección IP de la tarjeta.**

Una vez la conexión se ha establecido la siguiente meta es recibir un paquete UDP. Para ello se han planteado dos opciones, generar un nuevo programa a través de STM32CubeMX y crear un echo server o modificar el programa anterior. En un primer momento se ha tratado de generar un proyecto nuevo configurando Ethernet y LwIP a través de STM32CubeMX, pero finalmente por simplicidad se ha optado por modificar el programa anterior, ya que ya tiene iniciada la pila LwIP y realizadas las configuraciones para Ethernet.

Para comenzar se ha eliminado todo el código que no se va a utilizar en este proyecto como el uso de un sistema operativo, hilos, semáforos, un servidor con página web, etc. De esta forma únicamente se han mantenido configuraciones básicas y la inicialización de la tarjeta. A continuación, se ha trasladado poco a poco el código C del ejemplo echo server, ya que contiene la programación de la recepción de paquetes UDP. Como hay partes del código programadas para otra tarjeta (pines, LEDs, etc.), se han realizado varias modificaciones ajustando el código a la placa con la que se trabaja. También se ha mantenido la configuración de red creada anteriormente.

Cuando el código del ejemplo LwIP echo server UDP se ha pasado sin errores, se ha compilado y cargado en la tarjeta STM32F767ZI. Una vez iniciado, se ha realizado de nuevo un ping a la dirección IP de la tarjeta para confirmar que la configuración de la red funciona. Este ejemplo contiene una aplicación externa, echotool, que se utiliza para hacer funcionar el programa. Para esta aplicación se debe seleccionar la IP destino, los puertos origen y destino, el protocolo de transporte, el número de solicitudes y el tiempo que de espera antes de cerrar la conexión. En la Figura 42 se ve el resultado, se ha seleccionado la IP de la tarjeta (172.29.21.10), el puerto de origen y destino es el mismo, 7, UDP como protocolo de transporte, 15 solicitudes y un tiempo de espera de 2 segundos. Las 15 solicitudes con la palabra “HOLA” a la tarjeta han llegado correctamente como se puede ver en los rectángulos rojos.

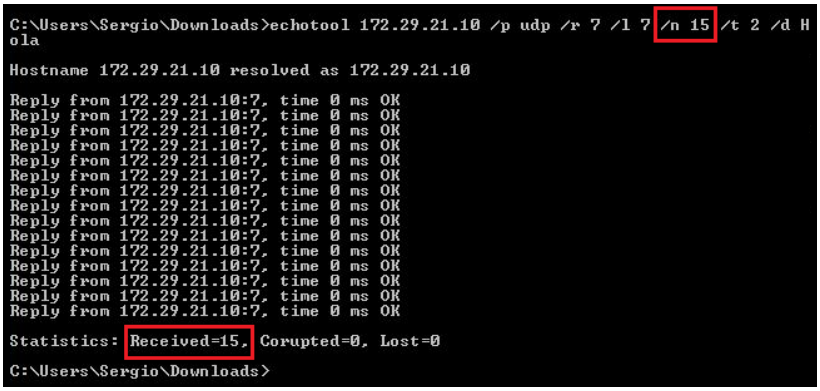



**Figura 42. Prueba echotool.**

Para comprobar el tráfico de paquetes a través de Ethernet se ha utilizado la herramienta Wireshark. Este programa es un analizador de protocolos que se utiliza para realizar análisis y solucionar problemas en redes de comunicaciones, análisis de datos y protocolos, y como herramienta didáctica. Con Wireshark se va a comprobar el tráfico de paquetes a través de Ethernet, ya que el programa que se ha probado anteriormente recibe solicitudes, pero además hace un eco de estas. En la Figura 43 aparece un análisis del tráfico Ethernet, en ella se pueden ver las 15 solicitudes que envía el PC a la tarjeta y los 15 ecos que hace la tarjeta hacia el PC.

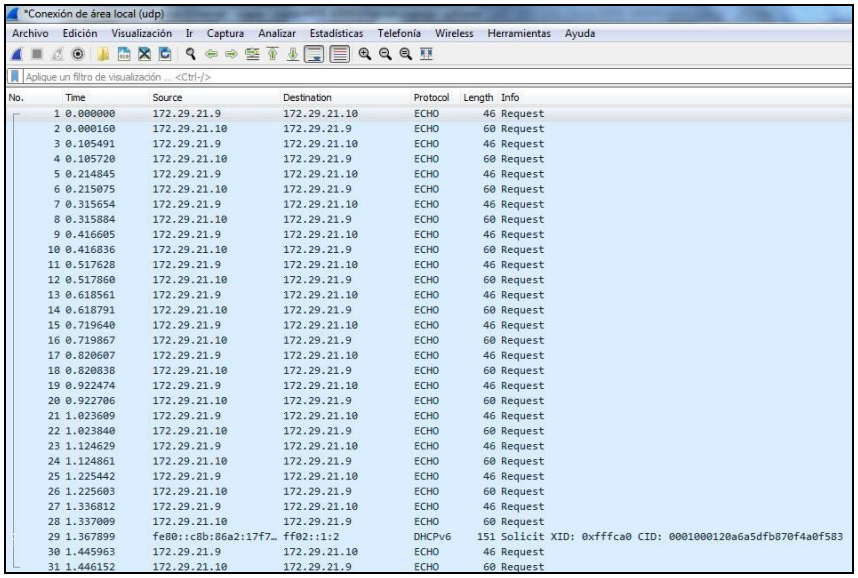



**Figura 43. Análisis de tráfico local con Wireshark.**

Esta herramienta también permite ver cada paquete y su contenido además de otras características. Se ha comprobado el primer paquete como se ve en la Figura 44. En ella se muestra la dirección IP de origen y destino, lo que se corresponde con un origen en el PC y un destino en la tarjeta, los puertos, el protocolo y el contenido de datos, en este caso la palabra "Hola".

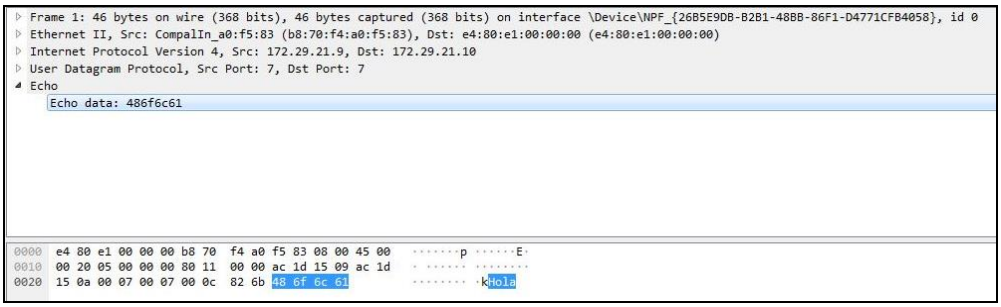



**Figura 44. Características primer paquete echotool PC-Tarjeta.**

También se ha comprobado el segundo paquete cuyas características se recogen en la Figura 45. En este caso, las direcciones IP se han intercambiado, siendo origen la tarjeta y destino el PC. Además, aparece de nuevo la palabra "Hola" en los datos del paquete confirmando de esta forma que se produce el eco.

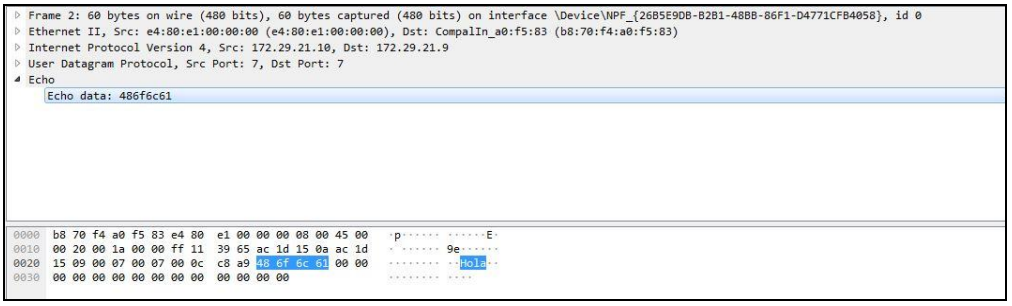



**Figura 45. Características segundo paquete echotool Tarjeta-PC.**

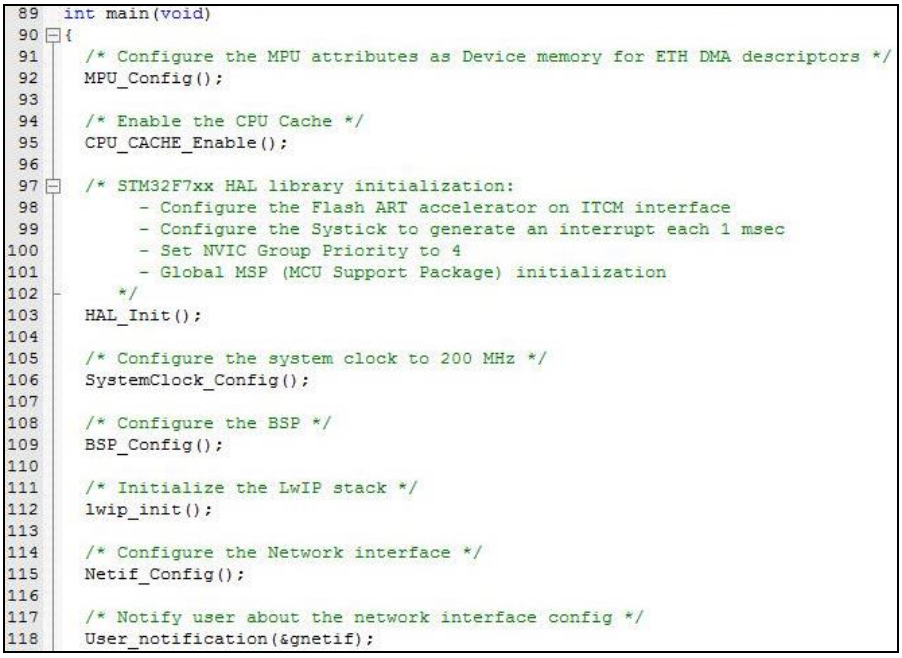
Una vez se ha conseguido que el ejemplo LwIP echo server UDP funcione en la tarjeta STM32F767ZI, se comienza a modificar el código para que funcione como se requiere. En primer lugar, se va a explicar el código que se mantiene después de eliminar todos los archivos que no son de utilidad para este proyecto.

Al iniciar el programa en el main se encuentran varias funciones, todas de inicialización y configuración. La primera de ellas MPU_Config (), configura los atributos de la unidad de protección de memoria MPU. Después, CPU_CACHE_Enable () activa la memoria caché de la CPU (*Central Processing Unit*), que contiene un pequeño conjunto de instrucciones a las que el microcontrolador accede con asiduidad de manera rápida, eficiente y sin obstáculos de por medio. La siguiente función, HAL_Init (), inicia la librería HAL de la que más adelante se utilizarán varias funciones. Otra de las funciones que se encuentran es SystemClock_Config (), que configura el reloj del sistema y de los periféricos, en este punto no se va a modificar, pero más adelante se va a revisar ya que es necesario para el uso de los Timers. También se encuentra BSP_Config (), que configura los LEDs de la tarjeta que se utilizarán como indicadores en diferentes situaciones.

La función lwip_init (), la más importante para realizar la comunicación UDP, inicia la pila LwIP. Esta función inicia todos los módulos necesarios para que la comunicación se pueda establecer. Dependiendo de si el programa tiene sistema operativo, los parámetros pueden variar. Dentro de esta función aparecen muchas otras sobre las que se apoya para un correcto funcionamiento. Como en el programa ejemplo se utiliza una comunicación UDP, no se van a realizar modificaciones, ya que como se ha comprobado el transporte de paquetes con el protocolo UDP funciona.

Otra de las funciones que se ven es Netif_Config (), que configura los parámetros de red como la IP, la máscara de red y el gateway. En la Figura 40 vista anteriormente se muestran estos parámetros.

Por último, antes del bucle while se encuentra la función User_notificaction (&genetif), que notifica al usuario por medio de los LEDs si la conexión se ha establecido satisfactoriamente. Por otro lado, el parámetro genetif es una estructura de red que guarda datos como la IP configurada. En la Figura 46 aparecen todas estas funciones.

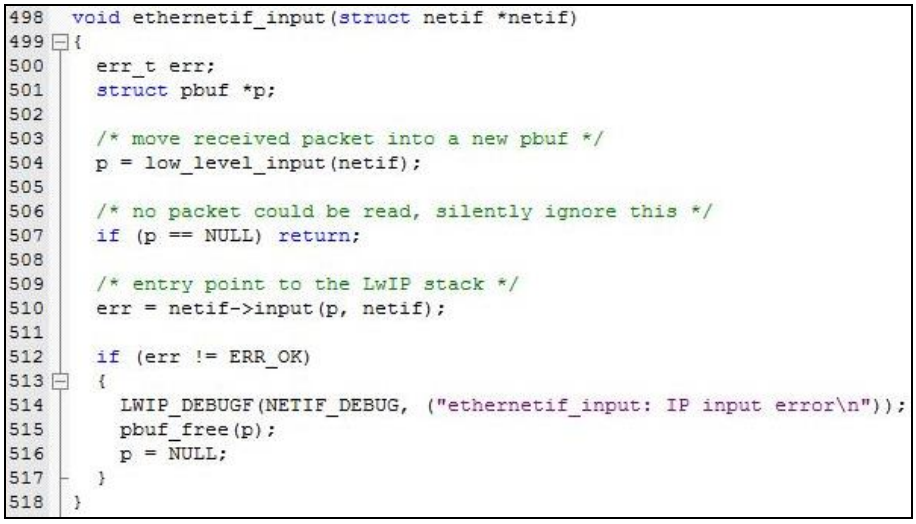


```
int main(void)
{
  /* Configure the MPU attributes as Device memory for ETH DMA descriptors */
  MPU_Config();

  /* Enable the CPU Cache */
  CPU_CACHE_Enable();

  /* STM32F7xx HAL library initialization:
       - Configure the Flash ART accelerator on ITCM interface
       - Configure the Systick to generate an interrupt each 1 msec
       - Set NVIC Group Priority to 4
       - Global MSP (MCU Support Package) initialization
     */
  HAL_Init();

  /* Configure the system clock to 200 MHz */
  SystemClock_Config();

  /* Configure the BSP */
  BSP_Config();

  /* Initialize the LwIP stack */
  lwip_init();

  /* Configure the Network interface */
  Netif_Config();

  /* Notify user about the network interface config */
  User_notification(&gnetif);
```

**Figura 46. Funciones de inicialización y configuración en el main ().**

Dentro del bucle while hay dos funciones, ethernetif_input (&genetif) y sys_check_timeouts (). La primera de ellas, es la función que se debe llamar todo el tiempo en el programa ya que, si algún paquete llega a la tarjeta a través de los buffers de Ethernet, esta función lo lee y manda a LwIP para su manipulación. La segunda, comprueba si ha expirado el tiempo de espera.

La función ethernetif_input (&genetif), que se encuentra en el archivo ethernetif.c, es sobre la que se ha de trabajar, ya que es en ella donde comienza la lectura y gestión de los paquetes. Esta función se debe llamar cuando un paquete está listo para ser leído desde la interfaz. Utiliza la función de bajo nivel low_level_input (), que maneja la recepción real de bytes desde la interfaz de red. Luego se determina el tipo de paquete recibido y se llama a la función de entrada apropiada. En la Figura 47 se muestra el código esta función.



```
void ethernetif_input(struct netif *netif)
{
  err_t err;
  struct pbuf *p;

  /* move received packet into a new pbuf */
  p = low_level_input(netif);

  /* no packet could be read, silently ignore this */
  if (p == NULL) return;

  /* entry point to the LwIP stack */
  err = netif->input(p, netif);

  if (err != ERR_OK)
  {
    LWIP_DEBUGF(NETIF_DEBUG, ("ethernetif_input: IP input error\n"));
    pbuf_free(p);
    p = NULL;
  }
}
```

**Figura 47. Función ethernetif_input.**

El resto de archivos que se han mantenido son: app_ethernet.c, en el que se encuentran varias funciones que indican el estado de la conexión, stm32f7xx_it.c, en el que están las rutinas de atención a las interrupciones y ethernetif.c, que contiene varias funciones para la gestión de paquetes.

El LiDAR manda paquetes a través de Ethernet, pero para poder trabajar desde casa sin necesidad de tener el dispositivo de forma física, me han proporcionado un archivo .pcapng grabado del LiDAR que se ha de replicar. Este archivo contiene datos reales del Velodyne que servirán para realizar las primeras pruebas antes de llevar el proyecto al laboratorio.

Para replicar este archivo, mi tutor me comentó que podría usar la herramienta PReplay que permite enviar de nuevo archivos de datos previamente capturados. Además, mantiene la diferencia de tiempo entre dos paquetes, aunque con una diferencia de micro/milisegundos. Primero lee el archivo que se va a transmitir y después determina la diferencia de tiempo con el siguiente paquete.

Tras unos primeros intentos la herramienta PReplay notificaba el error de la Figura 48. Para solucionarlo durante la búsqueda de información me di cuenta de que los archivos que permite replicar son tipo .cap o .pcap, de forma que el archivo del que se disponía no era válido. Para poder utilizarlo se ha pasado el archivo .pcapng a .pcap generándose con éxito los 127.914 paquetes en la siguiente prueba, Figura 49.

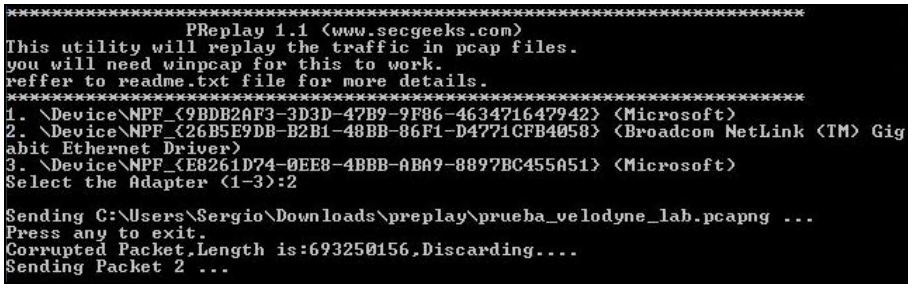



**Figura 48. Prueba errónea PReplay.**

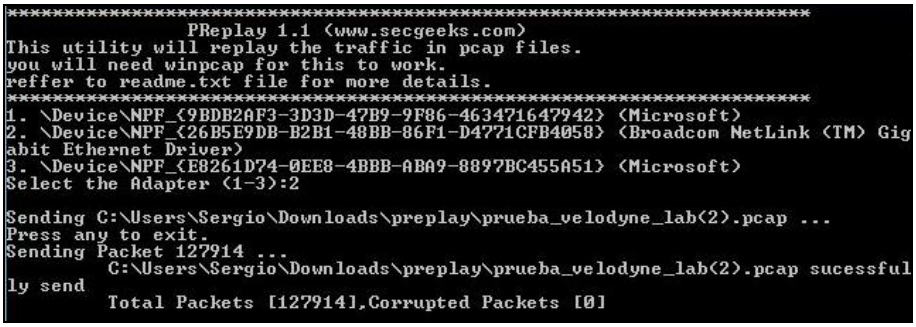
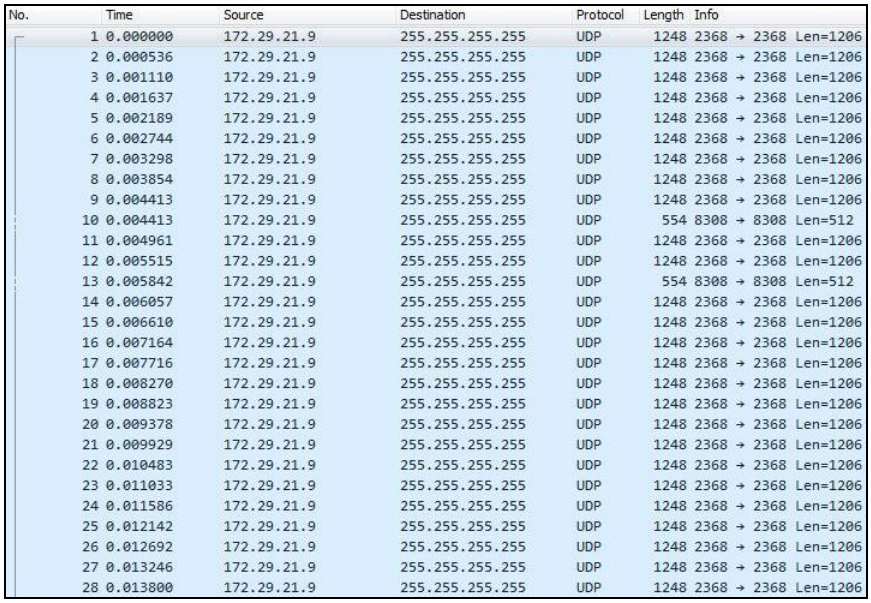



**Figura 49. Prueba correcta PReplay.**

Se ha utilizado Wireshark para comprobar si el archivo se replica correctamente, obteniéndose como resultado la Figura 50. Esta solo recoge los primeros 28 paquetes, pero en total se han trasmitido 127.914.



| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 2 | 0.000536 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 3 | 0.001110 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 4 | 0.001637 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 5 | 0.002189 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 6 | 0.002744 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 7 | 0.003298 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 8 | 0.003854 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 9 | 0.004413 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 10 | 0.004413 | 172.29.21.9 | 255.255.255.255 | UDP | 554 | 8308 → 8308 Len=512 |
| 11 | 0.004961 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 12 | 0.005515 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 13 | 0.005842 | 172.29.21.9 | 255.255.255.255 | UDP | 554 | 8308 → 8308 Len=512 |
| 14 | 0.006057 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 15 | 0.006610 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 16 | 0.007164 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 17 | 0.007716 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 18 | 0.008270 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 19 | 0.008823 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 20 | 0.009378 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 21 | 0.009929 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 22 | 0.010483 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 23 | 0.011033 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 24 | 0.011586 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 25 | 0.012142 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 26 | 0.012692 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 27 | 0.013246 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |
| 28 | 0.013800 | 172.29.21.9 | 255.255.255.255 | UDP | 1248 | 2368 → 2368 Len=1206 |

**Figura 50. Captura Wireshark de la réplica del paquete de datos del LiDAR.**

A continuación, ya se dispone de las herramientas necesarias para poder realizar pruebas sin necesitar el LiDAR. Para comprobar si los paquetes UDP llegan a la tarjeta, se va a configurar uno de los LEDs de la placa para que se encienda cada vez que se reciba un paquete. En la Figura 47 se ha mostrado el código de la función que lee si ha llegado un paquete o no, y es en ella donde se han realizado los cambios.

Para comprobarlo rápidamente, se ha añadido un else al if que comprueba si el puntero es NULL. En el if, se ha utilizado la función BSP_LED_Off (LED3) que apaga el LED rojo en ausencia de un paquete o cuando este no se puede leer. Por otro lado, en el else se ha añadido BSP_LED_On (LED3) que enciende el LED rojo de la tarjeta cuando existe un paquete válido.

Al comprobarlo utilizando la réplica del archivo del LiDAR, el LED rojo se enciende durante toda la transmisión, lo que indica que los paquetes se reciben correctamente, y se apaga cuando finaliza. Además, una vez se termina la transmisión de los paquetes, el LED se ilumina cada cierto tiempo ya que existe tráfico de red.

Después de confirmar que los paquetes llegan a la tarjeta, estas líneas de código que se han añadido se van a sustituir por la función de la Figura 51. Esta función modifica el estado del LED teniendo en cuenta si ya estaba encendido o apagado y la llegada de paquetes.

```
317  void Actualizar_Leds (char estado_led, char estado_paquete)
318  {
319    if((estado_led == 1)&&(estado_paquete == 0))
320    {
321      BSP_LED_Off(LED3);
322      flag_led = 0;     //flag_led = 0 indica que el led esta apagado.
323      return;
324    }
325    if((estado_led == 0)&&(estado_paquete == 1))
326    {
327      BSP_LED_On(LED3);
328      flag_led = 1;     //flag_led = 1 indica que el led esta encendido.
329    }
330  }
```

**Figura 51. Función de estado del LED ante la llegada de paquetes UDP.**

Para continuar con el proyecto, el siguiente paso consiste en leer lo que hay en cada paquete y filtrar la recepción de paquetes UDP, ya que solo son útiles los que proceden del LiDAR. Teniendo en cuenta la Figura 47, se puede observar que cada nuevo paquete se copia en la variable 'p' de tipo pbuf.

Dentro de la estructura pbuf del paquete hay varios campos, uno de ellos, len, contiene el tamaño del paquete, y se ha utilizado como primera solución a la hora de filtrarlos. Este método no es óptimo ya que pueden existir paquetes con el mismo tamaño que no se deseen.

Una buena solución es comparar la IP y puerto de origen, ya que de esta forma los paquetes UDP que se reciban serán exclusivos del LiDAR. Utilizando Whiresark, se ha seleccionado uno de los 127.914 paquetes disponibles del archivo pcap, para comprobar que información contiene, y en caso de estar la IP y el puerto, conocer su posición dentro de los datos. Para poder realizar esta comparación se ha de extraer la información del paquete, concretamente el campo payload de la estructura pbuf. En la Figura 52, se observa el contenido de uno de los paquetes.

La zona con fondo azul se corresponde con los 1.206 bytes de datos, pero antes de esta parte se encuentra la información que nos interesa en este momento. En los recuadros rojos se encuentran los valores en hexadecimal de la dirección IP ("ac 1d 15 09" se corresponde con 172.29.21.9) y el puerto origen (0940 con 2368), y en los verdes los de destino (ff ff ff ff con 255.255.255.255 y 0940 con 2368).

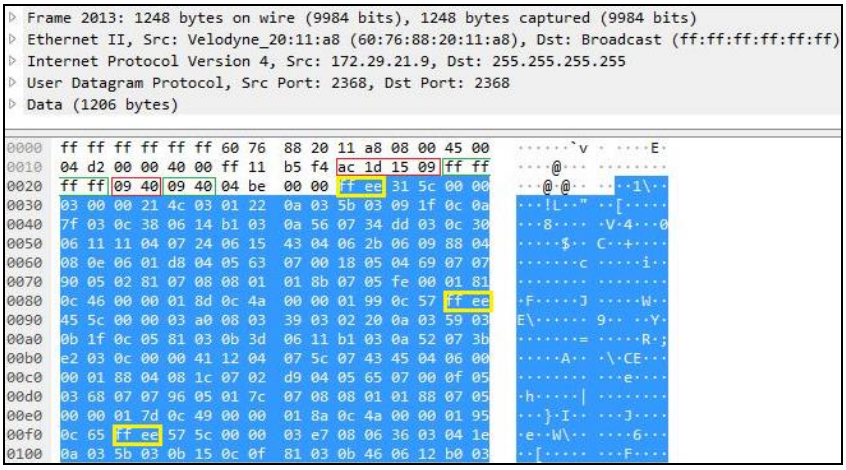



**Figura 52. Captura Wireshark con la información de un paquete del LiDAR.**

Conociendo que la información de la IP y el puerto se encuentra en el paquete, se ha de extraer utilizando una variable global para así poder verla en tiempo real, ya que Keil uVision no permite monitorizar variables locales. Se ha creado la variable '(char*) comprobar' para pasar los datos apuntados por payload. Además, en otra variable se ha guardado la dirección a la que apunta payload para ver los datos que hay en la memoria.

Para poder diferenciar correctamente el contenido del paquete se ha utilizado de nuevo la aplicación echotool. Con esta herramienta se ha enviado un solo paquete UDP con el siguiente contenido: "HOLAPPAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA". En la Figura 53 izquierda, se puede observar cómo al acceder a la dirección de memoria, aparecen las direcciones IP y los puertos origen y destino, y cómo a continuación, remarcado en azul, el texto en hexadecimal.

No se ha mandado directamente la réplica de datos del LiDAR, ya que es más sencillo buscar un dato repetido varias veces como la letra A, 0x41. Una vez se ha encontrado en memoria el contenido del paquete se ha probado con los datos del Velodyne. El resultado también se muestra en la Figura 53 derecha.

Utilizando este procedimiento y atendiendo a la Figura 53, se ha llegado a la conclusión de que la IP origen se encuentra en la posición 26 del paquete, teniendo en cuenta que cada posición es de 1 byte. La IP origen se corresponde con los bytes 26, 27, 28 y 29, mientras que el puerto origen lo hace con la 34 y 35, y el de destino con la 36 y 37.

Del mismo modo a partir de la posición 42, cuando finaliza la cabecera de 42 bytes del paquete, comienzan los datos, es decir la información que va a mandar el LiDAR.

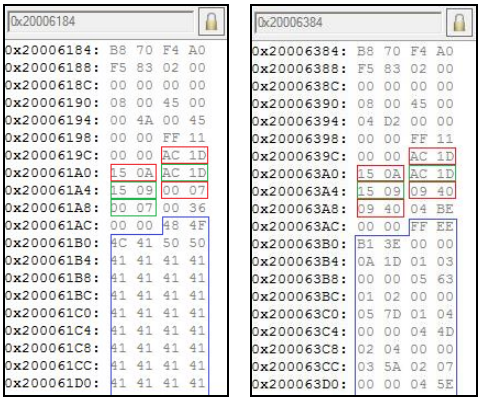



**Figura 53. Captura de memoria con la información del paquete.**

Para filtrar los paquetes se ha añadido el código que se muestra en la Figura 54. En el array 'puntero []' se guardan los datos del paquete desde la posición 26 hasta la 38 mediante un bucle for. Entre esos datos se encuentran las IP y los puertos origen y destino. Después, mediante el if que aparece en el código, se compara la IP origen y los puertos de cada paquete con los esperados. En caso de que el paquete pase el filtro, se actualiza el LED de estado de recepción de paquetes y se llama a la función Comprobar_Rotacional () que se explicará más adelante.

```
769  void ethernetif_input(struct netif *netif)
770  {
771    err_t err;
772    struct pbuf *p;
773
774    /* move received packet into a new pbuf */
775    p = low_level_input(netif);
776
777    /* no packet could be read, silently ignore this */
778    if (p == NULL)
779    {
780      flag_paquete = 0;     //flag_paquete = 0 indica que no ha llegado ningun paquete
781      Actualizar_Leds(flag_led, flag_paquete);
782      return;
783    }
784    /* verificacion de paquete valido */
785    else
786    {
787      comprobacion = (char*)p->payload;
788      k=0;
789      for(i=26;i<38;i++)
790        puntero[k++] = *(comprobacion+i);
791      if((puntero[0]==(char)(0xAC))&&(puntero[1]==(char)(0x1D))&&(puntero[2]==(char)(0x15))
792        &&(puntero[3]==(char)(0x09))&&(puntero[8]==(char)(0x09))&&(puntero[9]==(char)(0x40))
793        &&(puntero[10]==(char)(0x09))&&(puntero[11]==(char)(0x40)))
794        {
795          flag_paquete = 1;     //flag_paquete = 1 indica que ha llegado un paquete del LIDAR
796          Actualizar_Leds(flag_led, flag_paquete);
797          Comprobar_Rotacional (((struct HDLDataPacket *)(comprobacion+42)));
798        }
799    }
```

**Figura 54. Función ethernetif_input final.**

Hasta este momento se ha conseguido establecer la conexión necesaria para enviar y recibir paquetes UDP, y se han filtrado por IP y puerto estos paquetes para únicamente recibir aquellos generados por el LiDAR. Además, accediendo a los parámetros de la estructura pbuf, se ha extraído la dirección de memoria donde se encuentran los datos de los paquetes pudiendo visualizarse.

A continuación, se han de almacenar los datos que envía el LiDAR en cada paquete. Retomando lo explicado sobre el Velodyne, cada paquete de datos que envía tiene la estructura de la Figura 6. Teniendo esto en cuenta, para facilitar la gestión de la información de forma ordenada, se ha creado la estructura de la Figura 55.

```
#define HDL_GPS_PORT                8308
#define HDL_DATA_PORT               2368
#define HDL_NUM_ROT_ANGLES         36001
#define HDL_LASER_PER_FIRING          32
#define HDL_FIRING_PER_PKT            12
#define HDL_MAX_FIRING_PER_TURN     2500

extern int HDL_ANGLE_FIRE_1, HDL_ANGLE_FIRE_2, HDL_ANGLE_FIRE_3, HDL_ANGLE_FIRE_4, HDL_ANGLE_FIRE_5, HDL_ANGLE_FIRE_6;

#pragma pack(push, 1)

typedef struct HDLLaserReturn {
    unsigned short distance;
    unsigned char intensity;
} HDLLaserReturn;
#pragma pack(pop)

struct HDLFiringData {
    unsigned short blockIdentifier;
    unsigned short rotationalPosition;
    HDLLaserReturn laserReturns[HDL_LASER_PER_FIRING];
};

struct HDLDataPacket {
    struct HDLFiringData firingData[HDL_FIRING_PER_PKT];
    unsigned int gpsTimestamp;
    unsigned char blank1;
    unsigned char blank2;
};
```

**Figura 55. Estructura de gestión de la información del Velodyne.**

De los 1.248 bytes que contiene cada paquete, los 42 bytes correspondientes a la cabecera solo se utilizan para filtrar, por lo que no se van a guardar. De los 1.206 bytes restantes, 1.200 son de datos y los últimos 6 de estado.

La estructura principal, HDLDataPacket, sirve para almacenar estos 1.206 bytes. Está formada por un array de 12 unidades de otra estructura, HDLFiringData, y de 3 variables con un tamaño total de 6 bytes que almacenan los datos de estado.

En las 12 unidades del array de HDLFiringData se guardan los 1.200 bytes de datos distribuidos en bloques de 100 bytes. Como ya se ha mencionado, los 2 primeros bytes de cada bloque se corresponden con un identificador, los 2 siguientes con la posición, y los 96 bytes restantes con la información de disparo de los 32 láseres.

Para almacenar estos datos, HDLFiringData está formada por 2 variables de 2 bytes que contienen el identificador y la posición, y otro array de 32 unidades de la estructura HDLLaserReturn, formada por 2 variables de 2 y 1 byte que guardan respectivamente los datos de distancia e intensidad de cada láser. En la Tabla 1 se recoge como se almacenan los datos de cada paquete.

| 1.248 bytes | | | | | | |
|---|---|---|---|---|---|---|
| Cabecera | Datos | | | Datos de estado | | |
| 42 bytes | 1.200 bytes | | | 6 bytes | | |
| | HDLDataPacket | | | | | |
| | HDLFiringData [12] | | | gpsTimestamp | blank 1 | blank 2 |
| | 100 bytes | | | 2 bytes | 2 bytes | 2 bytes |
| | blockIdentifier | rotationalPosition | HDLLaserReturn [32] | | | |
| | 2 bytes | 2 bytes | 96 bytes | | | |
| | | | distance | intensity | | |
| | | | 2 bytes | 1 byte | | |

**Tabla 1. Almacenamiento de la información del paquete UDP.**

En la Figura 52 se muestra la información de un paquete del LiDAR. En los datos, zona azul, se puede ver como cada 100 bytes aparece un identificador 0xEEFF remarcado con un rectángulo amarillo que se almacena en blockIdentifier. Los dos siguientes bytes contienen la posición y se guardan en rotationalPosition, por último, llegan los datos de disparo de los 32 láseres que se almacenan en el array HDLLaserReturn. De esta forma se ubica la información de cada paquete en las diferentes variables que se han creado, y así tratarla y gestionarla en función de los valores recibidos.

Para comprobar que en las variables aparece correctamente la información correspondiente se ha creado una variable global, m_DATAPacket, del tipo de la estructura del paquete HDLDataPacket. Después se le asigna el valor de los datos del paquete a partir de la posición 42, para evitar introducir la cabecera. En la Figura 56 se muestra dicho código.

```
529     if((puntero[0]==(char)(0xAC))&&(puntero[1]==(char)(0x1D))&&(puntero[2]==(char)(0x15))
530       &&(puntero[3]==(char)(0x09))&&(puntero[8]==(char)(0x09))&&(puntero[9]==(char)(0x40))
531       &&(puntero[10]==(char)(0x09))&&(puntero[11]==(char)(0x40)))
532       {
533         flag_paquete = 1;     //flag_paquete = 1 indica que ha llegado un paquete con interes
534         Actualizar_Leds(flag_led, flag_paquete);
535         m_DATAPacket=*((struct HDLDataPacket *)(comprobacion+42));
```

**Figura 56. Código para el almacenamiento de los paquetes UDP.**

Al ejecutar el programa en tiempo real la variable m_DATAPacket se puede ver en la ventana de Keil uVision. En la Figura 57 se muestra captura de ventana de Keil uVision con estos datos. En ella se puede observar la dirección de memoria donde se encuentran los datos, 0x200003F8, y el valor asignado a las variables. Para determinar si esta asignación es correcta se ha de comprobar que tanto blockIdentifier como rotationalPosition tengan valores coherentes.

| Name | Value | Type |
|---|---|---|
| m_DATAPacket | 0x200003F8 &m_DAT... | struct HDLDataPacket |
| firingData | 0x200003F8 &m_DAT... | struct HDLFiringData[... |
| [0] | 0x200003F8 &m_DAT... | struct HDLFiringData |
| blockIdentifier | 0xEEFF | unsigned short |
| rotationalPosition | 0x0FB3 | unsigned short |
| laserReturns | 0x200003FC | struct HDLLaserReturn... |

**Figura 57. Captura de pantalla de las variables en Keil uVision en tiempo real.**

En esa imagen, blockIdentifier tiene el valor del identificador 0xEEFF y rotationalPosition un valor comprendido entre 0 y 36.000, como se indica en las hojas de características del dispositivo, ya que 0x0FB3 en decimal es 4.019. Para realizar una comprobación más, se han seleccionado 2 paquetes consecutivos del archivo .pcap del LiDAR y se ha comprobado el valor de sus variables blockIdentifier y rotationalPosition para confirmar que se almacenan correctamente. En la Tabla 2 se recoge la información.

| Paquete 1 | | | Paquete 2 | | |
|---|---|---|---|---|---|
| blockIdentifier | rotationalPosition | valor decimal | blockIdentifier | rotationalPosition | valor decimal |
| 0xEEFF | 0x3EB1 | 16.049 | 0xEEFF | 0x3F90 | 16.272 |
| 0xEEFF | 0x3EC3 | 16.067 | 0xEEFF | 0x3FA3 | 16.291 |
| 0xEEFF | 0x3ED6 | 16.086 | 0xEEFF | 0x3FB6 | 16.310 |
| 0xEEFF | 0x3EE9 | 16.105 | 0xEEFF | 0x3FC7 | 16.327 |
| 0xEEFF | 0x3EFC | 16.124 | 0xEEFF | 0x3FDB | 16.347 |
| 0xEEFF | 0x3F0E | 16.142 | 0xEEFF | 0x3FED | 16.365 |
| 0xEEFF | 0x3F21 | 16.161 | 0xEEFF | 0x4000 | 16.384 |
| 0xEEFF | 0x3F33 | 16.179 | 0xEEFF | 0x4012 | 16.402 |
| 0xEEFF | 0x3F46 | 16.198 | 0xEEFF | 0x4025 | 16.421 |
| 0xEEFF | 0x3F59 | 16.217 | 0xEEFF | 0x4038 | 16.440 |
| 0xEEFF | 0x3F6B | 16.235 | 0xEEFF | 0x40A4 | 16.458 |
| 0xEEFF | 0x3F7D | 16.253 | 0xEEFF | 0x405D | 16.477 |

**Tabla 2. Valores de blockIdentifier y rotationalPosition en dos paquetes consecutivos.**

Una vez se extrae la información de los paquetes del LiDAR correctamente se ha de trabajar con ella. El proyecto consiste en disparar las cámaras cuando estas se encuentren en ciertos ángulos, es decir dependiendo del valor de la variable rotationalPosition.

En el siguiente apartado se va a lleva a cabo la unión de los dos últimos bloques, la generación de la señal de disparo y la extracción de la información de los paquetes UDP del LiDAR.

## 4.3. Disparo de las cámaras en relación a la posición del LiDAR

En este bloque el objetivo que se ha planteado es recibir y leer los paquetes UDP como se ha hecho anteriormente y, además añadir la configuración de los Timers para disparar las cámaras, ya que se encuentran en dos programas diferentes. Este es el paso previo antes de acudir a realizar las pruebas al laboratorio.

Para llevar esto a cabo se ha mantenido el programa que realiza la recepción y lectura de los paquetes, y en él se han configurado los Timers con sus interrupciones como se ha explicado anteriormente.

Lo primero que se ha realizado ha sido comprobar cuál es la configuración del reloj de este programa, ya que como se ha visto, es necesario conocer a que frecuencia trabajan los Timers para conseguir el funcionamiento requerido.

La configuración actual tiene una frecuencia de 200 MHz en el reloj del sistema que hay que mantener, ya que afecta al Ethernet. En cuanto a los Timers es una frecuencia más tediosa de trabajar, debido a que está dividida entre 3 y aparecen números decimales, por lo que se ha decidido establecer la configuración realizada en la Figura 20, ya que mantiene los 200 MHz en el reloj del sistema y se tienen verificadas las operaciones con los Timers. En la Figura 58 se muestra la configuración final del reloj.

```
/**
  * @brief  System Clock Configuration
  *         The system Clock is configured as follow :
  *            System Clock source            = PLL (HSE)
  *            SYSCLK(Hz)                     = 200000000
  *            HCLK(Hz)                       = 200000000
  *            AHB Prescaler                  = 1
  *            APB1 Prescaler                 = 4
  *            APB2 Prescaler                 = 2
  *            HSE Frequency(Hz)              = 8000000
  *            PLL_M                          = 4
  *            PLL_N                          = 200
  *            PLL_P                          = 2
  *            PLL_Q                          = 8
  *            PLL_R                          = 2
  *            VDD(V)                         = 3.3
  *            Main regulator output voltage  = Scale1 mode
  *            Flash Latency(WS)              = 6
  * @param  None
  * @retval None
  */
```

**Figura 58. Configuración final del reloj del sistema.**

Una vez se tiene el reloj configurado se ha de copiar todo el código relacionado de los Timers como las funciones de inicialización, configuración o las interrupciones, así como los pines que se han utilizado como salida. Cuando se finaliza este proceso, se han repetido las pruebas realizadas con los Timers para comprobar que están operativos y si su configuración es correcta.

Con ambas partes funcionando en un mismo programa, se realiza la función que se va a encargar de comprobar la posición del LiDAR y realizar los disparos de las cámaras cuando se requiera. Esta función es Comprobar_Rotacional () y se muestra en el Anexo I.

Comprobar_Rotacional () ha sufrido modificaciones desde su inicio hasta su forma final, varias de ellas a causa de las pruebas que se han realizado y que se comentarán más adelante. Se han podido realizar test desde casa y solucionar algunos de los problemas que se han encontrado.

Esta función tiene como argumento un puntero dirigido hacia el paquete que acaba de llegar. Se han de recorrer los 12 bloques de 100 bytes existentes dentro del paquete, para ello se utiliza un bucle for desde 0 hasta HDL_FIRING_PER_PKT que tiene un valor de 12. Dentro de ese bucle se asigna a una variable, firingData de tipo struct HDLFiringData, cada uno de los bloques. Esta es la variable con la que se va a trabajar durante toda la función.

Los valores de rotationalPosition van de 0 hasta 36.000, o lo que es lo mismo desde 0 hasta 360 grados. Para ver si se ha realizado una rotación completa es tan sencillo como comprobar si la posición actual del LiDAR es menor que la posición anterior, ya que si la posición va aumentando hacia 360 grados no se ha completado una rotación, pero si se reinicia y empieza desde 0 grados significa que se ha completado una vuelta.

HDL_ANGLE_FIRE_1, HDL_ANGLE_FIRE_2, …, HDL_ANGLE_FIRE_6 contienen los valores de los ángulos donde las cámaras tienen que realizar una captura. El número de estas variables, del 1 al 6, indica que cámara es la que se ha de activar. Inicialmente, para disparar las cámaras solo se verificaba mediante un if si la posición actual del LiDAR es mayor que el ángulo de disparo, y si la posición anterior del LiDAR es menor que el ángulo de disparo.

De esta forma se puede saber cuándo el Velodyne pasa por ese punto y generar la señal de disparo de la cámara correspondiente, utilizando el método ya explicado.

Las variables c_angulo_1, c_angulo_2, …, c_angulo_6 del código se han utilizado para contabilizar las veces que se genera una señal de disparo en cada cámara. Teniendo en cuenta que el archivo del LiDAR tiene una duración de 63,65 seg de muestras, y que la frecuencia de rotación del Velodyne en modo automático varía entre 10 y 12 Hz, se debería disparar cada cámara unas 700 veces aproximadamente, tomando 11Hz como referencia.

Se ha realizado una prueba disparando las 6 cámaras, en ángulos distintos, y enviando el archivo completo del Velodyne. En la Figura 59 se ve la configuración de cada ángulo de disparo y a la derecha los resultados de la prueba. El ángulo de partida en el archivo que se envía es 0x3EB1 (16.049 o 16,049 grados) y el ultimo 0x2979 (10.617 o 10,617 grados.).

| num_cam | 6 | char |
|---|---|---|
| HDL_ANGLE_FIRE_1 | 6000 | int |
| HDL_ANGLE_FIRE_2 | 9000 | int |
| HDL_ANGLE_FIRE_3 | 12000 | int |
| HDL_ANGLE_FIRE_4 | 24000 | int |
| HDL_ANGLE_FIRE_5 | 27000 | int |
| HDL_ANGLE_FIRE_6 | 30000 | int |

| c_angulo_1 | 716 | int |
|---|---|---|
| c_angulo_2 | 716 | int |
| c_angulo_3 | 715 | int |
| c_angulo_4 | 716 | int |
| c_angulo_5 | 716 | int |
| c_angulo_6 | 716 | int |
| c_rotaciones | 716 | int |

**Figura 59. Configuración de los ángulos de disparo y resultados.**

De los resultados se puede observar que el LiDAR realiza 716 rotaciones, un valor muy cercano a las 700 rotaciones esperadas. Además, todas las salidas generan un pulso de disparo por cada paso en sus ángulos críticos menos para el HDL_ANGLE_FIRE_3. Esto se debe a los ángulos de inicio y fin del archivo que se manda, ya que no son el mismo.

Tras realizar varias pruebas más variando los ángulos de disparo, se han encontrado puntos críticos en los que no se han generado todas las señales esperadas para disparar las cámaras, como en los ángulos de 0 y 360 grados. En estos puntos en lugar de 716 disparos se han generado entre 0 y unos 110, por lo que para ellos el programa no está funcionando correctamente.

Para solucionarlo se ha modificado el if del código para que en esos puntos no se produzcan errores. El aspecto del código final es el que se encuentra en el Anexo I. Se han añadido dos comparaciones más a la ya mencionada descritas a continuación.

La primera de ellas comprueba si la posición del LiDAR es mayor o igual al ángulo de disparo y, además, si se ha producido una rotación o paso por el ángulo de 0 grados en ese momento. Con estas líneas de código se solucionan los errores en los ángulos de disparo iguales o próximos a 0 grados, ya que, si el punto de disparo es menor que la posición actual del LiDAR, y este acaba de pasar por 0 grados, se debe a que hay un punto de disparo de las cámaras. La segunda, comprueba de nuevo si se ha realizado un paso por 0 grados, y si la posición anterior del Velodyne es menor que el ángulo de disparo correspondiente. Si esto se produce es porque hay un punto de disparo muy cercano a los 360 grados y mediante este método se puede detectar.

Como comprobación se ha llevado a cabo la misma prueba que antes, pero se han desplazado los ángulos de disparo hacia los puntos críticos y sus proximidades, como se muestra en la Figura 60. En esta ocasión los resultados si son correctos, ya que se han generado todas las señales de disparo esperadas.

| | | |
|---|---|---|
| num_cam | 6 | char |
| HDL_ANGLE_FIRE_1 | 0 | int |
| HDL_ANGLE_FIRE_2 | 2 | int |
| HDL_ANGLE_FIRE_3 | 12000 | int |
| HDL_ANGLE_FIRE_4 | 24000 | int |
| HDL_ANGLE_FIRE_5 | 35993 | int |
| HDL_ANGLE_FIRE_6 | 36000 | int |

| | | |
|---|---|---|
| c_angulo_1 | 716 | int |
| c_angulo_2 | 716 | int |
| c_angulo_3 | 715 | int |
| c_angulo_4 | 716 | int |
| c_angulo_5 | 716 | int |
| c_angulo_6 | 716 | int |
| c_rotaciones | 716 | int |

**Figura 60. Configuración de los ángulos de disparo y resultados para puntos críticos.**

Las variables c_angulo_x, donde x es la cámara correspondiente, solo indican si se puede generar la señal de disparo, pero no la generan. Para ello se utilizan las funciones, que en el código se ubican dentro del if, y procedimiento explicado en el bloque sobre la generación de las señales de disparo.

Hasta este momento se cuenta con la base del proyecto sobre la que se realizarán las pruebas de laboratorio y se propondrán soluciones sobre los posibles errores que se encuentren. Además, se añadirán pequeñas mejoras una vez se haya testeado su correcto funcionamiento.

## 4.4. Desarrollo de pruebas con el LiDAR en el laboratorio

En este apartado se van a desarrollar las pruebas realizadas en el laboratorio, así como la solución a posibles errores. Por último, se añadirán mejoras al funcionamiento del proyecto.

Una vez en el laboratorio, con la supervisión de mi tutor y cotutor se han realizado diferentes pruebas para testear el programa. En primer lugar, sin necesidad del LiDAR, se ha utilizado el osciloscopio para comprobar la configuración de las señales de disparo de las cámaras. Hasta este momento únicamente se disponía de una medida aproximada, pero para verificar un pulso a nivel bajo de 50 ms es necesaria esta herramienta.

Para esta prueba se vuelve a emplear la réplica del archivo .pcap del LiDAR, pero en lugar de ver si se encienden los LEDs y el número de disparos acumulados en las variables c_angulo_x, se conecta la sonda del osciloscopio para ver en su pantalla su forma y medidas de sus parámetros. En las primeras medidas con el osciloscopio las señales aparecían con mucho ruido. Al principio se pensó que podría ser a causa de la sonda, pero al cambiar la sonda y no conseguir subsanar el problema se buscaron otras opciones. Finalmente, el ruido era producido por el PC, ya que al utilizar una batería externa como fuente de alimentación para la tarjeta STM32 el ruido disminuyó considerablemente. A partir de este punto para todas las pruebas se ha utilizado la fuente de alimentación externa.

En la Figura 61 se muestra la señal de disparo para una de las cámaras. Esta señal tiene un pulso negativo de 50 ms y una frecuencia de 9,398 Hz que se corresponde con la frecuencia de rotación del Velodyne (aunque anteriormente se ha comentado que esta frecuencia varía entre 10 y 12 Hz, al ampliar la señal en la pantalla del osciloscopio la frecuencia disminuye). También se puede observar como la tensión de salida en los pines de la tarjeta STM32F767ZI es de 3,44 V.

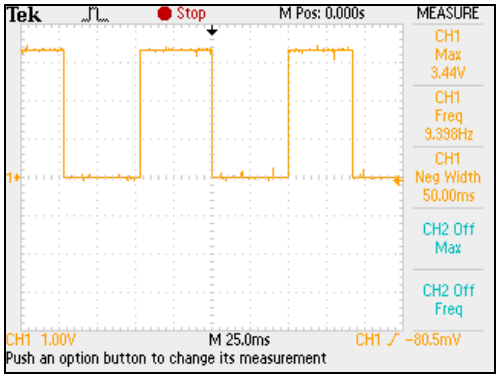



**Figura 61. Señal de disparo medida en el osciloscopio.**

A continuación, se han configurado los ángulos de disparo con incrementos de 60 grados entre ellos, y con ayuda de un segundo canal del osciloscopio, se ha evaluado la diferencia entre la primera señal, que será fija, y las demás en grupos de dos. Los ángulos de disparo elegidos son 0, 60, 120, 180, 240 y 300 grados.

Tomando como referencia la frecuencia de 9,398 Hz de la primera señal de disparo, se obtiene un periodo de 106,4 ms para esta señal. Una rotación completa del Velodyne consta de 360 grados que, al dividirlos entre los 60 grados de los incrementos, da como resultado un total de 6 desplazamientos de 60 grados sobre la señal inicial para realizar una vuelta completa.

Si ahora se divide el periodo entre 6 se obtiene el desplazamiento en tiempo cuando los ángulos de disparo tienen una diferencia de 60 grados. Al enfrentar la señal fija a las demás, se tiene que obtener un desfase teórico de 17,734 ms por cada 60 grados de diferencia entre ellas.

Se ha configurado el osciloscopio para utilizar dos señales simultáneamente y, además, con ayuda de los cursores de esta herramienta, se ha determinado la diferencia de tiempo entre dos puntos. En este caso, se han seleccionado los flancos de bajada de las señales, ya que marcan el inicio del pulso a nivel bajo. En la Figura 62 se puede observar fija la señal de disparo en un ángulo de 0 grados, color amarillo, enfrentada a dos señales, de color azul, cuyos ángulos de disparo son 60 y 120 grados respectivamente.

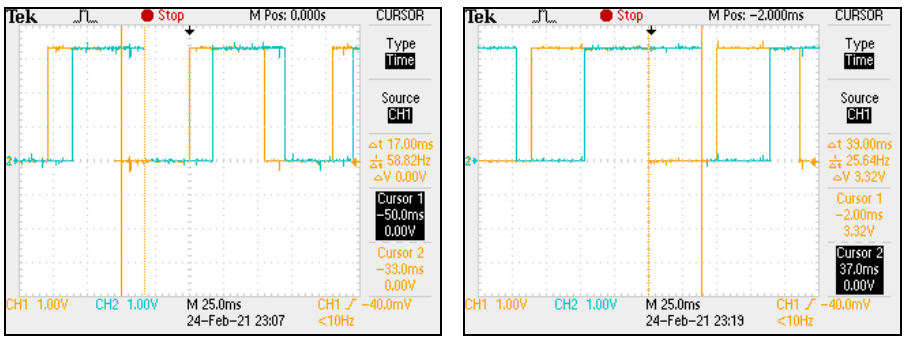



**Figura 62. Capturas de osciloscopio de las señales de disparo configuradas en distintos ángulos (i).**

En la imagen de la izquierda, donde existen 60 grados de desfase entre los ángulos de disparo (0 y 60 grados), se puede observar cómo hay una diferencia de tiempo de 17 ms entre ambos flancos de bajada. Esto se ajusta a los cálculos que se han realizado de forma teórica. Si ahora se observa la imagen de la derecha (0 y 120 grados), la diferencia de tiempo aumenta hasta 39 ms mientras que teóricamente debería ser de 35,468 ms. Aunque existe una diferencia de 3,5 ms entre los resultados, es un valor bastante cercano. Esta diferencia se debe a varios factores.

El primero de ellos es debido a que no se han tomado las capturas del osciloscopio en el mismo instante de tiempo, por lo que, si la frecuencia de rotación del LiDAR ha aumentado respecto a la primera captura, la diferencia de tiempo entre ellas será menor.

El segundo tiene relación con el primero, ya que los datos que recoge el LiDAR no siempre se corresponden con los mismos puntos en cada rotación, lo que quiere decir que, en ocasiones la señal de disparo para el ángulo configurado se inicia con una diferencia de microsegundos sobre su paso por ese punto en cada vuelta, lo que afecta a la frecuencia.

Por último, en el osciloscopio se obtiene la frecuencia para ese pequeño tramo de señal, lo que puede producir pequeñas variaciones en la frecuencia si se compara con la selección de esa misma señal durante un periodo de tiempo más grande, que se podría realizar cambiando la escala del osciloscopio, pero dificultaría el uso de los cursores.

En la Figura 63 se muestra la misma señal inicial fija, de color amarillo, y dos nuevas señales con desplazamientos de 180 y 240 grados, izquierda y derecha respectivamente. La imagen de la izquierda es una captura del osciloscopio de dos señales de disparo desfasadas 180 grados, como se puede observar a simple vista, ya que son opuestas. La diferencia entre ambos flancos de bajada es de 49 ms, mientras que teóricamente debería ser de unos 53,202 ms. Al igual que ocurría anteriormente, esta diferencia se produce por los factores ya mencionados.

En la imagen de la derecha las señales están desfasadas 240 grados y su diferencia entre flancos de bajada es de 68 ms, frente a los 70,936 teóricos.

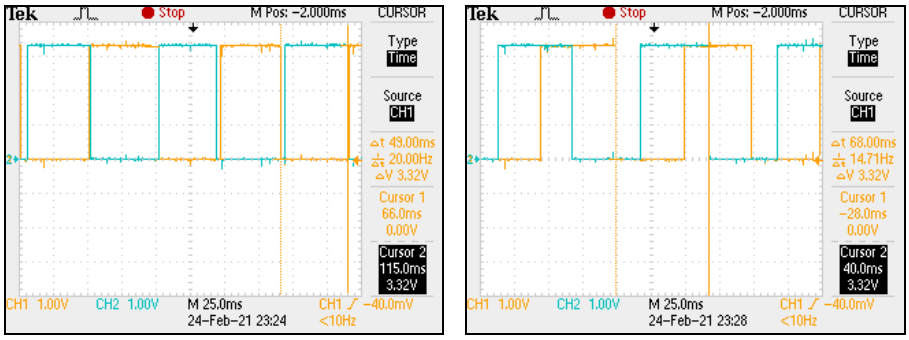



**Figura 63. Capturas de osciloscopio de las señales de disparo configuradas en distintos ángulos (ii).**

Por último, en la Figura 64 se puede observar la señal de disparo para un ángulo de 0 grados, color amarillo, comparada con la de un ángulo de disparo de 300 grados. Con la ayuda de los cursores se ha obtenido la diferencia de tiempo entre dos flancos de bajada, uno de cada señal. Este valor es de 88 ms, que es prácticamente igual a los 88,67 ms teóricos.

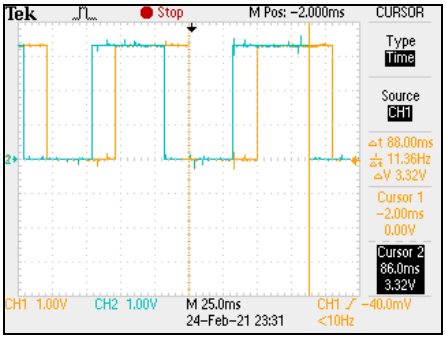



**Figura 64. Capturas de osciloscopio de las señales de disparo configuradas en distintos ángulos (iii).**

Durante este proceso de pruebas una de las señales, la correspondiente al Timer 9, aparecía en el osciloscopio con un pulso a nivel bajo de 25 ms en lugar de 50 ms. Como ya se ha comentado, se debía a que este Timer trabaja a una frecuencia distinta al pertenecer al bus APB2 (200 Hz frente a los 100 Hz de los otros Timers).

Para solucionarlo, en la Figura 30 aparece la configuración elegida para todos los Timers. En ella se puede ver como el prescaler del Timer 9 es el doble que el del resto, así se consigue compensar esa frecuencia mayor y se obtiene un mismo funcionamiento.

Como conclusiones sobre la prueba realizada se puede determinar que las señales de disparo tienen una forma de onda correcta, con un pulso a nivel bajo de 50 ms. Además, todas ellas se inician adecuadamente teniendo en cuenta la configuración de su ángulo de disparo. El desfase en el tiempo que se produce entre una señal y otra con un ángulo de disparo distinto es correcto, salvo pequeñas diferencias producidas por los factores ya comentados.

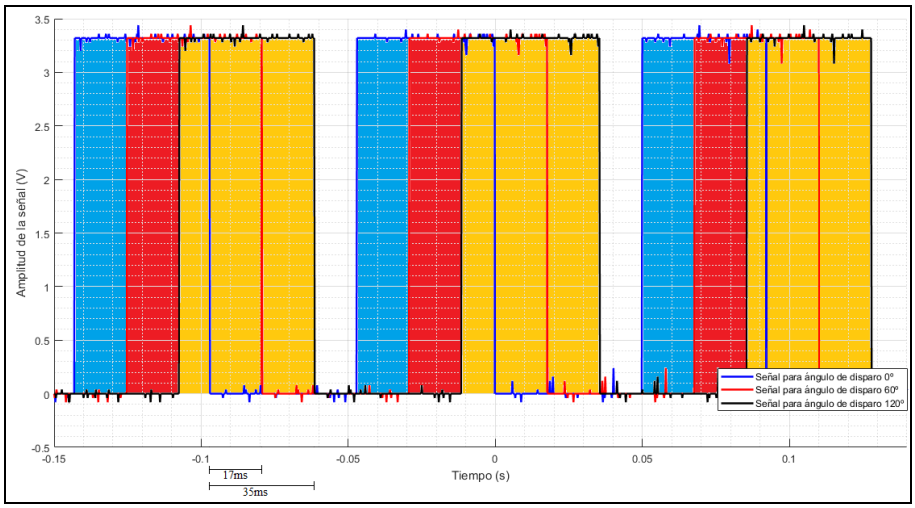



**Figura 65. Representación de 3 señales con ángulos de disparo de 0, 60 y 120 grados.**

En la Figura 65 se representan 3 señales de disparo configuradas con unos ángulos de disparo de 0, 60 y 120 grados. Para ello se ha utilizado el archivo Excel con los puntos de la señal de la Figura 61 proporcionados por el osciloscopio. Se ha empleado Matlab para realizar una representación de esos puntos, y se han replicado para generar las señales de 60 y 120 grados, pero desplazadas el tiempo correspondiente. Esta Figura tiene la finalidad de intentar mostrar de forma clara lo explicado anteriormente.

Con ayuda de la siguiente prueba se podrá determinar casi definitivamente si la señal de disparo se genera cuando el LiDAR se encuentra en el ángulo correcto. Para ello, en lugar de utilizar el archivo del LiDAR se va a emplear el propio Velodyne y las cámaras. Se han establecido las conexiones necesarias para poder realizar los test, como conectar el Ethernet del LiDAR hasta la tarjeta STM32, y las salidas de la tarjeta a las cámaras. Cuando el funcionamiento sea el deseado, se van a fabricar unos cables específicos para cada cámara. Se van a utilizar los conectores de disparo Hirose, de los que se cableará su entrada optoacoplada y su masa para sacarlos en un RJ45, que se fijará a la tarjeta STM32. Además, desde la tarjeta se van a soldar directamente los pines hasta el RJ45 para conseguir una instalación más segura y estable. En la Figura 66 se pueden ver los cables que se han fabricado para conectar las cámaras con los pines de la tarjeta. Los cables negro y morado que su muestran unidos se corresponde con la masa, mientras que el cable verde y el amarillo, son los puntos donde se conectarán las señales de disparo.

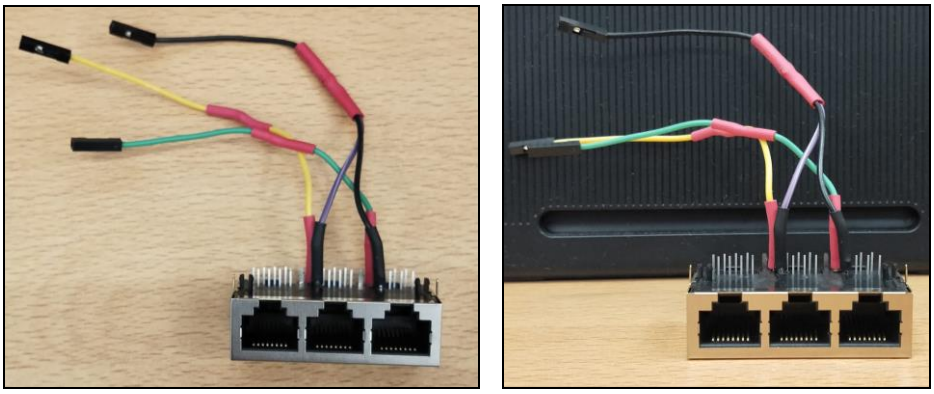


**Figura 66. Cables fabricados para cada cámara.**

En esta prueba se han configurado 4 ángulos de disparo con los siguientes grados: 0, 90, 180 y 270. El objetivo es comprobar si las cámaras se disparan en el ángulo correcto, para ello se ha de conocer en qué posición se encuentra el ángulo 0 en el LiDAR y el sentido en el que rota. En las hojas de características del Velodyne se especifica que la posición de 0 grados se encuentra 90 grados a la izquierda del cable del LiDAR, visto desde arriba, y la posición de 90 grados se sitúa sobre el cable. En la Figura 67 se muestra esta información, en la que también se incluye la dirección de rotación del Velodyne.

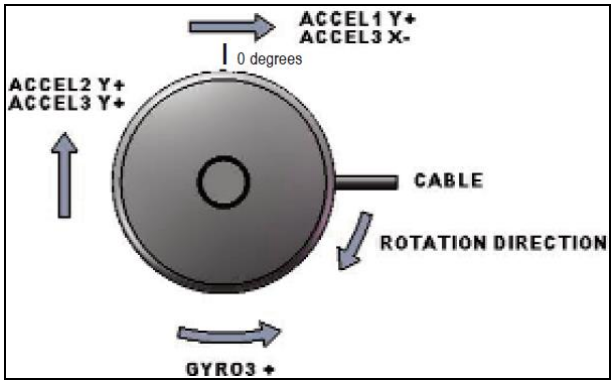



**Figura 67. Posición ángulo 0 LiDAR. [2]**

Para comenzar con la prueba se ha iniciado el Velodyne para que comience a enviar datos. La tarjeta STM32 extrae la información útil y activa las cámaras correspondientes. Mediante un programa del que mi tutor y cotutor disponían se pueden observar en tiempo real las capturas de las cámaras, de forma que al enfrentarlas al LiDAR se puede apreciar el punto donde están realizando la captura.

Como solo se dispone de dos cámaras, para la primera prueba se han seleccionado los ángulos de 0 y 90 grados. Para saber en qué posición del LiDAR disparan las cámaras se ha utilizado la pantalla en tiempo real. El LiDAR tiene una pequeña abertura por la que realiza las medidas, de forma que al enfrentar las cámaras al Velodyne en el ángulo donde se disparan, se puede observar esta abertura en la pantalla. Si, por el contrario, la cámara no se encuentra en la posición correcta, este orificio no aparece en la pantalla.

Al iniciar todo el proceso en la pantalla se ha observado el funcionamiento esperado del sistema, situándose las capturas de las cámaras en las posiciones adecuadas. Con el resto de parejas de ángulos seleccionadas el resultado ha sido el mismo.

Con el fin de ser más precisos en las pruebas, se han seleccionado los ángulos de disparo de 90 y 270 grados, de forma que sean opuestos. El objetivo es situar las cámaras perfectamente enfrentadas, colocando el LiDAR entre ellas con las posiciones de 90 y 270 grados apuntando a las cámaras. Con esta prueba, se quiere determinar si las cámaras se disparan en el centro del orificio del Velodyne, o si es un punto desplazado. En la Figura 68 se muestra esta prueba.

**Figura 68. Prueba real en el laboratorio con ángulos de disparo enfrentados.**

Al observar en la pantalla en tiempo real, se ve como, aunque ambas cámaras realizan sus capturas en un ángulo correcto, se produce un pequeño desplazamiento causado por el tiempo que se tarda en procesar la información y realizar el disparo. En un primer momento, propuse incluir un offset en el valor de posición que envía el Velodyne para anticipar la siguiente localización del LiDAR, y así centrar el punto de disparo, pero otro de los problemas que pueden surgir en el funcionamiento real del sistema sobre el vehículo, es la variación de la velocidad de rotación del LiDAR por el efecto de las condiciones meteorológicas u otros factores externos. Por ejemplo, al adquirir velocidad el vehículo, la fuerza de rozamiento del viento es mayor que si se compara con el vehículo parado o en ausencia de estas condiciones. De forma que si se incluye este offset y la velocidad de rotación se ve alterada, el disparo se descentraría de nuevo.

Con esta información, la solución que se ha llevado a cabo consiste en prever la próxima posición del LiDAR, teniendo en cuenta los datos disponibles en tiempo real, y así adelantar el disparo de las cámaras para que se produzca en el centro. Para ello, se calcula la diferencia entre la posición actual y la posición anterior del Velodyne, lo que cuantifica su avance en tiempo real. Este valor se almacena en un array junto con las 4 últimas diferencias calculadas, y se realiza la media.

El resultado final, es el offset en tiempo real que se utiliza para conocer la próxima posición del LiDAR. Además, soluciona el problema de la variación de la velocidad de rotación ya que, al utilizar las posiciones en tiempo real, si la velocidad baja también lo hará el valor calculado. En la Figura 69 se observa el array que se ha creado y la media resultante.

| ultimas_medidas | 0x200000A4 ultimas_... | int[5] |
|---|---|---|
| [0] | 18 | int |
| [1] | 18 | int |
| [2] | 19 | int |
| [3] | 19 | int |
| [4] | 19 | int |
| media_medidas | 18.6000004 | float |

**Figura 69. Array de diferencias entre la posición actual y anterior del LiDAR.**

Una vez se ha incluido esta mejora, se realiza de nuevo la misma prueba y se observa cómo la captura de las cámaras aparece más centrada. Durante esta prueba, se ha realizado una grabación de las capturas, y se han almacenado los datos generados por las cámaras, que se utilizarán para comprobar definitivamente el funcionamiento del sistema.

En primer lugar, se han inspeccionado los vídeos de ambas cámaras, fotograma a fotograma, con el objetivo de comprobar que en todas las capturas el LiDAR está enfocado hacia la cámara y así determinar que los disparos se realizan en el ángulo correcto. En la Figura 70 se muestran dos fotogramas extraídos de los videos de cada cámara para 90 y 270 grados. En estos fotogramas, aunque aparece un indicador de 0 y 180 grados, realmente se encuentran en las posiciones indicadas anteriormente. En ellos se puede apreciar la abertura del LiDAR, que indica que en el momento de la captura el Velodyne se encuentra posicionado frente a la cámara.


**Figura 70. Fotogramas de los vídeos de las cámaras para 90 y 270 grados.**

Después, se han analizado los datos de los vídeos de ambas cámaras. De los vídeos se generan 3 columnas de datos:

- Valor de tiempo en segundos de cada captura de la cámara. Este dato proviene de un reloj interno de la cámara que se desborda cada 128 segundos. Únicamente sirve para comparar con ella misma (tiempo entre capturas), ya que no es una referencia global. Este dato se ha omitido porque no sirve para comparar entre ambas cámaras.
- Hora UTC de cada captura realizada en formato hora:minutos:segundos:milisegundos. Esta referencia es global por lo que puede utilizarse para comparar entre cámaras, y con el Velodyne.

- Recoge la misma información que la columna anterior, pero en microsegundos desde una fecha determinada. También es una referencia global por lo que se puede usar para comparar entre cámaras y con el LiDAR. Estos datos son los que se van a utilizar para extraer la información que se necesita.

A partir de estos datos, lo primero que se ha comprobado es el tiempo entre capturas de cada cámara. Este tiempo tiene que ser constante siempre que se mantenga la frecuencia de rotación estable, como ocurre en este caso.

Además, sirve para confirmar que se realiza un disparo cada vez que el Velodyne se encuentra en la posición correspondiente, y no se producen perdidas de capturas. En la Figura 71 se observan las 3 columnas de datos generados por las cámaras.

| | | |
|---|---|---|
| 94.918274 | 18:00:16:034 | 64816034482 |
| 95.004001 | 18:00:16:120 | 64816120199 |
| 95.089138 | 18:00:16:205 | 64816205326 |
| 95.174277 | 18:00:16:290 | 64816290455 |
| 95.259415 | 18:00:16:375 | 64816375646 |
| 95.344629 | 18:00:16:460 | 64816460775 |
| 95.430280 | 18:00:16:546 | 64816546391 |
| 95.515417 | 18:00:16:631 | 64816631810 |
| 95.600632 | 18:00:16:716 | 64816716797 |

| | | |
|---|---|---|
| 95.600632 | 18:00:16:716 | 64816716797 |
| 95.685769 | 18:00:16:801 | 64816801985 |
| 95.770907 | 18:00:16:887 | 64816887099 |
| 95.856635 | 18:00:16:972 | 64816972869 |
| 95.941774 | 18:00:17:057 | 64817057954 |
| 96.026910 | 18:00:17:143 | 64817143167 |
| 96.112126 | 18:00:17:228 | 64817228302 |
| 96.197263 | 18:00:17:313 | 64817313414 |
| 96.282913 | 18:00:17:399 | 64817399270 |

**Figura 71. Datos generados por una de las cámaras.**

El primer análisis se ha llevado a cabo teniendo en cuenta los datos de la primera cámara, calculando el tiempo entre cada una de las capturas que ha realizado. Los resultados se muestran gráficamente en la Figura 72, que se ha generado mediante Matlab. En la imagen de la izquierda, se representa el tiempo entre capturas consecutivas de todas las muestras disponibles, obteniéndose una media de tiempo de 85,043 ms. Además, ninguna diferencia entre capturas es superior a 86,49 ms o inferior a 83,752 ms, lo que indica que se realizan todos los disparos esperados y no se produce pérdida de fotogramas. En la imagen de la derecha se representan los mismos datos, pero con un rango menor de muestras.

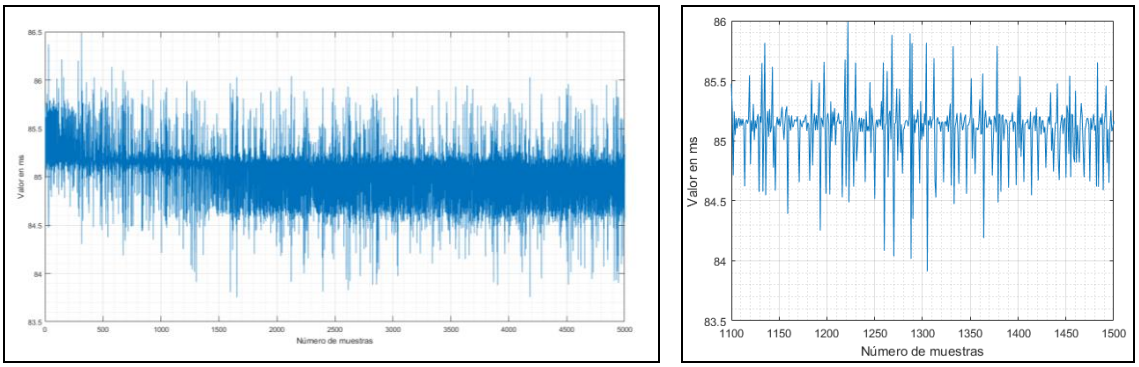



**Figura 72. Representación del tiempo entre capturas para la cámara 1.**

Se ha llevado a cabo el mismo procedimiento con los datos de la segunda cámara, de los cuales se han obtenido unos resultados muy similares a los anteriores. En la Figura 73 se muestran los resultados gráficamente. En la imagen de la izquierda se representan el total de las muestras, con unos valores que no superan los 85,841 ms ni son inferiores a 84,375 ms, obteniéndose las mismas conclusiones que para el caso anterior. La media de las diferencias entre capturas es de 85,043 ms, igual que para la cámara 1. La frecuencia de rotación real se puede calcular a partir de esta media de 85,043 ms, dando un resultado de 11,75 Hz.

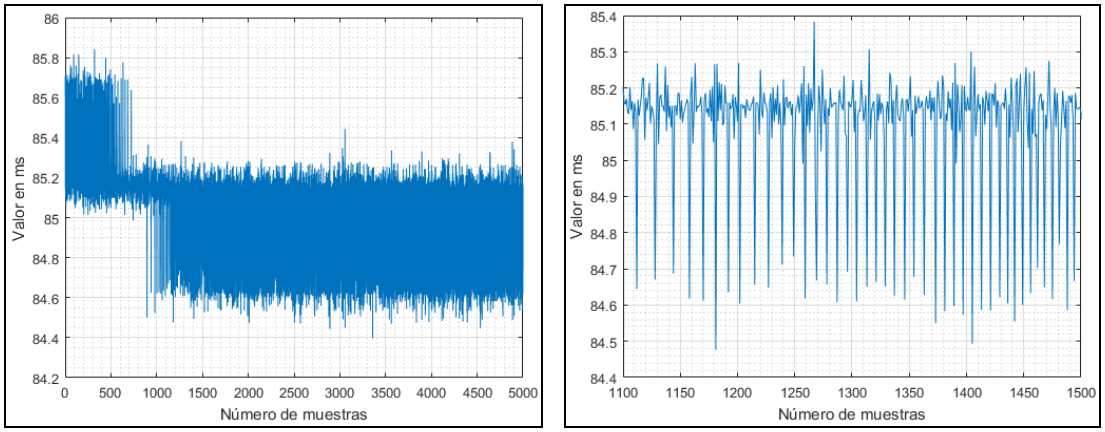



**Figura 73. Representación del tiempo entre capturas para la cámara 2.**

Durante la generación de los datos, las cámaras están posicionadas en los ángulos de 90 y 270 grados tomando como referencia el LiDAR, de forma que se encuentran enfrentadas. En este escenario la diferencia entre las capturas de ambas cámaras debe de ser constante, por lo que también se va a analizar. Como se ha comprobado, la diferencia entre las capturas de una misma cámara es de 85,043 ms, de forma que al comparar entre las dos ha de ser la mitad, unos 42,5 ms.

Las referencias de los tiempos que entregan las cámaras son globales, lo que permite analizar una cámara frente a la otra e incluso con el Velodyne. En la Figura 74, se puede ver los resultados obtenidos. En la imagen de la izquierda se aprecia que la diferencia entre los disparos de cada cámara es de 42,46 ms de media, lo que se ajusta a lo esperado. Además, sus valores más extremos so de 41,208 y 43,212 ms, pudiendo afirmar que las cámaras están correctamente sincronizadas. En la imagen de la derecha, se muestra un fragmento de los datos para facilitar su visualización.

Con esta última prueba, se comprueba que tanto el LiDAR como las cámaras están correctamente sincronizados, consiguiendo el funcionamiento buscado del sistema. Además, las cámaras se disparan siempre que se requiera, sin perder ninguna captura en el tiempo. La tarjeta STM32 procesa adecuadamente todos los datos y maneja de forma correcta todo el sistema.

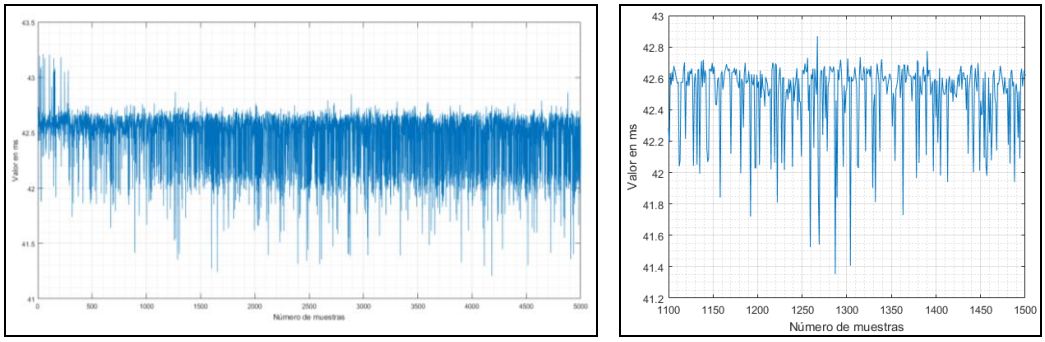



**Figura 74. Representación del tiempo entre capturas entre ambas cámaras.**

A continuación, se han grabado varios vídeos más desde las cámaras, en los que se ve a partir de los datos, cómo la frecuencia de rotación sufre pequeñas variaciones en cada uno de ellos, siempre entre 10 y 12 Hz. Estas variaciones no producen modificaciones en el funcionamiento del sistema. En el Anexo III se incluye una configuración.

Uno de los problemas que se ha encontrado para manejar el sistema es el cambio en los ángulos de disparo y el número de cámaras habilitadas, ya que se ha de modificar el código C. Para evitar este proceso y poder actualizar la configuración con el sistema en funcionamiento, se me propuso conseguir enviar un paquete de configuración a través de Ethernet.

## 4.5. Paquete de configuración del sistema a través de Ethernet

En este capítulo se va a tratar cómo se ha procedido para modificar y actualizar, en tiempo real, los ángulos de disparo de cada cámara y el número de cámaras activadas, sin tener que parar el sistema y modificar sus valores en el código.

Para ello, se me propuso emplear un paquete UDP con la información necesaria, para después procesarlo con la tarjeta STM32 de igual modo que con los datos del Velodyne, y actualizar el valor de las variables que contienen el ángulo de disparo.

El primer problema que se ha encontrado es cómo indicar a la tarjeta que se va a mandar un paquete de configuración. Esta acción es necesaria porque en el programa se están filtrando los paquetes, para únicamente prestar atención a los enviados por el LiDAR. Para ello se ha empleado el pulsador de la tarjeta STM32 y su interrupción, previamente testeados en los primeros pasos dados con STM32CubeMX.

La funcionalidad que se ha buscado es la siguiente: cada vez que se active el pulsador azul de la tarjeta se activará su interrupción, en la cual se modifica el modo de recepción de paquetes del bucle principal mediante un flag. Además, en esta interrupción se accionará el Timer 7, que se emplea como un contador de 30 segundos en los que, si no se recibe ningún paquete de configuración se vuelve al modo de funcionamiento normal del programa, modificando de nuevo el flag ya mencionado.

Para ello, a través de STM32CubeMX se va a realizar un nuevo proyecto que después se integrará en el código final, con la configuración del pulsador, su interrupción y de un último Timer. El pulsador se encuentra en el pin PC13, como se puede observar en la información de la Figura 75 extraída del esquemático de la tarjeta.

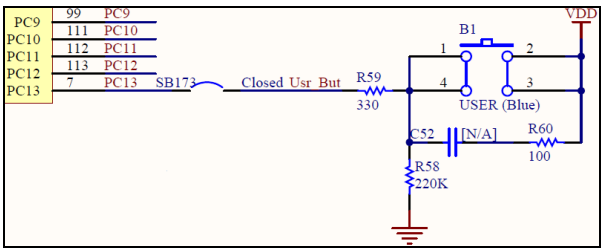



**Figura 75. Circuito eléctrico del pulsador extraído del esquemático. [1]**

Desde esta herramienta se configura gráficamente el pin PC13 como GPIO_EXTI13, es decir como interrupción externa. Desde la propia herramienta, en la Figura 76, se puede apreciar que se ha seleccionado el modo de activación de flancos ascendentes en dicho pin.

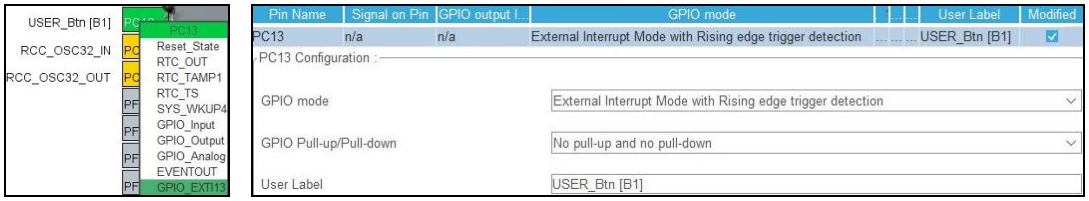



**Figura 76. Configuración del pin PC13 como GPIO_EXTI en STM32CubeMX.**

Además de esta configuración, es necesario habilitar la interrupción para que cuando se produzca el evento que la accione se pueda activar. En la Figura 77 aparece como se ha habilitado desde STM32CubeMX.


**Figura 77. Habilitación de la interrupción EXTI del pin PC13 en STM32CubeMX.**

Por último, queda activar y configurar el Timer 7 que pertenece al bus APB1, por lo que su frecuencia es de 100 MHz. Se quiere que funcione como un contador de 1 segundo, para ello su configuración se muestra en la Figura 78. En esta ocasión, a diferencia del resto de Timers no se va a emplear el modo one pulse mode.

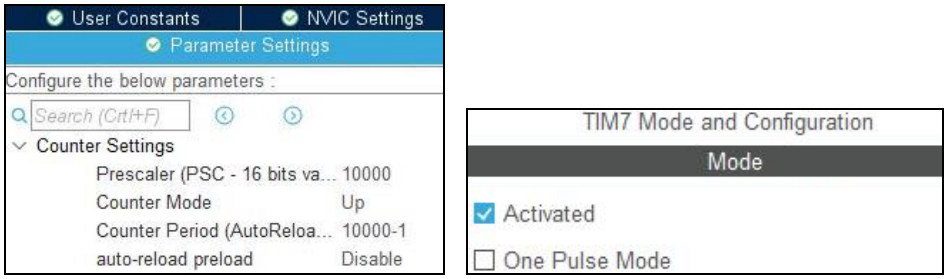


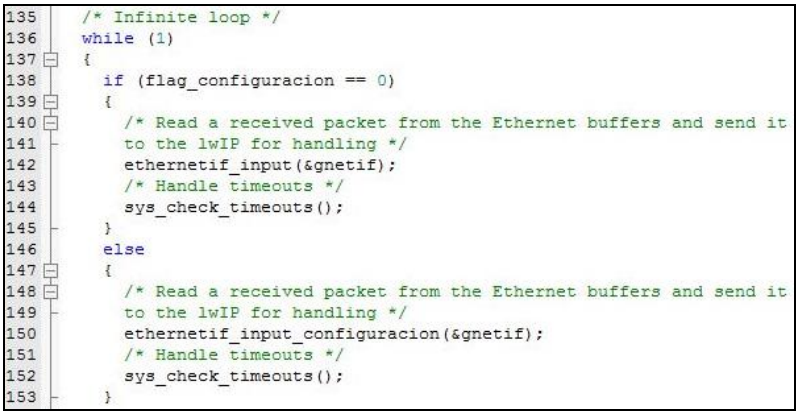

**Figura 78. Activación y configuración del Timer 7 en STM32CubeMX.**

Una vez se han realizado todas las configuraciones se genera el código y se añade al programa principal. A continuación, se ha de completar la rutina de atención a la interrupción del pulsador, en ella se debe activar el Timer 7, limpiar el flag de interrupción y cambiar el flag de configuración que modifica el modo de funcionamiento del sistema. En la Figura 79 se observa esta parte del código.

```
222 /**
223   * @brief This function handles EXTI line[15:10] interrupts.
224   */
225 void EXTI15_10_IRQHandler(void)
226 {
227   HAL_TIM_Base_Start_IT(&htim7);
228   HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
229   flag_configuracion = 1;
230 }
```

**Figura 79. Rutina de atención a la interrupción del pulsador.**

La modificación en la variable ‘flag_configuracion’ produce un cambio en el modo de funcionamiento del sistema. Como se observa en la Figura 80, al cambiar el valor de esta variable no se accede a la función ethernetif_input () como en el resto de ocasiones, sino que se llama a una función similar, ethernetif_input_configuracion (), que se ha creado para detectar y extraer la información de un posible paquete de configuración. A cambio, no se leen los paquetes enviados por el LiDAR durante este modo de funcionamiento, de 30 segundos de duración o hasta la recepción de un paquete de configuración.

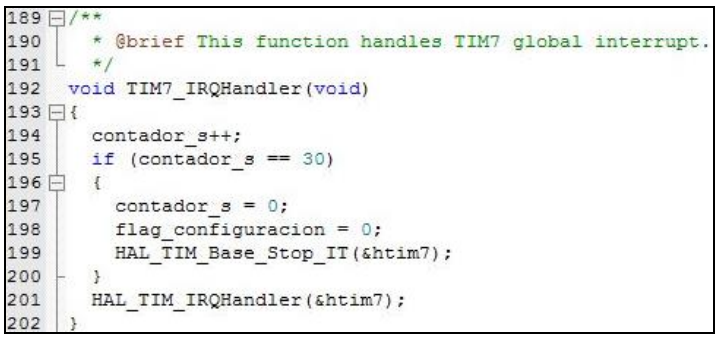


```
135   /* Infinite loop */
136   while (1)
137   {
138     if (flag_configuracion == 0)
139     {
140       /* Read a received packet from the Ethernet buffers and send it
141       to the lwIP for handling */
142       ethernetif_input(&gnetif);
143       /* Handle timeouts */
144       sys_check_timeouts();
145     }
146     else
147     {
148       /* Read a received packet from the Ethernet buffers and send it
149       to the lwIP for handling */
150       ethernetif_input_configuracion(&gnetif);
151       /* Handle timeouts */
152       sys_check_timeouts();
153     }
```

**Figura 80. Bucle infinito modificado del programa principal.**

Para conseguir un tiempo de espera de 30 segundos se utiliza el Timer 7 configurado previamente. Al iniciarse interrumpe con una frecuencia de 1 Hz, es decir cada segundo. En su rutina de interrupción se utiliza la variable ‘contador_s’ como un contador que se incrementa en cada interrupción y se limpiar el flag de interrupción.

Si el valor de la cuenta llega a 30 se reinicia el contador, se vuelve a modificar la variable ‘flag_configuracion’ que cambia de nuevo el funcionamiento principal del sistema, estableciendo su modo habitual, y por último se para la cuenta del Timer 7. En la Figura 81 aparece esta parte del código.



```
189 /**
190   * @brief This function handles TIM7 global interrupt.
191   */
192 void TIM7_IRQHandler(void)
193 {
194   contador_s++;
195   if (contador_s == 30)
196   {
197     contador_s = 0;
198     flag_configuracion = 0;
199     HAL_TIM_Base_Stop_IT(&htim7);
200   }
201   HAL_TIM_IRQHandler(&htim7);
202 }
```

**Figura 81. Código de la rutina de atención a la interrupción del Timer 7.**

Como se observa en la Figura 80, durante el tiempo en el que la variable ‘flag_configuracion’ permanece con un valor distinto de 0, no se accede a la función que lee los paquetes enviados por el LiDAR. En este caso se accede a la función ethernetif_input_configuracion ().

Esta función tiene una estructura muy similar a ethernetif_input (), ya que también va a leer los paquetes recibidos, pero en lugar de filtrar los enviados por el Velodyne, va a filtrar los paquetes destinados a la configuración del sistema.

Dado que los paquetes de configuración que se van a mandar también utilizan el protocolo UDP, poseen las mismas características que los enviados por el LiDAR, entre ellas las posiciones de las IP, puertos y la carga útil de datos. Para filtrar estos paquetes no se va a utilizar la IP, ya que podrían mandarse desde dispositivos con IP diferentes, sino que se empleará el puerto de origen y destino. En este caso el puerto escogido es el 51103, tanto como de origen como destino. En la Figura 82 se muestra el código de dicha función, en ellas se comprueba si los dos puertos tienen el valor mencionado anteriormente, 0xC79F (51103).

Si el paquete pasa el filtro, toda la información que se encuentra a partir de la posición 42 se clasifica en una estructura que se ha creado previamente. Después, mediante el uso de punteros se envían estos datos como argumento en una función que se encarga de comprobar la validez de los mismos, y en caso afirmativo actualizar las variables correspondientes.

Además, se reinicia el valor de la variable 'contador_s', se vuelve a cambiar al modo de funcionamiento principal del sistema y se para el Timer 7 ya que, aunque no sea un paquete de configuración útil, se determina que el paquete de configuración ha llegado y se pasa al funcionamiento habitual del sistema. En caso de querer cambiar la configuración de nuevo se ha de repetir el mismo proceso.

```
820  void ethernetif_input_configuracion(struct netif *netif)
821  {
822    err_t err;
823    struct pbuf *p;
824
825    /* move received packet into a new pbuf */
826    p = low_level_input(netif);
827
828    /* no packet could be read, silently ignore this */
829    if (p == NULL)
830      return;
831    /* verificacion de paquete valido */
832    else
833    {
834      comprobacion = (char*)p->payload;
835      k=0;
836      for(i=26;i<38;i++)
837        puntero[k++] = *(comprobacion+i);
838      if((puntero[8]==(char)(0xC7))&&(puntero[9]==(char)(0x9F))
839        &&(puntero[10]==(char)(0xC7))&&(puntero[11]==(char)(0x9F)))
840        {
841          Comprobar_Configuracion (((struct ConfigPacket *)(comprobacion+42)));
842          contador_s = 0;
843          flag_configuracion = 0;
844          HAL_TIM_Base_Stop_IT(&htim7);
845        }
846    }
```

**Figura 82. Código de la función ethernetif_input_configuracion ().**

Antes de continuar comentando el resto del código se va a explicar el formato del paquete de configuración y el método empleado para enviarlo. Dada la necesidad de enviar paquetes UDP desde un dispositivo como un ordenador a través de Ethernet, se va a emplear la herramienta Packet Sender, que permite enviar paquetes IP tanto UDP como TCP.

Mediante esta herramienta se va a escribir el texto de configuración que se va a mandar a la tarjeta STM32. El formato del paquete que se ha escogido es el mostrado en la Figura 83. En él se ha de seleccionar el número de cámaras, entre 1 y 6, y el valor de los ángulos de disparo para cada una de ellas. Si el número de ángulos es mayor que el de cámaras, los ángulos sobrantes se desecharán, mientras que si el número de ángulos es menor que el de cámaras el paquete no será válido.

Como se ve en la Figura 83 en Packet Sender se ha de seleccionar la dirección IP de destino, en este caso la de la tarjeta STM32, el puerto escogido, 51103, y por último el protocolo UDP.


**Figura 83. Ejemplo de envío de paquete de configuración a través de Packet Sender.**

Se ha probado a enviar un paquete de configuración y ver su contenido utilizando Wireshark. El resultado obtenido es el que se muestra en la Figura 84. El contenido se envía de forma correcta, manteniendo los datos y su información en las posiciones esperadas.

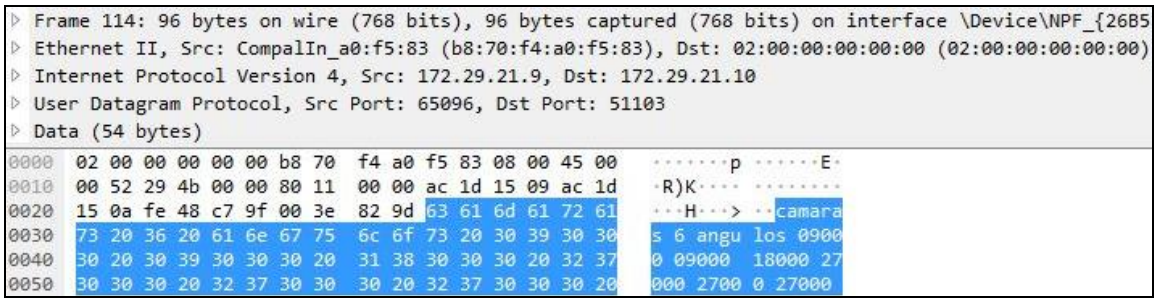



**Figura 84. Análisis del paquete de configuración mediante Wireshark.**

Conocido el formato del paquete de configuración se ha creado la estructura de la Figura 85. En esta estructura se busca que los espacios se encuentren en las variables 'blankx' y los valores de los ángulos se encuentren entre unos límites válidos. Si esto no ocurre el paquete no se da por bueno y no se actualizan las variables. La función encargada de realizar esta gestión es Comprobar_Configuracion (), cuyo código se encuentra en el Anexo II.

```
59  struct ConfigPacket {
60      unsigned char dipositivo1[7];
61      unsigned char blank1;
62      unsigned char num_camaras;
63      unsigned char blank2;
64      unsigned char dipositivo2[7];
65      unsigned char blank3;
66      unsigned char angulo1[5];
67      unsigned char blank4;
68      unsigned char angulo2[5];
69      unsigned char blank5;
70      unsigned char angulo3[5];
71      unsigned char blank6;
72      unsigned char angulo4[5];
73      unsigned char blank7;
74      unsigned char angulo5[5];
75      unsigned char blank8;
76      unsigned char angulo6[5];
77  };
```

**Figura 85. Estructura para el paquete de configuración.**

Tras la realización de varias pruebas en casa y comprobar que las modificaciones de los ángulos y el número de cámaras funciona correctamente, se ha llevado el proyecto al laboratorio. Allí se ha probado el funcionamiento completo del sistema modificando los ángulos de disparo de las cámaras. En la grabación de las mismas se puede observar cómo el hueco del LiDAR se encuentra en diferentes posiciones según hayan variado los ángulos de disparo.

Esta es la última prueba realizada del sistema, dándose por finalizado el TFG una vez se concluya este documento.

# 5. Conclusiones

El desarrollo del presente TFG ha permitido la implementación de un sistema de sincronización entre el paso de un LiDAR por distintos ángulos y el disparo de unas cámaras, todo ello para formar parte de un proyecto de un vehículo autónomo.

El método realizado extrae la información del Velodyne integrado en el vehículo y la utiliza para generar la señal de disparo que activa las cámaras. Todo el sistema está controlado mediante el microcontrolador de la tarjeta STM32.

En este trabajo se ha aportado el uso de Timers hardware, frente a los contadores software que se usan en otros proyectos, y una mayor flexibilidad a la hora de seleccionar el número de cámaras para el sistema.

Durante el desarrollo del proyecto se han comentado las decisiones que se han tomado mediante diferentes análisis. También se ha justificado el correcto funcionamiento de todas las partes del proyecto y el sistema global realizando varias pruebas para comprobar su rendimiento.

El estudio continuo sobre los vehículos autónomos y sus sistemas hace que estos estén en continua evolución. Por ello la tecnología utilizada en este TFG se tendrá que actualizar en relación a los nuevos avances. Gracias a esto, los datos recogidos y el funcionamiento general del sistema mejorará con el paso del tiempo.

Este sistema se puede ampliar utilizando más de un LiDAR para captar con mayor amplitud y precisión el entorno que rodea al vehículo. El problema es que se tendrían que sincronizar ambos láseres y todos los datos que recogen, siendo una labor más tediosa frente al uso de un solo Velodyne.

También se pueden incorporar otros sensores y utilizar el GPS integrado del que dispone el LiDAR, además de realizar variaciones entre los diferentes tipos de disparo presentes en cada modelo de cámara, e incluso crear una interfaz web que permita cambiar los ángulos de disparo de las cámaras a distancia.

Por último, existe la posibilidad de realizar estudios sobre la inclusión del software de adquisición de las cámaras en una tarjeta de este estilo, teniendo todo el sistema unificado.

# 6. Bibliografía


[1] FLIR Systems, «flir.es,» FLIR Systems, Inc, 2020. https://www.flir.es/.

[2] Velodyne LiDAR, Inc, «User's manual and programming guide,» Velodyne LiDAR, Morgan Hill, CA, 2012.

[3] STMicroelectronics, «UM1713 User Manual Developing applications on STM32Cube with LwIP TCP/IP stack,» 2015.

[4] FLIR Integrated Imaging Solutions Inc, «Grasshopper3 U3 Technical Reference,» 2017.

[5] STMicroelectronics, «UM1974 User Manual STM32 Nucleo-144 boards,» 2017.

[6] H. G. Norbye, «Camera-LiDAR sensor fusion in real time,» 2019.

[7] M. H. T. L. a. H. W. S. Schneider, «Fusing vision and LiDAR - Synchronization, correction and occlusion reasoning,» 2010.

[8] https://www.highmotor.com/imagen-muestra-que-conduccion-autonoma-requiere-tantos-sensores.html

[9] M. V. A. L. B. S. F. D. G. S. a. W. B. D. Cattaneo, «CMRNet: Camera to LiDAR-Map Registration,» 2019.

[10] https://thetechshow.tv/2019/03/22/gates-backed-lumotive-upends-lidar-conventions-using-metamaterials/

[11] https://www.youtube.com/watch?v=YCBQT0xSGdI

[12] https://www.youtube.com/watch?v=jGYQBLauvQo

[13] https://www.youtube.com/watch?v=BysPueIBPL0

[14] https://www.youtube.com/watch?v=nJkRcbdZrSk

[15] https://www.youtube.com/watch?v=BJdXR0Al6os

[16] https://www.youtube.com/watch?v=Kptv3jVr1II

[17] http://elb105.com/stm32f4-primeros-pasos-con-el-entorno-de-desarrollo/

[18] https://www.youtube.com/watch?v=uCc9VGE2VVM

[19] https://github.com/lsgunth/lwip_contrib/tree/master/ports/stm32f2x7/netif

[20] http://www.nongnu.org/lwip/2_0_x/structpbuf.html

[21] https://folk.ntnu.no/edmundfo/msc2019-2020/norbye-LiDAR-camera-reduced.pdf

[22] https://ieeexplore.ieee.org/document/5548079

[23] https://www.youtube.com/watch?v=NXGjhOa5oR8

[24] http://elb105.com/stm32f4-configurar-el-pwm-con-timers/

[25] https://www.youtube.com/watch?v=EX7g3_NUDgk

# ANEXO I

```
/* @brief Esta función busca en los 12 bloques de cada paquete valido que llega
      * los ángulos en los que las cámaras deben de dispararse. También indica cuando se
      * ha realizado una rotación completa. Lanza el disparo de las cámaras a través del
      * Timer 3 en modo interrupción.
      *
      * @param se utiliza la estructura creada para los paquetes UDP que llegan.
      *
      */

void Comprobar_Rotacional (struct HDLDataPacket* m_DATAPacket)
{
   for (int i = 0; i < HDL_FIRING_PER_PKT; ++i)
      {
         struct HDLFiringData firingData = m_DATAPacket->firingData[i];

         //Calcular la diferencia entre dos disparos consecutivos.
         if (firingData.rotationalPosition < m_lastRotationalPosition)
            aux_dif_rotationalPosition = (36000-m_lastRotationalPosition) + firingData.rotationalPosition;
         else
            aux_dif_rotationalPosition = (firingData.rotationalPosition - m_lastRotationalPosition);

         //Aquí se calcula la media de las ultimas 5 diferencias entre dos disparos consecutivos.
         ultimas_medidas[medida] = aux_dif_rotationalPosition;
         media_medidas = ((float) (ultimas_medidas [0]+ultimas_medidas [1]+ultimas_medidas [2]+ultimas_medidas
         [3]+ultimas_medidas [4]) /5);
         medida++;

         if (medida == 5)
            medida = 0;

         //Variables que muestran el disparo actual y el próximo disparo esperado.
         disparo_actualizado = firingData.rotationalPosition + (int)media_medidas;
         disparo_actual = firingData.rotationalPosition;

         //Se ajusta el próximo valor de disparo a un rango entre 0 y 35999
         if (disparo_actualizado > 35999)
            disparo_actualizado -= 35999;

         //Evita primer falso disparo y cuenta el número de rotaciones completas.
         if ((disparo_actualizado < m_lastRotationalPosition_actualizada) && (flag_start == 1))          // Rotación completa
         {
            flag_rotacion = 1;
            c_rotaciones++;
         }
         else
         {
            flag_rotacion = 0;
            flag_start = 1;
         }

         if ((((disparo_actualizado >= HDL_ANGLE_FIRE_1) && (m_lastRotationalPosition_actualizada < HDL_ANGLE_FIRE_1)) ||
          ((disparo_actualizado >= HDL_ANGLE_FIRE_1) &&(flag_rotacion==1)) ||((flag_rotacion==1) &&
          (m_lastRotationalPosition_actualizada < HDL_ANGLE_FIRE_1))) && (1 <= num_cam))
         {
            HAL_GPIO_WritePin (GPIOE, Output_0_Pin, GPIO_PIN_RESET);
            HAL_TIM_Base_Start_IT(&htim2);
            c_angulo_1++;
         }

         if ((((disparo_actualizado >= HDL_ANGLE_FIRE_2) && (m_lastRotationalPosition_actualizada < HDL_ANGLE_FIRE_2)) ||
          ((disparo_actualizado >= HDL_ANGLE_FIRE_2) &&(flag_rotacion==1)) ||((flag_rotacion==1) &&
          (m_lastRotationalPosition_actualizada < HDL_ANGLE_FIRE_2))) && (2 <= num_cam))
         {
            HAL_GPIO_WritePin (GPIOE, Output_1_Pin, GPIO_PIN_RESET);
            HAL_TIM_Base_Start_IT(&htim3);
            c_angulo_2++;
         }
```

```
        if ((((disparo_actualizado >= HDL_ANGLE_FIRE_3) && (m_lastRotationalPosition_actualizada < HDL_ANGLE_FIRE_3)) ||
         ((disparo_actualizado >= HDL_ANGLE_FIRE_3) &&(flag_rotacion==1)) ||((flag_rotacion==1) &&
         (m_lastRotationalPosition_actualizada < HDL_ANGLE_FIRE_3))) && (3 <= num_cam))
        {
           HAL_GPIO_WritePin (GPIOE, Output_2_Pin, GPIO_PIN_RESET);
           HAL_TIM_Base_Start_IT(&htim4);
           c_angulo_3++;
        }

        if ((((disparo_actualizado >= HDL_ANGLE_FIRE_4) && (m_lastRotationalPosition_actualizada < HDL_ANGLE_FIRE_4)) ||
         ((disparo_actualizado >= HDL_ANGLE_FIRE_4) &&(flag_rotacion==1)) ||((flag_rotacion==1) &&
         (m_lastRotationalPosition_actualizada < HDL_ANGLE_FIRE_4))) && (4 <= num_cam))
        {
           HAL_GPIO_WritePin (GPIOE, Output_3_Pin, GPIO_PIN_RESET);
           HAL_TIM_Base_Start_IT(&htim5);
           c_angulo_4++;
        }

        if ((((disparo_actualizado >= HDL_ANGLE_FIRE_5) && (m_lastRotationalPosition_actualizada < HDL_ANGLE_FIRE_5)) ||
         ((disparo_actualizado >= HDL_ANGLE_FIRE_5) &&(flag_rotacion==1)) ||((flag_rotacion==1) &&
         (m_lastRotationalPosition_actualizada < HDL_ANGLE_FIRE_5))) && (5 <= num_cam))
        {
           HAL_GPIO_WritePin (GPIOE, Output_4_Pin, GPIO_PIN_RESET);
           HAL_TIM_Base_Start_IT(&htim9);
           c_angulo_5++;
        }

        if ((((disparo_actualizado >= HDL_ANGLE_FIRE_6) && (m_lastRotationalPosition_actualizada < HDL_ANGLE_FIRE_6)) ||
         ((disparo_actualizado >= HDL_ANGLE_FIRE_6) &&(flag_rotacion==1)) ||((flag_rotacion==1) &&
         (m_lastRotationalPosition_actualizada < HDL_ANGLE_FIRE_6))) && (6 <= num_cam))
        {
           HAL_GPIO_WritePin (GPIOE, Output_5_Pin, GPIO_PIN_RESET);
           HAL_TIM_Base_Start_IT(&htim12);
           c_angulo_6++;
        }
      m_lastRotationalPosition = firingData.rotationalPosition;
      m_lastRotationalPosition_actualizada = disparo_actualizado;
      }
}
```

# ANEXO II

```
/**
 * @brief Esta función saca la información del paquete de configuración, de él extrae
            * el número de cámaras y el valor de los ángulos en los que las cámaras deben de dispararse.
            *
            * @param se utiliza la estructura creada para los paquetes UDP de configuración que llegan.
            *
            */

void Comprobar_Configuracion (struct ConfigPacket* m_ConfigPacket)
{
   char aux = 0, u = 0;
   if ((m_ConfigPacket->num_camaras <= 57) && (m_ConfigPacket->num_camaras >= 48))
   num_camaras = (char)Convertir_Decimal_a_ASCII((int)m_ConfigPacket->num_camaras);
   else
      return;
   if (((m_ConfigPacket->blank1) &&(m_ConfigPacket->blank2) &&(m_ConfigPacket->blank3) == 32) &&(num_camaras<=6)
   &&(num_camaras>0))
   {
      num_cam = num_camaras;
      num_camaras = 1;
      switch (num_camaras)
      {
         case 1:
            nuevo_angulo = 0;
            for (u=0; u<5; u++)
            {
               aux = (char)m_ConfigPacket->angulo1[u];
               aux = atoi(&aux);
               nuevo_angulo = nuevo_angulo + (double)aux*pow((double)10, (double)(4-u));
            }
            if ((nuevo_angulo>=0) &&(nuevo_angulo<36000))
               HDL_ANGLE_FIRE_1 = nuevo_angulo;
            if(num_cam>1)
               num_camaras = 2;
            else
               break;
         case 2:
            if(m_ConfigPacket->blank4 == 32)
            {
               nuevo_angulo = 0;
               for (u=0; u<5; u++)
               {
                  aux = (char)m_ConfigPacket->angulo2[u];
                  aux = atoi(&aux);
                  nuevo_angulo = nuevo_angulo + (double)aux*pow((double)10, (double)(4-u));
               }
               if ((nuevo_angulo>=0) &&(nuevo_angulo<36000))
                  HDL_ANGLE_FIRE_2 = nuevo_angulo;
               if(num_cam>2)
                  num_camaras = 3;
               else
                  break;
            }
            else
               break;
         case 3:
            if(m_ConfigPacket->blank5 == 32)
            {
               nuevo_angulo = 0;
               for (u=0; u<5; u++)
               {
                  aux = (char)m_ConfigPacket->angulo3[u];
                  aux = atoi(&aux);
                  nuevo_angulo = nuevo_angulo + (double)aux*pow((double)10, (double)(4-u));
               }
               if ((nuevo_angulo>=0) &&(nuevo_angulo<36000))
                  HDL_ANGLE_FIRE_3 = nuevo_angulo;
```

```
                if(num_cam>3)
                    num_camaras = 4;
                else
                    break;
            }
            else
                break;
        case 4:
            if(m_ConfigPacket->blank6 == 32)
            {
                nuevo_angulo = 0;
                for (u=0; u<5; u++)
                {
                    aux = (char)m_ConfigPacket->angulo4[u];
                    aux = atoi(&aux);
                    nuevo_angulo = nuevo_angulo + (double)aux*pow((double)10, (double)(4-u));
                }
                if ((nuevo_angulo>=0) &&(nuevo_angulo<36000))
                    HDL_ANGLE_FIRE_4 = nuevo_angulo;
                if(num_cam>4)
                    num_camaras = 5;
                else
                    break;
            }
            else
                break;
        case 5:
            if(m_ConfigPacket->blank7 == 32)
            {
                nuevo_angulo = 0;
                for (u=0; u<5; u++)
                {
                    aux = (char)m_ConfigPacket->angulo5[u];
                    aux = atoi(&aux);
                    nuevo_angulo = nuevo_angulo + (double)aux*pow((double)10, (double)(4-u));
                }
                if ((nuevo_angulo>=0) &&(nuevo_angulo<36000))
                    HDL_ANGLE_FIRE_5 = nuevo_angulo;
                if(num_cam>5)
                    num_camaras = 6;
                else
                    break;
            }
            else
                break;
        case 6:
            if(m_ConfigPacket->blank8 == 32)
            {
                nuevo_angulo = 0;
                for (u=0; u<5; u++)
                {
                    aux = (char)m_ConfigPacket->angulo6[u];
                    aux = atoi(&aux);
                    nuevo_angulo = nuevo_angulo + (double)aux*pow((double)10, (double)(4-u));
                }
                if ((nuevo_angulo>=0) &&(nuevo_angulo<36000))
                    HDL_ANGLE_FIRE_6 = nuevo_angulo;
                break;
            }
            else
                break;
    }
  }
}
```

## ANEXO III

En este anexo se incluye la configuración para una de las cámaras en las pruebas realizadas en el laboratorio. Este documento se genera una vez se ha finalizado la grabación y toma de datos.

```
//Point Grey Cameras - configuration file parameters
CAMERAS.NUMBER = 2
CAMERA[0].MODEL = Grasshopper3 GS3-U3-23S6C
CAMERA[0].SERIAL_NUMBER = 15420604
CAMERA[0].WIDTH = 1920
CAMERA[0].HEIGHT = 1200
CAMERA[0].PIXEL_DEPTH = 1
CAMERA[0].NCHANNEL = 1
CAMERA[0].ISRAW = TRUE
CAMERA[0].BAYER_TILE = RGGB
CAMERA[0].FPS = 11.773677
CAMERA[1].MODEL = Grasshopper3 GS3-U3-23S6C
CAMERA[1].SERIAL_NUMBER = 15420594
CAMERA[1].WIDTH = 1920
CAMERA[1].HEIGHT = 1200
CAMERA[1].PIXEL_DEPTH = 1
CAMERA[1].NCHANNEL = 1
CAMERA[1].ISRAW = TRUE
CAMERA[1].BAYER_TILE = RGGB
CAMERA[1].FPS = 11.773706
```

Universidad de Alcalá