Universidad Tecnológica Nacional
Facultad Regional Bahía Blanca

Ingeniería Electrónica

Trabajo Final de Grado

Desarrollo de una red de distribución de vídeo RADAR en protocolo ASTERIX CAT240 sobre Ethernet Gigabit

Autor/es
Tec. Agustin Emanuel Allende
Tec. Juan José Miguel Aguirre

Director o Tutor
Mg. Guillermo Friedrich
Ing. Adrián Laiuppa

Bahía Blanca | 15 de Septiembre de 2023

Índice de contenidos

Resumen

En este informe se describe el desarrollo e implementación de una red de distribución de vídeo RADAR en protocolo ASTERIX CAT240 sobre Ethernet Gigabit.

Este proyecto se realiza en el marco del proyecto Desarrollo de una consola de operaciones que pueda vincularse mediante una interfase de comunicación bidireccional al Sistema de Comando y Control de unidades de la Armada Argentina llevado adelante por el grupo Soluciones Embebidas Aplicadas de la Escuela de Oficiales de la Armada y del Servicio de Análisis Operativos, Armas y Guerra Electrónica de la Armada Argentina.

Palabras Clave: *FPGA, HPS, RADAR, ASTERIX CAT240, handshake, Ethernet, Gigabit, UDP.*

Lista de abreviaciones

ADC: Analog-to-Digital Converter.
ARM: Advanced RISC Machine.
ASTERIX: All purpose STructured Eurocontrol suRveillance Information eXchange.
API: Application Programming Interface.
AWM: Appliance Wiring Material.
AXI: Advanced eXtensible Interfaces.
BI: Bearing Increment.
CAT8: CATegory 8.
CAT240: CATegory 240.
CPU: Central Processing Unit.
CRC: Cyclic Redundancy Check.
CS: Class Selector.
DHCP: Dynamic Host Configuration Protocol.
DMA: Direct Memory Access.
DSCP: Differentiated Services Code Point.
ECN: Explicit Congestion Notification.
FCS: Frame Check Sequence.
FPGA: Field-Programmable Gate Array.
FSPEC: Field SPECification.
FTP: Foiled Twisted Pair.
GPIO: General Purpose Input/Output.
GUI: Graphic User Interface.
HM: Heading Marker.
HPS: Hard Processor System.
IHL: Internet Header Length.
IP: Internet Protocol.
LEN: LENght indicator.
MAC: Media Access Control.
MBPS: MegaBits Per Second.
MF: More Fragments.
MTU: Maximun Transmission Unit/Maximun Transfer Unit.
OS: Operating System.
OSI: Open Systems Interconnection.
PC: Personal Computer.
PHY: PHYsical layer.
PLL: Phase-Locked Loop.
RADAR: Radio Detection And Ranging.
RAM: Random Access Memory.
RGMII: Reduced Gigabit Media-Independent Interface.
RISC: Reduced Instruction Set Computer.
ROM: Read Only Memory.
SAC: System Area Code.
SIC: System Identification Code.
SD: Secure Digital.
SFD: Start Frame Delimiter.
SoC: System on Chip.
SRAM: Static Random Access Memory.
SSH: Secure SHell.

TCP: Transfer Control Protocol.
TOS: Type Of Service.
TTL: Time To Live.
UDP: User Datagram Protocol.

Lista de figuras

Lista de tablas

1. Introducción

La finalidad del proyecto consiste en diseñar e implementar una red para la distribución de video RADAR digitalizado y empaquetado en protocolo ASTERIX CAT240 a velocidades de Gigabit, o lo más cercano a ésta, utilizando UDP.

Para la implementación del sistema propuesto, luego de sortear diferentes alternativas, se optó por utilizar un SoC contenido dentro de una FPGA ya que es, en igual proporción, simple y económica además de ir acorde a los tiempos de desarrollo del proyecto en cuestión. La herramienta elegida es el kit DE1-SoC, el cual contiene la FPGA Intel Cyclone V 5CSEMA5F31.

Luego, de manera extensa y detallada, se describe la investigación realizada y el paso a paso seguido en el desarrollo para la correcta configuración y puesta en marcha del kit mencionado para comenzar así con la aplicación requerida.

Posteriormente, se detalla el software realizado para la comunicación entre la FPGA y el HPS, el cual se divide fundamentalmente en dos partes: una realizada en *Qsys*, en la cual se instancia todo el hardware necesario para la comunicación entre la FPGA y el HPS, y otra parte realizada en *QtCreator*, la cual se utiliza para programar el HPS.
En pocas palabras, el software realizado se encarga, en primera instancia, de mapear los puertos para lectura de memoria de la FPGA a memorias virtuales para que las mismas puedan ser accedidas desde el lado del HPS. Dicha memoria es una RAM dual port, que es la memoria compartida entre la FPGA y el HPS. Luego, se lee dicha memoria desde el HPS, la cual previamente fue cargada con datos empaquetados en protocolo ASTERIX CAT240 desde el lado de la FPGA, para, finalmente, transferir el contenido de la misma a un arreglo y poder así enviar dichos datos por Ethernet a través del socket UDP.

Por último, se detallan los ensayos realizados los cuales son pruebas de medición de tiempos. Dichas pruebas son realizadas con el fin de conocer los tiempos de lectura y de transmisión de datos empaquetados en protocolo ASTERIX CAT240 y poder así descubrir el tiempo máximo que el HPS necesita para leer un dato en memoria, copiarlo y transmitirlo por Ethernet a través de un socket UDP dando como resultado una velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real.
Las diferentes pruebas realizadas están relacionadas entre sí, es decir, siguen una línea temporal dado que la segunda es predecesora de la primera, y así sucesivamente con las diferentes pruebas llevadas a cabo.

Vale destacar que se realizaron dos tipos de software y dos tipos de ensayos: los primeros fueron sin la lógica de sincronización entre la FPGA y el HPS, es decir, sin el *handshake* mientras que los segundos fueron con la realización del *handshake* entre la escritura y lectura de la memoria compartida, donde la FPGA se encarga de escribir y el HPS de leer.

2. Descripción del proyecto

2.1. Propuesta de proyecto

La implementación elegida para la realización del sistema propuesto es un SoC contenido dentro de una FPGA. La herramienta por la que se optó es el kit de desarrollo DE1-SoC, el cual contiene la FPGA Intel Cyclone V 5CSEMA5F31C6.

A continuación, se observa un diagrama en bloques del sistema propuesto.

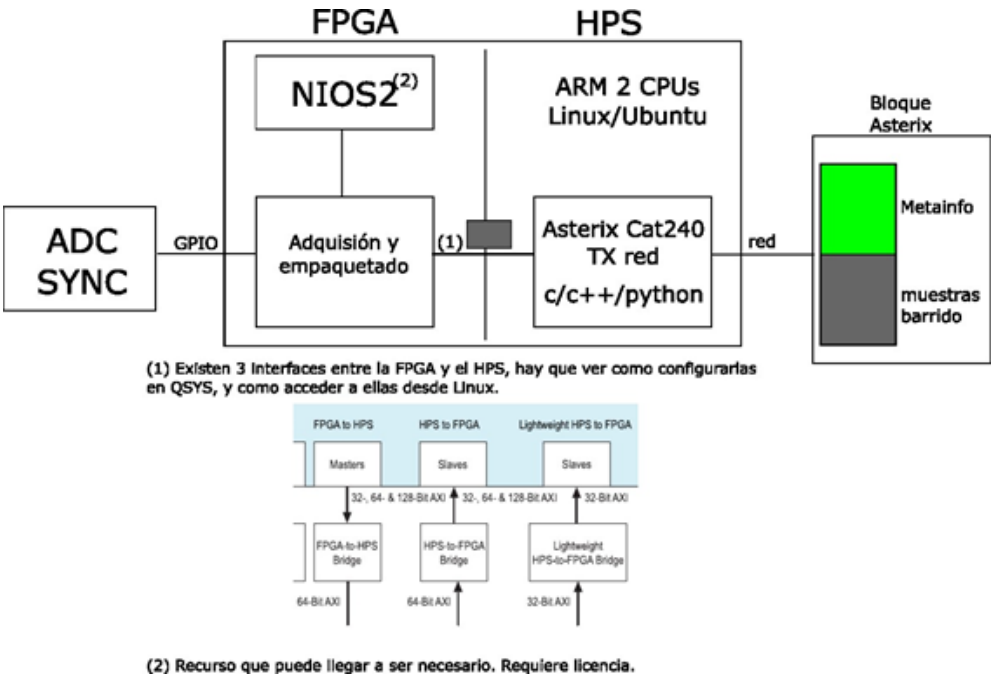

Figura 1. Diagrama en bloques del sistema propuesto.

Tal como se observa en la figura *ut supra*, la FPGA se compone de dos partes: una FPGA propiamente dicha y un HPS embebido en la FPGA.

La idea básica es, una vez que son capturadas las muestras de video RADAR crudo mediante un conversor AD conectado por GPIO a la FPGA y posteriormente empaquetadas en protocolo ASTERIX CAT240, realizar el *handshake* para que dichas muestras pasen de la FPGA al HPS y así poder realizar un empaquetamiento de las mismas en protocolo UDP para, finalmente, enviar el bloque a través de la red Gigabit Ethernet utilizando un bridge de comunicación entre la FPGA y el HPS.

En el Anexo B podrá encontrarse una breve reseña del protocolo ASTERIX CAT240. Por su parte, en el Anexo C podrá encontrarse un pequeño resumen del modelo OSI.

La ventaja principal es que el HPS ejecuta un OS dentro del mismo, un Linux (Ubuntu), el cual ya resuelve en su totalidad la parte de los protocolos de red.

A continuación, puede observarse en forma de diagrama las partes del proyecto en cuestión.

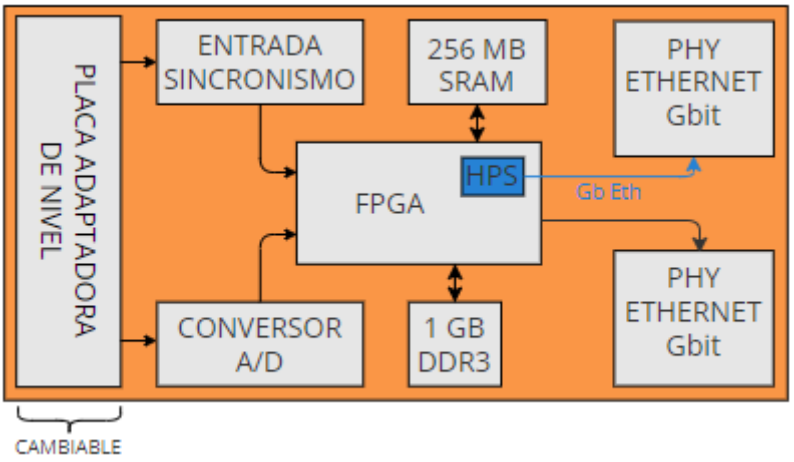

Figura 2. Diagrama en bloques del sistema propuesto. Partes del proyecto.

Más en detalle, se muestra en forma de diagrama la parte realizada por nosotros.

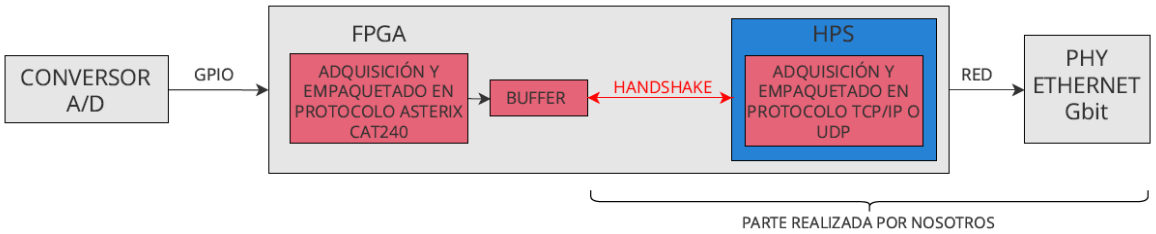

Figura 3. Diagrama en bloques del sistema propuesto. Parte del proyecto realizada por nosotros.

En resumen, las partes a resolver serían dos principalmente:

- Realización del *handshake* para la posterior adquisición de las muestras empaquetadas en protocolo ASTERIX CAT240.
- Empaquetado de dichas muestras en protocolo UDP dentro la FPGA, a su vez de implementar un bridge que se comunique con el HPS para así enviar dicho bloque a través de la red Gigabit Ethernet.

Tal como se mencionó anteriormente, el SoC, el cual está contenido dentro de una FPGA, formará parte de una red en la cual, dicho dispositivo, recibirá muestras de video RADAR crudo. Además, dicho dispositivo transmitirá muestras de video RADAR digitalizadas y empaquetadas en protocolo ASTERIX CAT240 a velocidades de Gigabit, o lo más cercana a ésta, utilizando UDP, a los dispositivos que las hayan solicitado.

Se detalla a continuación un diagrama de conexión generalizado de los dispositivos que forman parte de la red.

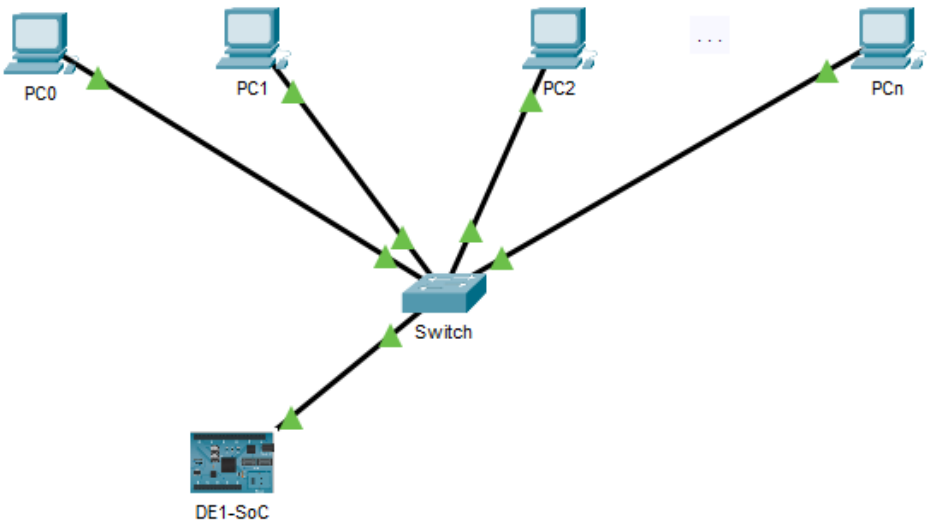

Figura 4. Diagrama de conexión generalizado. Dispositivos que forman parte de la red.

2.2. Puesta en marcha y configuración del kit DE1-SoC

Haciendo uso de la consola de Linux, se creó el archivo ejecutable, el cual se guardará en una tarjeta SD que luego será insertada en el kit de desarrollo DE1-SoC, con la finalidad de que el kit ejecute dicho archivo y pueda bootear así un OS Linux.

En el Anexo A, además de brindar una descripción general del kit de desarrollo mencionado, se describen las características del HPS, el proceso de booteo y la estructura del proyecto utilizada.

En la Figura 5 a continuación, se detallan los pasos seguidos, además de los comandos empleados en la consola de Linux, para realizar lo comentado previamente.

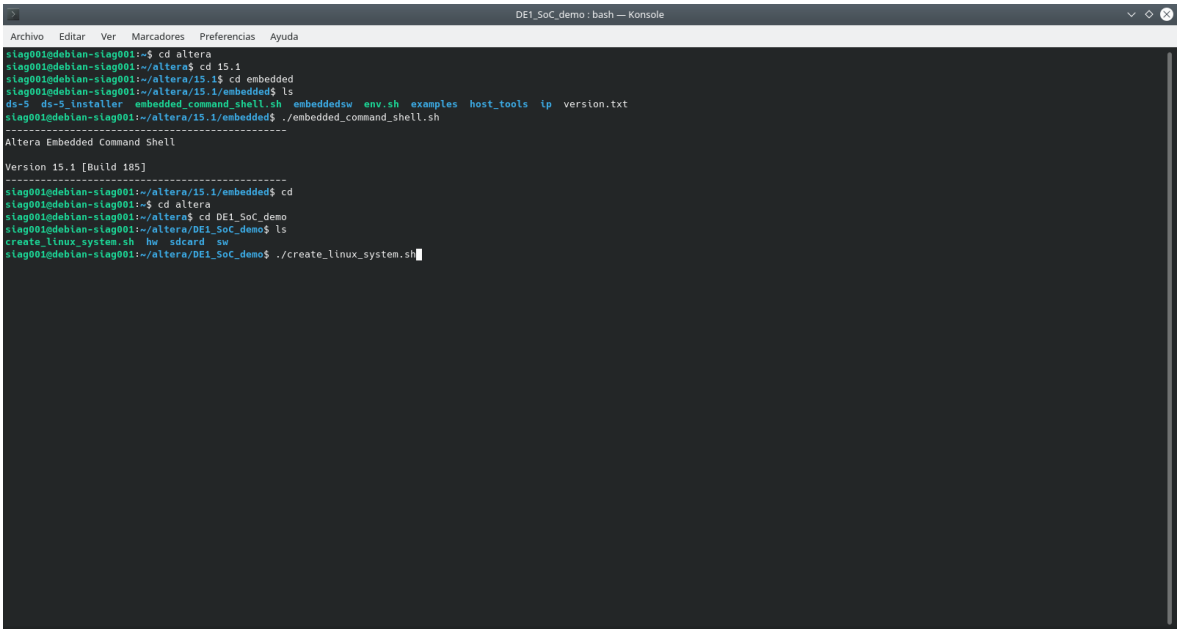

Figura 5. Creación del archivo ejecutable que se guardará en la tarjeta SD. Comandos empleados.

En primera instancia, se ejecutó el comando *./embedded_command_shell.sh*, el cual se encarga de buscar todos los archivos ejecutables relacionados a Altera. Un detalle importante a tener en cuenta es que todas las instrucciones provistas en la guía de referencia[1], las cuales además se comentan y describen en la presente sección del informe, deben de ser ejecutadas posterior a ejecutar en la consola de Linux el ejecutable *Altera Embedded Comman Shell*, el cual se encuentra en el directorio *altera_install_directory/version/embedded/embedded_command_shell.sh.*

Una vez ejecutado el script, haciendo uso del comando *./create_linux_system.sh*, se tuvieron dos errores relacionados a *yylloc*. La solución encontrada[2] radica en cambiar la línea *YYLTYPE yylloc* por la línea *extern YYLTYPE yylloc* en los archivos cuyos directorios se observan a continuación.

DE1_SoC_Demo → sw → hps → linux → source → scripts → dtc → dtc-lexer.lex.c
DE1_SoC_Demo → sw → hps → linux → source → scripts → dtc → dtc-lexer.lex.c_shipped

Una vez corregidos dichos archivos, los errores relacionados a *yylloc* fueron solucionados por lo que pudo compilarse de manera exitosa. El error que continuaba apareciendo no era un error de compilación en sí, tal como el que se tenía previamente, sino que era un error debido a que la tarjeta SD no estaba insertada.

Hecho esto, fue necesario vincular el usuario a grupos pertenecientes a Debian 11 con el fin de, posteriormente, poder ejecutar el script. Los comandos empleados para tal fin son los observados en la Figura 6 a continuación.

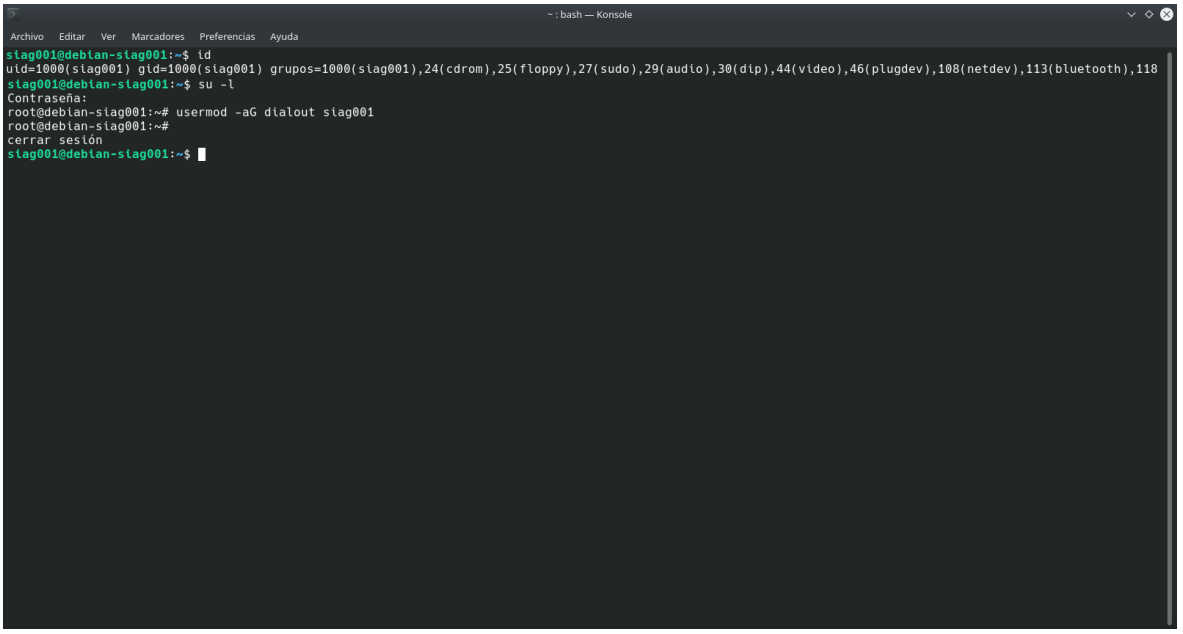

Figura 6. Vinculación de usuarios.

El comando *id* es utilizado para ver a qué grupos pertenece el usuario. Por otra parte, el comando *cat /etc/group* es utilizado para ver qué grupos tiene asociados el usuario y su contenido. El grupo 20 (*dialout*), por ejemplo, es empleado para utilizar los periféricos de la FPGA a través de los puertos de la PC.

Posteriormente, fue necesario desloguearse y volver a loguearse para que los cambios realizados tengan efecto.

En el script *create_linux_system.sh* se modificó el parámetro *sdcard_partition_size_linux*, el cual se encuentra en la línea 66, y se lo igualó a *4096M*.

```
65  sdcard_partition_size_fat32="32M"
66  sdcard_partition_size_linux="4096M"
```

Figura 7. Script create_linux_system.sh. Modificación de parámetros.

Además, se debe corroborar que el parámetro *fpga_device_part_number*, el cual se encuentra en la línea 19, coincida con el nombre de la FPGA que se esté utilizando. En nuestro caso, y por default, es la FPGA 5CSEMA5F31C6.

```
19  fpga_device_part_number="5CSEMA5F31C6" # 5CSEMA4U23C6
```

Figura 8. Script create_linux_system.sh. Modificación de parámetros.

Se comentaron, además, las líneas que van desde la 495 a la 499 ya que la compilación del Linux se realiza por única vez. Será necesario descomentar las líneas 495 y 496 a medida que se realicen cambios en el proyecto de *Quartus*.

```
495  #compile_quartus_project
496  #compile_preloader
497  #compile_uboot
498  #compile_linux
499  #create_rootfs
```

Figura 9. Script create_linux_system.sh. Modificación de parámetros.

En la Figura 10 a continuación puede observarse la ejecución del comando *lsblk*, el cual muestra las particiones que se tienen en el dispositivo.

```
DE1_SoC_demo : bash — Konsole
Archivo  Editar  Ver  Marcadores  Preferencias  Ayuda
siag001@debian-siag001:~$ lsblk
NAME   MAJ:MIN RM   SIZE RO TYPE MOUNTPOINT
sda      8:0    0 894,3G  0 disk
├─sda1   8:1    0   243M  0 part /boot/efi
├─sda2   8:2    0   343M  0 part /boot
├─sda3   8:3    0     1K  0 part
├─sda5   8:5    0  22,4G  0 part [SWAP]
├─sda6   8:6    0  33,2G  0 part /
└─sda7   8:7    0 838,2G  0 part /home
sdb      8:16   1  14,4G  0 disk
└─sdb1   8:17   1  14,4G  0 part
└─sdb1   8:17   1  14,4G  0 part
```

Figura 10. Comando lsblk.

La partición que nos interesa es *sdb*, que es la tarjeta SD que será insertada en el kit de desarrollo DE1-SoC, la cual almacenará el archivo ejecutable del OS Linux generado.

En resumen, los comandos necesarios para realizar en su totalidad el procedimiento descrito, es decir, crear el ejecutable del OS Linux y guardarlo en la tarjeta SD, son los observados en la Figura 11 a continuación.

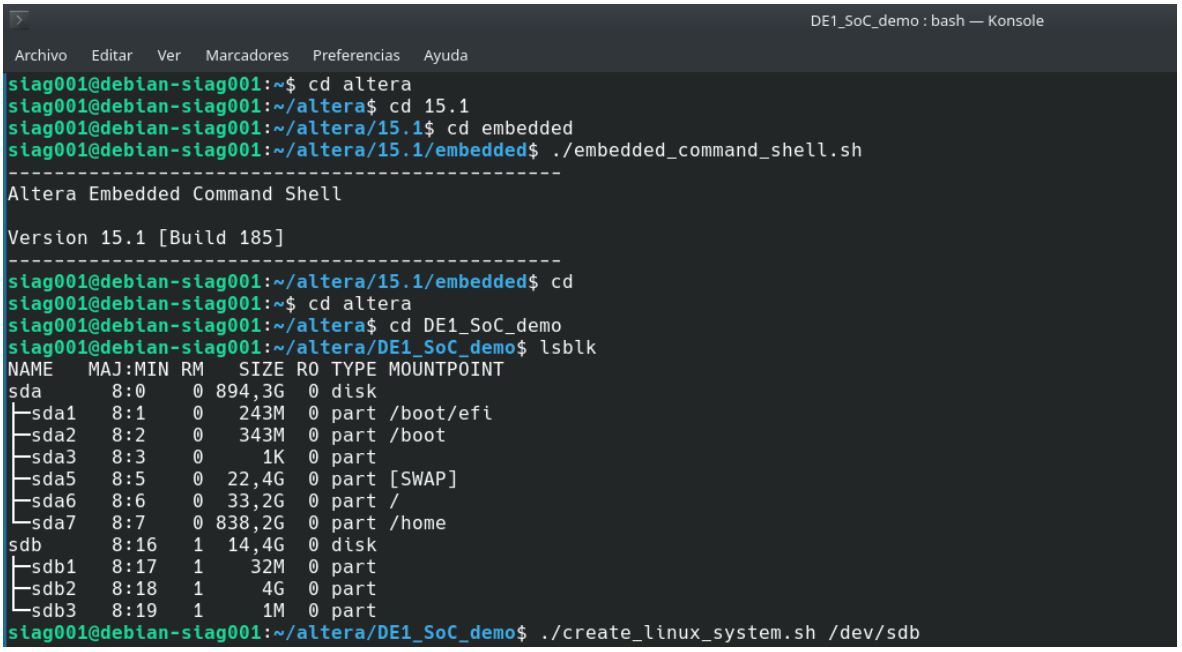

Figura 11. Creación del archivo ejecutable que se guardará en la tarjeta SD. Comandos empleados.

El script */create_linux_system.sh* realiza, de forma automática y tal como se observa en la Figura 12, las siguientes tareas:

- Compila el proyecto de *Quartus Prime*.
- Genera, configura y compila el *preloader*.
- Descarga, configura y compila el *u-boot*.
- Descarga, configura y compila el *Linux*.
- Descarga y configura el sistema de archivos raíz del *Ubuntu Core*.
- Particiona la tarjeta SD.
- Escribe la tarjeta SD.

Tal como puede observarse en la figura *ut supra*, se detalla todo el procedimiento seguido desde la ejecución del comando *./embedded_command_shell.sh*, el cual busca todos los ejecutables relacionados a Altera, hasta la ejecución del comando *./create_linux_system.sh*, script necesario para crear el ejecutable del OS Linux y guardarlo en la tarjeta SD que luego será insertada en la FPGA.
Algo a destacar es que el comando */dev/sdb*, comando concatenado al comando *./create_linux_system.sh*, es utilizado para guardar en la partición *sdb* el resultado de ejecutar el comando *./create_linux_system.sh*, que sería el archivo ejecutable del OS Linux.

En este punto, se tuvo un error en la compilación del script *create_linux_system.sh* y no pudo crearse el *u-boot*. Por lo tanto, para solucionarlo, se debió entrar al script *create_linux_system.sh* e ingresar los comandos enunciados a continuación a través de la consola de Linux.

gvim create_linux_system.sh
cd sw/hps/u-boot/
make distclean
git checkout "b104b3dc1dd90cdbf67ccf3c51b06e4f1592fe91"
less Makefile

make "socfpga_cyclone5_config"
less Makefile

Tal como se observa en la Figura 12 a continuación, el script *create_linux_system.sh* debería de hacerlo de forma automática pero, por alguna razón que se desconoce, no lo hace. De todas maneras, el haber ingresado los comandos mediante la consola de Linux soluciona el problema.

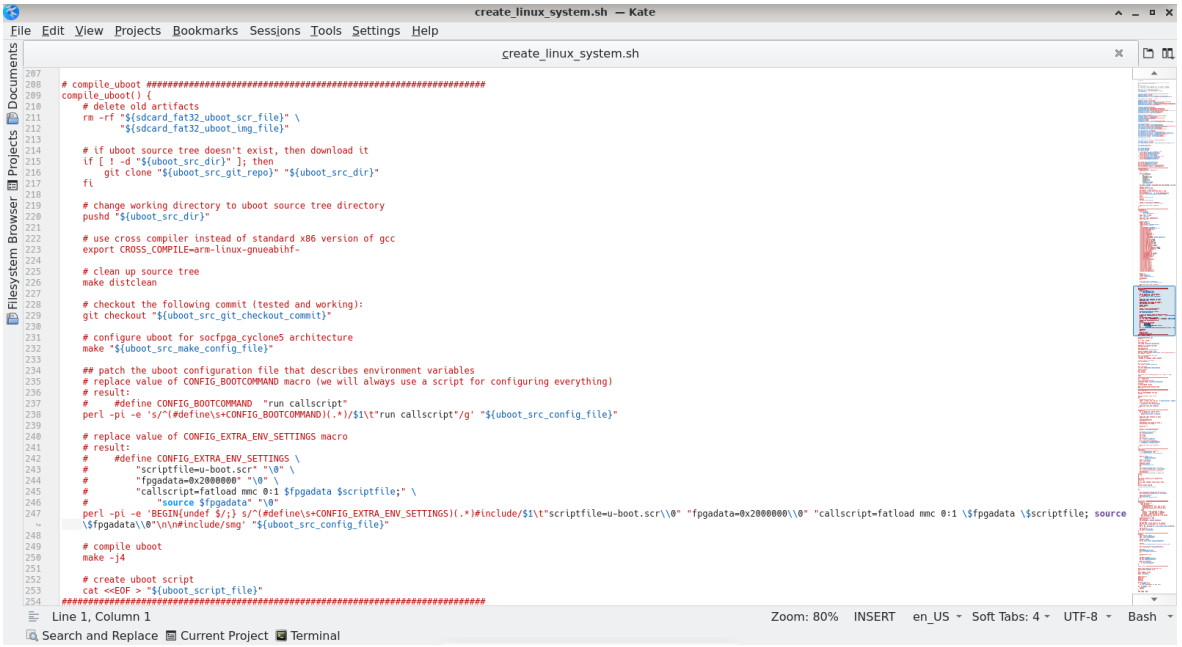

Figura 12. Script create_linux_system.sh. Sección u-boot.

Además, haciendo uso de la consola de Linux y empleando el comando *sudo apt install u-boot-tools*, se instalaron las herramienta necesarias para que la PC, una vez ejecutado el script *create_linux_system.sh*, cree y guarde en la tarjeta SD los archivos necesarios para que el kit DE1-SoC pueda bootear el OS.

Una vez realizado lo comentado anteriormente, se ejecutó nuevamente el script *create_linux_system.sh* mediante el comando *./create_linux_system.sh /dev/sdb* para que los archivos generados, los cuales se observan en la Figura 13 y en la Figura 14, se guarden en la tarjeta SD.

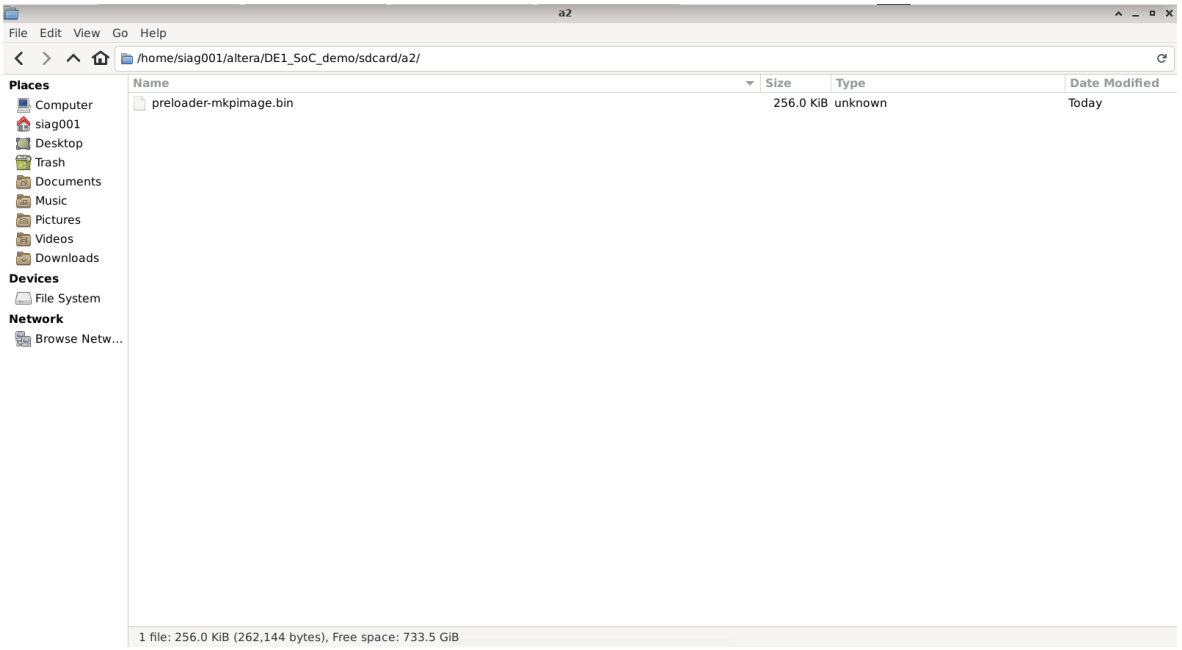

Figura 13. Script create_linux_system.sh. Archivos generados.

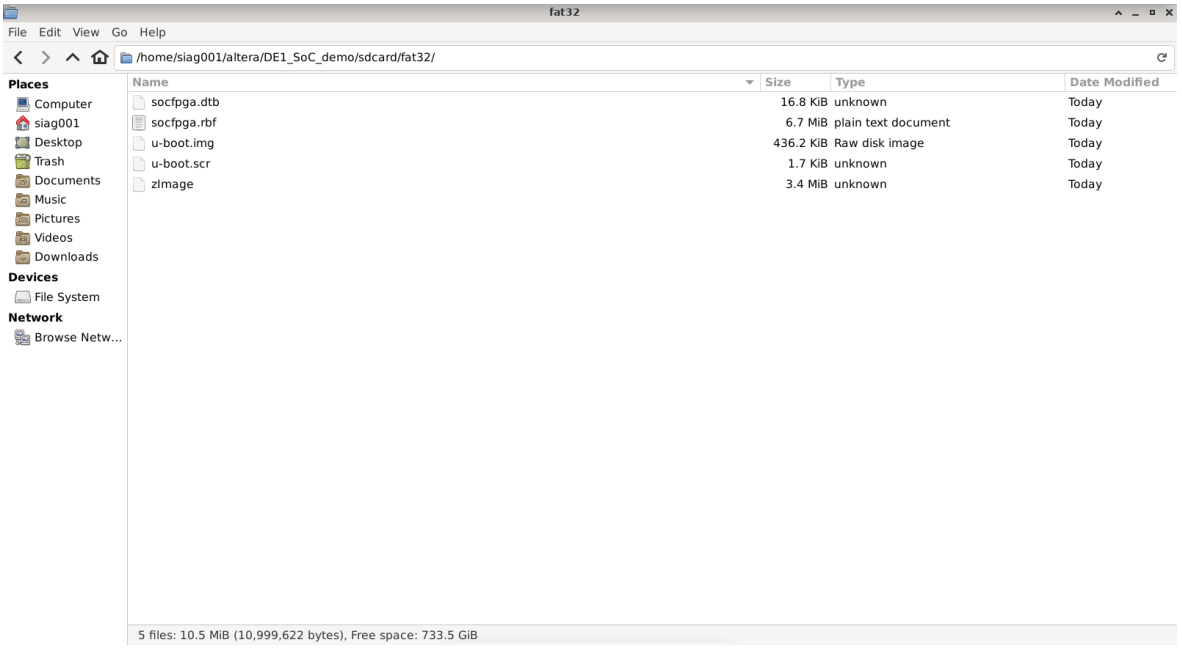

Figura 14. Script create_linux_system.sh. Archivos generados.

La ejecución del script y posterior creación de archivos, tal como se observa de las figuras *ut supra*, fue realizada con éxito.

Por lo tanto, se extrajo la tarjeta SD y posteriormente se la insertó en la FPGA.

Acto seguido, se encendió la FPGA y se la conectó a la PC mediante una interfaz serial, haciendo uso del terminal *CuteCom*, para poder así comunicarse con ésta.
El usuario (*username*) empleado es *sahand* mientras que la contraseña (*password*) es *1234*, tal como indica la guía de referencia utilizada[1].

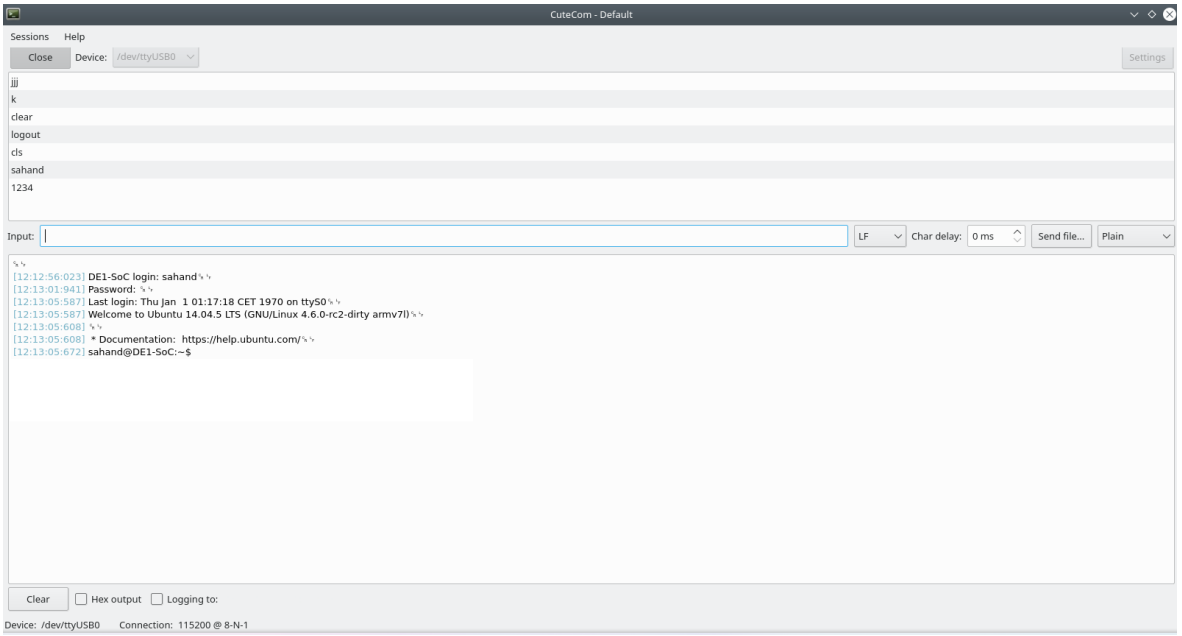

Figura 15. Conexión mediante interfaz serial entre la PC y la FPGA.

Tal como puede observarse en la Figura 15, el OS Linux Ubuntu fue booteado con éxito en la FPGA.

Luego, utilizando cables de red, se conecta tanto el kit DE1-SoC como la PC al switch para así poder tener acceso a la red de trabajo que, en nuestro caso, es *SIAG_INGE*.
Un detalle a tener en cuenta es que la conexión de dispositivos a la red de trabajo, en este caso la PC con la que se realiza toda la configuración, debe ser cableada ya que, si la misma es realizada de manera inalámbrica, la latencia es muy alta lo que dificulta en demasía la configuración por terminal.

Utilizando el terminal *CuteCom*, se instaló el *SSH Server* en el kit mediante el comando *sudo apt install openssh-server*. Posteriormente, se pide una contraseña, la cual es *1234*. Con el servidor SSH se accede de manera remota al kit DE1-SoC desde la PC.

Por la manera en la que está configurado el kit de desarrollo, una vez que el mismo se conecta a la red de trabajo, se le asigna una dirección IP por DHCP. Por lo tanto, haciendo uso de la terminal *CuteCom*, empleando el comando *ip addr*, puede conocerse la dirección IP asignada al kit de desarrollo. En nuestro caso, la misma es 192.168.15.170. Luego, desde la PC, se le da click derecho al ícono de red que se encuentra en la barra de tareas y se accede a *Edit Connections* .... Hecho esto, en la sección *Ethernet*, se ingresa a *Wired Connection* y en la sección *IPv4 Settings* se agrega una dirección IP estática. En nuestro caso, la misma es 192.168.7.20. El poder asignarle dos direcciones IP a un mismo dispositivo es una gran ventaja en Linux.

Una vez que fue instalado con éxito el *SSH Server* se debe ingresar a la consola de Linux y, por medio del comando *ssh sahand@192.168.15.170*, acceder al kit DE1-SoC.
Un detalle a aclarar es que la primera vez que se realiza este procedimiento, tal como se observa en la Figura 16 a continuación, se pregunta si se desea realizar la conexión. En caso afirmativo, aceptar ingresando por consola la palabra *yes* (no ingresar *y*, como suele ser habitualmente). Posteriormente, se solicitará una contraseña, la cual es *1234*.

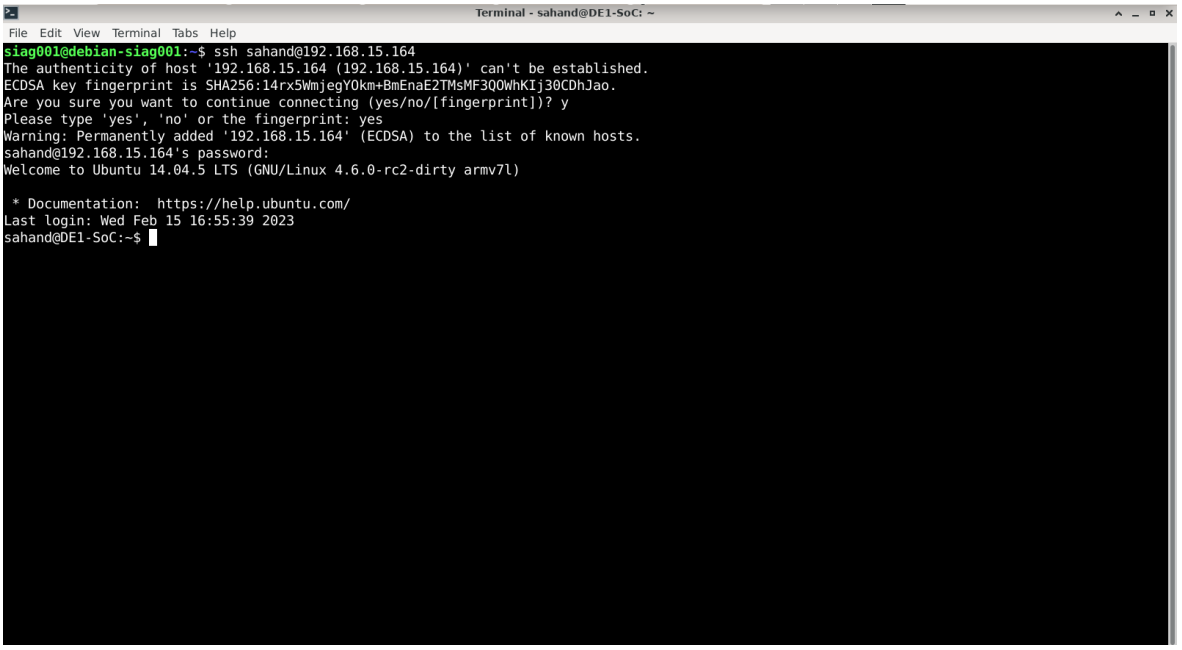

Figura 16. Acceso a la FPGA utilizando SSH. Dirección IP asignada por DHCP.

El objetivo de la presente configuración es asignarle al kit DE1-SoC una dirección IP mediante DHCP y otra dirección IP de manera estática, de igual manera que se realizó con la PC. La dirección IP asignada por DHCP es la utilizada para tener acceso a Internet en ambos dispositivos y realizar así las descargas y actualizaciones necesarias mientras que la dirección IP fija o estática es la utilizada para la comunicación entre dispositivos a la hora de realizar las configuraciones.

Por lo tanto, haciendo uso de la consola de Linux y empleando el comando *sudo nano /etc/network/interfaces*, se configuran las interfaces del kit DE1-SoC.

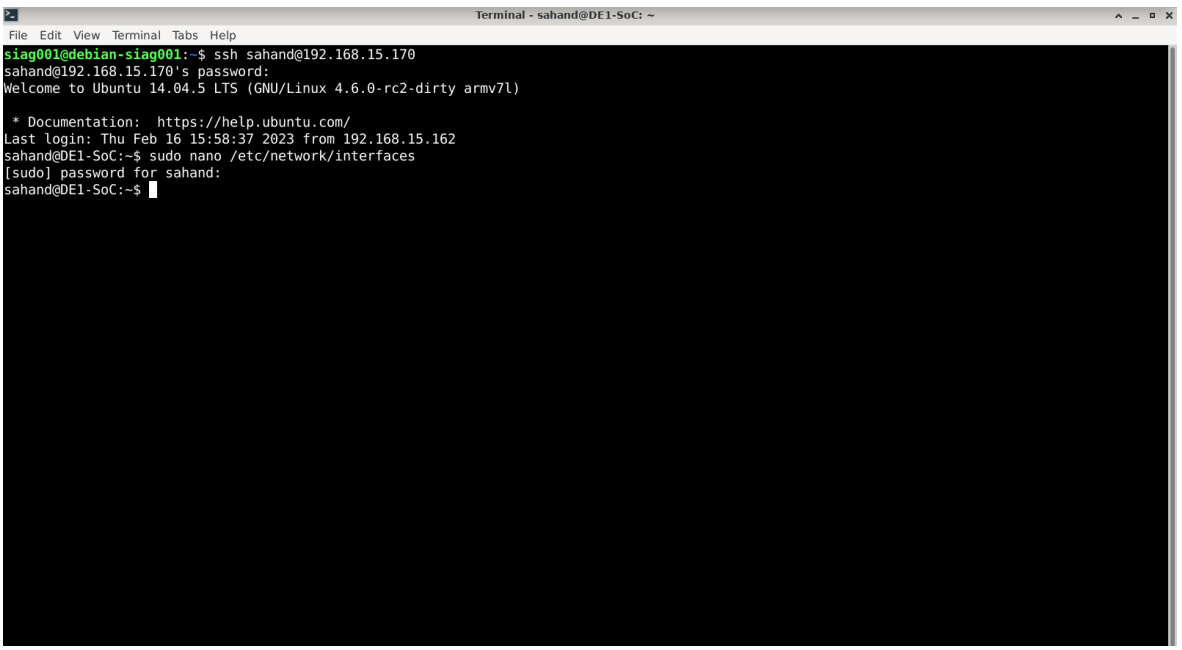

Figura 17. Kit de desarrollo DE1-SoC. Acceso a la /etc/network/interfaces.

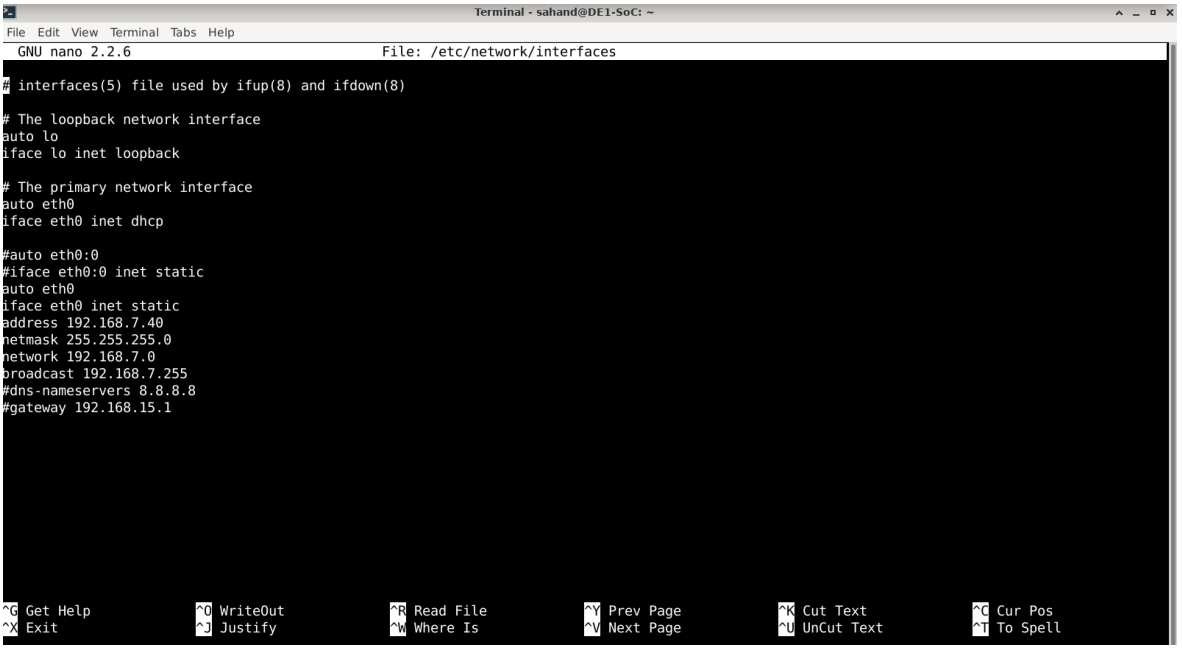

Figura 18. Kit de desarrollo DE1-SoC. Acceso a la /etc/network/interfaces.

Finalmente, se deben reiniciar los dispositivos para que la configuración realizada tenga efecto. Para ello, se emplea el comando *sudo reboot* a través de la consola de Linux.

En este punto, ya puede accederse al kit DE1-SoC mediante SSH utilizando la dirección IP fija o estática, tal como se observa en la Figura 19 a continuación.

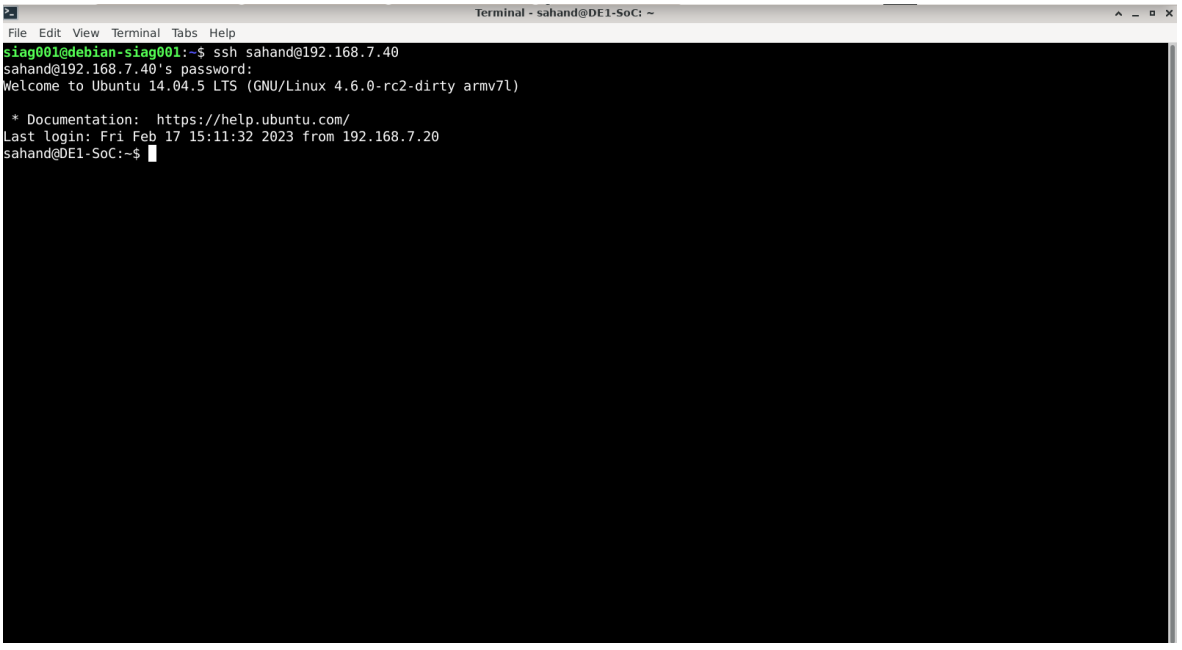

Figura 19. Acceso a la FPGA utilizando SSH. Dirección IP asignada por DHCP.

Empleando la consola de Linux, mediante el comando *sudo apt install mc htop build-essential*, se instalaron las siguientes herramientas en el kit de desarrollo DE1-SoC:

- *mc*, el cual es un administrador de archivos.
- *htop*, el cual es un administrador de procesos.

- *build-essential*, las cuales son todas las herramientas de compilación necesarias. En pocas palabras, instala lo básico para poder comenzar a escribir y así programar en Lenguaje C.

Una vez comenzada la instalación de dichas herramientas, la misma es interrumpida por el error *Unable to locate*. Para solventar dicho inconveniente, en primera instancia, se empleó el comando *sudo apt update* por medio de la consola de Linux, el cual es utilizado para corroborar si existe alguna actualización disponible para así instalarla. Hecho esto, se intentó nuevamente instalar dicha herramientas pero el error subsistía. Luego, por medio del comando *sudo nano /etc/apt/sources.list*, se habilitaron los repositorios para que el sistema pueda buscar y descargar las actualizaciones requeridas.

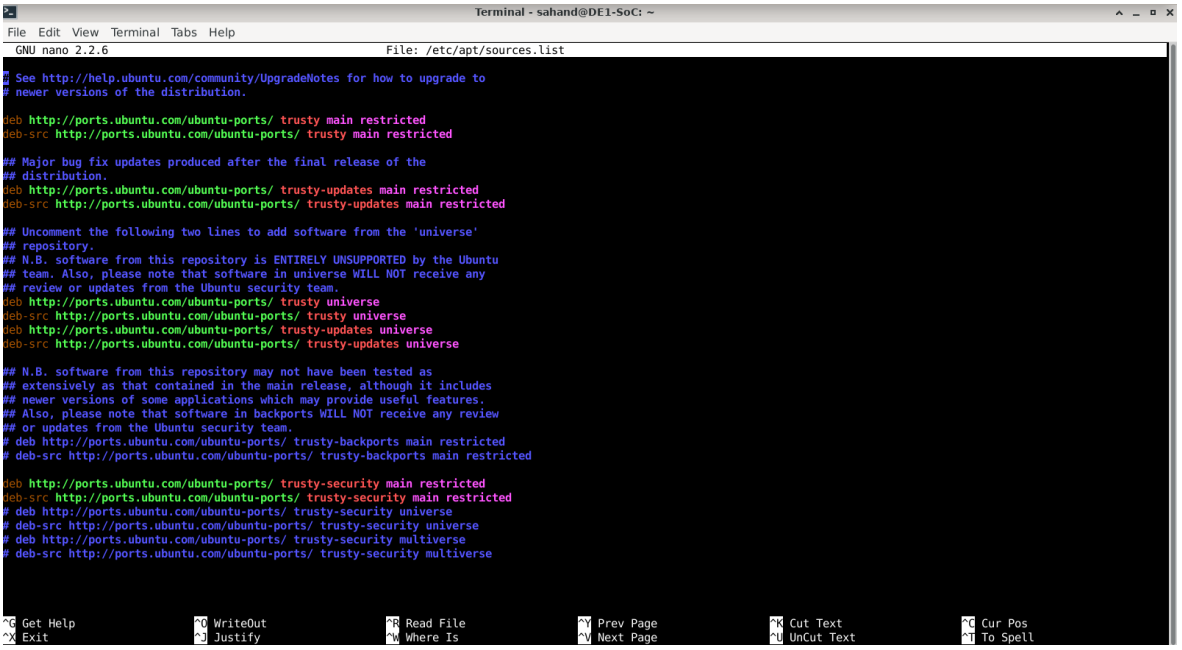

Figura 20. Kit de desarrollo DE1-SoC. Acceso a la /etc/apt/sources.list. Repositorios universe.

Tal como puede observarse de la figura *ut supra*, se habilitaron (se descomentaron eliminando el #) los últimos 4 repositorios *universe*.

Hecho esto, se volvió a ejecutar el comando *sudo apt update* seguido del comando *sudo apt install mc htop build-essential* para instalar así las herramientas requeridas con éxito y sin error alguno. Además, por medio del comando *sudo apt install rsync*, se instaló una herramienta para transferencias de archivos, la cual es muy utilizada para realizar back-up.

Posteriormente, volviendo a hacer uso de la consola de Linux, se realizó la instalación de dos software sobre la PC: el primero, a través del comando *sudo apt install cmake*, fue *CMake*. Dicho software genera los archivos de compilación de otro sistema. En nuestro caso, será utilizado en conjunto con el software *QtCreator*. El segundo software instalado fue *QtCreator*, haciendo uso del comando *sudo apt install qtcreator*. Una vez instalado, tuvo que ser configurado para así poder vincular y ejecutar los archivos en el kit DE1-SoC. Para ello, se instalaron el compilador y el depurador haciendo uso de la consola Linux y de los comandos *sudo apt install g++-9-arm-linux-gnueabihf* y *sudo apt install gbd-multiarch* respectivamente. El compilador es propio para el procesador que se tiene en el kit de desarrollo DE1-SoC, el cual crea los archivos binarios mediante el software *QtCreator* y los manda al kit. Por su parte, el depurador del software *QtCreator* se conecta por red al

depurador del kit. Por otro lado, mediante el comando *sudo apt install gdbserver*, se instaló en el kit un servidor que se ejecuta con el archivo *.bin* y el número de puerto. Dicho servidor espera y ejecuta comandos de una PC. La ventaja que se tiene con esto es que se pueden mandar comandos por consola y ejecutarlos desde el kit de desarrollo DE1-SoC.

Posteriormente, una vez realizada las instalaciones, mediante el comando *sudo reboot*, reiniciamos el OS para que los cambios tengan efecto.

Un detalle importante a mencionar es que se debe habilitar el usuario *root* en *SSH*[3][4] debido a que, una vez que se ejecute el programa posterior a haber realizado toda la configuración necesaria explicada a continuación, se pedirá acceso a lugares con privilegios como, por ej. a la memoria DMA. Es por esta razón que es necesario brindar acceso *root*. Esta configuración debe realizarse del lado del servidor, es decir, debe hacerse del lado del kit DE1-SoC.

Una vez iniciado el software *QtCreator*, nos dirigimos a *Tools → Options*.

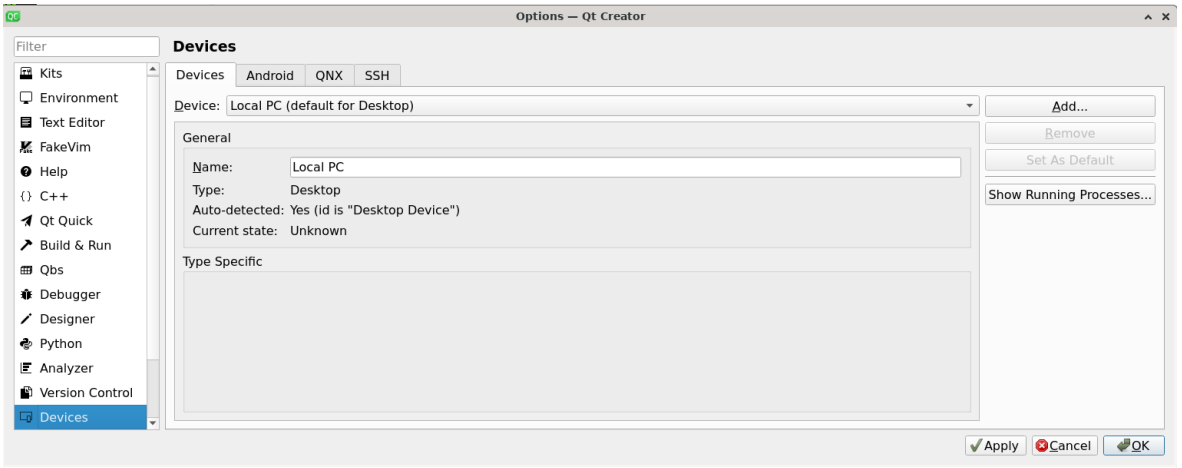

Figura 21. Software QtCreator. Configuración.

En la parte de *Devices*, en la viñeta *Devices*, se debe de agregar un dispositivo mediante la opción *Add*. El tipo de dispositivo que se debe seleccionar es *Generic Linux Device*.

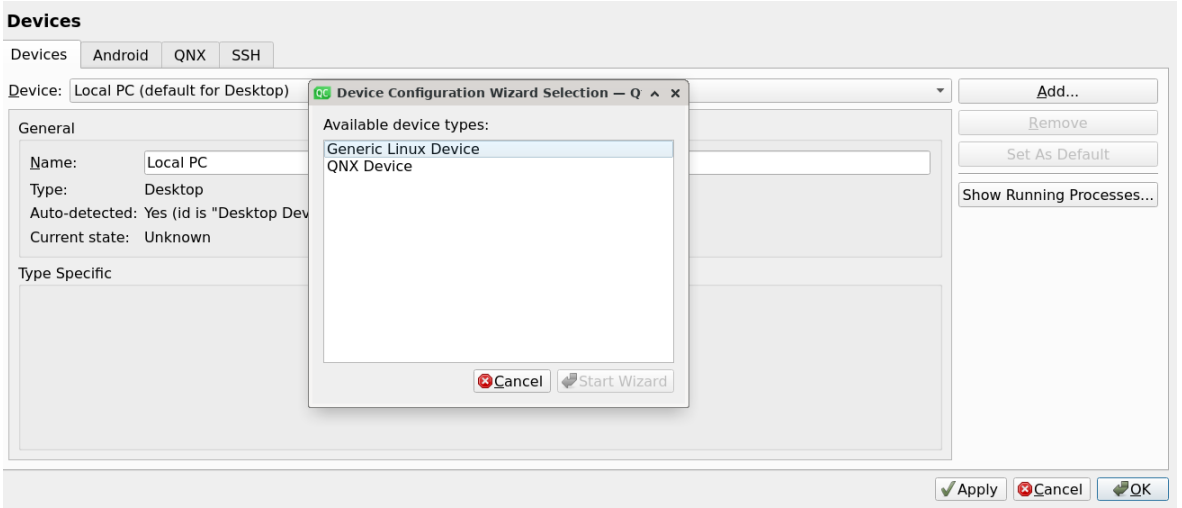

Figura 22. Software QtCreator. Configuración.

A continuación, se le debe colocar un nombre al dispositivo. En nuestro caso, se le asignó el nombre *DE1-SoC*. Luego, le colocamos la dirección IP fija o estática asignada anteriormente que, es nuestro caso, es 192.168.7.40 y, posteriormente, un nombre de usuario para loguearse al dispositivo. En nuestro caso, el nombre de usuario empleado es *root*.

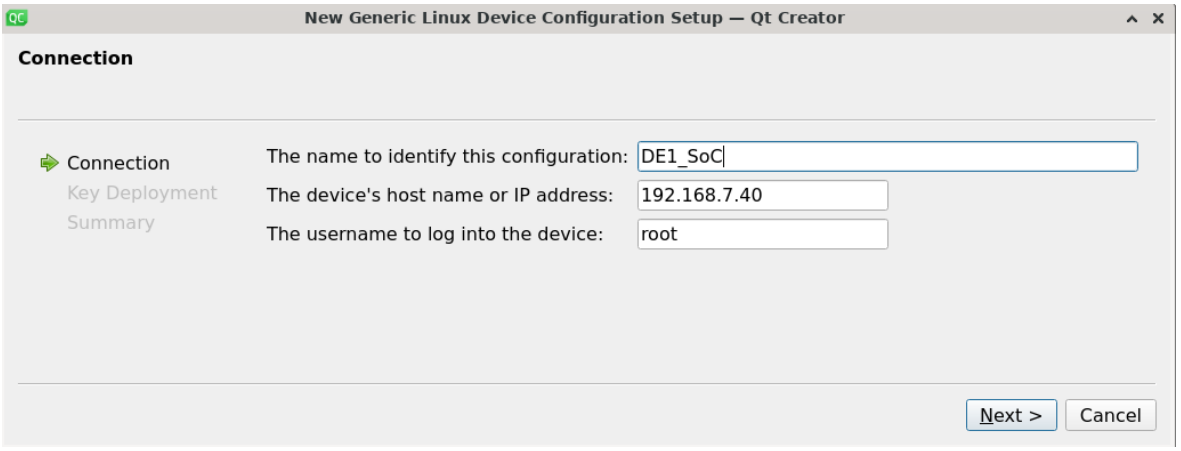

Figura 23. Software QtCreator. Configuración.

Hecho esto, se debe crear una *key*. Para ello, se debe clickear en *Create New Key Pair*. Al hacerlo, se despliega una ventana, en la cual debemos seleccionar *Generate And Save Key Pair*.

Figura 24. Software QtCreator. Configuración.

Posteriormente, dicha ventana se cierra y, en la ventana en la que se estaba previamente, se debe seleccionar *Deply Public Key*.

Figura 25. Software QtCreator. Configuración.

Al hacerlo, aparece una ventana emergente en la cual se solicita la contraseña del nombre de usuario empleado, la cual es *1234*.

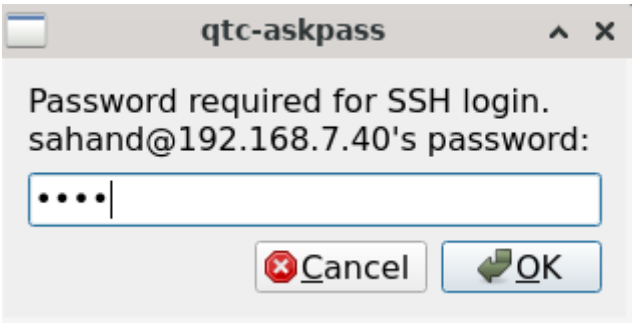

Figura 26. Software QtCreator. Configuración.

Si todo el procedimiento descrito anteriormente fue realizado con éxito, aparecerá una ventana al respecto. Para ir finalizando, se debe clickear en *Next* seguido de *Finish*. Luego, de forma automática, se realiza un test de conectividad.

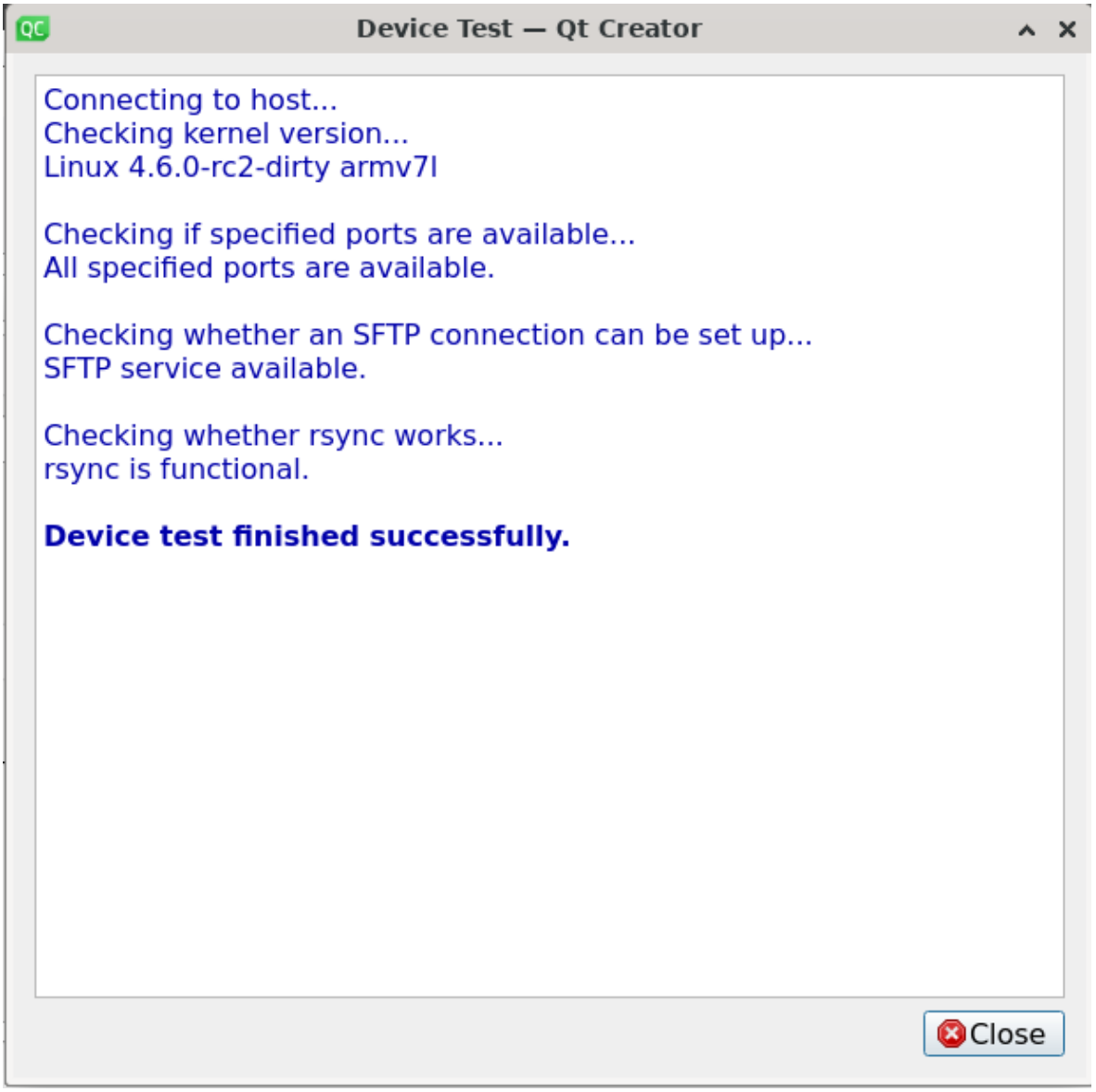

Figura 27. Software QtCreator. Configuración.

Finalmente, clickeamos en *Apply*.

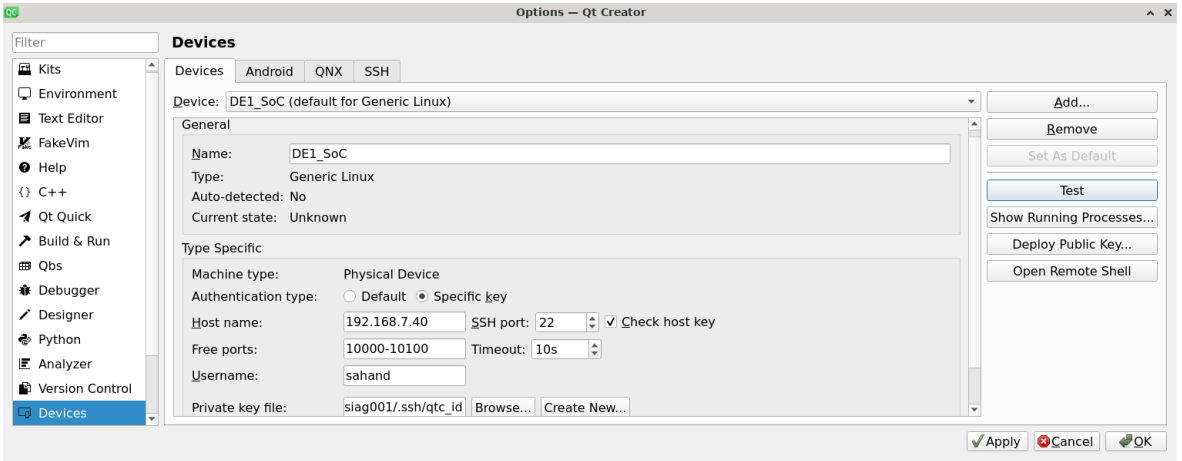

Figura 28. Software QtCreator. Configuración.

En la parte de *Kits*, en la viñeta *Debuggers*, se debe agregar un debugger mediante la opción *Add*.

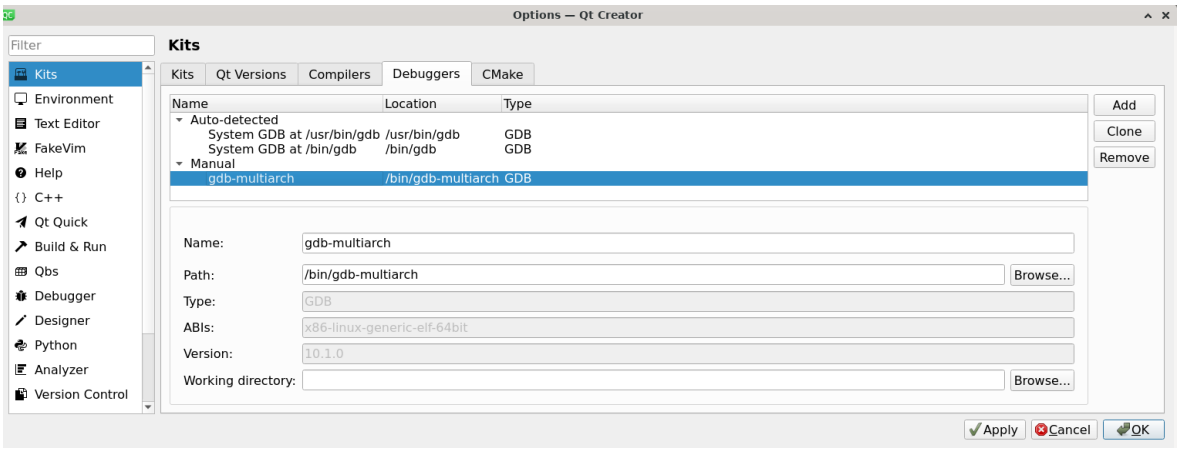

Figura 29. Software QtCreator. Configuración.

En la parte de *Kits*, en la viñeta *Compilers*, debe de aparecer el compilador que fue instalado previamente.

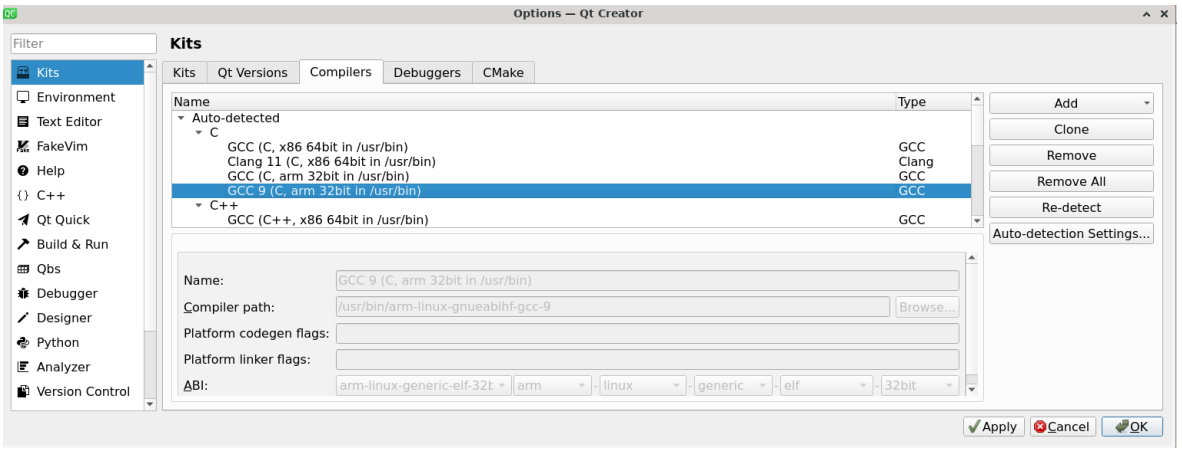

Figura 30. Software QtCreator. Configuración.

Una vez realizada la configuración comentada, en la parte de *Kits*, en la viñeta *Kits*, se debe asociar al *device* creado inicialmente con el compilador y el debugger.

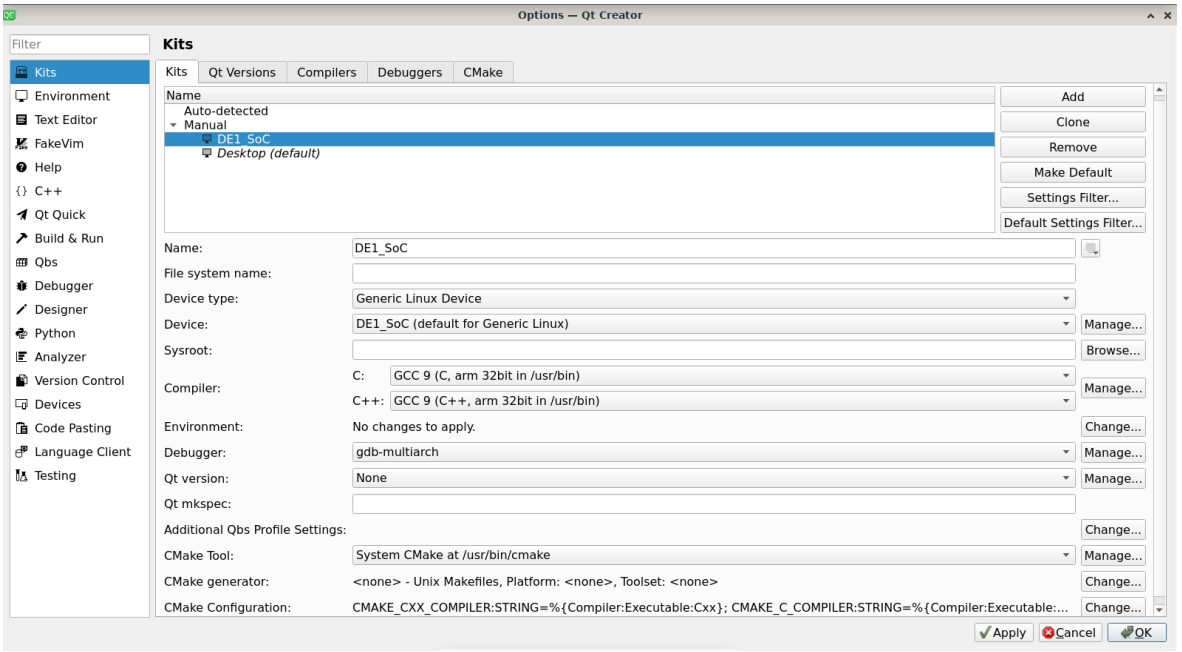

Figura 31. Software QtCreator. Configuración.

Posteriormente, para que el debugger pueda ser ejecutado, se realiza la configuración mostrada a continuación.

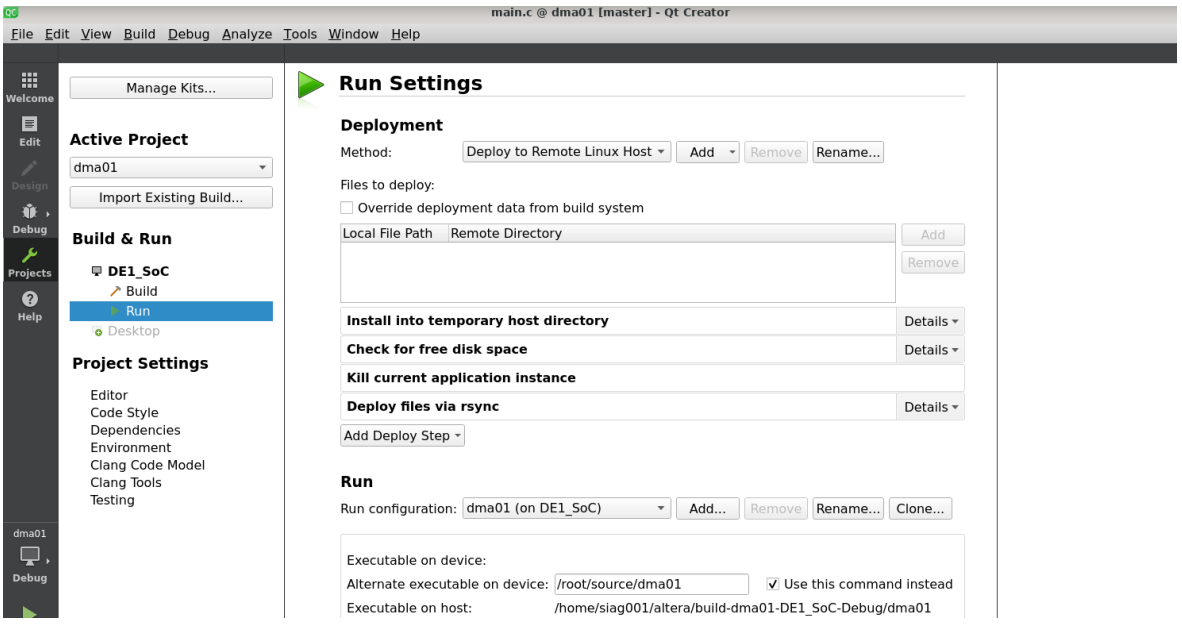

Figura 32. Software QtCreator. Configuración.

Una vez abierto el proyecto *dma01*, utilizando el software *QtCreator*, se accede a *CMakeLists.txt* para setear la ruta de acceso en donde, una vez compilado y realizado el *deploy* del proyecto, se guarda el ejecutable del mismo en el kit DE1-SoC. Una vez que se abre el *QtCreator*, el archivo *CMakeLists.txt* se ejecuta automáticamente, definiendo así la ruta mencionada.

Luego, se debe ingresar a la consola de Linux y por medio del comando *ssh root@192.168.7.40* se accede al kit DE1-SoC. Una vez ahí, mediante el comando *mkdir*, se

crea la carpeta o directorio *source* que es en donde se guarda el ejecutable creado por *QtCreator* luego de haber compilado el proyecto y realizado el *deploy* del mismo.

Para comenzar con la configuración del HPS, se ejecutó el *Qsys*. El mismo es abierto haciendo uso de la consola Linux, mediante el comando *quartus &*, tal como se observa en la Figura 33 a continuación.

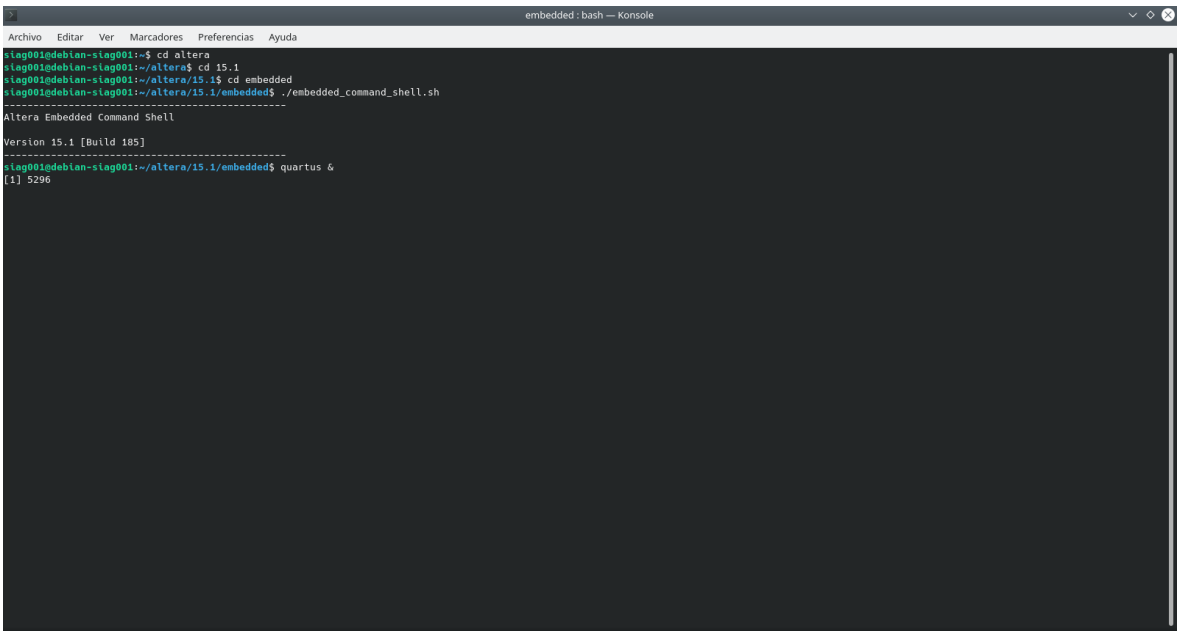

Figura 33. Ejecución del comando quartus &.

En pocas palabras, el *Qsys* es una herramienta gráfica que se utiliza para el diseño digital de hardware. Dicha herramienta contiene procesadores, memorias, interfaces de I/O, timers, etc. La herramienta *Qsys* le permite al usuario, mediante una GUI, elegir los componentes deseados y automáticamente genera el hardware para conectar dichos componentes.

En este punto, se tuvo un error al momento de compilar el proyecto, tal como se observa en la Figura 34 a continuación.

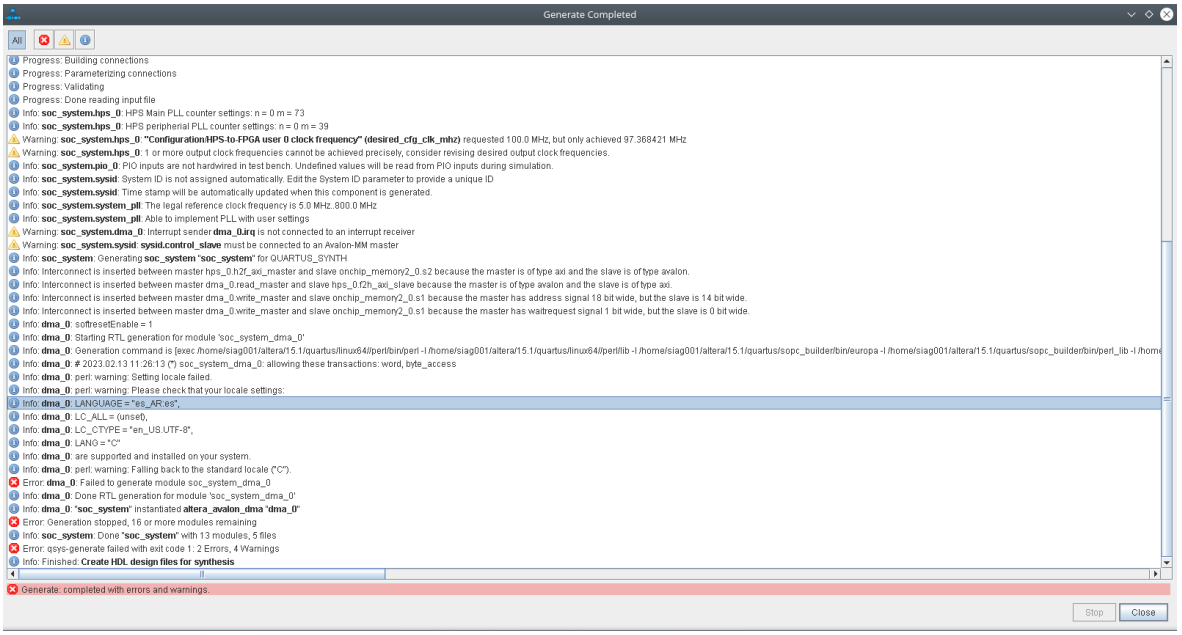

Figura 34. Compilación del proyecto en la herramienta Qsys del software Quartus. Error arrojado.

Tal como puede observarse de la figura *ut supra*, los errores son debidos al *locate*, es decir, al idioma (el OS de la PC estaba en Español y el *Qsys* lo requería en Inglés). Por lo tanto, para solventar dicho inconveniente, haciendo uso de la consola de Linux, se cambió el idioma[5] del OS Debian 11. Posteriormente, se reinició el mismo para que los cambios surtan efecto. Una vez reiniciado el equipo, la interfaz gráfica del OS no arrancaba. Luego de realizados varios intentos para solucionar la problemática enfrentada, se optó por cambiar de entorno gráfico[6]. En este punto, volvió a reiniciarse el OS. Una vez iniciado el mismo, se observó el cambio en el idioma. Luego, se generó el subsistema (*SoC subsystem*) en *Qsys* y, posteriormente, se lo instanció en el proyecto.

Hecho esto, se volvió a compilar de manera exitosa el *Qsys* por lo que, en este punto, ya se puede empezar a trabajar sobre el HPS.

Como se necesita al HPS para poder acceder a los periféricos que forman parte de la estructura de la FPGA, se debe de generar un *header* el cual, posteriormente, debe ser instanciado en el proyecto de *Qsys*. Para ello, se necesita entrar a la carpeta *cd/altera/DE1-SoC_demo/hw/quartus* y ejecutar, mediante la consola de Linux, el comando *sopc-create-header-files soc_system.sopcinfo --single hps_soc_system.h --module hps_0*. Tal como puede observarse, el nombre del *header* generado es *hps_soc_system.h*.

Un detalle importantísimo a tener en cuenta, previo a realizar cualquier ensayo, es que se debe de poner el switch MSEL, el cual se encuentra en la parte inferior del kit de desarrollo DE1-SoC, en “00000”[1].

Además, cada vez que se desea realizar un ensayo sobre el kit de desarrollo DE1-SoC, se debe grabar la tarjeta SD con el *preloader* y el archivo *.rbf*. Por su parte, el archivo *.rbf* es el archivo con el que el HPS programa la FPGA. El software *Quartus* genera el *SRAM Object File* (*.sof*), archivo necesario para programar la FPGA. El archivo *.sof* emplea el *header*, el cual debe volver a generarse solo sí se realizó algún cambio en el *Qsys*. Como nosotros trabajamos con el HPS, necesitamos el archivo *RAW Binary File* (*.rbf*) para programar la FPGA. La conversión del archivo *.sof* a *.rbf* la realiza de forma automática,

una vez que es ejecutado, el script *create_linux_system.sh* pero debe de grabarse a mano en la tarjeta SD.

Tal como se observa en la figura *ut infra*, la partición de la tarjeta SD que contiene el *preloader* es la *sdb3*. Puede notarse que la misma tiene el identificador *a2* y el tipo es *unknown*.

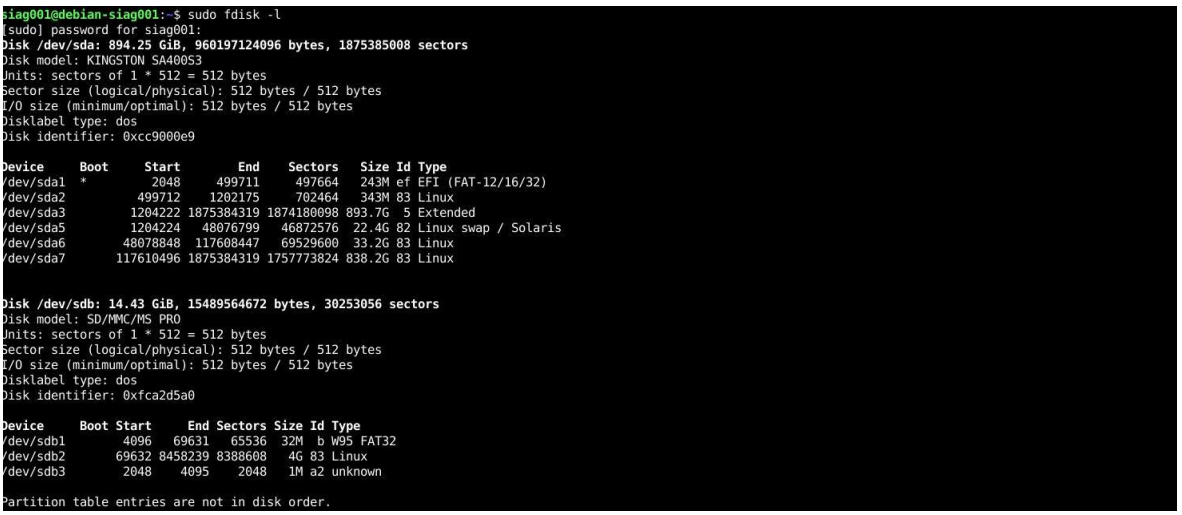

Figura 35. Partición de la tarjeta SD. Ubicación del preloader.

Por su parte, en la Figura 36 a continuación se observa la ruta en donde debe de grabarse el *preloader* además del comando utilizado para hacerlo, el cual es *sudo dd if=preloader-mkpimage.bin of=/dev/sdb3 bs=64k seek=0*, seguido del comando *sync* que es utilizado para transferencia de archivos.

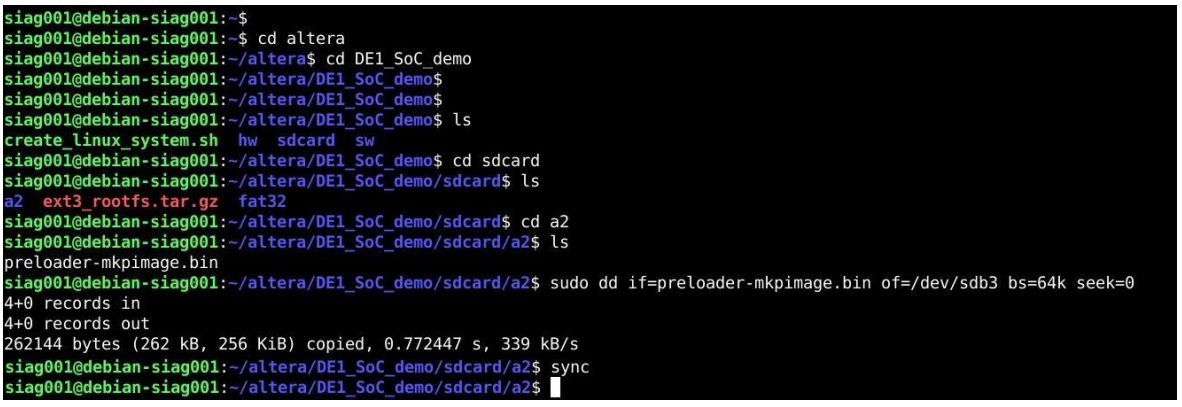

Figura 36. Grabado del preloader en la ruta de acceso correspondiente.

2.3. Software (sin la implementación del handshake entre la FPGA y el HPS)

En esta sección del informe se comenta en detalle el software realizado para la comunicación entre la FPGA y el HPS, el cual se divide fundamentalmente en dos partes: una realizada en *Qsys*, en la cual se instancia todo el hardware necesario para la comunicación entre la FPGA y el HPS, y otra parte realizada en *QtCreator*, la cual se utiliza para programar el HPS.

A grandes rasgos, el software realizado se encarga, en primera instancia, de mapear las direcciones de la FPGA a una memoria virtual para que las mismas puedan ser accedidas desde el lado del HPS. Luego, se escribe la memoria de la FPGA en su totalidad (8192 [bytes]) con datos fijos empaquetados en protocolo ASTERIX CAT240 y posteriormente se la lee desde el HPS. Luego, se transfiere el contenido de la memoria a un arreglo para, finalmente, enviar dichos datos por Ethernet a través del socket UDP.

2.3.1. Parte Qsys

El *Qsys* es una herramienta gráfica que se utiliza para el diseño digital de hardware. Dicha herramienta contiene procesadores, memorias, interfaces de I/O, timers, etc. La herramienta *Qsys* le permite al usuario, mediante una GUI, elegir los componentes deseados y automáticamente genera el hardware para conectarlos.

La herramienta *Qsys* permite diseñar el sistema y la comunicación entre el HPS y la FPGA. El HPS y la FPGA están conectadas a través de una serie de *AXI Bridges*. En este caso, nosotros trabajaremos desde el lado del HPS por lo que, para la comunicación entre ellos, se utilizan dos *bridges*:

- HPS-to-FPGA bridges (*h2f*).
- Lightweight HPS-to-FPGA bridge (*lwh2f*).

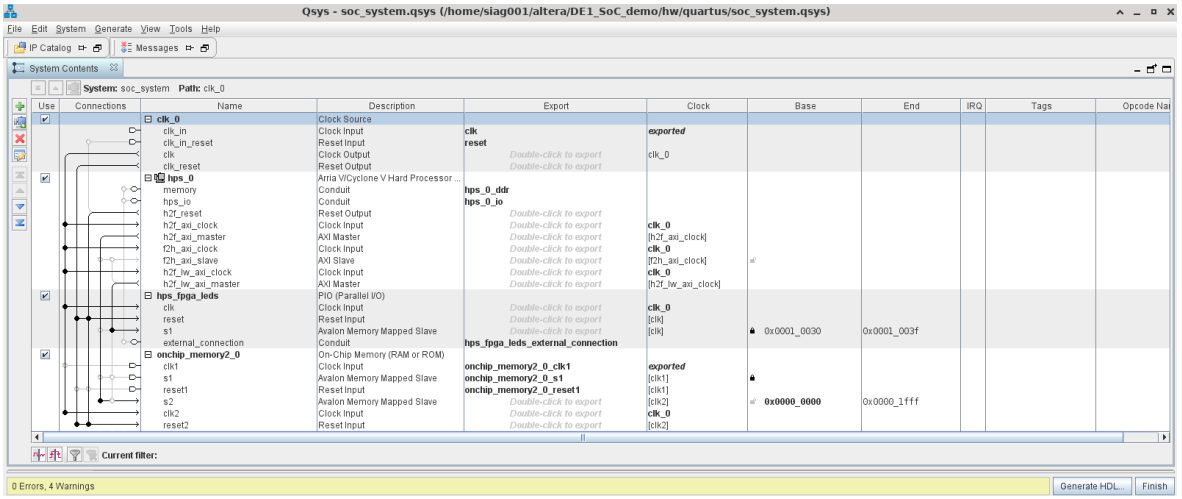

Figura 37. Qsys. Componentes instanciados.

Una vez que todo el hardware ha sido configurado correctamente, la comunicación entre el HPS y la FPGA se programa a través de una aplicación en C mapeada en memoria. Dicho mapeo de memoria permite que la CPU vea y acceda al espacio de direcciones de la FPGA, la cual contiene los componentes, para poder así leer/escribir información según sea necesario.

La aplicación en C desarrollada utiliza una API para enviar, o recibir, datos de escritura a, y desde, direcciones de memoria especificadas.

2.3.2. Parte QtCreator

Una vez configurado el hardware para la comunicación entre el HPS y la FPGA a través de la herramienta gráfica de diseño digital de hardware *Qsys*, se procede a programar el HPS.

Como se necesita que el HPS acceda a los periféricos que forman parte de la estructura de la FPGA, lo primero que se debe de realizar es la instanciación en el proyecto del *header*, el cual es generado una vez realizada la configuración en *Qsys*, mediante *#include "hps_soc_system.h"*.

```
16      #include "hps_soc_system.h"
```

Figura 38. QtCreator. Instanciación del header generado en Qsys.

Luego, se definen todas las variables y funciones necesarias para realizar los *bridges* entre el HPS y la FPGA.

```
18      // ======================================
19      #define H2F_AXI_MASTER_BASE   0xC0000000
20      // main bus; scratch RAM, conectada a h2f_axi_master
21      #define FPGA_ONCHIP_BASE      ONCHIP_MEMORY2_0_BASE
22      #define FPGA_ONCHIP_SPAN      ONCHIP_MEMORY2_0_SPAN
23      // h2f bus
24      // RAM FPGA port s2
25      // main bus addess 0x0800_0000
26      void *h2f_axi_master_virtual_base;
27      volatile unsigned int * sram_ptr = NULL ;
28      void *sram_virtual_base;
29      // ======================================
30      // lw_bus;
31      // h2f_lw_axi_master -> control port
32      // read_master       -> f2h_axi_slave, puedo leer cualquier
33      //                      periferico que este en este bus,
34      // write_master      -> onchip_memory2_0.s1 = 0x00020000
35      #define H2F_LW_AXI_MASTER_BASE        0xff200000
36      #define HW_REGS_SPAN        0x00005000
37      // the h2f light weight bus base
38      void *h2p_lw_virtual_base;
39      // ======================================
40      // HPS onchip memory base/span
41      // 2^16 bytes at the top of memory
42      #define HPS_ONCHIP_BASE     0xffff0000
43      #define HPS_ONCHIP_SPAN     0x00010000
44      // HPS onchip memory (HPS side!)
45      volatile unsigned int * hps_onchip_ptr = NULL ;
46      void *hps_onchip_virtual_base;
```

Figura 39. QtCreator. Definición de todas las variables y funciones.

Puede observarse que cada uno de los componentes poseen una dirección base. Dichas direcciones son utilizadas para acceder, controlar y enviar datos desde y hacia el SoC.

Para mapear las direcciones físicas a direcciones virtuales, lo primero que se realiza es una llamada abierta al sistema para abrir el controlador o *driver* del dispositivo de memoria *"/dev/mem"* seguido de la llamada al sistema *mmap*, la cual es utilizada para asignar la dirección física del HPS a una dirección virtual representado por el puntero *h2f_axi_master_virtual_base*.

```
56      // /dev/mem file id
57      int fd;
```

Figura 40. QtCreator. ID del archivo /dev/mem.

```
71      // === get FPGA addresses ==================
72      // Open /dev/mem
73      if( ( fd = open( "/dev/mem", ( O_RDWR | O_SYNC ) ) ) == -1 )    {
74          printf( "ERROR: could not open \"/dev/mem\"...\n" );
75          return( 1 );
76      }
77
78      //=============================================
79      // get virtual addr that maps to physical
80      // for light weight bus
81      h2p_lw_virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, H2F_LW_AXI_MASTER_BASE );
82      if( h2p_lw_virtual_base == MAP_FAILED ) {
83          printf( "ERROR: mmap1() failed...\n" );
84          close( fd );
85          return(1);
86      }
87
88      // PIO_0
89      volatile unsigned int* pio_0_base = (unsigned int *)(h2p_lw_virtual_base + PIO_0_BASE);
90
91      //=============================================
92
93      //  RAM FPGA parameter addr
94      h2f_axi_master_virtual_base = mmap( NULL, ONCHIP_MEMORY2_0_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, H2F_AXI_MASTER_BASE );
95      sram_ptr = (unsigned int *)(h2f_axi_master_virtual_base + ONCHIP_MEMORY2_0_BASE);
96
97      // =============================================
98
99      // HPS onchip ram
100     hps_onchip_virtual_base = mmap( NULL, HPS_ONCHIP_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, HPS_ONCHIP_BASE);
101
102     if( hps_onchip_virtual_base == MAP_FAILED ) {
103         printf( "ERROR: mmap3() failed...\n" );
104         close( fd );
105         return(1);
106     }
107     // Get the address that maps to the HPS ram
108     hps_onchip_ptr =(unsigned int *)(hps_onchip_virtual_base);
```

Figura 41. QtCreator. Mapeo de direcciones físicas a direcciones virtuales.

La dirección virtual de *AXI_MASTER_BASE* está representada por *h2f_axi_master_virtual_base*, que es un puntero con el cual se podrá acceder directamente a los registros en el controlador.

Esto se realiza con el fin de poder acceder a los periféricos que forman parte de la estructura de la FPGA, lo cual se logra mapeando las direcciones de la FPGA a una memoria virtual para que las mismas puedan ser accedidas desde el lado del HPS.

Para la transmisión de datos empaquetados en UDP a través de Ethernet, se realiza un *socket UDP*[7]. Para ello, en primera instancia, se deben agregar las librerías *#include <sys/socket.h>*, *#include <unistd.h>* y *#include <arpa/inet.h>*.

```
13      #include <sys/socket.h>
14      #include <unistd.h>
15      #include <arpa/inet.h>
```

Figura 42. QtCreator. Librerías dedicadas al socket UDP.

Posteriormente, se crea el *socket* mediante la función *socket()*[8]. Para ello, debe definirse el descriptor de archivo de socket que, en este caso, es *sfd* (*socket file descriptor*).

```
110     int sfd = socket(AF_INET, SOCK_DGRAM, 0);
111     if(sfd < 0){
112         printf("Socket failed\n");
113         exit(1);
114     }
115     printf("socket open\n");
```

Figura 43. QtCreator. Creación del socket.

Luego, mediante la función *setsockopt()*[9], se establecen o setean los parámetros del *socket* creado.

```
117     int optval=1;
118     setsockopt(sfd, SOL_SOCKET, SO_REUSEPORT, &optval, sizeof(optval));
```

Figura 44. QtCreator. Seteo de parámetros del socket creado.

Además, se debe definir un puerto, que en nuestro caso será el *port=7000*, y una estructura *sockaddr_in*, la cual será empleada para inicializar el *socket* creado.

Además, se debe inicializar la dirección del *socket* fuente.

```
120     int port=7000;
121     struct sockaddr_in address, dst_address;
122     address.sin_family = AF_INET;
123     address.sin_addr.s_addr = htonl(INADDR_ANY);
124     address.sin_port = htons(port);
```

Figura 45. QtCreator. Configuración para recibir.

Luego, por medio de la función *bind()*[10], se asocia el *socket* creado a una dirección particular o a un puerto específico.

```
126     int r = 0;
127     r = bind(sfd, (struct sockaddr*) &address, sizeof(address));
128     if(r < 0){
129         printf("bind failed");
130     }
131     printf("bind success\n");
```

Figura 46. QtCreator. Asociación del socket a una dirección o puerto específico.

Finalmente, se debe inicializar el *socket* de destino.

```
133     dst_address.sin_family = AF_INET;
134     inet_pton(AF_INET, "192.168.7.20", &dst_address.sin_addr);
135     dst_address.sin_port = htons(port);
```

Figura 47. QtCreator. Configuración para enviar o transmitir.

En este punto, se generó una trama en protocolo ASTERIX CAT240 mediante un software diseñado por el Ing. Diego Martinez (perteneciente al SIAG). La trama generada es almacenada en un archivo *.bin* el cual, mediante el software *Okteta*, es convertido a *.hex*. Este archivo *.hex* es el que se carga en la memoria RAM dual port, memoria compartida entre la FPGA y el HPS, desde la herramienta gráfica *Qsys* para simular así que la misma es escrita o cargada con datos empaquetados en protocolo ASTERIX CAT240 desde el lado de la FPGA.

Un detalle a tener en cuenta es que la trama empaquetada en protocolo ASTERIX CAT240 está compuesta de datos fijos. En otras palabras, tenemos “basura” empaquetada en protocolo ASTERIX CAT240. No hay que perder de vista que el objetivo principal del software diseñado es leer la memoria RAM dual port, copiar su contenido y empaquetarlo en protocolo UDP para su posterior transmisión por Ethernet a través de un socket UDP por lo que el contenido de la trama ASTERIX CAT240, a priori, no es de interés. De todas

formas, el software diseñado está pensado para trabajar con tramas empaquetadas en protocolo ASTERIX CAT240 compuesta de datos variables, que sería el caso real.

Una vez generada la trama ASTERIX CAT240, se utilizan las herramientas de manejo de archivos[11] para abrir el archivo y copiar el contenido del mismo, que se encuentra cargado en la memoria RAM dual port, al puntero *sram_ptr* con la finalidad de acceder a los datos desde el lado del HPS.

```
137     FILE *asterix_file = fopen("asterix(4096).bin", "r");
138     if(!asterix_file){
139         printf("Error al abrir asterix.bin,\n");
140         exit(1);
141     }
142     size_t elem = fread((void *)sram_ptr, 1, ONCHIP_MEMORY2_0_SPAN, asterix_file);
143     if(elem == 0){
144         printf("Error al leer el archivo.\n");
145         exit(1);
146     }
147     if(ferror(asterix_file)){
148         printf("error en asterix_file.\n");
149         exit(1);
150     }
151     if(feof(asterix_file)){
152         printf("EOF en asterix_file.\nSe leyeron %d bytes\n", elem);
153     }
```

Figura 48. QtCreator. Manejo de archivos. Copiado de la trama ASTERIX CAT240 para acceder a la misma desde el HPS.

Posteriormente, fueron creadas dos *uniones*[12]: *w32* y *w16*. Una vez creadas, se declaran tres variables del tipo *union*: *msg_index*, *start_az* y *end_az*. Dichas variables son empleadas para manejar la parte de interés de la trama ASTERIX CAT240[13].

```
155     union w32{
156         uint32_t data __attribute__((aligned(4)));
157         char c[4];
158     };
159     union w16{
160         uint16_t data __attribute__((aligned(4)));
161         char c[2];
162     };
163
164     union w32 msg_index;
165     union w16 start_az, end_az;
166
167     msg_index.data = 0;
168     start_az.data = 0;
169     end_az.data = 0;
```

Figura 49. QtCreator. Creación de uniones. Declaración de variables pertenecientes a la trama ASTERIX CAT240.

Tal como puede observarse en la figura *ut supra*, al declarar las uniones *w32* y *w16*, se le especificó a la variable *data* un atributo especial del tipo *aligned*[14] con la finalidad de optimizar el código. Al utilizar el atributo especial *aligned*, se busca que el compilador realice menos operaciones lo que mejora considerablemente la eficiencia del tiempo de ejecución.

Luego, fueron creadas las variables *data*, que es un arreglo de datos utilizado para cargar o copiar el contenido al que apunta el puntero *sram_ptr*, que es nada más ni nada menos que el archivo que contiene la trama ASTERIX CAT240 generada cargado en la memoria SRAM dual port, y la variable *dr*, que es la variable utilizada para calcular el data rate o la tasa de transferencia de datos.

```
172     int data[16384] __attribute__((aligned(4)));
173     int dr __attribute__((aligned(4)));
```

Figura 50. QtCreator. Declaración de variables.

Al igual que para el caso de las uniones *w32* y *w16*, a las variables *data* y *dr* también se le especificaron un atributo especial del tipo *aligned*.

Posteriormente, se ejecuta un ciclo *while* en el cual se realiza, a grosso modo, el copiado del contenido al que apunta *sram_ptr* al arreglo *data*, el cargado de los datos específicos pertenecientes a la trama ASTERIX CAT240, los cuales son *msg_index*, *start_az* y *end_az*, para posteriormente poder graficar, el envío o transmisión del arreglo *data* a través del socket UDP y el cálculo de la tasa de transferencia.

```
175     while(1){
176         if(start_az.data == 0){
177             gettimeofday(&t1, NULL);
178         }
179         if(start_az.data == 65535){
180             gettimeofday(&t2, NULL);
181             elapsedTime = (t2.tv_sec - t1.tv_sec) * 1000000.0;      // sec to us
182             elapsedTime += (t2.tv_usec - t1.tv_usec) ;   // us to
183             dr = (start_az.data*elem / (uint32_t)elapsedTime) * 8;
184             printf("sendto() scan T=%.2f uSec rate=%d MBits/Sec\n\r", elapsedTime, dr);
185         }
186
187         memcpy((void*)data, (const void*)sram_ptr, elem);
188
189         *((uint8_t *)data + 11) = msg_index.c[0];
190         *((uint8_t *)data + 10) = msg_index.c[1];
191         *((uint8_t *)data + 9) = msg_index.c[2];
192         *((uint8_t *)data + 8) = msg_index.c[3];
193         *((uint8_t *)data + 13) = start_az.c[0];
194         *((uint8_t *)data + 12) = start_az.c[1];
195         *((uint8_t *)data + 15) = end_az.c[0];
196         *((uint8_t *)data + 14) = end_az.c[1];
197
198         sendto(sfd, (void *)data, elem, 0, (struct sockaddr *)&dst_address, sizeof(dst_address));
199
200         start_az.data += 1;
201         end_az.data += 1;
202         msg_index.data += 1;
203
204     } // end while(1)
```

Figura 51. QtCreator. Copiado de la trama ASTERIX CAT240 y posterior transmisión de la misma por Ethernet a través del socket UDP.

Desglosando lo comentado previamente, al comienzo del ciclo *while* se tienen dos sentencias *if*, las cuales son utilizadas para marcar el comienzo y final, respectivamente, del barrido o vuelta completa al radar al momento de graficar. Además, una vez completado el barrido, haciendo uso de las funciones *gettimeofday()*[15] y *elapsedTime*, la cual retorna el tiempo transcurrido entre dos valores de tiempo, calculados previamente con la función *gettimeofday()*, se calcula el data rate o tasa de transferencia de datos.

```
175     while(1){
176         if(start_az.data == 0){
177             gettimeofday(&t1, NULL);
178         }
179         if(start_az.data == 65535){
180             gettimeofday(&t2, NULL);
181             elapsedTime = (t2.tv_sec - t1.tv_sec) * 1000000.0;      // sec to us
182             elapsedTime += (t2.tv_usec - t1.tv_usec) ;   // us to
183             dr = (start_az.data*elem / (uint32_t)elapsedTime) * 8;
184             printf("sendto() scan T=%.2f uSec rate=%d MBits/Sec\n\r", elapsedTime, dr);
185         }
```

Figura 52. QtCreator. Cálculo del data rate.

Haciendo uso de la función *memcpy()*[16], se transfiere el contenido al que apunta el puntero *sram_ptr* al arreglo *data*, que es el archivo que contiene la trama ASTERIX CAT240 generada cargado en la memoria SRAM dual port, memoria compartida entre la FPGA y el HPS.

```
187         memcpy((void*)data, (const void*)sram_ptr, elem);
```

Figura 53. QtCreator. Copiado del contenido al que apunta sram_ptr al arreglo data mediante la instrucción memcpy().

Una vez copiado el contenido de *sram_ptr* al arreglo *data*, se realiza el copiado de los parámetros específicos de la trama ASTERIX CAT240 con la finalidad de poder realizar la gráfica.

El cómo o la forma en que se realiza la transferencia de datos es de acuerdo a la arquitectura del microprocesador empleado[17]. Hay que tener en cuenta que la PC es *little-endian* mientras que el protocolo ASTERIX CAT240 es *big-endian.*

```
189         *((uint8_t *)data + 11) = msg_index.c[0];
190         *((uint8_t *)data + 10) = msg_index.c[1];
191         *((uint8_t *)data + 9) = msg_index.c[2];
192         *((uint8_t *)data + 8) = msg_index.c[3];
193         *((uint8_t *)data + 13) = start_az.c[0];
194         *((uint8_t *)data + 12) = start_az.c[1];
195         *((uint8_t *)data + 15) = end_az.c[0];
196         *((uint8_t *)data + 14) = end_az.c[1];
```

Figura 54. QtCreator. Transferencia de parámetros específicos de la trama ASTERIX CAT240.

Finalmente, por medio de la función *sendto()*[18], se envían los paquetes o datos empaquetados en UDP, previamente empaquetados en protocolo ASTERIX CAT240, a través de Ethernet por medio del *socket* creado inicialmente.

```
197         sendto(sfd, (void *)data, elem, 0, (struct sockaddr *)&dst_address, sizeof(dst_address));
```

Figura 55. QtCreator. Envío de paquetes a través del socket.

Además, es importante incrementar los valores de las variables *msg_index*, *start_az* y *end_az* ya que son las utilizadas para graficar y para evaluar las sentencias *if*.

```
200         start_az.data += 1;
201         end_az.data += 1;
202         msg_index.data += 1;
```

Figura 56. QtCreator. Incremento de las variables.

Por último, se cierra el archivo.

```
206     fclose(asterix_file);
```

Figura 57. QtCreator. Manejo de archivos. Cierre del archivo.

2.4. Ensayos (sin la implementación del handshake entre la FPGA y el HPS)

En la presente sección del documento se detallan los ensayos realizados empleando el kit de desarrollo DE1-SoC, utilizando el software desarrollado y explicado en la sección previa del presente informe.

El primer ensayo a realizar consiste en una simple prueba de rendimiento en la cual, haciendo uso del comando *iperf*, se realiza la transmisión y recepción de paquetes UDP con el fin de verificar el correcto funcionamiento del OS Linux Ubuntu embebido dentro del HPS, el cual se encarga de realizar toda la parte de red, ya que se requiere que la velocidad (t*hroughput*) o tasa de transmisión y/o recepción efectiva de datos UDP en tiempo real de la aplicación a diseñar sea del orden de los 1000 Mbps/1 Gbps, o lo más cercano posible a ésta.

El segundo ensayo a realizar consiste en una prueba de medición de tiempos con el fin de conocer los tiempos de lectura y de transmisión de datos fijos, los cuales fueron cargados en una memoria RAM dual port, que es la memoria compartida entre la FPGA y el HPS, y así descubrir el tiempo máximo que el HPS necesita para leer un dato en memoria, copiarlo y transmitirlo por Ethernet a través de un socket UDP.

El tercer y último ensayo a realizar es un símil del segundo con la salvedad de que, en este caso, los datos fijos cargados en la memoria RAM dual port están empaquetados en protocolo ASTERIX CAT240. Además, se realizará una prueba de visualización de los datos transmitidos por Ethernet haciendo uso del software *RadarView*, de Cambridge Pixel. Mediante este ensayo verificaremos de manera fehaciente si la línea de trabajo adoptada cumple o no con las expectativas y requerimientos solicitados.

2.4.1. Transmisión y recepción de paquetes UDP utilizando el comando *iperf*

Haciendo uso del switch administrable L2+ JetStream TL-SG3210, de una PC y del kit de desarrollo DE1-SoC, se llevó a cabo la primera prueba de rendimiento mediante la cual se medirán diferentes parámetros tal como el ancho de banda (*bandwidth*), la velocidad (*throughput*), el jitter, la pérdida de paquetes (*packet loss*) y la latencia (*latency*). Para ello, se conecta la PC al puerto Ethernet 1 y el kit de desarrollo DE1-SoC al puerto Ethernet 2 del switch mediante cables/patchcords directos/derechos (straight-through) RJ45 de la marca Vention (CAT8 FTP PATCH CABLE 4PAIRS AWM PVC 75°C EIA/TIA 568B).

Las pruebas de rendimiento se llevan a cabo utilizando la consola de Linux, donde la PC deberá de fungir como servidor mientras que el kit de desarrollo DE1-SoC de cliente, y haciendo uso del comando específico *iperf*, el cual será empleado para medir los parámetros mencionados anteriormente. La herramienta *iperf* es muy utilizada para el diagnóstico de red.

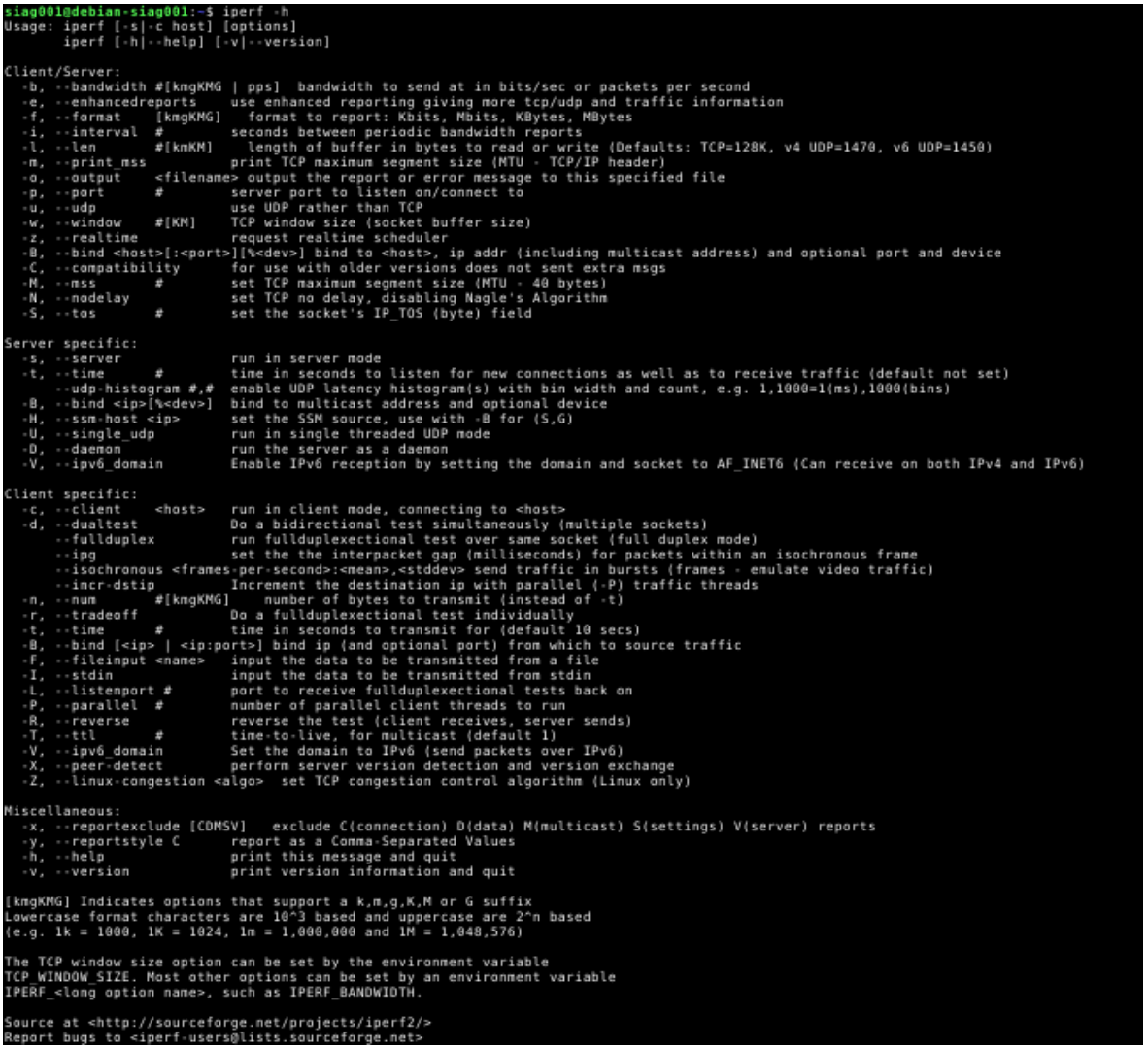

Figura 58. Comando iperf. Sintaxis y uso.

Por lo tanto, los comandos a utilizar son los siguientes:

- Cliente (PC con IP 192.168.7.40)

iperf3.exe -c 192.168.7.20 -u -i 1 -b 1000M -t 60

Donde *-c* hace referencia a que es cliente, *192.168.7.20* es la dirección IP del servidor, *-u* es referido a UDP, *-i 1* hace referencia al intervalo en segundos entre reportes (*1* es que los intervalos entre reportes son cada 1 segundos), *-b 1000M* es el ancho de banda (*-b* es de bandwidth y *1000M* es de 1000 Mbits/s) y *-t 60* es la duración de la prueba a realizar (*-t* es de tiempo y *60* es la duración en segundos).

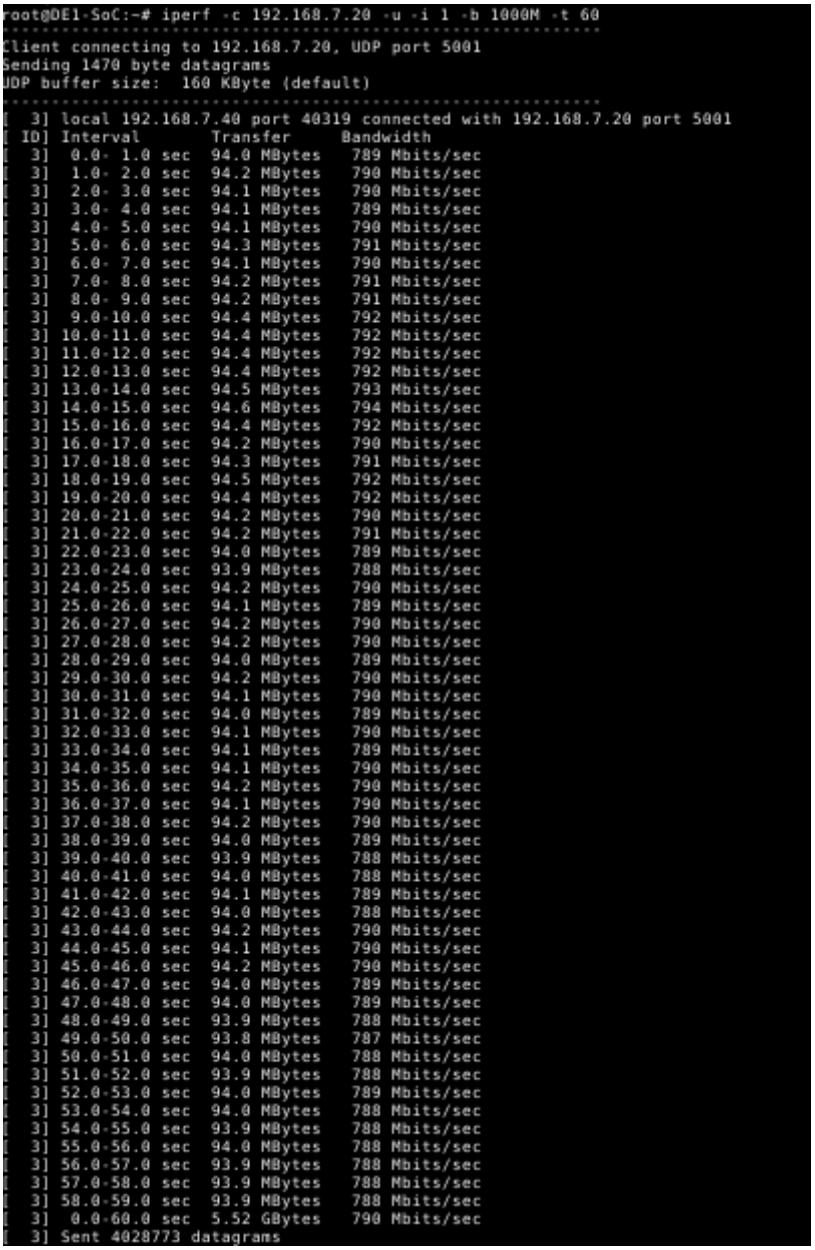

Figura 59. Consola de Linux. Comando iperf. Cliente.

- Servidor (PC con IP 192.168.7.20)

iperf3.exe -s -u -i 1 -b 1000M -t 60

Donde *-s* hace referencia a que es servidor, *-u* es referido a UDP, *-i 1* hace referencia al intervalo en segundos entre reportes (*1* es que los intervalos entre reportes son cada 1 segundos), *-b 1000M* es el ancho de banda (*-b* es de bandwidth y *1000M* es de 1000 Mbits/s) y *-t 60* es la duración de la prueba a realizar (*-t* es de tiempo y *60* es la duración en segundos).

```
siag001@debian-siag001:~$ iperf -s -u -i 1 -b 1000M -t 60
------------------------------------------------------------
Server listening on UDP port 5001 with pid 2501
Read buffer size: 1.44 KByte (Dist bin width= 183 Byte)
UDP buffer size:  208 KByte (default)
------------------------------------------------------------
```

Figura 60. Consola de Linux. Comando iperf. Servidor.

La velocidad (*throughput*) o tasa de transmisión y/o recepción efectiva de datos UDP en tiempo real se calculó de la siguiente manera:

$$\frac{Transfer[GBytes]\cdot 8[bits]}{Segundos} = Velocidad[Gbps]$$

Por lo tanto, tenemos que:

$$\frac{5.52[GBytes]\cdot 8[bits]}{60\ segundos} = 0.736[Gbps] = 736[Mbps]$$

Mediante el presente ensayo se verificó el correcto funcionamiento del OS Linux Ubuntu embebido dentro del HPS, el cual se encarga de realizar toda la parte de red.

2.4.2. Medición de tiempos de lectura y de transmisión con datos fijos

Haciendo uso de una PC y del kit de desarrollo DE1-SoC, se realizará una prueba de medición de tiempos. Dicho ensayo consiste en la escritura y posterior lectura de datos fijos en una memoria RAM dual port, que es la memoria compartida entre la FPGA y el HPS, para medir así los tiempos de lectura y de transmisión con el fin de conocer el tiempo máximo que el HPS necesita para leer un dato en memoria, copiarlo y transmitirlo por Ethernet a través de un socket UDP. Vale destacar que la escritura de datos es realizada desde el lado de la FPGA mientras que la lectura de los mismos es realizada desde el lado del HPS.

La prueba de medición de tiempos se lleva a cabo utilizando la consola de Linux, donde se ejecuta el código *dma01* desde el lado del HPS. El código empleado para realizar el ensayo en cuestión está basado en un código encontrado en Internet[19][20]. En pocas palabras, el código se encarga, en primera instancia, de mapear las direcciones de la FPGA a una memoria virtual para que las mismas puedan ser accedidas desde el lado del HPS. Luego, se escribe la memoria de la FPGA en su totalidad (8192 [bytes]) con datos fijos y posteriormente se la lee, haciendo uso de un puntero denominado *sram_ptr*, desde el HPS. A continuación, haciendo uso de la función *memcpy()*[16], se transfiere el contenido del puntero *sram_ptr* a un arreglo *data* para, finalmente, enviar dichos datos por Ethernet a través del socket UDP empleando la función *sendto()*[18]. Para calcular los tiempos, se emplean las funciones *gettimeofday()*[15] y *elapsedTime*. Esta última retorna el tiempo transcurrido entre dos valores de tiempo, calculados previamente con la función *gettimeofday()*.

Conociendo a grosso modo la función del código *dma01*, se procede a realizar las pruebas comentadas. Para ello, desde la consola de Linux ejecutamos dicho código, tal y como se observa en la imagen a continuación.

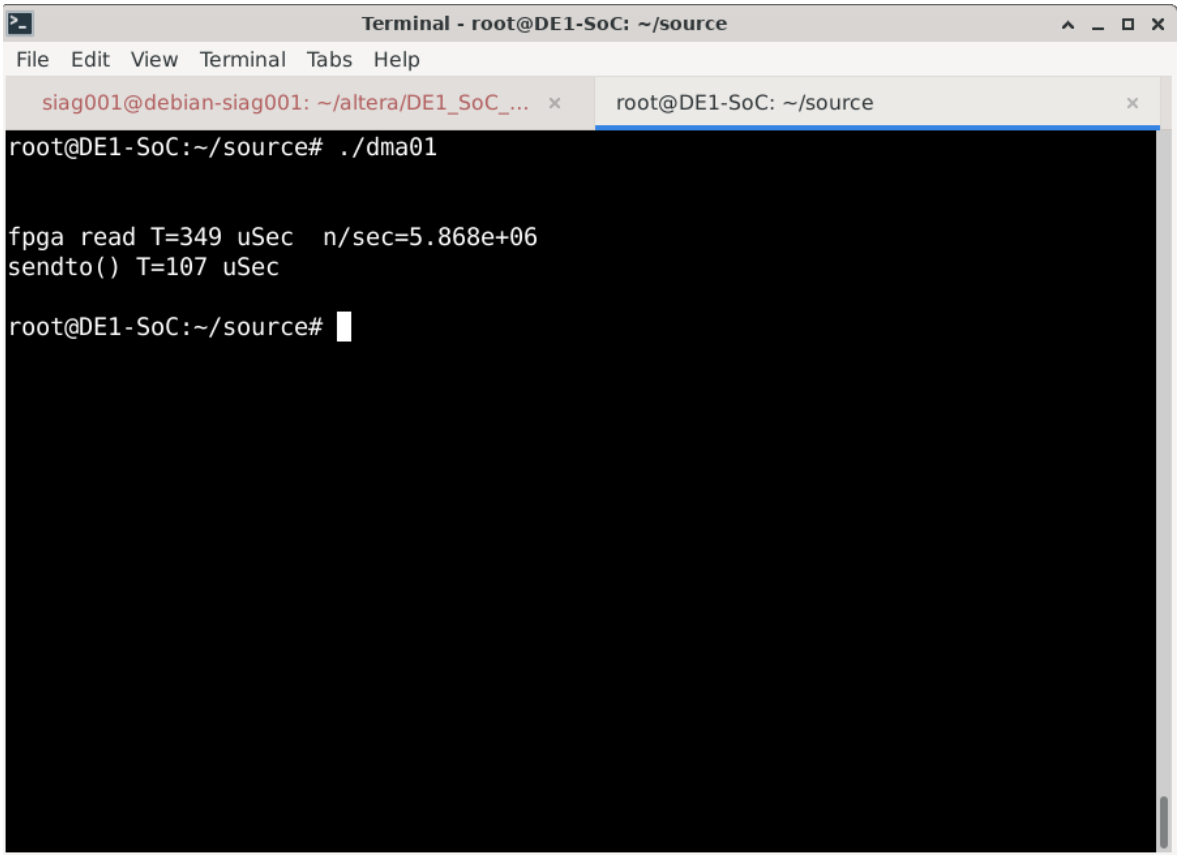

Figura 61. Consola de Linux. Ejecución código dma01. Tiempos de lectura y de transmisión.

La velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real se calculó de la siguiente manera:

$$\frac{Tamaño\ de\ la\ memoria[Bytes]\cdot 8[bits]}{Tiempo\ de\ transmisión[segundos]} = Velocidad[Mbps]$$

Por lo tanto, tenemos que:

$$\frac{8192[Bytes]\cdot 8[bits]}{107\cdot 10^{-6}\,[segundos]} \approx 600[Mbps]$$

Luego, mediante el software *Wireshark*, realizamos la captura del tráfico de red. Con esto, nos aseguramos que el socket UDP realizado funciona correctamente. Vale destacar que los datos son transmitidos desde el HPS, cuya dirección IP es 192.168.7.40, y son recibidos por la PC, cuya dirección IP es 192.168.7.20.

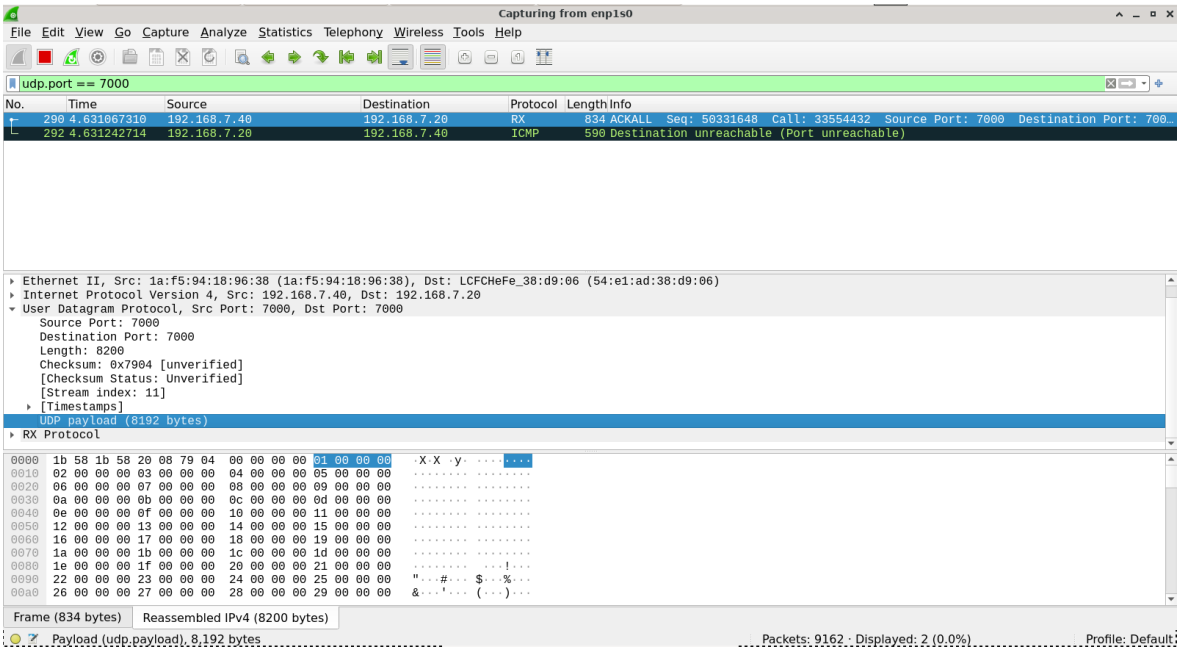

Figura 62. Software Wireshark. Captura de tráfico.

2.4.3. Medición de tiempos de lectura y de transmisión con datos fijos empaquetados en protocolo ASTERIX CAT240

Haciendo uso de una PC y del kit de desarrollo DE1-SoC, se realizará una prueba de medición de tiempos similar a la anterior con la salvedad de que, en este caso, los datos fijos cargados en la memoria RAM dual port están empaquetados en protocolo ASTERIX CAT240. Por lo tanto, la medición de los tiempos de lectura y de transmisión, es decir, el tiempo máximo que el HPS necesita para leer un dato en memoria, copiarlo y transmitirlo por Ethernet a través de un socket UDP, nos darán una velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real.

La prueba de medición de tiempos se lleva a cabo utilizando la consola de Linux, donde se ejecuta el código *dma01* desde el lado del HPS. El código empleado para realizar el ensayo en cuestión está basado en un código encontrado en Internet[19][20], el cual fue comentado de forma generalizada en el ensayo anterior[1]. De todas maneras, el software realizado tuvo que ser modificado debido a que, para este caso, se debe de generar primero una trama en protocolo ASTERIX CAT240, la cual posteriormente es cargada en memoria, leída a través del HPS y enviada por Ethernet a través del socket UDP. La trama ASTERIX es generada por medio de un software, el cual fue diseñado por el Ing. Diego Martinez (perteneciente al SIAG). La misma es almacenada en un archivo *.bin* el cual, mediante el software *Okteta*, es convertido a *.hex*. Este archivo *.hex* es el que se carga en la memoria RAM dual port para simular así que la misma es escrita con datos empaquetados en protocolo ASTERIX CAT240 desde el lado de la FPGA. Posteriormente, se lee dicha memoria, se empaquetan los datos cargados previamente en la memoria en protocolo UDP para, finalmente, enviar los mismos por Ethernet a través de un socket UDP.

En el Anexo B podrá encontrarse una breve reseña del protocolo ASTERIX CAT240, además de una explicación sobre cómo se arma un paquete o trama empaquetado en dicho protocolo y la correspondencia entre el mismo y su representación gráfica, lo cual resultará

[1] ver ensayo 2.4.2.

útil conocer al momento de observar las capturas de tráfico de red realizadas en los ensayos detallados a continuación.

Como punto de partida, se debe asegurar una velocidad de transmisión efectiva de datos que sea igual a la velocidad a la que se cargan los datos en memoria, los cuales dijimos son de tamaño variable. Es decir, como mínimo se debe ser capaz de transmitir a la misma velocidad en que almacenan los datos en memoria para así, mientras se transmite la primera trama, poder leer la segunda y prepararla para la transmisión. Esta sería la peor condición que debemos cumplir o asegurar.

Sabemos que la velocidad de muestreo del ADC es de 16 [MS/s] y cada muestra es de 2 [bytes]. Por lo tanto, la velocidad a la que se cargan los datos en memoria se calcula como:

$$16[MS/s] \cdot 2[Bytes] = 32[MB/s]$$

A lo largo de los ensayos descritos en la presente sección, se observará que se optimiza bajo ciertas condiciones. No hay que perder de vista que estos son los primeros ensayos que se realizan, los cuales brindarán un nuevo punto de partida para ensayos o pruebas futuras. A la fecha, nuestra peor condición a cumplir es el tiempo que se tarda en almacenar o cargar los datos en memoria, es decir, debemos ser capaces de superar una tasa de transferencia de datos de 32[MB/s] la cual está dada por la velocidad de muestreo del ADC.

Durante los ensayos, además, se notará que el tamaño del paquete o trama empaquetada en protocolo ASTERIX CAT240 es variable, es decir, comenzaremos utilizando tramas de 1056 [bytes], seguiremos con tramas de 2080 [bytes] y finalizaremos con tramas de 4128 [bytes].

2.4.3.1. Tamaño de trama de 1056 [bytes]

El primer ensayo realizado es para un tamaño de paquete o trama empaquetada en protocolo ASTERIX CAT240 de 1056 [bytes], donde los datos o información útil, es decir, sin contar encabezados y demás, son 1024 [bytes].

Por lo tanto, tenemos que:

```
root@DE1-SoC:~/source# ./dma01
socket open
bind success
EOF en asterix_file.
Se leyeron 1056 bytes
sendto() scan T=1379786.00 uSec rate=400 MBits/Sec
sendto() scan T=1380547.00 uSec rate=400 MBits/Sec
sendto() scan T=1379938.00 uSec rate=400 MBits/Sec
sendto() scan T=1381119.00 uSec rate=400 MBits/Sec
sendto() scan T=1382061.00 uSec rate=400 MBits/Sec
sendto() scan T=1381881.00 uSec rate=400 MBits/Sec
sendto() scan T=1380899.00 uSec rate=400 MBits/Sec
sendto() scan T=1382052.00 uSec rate=400 MBits/Sec
sendto() scan T=1381185.00 uSec rate=400 MBits/Sec
sendto() scan T=1382301.00 uSec rate=400 MBits/Sec
sendto() scan T=1380898.00 uSec rate=400 MBits/Sec
sendto() scan T=1381089.00 uSec rate=400 MBits/Sec
sendto() scan T=1381833.00 uSec rate=400 MBits/Sec
sendto() scan T=1380115.00 uSec rate=400 MBits/Sec
```

Figura 63. Consola de Linux. Ejecución código dma01. Tiempos de transmisión.

Podemos observar que un barrido o una vuelta completa de radar tarda en completarse 1.38 segundos aproximadamente.

Por lo tanto, la tasa de transferencia de datos en tiempo real se calculó de la siguiente manera:

$$\frac{Cantidad\ de\ paquetes \cdot Tamaño\ de\ trama[Bytes]}{Tiempo\ de\ transmisión[segundos]} = Tasa\ de\ transferencia[MB/s]$$

Considerando datos o información útil, tenemos que:

$$\frac{65536\ paquetes \cdot 1024[Bytes]}{1.38[segundos]} \approx 48.63[MB/s]$$

La velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real se calculó de la siguiente manera:

$$\frac{65536\ paquetes \cdot 1024[Bytes] \cdot 8[bits]}{1.38[segundos]} \approx 389[Mbps]$$

Luego, mediante el software *Wireshark*, se realiza la captura del tráfico de red. Con esto, nos aseguramos que el socket UDP realizado funciona correctamente.

Vale destacar que los datos son transmitidos desde el HPS, cuya dirección IP es 192.168.7.40, y son recibidos por la PC, cuya dirección IP es 192.168.7.20.

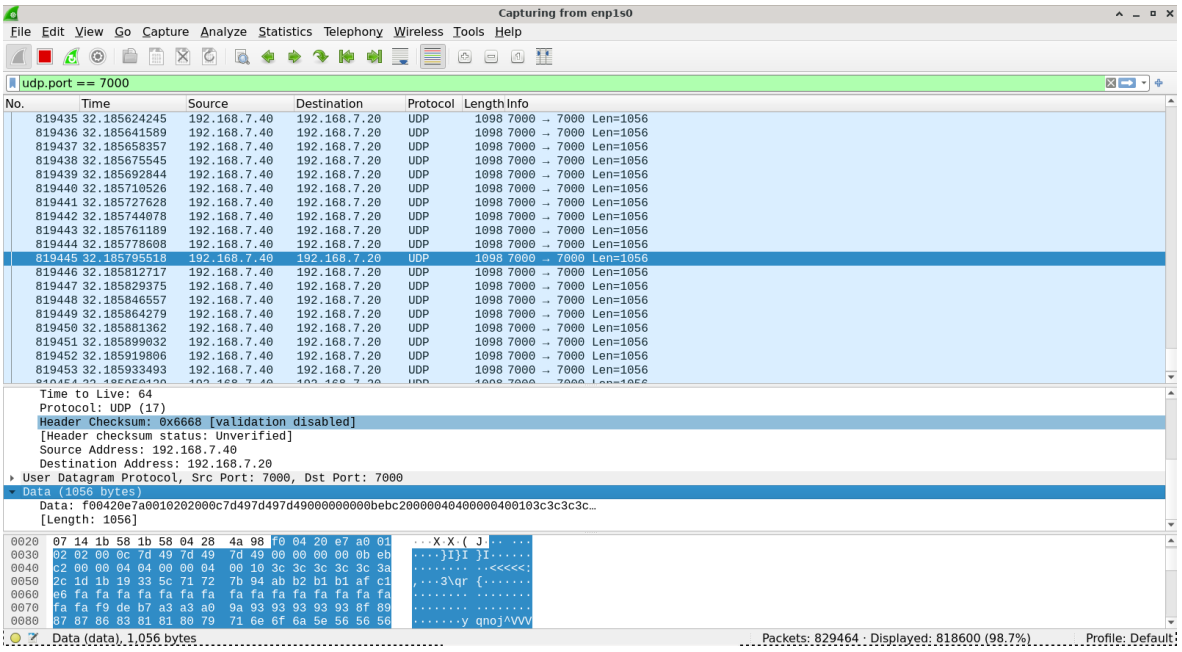

Figura 64. Software Wireshark. Captura de tráfico.

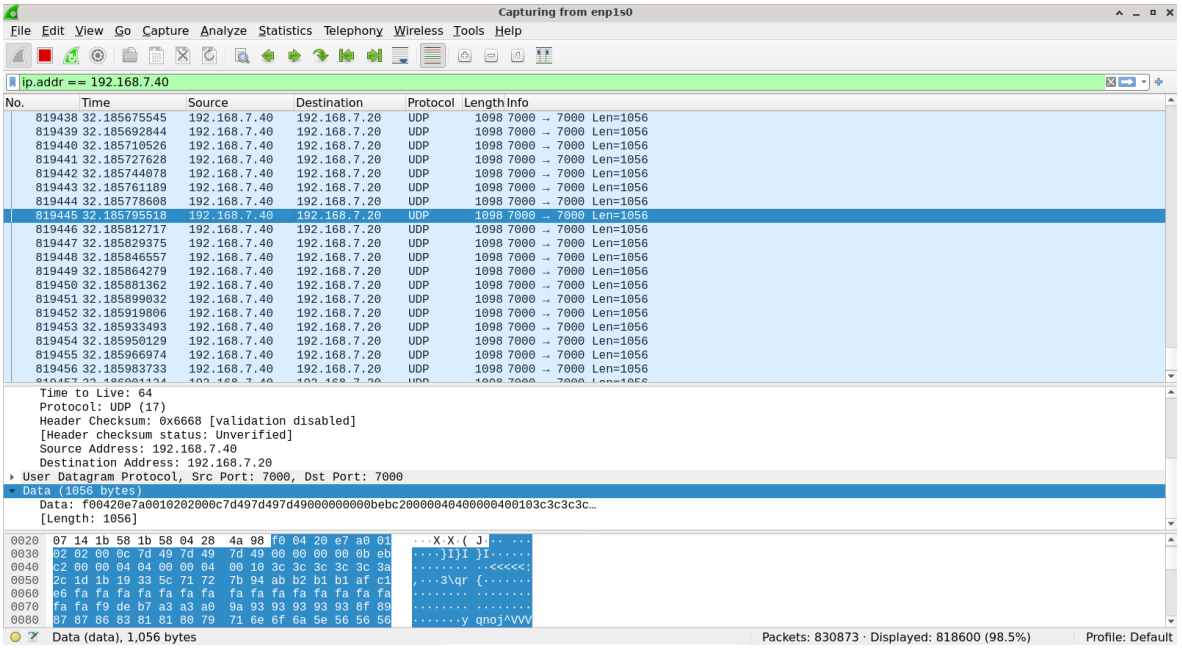

Figura 65. Software Wireshark. Captura de tráfico.

2.4.3.2. Tamaño de trama de 2080 [bytes]

El segundo ensayo realizado es para un tamaño de paquete o trama empaquetada en protocolo ASTERIX CAT240 de 2080 [bytes], donde los datos o información útil, es decir, sin contar encabezados y demás, son 2048 [bytes].

Por lo tanto, tenemos que:

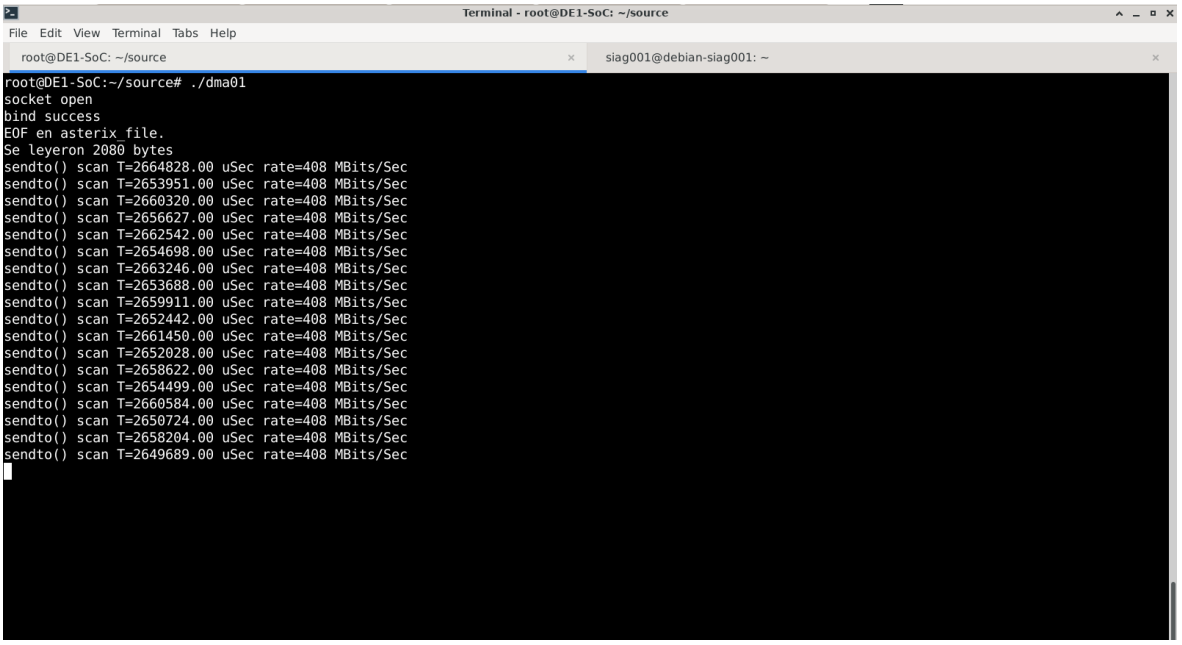

Figura 66. Consola de Linux. Ejecución código dma01. Tiempos de transmisión.

Podemos observar que un barrido o una vuelta completa de radar tarda en completarse 2.65 segundos aproximadamente.

Considerando datos o información útil, tenemos que:

$$\frac{65536\ paquetes \cdot 2048[Bytes]}{2.65[segundos]} \approx 50.648[MB/s]$$

La velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real se calculó de la siguiente manera:

$$\frac{65536\ paquetes \cdot 2048[Bytes] \cdot 8[bits]}{2.65[segundos]} \approx 405.2[Mbps]$$

Haciendo uso del software *RadarView*, de Cambridge Pixel, lo que hacemos es graficar los datos enviados por Ethernet a través del socket UDP.

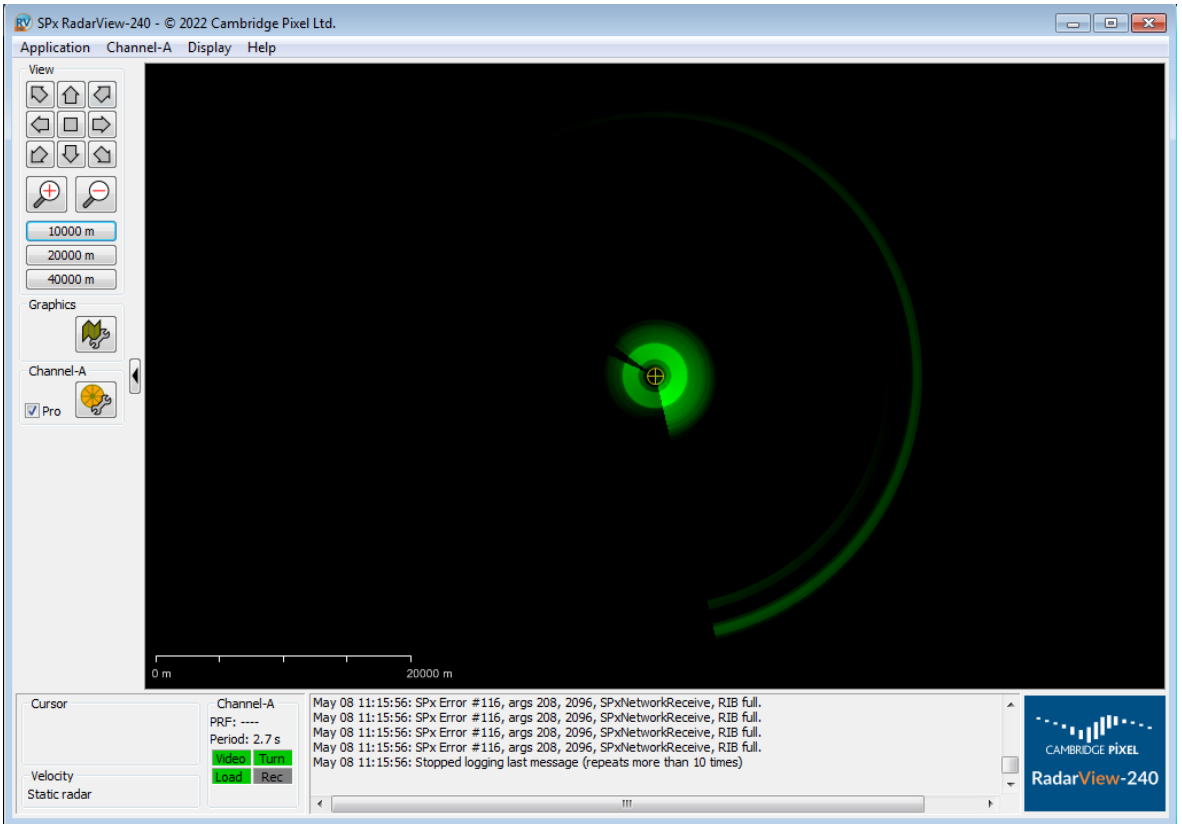

Figura 67. Software RadarView. Visualización de paquetes transmitidos en protocolo ASTERIX CAT240.

Luego, mediante el software *Wireshark*, se realiza la captura del tráfico de red.

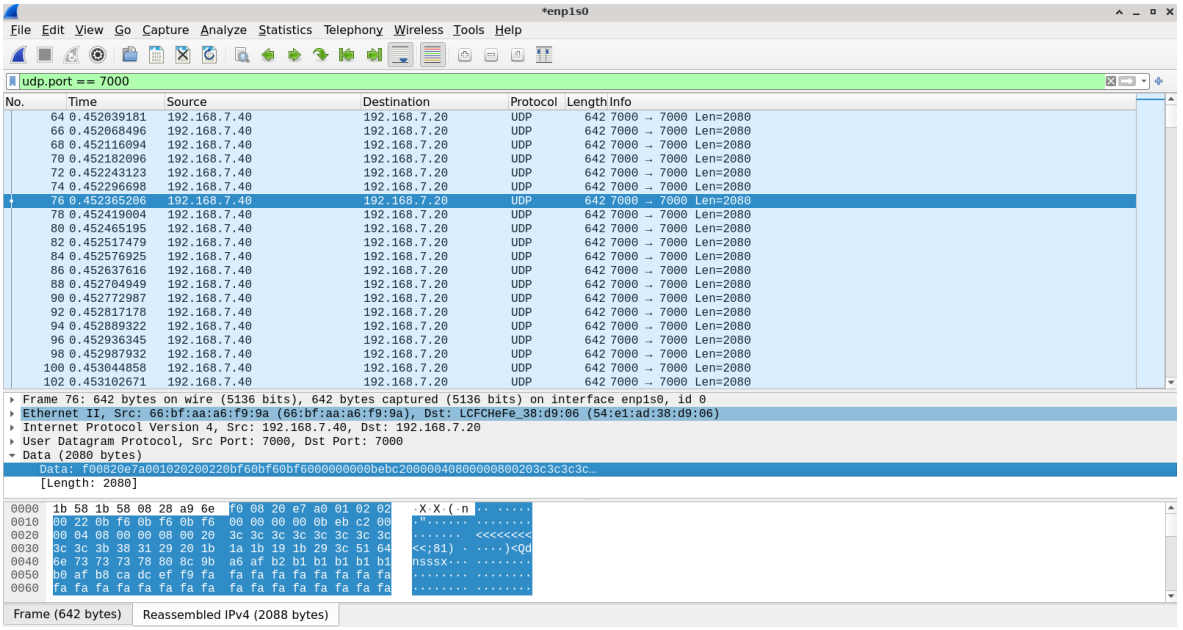

Figura 68. Software Wireshark. Captura de tráfico.

Puede observarse que la trama está definitivamente empaquetada en protocolo ASTERIX CAT240 debido a que se identifica el primer byte de la misma con *f0*.

Además, pudo observarse que los datos empaquetados en protocolo ASTERIX CAT240, al transmitirlos por UDP, son recortados y luego reensamblados para su posterior representación utilizando el software *RadarView*. Esta división de paquetes se debe a que se supera el *MTU* que, para el caso de Ethernet, es de 1500 [bytes].

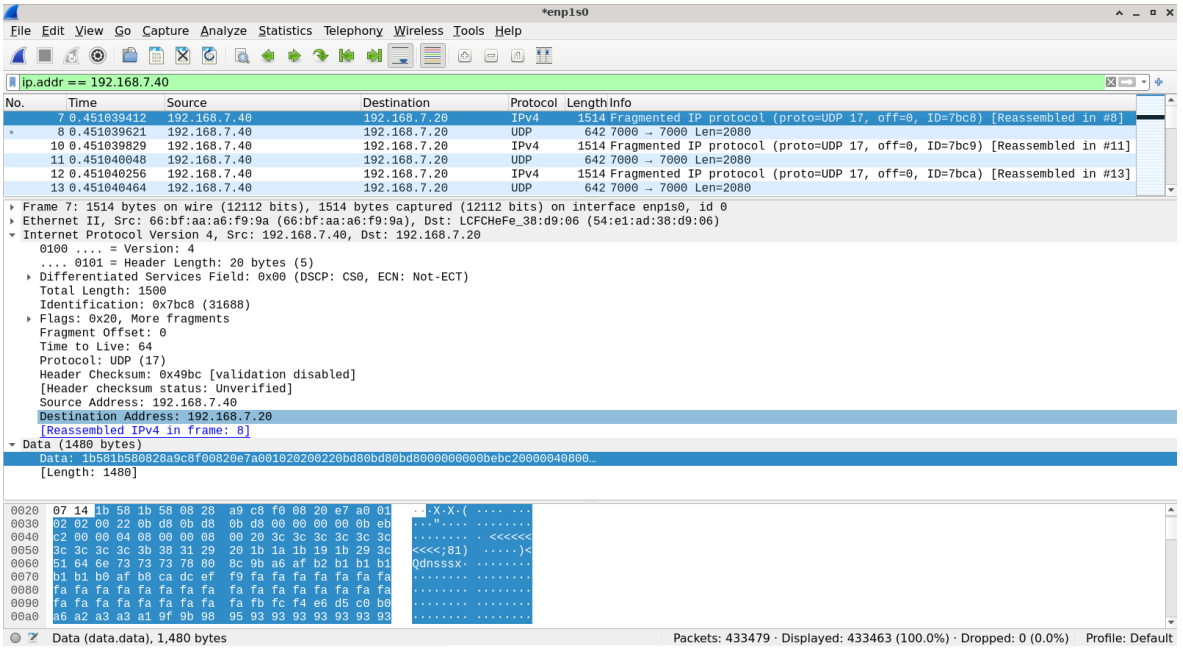

Figura 69. Software Wireshark. Captura de tráfico. Recortado y reensamblado de paquetes.

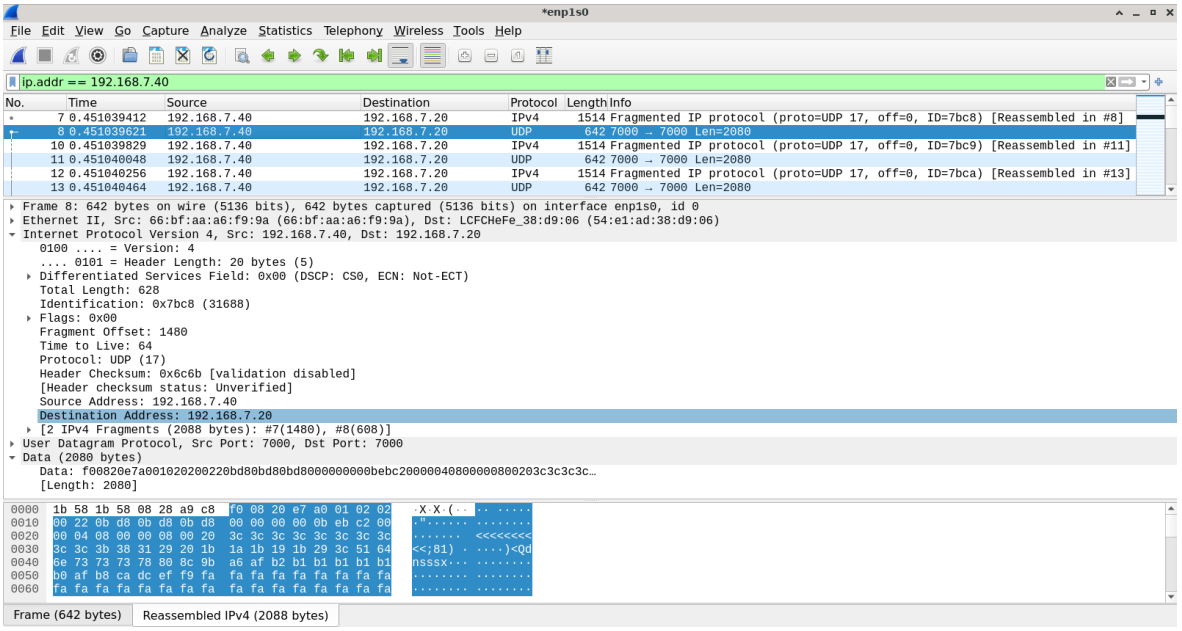

Figura 70. Software Wireshark. Captura de tráfico. Recortado y reensamblado de paquetes.

2.4.3.3. Tamaño de trama de 4128 [bytes]

Una forma de optimizar y, por consiguiente, aumentar la velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real, es aumentando el tamaño de trama.
Pudo observarse que con un tamaño de trama de 1024 [bytes], la tasa de transferencia obtenida fue de aproximadamente 48.63[MB/s][2] mientras que con un tamaño de trama 2048 [bytes], la misma fue de 50.65[MB/s][3]. Esto se debe al tiempo que se tarda, haciendo uso de la instrucción *memcpy*, en copiar en un buffer lo que se encuentra almacenado en memoria y transmitirlo. Este es un tiempo fijo de procesamiento que se pierde. Por lo tanto, a medida que se aumenta el tamaño de trama, dicho tiempo se hace cada vez menos considerable debido a que se está manejando un volumen mayor de datos pero si, en cambio, se trabaja con tamaño de tramas pequeños, dicho tiempo comienza a tomar importancia lo que impacta directamente en la velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real.

Entonces, si ahora el tamaño del paquete o trama empaquetada en protocolo ASTERIX CAT240 es de 4128 [bytes], donde los datos o información útil, es decir, sin contar encabezados y demás, son 4096 [bytes], tenemos que:

[2] ver ensayo 2.4.3.1.
[3] ver ensayo 2.4.3.2.

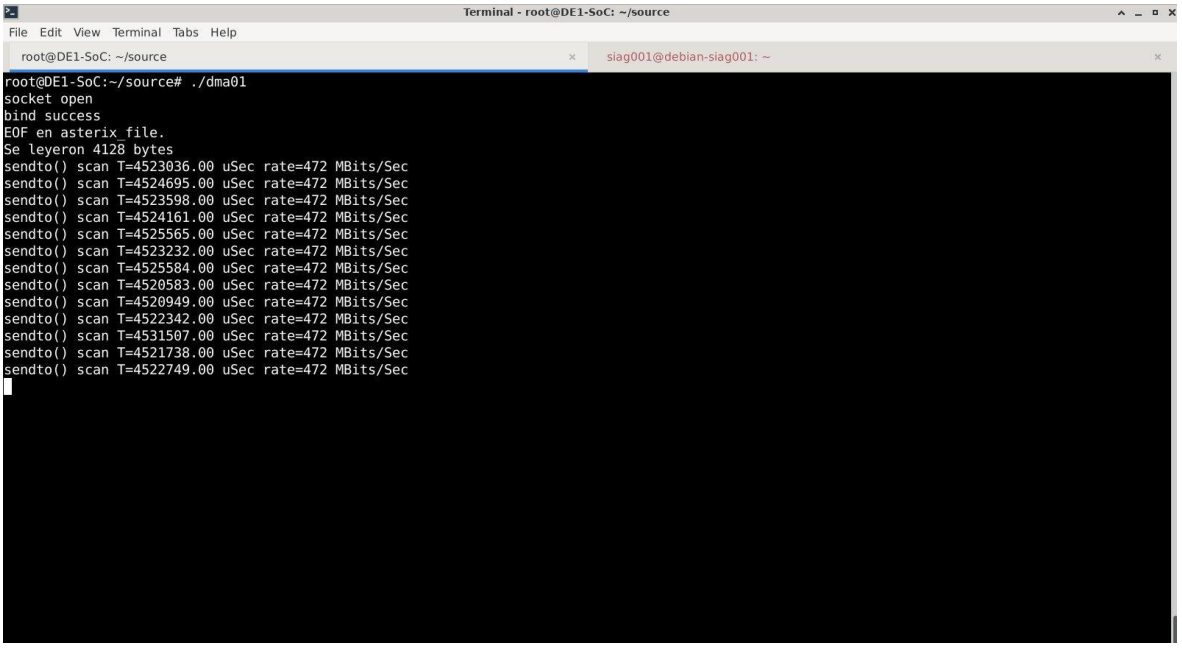

Figura 71. Consola de Linux. Ejecución código dma01. Tiempos de transmisión.

Podemos observar que, para este caso, un barrido o una vuelta completa de radar tarda en completarse 4.55 segundos aproximadamente.

Considerando datos o información útil, tenemos que:

$$\frac{65536\ paquetes \cdot 4096[Bytes]}{4.55[segundos]} \approx 59[MB/s]$$

La velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real se calculó de la siguiente manera:

$$\frac{65536\ paquetes \cdot 4096[Bytes] \cdot 8[bits]}{4.55[segundos]} \approx 472[Mbps]$$

Luego, mediante el software *Wireshark*, se realiza la captura del tráfico de red.

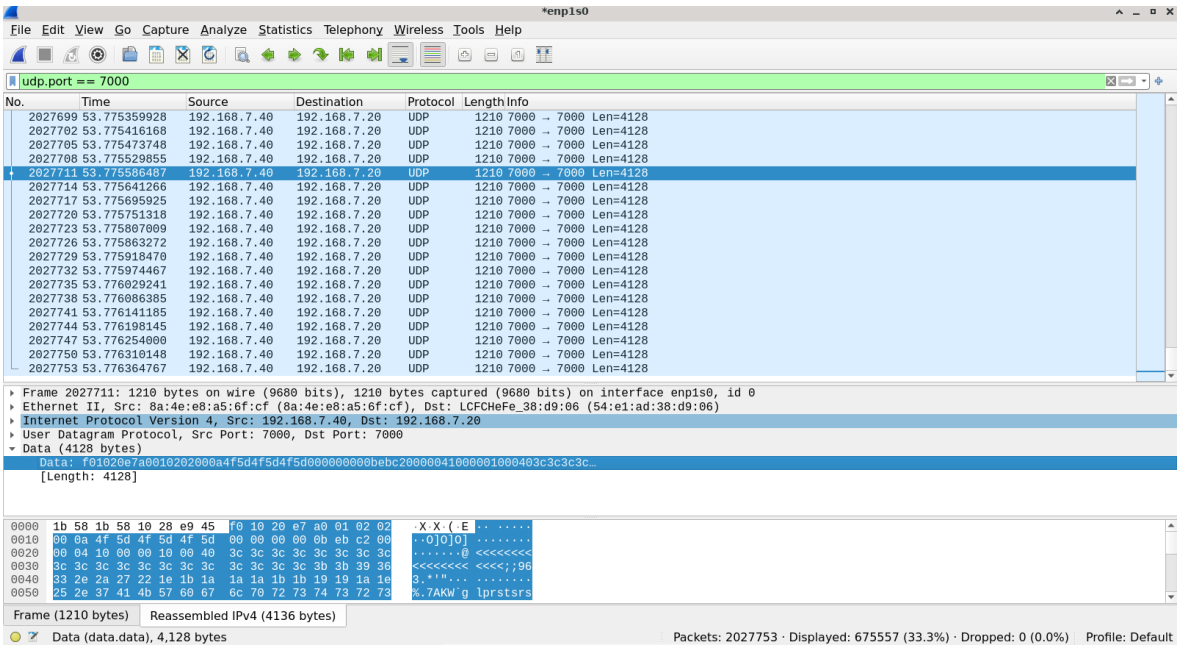

Figura 72. Software Wireshark. Captura de tráfico.

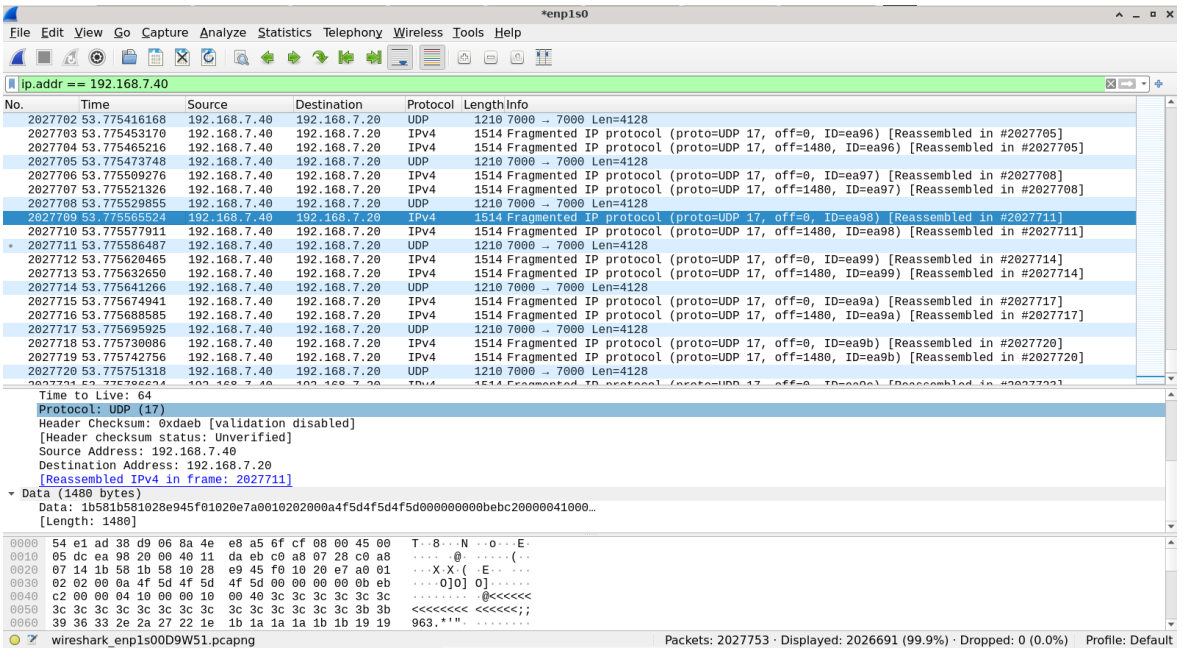

Figura 73. Software Wireshark. Captura de tráfico. Recortado y reensamblado de paquetes.

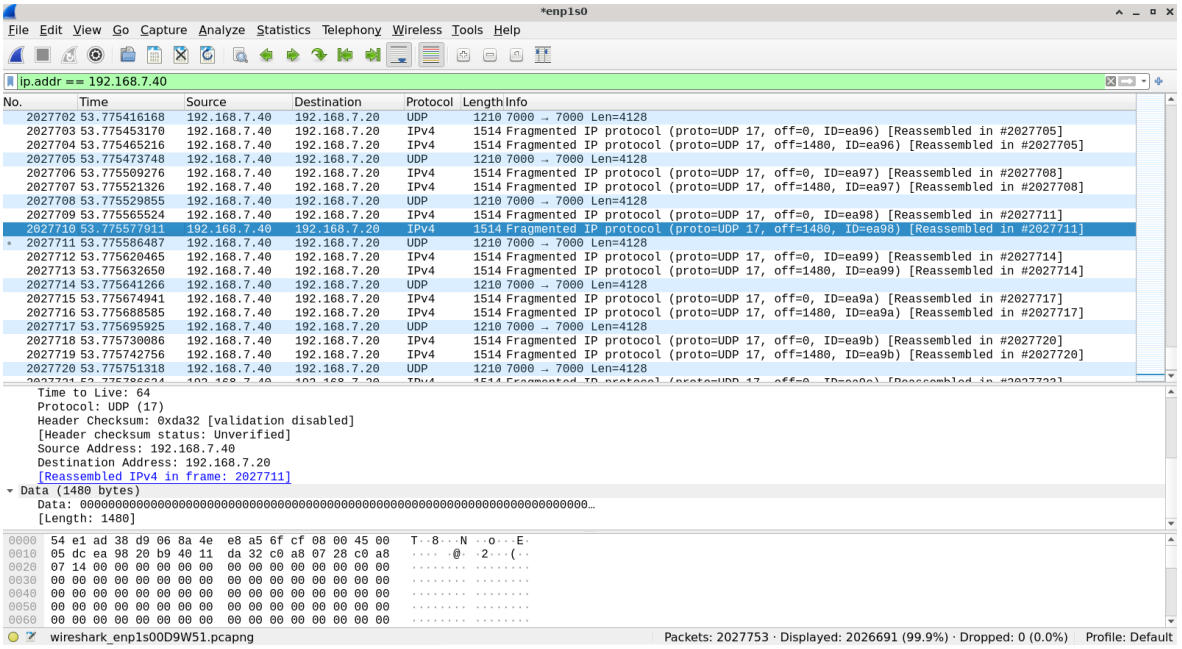

Figura 74. Software Wireshark. Captura de tráfico. Recortado y reensamblado de paquetes.

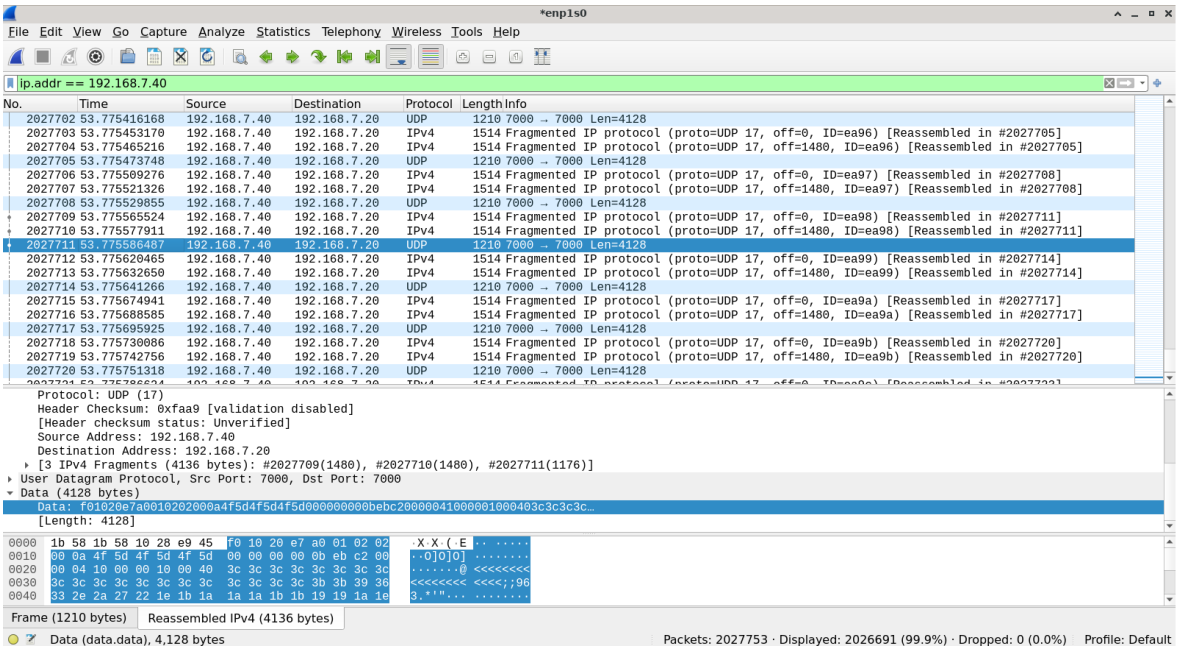

Figura 75. Software Wireshark. Captura de tráfico. Recortado y reensamblado de paquetes.

Al igual que en el ensayo anterior[4], pudo observarse que los paquetes empaquetados en protocolo ASTERIX CAT240, al transmitirlos por UDP, son recortados y luego reensamblados para su posterior representación utilizando el software *RadarView*, debido a que se supera el *MTU* que, para el caso de Ethernet, es de 1500 [bytes]. La única diferencia con el ensayo anterior es que la fragmentación es mayor debido a que el tamaño del paquete o trama es mayor.

[4] ver ensayo 2.4.3.2.

2.4.3.4. Modo debug vs. Modo release

Con la finalidad de optimizar el código, se llevó a cabo otra prueba la cual consiste en pasar el código de *debug* a *release*.

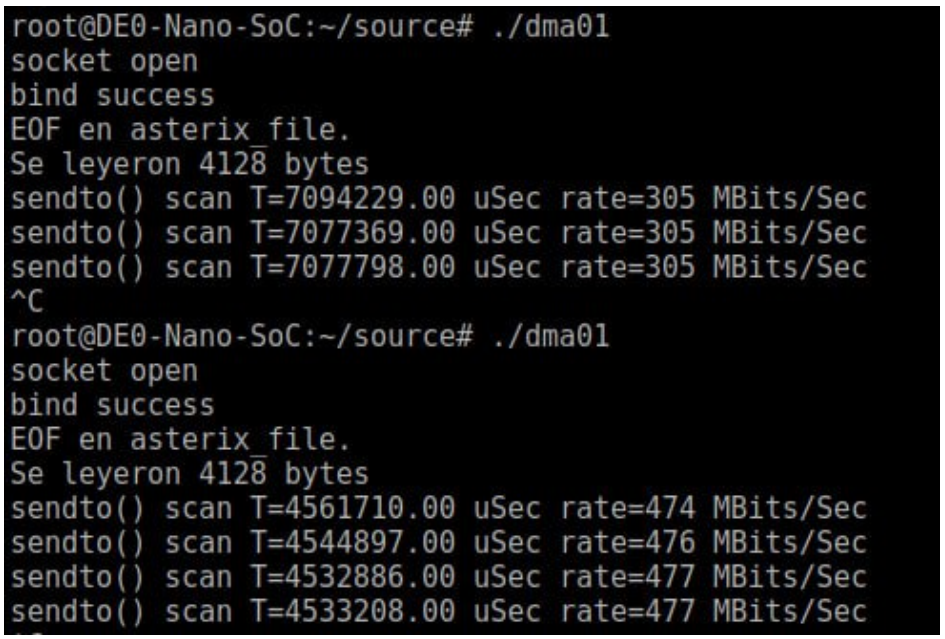

Figura 76. Consola de Linux. Ejecución código dma01. Modo debug vs modo release. Tiempos de transmisión.

Puede observarse que la tasa de velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real es menor con el código optimizado (modo *release*) respecto al código no optimizado (modo *debug*). Esto ocurre debido a que la instrucción *memcpy* es más óptima cuando el código no está optimizado (modo *debug*)[19].

2.4.3.5. Capacidad del canal

Posteriormente, se realizó otro ensayo en el cual se cambió de lugar la instrucción *memcpy* en el código realizado.

El código empleado está realizado para tramas o paquetes variables pero como, en este caso, estamos trabajando con tramas o paquetes fijos, este cambio es realizado con el fin de obtener la capacidad del canal.

La capacidad de canal es una medida de la máxima cantidad de información que puede transmitirse de forma fiable a través de un canal de comunicaciones.

Por lo tanto:

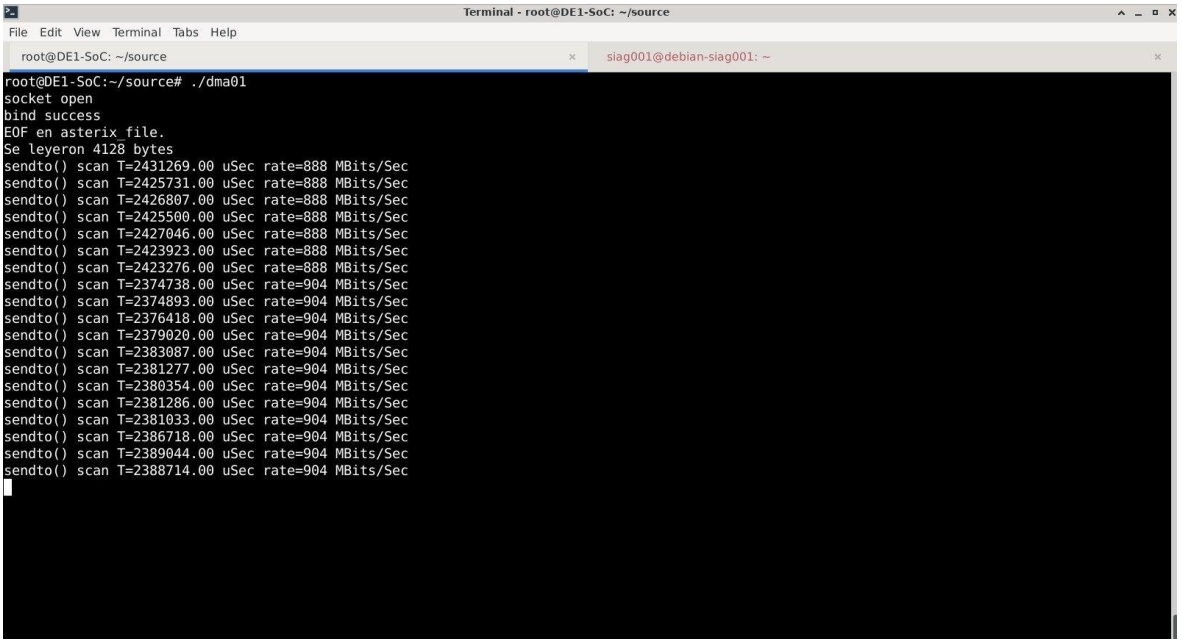

Figura 77. Consola de Linux. Ejecución código dma01. Tiempos de transmisión.

En pocas palabras, a diferencia de los ensayos anteriores[5], la copia de los datos o trama empaquetada en protocolo ASTERIX CAT240 se realiza por única vez por lo que, dentro del ciclo, solo se realiza el incremento del *azimut* para, posteriormente, poder graficar.

Considerando datos o información útil, tenemos que:

$$\frac{65536\ paquetes \cdot 4096[Bytes]}{2.4[segundos]} \approx 112[MB/s]$$

La velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real se calculó de la siguiente manera:

$$\frac{65536\ paquetes \cdot 4096[Bytes] \cdot 8[bits]}{2.4[segundos]} \approx 895[Mbps]$$

De los resultados obtenidos, puede observarse que el canal permite alcanzar velocidades de hasta casi 900 [Mbps].

2.4.3.6. Modificación de la frecuencia del clock del PLL *h2f_axi_clk*

Se modificó la frecuencia del clock del PLL *h2f_axi_clk*, que es el clock encargado de manejar los clocks de los bridges entre la FPGA y el HPS.

La finalidad de modificar la frecuencia de dicho clock es observar si existe un incremento o una mejoría en la velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real.

Hasta el momento, todos los ensayos realizados fueron para una frecuencia del *h2f_axi_clk* de 100 [MHz].

[5] ver ensayos 2.4.3.1, 2.4.3.2, 2.4.3.3 y 2.4.3.4.

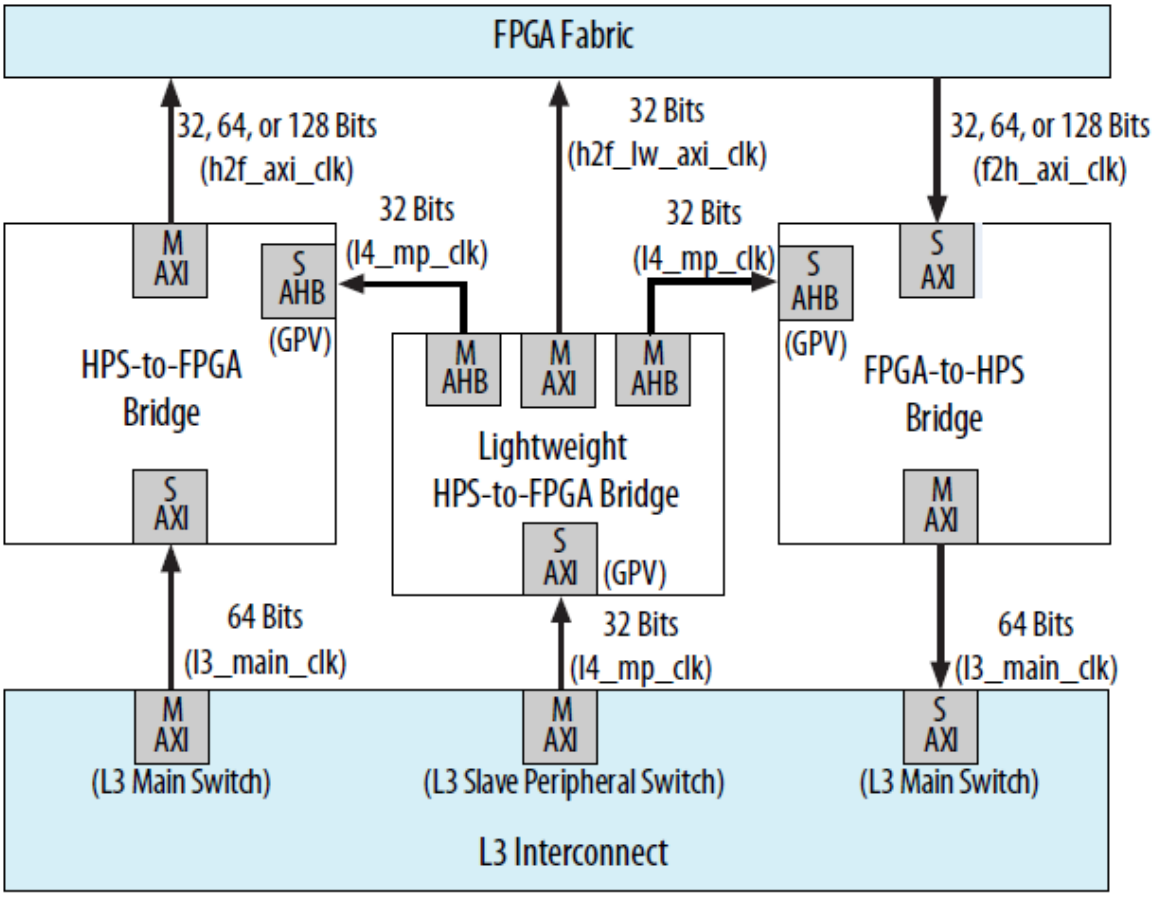

Figura 78. Bridges HPS-FPGA.

Para el caso de *h2f_axi_clk = 200 MHz* y un tamaño de paquete o trama empaquetada en protocolo ASTERIX CAT240 de 1056 [bytes], donde los datos o información útil, es decir, sin contar encabezados y demás, son 1024 [bytes], tenemos que:

```
root@DE1-SoC:~/source# ./dma01
socket open
bind success
EOF en asterix_file.
Se leyeron 1056 bytes
sendto() scan T=1226224.00 uSec rate=448 MBits/Sec
sendto() scan T=1228217.00 uSec rate=448 MBits/Sec
sendto() scan T=1227971.00 uSec rate=448 MBits/Sec
sendto() scan T=1227722.00 uSec rate=448 MBits/Sec
sendto() scan T=1227289.00 uSec rate=448 MBits/Sec
sendto() scan T=1227867.00 uSec rate=448 MBits/Sec
sendto() scan T=1226744.00 uSec rate=448 MBits/Sec
sendto() scan T=1228278.00 uSec rate=448 MBits/Sec
sendto() scan T=1230175.00 uSec rate=448 MBits/Sec
sendto() scan T=1227631.00 uSec rate=448 MBits/Sec
sendto() scan T=1227121.00 uSec rate=448 MBits/Sec
sendto() scan T=1228706.00 uSec rate=448 MBits/Sec
```

Figura 79. Consola de Linux. Ejecución código dma01. Tiempos de transmisión.

Considerando datos o información útil, tenemos que:

$$\frac{65536\ paquetes \cdot 1024[Bytes]}{1.23[segundos]} \approx 54.56[MB/s]$$

Por lo que la velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real se calculó de la siguiente manera:

$$\frac{65536\ paquetes \cdot 1024[Bytes] \cdot 8[bits]}{1.23[segundos]} \approx 436.48[Mbps]$$

En cambio, para un tamaño de paquete o trama empaquetada en protocolo ASTERIX CAT240 de 4128 [bytes], donde los datos o información útil, es decir, sin contar encabezados y demás, son 4096 [bytes], tenemos que la velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real obtenida para diferentes frecuencias del clock del PLL *h2f_axi_clk*.

- Para el caso de *h2f_axi_clk = 110 MHz*, se obtuvo una tasa de transmisión efectiva de datos UDP de aproximadamente 494 MHz.
- Para el caso de *h2f_axi_clk = 125 MHz*, se obtuvo una tasa de transmisión efectiva de datos UDP de aproximadamente 509 MHz.
- Para el caso de *h2f_axi_clk = 150 MHz*, se obtuvo una tasa de transmisión efectiva de datos UDP de aproximadamente 520 MHz.

Para el caso de *h2f_axi_clk = 200 MHz*, manteniendo el tamaño de paquete o trama empaquetada en protocolo ASTERIX CAT240 de 4128 [bytes], donde los datos o información útil son 4096 [bytes], tenemos que:

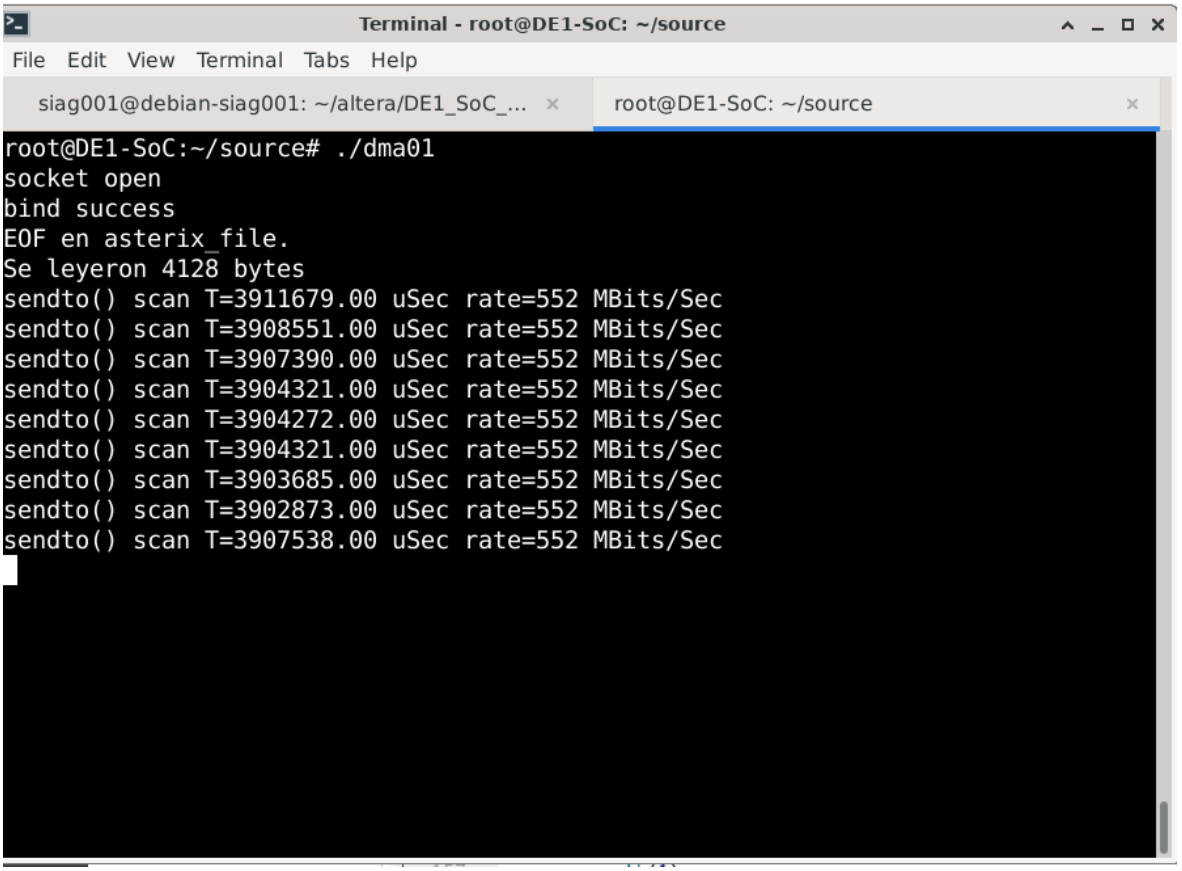

Figura 80. Consola de Linux. Ejecución código dma01. Tiempos de transmisión.

Considerando datos o información útil, tenemos que:

$$\frac{65536\ paquetes \cdot 4096[Bytes]}{3.9[segundos]} \approx 69[MB/s]$$

Por lo que la velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real se calculó de la siguiente manera:

$$\frac{65536\ paquetes \cdot 4096[Bytes] \cdot 8[bits]}{3.9[segundos]} \approx 550[Mbps]$$

De los resultados obtenidos, puede observarse que un incremento en la frecuencia del clock del PLL *h2f_axi_clk* conlleva a un aumento en la velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real.

2.4.3.7. Medición de la velocidad de lectura de la memoria compartida entre la FPGA y el HPS

Otro ensayo realizado consiste en realizar la medición de los tiempos de lectura de datos desde la memoria compartida entre la FPGA y el HPS, cuyo tamaño es de 8192 [bytes].

Además, se realizan tres lecturas de memoria por ejecución de programa (se ejecuta la instrucción *memcpy* tres veces consecutivas) y se realiza un promedio de los tiempos de lectura arrojados con la finalidad de obtener un tiempo fidedigno.

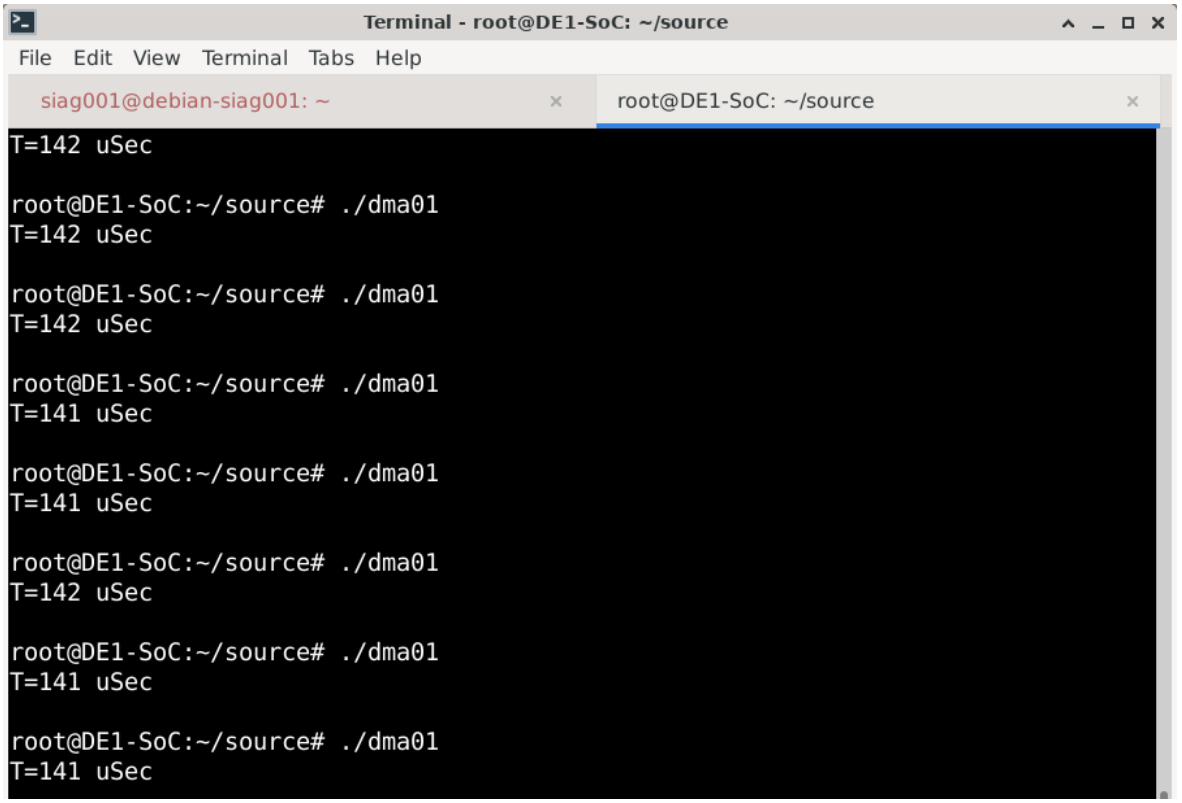

Figura 81. Consola de Linux. Ejecución código dma01. Tiempos de lectura.

Entonces, tenemos que:

$$\frac{Lectura\ de\ la\ memoria[veces]\cdot Tamaño\ de\ la\ memoria[Bytes]}{Tiempo\ de\ lectura[segundos]} = \frac{3\cdot 8192[Bytes]}{141\cdot 10^{-6}[segundos]} \approx 166[MB/s]$$

Por lo que la velocidad de lectura de la memoria en la FPGA desde el HPS se calculó de la siguiente manera:

$$Velocidad\ de\ lectura\ de\ la\ memoria\ [\frac{MB}{s}]\cdot 8[bits] = 166[\frac{MB}{s}]\cdot 8[bits] \approx 1330[Mbps]$$

A continuación, en forma de resumen, se realizan tablas comparativas con la finalidad de observar de manera más sencilla los resultados obtenidos en los ensayos realizados.

Tamaño de trama [bytes]	**Tasa de transferencia**		**Modo**	**Frecuencia del clock del PLL *h2f_axi_clk* [MHz]**
	[MB/s]	**[Mbps]**		
1056 (1024 bytes útiles)	~ 48.63	~ 389	Debug	100
1056 (1024 bytes útiles)	~ 54.56	~ 436.48	Debug	200
2080 (2048 bytes útiles)	~ 50.65	~ 405.2	Debug	100
4128 (4096 bytes útiles)	~ 59	~ 472	Debug	100
4128 (4096 bytes útiles)	~ 37.81	~ 302.46	Release	100
4128 (4096 bytes útiles)	~ 61.75	~ 495	Debug	110
4128 (4096 bytes útiles)	~ 63.63	~ 510	Debug	125
4128 (4096 bytes útiles)	~ 65	~ 520	Debug	150
4128 (4096 bytes útiles)	~ 69	~ 550	Debug	200

Tabla 1. Tasa de transferencia obtenida para diferentes tamaños de trama o paquete.[6]

[6] ver ensayos 2.4.3.1, 2.4.3.2, 2.4.3.3, 2.4.3.4 y 2.4.3.6.

Tamaño de trama [bytes]	Tasa de transferencia		Modo	Frecuencia del clock del PLL *h2f_axi_clk* [MHz]
	[MB/s]	[Mbps]		
4128 (4096 bytes útiles)	~ 112	~ 895	Debug	100

Tabla 2. Capacidad del canal.[7]

Tamaño de la memoria [bytes]	Tasa de transferencia		Modo	Frecuencia del clock del PLL *h2f_axi_clk* [MHz]
	[MB/s]	[Mbps]		
8192	~ 166	~ 1330	Debug	200

Tabla 3. Velocidad de lectura de la memoria compartida entre la FPGA y el HPS.[8]

2.5. Software (con la implementación del handshake entre la FPGA y el HPS)

En esta sección del informe se comenta en detalle el software realizado para la comunicación entre la FPGA y el HPS, el cual se divide fundamentalmente en dos partes: una realizada en *Qsys*, en la cual se instancia todo el hardware necesario para la comunicación entre la FPGA y el HPS, y otra parte realizada en *QtCreator*, la cual se utiliza para programar el HPS.

El software realizado se encarga, en primera instancia, de mapear los puertos para lectura de memoria de la FPGA a memorias virtuales para que las mismas puedan ser accedidas desde el lado del HPS. Además, se agrega un registro extra con el cual se realiza el *handshake* entre la FPGA y el HPS. En pocas palabras, a través de dicho registro, la FPGA le dice al HPS qué memoria está lista para leer y qué cantidad de datos tiene almacenados la misma. Las memorias, las cuales tienen un tamaño máximo de 2048 [bytes], se van llenando con datos empaquetados en protocolo ASTERIX CAT240 desde el lado de la FPGA y, una vez que éstas estén listas, el HPS las va leyendo. Posteriormente, se transfiere el contenido de dichas memorias a un arreglo para, finalmente, enviar dichos datos por Ethernet a través del socket UDP.

2.5.1. Parte Qsys

El *Qsys* es una herramienta gráfica que se utiliza para el diseño digital de hardware. Dicha herramienta contiene procesadores, memorias, interfaces de I/O, timers, etc. La herramienta *Qsys* le permite al usuario, mediante una GUI, elegir los componentes deseados y automáticamente genera el hardware para conectarlos.

La herramienta *Qsys* permite diseñar el sistema y la comunicación entre el HPS y la FPGA. El HPS y la FPGA están conectadas a través de una serie de *AXI Bridges*. En este caso, nosotros trabajaremos desde el lado del HPS por lo que, para la comunicación entre ellos, se utilizan dos *bridges*:

[7] ver ensayo 2.4.3.5.
[8] ver ensayo 2.4.3.7.

- HPS-to-FPGA bridges (*h2f*).
- Lightweight HPS-to-FPGA bridge (*lwh2f*).

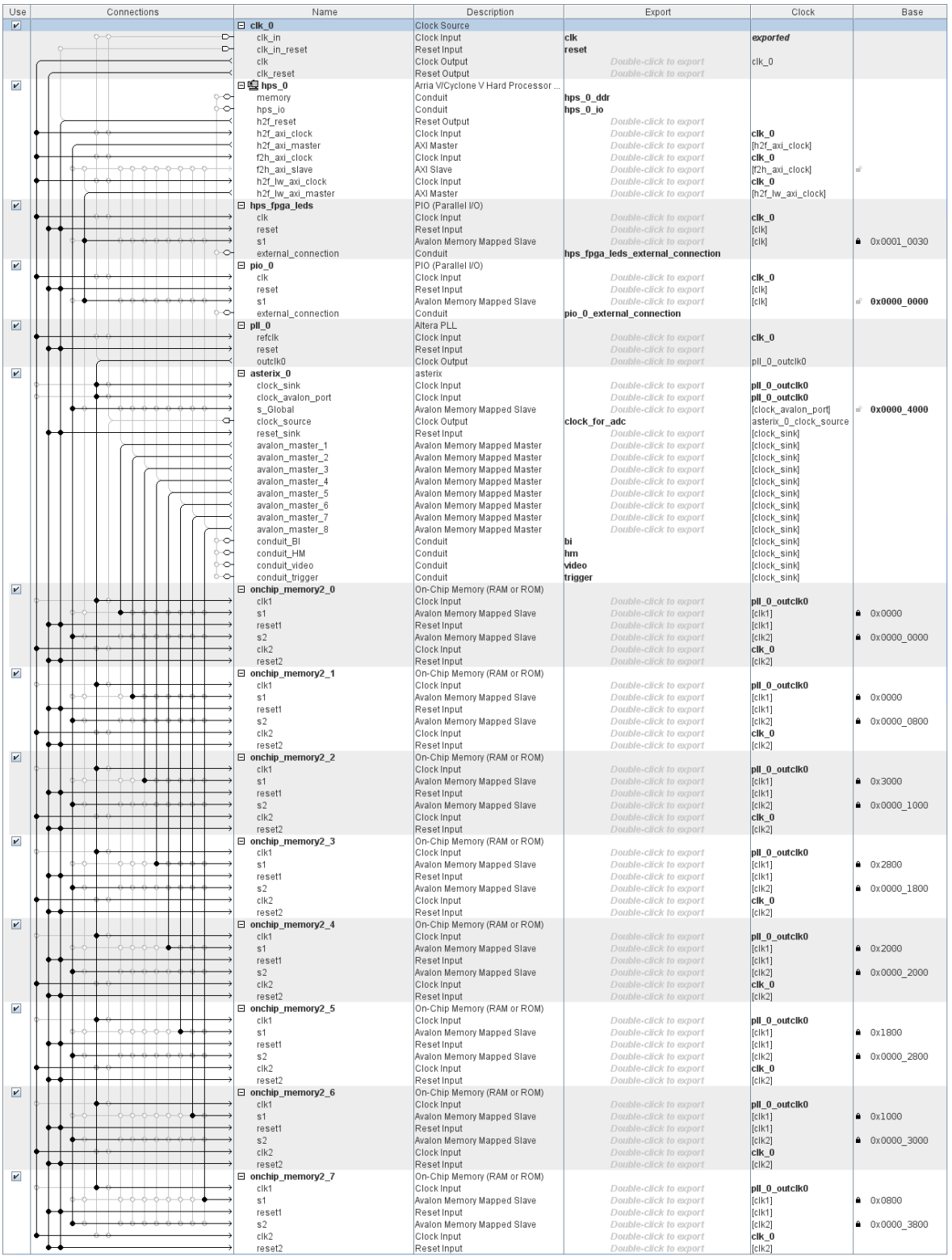

Figura 82. Qsys. Componentes instanciados.

Tal como puede observarse en la imagen *ut supra*, se instancian 8 memorias, denominadas *onchip_memory2_X*, con *X* de 0 a 7. Por cada memoria se almacena un paquete o trama empaquetadas en protocolo ASTERIX CAT240.

Además, se instancia un componente denominado *asterix_0*, el cual se encarga de recibir las señales de sincronismo, de empaquetar los datos en protocolo ASTERIX CAT240, de realizar el *handshake* entre la FPGA y el HPS, de grabar las memorias con los paquetes o tramas empaquetadas en protocolo ASTERIX CAT240, entre otras funciones. Dicho componente es parte del desarrollo realizado por el Dr. Ricardo Cayssials y el alumno Mariano Valdez en el marco del proyecto.

Una vez que todo el hardware ha sido configurado correctamente, la comunicación entre el HPS y la FPGA se programa a través de una aplicación en C mapeada en memoria. Dicho mapeo de memoria permite que la CPU vea y acceda al espacio de direcciones de la FPGA, la cual contiene los componentes, para poder así leer/escribir información según sea necesario.

La aplicación en C desarrollada utiliza una API para enviar, o recibir, datos de escritura a, y desde, direcciones de memoria especificadas.

2.5.2. Parte QtCreator

Una vez configurado el hardware para la comunicación entre el HPS y la FPGA a través de la herramienta gráfica de diseño digital de hardware *Qsys*, se procede a programar el HPS.

Como se necesita que el HPS acceda a los periféricos que forman parte de la estructura de la FPGA, lo primero que se debe de realizar es la instanciación en el proyecto del *header*, el cual es generado una vez realizada la configuración en *Qsys*, mediante *#include "hps_soc_system.h"*.

```
16    #include "hps_soc_system.h"
```

Figura 83. QtCreator. Instanciación del header generado en Qsys.

Luego, se definen todas las variables y funciones necesarias para realizar los *bridges* entre el HPS y la FPGA.

```
19  // ====================================
20
21  #define H2F_AXI_MASTER_BASE     0xC0000000
22
23  // main bus; scratch RAM, conectada a h2f_axi_master
24  // h2f bus
25  // RAM FPGA port s2
26  // main bus addess 0x0800_0000
27  void *h2f_axi_master_virtual_base;
28
29  // ====================================
30  // lw_bus;
31  // h2f_lw_axi_master -> control port
32  // read_master       -> f2h_axi_slave, puedo leer cualquier periferico que este en este bus,
33  // write_master      -> onchip_memory2_0.s1 = 0x00020000
34  #define H2F_LW_AXI_MASTER_BASE      0xff200000
35  #define HW_REGS_SPAN                0x00005000
36
37  // the h2f light weight bus base
38  void *h2p_lw_virtual_base;
39
40  // ====================================
41
42  // HPS onchip memory base/span
43  // 2^16 bytes at the top of memory
44  #define HPS_ONCHIP_BASE     0xffff0000
45  #define HPS_ONCHIP_SPAN     0x00010000
46  // HPS onchip memory (HPS side!)
47  volatile unsigned int * hps_onchip_ptr = NULL ;
48  void *hps_onchip_virtual_base;
49
50  //====================================
```

Figura 84. QtCreator. Definición de todas las variables y funciones.

Puede observarse que cada uno de los componentes poseen una dirección base. Dichas direcciones son utilizadas para acceder, controlar y enviar datos desde y hacia el SoC.

Para mapear las direcciones físicas a direcciones virtuales, lo primero que se realiza es una llamada abierta al sistema para abrir el controlador o *driver* del dispositivo de memoria *"/dev/mem"* seguido de la llamada al sistema *mmap*, la cual es utilizada para asignar la dirección física del HPS a una dirección virtual representado por el puntero *h2f_axi_master_virtual_base*.

```
54  // /dev/mem file id
55  int fd;
```

Figura 85. QtCreator. ID del archivo /dev/mem.

```
71  //map variables
72
73  volatile unsigned *global;
74  volatile unsigned *mem1;
75  volatile unsigned *mem2;
76  volatile unsigned *mem3;
77  volatile unsigned *mem4;
78  volatile unsigned *mem5;
79  volatile unsigned *mem6;
80  volatile unsigned *mem7;
81  volatile unsigned *mem8;
```

Figura 86. QtCreator. Definición de todas las variables necesarias para realizar el mapeo de memoria.

La variable *global* es la que será utilizada para la realización del *handshake*, entre otras cosas, entre la FPGA y el HPS, tal como se explicará más adelante en la presente sección del informe.

```
uint8_t mapping(void){

    // Declare volatile pointers to I/O registers (volatile
    // means that IO load and store instructions will be used
    // to access these pointer locations,
    // instead of regular memory loads and stores)

    // === get FPGA addresses ==================
    // Open /dev/mem
    if( ( fd = open( "/dev/mem", ( O_RDWR | O_SYNC ) ) ) == -1 )    {
        printf( "ERROR: could not open \"/dev/mem\"...\n" );
        return( 1 );
    }

//=============================================
    // get virtual addr that maps to physical
    // for light weight bus
    // DMA status register
    h2p_lw_virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, H2P_LW_AXI_MASTER_BASE );
    if( h2p_lw_virtual_base == MAP_FAILED ) {
        printf( "ERROR: mmap1() failed...\n" );
        close( fd );
        return(1);
    }

//=============================================

    //  Mapeo de puertos para lectura de memoria y Handshake
    h2f_axi_master_virtual_base = mmap( NULL, 0x3C000000, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, H2F_AXI_MASTER_BASE );
    global = (volatile unsigned *)((uint8_t *)h2f_axi_master_virtual_base + ASTERIX_0_BASE);
    mem1 = (volatile unsigned *)((uint8_t *)h2f_axi_master_virtual_base+ONCHIP_MEMORY2_0_BASE);
    mem2 = (volatile unsigned *)((uint8_t *)h2f_axi_master_virtual_base+ONCHIP_MEMORY2_1_BASE);
    mem3 = (volatile unsigned *)((uint8_t *)h2f_axi_master_virtual_base+ONCHIP_MEMORY2_2_BASE);
    mem4 = (volatile unsigned *)((uint8_t *)h2f_axi_master_virtual_base+ONCHIP_MEMORY2_3_BASE);
    mem5 = (volatile unsigned *)((uint8_t *)h2f_axi_master_virtual_base+ONCHIP_MEMORY2_4_BASE);
    mem6 = (volatile unsigned *)((uint8_t *)h2f_axi_master_virtual_base+ONCHIP_MEMORY2_5_BASE);
    mem7 = (volatile unsigned *)((uint8_t *)h2f_axi_master_virtual_base+ONCHIP_MEMORY2_6_BASE);
    mem8 = (volatile unsigned *)((uint8_t *)h2f_axi_master_virtual_base+ONCHIP_MEMORY2_7_BASE);

// =============================================

    // HPS onchip ram
    hps_onchip_virtual_base = mmap( NULL, HPS_ONCHIP_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, HPS_ONCHIP_BASE);

    if( hps_onchip_virtual_base == MAP_FAILED ) {
        printf( "ERROR: mmap3() failed...\n" );
        close( fd );
        return(1);
    }
    // Get the address that maps to the HPS ram
    hps_onchip_ptr =(unsigned int *)(hps_onchip_virtual_base);

// =============================================

    return (0);
}
```

Figura 87. QtCreator. Mapeo de direcciones físicas a direcciones virtuales. Función mapping.

La dirección virtual de *AXI_MASTER_BASE* está representada por *h2f_axi_master_virtual_base*, que es un puntero con el cual se podrá acceder directamente a los registros en el controlador.

Esto se realiza con el fin de poder acceder a los periféricos que forman parte de la estructura de la FPGA, lo cual se logra mapeando las direcciones de la FPGA a una memoria virtual para que las mismas puedan ser accedidas desde el lado del HPS.

Para la transmisión de datos empaquetados en UDP a través de Ethernet, se realiza un *socket UDP*[1]. Para ello, en primera instancia, se deben agregar las librerías *#include <sys/socket.h>*, *#include <unistd.h>*, *#include <arpa/inet.h>* y las variables necesarias.

```
13  #include <sys/socket.h>
14  #include <unistd.h>
15  #include <arpa/inet.h>
```

Figura 88. QtCreator. Librerías dedicadas al socket UDP.

```
61  //=====================================
62
63  //socket variables
64  int sfd;
65  int port = 8000;
66  struct sockaddr_in address, dst_address;
67
68  //=====================================
```

Figura 89. QtCreator. Definición de todas las variables necesarias para la realización del socket UDP.

```
82  //=====================================
83
84  void init_socket(void){
85
86      sfd = socket(AF_INET, SOCK_DGRAM, 0);
87
88      if(sfd < 0){
89          printf("Socket failed\n");
90          exit(1);
91      }
92      printf("socket open\n");
93
94      int optval=1;
95      setsockopt(sfd, SOL_SOCKET, SO_REUSEPORT, &optval, sizeof(optval));
96
97      address.sin_family = AF_INET;
98      address.sin_addr.s_addr = htonl(INADDR_ANY);
99      address.sin_port = htons(port);
100
101     int r = 0;
102     r = bind(sfd, (struct sockaddr*) &address, sizeof(address));
103     if(r < 0){
104         printf("bind failed");
105     }
106     printf("bind success\n");
107
108     dst_address.sin_family = AF_INET;
109     inet_pton(AF_INET, "192.168.7.21", &dst_address.sin_addr);
110     dst_address.sin_port = htons(port);
111 }
112 //=====================================
```

Figura 90. QtCreator. Creación del socket UDP. Función init_socket.

Tal como puede observarse, la creación del *socket* se realiza mediante la función *socket()*[8]. Para ello, debe definirse el descriptor de archivo de socket que, en este caso, es *sfd* (*socket file descriptor*).

Mediante la función *setsockopt()*[9], se establecen o setean los parámetros del *socket* creado.

Además, se debe definir un puerto, que en nuestro caso será el *port=8000*, y una estructura *sockaddr*, la cual será empleada para inicializar el *socket* creado.

Por lo tanto, se debe inicializar la dirección del *socket* fuente, denominado *in_address*, por medio de la función *bind()*[10]. Dicha función asocia el *socket* creado a una dirección particular o a un puerto específico. Lo mismo debe de realizarse para inicializar el *socket* de destino, denominado *dst_address*.

Posteriormente, dentro del *main()*, se realiza el llamado a las funciones *init_socket()* y *mapping()*, funciones comentadas anteriormente, además de la declaración de variables, cuyo uso o función se comentará a continuación.

```
169  // ============================================
170
171  int main(void)
172  {
173
174  init_socket();
175
176  //============================================
177
178  uint8_t a;
179
180  a = mapping();
181
182  if(a != 0) return( EXIT_FAILURE );
183
184  //============================================
185
186      unsigned int data[1024] __attribute__((aligned(4)));
187      unsigned int reg0 __attribute__((aligned(4)));
188      unsigned int num_mem __attribute__((aligned(4)));
189      unsigned int payload_size __attribute__((aligned(4)));
190      unsigned int dr __attribute__((aligned(4)));
191      volatile unsigned *mem[] = {mem1, mem2, mem3, mem4, mem5, mem6, mem7, mem8};
192
193      int old_buff __attribute__((aligned(4)));
194      int resta __attribute__((aligned(4)));
195
196      num_mem = 0;
197      payload_size = 0;
198      dr = 0;
199      old_buff = 0;
200
201      global[2] = 0x3E8;
202      global[3] = 0x3;
203      global[7] = 0x10;
```

Figura 91. Llamado a funciones init_socket() y mapping(). Declaración de variables.

La variable *data* es un arreglo de datos utilizado para cargar o copiar el contenido que se tiene cargado en la memoria SRAM dual port, que es nada más ni nada menos que el paquete o trama empaquetada en protocolo ASTERIX CAT240 desde el lado de la FPGA.
La variable *reg0* es utilizada para saber si hay o no datos almacenados o disponibles en alguna de las memorias.
La variable *num_mem* es utilizada para guardar el número de memoria a leer, es decir, guardar el número de la memoria que esté disponible con datos.

La variable *payload_size* es utilizada para guardar el tamaño del paquete o trama empaquetado en protocolo ASTERIX CAT240 que se encuentra disponible en memoria.
La variable *dr* es la variable utilizada para calcular el data rate o la tasa de transferencia de datos.
Las variables *old_buff* y *resta*, además de la ya mencionada variable *num_mem*, son las variables utilizadas para la verificación de pérdida de paquetes.

Tal como puede observarse en la figura *ut supra*, se les especificó a las variables comentadas previamente un atributo especial del tipo *aligned*[14] con la finalidad de optimizar el código. Al utilizar el atributo especial *aligned*, se busca que el compilador realice menos operaciones lo que mejora considerablemente la eficiencia del tiempo de ejecución.

Tal como se comentó inicialmente, para la realización del *handshake* entre la FPGA y el HPS, se agregó un registro extra el cual se conecta al HPS para que éste pueda acceder a los registros internos del módulo *asterix_0* de la FPGA. Dicho registro es mapeado a través del puntero *global*.

- Registro 0 (*global[0]*). Es de solo lectura y entrega 32 [bits].
    - Si no hay mensaje para enviar desde el lado de la FPGA, entrega *0xFFFFFFFF*.
    - Si hay mensaje para enviar desde el lado de la FPGA, entrega:
        - En los 16 [bits] menos significativos, el número de memoria en dónde se encuentran los datos almacenados. Como son 8 memorias, su valor va de 0 a 7.
        - En los 16 [bits] más significativos, el tamaño de trama o paquete (*payload* + encabezado) empaquetado en protocolo ASTERIX CAT240. Este tamaño de trama o paquete es coincidente con el campo *LEN* del paquete o trama empaquetada en protocolo ASTERIX CAT240.
- Registro 1 (*global[1]*). Es de lectura y escritura. Entrega el valor del encoder del RADAR, el cual por defecto es 4096. Básicamente, genera el *HM* cuando se generan internamente las señales de vídeo RADAR (autogeneración de vídeo).
- Registro 2 (*global[2]*). Es de lectura y escritura. Entrega el número de muestras. Por lo tanto, el tamaño de la trama dependerá del ancho de la muestra, es decir, dependerá de si se considera vídeo RADAR de 8 [bits] o de 16 [bits]. Un detalle a tener en cuenta es que el tamaño de paquete o trama no puede ser mayor a 1500 [bytes] ya que, al momento de transmitir por Ethernet UDP, se fragmenta ni tampoco puede ser mayor a 2048 [bytes], que es el tamaño de la memoria. Recordar que se almacena una trama o paquete por memoria. Otro detalle a tener en cuenta es que la trama o paquete se arma dependiendo del *Trigger* y/o del *BI* por lo que su tamaño es variable.
- Registro 3 (*global[3]*). Es de lectura y escritura. Tiene 2 [bits].
    - *Bit[0] = 0*, toma las señales de vídeo RADAR externas.
    - *Bit[0] = 1*, genera internamente las señales de vídeo RADAR (autogeneración de vídeo).
    - *Bit[1] = 0*, considera vídeo RADAR de 8 [bits].
    - *Bit[1] = 1*, considera vídeo RADAR de 16 [bits].
- Registro 4 (*global[4]*). Solo válido en autogeneración de vídeo. Entrega la cantidad de períodos de reloj de ADC (16 [MHz]) entre *Triggers*. Por defecto está en 16000. Recordar que cada vuelta de RADAR son 32 [MS], es decir, 2 segundos por vuelta.

- Registro 5 (*global[5]*). Solo válido en autogeneración de vídeo. Entrega la cantidad de períodos de reloj de ADC (16 [MHz]) entre *BI*. Por defecto está en 7812. Recordar que cada vuelta de RADAR son 32 [MS], es decir, 2 segundos por vuelta, y se desea tener 4096 *BI* por vuelta.
- Registro 6 (*global[6]*). Entrega el valor de *CELL_DUR* en femtosegundos. Por defecto está en 62500000 [femtosegundos] para 16 [MHz] de clock de conversor AD.
- Registro 7 (*global[7]*). Solo válido en autogeneración de vídeo. Entrega la cantidad de *BI* entre generación de *HM*. En otras palabras, entrega el incremento de ángulo por cada *BI* en resolución $360° = 2^{16} = 65536$. Por defecto está en 16 ya que $azimut/BI = 65536/4096 = 16$.

Por lo tanto, se le asignaron valores a algunos de los registros internos comentados anteriormente, tal como puede observarse en la figura *ut supra*.
Se le asignó el valor decimal 1000 (*0x3E8* en hexadecimal) a *global[2]*, que es el registro que se utiliza para modificar el tamaño del paquete o trama empaquetada en protocolo ASTERIX CAT240. Un detalle a tener en cuenta es que a *global[2]*, como máximo, se le puede asignar el valor decimal 1440 (*0x5A0* en hexadecimal) para que, una vez que los datos son encapsulados primero en protocolo ASTERIX CAT240 y luego en protocolo UDP para ser enviados por Ethernet a la red, no se fragmente debido a que se supera el *MTU*. Cabe recordar que el *MTU* es de 1500 [bytes], el encabezado IP es de 20 [bytes], el encabezado UDP es de 8 [bytes] y el encabezado ASTERIX CAT240 es de 32 [bytes] dando como resultado un *payload* máximo de 1440 [bytes][9]. Por otro lado, se le asignó el valor decimal 3 (*0x3* en hexadecimal) a *global[3]*, que es el registro que se utiliza para la generación interna o toma externa de las señales de vídeo RADAR. Al ser su valor 3, significa que el *Bit[0] = 1* y *Bit[1] = 1*, por lo que genera internamente las señales de vídeo RADAR (autogeneración de vídeo) y además considera vídeo RADAR de 16 [bits]. Finalmente, se le asignó el valor decimal 16 (*0x10* en hexadecimal) a *global[7]*, que es el registro que se utiliza para modificar la relación entre el incremento de ángulo (*azimut*) y el *BI*.

Tal como puede observarse en la figura *ut infra*, se ejecuta un ciclo *while* en el cual se realiza, a grosso modo, un ciclo *do* en el cual se lee el *Registro 0* con el fin de verificar si hay datos disponibles. Si es así, se lee en qué memoria se encuentra y el tamaño del mismo. Luego, se realiza una verificación de pérdida de paquetes. Posteriormente, se realiza el copiado del contenido almacenado en la memoria al arreglo *data* para realizar el envío o transmisión del mismo a través del socket UDP. Finalmente, se realiza el cálculo de la tasa de transferencia.

[9] ver Anexo B y Anexo C.

```
234     while(1){
235
236         gettimeofday(&t1, NULL);
237
238         do{
239
240             reg0 = global[0];
241
242             if(reg0 == 0xFFFFFFFF) continue;
243
244             else{
245                 num_mem = (reg0 & 0xFFFF);
246                 payload_size = (reg0 & 0xFFFF0000) >> 16;
247
248                 resta = num_mem - old_buff;
249
250                 if( !(resta == 1 || resta == -7) ){
251                     printf("Error: %d\n", resta);
252                     printf("Actual: %d Anterior: %d\n", num_mem, old_buff);
253
254                 }
255
256                 else
257                     printf("-");
258
259                 break;
260             }
261
262         }while(1);
263
264         memcpy((void*)data, (const void*)mem[num_mem], payload_size);
265
266         sendto(sfd, (void *)data, payload_size, 0, (struct sockaddr *)&dst_address, sizeof(dst_address));
267
268         old_buff = num_mem;
269
270         gettimeofday(&t2, NULL);
271
272         elapsedTime = (t2.tv_sec - t1.tv_sec) * 1000000.0;
273         elapsedTime += (t2.tv_usec - t1.tv_usec);
274
275         dr = (payload_size / (uint32_t)elapsedTime) * 8;
276
277         printf("data rate = %d MBits/Sec\n\r", dr);
278
279         }
280     }
```

Figura 92. QtCreator. Lectura del Registro 0, adquisición de la trama ASTERIX CAT240, verificación de pérdida de paquetes, copiado y posterior transmisión de la trama por Ethernet a través del socket UDP y cálculo de la tasa de transferencia.

Desglosando lo comentado previamente, al comienzo del ciclo *while* se tiene un ciclo *do*, en el cual se lee el valor del *Registro 0* y se lo compara con el valor hexadecimal *0xFFFFFFFF*. Si es verdadero, significa que no hay datos almacenados en memoria. En cambio, si es falso, significa que sí hay datos almacenados en memoria por lo que se leen los 16 [bits] menos significativos para saber en qué memoria (número de memoria) se encuentran y los 16 [bits] más significativos para conocer el tamaño.
Posteriormente, por medio de dos sentencias *if*, se realiza una simple verificación de pérdida de paquetes. Cabe recordar que se almacena un paquete o trama empaquetada en protocolo ASTERIX CAT240 por buffer o memoria por lo que la idea es, al momento de leer la memoria compartida entre la FPGA y el HPS desde el lado de éste último, chequear de alguna manera que no se está omitiendo ninguna memoria o, dicho de otra manera, perdiendo paquetes. Por lo tanto, en el caso de que se llegasen a perder paquetes, se imprime un error en el cual se informa el valor de la variable *resta* con el que se sabe qué cantidad de buffer o memoria fueron omitidas mientras que si se imprime el carácter *"-"* significa que no se perdió ningún paquete o trama.

Haciendo uso de las funciones *gettimeofday()*[15] y *elapsedTime*, la cual retorna el tiempo transcurrido entre dos valores de tiempo, calculados previamente con la función *gettimeofday()*, se calcula la tasa de transferencia de datos.
Puede observarse que el cálculo de la tasa de transferencia de datos contempla los tiempos muertos y no solo la adquisición y posterior transmisión de los datos a través de Ethernet por medio del *socket* UDP.

Haciendo uso de la función *memcpy()*[16], se transfiere el contenido almacenado en la memoria *mem[num_mem]* de tamaño *payload_size* al arreglo *data.*

```
264         memcpy((void*)data, (const void*)mem[num_mem], payload_size);
```

Figura 93. QtCreator. Copiado del contenido almacenado en memoria mem[num_mem] de tamaño payload_size al arreglo data mediante la instrucción memcpy().

Finalmente, por medio de la función *sendto()*[18], se envían los datos empaquetados en UDP, previamente empaquetados en protocolo ASTERIX CAT240, a través de Ethernet por medio del *socket* creado inicialmente.

```
266         sendto(sfd, (void *)data, payload_size, 0, (struct sockaddr *)&dst_address, sizeof(dst_address));
```

Figura 94. QtCreator. Envío de datos a través del socket.

2.6. Ensayos (con la implementación del handshake entre la FPGA y el HPS)

En la presente sección del documento se detallan los ensayos realizados empleando el kit de desarrollo DE1-SoC, utilizando el software desarrollado y explicado en la sección previa del presente informe.

El ensayo a realizar consiste en una prueba de medición de tiempos con el fin de conocer los tiempos de lectura y de transmisión de datos empaquetados en protocolo ASTERIX CAT240. Dichos datos son generados, empaquetados en protocolo ASTERIX CAT240 y posteriormente cargados en memoria RAM dual port, que es la memoria compartida entre la FPGA y el HPS, desde el lado de la FPGA. Se implementa, además, un *handshake* entre la FPGA y el HPS con el fin de que exista una sincronización entre ambos. Por lo tanto, por medio del mismo, la FPGA le comunica al HPS cuando el dato está listo para que éste lo lea, lo copie y posteriormente lo transmita por Ethernet a través de un socket UDP dando como resultado una velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real. Además, se realizará una prueba de visualización de los datos transmitidos por Ethernet haciendo uso del software *RadarView*, de Cambridge Pixel.

2.6.1. Medición de tiempos de lectura y de transmisión con datos empaquetados en protocolo ASTERIX CAT240

Haciendo uso de una PC y del kit de desarrollo DE1-SoC, se realizará una prueba de medición de tiempos. Dicho ensayo consiste en la lectura de datos empaquetados en protocolo ASTERIX CAT240, los cuales fueron previamente generados, empaquetados en dicho protocolo y cargados en memoria RAM dual port, que es la memoria compartida entre la FPGA y el HPS, desde el lado de la FPGA. Además, se realiza la implementación de un *handshake* entre la FPGA y el HPS con el fin de que exista una sincronización entre ambos. Así, la FPGA le comunicará al HPS cuando el dato está listo para que éste lo lea, lo

copie y posteriormente lo transmita por Ethernet a través de un socket UDP. Por lo tanto, la medición de los tiempos de lectura y de transmisión, es decir, el tiempo máximo que el HPS necesita para leer un dato en memoria, copiarlo y transmitirlo por Ethernet a través de un socket UDP, darán como resultado una velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real.

La prueba de medición de tiempos se lleva a cabo utilizando la consola de Linux, donde se ejecuta el código *dma01* desde el lado del HPS. El código empleado para realizar el ensayo en cuestión está basado en un código encontrado en Internet[19][20]. En pocas palabras, el código se encarga, en primera instancia, de mapear las direcciones de la FPGA a una memoria virtual para que las mismas puedan ser accedidas desde el lado del HPS. Luego, a través de un registro, se realiza el *handshake* entre la FPGA y el HPS. En pocas palabras, por medio de dicho registro, la FPGA le dice al HPS qué memoria está lista para leer (*num_mem*) y qué cantidad de datos tiene almacenados la misma (*payload_size*). Las memorias, las cuales tienen un tamaño máximo de 2048 [bytes], se van llenando con datos empaquetados en protocolo ASTERIX CAT240 desde el lado de la FPGA y, una vez que éstas estén listas, el HPS las va leyendo haciendo uso de un puntero denominado *mem[num_mem]*. A continuación, haciendo uso de la función *memcpy()*[16], se transfiere el contenido del puntero *mem[num_mem]* a un arreglo *data* para, finalmente, enviar dichos datos por Ethernet a través del socket UDP empleando la función *sendto()*[18]. Para calcular los tiempos, se emplean las funciones *gettimeofday()*[15] y *elapsedTime*. Esta última retorna el tiempo transcurrido entre dos valores de tiempo, calculados previamente con la función *gettimeofday()*.

Tal como se mencionó en la sección anterior del presente informe, para la realización del *handshake* entre la FPGA y el HPS, se agregó un registro el cual es mapeado a través del puntero *global*. A través de dicho puntero, se podrán realizar modificaciones a ciertos registros[10].

En el Anexo B podrá encontrarse una breve reseña del protocolo ASTERIX CAT240, además de una explicación sobre cómo se arma un paquete o trama empaquetado en dicho protocolo y la correspondencia entre el mismo y su representación gráfica, lo cual resultará útil conocer al momento de observar las capturas de tráfico de red realizadas en los ensayos detallados a continuación.

Como punto de partida, se debe asegurar una velocidad de transmisión efectiva de datos que sea igual a la velocidad a la que se cargan los datos en memoria, los cuales dijimos son de tamaño variable. Es decir, como mínimo se debe ser capaz de transmitir a la misma velocidad en que almacenan los datos en memoria para así, mientras se transmite la primera trama, poder leer la segunda y prepararla para la transmisión. Esta sería la peor condición que debemos cumplir o asegurar.

Sabemos que la velocidad de muestreo del ADC es de 16 [MS/s] y cada muestra es de 2 [bytes]. Por lo tanto, la velocidad a la que se cargan los datos en memoria se calcula como:

$$16[MS/s] \cdot 2[Bytes] = 32[MB/s]$$

[10] ver software 2.5.

Por lo tanto, nuestra peor condición a cumplir es el tiempo que se tarda en almacenar o cargar los datos en memoria, es decir, debemos ser capaces de superar una tasa de transferencia de datos de 32[MB/s] la cual está dada por la velocidad de muestreo del ADC.

2.6.1.1. Tamaño de trama variable

El ensayo realizado es para un tamaño de paquete o trama empaquetada en protocolo ASTERIX CAT240 variable.
Tal como se explica en el Anexo B, la trama o paquete se arma con un *trigger* o con un *BI* o, dicho de otra manera, por cada señal de *trigger* o *BI* nueva, se finaliza una trama y se comienza una nueva.
De todas maneras, a través de *global[2]*, que es el registro que se utiliza para modificar el tamaño del paquete o trama empaquetada en protocolo ASTERIX CAT240, se asignó el valor decimal 1000 (*0x3E8* en hexadecimal) por lo que la trama tendrá como máximo un valor de 1000 [bytes] fuera de que puede adoptar valores fijados por las señales de *trigger* o *BI* menores o iguales a 1000 [bytes].
Además, se le asignó el valor decimal 3 (*0x3* en hexadecimal) a *global[3]*, que es el registro que se utiliza para la generación interna o toma externa de las señales de vídeo RADAR. Al ser su valor 3, significa que el *Bit[0] = 1* y *Bit[1] = 1*, por lo que genera internamente las señales de vídeo RADAR (autogeneración de vídeo) y además considera vídeo RADAR de 16 [bits].
Finalmente, se le asignó el valor decimal 16 (*0x10* en hexadecimal) a *global[7]*, que es el registro que se utiliza para modificar la relación entre el incremento de ángulo (*azimut*) y el *BI*.

Por lo tanto, tenemos que:

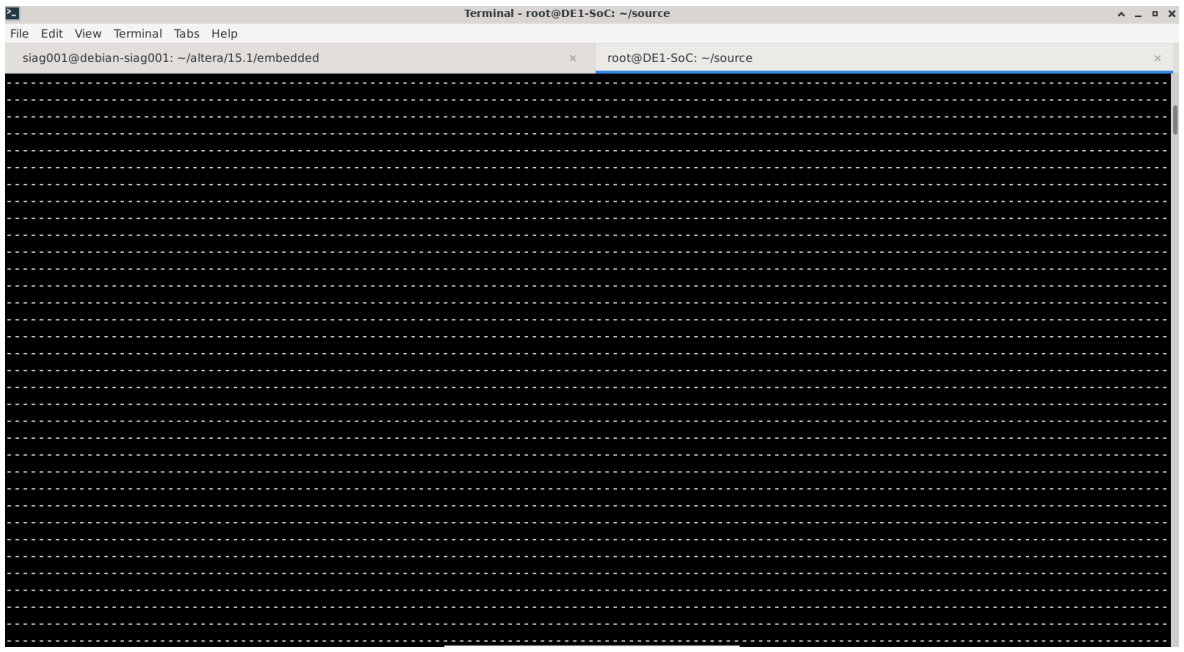

Figura 95. Consola de Linux. Ejecución código dma01. Verificación de pérdida de paquetes.

Podemos observar que, al momento de ejecutar el código, no se omite ningún buffer o memoria por lo que no hay pérdida de paquetes.

Figura 96. Consola de Linux. Ejecución código dma01. Tiempos de transmisión.

Además, podemos observar que la velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real es variable debido a que el tamaño del paquete o trama empaquetada en protocolo ASTERIX CAT240 es variable.
De todas maneras, podemos apreciar que la máxima tasa de transmisión efectiva de datos UDP en tiempo real lograda es del orden de los 270 [Mbps].

Con el fin de constatar la tasa de transmisión efectiva de datos UDP en tiempo real lograda, la cual fue calculada por software[11], se utilizaron los software *NetPerSec* y el *Administrador de Tareas*. Por medio de los mismos, se midió la tasa de recepción de datos UDP en tiempo real, tal como se observa de las figuras *ut infra*.

[11] ver software 2.5.

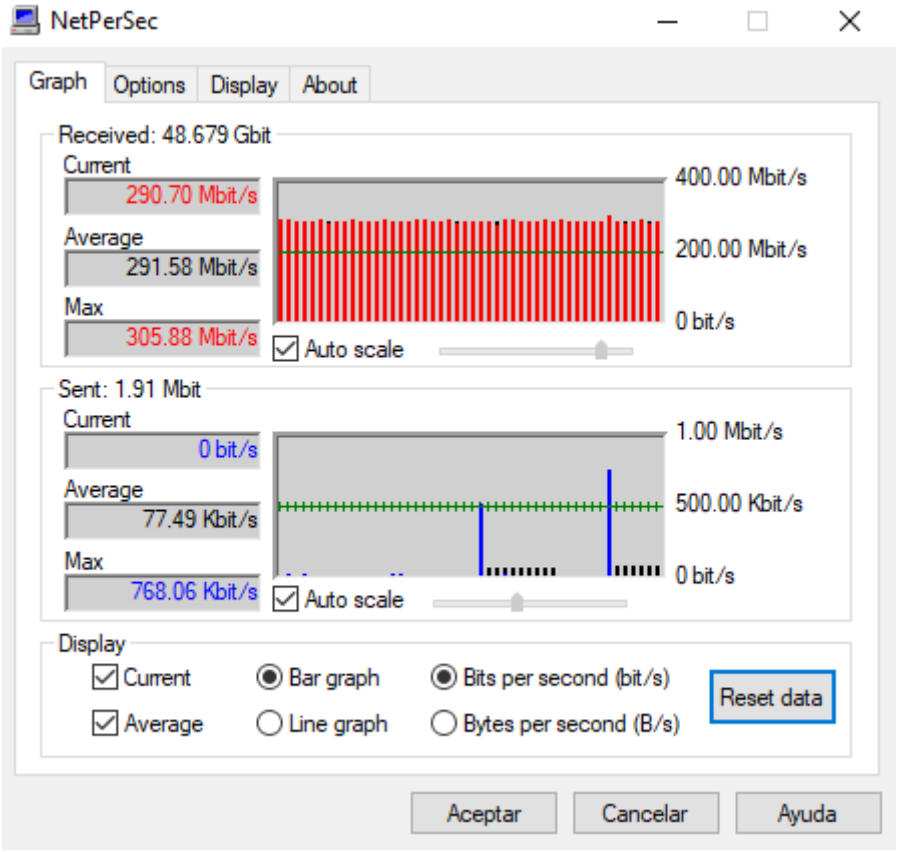

Figura 97. Software NetPerSec. Tasa de recepción de datos UDP en tiempo real.

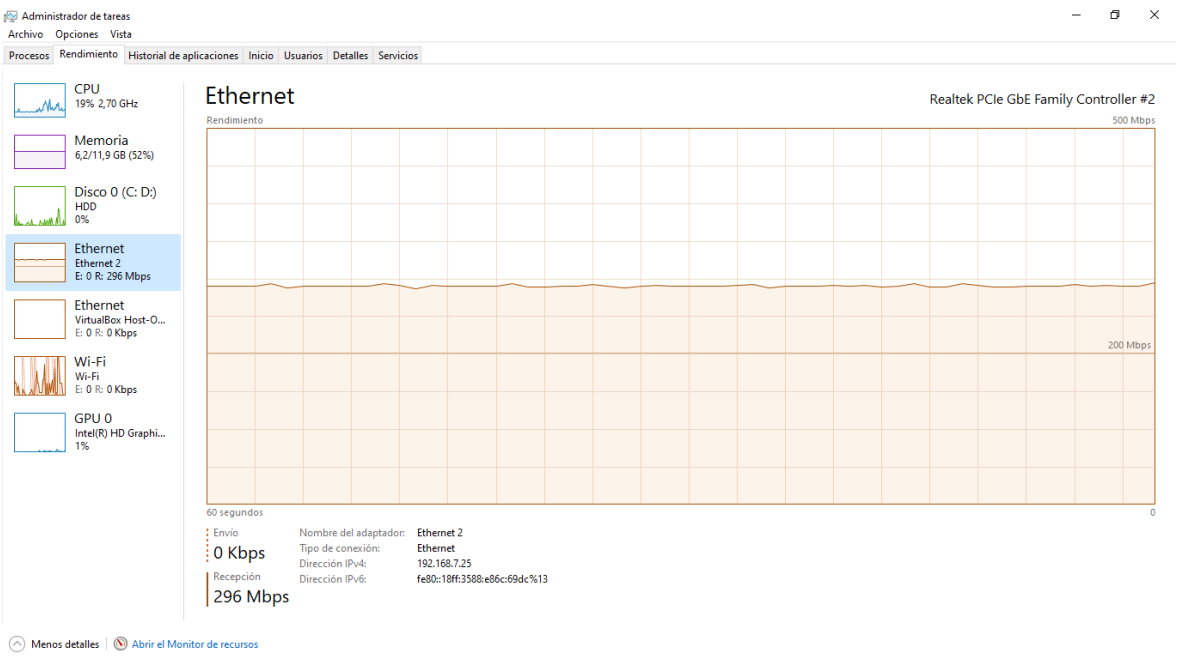

Figura 98. Administrador de Tareas. Rendimiento. Tasa de recepción de datos UDP en tiempo real.

Puede decirse entonces que, a efectos prácticos, ambas tasas (de transmisión y de recepción) son coincidentes. Se estima que la leve discrepancia existente entre ambas radica en la forma en la que se realiza el cálculo de la tasa de transmisión por software ya que la arrojada tanto por el software *NetPerSec* y el *Administrador de Tareas* son muy similares.

Haciendo uso del software *RadarView*, de Cambridge Pixel, lo que hacemos es graficar los datos enviados por Ethernet a través del socket UDP.

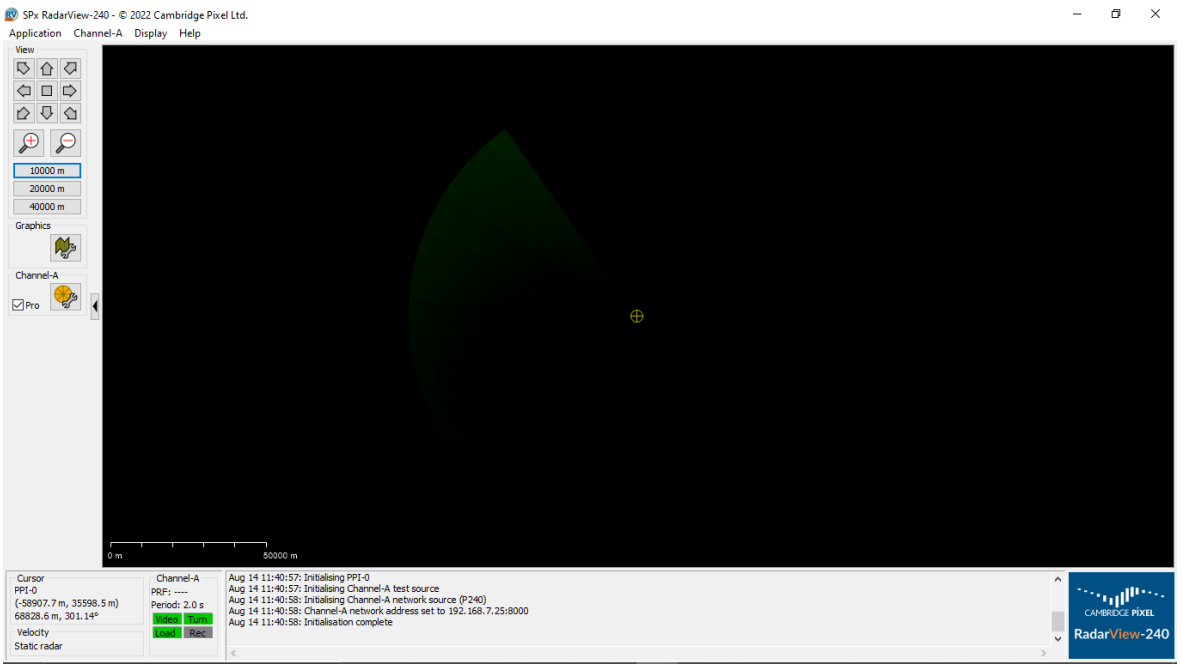

Figura 99. Software RadarView. Visualización de paquetes transmitidos en protocolo ASTERIX CAT240.

En este punto, es necesario realizar una aclaración. Debido a que el conversor AD utilizado es de 14 [bits], la resolución de datos es de 16 [bits] (14 [bits] del conversor AD utilizado + 2 [bits] de padding) por lo que es necesario, para poder visualizar los datos enviados por Ethernet a través del socket UDP en el software *RadarView*, modificar manualmente el parámetro *ChanAPimMaxBytesPerSample* en el archivo de configuración *SPxRadarView-240.rpi*.

Tal como se observa en la figura *ut infra*, dicho parámetro viene con el valor 1 por defecto, es decir, viene dado para una resolución de datos de 8 [bits].

Parameter Name	Start-up?	Type	Description	Default Value
ChanAPimMaxBytesPerSample ChanBPimMaxBytesPerSample				
	Yes	INT	Maximum number of bytes per sample that can be received from an input source. Set to either 1 or 2. If set to 2 then RadarView can receive 2-byte samples from a source; however at present the samples are truncated to 1 byte for processing and display. Enabling support for 2-byte samples increase the memory resources required by RadarView at start-up.	1

Figura 100. Software RadarView. Manual de Usuario. Parámetro ChanAPimMaxBytesPerSample.

Por lo tanto, fue necesario modificar el valor del parámetro *ChanAPimMaxBytesPerSample* de 1 a 2 para así poder visualizar los datos enviados por

Ethernet a través del socket UDP en el software *RadarView* de manera correcta. Para ello, utilizando el *Block de Notas*, se debe abrir el archivo *SPxRadarView-240.rpi* y modificar manualmente dicho parámetro tal y como se observa en la figura a continuación.

Figura 101. Software RadarView. Modificación del parámetro ChanAPimMaxBytesPerSample en el archivo .rpi.

Luego, mediante el software *Wireshark*, se realiza la captura del tráfico de red.

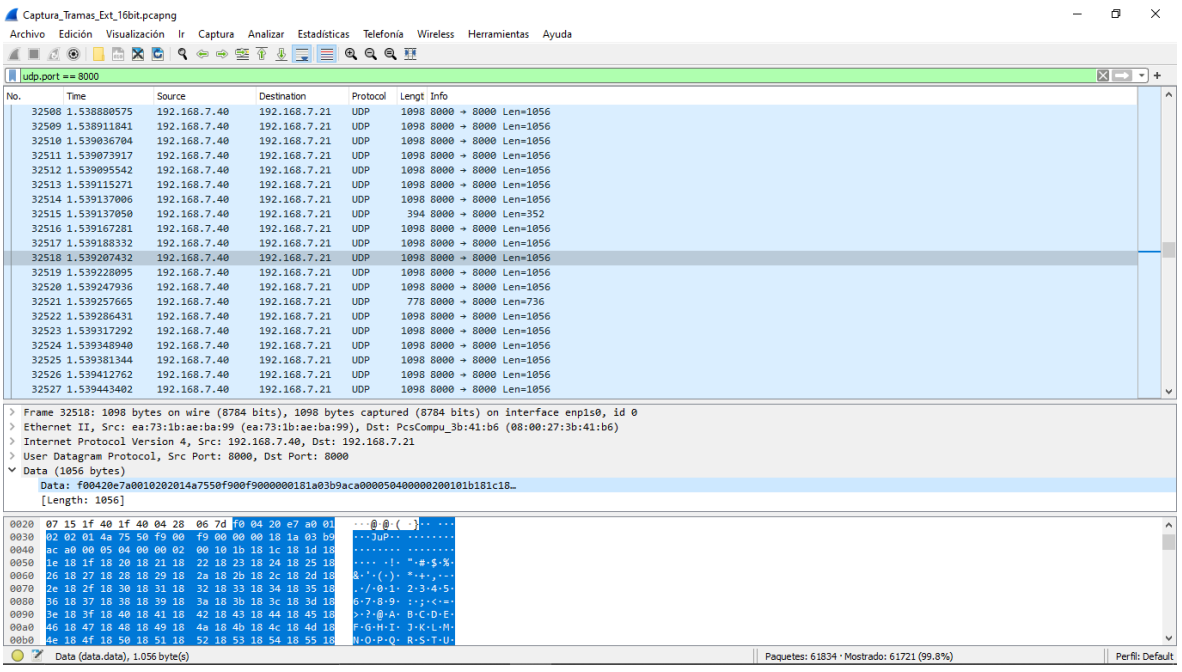

Figura 102. Software Wireshark. Captura de tráfico.

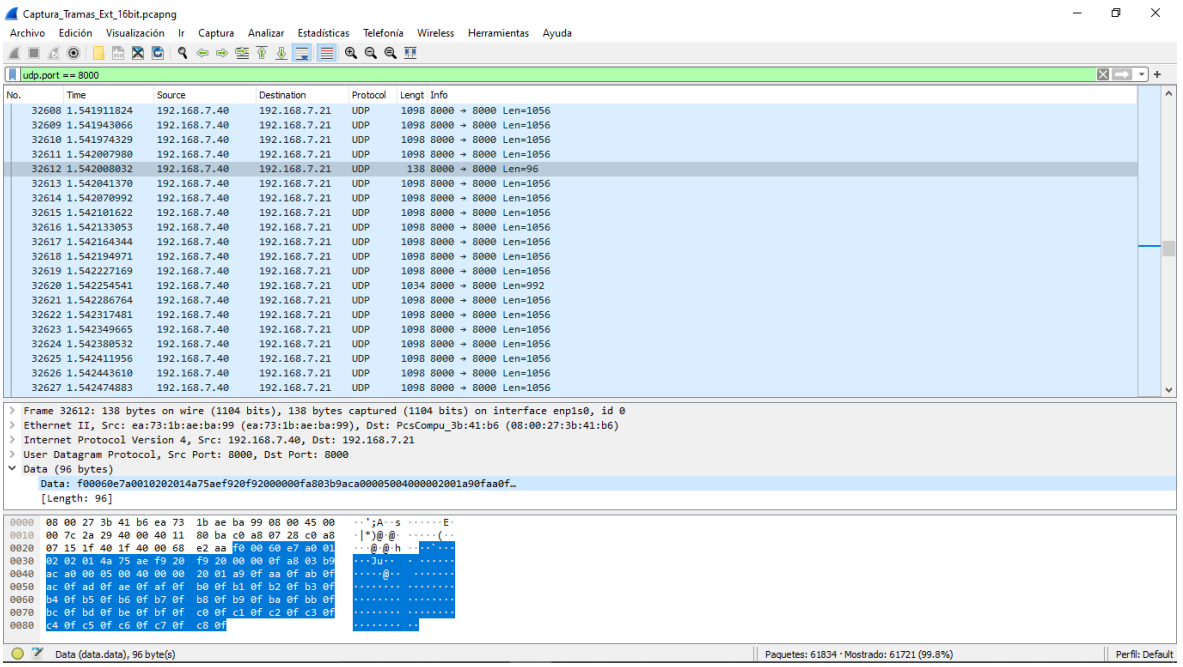

Figura 103. Software Wireshark. Captura de tráfico.

Vale destacar que los datos son transmitidos desde el HPS, cuya dirección IP es 192.168.7.40, y son recibidos por la PC, cuya dirección IP es 192.168.7.21.

Puede observarse que la trama está definitivamente empaquetada en protocolo ASTERIX CAT240 debido a que se identifica el primer byte de la misma con *f0*.

Además, ingresando a la viñeta *Estadísticas -> Propiedades de archivo de captura* del software *Wireshark*, puede realizarse un análisis más minucioso del tráfico de red capturado. En la figura *ut supra* puede observarse la tasa de recepción de datos UDP en tiempo real, tasa prácticamente coincidente con la arrojada por el software *NetPerSec* y el *Administrador de Tareas*, constatando así la tasa de transmisión efectiva de datos UDP en tiempo real lograda.

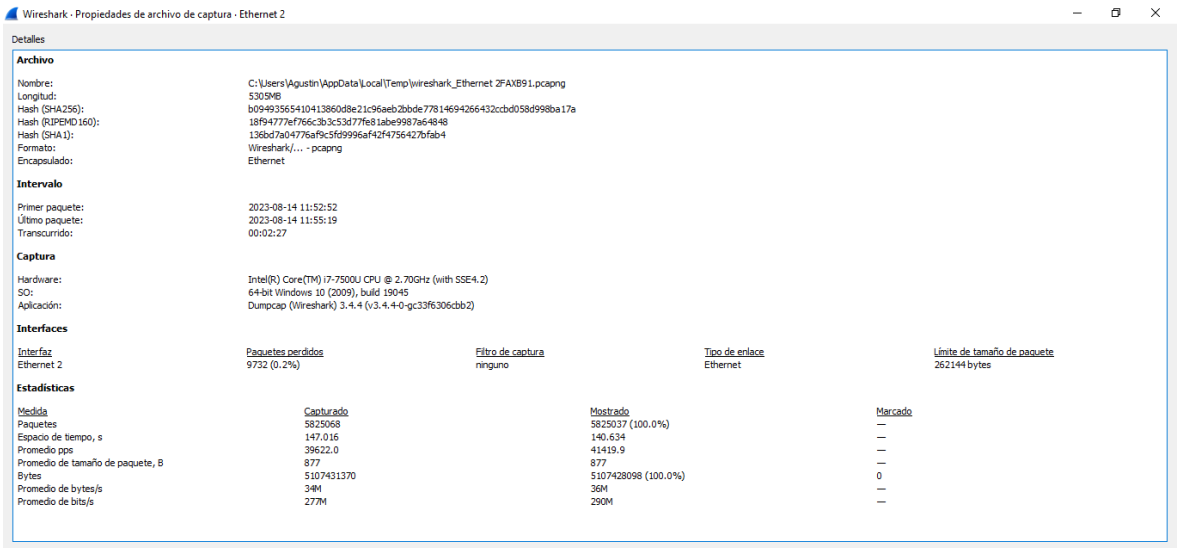

Figura 104. Software Wireshark. Tasa de recepción de datos UDP en tiempo real.

Haciendo uso del software *AsterixInspector*, se puede realizar un análisis minucioso de un paquete o trama empaquetada en protocolo ASTERIX CAT240 capturado.

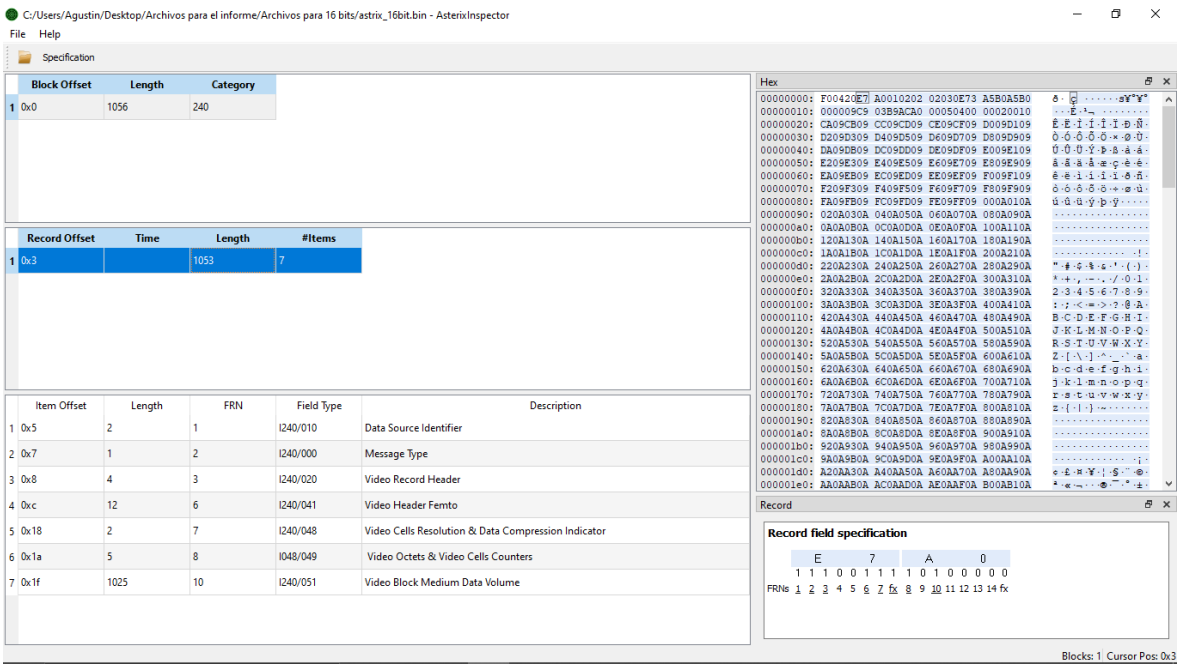

Figura 105. Software AsterixInspector. Análisis de paquete o trama empaquetada en protocolo ASTERIX CAT240.

Mediante dicho software, pueden observarse los diferentes campos del protocolo ASTERIX CAT240 tal y como se detallan en el Anexo B[12].

A continuación, se realiza una tabla comparativa con la finalidad de observar de manera más sencilla los resultados obtenidos en los ensayos realizados con y sin el *handshake* entre la FPGA y el HPS.

Tamaño de trama [bytes]	**Tasa de transferencia Sin Handshake[13]**		**Tasa de transferencia Con Handshake[14]**		**Frecuencia del clock del PLL *h2f_axi_clk* [MHz]**
	[MB/s]	**[Mbps]**	**[MB/s]**	**[Mbps]**	
1056 (1024 bytes útiles)	~ 49	~ 389	~ 36	~ 290	100

Tabla 4. Comparativa en la tasa de transferencia. Sin Handshake vs. Con handshake.

[12] ver Tabla B.1.
[13] ver ensayo 2.4.3.1.
[14] ver ensayo 2.6.1.1.

3. Conclusiones

El proyecto aquí presentado constituye varias etapas que se fueron llevando a cabo a lo largo del tiempo, dando como resultado distintas conclusiones para cada una:

La primera etapa de investigación y desarrollo, en donde se evaluaron diferentes alternativas a través de una extensa investigación y se eligió la implementación utilizada, la cual permitió absorber nuevos conocimientos y permitió el desarrollo de un juicio crítico para elegir un camino que sea acorde a los tiempos de desarrollo del proyecto.

La segunda etapa, en donde se realizó toda la configuración necesaria para poner el marcha el kit utilizado en el desarrollo, permitió la introducción en un nuevo campo de estudio y también aprender nuevos conceptos en materia de sistemas embebidos.

La configuración realizada es compleja por lo que debe hacerse de manera minuciosa ya que cualquier detalle omitido puede dar como resultado una mala configuración y posterior errónea puesta en marcha del kit de desarrollo DE1-SoC.

La tercera etapa consistió en ensayos de la implementación sobre el HPS, para conocer la capacidad del canal y la tasa de transmisión de datos, a su vez la velocidad de lectura del mismo sobre una memoria compartida entre él y la FPGA.

Una vez realizados los ensayos o pruebas, se puede decir que efectivamente se logró el objetivo propuesto, el cual es leer de una memoria compartida entre la FPGA y el HPS datos o tramas empaquetadas en protocolo ASTERIX CAT240, copiarlas y transmitirlas por Ethernet a una velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real mayor a 32[MB/s], que es la velocidad a la que se escribe la memoria desde el lado de la FPGA.

De los ensayos realizados pudo observarse que, a medida que se aumenta el tamaño del paquete o trama empaquetada en protocolo ASTERIX CAT240, la velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real también lo hace aunque, durante la transmisión de los mismos por Ethernet, dichos paquetes debian de ser fragmentados y luego reensamblados debido a que se supera el *MTU* que, para el caso de Ethernet, es de 1500 [bytes]. Desde otra perspectiva, mientras más pequeño sea el tamaño de la trama, menor velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real se logra fuera de que, para estos casos, no existía fragmentación de paquetes. Esta baja en el rendimiento se debe a la instrucción *memcpy*, la cual agrega un tiempo de procesamiento fijo que se pierde, que es el tiempo que se necesita para copiar en un buffer lo que se encuentra almacenado en memoria, cualquiera sea el tamaño de trama o paquete. De los ensayos realizados se observó que, a medida que se aumenta el tamaño de trama, dicho tiempo se hace cada vez menos considerable debido a que se está manejando un volumen mayor de datos pero si, en cambio, se trabaja con tamaño de tramas pequeños, dicho tiempo comienza a tomar importancia lo que impacta directamente en la velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real. Con esto, queda demostrado que la fragmentación de paquetes no impacta negativamente en el rendimiento sino que el causante de la baja en la velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real se debe al tiempo de procesamiento fijo que se pierde al emplear la instrucción *memcpy* para tamaños pequeños de tramas o paquetes.

Pudo observarse además que el limitante que se tiene en la velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real no es la lectura de los datos en la FPGA desde el HPS sino que es la transmisión de los mismos por Ethernet, es decir, el cuello de botella se encuentra en la capacidad del canal. De los ensayos realizados, pudimos observar que la lectura de los datos se hizo a una velocidad de 166 [MB/s][15] mientras que la capacidad máxima del canal fue de 112 [MB/s][16].

Cada ensayo brindó individualmente aportes y conclusiones, las cuales fueron necesarias para el desarrollo del ensayo o prueba posterior. Por lo tanto, puede concluirse que la línea de trabajo adoptada cumple con las expectativas y requerimientos solicitados.

Todas las pruebas expuestas hasta este punto fueron realizadas sin la lógica de sincronización entre la FPGA y el HPS, es decir, sin el *handshake* entre la escritura y lectura de la memoria compartida, donde la FPGA se encarga de escribir y el HPS de leer.

La cuarta etapa conlleva la unificación de dos partes: una implementación en FPGA, la cual recibe muestras de video RADAR además de generar internamente las señales *trigger*, *BI* y *HM*, y genera paquetes de tamaño variable en el protocolo ASTERIX CAT240, que son grabados en buffers compartidos por ambas partes (FPGA y HPS). La segunda parte es un software que se ejecuta sobre un OS montado en un HPS en donde se genera una sincronización entre ambas partes para enviar el contenido de dichos buffers por Ethernet hacia una computadora que ejecute un software de graficación de video RADAR en el protocolo mencionado anteriormente.

Una vez realizado el ensayo o prueba, se pudo concluir que efectivamente se logró el objetivo propuesto, el cual es leer de una memoria compartida entre la FPGA y el HPS datos o tramas empaquetadas en protocolo ASTERIX CAT240, copiarlas y transmitirlas por Ethernet a una velocidad (*throughput*) o tasa de transmisión efectiva de datos UDP en tiempo real mayor a 32[MB/s], que es la velocidad a la que se escribe la memoria desde el lado de la FPGA. De esta manera, puede concluirse que la línea de trabajo adoptada cumple con las expectativas y requerimientos solicitados.

La tasa de transmisión obtenida en la cuarta etapa fue del orden de los 290 [Mbps][17]. La realización del *handshake* tiene un efecto negativo en la tasa de transferencia debido al tiempo muerto que aparece al estar el HPS esperando a que la FPGA llene un buffer.

Aumentar la frecuencia del bus master del HPS, la cual se optó en dejar a 100 [MHz], no mostró mejoras significativas en esta etapa. De hecho, una frecuencia mayor a 150 [MHz] provocó que el sistema se congelara.

La tasa de transmisión puede mejorarse, pero para eso es necesario aumentar el tamaño máximo de muestras modificando el registro pertinente por software, lo cual, en base a los ensayos realizados, si el tamaño de muestras es cercano al tamaño máximo del buffer, se generan desincronizaciones y pérdidas de paquetes. Por tanto, luego de los ensayos realizados, se determinó que tramas de hasta 1440 [bytes][18] son ideales para evitar cualquier tipo de retraso y desincronización en ambas partes, conllevando la pérdida de

[15] ver ensayo 2.4.3.7.
[16] ver ensayo 2.4.3.5.
[17] ver ensayo 2.6.1.1.
[18] ver Anexo C.

paquetes, efecto para nada deseado. Aún así, es posible aumentar por hardware el tamaño de los buffers, de tal manera de aumentar el tamaño de muestras de los paquetes y que no se produzca pérdida de los mismos, pero esto no se consideró necesario por el momento. Además, considerar un tamaño de trama mayor a 1440 [bytes] conlleva a que se supere el *MTU* permitido por Ethernet, generando una fragmentación de paquetes. Cabe destacar que el valor de *MTU* está dado para obtener la mayor eficiencia en la red ya que fragmentar implica perder eficiencia debido a que se deben procesar más encabezados[32]. Aunque para UDP, que es el tipo de transmisión de datos por Ethernet empleada, la fragmentación es transparente (en realidad es el protocolo IP el que realiza este trabajo, maneja fragmentación en el origen y reensamblado en el destino) no existe ninguna garantía de que los paquetes lleguen o lo hagan en el mismo orden en que fueron enviados sin contar que no realiza verificación o control de pérdida de los mismos[33].

Anexo A: Placa DE1-SoC de Terasic

1. Placa DE1-SoC de Terasic

1.1. DE1-SoC de Terasic. Diagrama en bloque

En la Figura A.1 observada a continuación se detalla el diagrama en bloque del kit de desarrollo DE1-SoC de Terasic, el cual contiene la FPGA Intel Cyclone V 5CSEMA5F31C6.

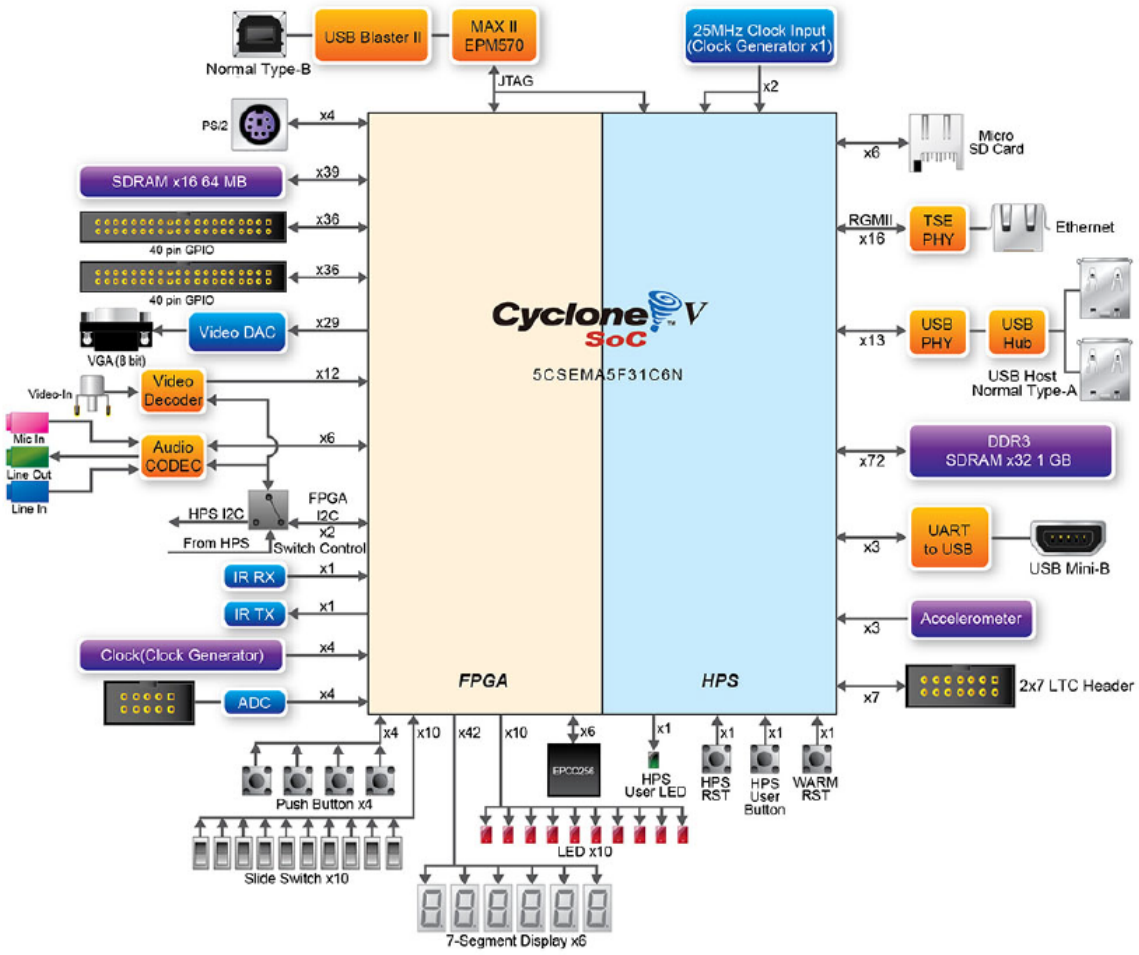

Figura A.1. Kit de desarrollo DE1-SoC de Terasic. Diagrama en bloque.

1.2. DE1-SoC de Terasic. Layout

En la Figura A.2 y Figura A.3, observadas a continuación, se detalla el layout del kit de desarrollo DE1-SoC de Terasic.

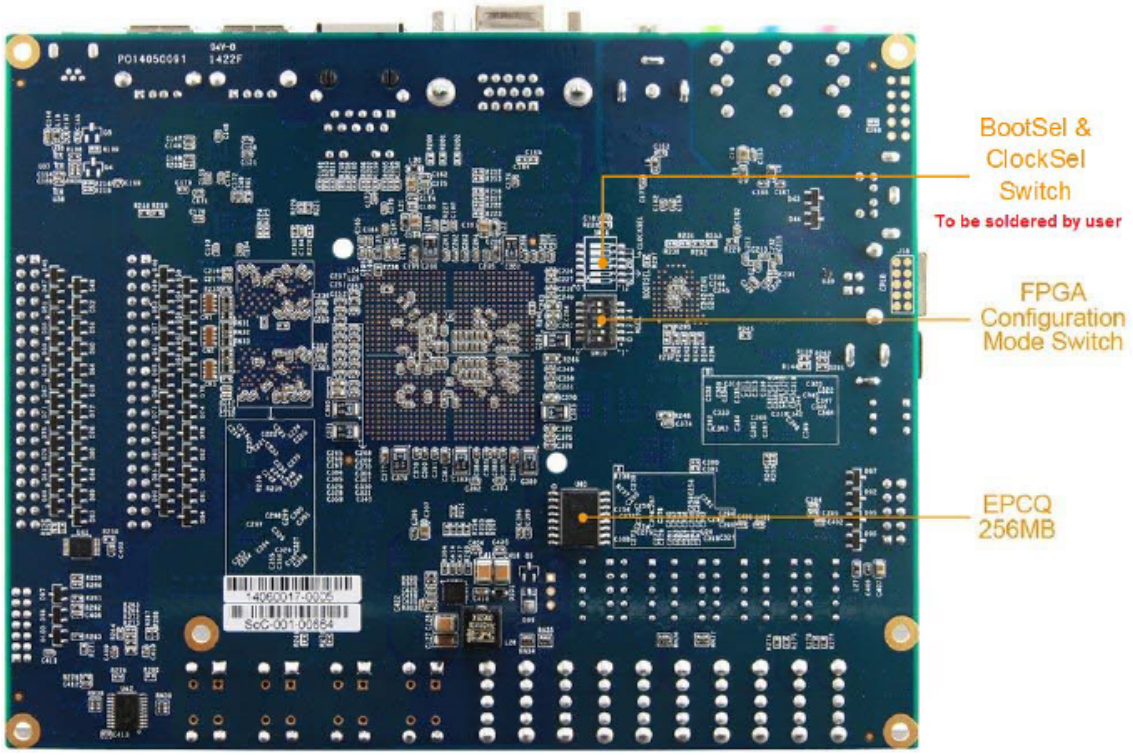

Figura A.2. Kit de desarrollo DE1-SoC de Terasic. Layout. Parte posterior.

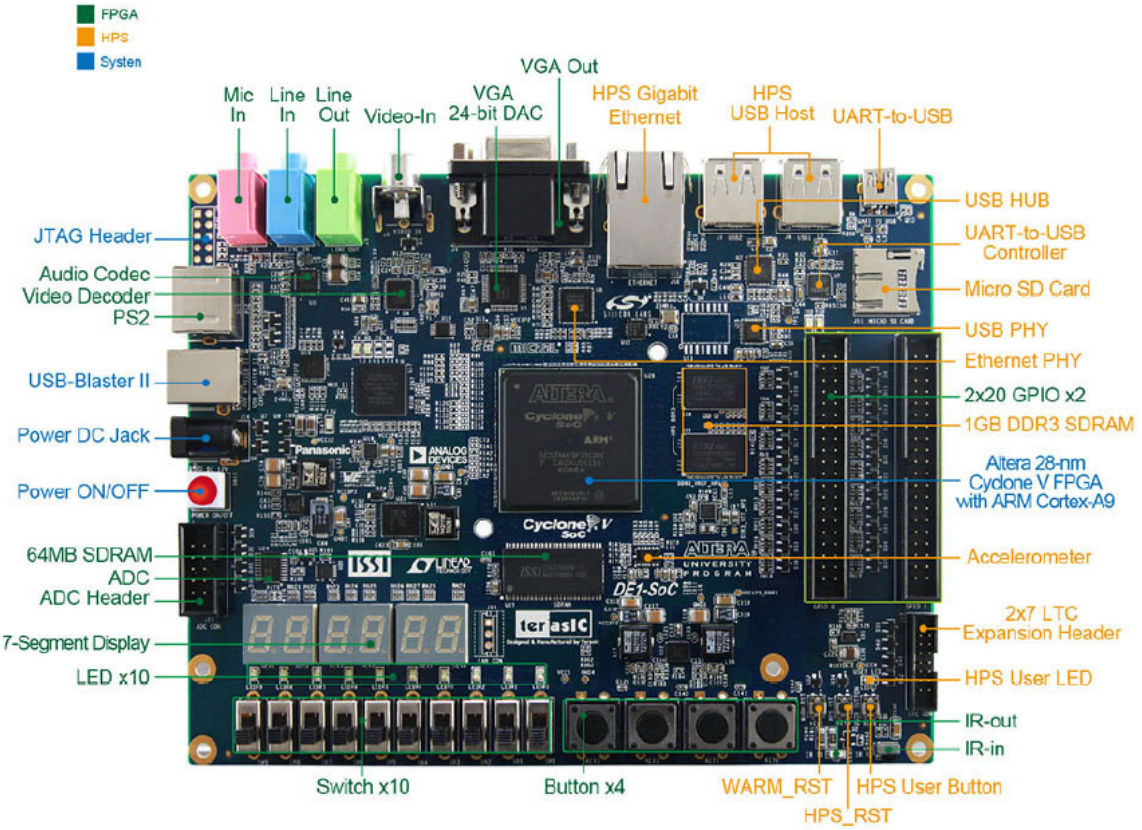

Figura A.3. Kit de desarrollo DE1-SoC de Terasic. Layout. Parte superior.

Para un análisis más extenso y detallado de las características y/o especificaciones del kit de desarrollo DE1-SoC, además de otra documentación relevante, referirse a la bibliografía correspondiente[1][21][22][23].

2. Cyclone V. Descripción general

2.1. Introducción al HPS de Cyclone V

Tal como se observa en la Figura A.4 a continuación, el chip SoC Cyclone V 5CSEMA5F31C6 consiste o se divide en dos partes: la porción HPS y la porción FPGA.

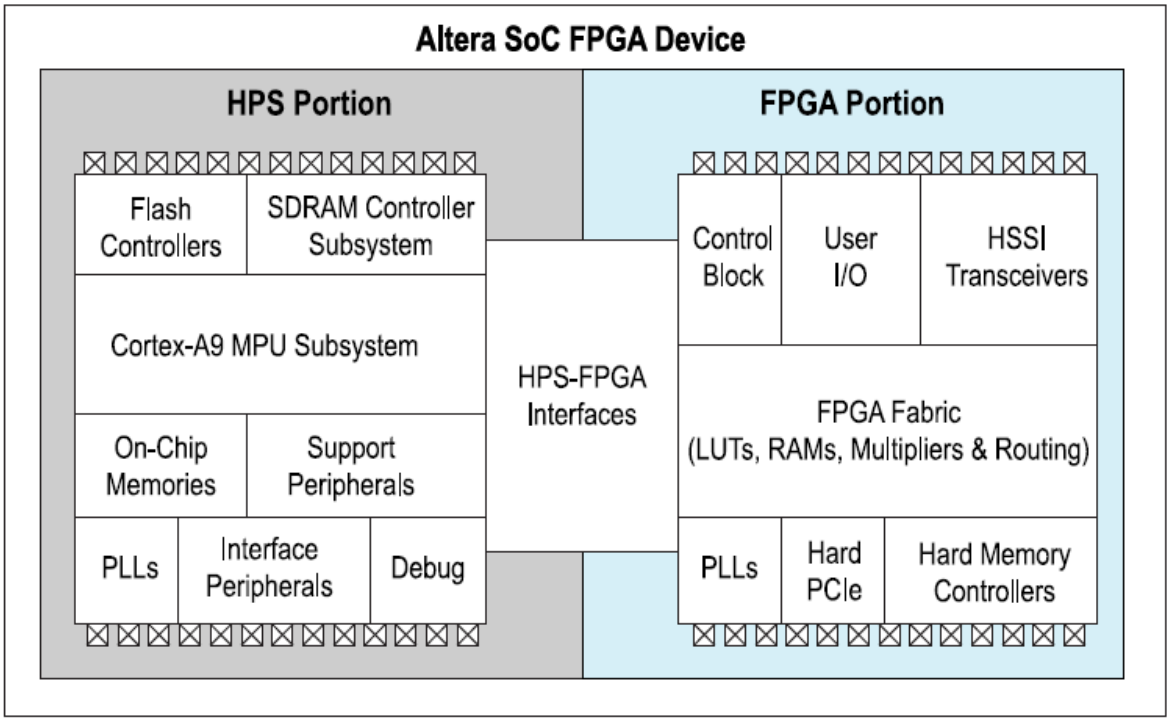

Figura A.4. Chip SoC Cyclone V 5CSEMA5F31C6. Porción FPGA y porción HPS.

Cada una de las porciones del dispositivo, es decir, el HPS y la FPGA, tienen sus propios pines. Los pines no son compartidos libremente entre ambos. Los pines I/O de la FPGA son configurados a través del HPS o cualquier fuente externa soportada por el dispositivo. Los pines I/O del HPS, en cambio, son configurados por software ejecutado en el HPS. El software que configura los pines I/O del HPS es el *preloader*, tal como se explicará en la sección siguiente del presente informe.

2.2. Características del HPS

En la Figura A.5 a continuación se observa un diagrama en bloque de las características más relevantes del HPS.

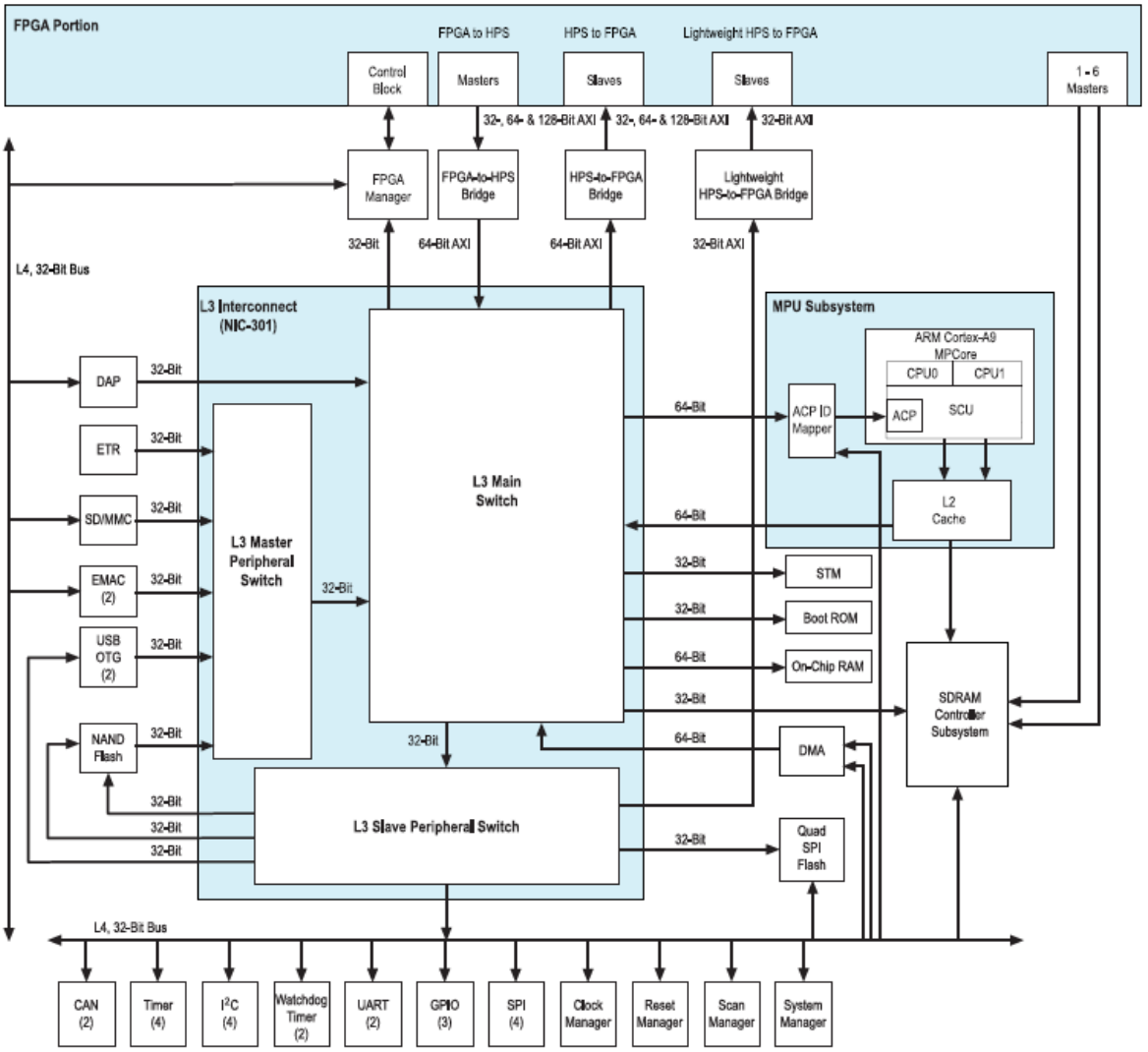

Figura A.5. HPS. Diagrama en bloque.

2.3. Interfaces HPS-FPGA

Las interfaces HPS-FPGA proveen de una variedad de canales de comunicación entre el HPS y la FPGA. Dichas interfaces, las que utilizaremos son las que se enuncian a continuación:

- Bridge de FPGA a HPS.
- Bridge de HPS a FPGA.

2.4. Proceso de booteo del HPS

Bootear software en el HPS es un proceso multietapa. Cada etapa es responsable de cargar la siguiente. La primera etapa del software es la *ROM de arranque* cuyo código localiza y ejecuta la segunda etapa, denominada *preloader*, que localiza (sí está presente) la siguiente etapa. El *preloader* y las etapas de software subsecuentes son denominadas colectivamente como *software de usuario* (*software user*).

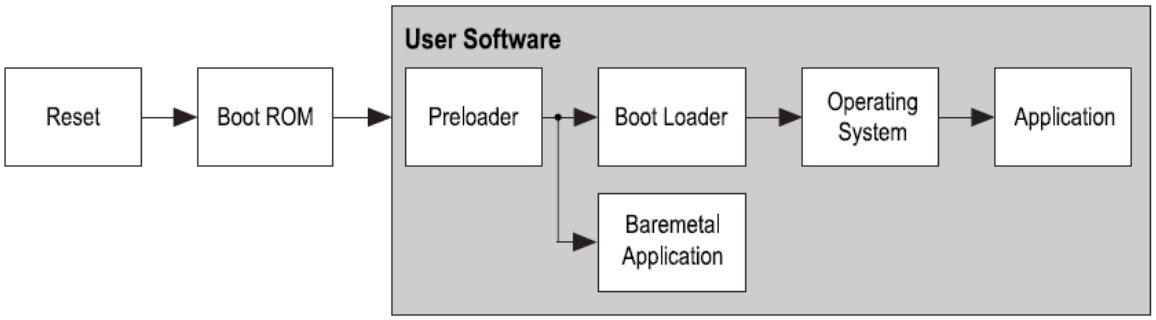

Figura A.6. HPS. Flujos de arranque o booteo.

El *preloader* es una de las etapas de arranque o de booteo más importantes. En realidad, es lo que se denomina como *source* o *fuente* debido a que todas las etapas anteriores no pueden ser modificadas.

En la Figura A.6 *ut supra* se muestran los flujos de arranque o de booteo disponibles del HPS. Las etapas de *reinicio* y de *ROM de arranque* son las únicas partes *fijas* del proceso de booteo o de arranque. Todo lo contenido dentro de las etapas del software de usuario (*software user*) puede ser *personalizado*.

Las etapas de *reinicio*, la *ROM de arranque* y el *preloader* siempre están presentes en el flujo de booteo o de arranque del HPS. Lo que viene después del *preloader* depende del tipo de aplicación que se desee ejecutar.

El HPS puede ejecutar dos tipos de aplicaciones:

- Aplicaciones *bare-metal*.
- Aplicaciones sobre un OS (Linux).

En nuestro caso, como veremos en la sección siguiente del presente informe, utilizaremos ésta última.

3. Uso del Cyclone V. Información general

Es posible utilizar el SoC Cyclone V 5CSEMA5F31C6 en 3 (tres) configuraciones diferentes:

- FPGA-only.
- HPS-only.
- HPS & FPGA.

La configuración que nos interesa a nosotros es la última, es decir, la configuración HPS & FPGA. Para ello, es necesario correr una aplicación sobre un OS (Linux). A continuación, ampliaremos.

3.1. Proceso de booteo del HPS

Ejecutar código sobre un OS Linux tiene varias ventajas. En primer lugar, el kernel libera la CPU1 del reinicio al arrancar por lo que todos los procesadores están disponibles. Además, el kernel se inicializa y hace que la mayoría, por no decir todos, los periféricos

del HPS estén disponibles para su uso por el programador. Esto es posible ya que el kernel de Linux tiene acceso a una gran cantidad de drivers. El código multiproceso es mucho más fácil de escribir ya que el programador tiene acceso a las llamadas de las familias de subprocesos del sistema. Finalmente, el kernel de Linux no se limita a ejecutar programas compilados en C. De hecho, siempre se puede ejecutar código escrito en otro lenguaje de programación siempre y cuando se instale previamente el entorno de ejecución requerido (debe estar disponible para los procesadores ARM).

Sin embargo, ejecutar una aplicación embebida sobre un OS también tiene desventajas. Debido al sistema de memoria virtual establecido por el OS, un programa no puede acceder directamente a los periféricos del HPS a través de sus direcciones asignadas o mapeadas de memoria física. En su lugar, primero se necesita mapear las direcciones físicas de interés en el espacio de direcciones virtuales del programa en ejecución. Sólo entonces será posible acceder a los registros de un periférico. Idealmente, el programador debería escribir un driver de dispositivo para cada componente específico que esté diseñado para tener una interfaz limpia entre el código de usuario y los accesos al dispositivo.

Al final del día, las aplicaciones bare-metal y las aplicaciones que ejecutan código sobre Linux pueden hacer lo mismo. En términos generales, la programación sobre Linux es superior y mucho más fácil en comparación al código bare-metal ya que sus ventajas superan ampliamente sus desventajas.

3.2. Estructura del proyecto

El proceso de desarrollo crea muchos más archivos en comparación con un diseño FPGA-only. Se usará la estructura de carpetas que se observan en la Figura A.7 para organizar el proyecto. Se utilizará *DE1_SoC_demo* como nombre de proyecto.

- El directorio *hw* contiene todos los archivos relacionados con el hardware.
- El directorio *sw* contiene todos los archivos relacionados con el software.
- El directorio *sdcard* contiene todo lo necesario para crear una tarjeta SD válida desde la que el DE1-SoC pueda bootear o arrancar.

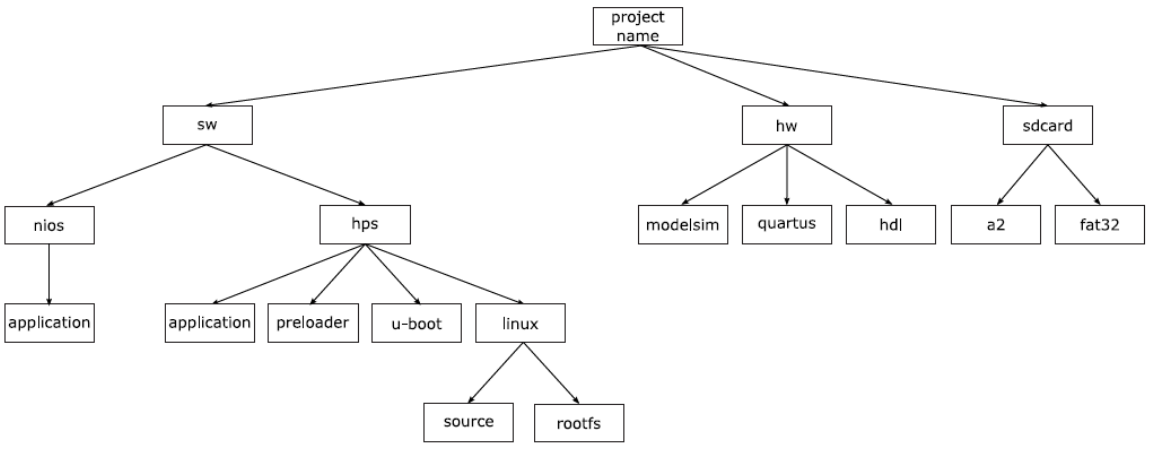

Figura A.7. Estructura del proyecto DE1_SoC_demo.

El diseño completo del tutorial puede encontrarse en *DE1_SoC_demo.zip*[24].

Anexo B: Protocolo ASTERIX CAT240

ASTERIX es un conjunto de protocolos estándar empleado para el intercambio de datos de información de RADAR entre sistemas, propuesto por EUROCONTROL.
Los estándares ASTERIX identifican una colección de tipos de mensajes, llamados categorías o CAT. En nuestro caso, es CAT240, es decir, RADAR Video Transmission, que se utiliza para transferir datos de videos RADAR o, dicho de otra manera, para distribución de video RADAR.
Después de su especificación en el año 2009, el protocolo ASTERIX ha sido adoptado como el estándar de vídeo en red[25][26].

En la Tabla B.1, puede observarse la composición o conformación del mensaje empaquetado en protocolo ASTERIX CAT240[13].

En la imagen *ut infra*, en cambio, puede observarse la correspondencia entre el paquete ASTERIX CAT240 y su representación gráfica[25]. Vemos que el protocolo ASTERIX maneja coordenadas polares (*azimut*).

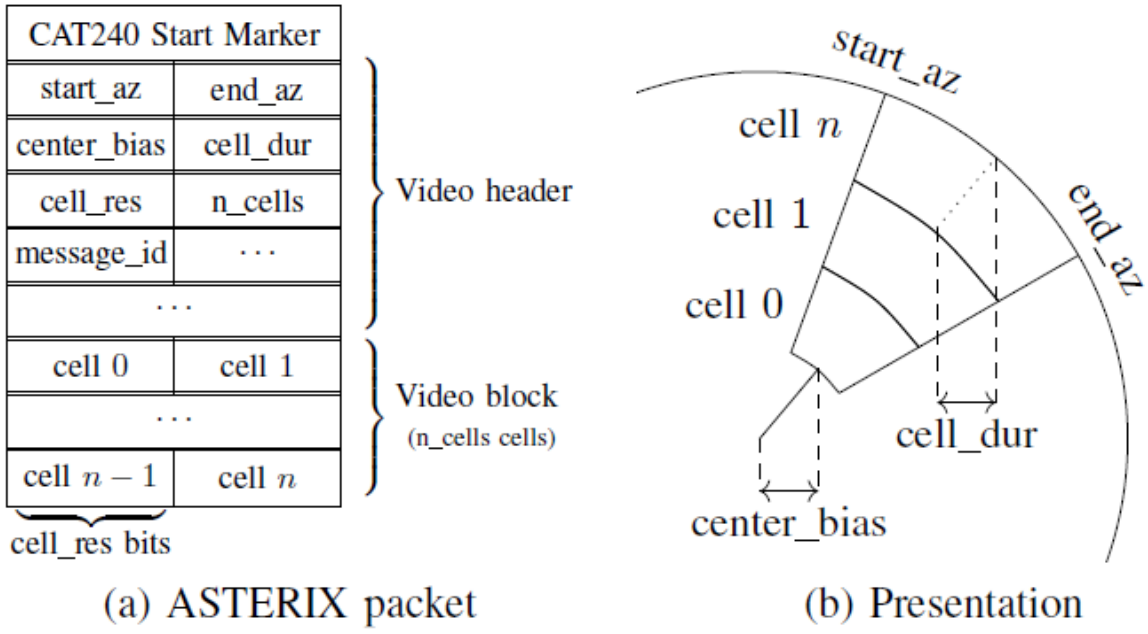

Figura B.1. Correspondencia entre el paquete ASTERIX CAT240 y su representación gráfica.

Field	Length [bytes]	Type of data	Hexadecimal value/Description
ASTERIX category	1	fixed	*f0*
Length	2	variable	Total length in bytes of the current frame, including the CAT and LEN fields.
FSPEC	2	fixed	*e7a0*
Data source identifier	2	variable	Identification of the system from which the data are received. SIC and SAC fields.
Message type	1	fixed	*02* (Video Message).
Video record header	4	variable	Incremental 32 bits number (*MSG_INDEX*).
Start azimuth	2	variable	Start azimuth of the cell group.
End azimuth	2	variable	End azimuth of the cell group.
Starting range	4	variable	Starting range in number of cells.
Cell duration	4	variable	Duration of a video cell in femto seconds.
Compression	1	fixed	*00* (No Compression applied).
Resolution[19]	1	fixed	*04* (High Resolution Coding Length in bits: 8). ---------- *05* (Very High Resolution Coding Length in bits: 16).
Number of video bytes	2	variable	Number of video bytes (without compression).
Number of cells	3	variable	Total number of radar video cells.
Repetition	1	variable	Number of 64 bytes video blocks.
Medium video block	64 x Repetition	variable	Radar video data (cells).

Tabla B.1. Protocolo ASTERIX CAT240. Conformación de la trama o paquete.

Tanto de la Tabla B.1 como de la Figura B.1 puede notarse que el paquete encapsulado en protocolo ASTERIX consiste o está formado por un encabezado (*video header*) y un bloque de datos (*video data block*). El encabezado, el cual es de 32 [bytes] fijos de

[19] *Resolution = 0x04*, ver ensayo 2.4.3.; *Resolution = 0x05*, ver ensayo 2.6.1.

información no útil, referencia la información que posteriormente es representada en un visualizador de video RADAR.

En el caso real, la trama o paquete se arma con un *trigger* o con un *BI*, que son señales enviadas por el radar. En otras palabras, por cada señal de *trigger* o *BI* nueva, se finaliza una trama y se comienza una nueva.

La frecuencia del *trigger* varía dependiendo del tipo de radar.

Por otro lado, para comenzar a graficar, siempre se espera el primer *trigger* luego del *HM*, que también es una señal enviada por el radar. El *HM* marca el final e inicio del barrido, el cual puede estar referenciado al norte (*north-up*), a proa (*head-up*) o al curso actual de la embarcación (*course-up*)[27]. Un detalle a tener en cuenta es que el protocolo ASTERIX define que el video RADAR debe estar referenciado al Norte[13].

En las imágenes observadas a continuación, se observa una representación gráfica para clarificar un poco lo comentado.

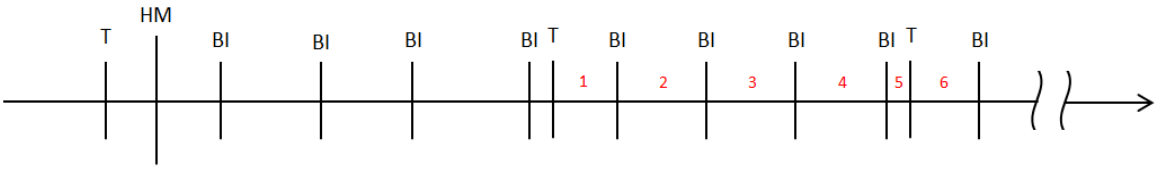

Figura B.2. Señales enviadas por el radar.

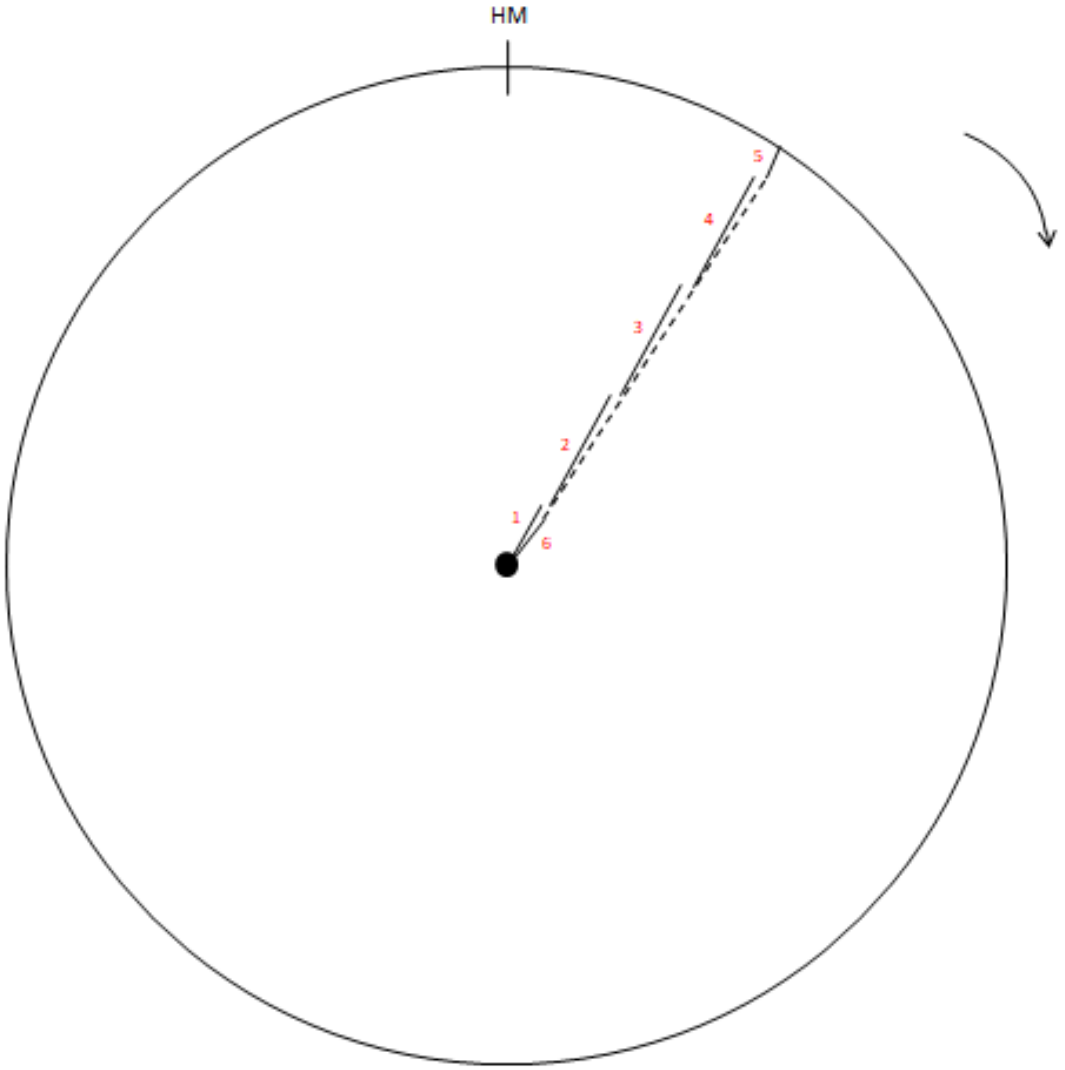

Figura B.3. Representación gráfica de los datos en base a las señales provenientes del radar.

De las imágenes observadas anteriormente, puede decirse que por cada pulso de *trigger* se retorna al origen o centro del radar mientras que por cada pulso de *BI* se concatenan los datos hasta el nuevo pulso de *trigger*, con el cual se vuelve al centro del radar, dando además el *rango* del radar. En pocas palabras, podemos decir que tanto el *trigger* como el *BI* referencian las muestras, las cuales son de tamaño variable debido a que no existe sincronización entre el *trigger* y el *BI*.

Además, por barrido o vuelta completa al radar, se tienen 65536 paquetes cuyo tamaño dependerá del tamaño del paquete o trama empaquetada en protocolo ASTERIX CAT240, el cual es variable. En otras palabras, se divide el barrido en 65536 partes. Por lo tanto, el *azimut* es una palabra de 16 bits ($2^{16} = 65536$). Esto está definido por el protocolo ASTERIX CAT240 empleado[13].

Anexo C: Modelo OSI

El modelo OSI corresponde a un estándar de los protocolos de red cuyo objetivo se basa en interconectar sistemas de distinta procedencia simplificando el intercambio de información. La implementación del presente trabajo se adhiere a los protocolos existentes para establecer conectividad con cualquier dispositivo estándar.

A continuación se describen las diferentes capas que componen el modelo OSI. Cabe destacar que en el desarrollo se implementan sólo las cuatro primeras capas del modelo: capa física, capa de enlace de datos, capa de red y capa de transporte.

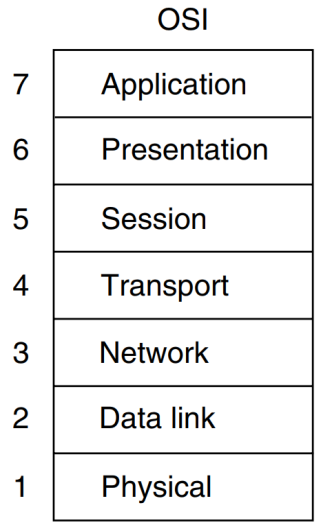

Figura C.1. Capas del modelo OSI.

1. Capa física

La capa física ya está implementada por un PHY externo. Por lo tanto, no se requiere realizar ninguna tarea al respecto con la FPGA. La conexión de la capa física con la MAC es independiente del medio. Para el proyecto es necesario utilizar el protocolo RGMII.

1.1. Capa de enlace de datos: Ethernet II

La figura a continuación muestra los diferentes campos que forman la trama Ethernet II[28][29]. A continuación se detalla cada uno de ellos.

802.3 Ethernet packet and frame structure

Layer	Preamble	Start of frame delimiter	MAC destination	MAC source	802.1Q tag (optional)	Ethertype (Ethernet II) or length (IEEE 802.3)	Payload	Frame check sequence (32-bit CRC)	Interpacket gap
	7 octets	1 octet	6 octets	6 octets	(4 octets)	2 octets	46-1500 octets	4 octets	12 octets
Layer 2 Ethernet frame			← 64–1522 octets →						
Layer 1 Ethernet packet & IPG	← 72–1530 octets →								← 12 octets →

Figura C.2. Campos que conforman la trama Ethernet II.

- **Header:**

- **Preamble:** (7 [bytes]) es una secuencia de bits usada por el medio físico para la estabilización y sincronización. Indica el comienzo de un paquete y permite al emisor y receptor establecer la sincronización. El patrón del preámbulo es el siguiente:

  10101010 10101010 10101010 10101010 10101010 10101010 10101010

- **Start frame delimiter:** (1 [byte]) indica el final del preámbulo y el comienzo de la trama Ethernet. Los bits que integran el campo SFD son: *10101011*. La recepción de la secuencia anterior significa que los datos recibidos a continuación son válidos.

- **Destination MAC address:** (6 [bytes]) contiene la dirección MAC del dispositivo de destino.

- **Source MAC address:** (6 [bytes]) contiene la dirección MAC del dispositivo de origen.

- **Ethertype:** (2 [bytes]) valores menores a 1500 indican que el campo se utiliza para determinar el tamaño del paquete y valores mayores a 1536 indican el protocolo de encapsulamiento (EtherType). En el segundo caso, el largo de la trama se determina por la ubicación del espacio entre paquetes y FCS. El valor *0x0800* (2048 en decimal, por lo tanto, correspondiente a EtherType) indica que se utiliza el protocolo IPv4.

- **Data:** (mayor o igual a 46 [bytes]) incluye información útil (*payload*). En el caso de que la información sea menor a la cantidad mínima permitida (46 [bytes]) se realiza padding. Este consiste en agregar ceros hasta completar la cantidad de bytes mínima permitida.

- **Trailer:**

- **FCS:** (4 [bytes]) corresponde al CRC. Permite la detección de datos corruptos una vez que son recibidos. El valor de FCS se calcula como una función de los campos protegidos de MAC: *dirección de origen* y *destino*, *Ethertype*, *datos* y *padding*. Se utiliza un polinomio específico para realizar la función CRC32[30]. El receptor

debe calcular un nuevo FCS cuando recibe los datos y comparar el FCS recibido con el calculado.

- **Interpack Gap o Interframe Gap**: es un tiempo de espera entre cada paquete de datos, tiene una longitud mínima de 12 [bytes] en cero.

1.2. Capa de red: Trama IPv4

En la figura a continuación se muestra el frame IPv4 y se detallan cada uno de sus campos:

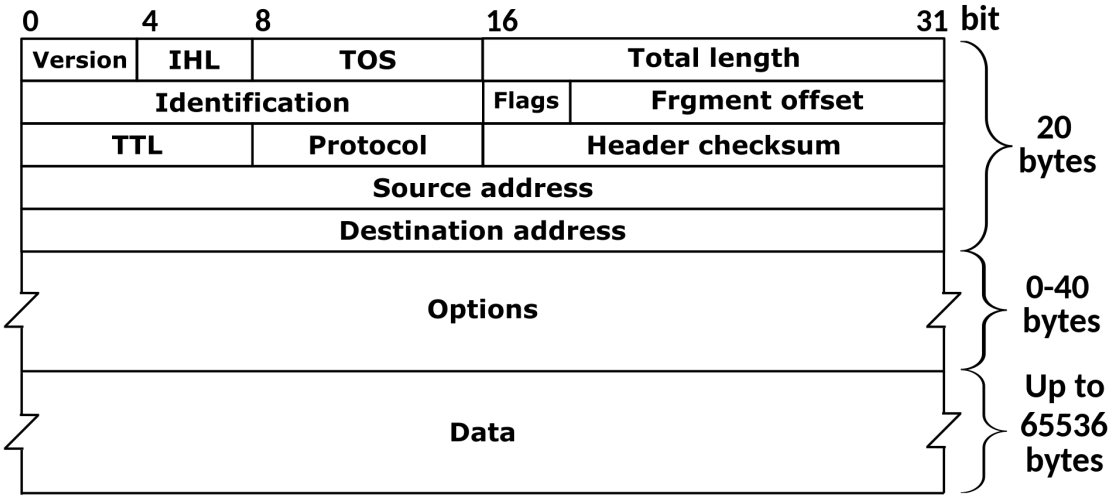

Figura C.3. Campos que conforman la trama IPv4.

- **Versión:** (4 [bits]) para IPv4 es 4.

- **IHL:** (4 [bits]) es la longitud de la cabecera IPv4. Se mide cada 4 [bytes], por lo que una longitud de 5 se refiere a 5 x 4B = 20 [bytes]. El mínimo tamaño será 5 si la cabecera no posee opciones IP. Caso contrario, si posee IP, la mayor longitud podría ser 15.

- **DSCP y ECN:** (1 [byte]) estos campos indican el valor de precedencia, la probabilidad de pérdida de paquetes y el servicio de red utilizado. Los primeros 6 bits para colocar el DSCP y los últimos 2 bits para ECN.

Figura C.4. DSCP y ECN.

DSCP define un CS que corresponde al campo precedencia del antiguo TOS.
Se reemplazó este campo por los actuales DSCP y ECN.
Para tráfico normal: DSCP = 000000 y ECN = 00.

- **Total length:** (2 [bytes]) se refiere a la longitud total del paquete en [bytes] incluyendo tanto la cabecera como el *payload* (*data*).

- **Identification:** (2 [bytes]) es un identificador de grupo de segmentos de un mismo datagrama IP. Es decir, si un paquete es fragmentado durante la transmisión, todos los fragmentos contendrán el mismo número de identificación del paquete IP original al que pertenecen.

- **Flags:** (3 [bits]) se considera el bit más significativo primero:
  - Bit 0: debe ser 0, es un bit reservado.
  - Bit 1: se coloca en 1 si el paquete no debe ser fragmentado. Para la comunicación realizada debe ser 1 (sin fragmentar).
  - Bit 2: MF es 1 si el paquete está fragmentado, sino es 0. Para este caso debe ser 0.

- **Fragment Offset:** indica el lugar del datagrama donde pertenece un fragmento. Se mide en unidades de 8 [bytes]. El primer fragmento tiene un *fragment offset* igual a 0.

- **TTL:** (1 [byte]) en cada *salto*, por ejemplo cuando un mensaje se transmite entre routers, el tiempo de vida del paquete disminuye en una unidad y cuando llega a cero es descartado. El valor de este campo por defecto es 128.

- **Protocol:** (1 [byte]) define qué protocolo se usa en el campo *data*. El valor 17 corresponde a UDP mientras que el valor 6 corresponde a TCP.

- **Header Checksum:** (2 [bytes]) su función es detectar discrepancias por cambios accidentales en la información. Si el checksum calculado en el destino con el recibido difiere el paquete es rechazado. Se calcula teniendo en cuenta los campos de la cabecera de la trama IP. Se halla el valor sumando todos los campos de la cabecera (exceptuando el propio checksum) y realizando el complemento a uno.

- **Source IP Address:** (4 [bytes]) dirección IP del dispositivo de origen.

- **Destination IP Address:** (4 [bytes]) dirección IP del dispositivo de destino.

- **IP Options:** (variable entre 0 y 40 [bytes]) campo para variedad de propósitos que no es frecuentemente usado.

- **Padding:** (variable) garantiza que la cabecera IP finalice con una longitud múltiplo de 4 [bytes], agregando ceros de ser necesario.

1.3. Capa de transporte

1.3.1. Trama UDP

- **Source port:** (2 [bytes]) puerto de origen. Cada proceso tiene un puerto asignado. El puerto de origen puede ser cero si no se espera respuesta.

- **Destination port:** (2 [bytes]) puerto de destino.

- **Length:** (2 [bytes]) es la longitud en bytes del paquete incluyendo el *header* y los *datos*. El valor mínimo es 8 [bytes] (longitud del header).

- **Checksum:** (2 [bytes]) corresponde a la cabecera UDP, los datos, junto con los campos *length, protocol, source and destination address* de la trama IP. En IPv4 este campo es opcional, por lo tanto, puede completarse con ceros.

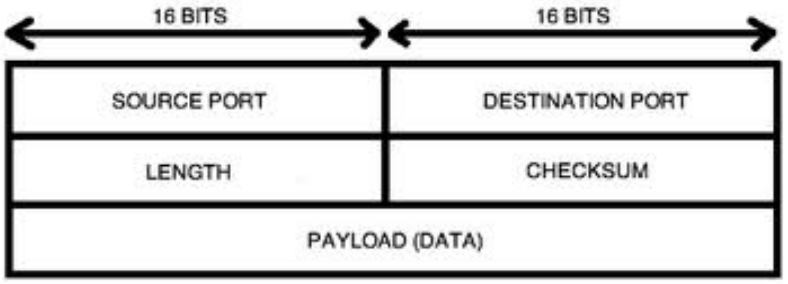

Figura C.5. Campos que conforman la trama UDP.

1.3.1.1. Tamaño máximo del *payload* en del mensaje UDP

Existe un límite máximo en la longitud de los mensajes que pueden transmitirse por Ethernet v2 correspondiente al *MTU*, el cual, para Ethernet es de 1500 [bytes] aunque el campo *length* del protocolo UDP indique que puede enviarse un *payload* de 65.535 [bytes], Ethernet es el limitante.

La cantidad máxima de bytes del *payload* que pueden transmitirse por Ethernet utilizando los protocolos IPv4 y UDP puede definirse por la siguiente ecuación[31]:

$$Payload\ máx.\ Ethernet = 1500[bytes]\ (MTU) - 20[bytes]\ (IP\ Header) - 8[bytes]\ (UDP\ Header)$$

$$Payload\ máx.\ Ethernet = 1472[bytes]$$

Por lo tanto, el tamaño máximo del *payload* del mensaje UDP que puede transmitirse a través de Ethernet sin realizar fragmentación es de 1472 [bytes].

Agradecimientos

A la familia y amigos cercanos durante todo el transcurso de la carrera y el presente proyecto final, por su apoyo incondicional, su cariño y su infinita paciencia para con nosotros, aún en tiempos turbulentos.

Al Ing. Christian L. Galasso, investigador del SIAG y docente de la Universidad Tecnológica Nacional Facultad Regional Bahía Blanca, por confiar en nosotros y darnos la oportunidad de pertenecer al presente proyecto, además de su apoteósica paciencia y predisposición para resolver cada una de las inquietudes y conflictos que se nos fueron presentando a lo largo del tiempo.

Al abnegado **Ing. Diego Martínez**, personal civil del SIAG, por su inestimable colaboración, tanto al introducirnos con la implementación que se llevó a cabo en este proyecto como en el desarrollo del mismo.

Al abnegado **Dr. Ricardo Cayssials** y al alumno **Mariano Valdez**, por desarrollar la parte de diseño de hardware del proyecto (captura de muestras de video RADAR y posterior empaquetado en protocolo ASTERIX CAT240 de las mismas) y su inestimable predisposición a la hora de depurar los problemas que iban surgiendo en los ensayos, siempre en forma puntual y con la mejor voluntad.

A los docentes de la Universidad Tecnológica Nacional Facultad Regional Bahía Blanca, por los conocimientos y experiencia compartidos para con nosotros, siempre con el objetivo de formarnos como profesionales de la manera más óptima.

A todas aquellas personas que fueron parte, directa o indirectamente, de nuestro objetivo y que por motivos de desafortunado olvido no hemos mencionado.

A todos… **¡Gracias!**

4. Referencias

[1] https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/SoC-FPGA%20Design%20Guide_EPFL.pdf
[2] https://github.com/BPI-SINOVOIP/BPI-M4-bsp/issues/4
[3] https://www.sysadmit.com/2020/03/linux-habilitar-ssh-root.html
[4] https://www.ochobitshacenunbyte.com/2015/04/30/habilitar-usuario-root-conexiones-ssh-debian/
[5] https://infoaleph.wordpress.com/2008/05/20/como-cambiar-el-idioma-del-ambiente-grafico-de-gnulinux-debian-despues-de-la-instalacion/
[6] https://ciksiti.com/es/chapters/8566-how-to-change-debian-desktop-environment
[7] https://users.cs.jmu.edu/bernstdh/web/common/lectures/summary_unix_udp.php
[8] https://man7.org/linux/man-pages/man2/socket.2.html
[9] https://man7.org/linux/man-pages/man2/setsockopt.2.html
[10] https://man7.org/linux/man-pages/man2/bind.2.html
[11] https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C/Manejo_de_archivos
[12] https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C/Estructuras_y_Uniones#Uniones
[13] https://www.eurocontrol.int/sites/default/files/content/documents/nm/asterix/20150513-asterix-cat240-v1.3.pdf
[14] https://gcc.gnu.org/onlinedocs/gcc-3.3/gcc/Type-Attributes.html
[15] https://linuxhint.com/gettimeofday_c_language/
[16] https://www.tutorialspoint.com/c_standard_library/c_function_memcpy.htm
[17] https://es.wikipedia.org/wiki/Endianness
[18] https://man7.org/linux/man-pages/man2/sendto.2.html
[19] https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/HPS_peripherials/FPGA_addr_index.html
[20] https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/HPS_FPGA/DMA/HPS_to_FPGA/DMA_1.c
[21] https://www.terasic.com.tw/en/
[22] https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=165&No=836#contents
[23] https://www.intel.com/content/www/us/en/homepage.html
[24] https://github.com/sahandKashani/SoC-FPGA-Design-Guide/blob/master/DE1_SoC_demo.zip
[25] Giacomo Longo, Enrico Russo, Alessandro Armando and Alessio Merlo (2022). *Attacking (and defending) the Maritime Radar System*.
[26] Dr. David G. Johnson and Mr. Richard Warren. *Using ASTERIX CAT-240 for Radar Video Distribution - Practical Considerations from Deployed Applications*.
[27] Alan Bole, Alan Wall and Andy Norris. (2014). *RADAR AND ARPA MANUAL. Radar, AIS and Target Tracking for Marine Radar Users* (Third Edition). UK: Elsevier.
[28] https://es.wikipedia.org/wiki/Ethernet
[29] https://www.geeksforgeeks.org/ethernet-frame-format/
[30] https://docs.xilinx.com/v/u/en-US/xapp209
[31] Processing efficiency : "The most reliable and efficient UDP packet size", Stack Overflow
[32] https://netwgeeks.com/fragmentacion-ipv4/

[33] https://www.enmimaquinafunciona.com/pregunta/793/pueden-paquetes-tcp-y-udp-ser-divididas-en-partes