Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE VALLADOLID
ESCUELA DE INGENIERIAS INDUSTRIALES

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

COMUNICACIÓN ETHERNET ENTRE UN ORDENADOR Y UNA FPGA EMPLEANDO UN PROCESADOR TEENSY

Autor:

Serna Bravo, Pablo

Tutor:

de Pablo Gómez, Santiago

Departamento de Tecnología Electrónica

Valladolid, julio de 2022

AGRADECIMIENTOS

A Santiago de Pablo, por haberme dado la oportunidad de descubrir un campo prácticamente nuevo para mí, y por su esfuerzo y dedicación para que aprendiese con este trabajo.

A los profesores y personas de la Universidad de Valladolid que me han enseñado a pensar como un ingeniero y gracias a los cuales he podido enfrentarme a todos los retos de realizar este trabajo.

A mi familia, por estar siempre en los momentos más duros y darme un apoyo incondicional en esta etapa.

Y a Ane, por compartir este camino conmigo y por el gran equipo que hemos formado.

RESUMEN Y PALABRAS CLAVE

En este trabajo se ha implementado un enlace Ethernet entre un ordenador y un dispositivo reconfigurable tipo FPGA. Se ha empleado un microcontrolador Teensy 4.1 con una conexión Ethernet de 100 Mbps que se comunica con el ordenador a través de tramas TCP. Entre este dispositivo y la FPGA se ha implementado una interfaz paralela que permite comunicarse con múltiples componentes internos de la FPGA.

Se ha empleado el software de Qt y de Arduino para crear programas en ambos dispositivos. Se describe el código utilizado y se obtienen resultados de la velocidad de comunicación.

Palabras clave: Ethernet, FPGA, Teensy, Qt, Arduino.

ABSTRACT & KEYWORDS

In this paper an Ethernet link has been implemented between a computer and a reconfigurable FPGA device. A Teensy 4.1 microcontroller has been used with a 100 Mbps Ethernet connection that communicates with the computer through TCP frames. A parallel interface has been implemented between this device and the FPGA to communicate with multiple internal components on the FPGA.

Qt and Arduino software have been used to create programs on those devices. The used code is described and communication speed results are obtained.

Keywords: Ethernet, FPGA, Teensy, Qt, Arduino.

ÍNDICE

ANEXOS DE CÓDIGO

LISTADO DE FIGURAS

LISTADO DE TABLAS

LISTADO DE ACRÓNIMOS

ASIC	Application Specific Integrated Circuit
CPLD	Complex Programmable Logic Device
FPGA	Field Programmable Gate Arrays
USB	Universal Serial Bus
PC	Personal Computer
TCP	Transmission Control Protocol
RAM	Random Access Memory
EEPROM	Electrically Erasable Programmable Read-Only Memory
SPI	Serial Peripheral Interface
I2C	Inter Integrated Circuits
SDIO	Secure Digital Input/Output
SD	Secure Digital
DMA	Direct Memory Access
IDE	Integrated Development Environment
LED	Light Emitting Diode
LAN	Local Area Network
IEEE	Institute of Electrical and Electronics Engineers
MAC	Media Access Control
IP	Internet Protocol
ARP	Address Resolution Protocol
NIC	Network Information Center
UDP	User Data Protocol
CPU	Central Processing Unit
GUI	Graphical User Interface

CLB	Configurable Logic Block
SRAM	Static Random Access Memory
DNS	Domain Name System
UTF-8	8-bit Unicode Transformation Format

CAPÍTULO 1 INTRODUCCIÓN

1.1 CONTEXTO

El mundo actual gira alrededor de los dispositivos electrónicos. La información, las comunicaciones, la tecnología y, en general, todos los elementos de nuestro día a día, incorporan algún tipo de procesador que permite al dispositivo tomar decisiones relativamente complejas y facilitar la vida del usuario. Sin embargo, el desarrollo de estos dispositivos ha sido un camino largo, con líneas de trabajo muy diferentes que intentaban alcanzar nuevos logros. En este sentido, se han diferenciado desde el principio dos vertientes de avance claras: una basada en el hardware, que centra sus esfuerzos en crear nuevos dispositivos capaces de optimizar tareas concretas, como los ASIC (*Application Specific Integrated Circuit*); y otra que avanza centrándose en el software, con procesadores más potentes de propósito general que ofrezcan gran flexibilidad de cara a crear y utilizar programas de más alto nivel.

En la década de los 80 empiezan a aparecer dispositivos que intentan aunar las ventajas de ambas corrientes, buscando tener un hardware capaz de optimizarse y utilizarse en tareas específicas sin perder la flexibilidad de un dispositivo programable. Aparecen los CPLD (*Complex Programmable Logic Device*), circuitos basados en puertas lógicas capaces de interconectarse entre sí para formar estructuras electrónicas complejas, pero diseñados para poderse programar y borrar para cambiar las estructuras que contienen. Finalmente evolucionan hasta las FPGA (*Field-Programmable Gate Array*), formadas por bloques lógicos que se pueden programar internamente para definir su comportamiento e interconectar entre sí para definir estructuras complejas, como por ejemplo procesadores [1].

Con una FPGA podemos comprobar diseños de procesadores de forma más fiable que con simulaciones con modelos matemáticos y, además, permiten cambiar partes concretas de la estructura interna sin afectar al resto del circuito, con lo que el diseño de procesadores se vuelve mucho más rápido y cómodo. Actualmente, se utilizan tanto para la verificación de circuitos electrónicos de todo tipo como para funcionar como procesadores variables y actualizables en aplicaciones específicas o para la docencia.

1.2 OBJETIVOS Y MOTIVACIÓN

En el Departamento de Tecnología Electrónica de la Universidad de Valladolid se trabaja con simulaciones de procesadores distintos para que los alumnos de las diferentes asignaturas que se imparten puedan observar y experimentar con el comportamiento de estos. Actualmente se utilizan modelos matemáticos hechos con Simulink, una extensión del programa Matlab, una plataforma para programación y cálculo numérico capaz de analizar datos y crear modelos matemáticos precisos.

Sin embargo, no deja de ser una simulación aproximada por un modelo matemático, que muchas veces difiere de la realidad y no nos permite observar los fenómenos que pueden aparecer en un dispositivo real. Para solucionarlo, se están desarrollando varias líneas de trabajo que buscan implementar los procesadores en una FPGA que sea capaz de interactuar e intercambiar datos a tiempo real con el usuario en el laboratorio. Para ello, hace falta conectar un ordenador que sea capaz de programar el circuito interno de la FPGA, pero también dar órdenes e intercambiar información para poder usarla una vez programada.

La idea global es conectar un ordenador a un microcontrolador intermedio para que procese las órdenes que se le envíen y las traduzca para poder controlar la FPGA. Se está trabajando en opciones de conexión a través de USB, WiFi y Ethernet, cada uno con microcontroladores distintos especializados en cada tipo de conexión. En este trabajo, nos centraremos en la conexión a través de Ethernet.

El objetivo es conectar un ordenador con una FPGA de manera que el PC actúe como *master* enviando las órdenes y la FPGA como *slave*, ejecutando lo que le llegue. Para ello, crearemos un programa en el ordenador con el que podamos elegir qué queremos que haga la FPGA, que se encargará de mandar mensajes a un procesador intermedio a través de un cable ethernet con el protocolo TCP; el procesador intermedio, en nuestro caso el Teensy 4.1, se encargará de procesar las órdenes que le lleguen y enviarlas a la FPGA, a la que estará conectado a través de sus pines.

Este trabajo se centra en conseguir que el programa comunique con la FPGA y pueda controlar algunos espacios de memoria. No son objetivos prioritarios la velocidad ni el flujo de datos que se puedan intercambiar, si no que se intentará conseguir el mejor posible para que se puedan optimizar con trabajos posteriores partiendo de un diseño que funcione y sea robusto.

1.3 ESQUEMA DEL PROYECTO

A continuación, se define brevemente el contenido de los capítulos en los que se estructura el proyecto.

- Capítulo 1: Introducción
  Es el desarrollado hasta ahora, que plantea el contexto de este trabajo, los objetivos y motivación y la estructura de este.

- Capítulo 2: Antecedentes y desarrollo teórico
  Desarrolla el estado de la técnica y explica los conocimientos teóricos necesarios para el proyecto. Se describe el Teensy 4.1, la conexión Ethernet, el protocolo TCP/IP, los softwares de programación y la FPGA.

- Capítulo 3: Desarrollo del proyecto
  Define al completo los diferentes elementos que componen el proyecto y cómo se conectan y comunican entre sí. Explica primero el esquema general y los pasos previos al diseño de programas y después describe los programas creados tanto para el PC como para el Teensy.

- Capítulo 4: Resultado final, testeo y verificación
  Comprueba el funcionamiento del proyecto creado, mostrando los resultados prácticos y muestra pruebas realizadas para comprobar la velocidad de comunicación.

- Capítulo 5: Conclusiones
  Presenta los objetivos conseguidos y las conclusiones obtenidas del proyecto y plantea las líneas futuras.

CAPÍTULO 2 ANTECEDENTES Y DESARROLLO TEÓRICO

En este apartado se revisará el estado de la técnica y se explicarán los conocimientos teóricos previos necesarios para el entendimiento de este trabajo, así como la descripción del material, herramientas y estructuras utilizadas en su desarrollo. Lo abordaremos por el lado del hardware, en el que hablaremos del procesador utilizado, los puertos y las conexiones; y por el lado de software, en el que hablaremos del lenguaje de programación y librerías, programas y herramientas de desarrollo.

Desarrollaremos por separado la explicación relativa a la FPGA, ya que no es el objetivo principal del proyecto, pero será necesaria para la definición de elementos del mismo.

2.1 MICROCONTROLADOR TEENSY 4.1

Los Teensy son una familia de microcontroladores basados en USB capaces de implementar muchos tipos de proyectos. Han sido desarrollados por dos estadounidenses que se agrupan con el nombre de *PJRC.COM*. El Teensy 4.1 es el último modelo desarrollado de esta familia [2].

El microcontrolador Teensy se ha ido desarrollando y actualizando hasta llegar a la versión actual 4.1, por lo que no hay proyectos recientes que utilicen este diseño, pero sí otros que utilizan diseños anteriores y que pueden darnos información útil para el proyecto actual. Sin embargo, como suele ocurrir con estos controladores, son un medio que se utiliza en proyectos más grandes y no se describe demasiado el código utilizado para el procesador, ni su funcionalidad.

El Teensy 4.0 se ha utilizado transmitiendo datos a través de USB en un proyecto más complejo para medir el consumo de una placa de desarrollo ligera [3].

Otros proyectos utilizan el Teensy 3.6 como elemento auxiliar para validar el comportamiento de una FPGA, en este caso para la generación de ondas sonoras [4].

También podemos encontrar un proyecto similar al nuestro que se centra en la FPGA para que pueda ser programada utilizando el Teensy, que está a su vez conectado al PC por USB. Es un proyecto muy completo pero que define muy poco la parte del Teensy, por lo que tan solo aportará un ejemplo cercano para este trabajo [5].

En este proyecto se elige el Teensy 4.1 por encima del resto de microcontroladores de esta familia por ser el único que incorpora conexión a través de Ethernet. Destaca frente a otros microcontroladores por ser capaz de alcanzar los 100 Mbits por segundo y, además, puede funcionar a 600 MHz, por lo que su velocidad no será limitante para nuestro diseño; e incorpora todas las funcionalidades necesarias para el desarrollo del proyecto.

Las especificaciones del Teensy 4.1 que más nos interesan son las siguientes:

- Procesador ARM Cortex-M7 a 600 MHz.
- Unidad matemática de coma flotante de 64 y 32 bits.
- 7936K de memoria Flash, 1024K de RAM y 4K de EEPROM.
- Conexión USB a 480 Mbit/s.
- 55 pines digitales de entrada y salida.
- 8 puertos serie, 3 SPI y 3 I2C.
- 1 puerto SDIO (4 bit) para la tarjeta SD incorporada.
- Ethernet de 10/100 Mbit/s con el módulo DP83825 PHY.
- 32 canales DMA de propósito general

2.1.1 Procesador

El ARM Cortex-M7 es un procesador de 32 bits que funciona a 600 MHz y que es capaz de ejecutar dos instrucciones en un mismo ciclo de reloj entorno al 50% de las veces con un código optimizado. Es capaz de acceder por separado a la memoria de programa y de datos, pudiendo hacer varios accesos en el mismo ciclo, y utiliza lo que se denomina "*branch prediction*", una estructura capaz de predecir y simplificar las instrucciones cuando se repiten en un bucle.

2.1.2 Pines

El microcontrolador cuenta con pines analógicos y digitales de entrada y salida. Podemos configurar estos pines como *pullup* o *pulldown* para utilizar las resistencias internas que nos permiten controlar el paso de corriente desde el circuito exterior.

La disposición de los pines en el Teensy 4.1 es la siguiente:

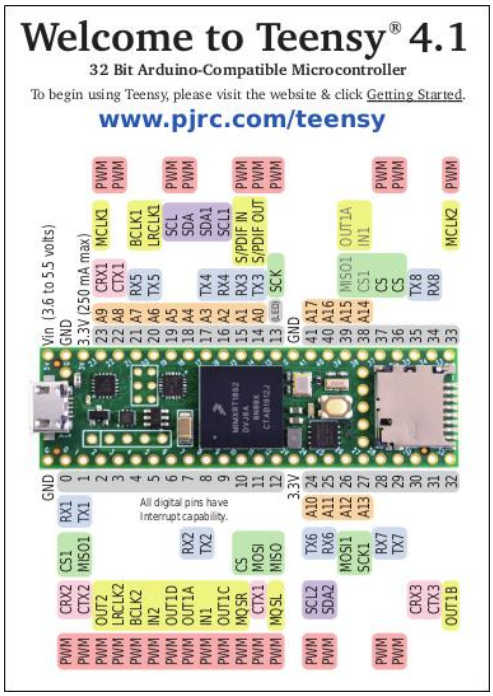

Figura 1 Conectores laterales Teensy 4.1 [2]

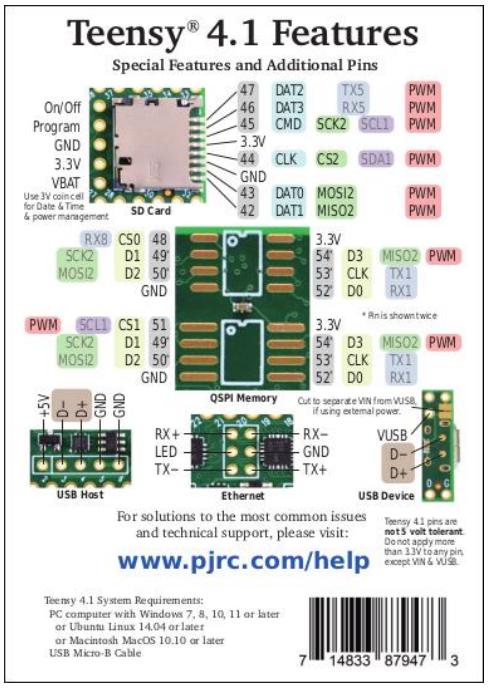

Figura 2 Conectores adicionales Teensy 4.1 [2]

En este proyecto utilizaremos los pines de Ethernet para conectar el microcontrolador al PC y algunos de los pines de entrada y salida para conectarlo con la FPGA.

2.1.3 Comunicación

La principal vía de comunicación del Teensy es el puerto USB, que permite transferir el programa desde el PC e intercambiar datos con el protocolo serie. Se transmiten bytes en ambas direcciones y, con el software propio del Teensy, se pueden enviar mensajes y visualizarlos en el PC. Además, sirve como canal de alimentación para el microcontrolador.

Tiene otros puertos de comunicación (serie, SPI, I2C) que no se utilizan en este proyecto, pero podrían llegar a utilizarse en líneas futuras para enviar otro tipo de instrucciones a la FPGA o intercambiar información con otros dispositivos.

Nuestro Teensy 4.1 recibe instrucciones e intercambia datos con el PC a través de Ethernet. Tiene un controlador y un módulo DP83825 PHY que permiten gestionar las comunicaciones por un cable Ethernet sin necesidad de un *Ethernet Shield* (módulos SPI o I2C que permiten conexión a Ethernet a otros microcontroladores).

El fabricante recomienda un kit de conexión para un cabezal que permite conectar el cable Ethernet y ofrece funcionalidades relacionadas con el software. En este proyecto, por ser un prototipo, utilizaremos un cabezal simple conectado directamente del que hablaremos más adelante.

2.1.4 Memoria

El Teensy 4.1 puede almacenar 8 Mbytes de código de programa y variables de solo lectura en la memoria *flash*. Tiene una memoria RAM de 1024K, de la cual a la mitad se accede como *"tightly coupled memory"* o memoria fuertemente acoplada, lo que permite accesos rápidos para un mejor desempeño. Parte de la memoria *flash* se puede utilizar para emular una memoria EEPROM.

Además, permite hacer expansiones de memoria e incorpora una tarjeta SD.

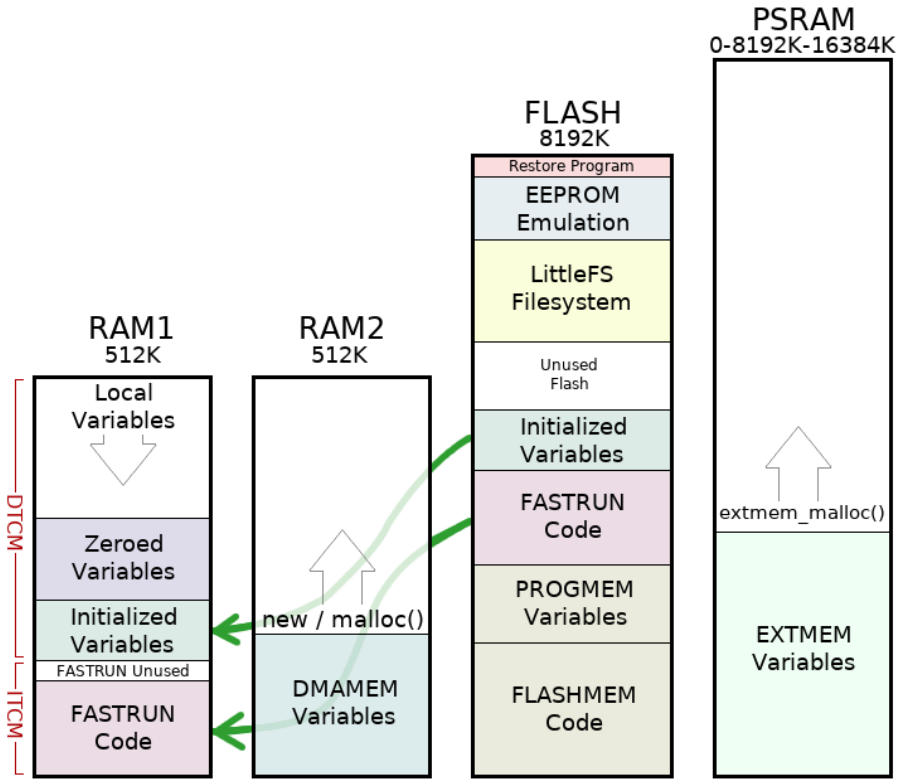

Figura 3 Esquema de memoria Teensy 4.1 [2]

2.1.5 Software y programación

El Teensy 4.1 se puede programar con diferentes herramientas. Nosotros vamos a utilizar la opción del entorno de desarrollo integrado (IDE) de Arduino junto con la ampliación específica *Teensyduino*, que nos permite desarrollar programas para nuestro microcontrolador. Además, incluye librerías especialmente optimizadas para el Teensy.

Una segunda opción sería utilizar la expansión Visual Micro para el software de Microsoft Visual Studio si quisiésemos programar tanto para el PC como para el Teensy en un mismo entorno de desarrollo.

El Teensy incorpora un software automático que entra en el modo de programación y que se indica con un LED rojo sobre la placa. Para entrar en este modo de forma manual se puede utilizar también el botón que vemos en la Figura 5.

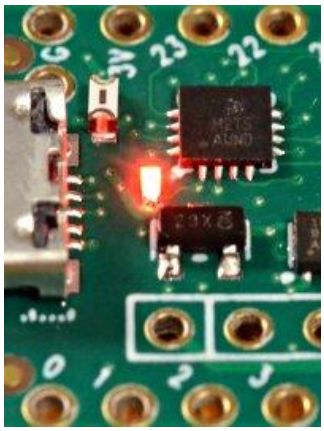

Figura 4 LED Rojo del modo programación [2]

Figura 5 Botón de modo programación manual [2]

2.2 COMUNICACIÓN ETHERNET

Ethernet es una tecnología para intercambio de información entre dispositivos electrónicos que combina hardware y software. Se utiliza principalmente para interconexión en redes locales (LAN). Se corresponde con el estándar **IEEE 802.3** del *Institute of Electrical and Electronic Engenieers (IEEE)* [6].

El protocolo Ethernet permite a todos los dispositivos interconectados enviar y recibir paquetes de datos. Para conectar dispositivos entre sí se utilizan los puertos Ethernet cableados directamente o conectados a un elemento intermedio (enrutadores, *hubs* o *switchs*) que se encarga de dirigir los mensajes que le llegan a su destino, creando así una red local.

Para poder identificar tanto el emisor como el receptor del mensaje el protocolo utiliza una dirección MAC, una dirección única compuesta por 6 bytes e implementada por el IEEE que identifica a cada dispositivo electrónico.

El cableado de Ethernet se divide en 7 categorías y sus variantes, según el volumen de datos que son capaces de intercambiar, las protecciones que incorpore el cable y su longitud. En nuestro caso, vamos a utilizar un cable CAT5e UTP, formado por cuatro pares de cables trenzados sin apantallamiento, que es capaz de transmitir hasta 1000 Mbits por segundo, más que suficiente ya que en el Teensy 4.1 nos encotramos con un límite de 100 Mbits [17], [18], [19].

Cada uno de los cables trenzados que contiene tiene una función y un código de colores (T-568B), y terminan en un conector denominado RJ45 que separa cada uno de los hilos y los conecta al puerto Ethernet.

Cable recto (normal/paralelo) T568B

Pin Nº	Extremo 1	Extremo 2	Color	Función
1			Blanco - Naranja	Transceive data +
2			Naranja	Transceive data -
3			Blanco - Verde	Receive data +
4			Azul	Bi-directional Data +
5			Blanco - Azul	Bi-directional Data -
6			Verde	Receive data -
7			Blanco - Marrón	Bi-directional Data +
8			Marrón	Bi-directional Data -

Figura 6 Código de color hilos cable CAT 5 [20]

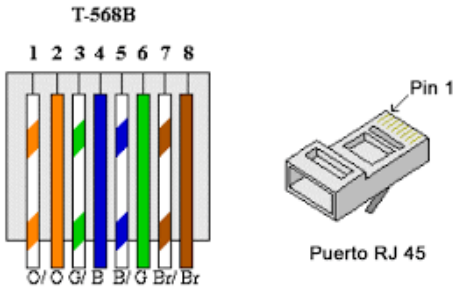

Figura 7 Código de color hilos cable CAT 5 [21]

En nuestro caso, tenemos un conector con un puerto ethernet que se conecta al Teensy a través de 6 pines (Figura 8).

Figura 8 Conector Ethernet para el Teensy 4.1

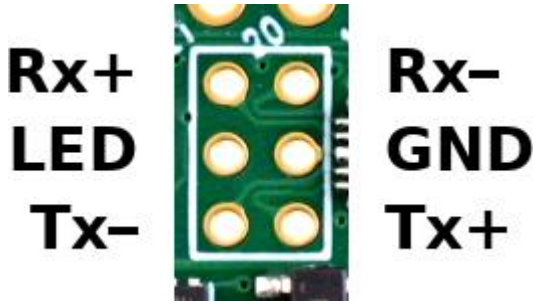

Figura 9 Pines de Ethernet del Teensy 4.1 [2]

Como podemos ver en la Figura 9, el Teensy tiene dos pares de pines para los hilos de transmisión y recogida de datos (Tx, Rx), y ululiza dos de los hilos bidireccionales para conexión a tierra (GND) y para encender un LED. Sin embargo, el pin del LED solo es útil cuando se tiene el kit de Ethernet para el Teensy del que hemos hablado anteriormente, o en su defecto un *Ethernet shield*.

En cuanto a la conexión entre el PC que actua de *master* y el Teensy, el objetivo es conectarlos directamente con un cable Ethernet cruzado, es decir, que uno de los extremos intercambie los hilos Tx y Rx para que ambos dispositivos puedan hablar entre sí.

Sin embargo, para el prototipo que hemos utilizado para las pruebas, se introduce un *router* (enrutador) entre medias debido a las limitaciones de hardware del lugar de trabajo, por lo que el Teensy estará conectado con un cable Ethernet al router y este se comunicará con el PC a través de WiFi. Esto puede disminuir la velocidad de intercambio de datos, pero no perjudica nuestro principal objetivo que es la comunicación y robustez.

2.3 PROTOCOLO TCP/IP

La conexión Ethernet es tan solo el primer paso para poder establecer comunicación entre distintos dispositivos, es la primera capa de lo que se conoce como protocolos de internet o modelo TCP/IP que podemos ver en la Figura 10. La **capa Ethernet** se clasifica como capa de "Acceso al medio", que se compone de la estructura física de conexión y un protocolo de enlace de datos [7].

Sin embargo, para poder hacer un intercambio efectivo y seguro de datos hacen falta otra serie de capas según el siguiente esquema:

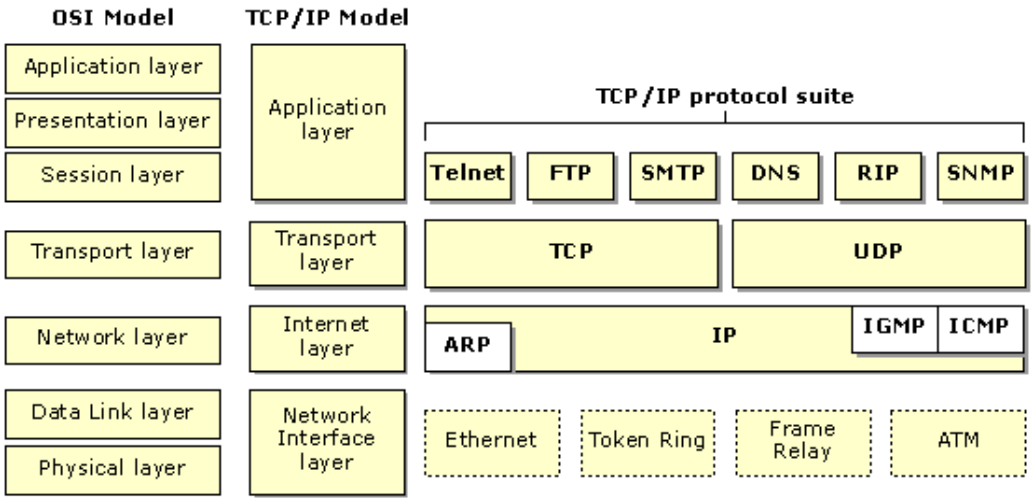

Figura 10 Capas del modelo TCP/IP frente a sus protocolos correspondientes [22]

Por encima de la capa de Ethernet aparece la **capa de internet**, gesionada por el protocolo IP. Esta capa se encarga de etiquetar los paquetes de información intercambiados con la referencia del dispositivo emisor y del destinatario. Cualquier dispositivo capaz de conectarse a internet tiene una dirección única (dirección IP) que lo identifica y lo distingue del resto.

Hay varios niveles de direcciones. Como hemos visto, el protocolo Ethernet utiliza las direcciones MAC del hardware de las tarjetas de red. Para poder pasar de unas capas a otras las direcciones se traducen con distintos protocolos. Para conocer la dirección MAC a partir de la dirección IP se utiliza el protocolo ARP.

El protocolo IP define una dirección de 32 bits que se separan por grupos de 8 bits notados en decimal y separados por puntos. Se dividen en grupos de red y de máquina. El primer grupo identifica la red de dispositivos y lo define el NIC (*Network Information Center*), mientras que el segundo grupo ordena los

distintos dispositivos que forman parte de la red. La cantidad de bits de cada grupo dependerá del número de dispositivos que pueden pertenecer la red.

Una vez incluidas las direcciones IP en el mensaje a enviar, el protocolo se encarga de direccionar los paquetes por la mejor ruta posible y evitar que se produzcan congestiones en los nodos intermedios. Sin embargo, es un protocolo "no orientado a la conexión", por lo que no se preocupa de que los paquetes lleguen organizados, ni detecta cuando un paquete se pierde, se duplica o se modifica; por lo que serán las capas superiores las que deban detectar y corregir estos errores.

La **capa de transporte** permitie la comunicación entre extremos de la red (*host to host*). En esta capa aparecen dos protocolos distintos: el protocolo TCP (*Transmission Control Protocol*), que está orientado a la conexión, por lo que los paquetes llegan ordenados, sin errores y además no saturan la red; y el protocolo UDP (*User Datagram Protocol*), no orientado a la conexión, cuyos paquetes no están controlados, pero se intercambian mucho más rápido.

El protocolo TCP se encarga de segmentar en paquetes la información que se quiere intercambiar y enumerar cada paquete para poder verificar y ordenar la información una vez recibida. Cuando se recibe la información, el protocolo TCP agrupa y ordena los paquetes y, si alguno se ha perdido o llega modificado, avisa al emisor para que se vuelva a mandar.

Cuando queremos enviar pocos paquetes o no es necesario verificar que cada paquete ha llegado y está en orden, el protocolo TCP puede estar ralentizando inutilmente el proceso. Para estos casos se emplea el protocolo UDP, que es mucho más sencillo y rápido.

En este proyecto vamos a utilizar el protocolo TCP ya que se centra en la robustez y el control. Los mensajes de control deben llegar ordenados y completos, por lo que queremos asegurarnos de que el protocolo los verifique y los ordene si es necesario. Para etapas posteriores del proyecto en las que los datos a intercambiar sean gráficas o estados de la FPGA se podrá plantear diseñar un modelo análogo que gestione esos mensajes con el protocolo UDP.

Cada protocolo define su estructura de la información y construye sus mensajes, anidando estos en los mensajes de la estructura inferior, hasta llegar a la capa física y transportar el mensaje a su destino, donde será descompuesto e interpretado. El esquema global del que hemos hablado sería el que vemos en la Figura 11.

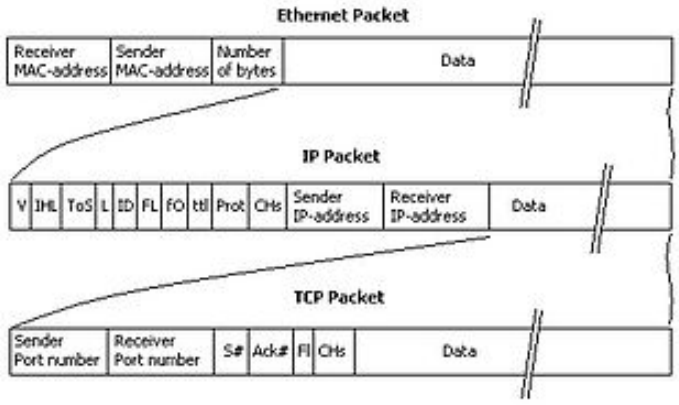

Figura 11 Esquema de la estructura de los paquetes del modelo TCP/IP [23]

Además de componer el mensaje, esta capa coordina los procesos de escucha o "*sockets*" en cada dispositivo, lo cual permite que detecten cuando llega un mensaje y sean capaces de recibirlo y guardarlo. Se estructura como un servidor que abre un socket y escucha a clientes que se conectan con su propio socket.

La capa más alta es la **capa de aplicación**, que agrupa los protocolos que ofrecen servicios al usuario para aplicaciones concretas, intercambiando información gracias a las capas inferiores. En este caso será la que queremos construir nosotros, por lo que se desarrollará a lo largo de este trabajo. La estructura más compleja que consiguen los protocolos se representa en la Figura 12.

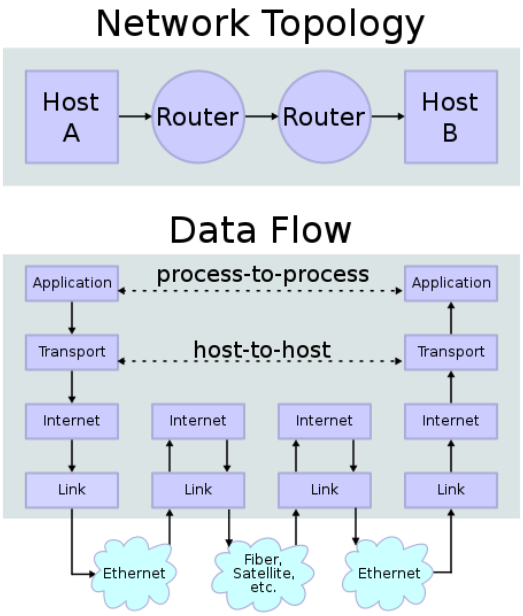

Figura 12 Estructura de la transmisión de información con el modelo TCP/IP [22]

2.4 ARDUINO IDE Y TEENSYDUINO

El Teensy 4.1 utiliza el entorno de desarrollo (IDE) de Arduino con una extensión específica denominada Teensyduino para desarrollar el código que se ejecutará en nuestro microcontrolador.

Arduino es una plataforma electrónica de código abierto que se enfoca en ser simple y fácil de usar. Desarrolla sus propias placas de microcontroladores, pero también ofrece un entorno de desarrollo multiplataforma que se puede utilizar para programar placas de otros proveedores. Su objetivo es ser abierto, flexible y asequible para que se pueda utilizar para aprender y desarrollar en cualquier ámbito [8].

Para descargar el software de Arduino accedemos a su página web (www.arduino.cc/en/software) y descargamos la última versión estable para nuestro sistema operativo. En nuestro caso será la versión 1.8.19 para Windows. Solo funciona a partir de Windows 7, lo cual no será un problema ya que utilizamos Windows 10. El software es completamente gratuito y bastará con ejecutarlo y completar la instalación. En esta debemos elegir el lugar para la instalación y nos permitirá también seleccionar el idioma.

Una vez instalado Arduino debemos descargar la extensión **Teensyduino**. Accedemos a la página web del fabricante (www.pjrc.com/teensy/td_download.html) y descargamos la versión que corresponda para la versión de Arduino que hemos instalado. En este caso, la última versión estable es la 1.56, que da soporte a las versiones de Arduino de la 1.8.5 a la 1.8.19. La descargamos para Windows y la ejecutamos. Para la instalación únicamente debemos seleccionar dónde tenemos instalado Arduino y elegir qué librerías queremos que nos instale, aunque el fabricante recomienda instalarlas todas, así que eso haremos en este caso.

En la página web se incorpora una guía gráfica de instalación y ayudas para problemas con el antivirus o con instalaciones fuera del usuario administrador del PC. En nuestro caso no ha habido ningún problema en la instalación.

Una vez instalados ambos ejecutamos Arduino para definir las opciones necesarias. Como podemos ver en la Figura 13, si accedemos a la pestaña de herramientas podemos elegir la placa para la que queremos programar. Si hemos instalado correctamente Teensyduino nos deberán salir los Teensy. Seleccionamos el Teensy 4.1 en nuestro caso.

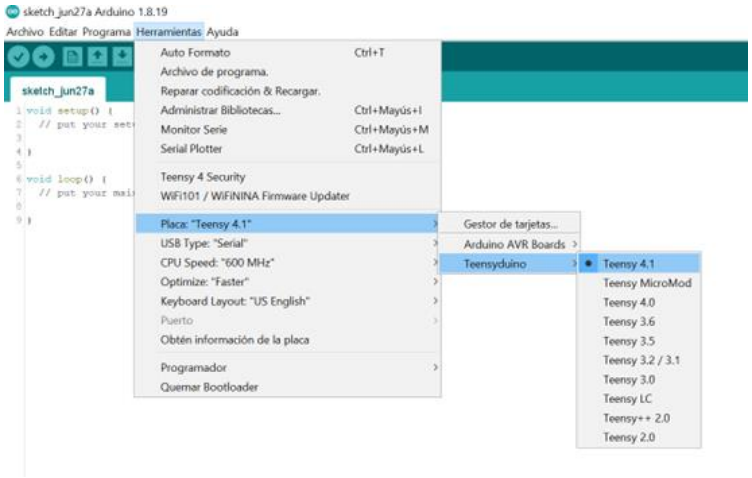

Figura 13 Elección de la placa en Arduino

En esta misma pestaña podemos seleccionar el tipo de entrada que nos llegará por el USB, ya que el Teensy puede enviar señales como distintos periféricos. Lo mantendremos en "Serial" para poder enviar mensajes de control al monitor serie. Nos permite también elegir la velocidad de procesamiento de la CPU, que en este caso está a 600 MHz. También permite definir el tipo de optimización, que mantendremos en faster y la disposición del teclado si quisiesemos usar el Teensy como tal.

Con estos entornos instalados podemos programar para el Teensy utilizando los lenguajes de C y C ++ y podemos utilizar las librerías que incorporan tanto Arduino como el Teensy.

Para poder transferir nuestro programa al Teensy debemos descargar también el **Teensy Loader** desde la página del fabricante (www.pjrc.com/teensy/loader.html). Elegimos el sistema operativo que estemos utilizando y descargamos el archivo. No es necesaria instalación, simplemente con ejecutarlo debemos ver una ventana similar a la Figura 14.

Figura 14 Teensy Loader desconectado

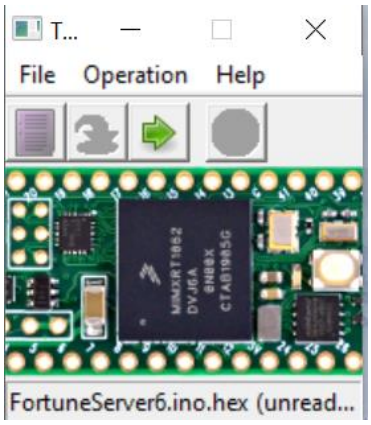

Figura 15 Teensy Loader conectado

Cuando conectamos nuestro Teensy el programa lo detecta automáticamente y aparece una imagen del Teensy que hayamos conectado, en nuestro caso el Teensy 4.1. Cuando pulsamos el boton señalado en el microcontrolador entramos en el modo de programación y podemos cargar el programa que hayamos realizado desde Arduino. Debajo de la imagen nos señala el nombre del achivo de Arduino que contiene el programa que está cargado en el Teensy.

Una vez cargado el programa se mantiene en el Teensy hasta que lo sobrescribamos, aunque no esté conectado a la fuente de energía. Si está conectado y no está en el modo de programación estará ejecutando el programa que tenga cargado.

2.5 ENTORNO DE DESARROLLO Qt

Qt es un entorno de trabajo o *framework* que se utiliza para desarrollar todo tipo de programas. Es muy potente ya que tiene una estructura de código propia y permite utilizarla en distintas plataformas. Tiene muchas facilidades para desarrollar interfaces gráficas de usuario (GUI) y es por eso que lo hemos elegido para programar el software que controlará la FPGA desde el PC, ya que necesitará una interfaz desde donde lo podamos manejar [9].

2.5.1 Descarga e instalación

La empresa ofrece una licencia de pago y otra de código abierto gratuita. Tienen una filosofía de sofware comunitario y libre y puesta en común de conocimientos que se reflejan en sus condiciones de compartir libremente el software que se genere con este tipo de licencias de código abierto. Puesto que nosotros no tenemos intención de monetizar este proyecto ni estamos trabajando para una empresa elegiremos este tipo de licencia.

Para descargarla se accede a su página web (www.qt.io/download) y se selecciona la licencia de código abierto. Detecta automáticamente el sistema operativo y ofrece la descarga del instalador correspondiente, aunque da la opción de elegir otra si fuese necesario. Si se ejecuta, el instalador pedirá crear una cuenta gratuita en Qt simplemente utilizando un correo electónico y verificará que conocemos las condiciones de la licencia que estamos utilizando, así como que no estamos utilizandola para ninguna empresa.

Más adelante deberemos señalar el directorio en el que queremos instalar Qt y el tipo de instalación. Es recomendable realizar una instalación personalizada y elegir únicamente la versión y los módulos que se vayan a utilizar en el proyecto, ya que son archivos que ocupan mucho espacio de almacenamiento. Para este proyecto hemos descargado la versión 6.2.4, la más actualizada en el momento de la descarga, y hemos excluido los módulos de desarrollo de aplicaciones para Android y desarrollo 3D. Es recomendable guardar el instalador ejecutable ya que nos permitirá modificar los módulos instalados sin necesidad de reinstalar el programa.

Para que Qt funcione correctamente una vez instalado hace falta dar un último paso [24]. Debemos buscar “Editar las variables de entorno del sistema” en nuestro buscador de aplicaciones del PC, lo que abriría una pestaña similar a la Figura 16.

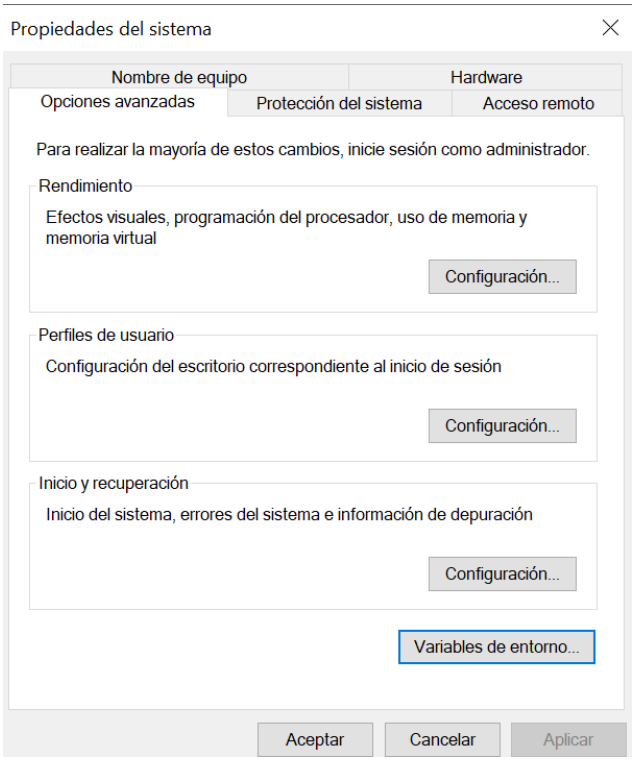

Figura 16 Entorno de propiedades del sistema de Windows 10

Si pinchamos en variables de entorno nos aparece una nueva ventana en la que debemos crear una nueva variable del sistema, llamarla QTDIR y conectarla al dirrectorio donde está instalado Qt, como se puede ver en la Figura 17.

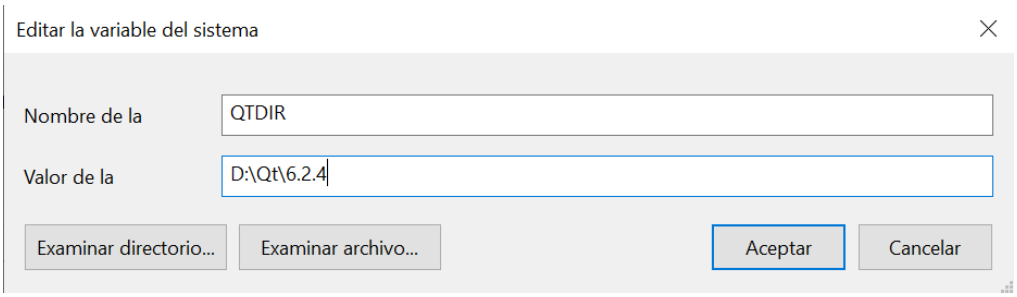

Figura 17 Nueva variable del sistema para Qt

Una vez hecho esto podremos utilizar el entorno de Qt para nuestro proyecto.

2.5.2 Estructura de programación

El entorno de Qt está diseñado para programar en C++, aunque también permite otros lenguajes. Desarrolla una estructura de programación orientada a objetos, es decir, crea elementos a los que se asocian atributos concretos y comportamientos u operaciones con los que podemos trabajar. Estos objetos se agrupan en clases.

Las clases son estructuras que definen el estado y comportamiento de un grupo de objetos. Una clase contiene una función constructora que nos permite crear nuevos objetos de la clase. Las clases pueden definir variables (atributos) y funciones (operaciones) para los objetos, lo que nos permite caracterizarlos y trabajar con los mismos. Además, los clasifica en públicos y privados, permitiendonos decidir si se pueden utilizar fuera de la clase o solo dentro de ella, respectivamente. Una clase puede heredar de otras clases estos elementos para poder utilizarlos dentro de la misma [10].

El corazón de la estructura de Qt es su clase *QObject*. La caracrerística principal de su modelo de objetos es la comunicación directa ente objetos que permiten las *signals* o señales y los *slots* o ranuras. Esta característica permite unir objetos entre sí sin necesidad de que estén relacionados [11].

Las **señales** se emiten cuando cambia el estado de un objeto de alguna forma que pueda ser útil conocer. Las señales se pueden emitir desde cualquier punto de nuestro programa.

Las **funciones slot** son prácticamente iguales a las funciones miembro de una clase de C++, por lo que se las puede llamar directamente. Sin embargo, las funciones slots pueden estar conectadas a señales. Una señal puede llamar a

un slot independientemente de si este es público o privado y de la clase a la que pertenezca.

Para conectar señales y slots utilizamos la función "connect". Esta función tiene cuatro argumentos:

- Puntero al objeto al que pertenece la señal (emisor)
- Señal del objeto
- Puntero al objeto al que pertenece el slot (receptor)
- Slot al que está conectada la señal

Y por tanto se define como *connect(*emisor, señal, *receptor, slot)*.

Así, cuando se emite la señal en el emisor, el slot del receptor se ejecuta instantáneamente. Una misma señal puede estar conectada a diferentes slots y un slot puede estar conectado también a diferentes señales. Incluso pueden conectarse señales entre sí para que se emita la segunda cuando se emite la primera. También pueden desconectarse elementos, pero no suele ser necesario ya que la conexión desaparece cuando se destruye alguno de los objetos implicados.

Para que un objeto pueda utilizar las señales y los slots debemos incluir la macroinstrucción *Q_OBJECT* en la definición de su clase, que define la estructura gracias al **Sistema de Meta-Objetos** de Qt [12].

2.5.3 Interfaz de usuario

En Qt se genera una interfaz gráfica de usuario gracias a la clase *QGuiApplication*, que gestiona y controla esta interfaz y sus interacciones. Por debajo de este nivel aparecen los ***widgets***, o "*Windows gadgets*", que son todos los elementos que conforman la interfaz gráfica.

Los widgets se estructuran como "padres" e "hijos" según la relación de pertenencia. Cuando se genera un widget hijo (por ejemplo, un botón) sobre un widget padre (por ejemplo, una ventana) aparece también una relación de dependencia que hace que el hijo solo exista mientras lo haga el padre.

Con las diferentes clases de widgets y sus atributos y relaciones se puede construir la interfaz gráfica de usuario al completo.

2.5.4 Estructura externa

Al igual que otros entornos de desarrollo de C++, Qt nos permite crear proyectos completos que divide en los siguientes archivos:

- Archivo de proyecto (.pro): Define los módulos de Qt que queremos incluir en el proyecto, así como el estandar de C++ y el resto de archivos que contiene.
- Archivos de cabecera (.h): Contiene las definiciones de variables, funciones y clases que se utilizan en el proyecto.
- Archivos de código (.cpp): Contienen el desarrollo del código del proyecto.
- Archivos de interfaz de usuario (.ui): Son únicos de la herramienta de desarrollo **Qt Creator** y albergan los widgets y la disposición de la GUI en fórma de código.

La herramienta Qt Creator nos permite crear widgets y posicionarlos en una interfaz cómoda y visual. Además, permite nombrarlos y generar las señales que nos interesen de cada widget, lo que hace mucho más sencilla la gestión de la interfaz. Al ser el primer proyecto en Qt realizado se ha decidido utilizar esta herramienta por ser más intuitiva.

2.6 ARQUITECTURA DE LA FPGA

Las FPGA o dispositivos lógicos programables en campo son dispositivos electrónicos basados en semiconductores que pueden ser programados para contener cualquier circuito o sistema. Su principal característica es la capacidad para ser reprogramados en cualquier momento y de forma rápida para cambiar el circuito de su interior [1], [13], [14].

Funcionan con **bloques lógicos configurables** (CLB) que implementan funciones lógicas, lo que nos permite reproducir cualquier estructura de puertas lógicas. Hay de distintos tipos para desempeñar funciones distintas, por ejemplo, de lógica general, de memoria y multiplicadores. Estos bloques se estructuran en una matriz bidimensional que supera habitualmente los 100.000 bloques lógicos.

Los bloques se interconectan entre sí mediante **recursos de interconexión** programables que conectan las funciones lógicas formando una estructura mayor capaz de reproducir cualquier dispositivo electrónico complejo.

Finalmente, los **bloques de entrada/salida**, que se conectan al resto del circuito y permiten conectar la FPGA con elementos externos.

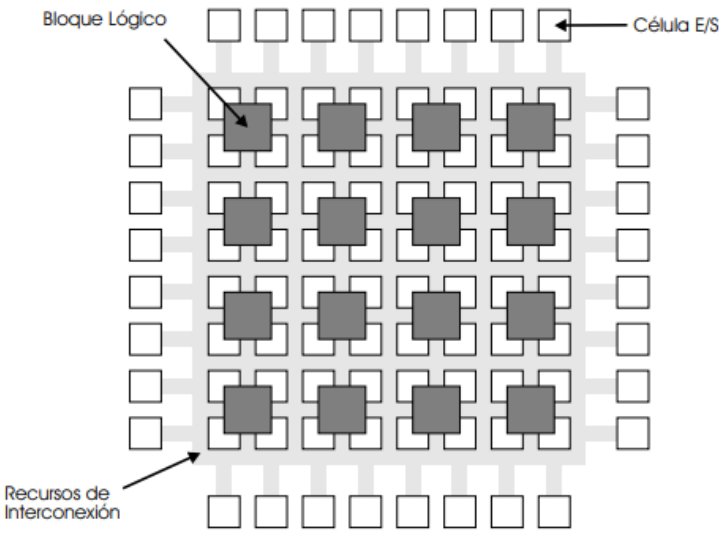

Figura 18 Estructura general de una FPGA [1]

La lógica de los bloques y las interconexiones solo se mantiene mientras la FPGA está alimentada, por lo que en muchos casos incorpora una memoria SRAM que almacena la configuración y la carga cada vez que encendemos la FPGA.

Para este proyecto empleamos una FPGA de **XILINX** modelo *Spartan 3E-100* [15] integrada en una placa *Basys 2* de **DILIGENT** [16].

El diseño interno de la FPGA y el protocolo de comunicación no forma parte de este trabajo, sino que ha sido realizado por otro equipo de trabajo que nos ha dado las especificaciones necesarias para que podamos comunicarnos con la FPGA desde el Teensy.

La FPGA se comportará como una máquina de estados, manteniéndose inicialmente en un estado inactivo, pero siempre atenta a las señales que le llegan. Cuando le lleguen instrucciones, pasará a un estado distinto que procesará las órdenes del Teensy y volverá al estado inicial cuando termine.

Para comunicarnos con la FPGA conectaremos seis cables a sus pines de entrada/salida que nos servirán para enviar las instrucciones y los datos a través de "*bit banging*", que consiste en controlar el valor (1 ó 0) de pines conectados entre sí y detectar los cambios que se producen para enviar mensajes.

Estos pines estarán divididos en dos pines de control llamados WrA y RdD que recibirán las instrucciones y cuatro pines de datos (D0 a D3) que intercambiarán cuatro bits cada vez. Además, la FPGA tendrá que estar

conectada a la tierra del Teensy a través de alguno de los pines GND. En la Figura 19 podemos ver representados los cables con los colores que hemos utilizado en realidad y el nombre que les corresponde.

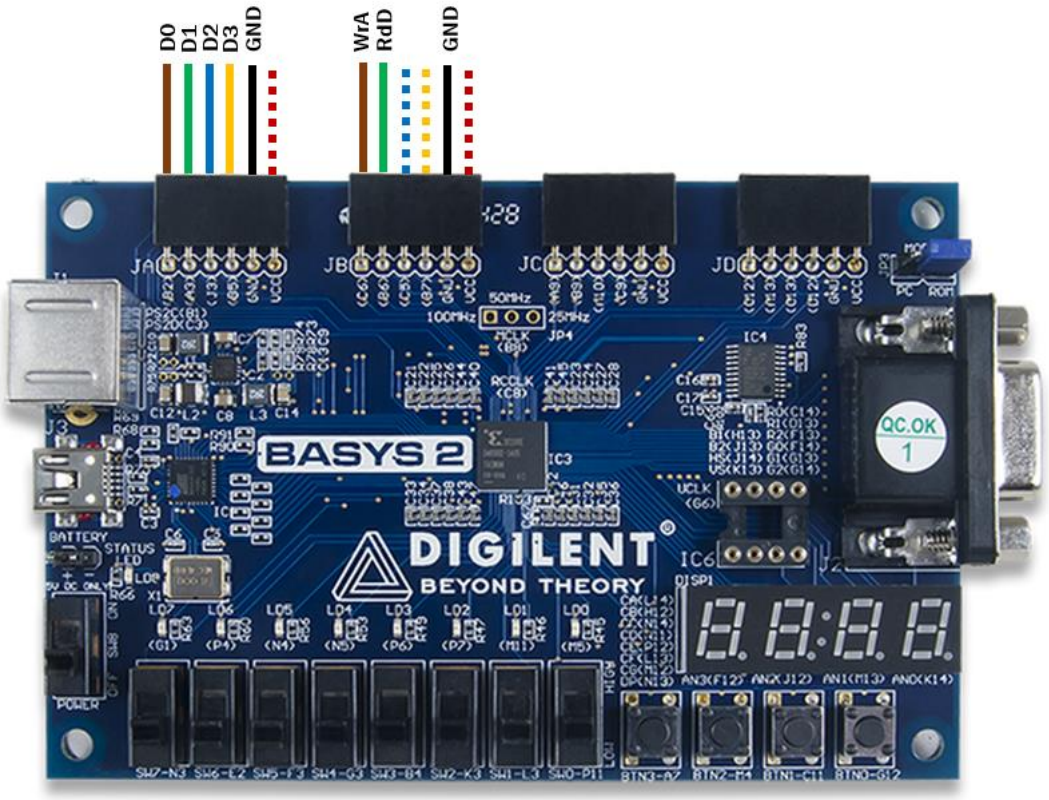

Figura 19 Conexiones a la FPGA

Como hemos visto, intercambiar datos con la FPGA cuando está en funcionamiento es esecialmente leer y escribir en espacios de memoria, por lo tanto, en nuestro caso está programada para que podamos leer y escribir una dirección de memoria y leer y escribir datos en la dirección que está guardada.

La dirección de memoria tendrá 32 bits, de los cuales los 8 más significativos corresponderan con el "*device*" o dispositivo y los otros 24 con la dirección dentro de ese dispositivo. Al ser el objetivo inicial establecer una comunicación fiable y robusta por ahora solo está programada con dos dispositivos:

- Un registro de 4 bits que enciende los LEDs del 0 al 3 (ver Figura 19) que corresponde con el device 0.
- Una memoria de 4 Kbytes que corresponde con el device 5.

La información se intercambiará en grupos de 4 bits, tomando el orden de los menos significativos a los más siginificativos.

Los pines **WrA** y **RdD** los controlará el Teensy y estarán normalmente a '1'. Cuando queramos hacer una escritura pasará a '0' primero **WrA** y después **RdD**, y el orden será el contrario en las lecturas. Una vez definido si es una lectura o escritura se enviarán pulsos de **WrA** para intercambiar bits de dirección o pulsos de **RdD** para intercambiar bits de datos, transfiriendo 4 bits por flanco ascendente y descendente. Para finalizar el acceso, ambas señales subirán a la vez y quedarán en el estado inactivo a '1'.

Representaremos el valor de los pines como una línea que sube al estado '1' y baja al estado '0', y los pines de datos como bloques que intercambian 4 bits a través de los pines **D0** a **D3**. Las cuatro instrucciones admitidas por la interfaz son las siguientes:

2.6.1 Escribir una dirección

Bajará primero **WrA** y después **RdD**. Para intercambiar bits cambiará de estado **WrA**. Por cada flanco de subida o bajada se escribirán 4 bits de la dirección, por lo que tendrá que realizar 4 pulsos de **WrA** para escribir la dirección completa. Finalmente, se detecta el fin de la escritucra cuando tanto **WrA** como **RdD** ascienden al estado inactivo '1'.

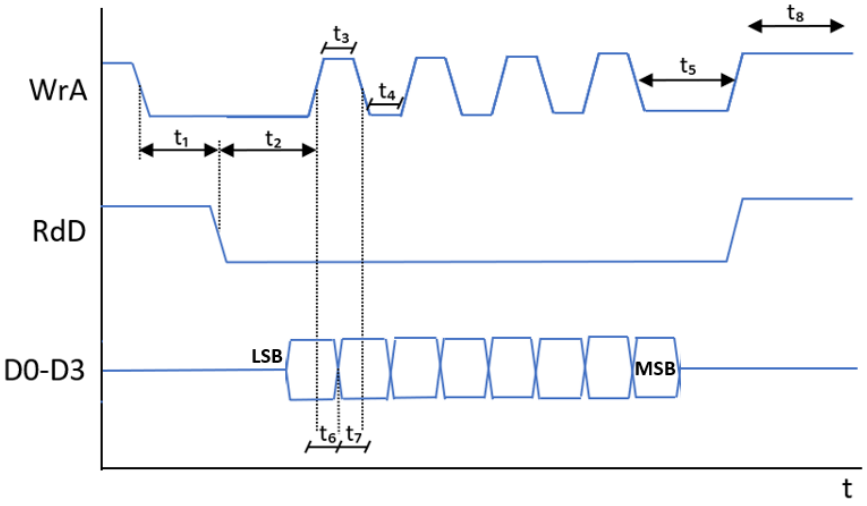

Figura 20 Bit banging de la escritura de dirección

2.6.2 Leer una dirección

Bajará primero **RdD** y después **WrA**. Para intercambiar bits cambiará de estado **WrA**. Por cada flanco de subida o bajada se leerán 4 bits de la dirección, por lo que tendrá que realizar 4 pulsos de **WrA** para leer la dirección completa. Finalmente, se detecta el fin de la lectura cuando tanto **WrA** como **RdD** ascienden al estado inactivo '1'.

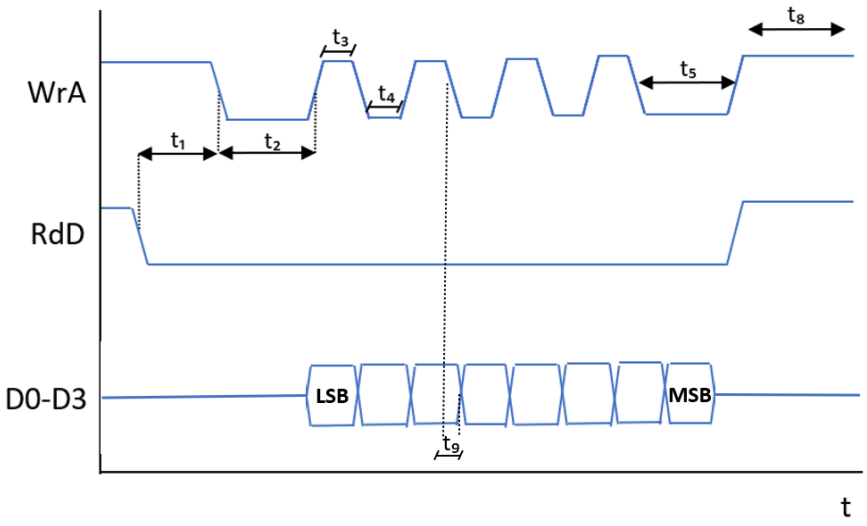

Figura 21 Bit banging de la lectura de dirección

2.6.3 Escribir datos

Bajará primero **WrA** y después **RdD**. Para intercambiar bits cambiará de estado **RdD**. Por cada flanco de subida o bajada se escribirán 4 bits de datos, por lo que se realizarán tantos pulsos de **RdD** como bytes a escribir. Finalmente, se detecta el fin de la escritucra cuando tanto **WrA** como **RdD** ascienden al estado inactivo '1'. La dirección se autoincrementa con cada byte escrito.

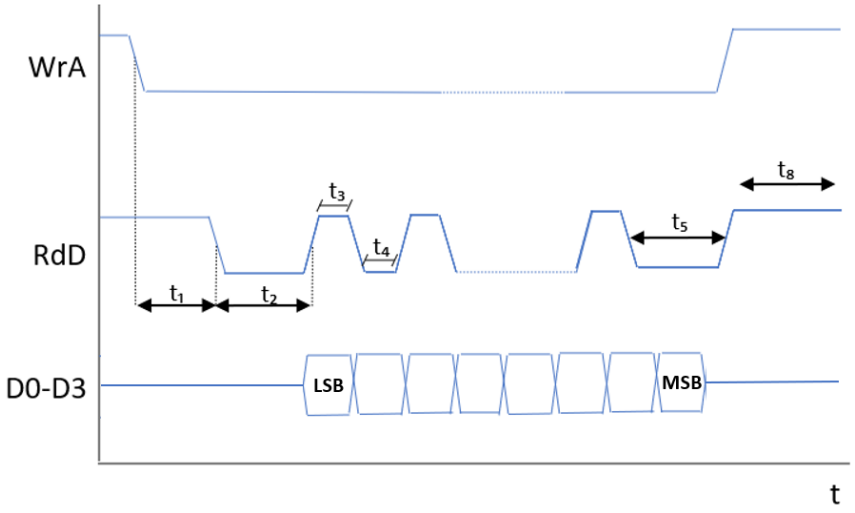

Figura 22 Bit banging de la escritura de datos

2.6.4 Leer datos

Bajará primero **RdD** y después **WrA**. Para intercambiar bits cambiará de estado **RdD**. Por cada flanco de subida o bajada se leerán 4 bits de datos, por lo que

se realizarán tantos pulsos de **RdD** como bytes a leer. Finalmente, se detecta el fin de la lectura cuando tanto **WrA** como **RdD** ascienden al estado inactivo '1'. La dirección se autoincrementa con cada byte leído.

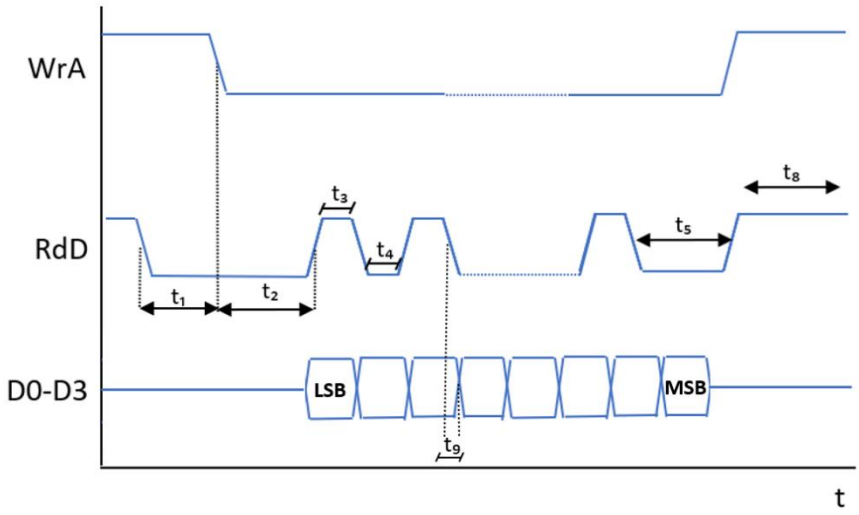

Figura 23 Bit banging de la lectura de datos

Para que la FPGA pueda procesar correctamente las instrucciones debemos respetar unos tiempos que nos vienen fijados cuando ejecutemos nuestras funciones. Podemos ver la duración mínima y máxima para los tiempos representados en las gráficas de las funciones tabulados en la Tabla 1.

Tabla 1 Tiempos de espera para la FPGA

	Tiempo mínimo (ns)	Tiempo máximo (ns)	Descripción
t_1	47	-	Tiempo entre flancos de bajada
t_2	47	-	Tiempo entre inicio e intercambio de datos
t_3	47	-	Tiempo tras un flanco de subida
t_4	47	-	Tiempo tras un flanco de bajada
t_5	47	-	Tiempo entre intercambio de datos y fin
t_6	15	-	Tiempo de *setup* de datos (estimado)
t_7	5	-	Tiempo de *hold* de datos (estimado)
t_8	63	-	Tiempo entre instrucciones
t_9	-	15	Tiempo de acceso a los datos (estimado)

Al estar programada para hacer verificaciones y comunicaciones robustas, de momento no hay límites máximos en las señales de la FPGA, lo que nos permite ser conservadores y emplear tiempos todo lo grandes que necesitemos. Cuando se añadan tiempos máximos de espera (*timeouts*) en líneas futuras del proyecto, se deberá ajustar el programa a esos tiempos o el mensaje dejará de ser escuchado.

CAPÍTULO 3 DESARROLLO DEL PROYECTO

Para la explicación del desarrollo del proyecto abordaremos primero la solución adoptada para la comunicación y los pasos previos necesarios. Despúes, dividiremos el proyecto entre el programa en PC, que funcionará como cliente, y el programa en el Teensy, que será el servidor.

3.1 ESTRUCTURA DE LA COMUNICACIÓN

Para poder enviar las cuatro instrucciones que nos permite la FPGA, de lectura y escritura de direcciones y datos, nuestro programa en el Teensy tendrá cuatro funciones que se comunicarán con la FPGA. Las funciones que manejan la dirección manejarán una variable de 32 bits como argumento, de la que leerán los bits de dirección y se los pasarán a la FPGA en caso de escritura, y al revés en caso de lectura. Para las funciones de datos recibirán un vector de bytes y una longitud de datos que se quieren leer o escibir y procederá de manera similar. Todas las funciones devolverán una variable booleana que nos indique si ha salido bien la comunicación o no.

Por otra parte, el Teensy debe recibir las instrucciones del PC y traducirlas a esas cuatro funciones. La comunicación se realizará a través del cable Ethernet con el protocolo TCP, por lo que utilizaremos las clases que nos ofrecen tanto Teensyduino como Qt para manejar esta capa de la comunicación. En nuestra capa de aplicación, enviaremos mensajes con la siguiente estructura:

- Tres bytes de control que corresponden a las letras “A”, “T” y una letra que identifica el tipo de instrucción; “W” para escritura (Write) y “R” para lectura (Read). Estos bytes de control nos permitirán detectar una nueva instrucción y su tipo.
- Cuatro bytes con la dirección en la que queremos leer o escirbir (un byte de device y tres de dirección dentro del mismo).
- Tres bytes con la longitud de los datos que estamos enviando o que queremos leer, que favorecen la uniformidad de la estructura y facilitan la interpretación del mensaje.
- Los bytes de datos en el caso de la escritura. Inicialmente se buscan tamaños de unos 100 Kb a 1Mb de datos máximos por mensaje, pero dependerán de las limitaciones del Teensy.

Los mensajes quedarían entonces como:

Escritura: *[A][T][W][dirección(4 bytes)][longitud(3 bytes)][datos(n bytes)]*

Lectura: *[A][T][R][dirección(4 bytes)][longitud(3 bytes)]*

Para seguir la misma lógica que la FPGA los valores de la dirección y la longitud se enviarán desde el byte menos significativo al más significativo y los datos que se reciban se interpretarán de la misma manera.

Para establecer la conexión y enviar los mensajes, el programa en Qt tendrá una clase que llamaremos *Teensy* para gestionar las instrucciones a través de diferentes funciones. La función de escritura recibirá una dirección, una longitud de datos y un vector de datos para componer el mensaje para el Teensy, que deberá escibir los datos que le enviamos en la dirección indicada. La función de lectura recibirá una dirección y una longitud de datos a leer; el Teensy leerá los datos de esa dirección y los enviará de vuelta al PC, que estará esperándolos. Ambas devolverán también una confirmación de que la comunicación ha salido bien.

Finalmente, una interfaz gráfica sencilla nos permitirá elegir si queremos leer o escribir, el device y la dirección en la que lo queremos hacer, y los datos que queremos escribir o la longitud que deseamos leer. De esta manera podremos verificar la funcionalidad y robustez de la comunicación, y fijaremos una base para estados más avanzados de la línea de trabajo en la que se incorpora este proyecto.

3.2 PASOS PREVIOS

Necesitamos dar una serie de pasos previos para poder establecer una comunicación efectiva entre el PC y el Teensy, que describiremos en este apartado.

3.2.1 Puerto Ethernet

Como ya hemos comentado en el capítulo 2.2, para conectar el Teensy al cable Ethernet hemos utilizado un puerto Ethernet que no se puede conectar directamente al Teensy. La separación y colocación de los huecos para los pines del Teensy de nuestro conector son distintos, por lo que hemos tenido que utilizar cables rígidos entre medias para poder conectarlo. En la Figura 24 y Figura 25 se etiquetan los pines que se conectan entre sí del Teensy al conector Ethernet.

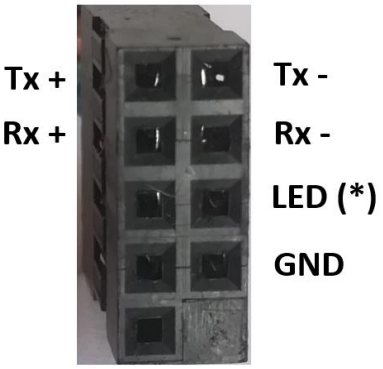

Figura 24 Conexiones conector Ethernet

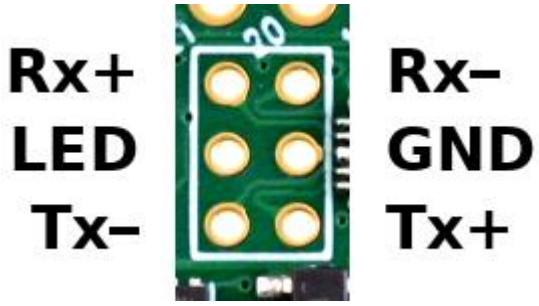

Figura 25 Conexiones Ethernet Teensy

(*)La conexión LED se conecta por uniformidad, pero no tiene ninguna funcionalidad para nuestro puerto Ethernet como ya hemos comentado en el capítulo 2.2.

La conexión final queda como ilustra la Figura 26.

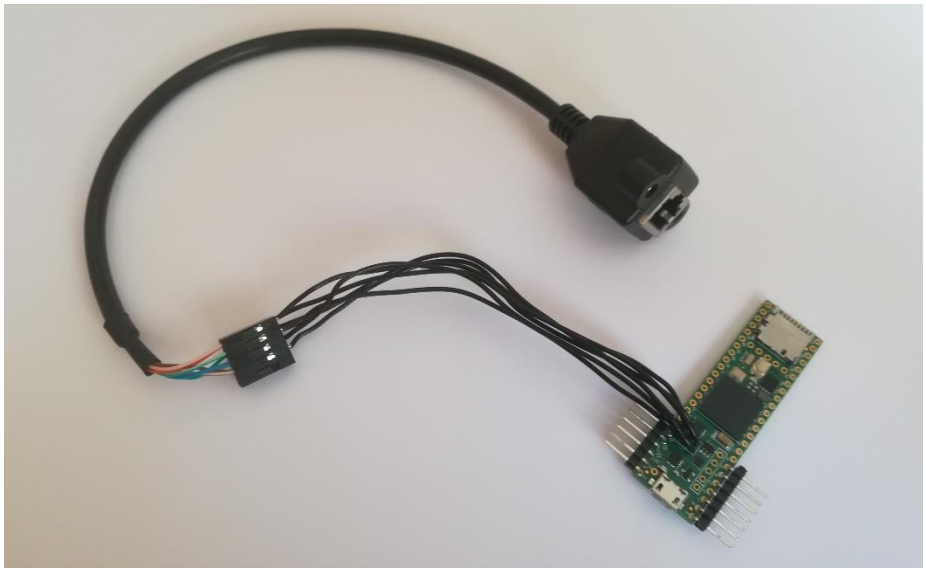

Figura 26 Conexión del puerto Ethernet al Teensy

3.2.2 Dirección MAC

Para poder establecer la comunicación Ethernet desde el Teensy necesitamos conocer su dirección MAC. Para ello utilizamos un fragmento de código que nos permite acceder a valores internos del microcontrolador y copiar esta dirección [25].

```
int mac[6];
for(uint8_t by=0; by<2; by++) mac[by]=(HW_OCOTP_MAC1 >> ((1-by)*8)) & 0xFF;
for(uint8_t by=0; by<4; by++) mac[by+2]=(HW_OCOTP_MAC0 >> ((3-by)*8)) & 0xFF;
Serial.printf("MAC: %02x:%02x:%02x:%02x:%02x:%02x\n",
              mac[0], mac[1], mac[2], mac[3], mac[4], mac[5]);
```

Figura 27 Fragmento de código para conocer la dirección MAC

Este fragmento de código extrae la dirección y la guarda en un vector de 6 bytes para después sacarla por pantalla como vemos en la Figura 28.

```
MAC: 04:e9:e5:0f:66:0d
Finalizado
```

Figura 28 Salida por pantalla de la dirección MAC del Teensy 4.1

Nos entrega la dirección MAC en hexadecimal con los bytes separados por dos puntos.

3.3.3 Direcciones IP

También debemos decidir las direcciones IP que se le asignarán tanto al Teensy como al PC. En nuestro caso, podemos dejar que sea el router el que asigne automáticamente las direcciones cuando se conecten los dispositivos, pero para ganar mayor control sobre el proceso y orientado hacia la conexión objetivo que será sin router, fijaremos nosotros estas direcciones.

En este caso la forma más sencilla de hacerlo es desde el propio router. Si accedemos a la primera dirección de la red (en este caso la 192.168.1.1) nos dirige a la interfaz del router y podemos modificar la dirección IP que asigna a cada dispositivo. Normalmente solo es necesario iniciar sesión con la contraseña del router y acceder a la configuración avanzada. Asignaremos al PC como dirección estática la dirección que ya le asignaba el router para asegurarnos de que no pertenece a ningún otro dispositivo. En este caso es la 192.168.1.150.

En el caso objetivo en el que no hay un router de por medio podemos hacer que el PC sea quien se asigne una dirección IP a sí mismo. Para ello, una vez conectado el cable Ethernet, debemos acceder a *Panel de control -> Redes y configuración -> Centro de redes y recursos compartidos* y hacer clic sobre la conexión Ethernet. Esto nos abrirá una pestaña de estado de Ethernet en la que deberemos clicar en el botón “propiedades”. Finalmente, buscaremos el protocolo de internet versión 4 (TCP/IP v4) y de nuevo clicaremos en sus propiedades. Debería aparecernos una ventana similar a la Figura 29 en la que podremos fijar los valores que necesitamos.

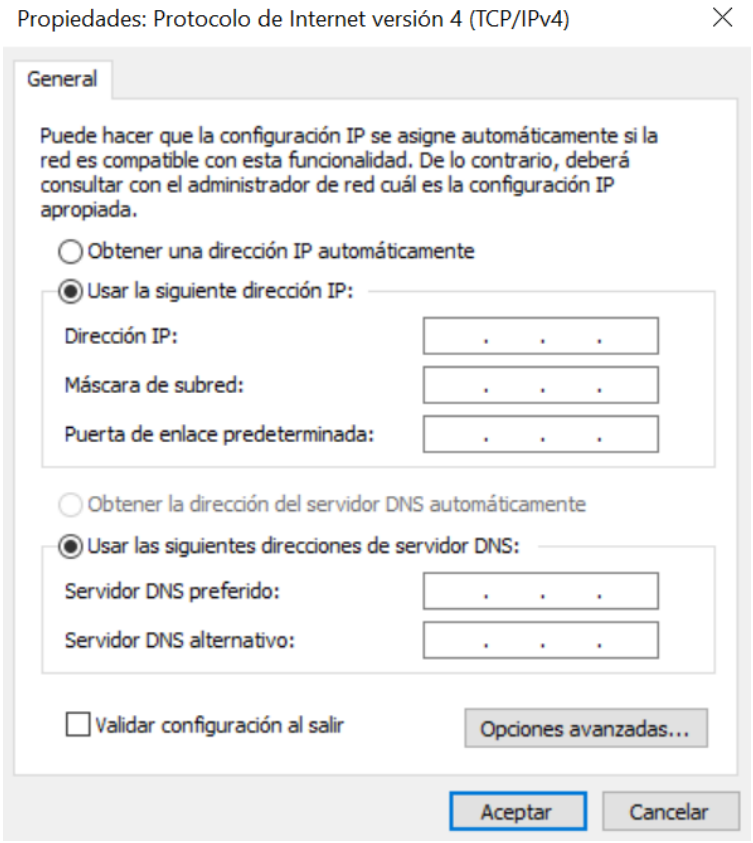

Figura 29 Propiedades de la dirección IP v4 Windows 10

En esta ventanda podremos fijar la dirección IP que queremos que el PC se asigne, así como la máscara de subred y la puerta de enlace (debemos dejar las predeterminadas).

La máscara de esta red local es la 255.255.255.0, la puerta de enlace la 192.168.1.1 y el servidor DNS el 192.168.1.1. No son relevantes ya que vamos a funcionar siempre dentro de la red local, pero de nuevo las fijaremos para tener control sobre el proceso.

En el caso del Teensy, fijaremos la dirección IP que se fija él mismo por defecto, la 192.168.1.177. Se ha verificado la validez de la IP y el buen funcionamiento de la conexión Ethernet empleando los ejemplos que incorpora Teensyduino sobre la clase NativeEthernet, en particular *WebServer* y *WebClient*, a los que se puede acceder desde el entorno de Arduino como aparece en la Figura 30.

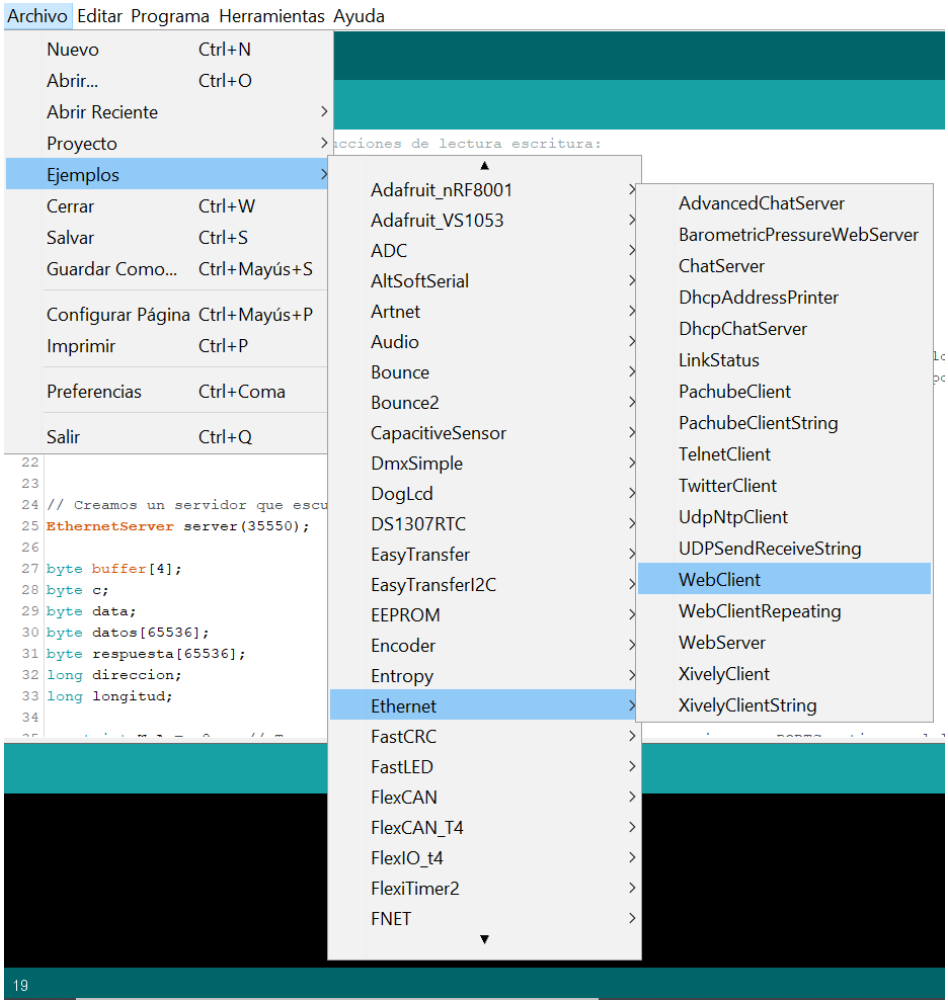

Figura 30 Ejemplos Ethernet Teensyduino

3.3.4 Puerto de enlace

Tanto el Teensy como el PC deben de utilizar el mismo puerto virtual por el que intercambian información para que puedan comunicarse. En este caso se ha elegido el puerto 35550, por ser del orden de los puertos que elige el PC al ejecutar los ejemplos de Qt que emplean las clases de comunicación Ethernet, *fortuneclient* y *fortuneserver*, que podemos encontrar en la carpeta *Examples -> network* de Qt.

3.3.5 Codificación de caracteres

Aunque entre el Teensy y el PC solo se van a intercambiar bytes, para nosotros puede que quieran representar caracteres y por lo tanto tienen que poder codificarlos en el mismo lenguaje.

Arduino utiliza por defecto el lenguaje de codificación UTF-8, por lo que mantendremos ese lenguaje y utilizaremos la clase de Qt *QTextStream*, que

traduce correctamente los caracteres a este estándar y gestiona también las lecturas independientemente del lenguaje utilizado.

3.4 CLIENTE EN Qt

Nuestro programa en Qt se estructura alrededor de la clase *Teensy* que hemos creado, que envía los mensajes al microcontrolador. Para crearla y poder gestionar la conexión TCP utilizamos la clase *QTcpSocket* que nos aporta Qt, que permite crear un objeto que gestiona la entrada y salida de mensajes TCP (socket).

Para crear un proyecto en el que podamos diseñar una interfaz de usuario tipo ventana con la clase *QWidget* creamos un proyecto nuevo desde *File -> New Project...* y nos aparece una ventana similar a la Figura 31.

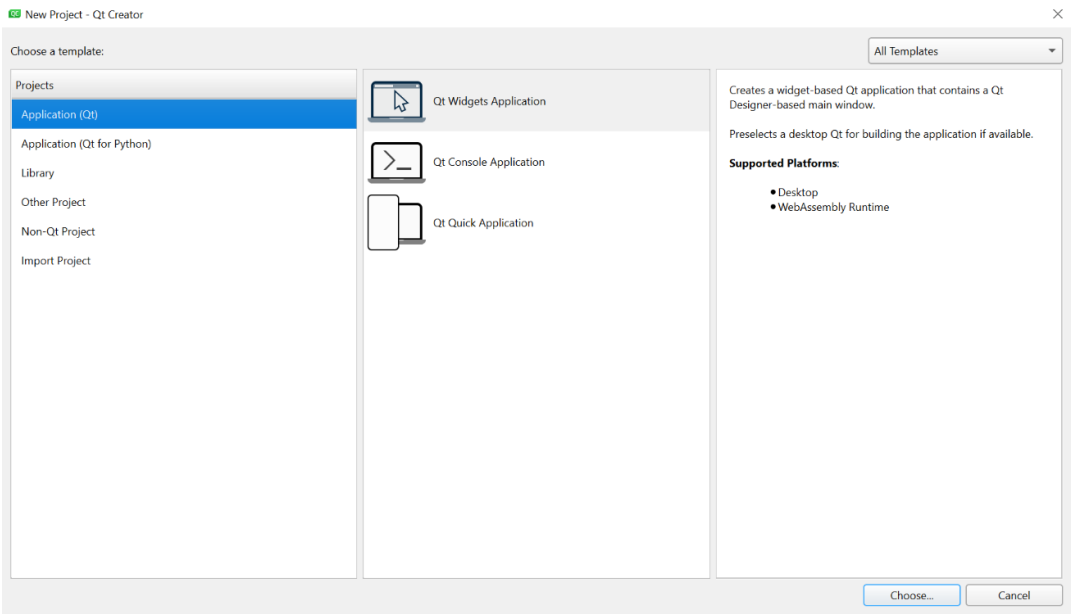

Figura 31 Nuevo proyecto Qt Creator

En esta ventana seleccionamos un proyecto tipo aplicación y con una interfaz de usuario (*Qt Widgets Application*). Deberemos definir el nombre y la localización del proyecto, así como el sistema de compilación (*built system*), del que debemos escoger **qmake**.

Escogeremos ahora el tipo de clase que queremos crear para manejar la interfaz de usuario, en este caso *QWidget*. Dejamos el resto de configuraciones por defecto y finalizamos.

Antes de empezar el proyecto debemos añadir los módulos de Qt que vamos a utilizar en el **archivo de proyecto** (.pro), en nuestro caso hemos utilizado **core**

para las clases de la aplicación, **gui** para las de la interfaz de usuario y **network** para el resto.

3.4.1 Clase Teensy

El proceso para crear una nueva clase en el proyecto es similar. En este caso debemos crear un archivo nuevo desde *File -> New Project...* como en la Figura 32.

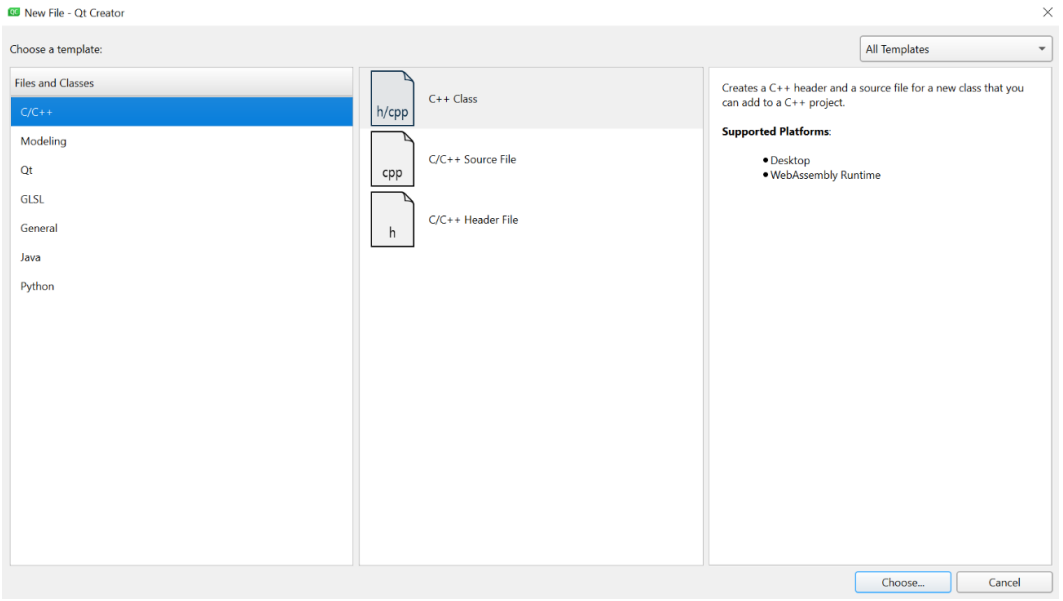

Figura 32 Nuevo archivo Qt Creator

Escogemos una clase de C++ con un archivo de cabecera y uno de código. Para esta clase escogeremos que se base en *QObject* y añadiremos la macro de **Q_OBJECT** que nos permite utilizar signals y slots como explicamos en el capítulo 2.5.2. Qt nos crea los archivos, la clase y el constructor, e incluye la herencia de *QObject*.

En el **archivo de cabecera** incluimos la clase *QTcpSocket* y declaramos las siguientes funciones como públicas:

```
bool read(const quint32 address, const quint32 length, QByteArray &data);
bool write(quint32 address, QByteArray data);

bool read(const quint32 address, const quint32 length, QString &data);
bool write(const quint32 address,  const QString data);

bool connection();
bool disconnection();
```

Figura 33 Funciones de la clase Teensy

Las funciones de lectura tienen como argumentos la dirección de la que leer y la longitud de datos, así como la variable donde se devolverán los datos pasada por referencia. Las de escritura tienen solo la dirección y los datos a escribir.

Ambas funciones están definidas para recibir los datos como vectores de bytes (QByteArray) o como cadenas de caracteres (QString), y todas devuelven una confirmación de la acción como variable booleana. La función de escritura de vectores de bytes necesita argumentos que no sean constantes para poder gestionar mensajes grandes.

Además, declaramos también las funciones para conectar y desconectar nuestro programa del PC delTeensy remoto.

En la parte privada de la clase creamos un puntero al socket que utilizaremos para gestionar la comunicación y declaramos una variable de control que utilizaremos para esperar la respuesta en una lectura.

```
private:
    QTcpSocket *mSocket; // Puntero para el socket
    bool respondido;     // Señal de control
```

Figura 34 Variables privadas de la clase Teensy

En el **archivo de código** incluimos también las clases *QDataStream*, *QTextStream* y *QNetwork*, que vamos a utilizar para manejar vectores de bytes, cadenas de caracteres y eventos en la interfaz, respectivamente.

En el **constructor** de la clase creamos un nuevo objeto de la clase *QTcpSocket* y se lo asignamos al puntero que habíamos creado. Además, en esta aplicación queremos que nada más crearse el socket se establezca la conexión, así que llamamos a la función que lo conecta al Teensy.

```
Teensy::Teensy(QObject *parent)
    : QObject{parent}
{
    mSocket = new QTcpSocket(this);
    Teensy::connection();

    connect(mSocket, &QTcpSocket::readyRead, [&]{
        respondido = true;
    });

}
```

Figura 35 Definición del constructor de la clase Teensy

También podemos ver que conectamos la señal de nuestro socket "*readyRead*", que se emite cuando llega un nuevo mensaje al socket, a una función definida en línea que simplemente cambia nuestra variable de control "respondido" a verdadero. Así podemos detectar cuando llega una respuesta.

Las **funciones de conexión y desconexión** utilizan funciones de la clase *QTcpSocket*, como podemos ver en la Figura 36.

```
bool Teensy::connection(){
    mSocket->connectToHost("192.168.1.177", 35550);
    return(mSocket->waitForConnected());
}

bool Teensy::disconnection(){
    mSocket->disconnectFromHost();
    return(true);
}
```

Figura 36 Funciones de conexión y desconexión de la clase Teensy

La función "*connectToHost*" intenta conectar a un servidor TCP con una IP y un puerto determinados, en este caso los del Teensy. Con "*waitForConnected*" nos aseguramos de que esperamos a que el socket esté conectado y devolvemos una confirmación de si ha sido capaz o no de conectarse. Para la desconexión, la función "*disconnectFromHost*" espera a que el socket se vacíe para desconectarlo, por lo que no hace falta ninguna otra función.

La **función de lectura** definida para una cadena de caracteres llama primero a la definición principal de la función, que nos entrega un vector de bytes, y después aprovecha la clase *QTextStream* para para convertir ese vector en una cadena de caracteres.

```
bool Teensy::read( const quint32 address, const quint32 length, QString &String_data){
    QByteArray ByteArray_data;
    bool lectura_ok = read(address, length, ByteArray_data);

    QTextStream TextStream_data(&String_data, QIODevice::WriteOnly);
    TextStream_data << ByteArray_data;

    return lectura_ok;

}
```

Figura 37 Función de lectura para cadenas de caracteres de la clase Teensy

Definiendo un objeto de la clase (en nuestro caso lo hemos llamado "*TextStream data*") que maneje la variable que almacena la cadena de caracteres, podemos simplemente utilizar el operador "<<" para convertir

nuestro vector "*ByteArray data*" en la cadena "*String data*". Mantenemos la variable booleana para comprobar que se ha ejecutado correctamente.

La definición de la función de lectura para un vector de bytes comienza poniendo la variable de control de la respuesta a "falso" para poder saber cuando llega una respuesta nueva, y comprobando que la longitud que se pide no es mayor a 3 bytes.

```
bool Teensy::read( const quint32 address, const quint32 length, QByteArray &data){
    respondido =false;

    if(length > 0x00FFFFFF){
        return false;

    }else{
        QByteArray mensaje;
        QDataStream out(&mensaje, QIODevice::WriteOnly);
        out << char('A') << char('T') << char('R');
        out << quint8(address & 0xFF) << quint8((address >> 8) & 0xFF)
            << quint8((address >> 16) & 0xFF) << quint8((address >> 24) & 0xFF);
        out << quint8(length & 0xFF) << quint8((length >> 8) & 0xFF)
            << quint8((length >> 16) & 0xFF);
        mSocket->write(mensaje);
        mSocket->flush();

        while (!respondido){
            QCoreApplication::processEvents(QEventLoop::AllEvents, 3000);
        }

        data = mSocket->read(length);
        while(data.length() < length){
            data.append(mSocket->read(length-data.length()));
            QCoreApplication::processEvents(QEventLoop::AllEvents, 3000);
        }

        return true;
    }
}
```

Figura 38 Función de lectura para un vector de bytes de la clase Teensy

Para componer el mensaje utiliza la clase *QDataStream*, que funciona de forma similar a *QTextStream*. Primero creamos un vector de bytes ("mensaje") y se lo asignamos a un objeto de la clase ("*out*"), en este caso con la condición de que solo vamos a escribir en él.

Con el operador "<<" metemos en nuestro vector primero los bytes de control, asegurándonos que entran como caracteres de 8 bits. Después vamos a ir metiendo bytes de la dirección y la longitud desde el menos significativo al más significativo, empleando el "*and*" lógico (&) para coger solo un byte y el operador de desplazamiento de bits (>>) para desplazar los bytes y coger el que nos interesa. Introducimos la dirección de 4 bytes y 3 bytes de la longitud (ya que hemos comprobado que no es mayor de 3 bytes).

Una vez compuesto el mensaje se utilizan las funciones "*write*" y "*flush*" de la clase *QTcpSocket* para ordenar que se envíe el mensaje y esperar a que se envíe, respectivamente.

En este punto hemos decidido que la función se quedará esperando a la respuesta (metodología "*blocking*"), por lo que tenemos un bucle que ejecuta la función "*processEvents*" de *QCoreApplication*, que permite a la aplicación ejecutar otros eventos de otras aplicaciones (por ejemplo, movimiento, interacción o cierre de la interfaz de usuario) para que no se quede "congelada" mientas espera a la respuesta. Cuando llega la respuesta, cambia la variable "respondido" y sale del bucle.

Las funciones del Teensy tienen una limitación para enviar mensajes de 1460 bytes, por lo que tendremos que fragmentar la respuesta y enviarla al PC en varios paquetes. Desde Qt intentamos leer la longitud pedida con la función "*read*" de la clase *QTcpSocket*. Con la función "*length*" comprobamos cuántos bytes llevamos leídos y, si son menos de los totales a leer, intentamos leer los que nos faltan y los concatenamos con la función "*append*". Para esperar a recibir nuevos paquetes volvemos a llamar a la función "*processEvents*".

Una vez hemos recibido la respuesta completa volvemos de la llamada de la función, confirmando que hemos leído los datos.

Ahora definiremos de forma análoga las funciónes de escritura.

```
bool Teensy::write( const quint32 address, const QString String_data){
    QByteArray ByteArray_data;
    QTextStream TextStream_data(&ByteArray_data, QIODevice::WriteOnly);
    TextStream_data << String_data;
    TextStream_data.flush();

    return(write(address, ByteArray_data));
}
```

Figura 39 Función de escritura para cadenas de caracteres de la clase Teensy

En este caso el paso es de una cadena de caracteres a un vector de bytes, con la diferencia de que la función principal se llama después de hacer el cambio y por lo tanto debemos llamar antes a la función "*flush*" de *QTextStream*, ya que necesitamos que se haga la transformación sin esperar a volver de la función.

En la función para el vector de bytes nos encontramos inicialmente con una limitación del tamaño de los mensajes. El Teensy no recibe correctamente mensajes de más de 12 Kbytes, por lo que debemos fragmentar nuestros mensajes a este tamaño.

```
bool Teensy::write(quint32 address, QByteArray data){

    while(data.length() > 12000){
        QByteArray data_frag = data.first(12000);
        if(!Teensy::write(address,data_frag)){
            return false;
        }
        address += 12000;
        data.remove(0, 12000);
    }

    QByteArray mensaje;
    QDataStream DataStream_out(&mensaje, QIODevice::WriteOnly);

    quint32 length =data.length();
    DataStream_out << char('A') << char('T') << char('W');
    DataStream_out << quint8(address & 0xFF) << quint8((address >> 8) & 0xFF)
                   << quint8((address >> 16) & 0xFF) << quint8((address >> 24) & 0xFF);
    DataStream_out << quint8(length & 0xFF) << quint8((length >> 8) & 0xFF)
                   << quint8((length >> 16) & 0xFF);

    mensaje.append(data);
    mSocket->write(mensaje);
    mSocket->flush();

    return true;
}
```

Figura 40 Función de escritura para un vector de bytes de la clase Teensy

Comenzamos la función comprobando si el mensaje ocupa más de 12 Kbytes con la función "*length*" del vector de bytes. Si es el caso, tomamos los primeros 12 Kbytes con la función "*first*" y los almacenamos en "*data frag*" para llamar de nuevo a la función de escritura. En este caso "*data frag*" cumple con el tamaño adecuado, por lo que no volverá a entrar en el bucle.

Tras cada fragmentación, incrementamos las posiciones necesarias en la dirección y elminamos los datos que ya se han enviado con la función "*remove*", por lo que quedarían ambas variables listas para continuar con la escritura en el siguiente mensaje. Esto nos permite enviar los bytes que queramos en una sola petición y que sea la clase la que se encargue de gestionar las limitaciones del Teensy.

El resto de la función compone un mensaje similar al de la lectura y concatena el vector de datos al final de este con la función "append" como vemos en la Figura 40. En este caso no hace falta esperar a la respuesta.

3.4.2 Interfaz de usuario

Si vamos a la barra de herramientas vertical de la derecha en el entrono de desarrollo de Qt podemos ver un apartado que se denomina "*Design*". En él, hemos creado nuestro diseño para la interfaz de usuario que se puede ver esquematizado en la Figura 41.

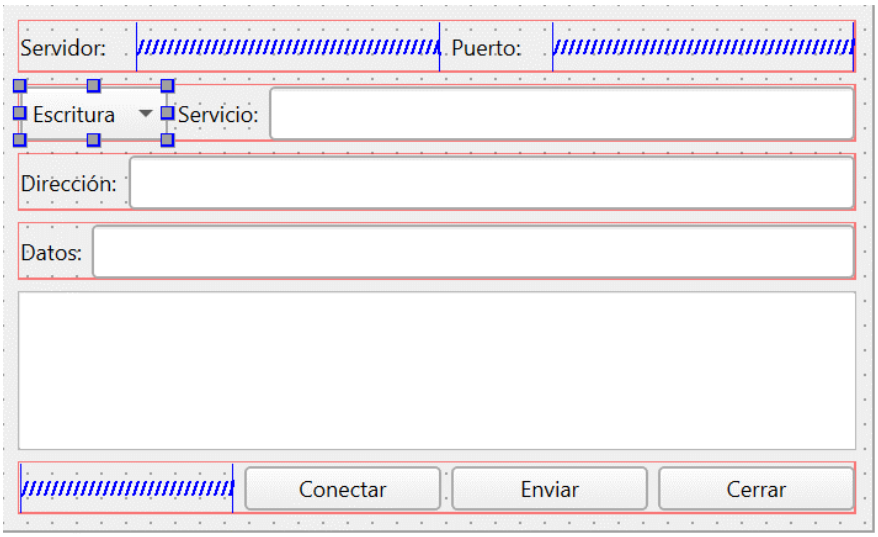

Figura 41 Esquema de diseño de la interfaz de usuario

Para crear este diseño, hemos ajustado el tamaño de ventana que queríamos y hemos ido colocando los widgets que nos interesaban, para finalmente agruparlos en bloques y darles la disposición final o "*layout*". Cada widget tendrá un nombre que nos permitirá controlarlos desde la clase *widget*.

La ventana tiene unas etiquetas en la parte superior que señalan el servidor y el puerto y que se rellenarán con la IP del servidor y el número del puerto elegido cuando hagamos clic en el botón de conectar.

Debajo encontramos una caja de selección que nos permite elegir entre el modo escritura y lectura y que se llamará "modo". A su lado tenemos tres etiquetas y tres cajas de texto homónimas en las que podremos escribir el servicio al que queremos acceder, la dirección y los datos. Cuando cambiemos al modo lectura querremos que la etiqueta de "Datos:" pase a decir "Longitud:".

A continuación tenemos una lista, que llamaremos "*listWidget*", en la que iremos informando al usuario de las acciones que realiza y en la que se recibirá la respuesta del Teensy.

Finalmente, tres botones que nos permitirán conectar al Teensy, enviar la petición de lectura o escritura o cerrar la aplicación.

La apariencia final de la ventana en los distintos modos sería la siguiente:

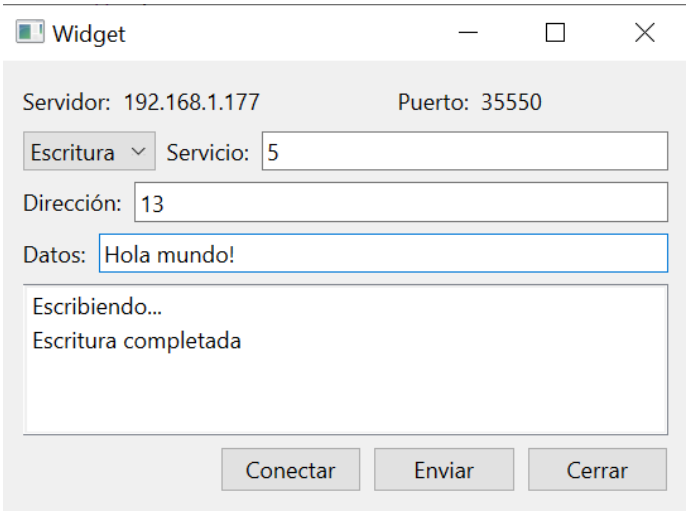

Figura 42 Interfaz de usuario para el modo escritura

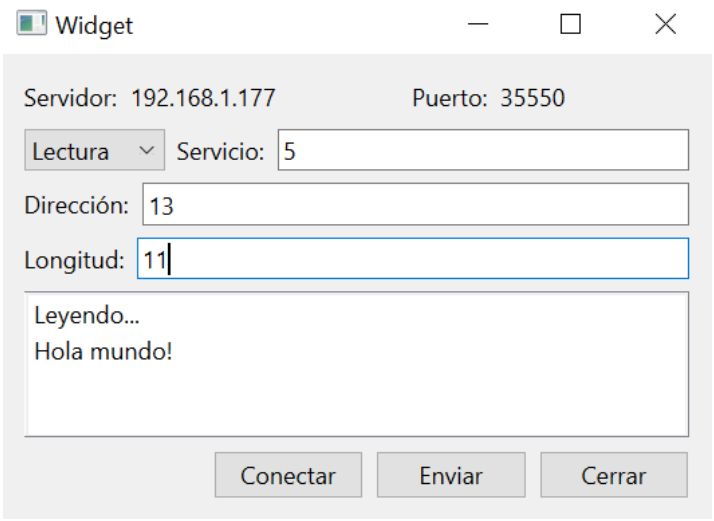

Figura 43 Interfaz de usuario para el modo lectura

Desde la herramienta de diseño podemos generar automáticamente los slots de la clase widget conectados a las señales de nuestros widgets que nos interesen. En nuestro caso hemos generado los siguientes:

- Clic en el botón conectar (*on_conectar_clicked)*
- Clic en el botón enviar (*on_enviar_clicked*)
- Clic en el botón cerrar (*on_cerrar_clicked*)
- Cambio de modo en la caja de selección (*on_modo_currentTextChanged*)

3.4.3 Clase widget

Esta clase será la encargada de recibir las señales de la interfaz de usuario y llamar a las funciones de la clase *Teensy*.

En su **archivo de cabecera** tenemos incluida la clase *QWidget* y el espacio de nombres de la interfaz de usuario. Debemos declarar aquí la clase *Teensy* para poder utilizar las funciones de esta. La clase añade por definición el constructor y la macro de objetos.

Al generar los slots desde la interfaz de usuario aparecen aquí declarados como privados. El entorno no recomienda utilizar los nombres que se generan automaticamente, sin embargo al ser pocos y descriptivos en este caso hemos optado por mantenerlos.

```
private slots:
    void on_conectar_clicked();

    void on_cerrar_clicked();

    void on_enviar_clicked();

    void on_modo_currentTextChanged(const QString &modo);

private:
    Ui::Widget *ui;
    Teensy *teensy4_1;
```

Figura 44 Variables y slots privados de la clase Widget

Los slots de los botones no tienen argumentos, pero el slot de la caja de selección pasa como argumento el modo al que se ha cambiado, lo que nos ayudará a detectar en que modo estamos.

Finalmente se crea un puntero para manejar la interfaz de usuario y otro para manejar un objeto de la clase *Teensy* al que hemos llamado “teensy4_1”.

En el **archivo de código** comienza declarando el constructor, que configura la interfaz de usuario y el destructor de la clase. Después definimos los slots que hemos creado.

```
void Widget::on_conectar_clicked()
{
    teensy4_1 = new Teensy(this);
    ui->servidor->setText("192.168.1.177");
    ui->puerto->setText("35550");
}


void Widget::on_cerrar_clicked()
{
    teensy4_1->disconnection();
    close();
}
```

Figura 45 Definición de los slots "conectar" y "cerrar" de la clase Widget

El slot del botón "conectar" asigna un nuevo objeto de la clase *Teensy* a nuestro puntero, lo cual establece la conexión con el servidor, por lo que lo mostramos en la interfaz escribiendo la dirección IP del servidor y el puerto al que nos estamos conectando con la función "*setText*" de los widgets.

En el slot de "cerrar" llamamos a la función "*disconnection*" de la clase *Teensy* para desconectarnos del servidor y después a "*close*", que cierra la ventana de la interfaz de usuario.

El slot de "enviar" es más complejo ya que tiene que gestionar los modos de lectura y escritura como vemos en la Figura 46.

```
void Widget::on_enviar_clicked()
{
    if(ui->modo->currentText() == "Lectura"){
        ui->listWidget->addItem("Leyendo...");
        quint32 direccion_completa = (ui->servicio->text().toInt() << 24)
                + ui->direccion->text().toInt();
        QString data;
        if(!teensy4_1->read(direccion_completa, ui->datos->text().toInt(), data)){
            ui->listWidget->addItem("Error en la lectura");
        }else{
            ui->listWidget->addItem(data);

            ui->servicio->clear();
            ui->direccion->clear();
            ui->datos->clear();
        }

    }else{
        ui->listWidget->addItem("Escribiendo...");
        quint32 direccion_completa = (ui->servicio->text().toInt() << 24)
                + ui->direccion->text().toInt();
        if(!teensy4_1->write(direccion_completa, ui->datos->text())){
            ui->listWidget->addItem("Error en la escritura");
        }else{
            ui->listWidget->addItem("Escritura completada");

            ui->servicio->clear();
            ui->direccion->clear();
            ui->datos->clear();
        }

    }
}
```

Figura 46 Definición del slot "enviar" de la clase Widget

Para poder saber si tenemos que hacer una lectura o una escritura comprobamos el modo en el que se encuentra la caja de selección con la función "*currentText*".

Si se encuentra en "Lectura", enviamos un mensaje de "Leyendo..." a nuestra "*listWidget*" con la función "*addItem*". Para componer la dirección completa y llamar a la función de lectura tenemos que transformar las cadenas de caracteres que lee la función "*text*" en números enteros con la función "*toInt*", para después desplazar el número de device al byte más significativo y sumarlo a la dirección.

Con la dirección completa, la longitud a leer que está escrita en el widget "datos" y una cadena de caracteres en la que queremos recibir la respuesta llamamos a la función "*read*" de la clase *Teensy*. Si falla, sacamos un mensaje de error por la interfaz, y si la lectura es correcta sacamos la respuesta que hayamos recibido del Teensy y borramos la entrada que había introducido el usuario con la función "*clear*" sobre los widgets de entradas.

En el caso de escritura, la estructura es idéntica, con las únicas diferencias de que en el widget de "datos" se lee el mensaje en forma de cadena de caracteres y es directamente el argumento de la función "*write*"; y que, al no recibir una respuesta cuando se han enviado correctamente los datos, se pone simplemente un mensaje de "Escritura completada" para confirmar al usuario el envío.

Finalmente, para cambiar el texto de la etiqueta de datos cuando cambia el modo utilizamos una estructura similar con el argumento que le pasa la llamada de la señal ("modo") y la función "*setText*", como podemos ver en la Figura 47.

```
void Widget::on_modo_currentTextChanged(const QString &modo)
{
    if(modo == "Lectura"){
        ui->etiqueta_datos->setText("Longitud:");
    }else{
        ui->etiqueta_datos->setText("Datos:");
    }

}
```

Figura 47 Definición del slot "modo" de la clase Widget

3.4.4 Verificación

Para verificar el código, corregir errores y comprobar las funciones que hemos creado, hemos utilizado la clase de Qt *QDebug*, que permite sacar por la consola del programa mensajes de control de las diferentes partes del código. Para ello es necesario estar desarrollando una versión de "*Debug*", que se puede seleccionar en la pestaña de la barra de herramientas izquierda en la que pone el nombre del proyecto.

Esta clase tiene la función "*qDebug*", que nos permite utilizar el operador "<<" para enviar mensajes o valores de variables a la consola de debug como podemos ver en el ejemplo de la Figura 48.

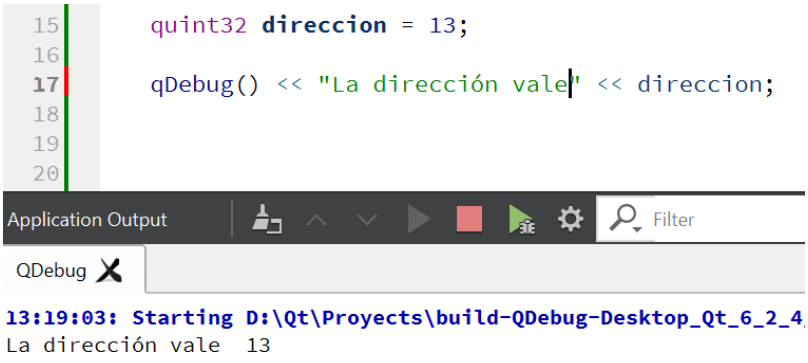

Figura 48 Ejemplo clase QDebug

3.5 SERVIDOR CON TEENSYDUINO

El programa para el Teensy utiliza la librería que incorpora el software de Teensyduino **NativeEthernet**, desarrollada por un usuario e incorporada como parte del producto por el fabricante del microcontrolador [26]. Para la comunicación con la FPGA utiliza sus pines de entrada y salida a base de "*bit banging*" como explicamos en el capítulo 2.6.

Para crear un nuevo proyecto en Arduino pinchamos en la pestaña de *Archivo -> Nuevo*. Los proyectos en Arduino se estructuran como una función "*setup*" que se ejecuta al alimentar al Teensy con corriente, en la que configuramos los elementos que queramos utilizar y hacemos las comprobaciones previas necesarias; y la función "*loop*", un bucle que se repite constantemente mientras el microcontrolador esté conectado.

Nuestro proyecto se dividirá en las declaraciones previas, la función setup, la función loop y las funciones de lectura y escritura en la FPGA.

3.5.1 Declaraciones previas

Lo primero que debemos hacer es incluir el archivo de cabecera de la librería, *NativeEthernet.h*, para poder utilizar las clases que contiene. Después, declaramos las diferentes variables que vamos a utilizar para la conexión TCP, la gestión de mensajes y el bit banging.

```
// Definimos los parámetros necesarios para la conexión TCP/IP con Ethernet:
byte mac[] = {0x04, 0xE9, 0xE5, 0x0F, 0x66, 0x0D}; // MAC del Teensy 4.1
IPAddress ip(192, 168, 1, 177);     // IP del servidor en el Teensy
IPAddress dnServer(192, 168, 1, 1); // IP del servidor DNS
IPAddress gateway(192, 168, 1, 1);  // IP de la puerta de enlace
IPAddress subnet(255, 255, 255, 0); // IP de la máscara de subred

// Creamos un servidor que escucha en el puerto 35550
EthernetServer server(35550);
```

Figura 49 Declaración de las direcciones necesarias para el servidor

Lo primero que declaramos es la dirección MAC de nuestro microcontrolador en forma de vector de bytes y después las direcciones IP del servidor, del servidor DNS, la puerta de enlace y la máscara de subred, de las que hablamos en el capítulo 3.2 de pasos previos. Para declararlas aprovechamos el tipo de variable "*IPAddress*" que nos permite definir direcciones IP de 4 bytes.

Finalmente creamos un objeto de la clase *EthernetServer* que llamaremos "*server*", que será el servidor encargado de escuchar los mensajes del cliente gracias a un socket por el puerto que hemos designado.

A continuación, declaramos las variables que vamos a utilizar para el loop, que podemos ver en la Figura 50.

```
byte buffer[4];
byte c;
byte data;
byte datos[12000];
byte respuesta[100000];
byte envio[1460];
ulong bytes_por_enviar;
ulong direccion;
ulong comprobar_direccion;
ulong longitud;
```

Figura 50 Declaración de las variables de gestión de mensajes en el Teensy

Las variables son las siguientes:

- Un vector de 4 bytes llamado "*buffer*" que sirve de registro intermedio para la dirección y la longitud.
- Una variable tipo byte llamada "c" para leer byte a byte la información que va llegando hasta encontrar los caracteres de control "A" y "T".
- Otra variable tipo byte llamada "*data*" para gestionar los envíos a la FPGA.
- Un vector de bytes de llamado "datos" que almacenan los datos que se escriben en la FPGA. Puede almacenar 12 Kbytes de datos ya que es el máximo que puede recibir la librería **NativeEthernet**.
- Un vector de bytes llamado "respuesta" que almacena hasta 100 Kbytes de datos que se leen de la FPGA. Este es el límite de datos que hemos decidido que se puedan leer de una vez ya que es el óptimo en cuanto a desempeño.
- Un vector de bytes llamado "envío" de 1460 bytes ya que es el máximo que puede enviar la librería.
- Una variable de 32 bits llamada "bytes por enviar" que utilizaremos para dividir los datos de lectura en bloques de 1460 bytes.
- Tres variables de 32 bits para almacenar la dirección, la comprobación de la dirección y la longitud.

Debemos ahora decidir qué pines del Teensy se van a conectar a la FPGA. Es necesario que los pines se puedan utilizar a la vez que Ethernet, lo cual no será un problema ya que este tiene sus pines propios en el centro de la placa. Además, es conveniente que estén cerca de los pines de tierra para poder conectar las tierras de la FPGA.

En nuestro caso hemos elegido los pines juntos, de tal manera que se puedan cambiar varios pines a la vez escribiendo en la variable interna de los puertos que tiene el Teensy. En este proyecto la velocidad no es un objetivo, por lo que no se utilizará esta característica, sino una función ya preparada de Arduino, sin embargo, queremos que el prototipo pueda estar listo para versiones futuras. Se han elegido los pines 0 y 1 para las señales **WrA** y **RdD** y del 2 al 5 para **D0 – D3** por estar juntos, poderse controlar desde un mismo puerto (ver anexo VII) y estar junto al pin de tierra. Se conectarán también los dos pines de tierra de la parte del Teensy cercana a su puerto USB. Se puede ver la definición de las variables de los pines en la Figura 51 y un esquema de la conexión en la Figura 52.

```
// Definimos los pines que vamos a utilizar para el bit banging
const int WrA =  0;
const int RdD =  1;
const int D0  =  2;
const int D1  =  3;
const int D2  =  4;
const int D3  =  5;
```

Figura 51 Definición de los pines del Teensy

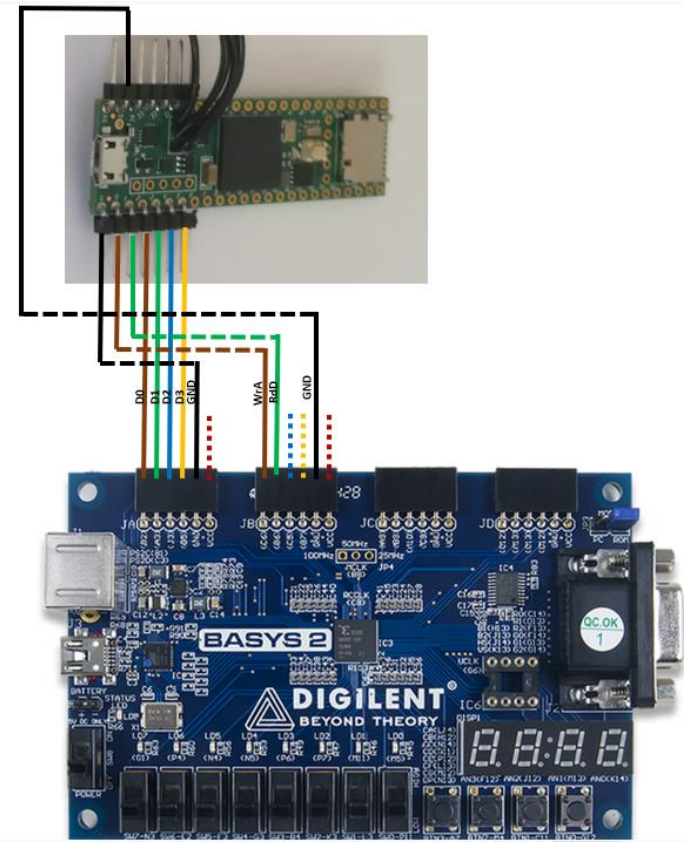

Figura 52 Esquema de conexión Teensy-FPGA

Por último, declaramos las funciones que controlan la FPGA.

```
bool write_address(ulong address);
bool read_address(ulong & address);

bool write_data(uint *data_array, ulong length);
bool read_data(uint *data_array, ulong length);
```

Figura 53 Declaración de las funciones de control de la FPGA

Las cuatro funciones se corresponden con los cuatro tipos de entradas a la FPGA, de lectura/escritura de direcciones o datos, que vimos en el capítulo 2.6. Las que manejan direcciones tienen como argumento una variable de 32 bits de la que leen o en la que escriben la dirección (por referencia). Las que manejan datos reciben un puntero a un vector de bytes de datos del que leen o en el que escriben los datos y la longitud a escribir o leer de la FPGA.

3.5.2 Función setup

Lo primero que debe hacer el Teensy es configurar los pines conectados a la FPGA para evitar que ambos escriban en estos pines y provoquen un cortocircuito, así como para dar el estado inicial a partir del cual se intercambian los mensajes. Para ello utilizamos las funciones de Arduino "*pinMode*" y "*digitalWrite*", que nos permiten establecer el modo de los pines (entrada/salida) y escribir en ellos.

```
// Inicializamos los pines WrA y RdD
pinMode(WrA, OUTPUT);
pinMode(RdD, OUTPUT);
digitalWrite(WrA, HIGH);
digitalWrite(RdD, HIGH);

// Inicializamos los pines de datos
pinMode(D0, INPUT_PULLUP);
pinMode(D1, INPUT_PULLUP);
pinMode(D2, INPUT_PULLUP);
pinMode(D3, INPUT_PULLUP);
```

Figura 54 Valor inicial de los pines del Teensy

Los pines **WrA** y **RdD** los definimos como salidas (*OUTPUT*) e inicialmente estarán a '1' o en "*HIGH*". Los pines de **DATA** los definimos como entradas de bajo consumo (*INPUT_PULLUP*) para que no intenten escribir a la vez que la FPGA, pero tampoco demanden corriente de ella.

En este proyecto vamos a utilizar el monitor serie de Arduino para enviarnos mensajes de control desde el Teensy, aprovechando que el propio cable de alimentación nos ofrece la conexión serie. Esto nos permite conocer las conexiones y peticiones del cliente sin necesidad de ejecutarlo en el mismo PC. Para una fase más avanzada del proyecto en la que no se alimente al Teensy desde un ordenador, se puede simplemente eliminar el control serie sin perjudicar al resto del código.

```
//  Abrimos el puerto serie y esperamos a que se utilice:
Serial.begin(9600);
while (!Serial) {
  // Esperamos al puerto serie
}
Serial.println("Control FPGA Server:");
```

Figura 55 Inicialización de la comunicación con el monitor serie

La comunicación con el monitor serie de Arduino se establece a través de la clase *Serial*. Para iniciar la comunicación utilizamos la función "*begin*" en el puerto que Arduino utiliza para esta conexión, el 9600. Después, esperamos a que se establezca la conexión con un bucle que comprueba el estado y cuando se establece la conexión sacamos por pantalla el nombre del programa. Para sacar tanto texto como variables por el monitor serie utilizamos las funciones "*print*" y "*println*", diferenciándose esta segunda únicamente en que añade un salto de línea al final de la escritura.

Por último, para comenzar la comunicación con el cliente, debemos definir el tamaño del socket que utilizamos y comenzar la comunicación TCP y el servidor.

```
size_t socketSize = 12288;
Ethernet.setSocketSize(socketSize);

// Iniciamos la conexión TCP/IP que hemos definido:
Ethernet.begin(mac, ip, dnServer, gateway, subnet);

// Iniciamos el server que hemos creado:
server.begin();
Serial.print("La IP del servidor es ");
Serial.println(Ethernet.localIP());
```

Figura 56 Inicialización de la comunicación TCP

Para ello hemos utilizado la clase *Ethernet* de la librería **NativeEthernet**, que nos permite fijar el tamaño de nuestro socket en el límite de 12K con la función "*setSocketSize*" y comenzar la conexión TCP/IP con la función "*begin*", utilizando las direcciones IP que hemos definido anteriormente.

Para iniciar el servidor utilizamos el objeto de la clase *EthernetClient* que hemos creado y la función "*begin*".

Finalmente, sacamos por pantalla un aviso con la IP que estamos utilizando para saber que el servidor está conectado.

3.5.3 Función loop

En el loop tenemos la parte principal del código, que se encarga de detectar cuándo llegan mensajes del PC y gestionarlos según sean lecturas o escrituras. Debido a que el Teensy se queda ejecutando esta parte del código indefinidamente debemos entregarle algo de tiempo para que pueda hacer labores internas del microcontrolador como mantenimiento, atención a entradas y salidas, etc. Para ello vamos a utilizar la función "*delay*" de Arduino que nos permite ceder el control al procesador durante una cantidad de milisegundos. Inicialmente vamos a dejar tiempos grandes para que el programa funcione y sea robusto, que se podrán ir reduciendo cuando se quiera mejorar el programa en velocidad.

Inicialmente vamos a esperar a que se conecte algún cliente, por lo que intentaremos crear un nuevo objeto de la clase *EthernetClient* con la función "*accept*" de nuestro servidor.

```
void loop() {
  delay(100);  // Dejamos algo de tiempo al procesador entre cliente y cliente

  // Creamos un objeto "client" que aloje a nuestro nuevo cliente
  EthernetClient client = server.accept();

  if(client) {
    Serial.print("Nuevo cliente: ");
    Serial.println(client.remoteIP());
```

Figura 57 Espera al cliente en la función loop

Entre intento de crear clientes o entre clientes reales dejamos 100 milisegundos al procesador. Cuando se crea un nuevo cliente, nos avisa por el monitor serie y lo identifica sacando su IP por pantalla con la función "*remote IP*".

Una vez conectado un cliente, estaremos atentos únicamente a él hasta que este se desconecte, ya que el programa está orientado a un solo cliente en el PC. Comprobaremos que sigue conectado entre mensaje y mensaje con la función "*connected*" que podemos ver en la Figura 58. Solo interpretaremos los mensajes cuando nos hayan enviado al menos 10 bytes, que corresponden

a los bytes de control, la dirección y la longitud. Esto se comprueba con la función "*available*".

```
while(client.connected()) {
  if(client.available() >= 10){
    c = client.read();
    delay(1);  // Dejamos algo de tiempo al procesador entre lecturas

    if(c == 'A'){ // Verificamos los bytes de control A y T
      c = client.read();
      if(c == 'T'){
        Serial.print("Nuevo mensaje: ");
        c = client.read();
        switch(c){    // Comprobamos el tipo de instrucción
```

Figura 58 Comprobación de cliente conectado e instrucción recibida

Ahora utilizamos la variable "c" y la función "*read*", que lee la información que llega del PC byte a byte, para comprobar cada byte que nos va enviando el cliente. Si detectamos nuestros bytes de control (AT) significa que comienza una nueva instrucción, por lo que el siguiente byte indica si es una instrucción de lectura (R) o de escritura (W). Avisamos del nuevo mensaje por el monitor serie y distinguimos estos casos utilizando una estructura switch.

El primer caso que hemos desarrollado es el de escritura, que tiene que leer del PC la dirección, llamar a la función de escritura de dirección en la FPGA, leer la longitud, leer los datos y llamar a la función de escritura de datos en la FPGA. Por último, comprobará la dirección en la que ha acabado la FPGA con la función correspondiente para asegurarse de que ha escrito la longitud indicada.

```
case 'W':    // Caso escritura
 Serial.println("Escritura");

 client.read(buffer, 4);
 direccion = 0;
 for(int i = 3; i >= 0; i--){
   direccion = (direccion << 8) + buffer[i];
 }

 if(write_address(direccion)){
   Serial.print("Se va a escribir en la dirección ");
   Serial.print(direccion & 0xFFFFFF);
   Serial.print(" del device ");
   Serial.println((direccion >> 24) & 0xFF);
 }else{
   Serial.println("Ha habido algun error al intentar escribir la dirección");
 }
```

Figura 59 Lectura de dirección del mensaje del PC y escritura a la FPGA

Comenzamos avisando de la lectura por el monitor serie y leyendo cuatro bytes del mensaje en el buffer auxiliar que vamos a utilizar para componer la dirección. En este caso utilizamos la función "*read*" pasando como argumento el lugar donde guardar los bytes que se lean y el número de bytes a leer.

Sabiendo que los mensajes envían primero el byte menos significativo y el último el más significativo, vamos a ir leyendo del más al menos significativo y desplazando 8 bits y sumando en cada lectura para componer la dirección tras cuatro iteraciones.

Llamamos a la función de escribir dirección en la FPGA y comprobamos la respuesta. Si la lectura ha salido bien avisamos en el monitor serie de la dirección (los tres bytes menos significativos) y el device (byte más significativo) en el que se va a escribir. Si sale mal avisamos del error.

No se ha programado en esta etapa del proyecto una gestión de errores con retroalimentación o que repita el código fallido ya que por cómo está programada la FPGA no podrá detectar errores en su propia ejecución. En una fase más avanzada de la línea de trabajo a la que pertenece el proyecto, si se programa la FPGA para detectar estos errores, se debería añadir la gestión de errores en vez de los avisos por el monitor serie.

La lectura de la longitud es similar a la de dirección pero de tres bytes, como podemos ver en la Figura 60.

```
client.read(buffer, 3);
longitud = 0;
for(int i = 2; i >= 0; i--){
  longitud = (longitud << 8) + buffer[i];
}
Serial.print("Longitud de los datos: ");
Serial.println(longitud);
```

Figura 60 Lectura de longitud del mensaje del PC

A continuación, debemos esperar a haber recibido el número de bytes que indica la longitud para poder leerlos. Para ello utilizamos la función "*available*" y la longitud en forma de entero con signo. Cedemos el control al procesador mientras se espera para poder recibir los bytes.

Finalmente, leemos la cantidad de bytes de datos que nos indica la longitud y llamamos a la función de escritura en la FPGA.

```
while(client.available() < long(longitud)){
  delay(1); // Cedemos el control al procesador
}
client.read(datos, longitud);
if(!write_data(datos, longitud)){
  Serial.println("Ha habido un error de escritura");
}

if(!read_address(comprobar_direccion)){
  Serial.println("La lectura de dirección ha fallado");
}else{
  if(comprobar_direccion - direccion != longitud){
    Serial.println("La lectura ha fallado");
  }
}
```

Figura 61 Lectura de datos del PC, escritura en la FPGA y comprobación

Para comprobar que se han escrito el número de bytes que indica la longitud llamamos a la función de lectura de dirección de la FPGA para saber en qué dirección ha acabado con la variable "comprobar dirección" y le restamos la dirección final. Si la resta no coincide con la longitud, emitimos un mensaje de error.

El caso de lectura utiliza el mismo código, con la excepción de que no necesita leer datos del mensaje si no que llama a la función de lectura de datos de la FPGA solo con el argumento de la longitud a leer y la variable respuesta en la que queremos guardar esos datos. La comprobación de la dirección final es idéntica.

Sin embargo, es necesario añadir un fragmento de código que divida la respuesta en mensajes de 1460 bytes para poder enviarlos al cliente.

```
bytes_por_enviar = longitud;
int i = 0;
while(bytes_por_enviar > 1460){
  for(int j = 0; j < 1460; j++){
    envio[j] = respuesta[i + j];
  }
  client.write(envio,1460);
  bytes_por_enviar -=1460;
  i += 1460;
}

for(int j = 0; j < bytes_por_enviar; j++){
    envio[j] = respuesta[i + j];
}
client.write(envio, bytes_por_enviar);
```

Figura 62 Envío de la respuesta dividida en paquetes

Comenzamos inicialmente fijando los bytes por enviar en la longitud total de datos pedidos. Creamos una variable auxiliar "i" que utilizaremos como puntero del vector de bytes "respuesta". Queremos tomar 1460 bytes de la respuesta y almacenarlos en envío mientras queden más de 1460 bytes por enviar.

Con la función "*write*" de nuestro objeto *client* enviamos los 1460 bytes almacenados en "envío", restamos esos bytes de los que quedan por enviar y desplazamos esas posiciones nuestro puntero "i". Cuando quedan menos de 1460 bytes enviamos los que queden.

Por último, cuando el cliente se desconecta de nuestro servidor, salimos de los bucles y desconectamos el cliente con la función "*stop*", como vemos en la Figura 63.

```
// Si se desconecta, eliminamos el cliente
client.stop();
Serial.println("Cliente desconectado");
```

Figura 63 Desconexión del cliente

El procesador quedaría entonces esperando a un nuevo cliente para volver a empezar el proceso.

3.5.4 Funciones de comunicación con la FPGA

Basándonos en las instrucciones que la FPGA es capaz de interpretar que describimos en el capítulo 2.6 vamos a utilizar el bit banging para enviarlas

desde el Teensy. La FPGA detecta flancos de subida y bajada de estas señales, es decir, cuando la señal pasa de valer '0' a '1' o viceversa. Al iniciar el procesador las señales de control WrA y RdD valen '1' y las de datos se fijan como entradas. Todas las funciones comenzarán y acabarán en este estado.

Vamos a utilizar las funciones de "*pinMode*", "*digitalWrite*" y "*digitalRead*" para establecer el modo en el que están los pines, escribir en ellos y leer de ellos. Además, para respetar la tabla 1 de tiempos de la FPGA vamos a utilizar la función "*delayMicroseconds*" que permite esperar un número concreto de microsegundos. Es bastante más tiempo del que necesita la FPGA, pero por ahora nos asegura que funcionará correctamente.

La función "*write address*" de **escritura de dirección** nos permite escribir en la FPGA una dirección de 32 bits que le pasamos como argumento.

```
bool write_address(ulong address){
  // Para escritura baja primero WrA y un tiempo después RdD
  digitalWrite(WrA, LOW);
  delayMicroseconds(1); // Tiempo entre bajadas (t1)
  digitalWrite(RdD, LOW);
  delayMicroseconds(1); // Tiempo entre inicio e intercambio de datos (t2)

  // Establecemos los pins de datos como outputs
  pinMode(D0,OUTPUT);
  pinMode(D1,OUTPUT);
  pinMode(D2,OUTPUT);
  pinMode(D3,OUTPUT);
```

Figura 64 Bit banging del comienzo de la escritura de una dirección

Para que la FPGA detecte que queremos escribir bajamos (estado *LOW*) primero la señal WrA y después RdD. Una vez la FPGA ha detectado la escritura podemos establecer los pines de datos como salidas (modo *OUTPUT*) ya que no hay riesgo de que ambos escriban a la vez. En caso de que ocurriera, la propia placa de la FPGA incorpora resistencias en serie que protegerían tanto a la FPGA como al Teensy.

```
for(int i = 0; i < 32; i += 8){
  digitalWrite(D0,(address >> (i + 0)) & 0x1);
  digitalWrite(D1,(address >> (i + 1)) & 0x1);
  digitalWrite(D2,(address >> (i + 2)) & 0x1);
  digitalWrite(D3,(address >> (i + 3)) & 0x1);

  delayMicroseconds(1); // Tiempo de setup (t6)

  digitalWrite(WrA, HIGH); // Flanco de subida -> enviamos 4 bits

  delayMicroseconds(1); // Tiempo de hold(t7)

  digitalWrite(D0,(address >> (i + 4)) & 0x1);
  digitalWrite(D1,(address >> (i + 5)) & 0x1);
  digitalWrite(D2,(address >> (i + 6)) & 0x1);
  digitalWrite(D3,(address >> (i + 7)) & 0x1);

  delayMicroseconds(1); // Tiempo de setup (t6)

  digitalWrite(WrA, LOW); // Flanco de bajada -> enviamos 4 bits

  delayMicroseconds(1); // Tiempo de hold (t7)
}
```

Figura 65 Bit banging envío de bytes de dirección

Para enviar una dirección completa utilizamos la variable “i” como puntero que apuntará al primer bit de cada uno de los 4 bytes que queremos mandar. Para mandar 4 bits primero fijamos el valor de los pines de data y después enviamos las señales de control.

Para fijar los valores de los pines hacemos desplazamientos de la dirección y un “*and*” lógico que nos da el valor del bit de cada posición. Para mandar 4 bits de dirección esperamos los tiempos fijados y enviamos flancos de la señal WrA, primero de subida y luego de bajada.

```
  // Devolvemos a los pines de datos a su estado de input
  pinMode(D0, INPUT_PULLUP);
  pinMode(D1, INPUT_PULLUP);
  pinMode(D2, INPUT_PULLUP);
  pinMode(D3, INPUT_PULLUP);

  // Fin de la escritura
  digitalWrite(WrA, HIGH);
  digitalWrite(RdD, HIGH);
  delayMicroseconds(1); // Tiempo entre acciones (t8)

  return true;
}
```

Figura 66 Bit banging del final de la escritura de una dirección

Antes de terminar la escritura devolvemos los pines de datos a su modo inicial y después devolvemos las señales de control a su valor '1'. Devolvemos una variable booleana que nos confirma que hemos completado la instrucción. De nuevo, como la FPGA no puede confirmar el buen funcionamiento no tiene sentido otro tipo de respuesta de la función.

La función "*write data*" de **escritura de datos** permite escribir tantos bytes de un vector de datos como indique la longitud. Escribe a partir de la dirección de memoria que tenga la FPGA escrita y la va autoincrementando con cada byte.

La estructura del bit banging del comienzo y final de la función es igual a la de escritura de dirección, sin embargo, el envío de bytes de datos cambia, como podemos ver en la Figura 67.

```
for(uint i = 0; i < length; i++){
  data = data_array[i];

  digitalWrite(D0,(data >> 0) & 0x1);
  digitalWrite(D1,(data >> 1) & 0x1);
  digitalWrite(D2,(data >> 2) & 0x1);
  digitalWrite(D3,(data >> 3) & 0x1);

  delayMicroseconds(1); // Tiempo de setup (t6)

  digitalWrite(RdD, HIGH); // Flanco de subida -> enviamos 4 bits

  delayMicroseconds(1); // Tiempo de hold(t7)

  digitalWrite(D0,(data >> 4) & 0x1);
  digitalWrite(D1,(data >> 5) & 0x1);
  digitalWrite(D2,(data >> 6) & 0x1);
  digitalWrite(D3,(data >> 7) & 0x1);

  delayMicroseconds(1); // Tiempo de setup (t6)

  digitalWrite(RdD, LOW); // Flanco de bajada -> enviamos 4 bits

  delayMicroseconds(1); // Tiempo de hold(t7)
}
```

Figura 67 Bit banging envío de bytes de datos

En este caso tenemos un bucle que envía byte por byte y se repite tantas veces como indique la longitud de los datos "*length*". Para poder trabajar con grandes vectores de datos utilizamos el paso por puntero del "*data array*", y utilizaremos la variable "data" para guardar el byte que queramos enviar cada vez. De nuevo, esta estructura es más lenta que si no utilizásemos la variable auxiliar, pero más robusta para la funcionalidad que busca este proyecto.

La otra diferencia con la función anterior es que en este caso es la señal RdD la que cambia de estado por ser una escritura de datos.

Para la **lectura de dirección** utilizamos la función "*read address*", que permite leer la dirección que tiene actualmente guardada la FPGA.

```
bool read_address(ulong & address){
  // Aseguramos primero los pins de datos como inputs
  pinMode(D0,INPUT_PULLUP);
  pinMode(D1,INPUT_PULLUP);
  pinMode(D2,INPUT_PULLUP);
  pinMode(D3,INPUT_PULLUP);

  // Ahora bajamos primero WrA y después RdD para avisar de la lectura
  digitalWrite(RdD, LOW);
  delayMicroseconds(1); // Tiempo entre bajadas (t1)
  digitalWrite(WrA, LOW);
  delayMicroseconds(1); // Tiempo entre inicio e intecambio de datos (t2)
```

Figura 68 Bit banging del comienzo de la lectura de una dirección

Para la lectura de dirección comenzamos asegurándonos de que los pines de datos están fijados como entradas. Después podemos bajar la señal RdD y seguidamente WrA para avisar de una nueva lectura.

```
address = 0;
for(int i = 0; i < 32; i += 8){
  digitalWrite(WrA, HIGH); // Flanco de subida -> pedimos 4 bits
  delayMicroseconds(1); // Tiempo de acceso (t9)

  address = address | (digitalRead(D0) << (i + 0));
  address = address | (digitalRead(D1) << (i + 1));
  address = address | (digitalRead(D2) << (i + 2));
  address = address | (digitalRead(D3) << (i + 3));

  digitalWrite(WrA, LOW); // Flanco de bajada -> pedimos 4 bits
  delayMicroseconds(1); // Tiempo de acceso (t9)

  address = address | (digitalRead(D0) << (i + 4));
  address = address | (digitalRead(D1) << (i + 5));
  address = address | (digitalRead(D2) << (i + 6));
  address = address | (digitalRead(D3) << (i + 7));
}
```

Figura 69 Bit banging de la petición de bytes de dirección

Para recibir los bytes de la dirección y guardarlos en la variable "*address*" vamos a recorrer la variable de la misma forma que en la escritura, pero en este caso asignándole el valor de cada bit con un "*or*" lógico, por lo que debemos inicializarla a cero antes del bucle y después compararla con el bit recibido en la posición adecuada gracias a desplazamientos de bits.

Para recibir 4 nuevos bits cambiamos de estado el pin WrA y esperamos un tiempo en lo que la FPGA coloca los datos.

Para terminar la instrucción suben de nuevo ambas señales igual que en la Figura 66, sin embargo, en este caso no es necesario cambiar el modo de los pines de datos.

Por último, para la **lectura de datos** utilizamos la función "*read data*", que mantiene las señales de control del comienzo y final y varía la parte de petición de bytes de datos.

```
for(uint i = 0; i < length; i++){
  data = 0;

  digitalWrite(RdD, HIGH); // Flanco de subida -> pedimos 4 bits
  delayMicroseconds(1); // Tiempo de acceso (t9)

  data = data | (digitalRead(D0) << 0);
  data = data | (digitalRead(D1) << 1);
  data = data | (digitalRead(D2) << 2);
  data = data | (digitalRead(D3) << 3);

  digitalWrite(RdD, LOW); // Flanco de bajada -> pedimos 4 bits
  delayMicroseconds(1); // Tiempo de acceso (t9)

  data = data | (digitalRead(D0) << 4);
  data = data | (digitalRead(D1) << 5);
  data = data | (digitalRead(D2) << 6);
  data = data | (digitalRead(D3) << 7);

  data_array[i] = data;
}
```

Figura 70 Bit banging de la petición de bytes de datos

La forma de acceder al puntero de datos ahora es contraria. Inicializamos primero la variable auxiliar a cero para poder recibir los bits con el "*or*" lógico y al final de cada iteración entregamos ese valor al vector de bytes. En este caso la variable RdD es la que cambia de estado.

CAPÍTULO 4 RESULTADO FINAL, TESTEO Y VERIFICACIÓN

En este capítulo vamos a dar una visión global del proyecto que permita hacerse una idea real del conjunto, así como verificar el funcionamiento y por tanto la consecución de los objetivos y testear los límites del hardware que hemos utilizado y el software que hemos creado.

4.1 ESQUEMA FINAL

En la Figura 71 podemos ver una fotografía de la disposición final de todos los elementos que componen el proyecto, conectados entre sí y funcionando.

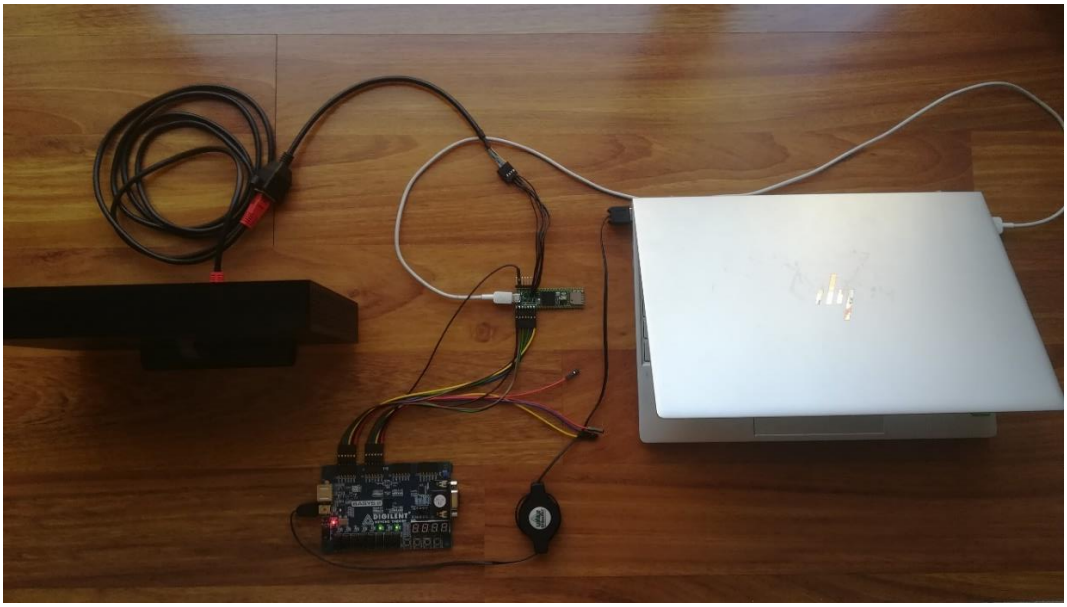

Figura 71 Disposición final de todos los elementos del proyecto

A la derecha vemos el PC que ejecuta el programa en Qt con la interfaz de usuario. Conectados a este están el Teensy 4.1, con un cable USB blanco; y la FPGA, con un cable USB negro. Entre sí ambos elementos se conectan con cables flexibles de diferentes colores según el esquema de la Figura 52. Finalmente, el Teensy se conecta al router del lado izquierdo con el conector Ethernet y el cable CAT5e (conectores RJ45 rojos). El PC estaría conectado al router a través de la red WiFi.

4.2 VERIFICACIÓN DEL PROYECTO

Si ejecutamos nuestro programa en Qt desde el PC se nos abre una ventana que aloja la interfaz de usuario que hemos creado. Vamos a abrir también el

monitor serie de Arduino e introducir las entradas necesarias para comprobar que el proyecto funciona.

Por ahora, la FPGA nos permite escribir en el device 0 para acceder a los LEDs y al device 5 para acceder a una pequeña memoria, como explicamos en el capítulo 2.6. Vamos a acceder primero al registro del device 0 y escribir un número para observarlo en los LEDs.

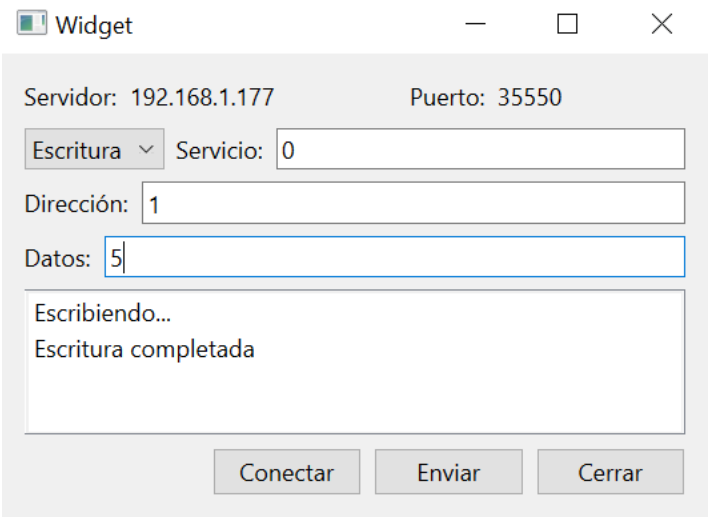

Figura 72 Verificación del programa escribiendo en los LEDs

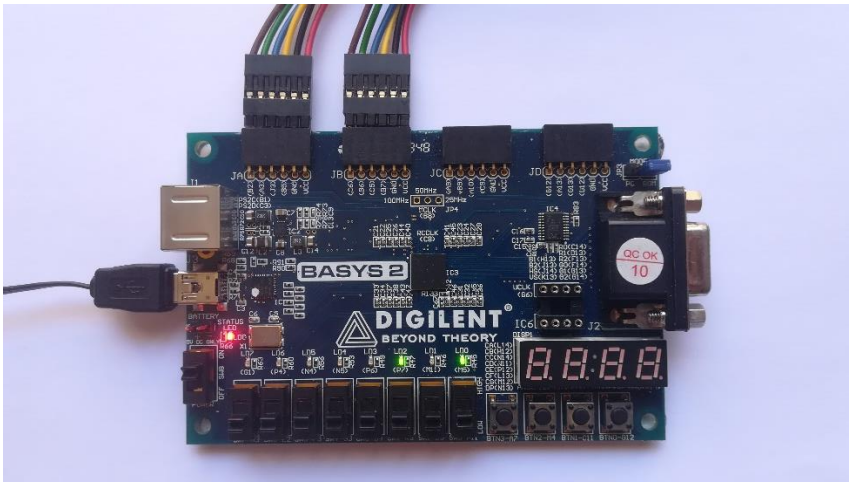

Figura 73 Reacción a la escritura en los LEDs de la FPGA

Como podemos ver en la Figura 73, se encienden los leds LD0 y LD2, mientras que el LD1 y el LD3 se quedan apagados, como corresponde al número '0101' en binario, es decir, el número 5 en decimal. (Por cómo está programada la interfaz el programa está escribiendo el código UTF-8 del número 5 en 8 bits, cuyos 4 bits menos significativos corresponden al número 5 en binario).

Mientras tanto, el monitor serie de Arduino nos avisa de la conexión y del mensaje de escritura (Figura 74).

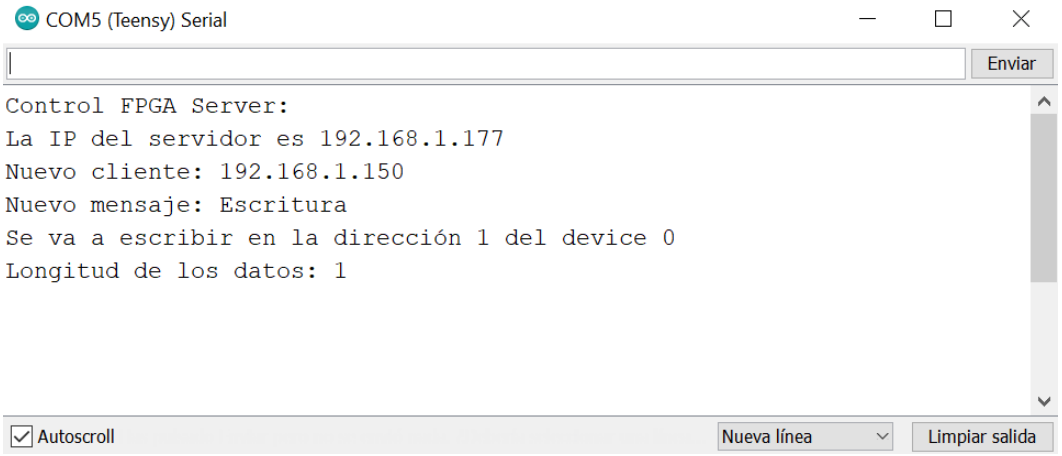

Figura 74 Respuesta del monitor serie a la escritura en los LEDs

Ahora escribiremos en la primera dirección de memoria del device 5 un mensaje y lo leeremos de vuelta.

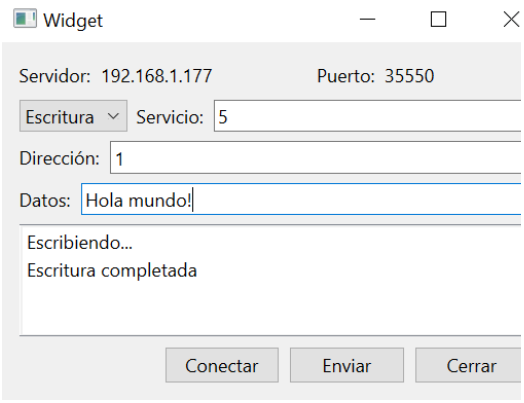

Figura 75 Escritura en la memoria

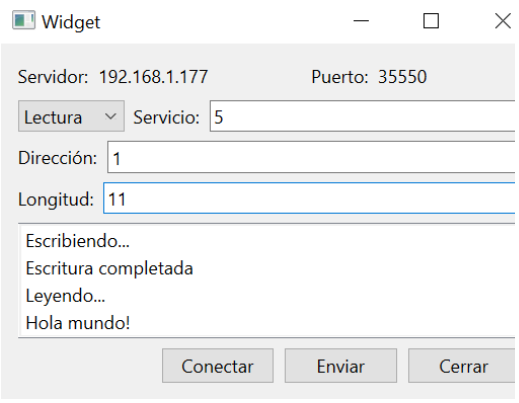

Figura 76 Lectura de la memoria

Figura 77 Respuesta del monitor serie a la escritura y lectura en la memoria

En el monitor serie podemos ver los avisos de conexión, escritura de 11 bytes y lectura de estos en la dirección 1 del device 5.

En general, se han probado los programas para todos los valores que se pueden leer y escribir en un byte y para todas las posiciones de memoria a las que se tiene acceso en la FPGA, con resultados favorables y estables.

4.3 TESTEO DE VELOCIDAD Y FLUJO DE DATOS

Para poder conocer las limitaciones del proyecto y ser capaces de definir las líneas futuras necesitamos conocer qué cantidad de bytes puede transmitir y sobre todo a qué velocidad.

Hemos creado un nuevo proyecto en Qt que compone un mensaje de la longitud que elijamos y lo envía tantas veces como queramos al Teensy. Al comenzar a transmitir se conecta al servidor y al terminar se desconecta para poder medir en el Teensy el tiempo que tarda. Vemos la función "*main*" en la Figura 78.

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    teensy4_1 = new Teensy();

    quint32 tamanio = 10000;
    quint32 repeticiones = 1;

    quint32 address = 0x05000001;
    QByteArray mensaje;
    QByteArray respuesta;

    for(quint32 j = 0; j < tamanio; j++){
        mensaje.append(0x1F);
    }

    for(quint32 i = 0; i < repeticiones; i++){
        teensy4_1->write(address, mensaje);
        //teensy4_1->read(address, mensaje.length(), respuesta);

        //QThread::msleep(500);
    }

    teensy4_1->disconnection();

    return a.exec();
}
```

Figura 78 Código de testeo de velocidad de la comunicación

Incluimos nuestra clase *Teensy* y la clase *QThread*, que utilizaremos para introducir retardos. Definimos un objeto de nuestra clase, un tamaño de paquetes y un número de repeticiones para el test. Escribiremos en el device 5 desde la primera dirección de memoria, por lo que le asignamos la dirección correspondiente en hexadecimal. Vamos a manejar más bytes de los que caben en la memoria programada en la FPGA, por lo que en la realidad solo se escribirán los primeros 4 Kbytes, sin embargo, nos dará una idea del tiempo que tarda el proceso de escritura y lectura del Teensy.

Para componer el mensaje declaramos un vector de bytes y lo inicializamos con un byte cualquiera (en este caso el número 1F en hexadecimal) con un bucle *for*. Comprobaremos la escritura y la lectura por separado, por lo que la función que no estemos utilizando la comentaremos (en este caso la función *read*). Se repetirá la lectura o la escritura el número de veces que hayamos fijado.

Hemos observado que al enviar varios paquetes seguidos el Teensy se satura, por lo que se añade un retardo en milisegundos con la función "*msleep*" de la clase *QThread* para ralentizar el flujo de mensajes que envía Qt, con el objetivo

de mejorar el desempeño del Teensy recibiendo y gestionando los mensajes. Se ha añadido este mismo retardo en la fragmentación en paquetes de 12 Kbytes que hace la función de escritura de la clase *Teensy*.

Finalmente nos desconectamos del servidor para que este pueda medir el tiempo que ha durado el intercambio.

En el código de Arduino hemos añadido algunas variables para detectar si ha sido una lectura o escritura, medir el tiempo que tarda el intercambio de datos y el número de bytes que se han intercambiado para poder hallar la velocidad (Figura 79).

```
char modo;
float t_inicial, t_final, bytes_totales;
float velocidad;
```

Figura 79 Variables añadidas para medir la velocidad en el Teensy

Cuando se conecta un nuevo cliente esperamos a que mande el primer mensaje y comenzamos a contar el tiempo y los bytes que se van enviando. Para medir el tiempo utilizamos la función "*micros*", que nos devuelve los microsegundos que han pasado desde que se inició el Teensy. Tomaremos la medida al comenzar la conexión y al desconectarse para calcular la duración con la diferencia.

```
if(client) {
  Serial.print("Nuevo cliente: ");
  Serial.println(client.remoteIP());

  while(!client.available()){
    // Esperamos a que mande el primer mensaje
  }
  t_inicial = micros();
  bytes_totales = 0;
```

Figura 80 Inicialización del tiempo y los bytes

Con cada mensaje recibido aumentaremos los bytes totales en la longitud de bytes escritos o leídos y sacaremos por pantalla un punto para asegurarnos de que el programa está funcionando correctamente. Finalmente calcularemos la velocidad de transmisión de datos en Kbytes por segundo y sacaremos un mensaje por pantalla según el modo que se haya probado, como podemos ver en la Figura 81.

```
        // Si se desconecta, eliminamos el cliente
        client.stop();
        Serial.println();
        Serial.println("Cliente desconectado");

        t_final = micros();
        velocidad = (bytes_totales*1000)/(t_final - t_inicial);


        switch(modo){
          case 'W':
            Serial.print("Velocidad de escritura: ");
            Serial.print(velocidad);
            Serial.println(" Kbytes/s");
            break;

          case 'R':
            Serial.print("Velocidad de lectura: ");
            Serial.print(velocidad);
            Serial.println(" Kbytes/s");
            break;

        }
        Serial.println();
    }
```

Figura 81 Cálculo de velocidad de intercambio de datos

Si ejecutamos el programa para un mensaje de 10 Kbytes tanto para la escritura como para la lectura la respuesta del monitor serie del Teensy es la que observamos en la Figura 82.

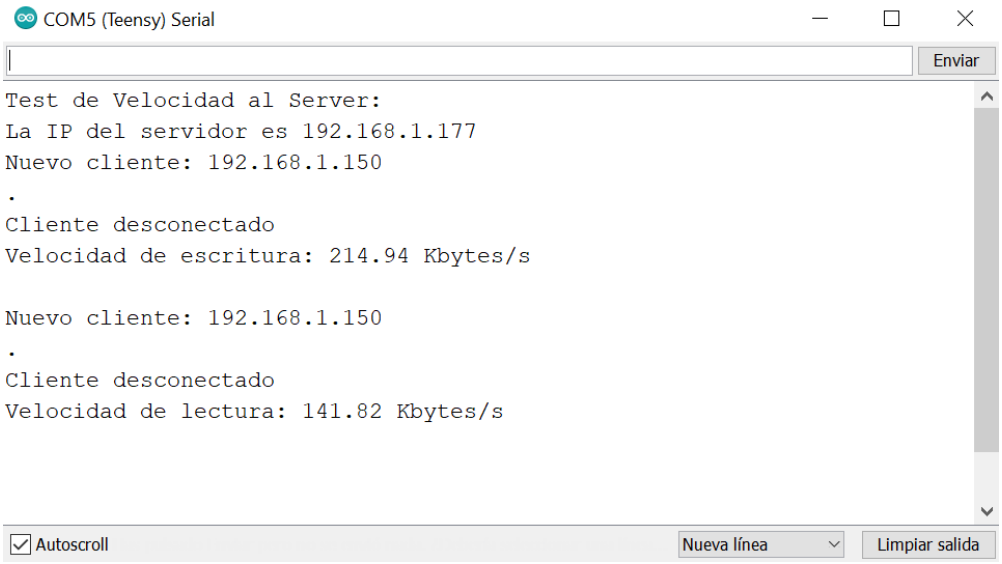

Figura 82 Salida del Teensy para el caso de escritura y lectura de 100 Kbytes

Como podemos ver, el cliente en Qt se conecta y envía el mensaje de escritura. Cuando se desconecta nos indica que la velocidad de escritura ha sido de 214,94 Kbytes/s.

Hace después otra conexión de lectura. En este caso la velocidad de lectura es de 141,82 Kbytes/s.

Vamos a utilizar el programa para enviar diferente número de bytes y mensajes para comprobar los límites de velocidad que alcanza y sacar conclusiones. Hemos presentado los resultados en las Tabla 2 y Tabla 3.

Tabla 2 Velocidad de escritura en Kbytes/s según el número de bytes y mensajes

Velocidad de escritura (Kbytes/s)		Número de bytes					
		1	10	100	1K	10K	100K
Número de mensajes	1	0,09	0,9	8,69	64,1	214,94	1,44
	10	0,09	0,9	8,7	64,16	2,51	≤ 1,44
	100	0,09	0,9	8,7	64,14	≤ 2,51	≤ 1,44
	1K	0,09	0,66	6,54	58,32	≤ 2,51	≤ 1,44
	10K	0,07	0,65	6,49	21,56	≤ 2,51	≤ 1,44
	100K	≤ 0,07	≤ 0,65	≤ 6,49	≤ 21,56	≤ 2,51	≤ 1,44

Como podemos ver, la velocidad de escritura de datos aumenta cuantos más bytes escribimos en un mismo mensaje, pero observamos una saturación del Teensy cuando mandamos demasiados mensajes seguidos y por tanto una velocidad menor.

Esto ocurre cuando mandamos más de 1.000 mensajes pequeños, pero sobre todo cuando mandamos varios mensajes de 10 Kbytes, o al enviar un mensaje de 100 Kbytes, ya que el programa en PC lo tiene que dividir en mensajes de 12 Kbytes.

Se han estimado algunos de los datos en los que la duración del test era excesiva.

Tabla 3 Velocidad de lectura en Kbytes/s según el número de bytes y mensajes

Velocidad de lectura (Kbytes/s)		Número de bytes					
		1	10	100	1K	10K	100K
Número de mensajes	1	0,02	0,31	7,17	55,41	141,82	255,94
	10	0,03	0,5	2,85	37,61	196,1	324,5
	100	0,05	0,52	4,84	35,31	194,16	339,25
	1K	0,04	0,47	4,59	36,64	187,5	-
	10K	0,04	0,47	3,74	35,72	≤ 187,5	-
	100K	≤ 0,04	≤ 0,47	≤ 3,74	≤ 35,72	≤ 187,5	-

En el caso de la lectura los datos son mucho más variables ya que dependen en mayor medida de la FPGA, sin embargo, los datos obtenidos nos servirán para hacernos una idea de la tendencia que toma.

Podemos observar que para un numero pequeño de bytes las lecturas son más lentas que las escrituras y que de nuevo se vuelven algo más lentas cuantos más mensajes seguidos enviamos. Sin embargo, la saturación del Teensy casi no ocurre para los mensajes de lectura.

La gran diferencia con la escritura es que es capaz de leer mensajes de 100 Kbytes mucho más rápido, debido a que el Teensy no tiene la limitación de tener que recibir varios mensajes y, aunque tenga que fragmentar la respuesta, los datos nos indican que puede hacerlo a una buena velocidad.

Estos datos nos hacen pensar también que los mensajes de 100 Kbytes son los óptimos en cuanto a velocidad.

CAPÍTULO 5 CONCLUSIONES

5.1 CONCLUSIONES GENERALES Y TRANSVERSALES

Una vez completado el proyecto, comprobado su funcionamiento y obtenidos unos resultados de velocidad, vamos a recapitular los objetivos que nos habíamos marcado y los avances que hemos logrado para sacar unas conclusiones que puedan aportar a otros proyectos y permitan continuar la línea de trabajo.

Enunciaremos los objetivos que hemos logrado y compararemos con los que nos propusimos al comienzo del trabajo:

- Hemos conseguido conectar entre sí el PC con el Teensy y este a su vez con la FPGA.
- Hemos establecido una comunicación TCP funcional entre una aplicación en el PC con Qt y un programa en el Teensy con Arduino.
- Hemos podido leer y escribir en espacios de memoria de una FPGA enviando mensajes desde el programa en PC y traduciéndolos con el Teensy.
- Conseguimos un flujo de datos estable y fiable que nos permitirá programar la FPGA e interactuar con elementos programados en ella.
- Conseguimos velocidades estables de hasta 215 Kbytes/s para escritura en la FPGA y hasta 339 Kbytes/s para lectura.

Nuestro principal objetivo era establecer la conexión y lograr que se pudiera acceder a la memoria de la FPGA desde el programa en PC, por lo que lo hemos cumplido y verificado.

Queríamos funcionalidad y fiabilidad ya que el proyecto forma parte de una línea de trabajo y hemos hecho que las clases y funciones creadas sean flexibles para poder utilizarlas con otras interfaces de usuarios o incluso en conexiones diferentes. Las funciones de bit banging con la FPGA del Teensy son independientes de la conexión TCP y se podrán utilizar para conexiones con WiFi o USB. El código de conexión TCP es flexible y podrá añadir nuevas instrucciones, por ejemplo, para configuración de la FPGA.

Buscábamos también que el proyecto fuese robusto, y para ello hemos tenido que solventar las limitaciones de las librerías del Teensy con fragmentación de los datos tanto en la escritura como en el envío de la respuesta de escritura. Hemos preparado los programas para poder modificar o eliminar esas modificaciones si se resolvieran las limitaciones. Para conseguir esta robustez

en la escritura de varios mensajes de grandes paquetes de datos ha sido necesario añadir retardos entre mensajes, lo cual es probablemente el punto más débil de este proyecto.

Aunque no era uno de nuestros objetivos, hemos analizado la velocidad de intercambio de datos que nos ofrece el proyecto. Por ahora son velocidades muy bajas teniendo en cuenta que las limitaciones del Ethernet del Teensy son de 10 Mbytes/s. Sin embargo, hemos podido comprobar que estas limitaciones se encuentran principalmente en la clase de gestión de Ethernet del Teensy, ya que limita la cantidad de bytes que puede recibir y enviar en un mensaje.

Las velocidades caen mucho al no poder recibir más de 12 Kbytes en la escritura y eso hace que los mensajes se acumulen y el Teensy se ralentiza o incluso falla si se envían demasiados mensajes al mismo tiempo. Sin embargo, es mucho más rápido almacenando grandes vectores de datos y enviándolos en paquetes pequeños como ocurre con la respuesta.

En cuanto a la FPGA, no se ha podido analizar en profundidad puesto que estaba programada tan solo para verificar el proyecto, pero parece que las escrituras son más estables y rápidas que las lecturas. En ese sentido aún queda bastante margen de mejora tanto desde el programa del Teensy como con el hardware si fuese necesario.

5.2 LÍNEAS FUTURAS

Como ya sabemos este proyecto forma parte de una línea de trabajo que busca poder programar rápidamente la FPGA y establecer una conexión lo suficientemente rápida como para intercambiar datos de variables internas a tiempo real. Este proyecto ofrece la funcionalidad y robustez necesarias para poder optimizar la velocidad de flujo de datos e incorporar otras señales y paquetes de datos.

Se plantean los siguientes pasos a dar:

- Mejorar el hardware utilizado. Es recomendable utilizar el kit de Ethernet que ofrece el fabricante y para el que posiblemente estén diseñadas las librerías. También sería una buena idea utilizar cables más cortos y fiables entre los elementos para ganar algo de velocidad y estabilidad.
- Probar el proyecto con un cable Ethernet cruzado entre un PC y el Teensy para dar más estabilidad a la conexión.
- Resolver los cuellos de botella de velocidad del proyecto según vayan encontrándose, comenzando por la comunicación TCP del Teensy. Las librerías aún están en desarrollo y tal vez se actualicen,

pero también se podría diseñar una comunicación TCP propia que utilice mejor los recursos del Teensy.

- Acelerar el proceso de lectura y escritura en la FPGA. Se han utilizado pines del Teensy que pueden accederse directamente escribiendo en los puertos del microcontrolador. En el anexo VII está el programa de Arduino que hemos creado para comprobar los pines que se pueden utilizar y cómo acceder a ellos.
- También se pueden acelerar las interacciones con la FPGA ajustando los retardos de las funciones de lectura y escritura a los de la Tabla 1, tal vez diseñando funciones con pocas instrucciones que obliguen al procesador del Teensy a "perder" cierta cantidad de nanosegundos.
- Utilizar el proyecto con una FPGA programada para detectar errores y capaz de ofrecernos más información sobre el proceso, permitiendo así eliminar el monitor serie de Arduino y creando un protocolo de gestión de errores que informe directamente al usuario.

BIBLIOGRAFÍA

En este trabajo se han utilizado referencias bibliográficas más consolidadas para los conocimientos teóricos y otras referencias complementarias que han ayudado a comprender ciertos conceptos o han aportado ilustraciones sencillas de entender. También se han tenido que utilizar foros de programación ya que parte del material utilizado no tiene referencias consolidadas al llevar utilizándose muy pocos años.

Se presentarán ambas bibliografías por separado siguiendo el orden de aparición en el texto de cada bloque.

BIBLIOGRAFÍA CONSOLIDADA

[1] M. L. López Vallejo y J. L. Ayala Rodrigo, «FPGA: Nociones básicas e implementación,» Universidad Politécnica de Madrid, Madrid, abril 2004. Último acceso: 11 julio 2022. [En línea]. Disponible en: http://www.miky.com.ar/fpga_2004.pdf

[2] P. J. Stoffregen y R. C. Coon. «Teensy 4.1 Development Board.» PJRC.com. https://www.pjrc.com/store/teensy41.html (Último acceso: 11 julio 2022).

[3] J. Lambert, R. Monahan, y K. Casey, «Power Consumption Profiling of a Lightweight Development Board: Sensing with the INA219 and Teensy 4.0 Microcontroller,» en *Electronics,* marzo 2021. [En línea]. Disponible en: https://www.mdpi.com/2079-9292/10/7/775?type=check_update&version=1

[4] N. Reid, C. Tilbury, y J. Wylie, «Revenge of the Synth: Polyphonic Audio Synthesis using a Field-Programmable Gate Array,» Universidad de Ciudad del Cabo, Ciudad del Cabo, 2020. Último acceso: 11 julio 2022. [En línea]. Disponible en: http://ocw.ee.uct.ac.za/courses/EEE4120F/HOF.html

[5] M. Acevedo Sanz, «FPGA-Based Hardware-In-the-Loop Co-Simulator Platform for SystemModeler,» Universidad de Linköping, Linköping, 2016. Último acceso: 11 julio 2022. [En línea]. Disponible en: https://www.diva-portal.org/smash/get/diva2:1059834/FULLTEXT01.pdf.

[6] C. E. Spurgeon y J. Zimmerman, Ethernet: The Definitive Guide, O'Reilly Media, Inc., 2014.

[7] M. Fernández Barcell. (2008). Tema VI: Protocolo TCP/IP. [PDF]. Disponible en: http://www.mfbarcell.es/redes_de_datos/tema_06/temaVI_tcpip.pdf. [Último acceso: 11 julio 2022].

[8] M. Banzi, D. Cuartielles, T. Igoe, y D. Mellis, «What is Arduino?» Arduino.cc. https://www.arduino.cc/en/Guide/Introduction. (Último acceso: 11 julio 2022).

[9] Q. Group, «Página principal de Qt.» qt.io. https://www.qt.io/. (Último acceso: 11 julio 2022).

[10] H. M. Deitel y P. J. Deitel, *Cómo programar C++*. Mexico: Pearson Educación, 2008.

[11] J. Blanchette y M. Summerfield, *C++ GUI Programming with Qt 4*. Prentice Hall, 2006.

[12] Qt Group, «Documentación de Qt sobre el sistema de metaobjetos.» doc.qt.io. https://doc.qt.io/qt-6.2/metaobjects.html . (Último acceso: 11 julio 2022).

[13] U. Farooq et al., «Capítulo 2: FPGA Architectures: An Overview,» de *Tree-Based Heterogeneous FPGA Architectures*, Nueva York, Springer Science y Business Media, 2012, doi: 10.1007/978-1-4614-3594-5_2

[14] I. Kuon, R. Tessier y J. Rose, «FPGA Architecture: Survey and Challenges,» *Foundations and Trends® in Electronic Design Automation,* vol. 2, nº 2, 2008, doi: 10.1561/1000000005

[15] *Spartan-3E FPGA Family Data Sheet*, AMD XILINX, 2018. [En línea]. Disponible en: https://docs.xilinx.com/v/u/en-US/ds312. [Último acceso: 11 julio 2022].

[16] *Basys 2 FPGA Board Reference Manual*, DILIGENT, 2016. [En línea]. Disponible en: https://digilent.com/reference/_media/basys2:basys2_rm.pdf . [Último acceso: 11 julio 2022].

BIBLIOGRAFÍA COMPLEMENTARIA

[17] O. Weis. «Qué es Ethernet y Cómo Funciona.» net-usb.com. https://www.net-usb.com/es/usb-over-ethernet-system/what-is-ethernet/. (Último acceso: 11 julio 2022).

[18] IONOS, «¿Qué es Ethernet (IEEE 802.3)?» ionos.es. https://www.ionos.es/digitalguide/servidores/know-how/ethernet-ieee-8023/. (Último acceso: 11 julio 2022).

[19] Y. Fernández, «Cable de red Ethernet: categorías, protecciones y cómo saber cuál comprar.» xataka.com. https://www.xataka.com/basics/cable-red-ethernet-categorias-protecciones-como-saber-cual-comprar. (Último acceso: 11 julio 2022).

[20] «Conector RJ45.» wikipedia.org. https://es.wikipedia.org/wiki/RJ-45. (Último acceso: 11 julio 2022).

[21] «Práctica de laboratorio: Armado de un cable cruzado Ethernet.» practicasuptxabraham. https://sites.google.com/site/practicasuptxabraham/4-2practica-de-laboratorio-armado-de-un-cable-cruzado-ethernet. (Último acceso: 11 julio 2022).

[22] «Protocolo TCP/IP.» aprendiendoarduino.wordpress.com. https://aprendiendoarduino.wordpress.com/category/tcpip/. (Último acceso: 11 julio 2022).

[23] «Ethernet.» Wikipedia.org. https://es.wikipedia.org/wiki/Ethernet. (Último acceso: 11 julio 2022).

[24] Thomas Kim. Qt 6 on Windows. (16 mayo 2021). Último acceso: 11 julio 2022. [Vídeo en línea]. Disponible en: https://youtu.be/t-lDoy16ErQ.

[25] «Foro de la dirección MAC del Teensy 4.1.» forum.pjrc.com. https://forum.pjrc.com/threads/62932-Teensy-4-1-MAC-Address. (Último acceso: 11 julio 2022).

[26] «Foro de la librería NativeEthernet para el Teensy 4.1.» forum.pjrc.com. https://forum.pjrc.com/threads/60857-T4-1-Ethernet-Library. (Último acceso: 11 julio 2022).