SISTEMA DE IDENTIFICACIÓN DE MATRÍCULAS

(SIM)

SISTEMAS ELECTRÓNICOS INTEGRADOS

Máster Universitario en Ingeniería de Telecomunicación

Profesor:

Javier Díaz Alonso

Autores:

Antonio Manuel Hervás Ramírez

Ángel Gómez Hurtado

Javier Linzoain Pedraza

Contenido

CONTEXTO Y MOTIVACIÓN

En la era digital actual, la integración de sistemas electrónicos en aplicaciones cotidianas se ha convertido en una práctica común y cada vez es más frecuente encontrar sistemas embebidos en distintas aplicaciones. En este contexto, el reconocimiento de matrículas automático ha emergido como una solución innovadora y de gran utilidad en una variedad de escenarios, desde la gestión del tráfico hasta la seguridad pública y privada.

El presente proyecto se enmarca en este contexto de innovación tecnológica, proponiendo una solución práctica y eficiente para el reconocimiento de matrículas mediante el uso de una Raspberry Pi, una cámara y un sensor, combinados con el poder del lenguaje de programación Python y el sistema operativo Linux. Esta combinación de hardware y software permite desarrollar un sistema robusto y versátil capaz de capturar imágenes de matrículas en tiempo real, procesarlas y extraer la información necesaria de manera automatizada. Si bien es cierto que existen tutoriales e implementaciones similares, el hecho de integrar un sensor junto con la Raspberry y diseñar un contenedor en 3D específico para el dispositivo es lo que lo hacen distinto a las aplicaciones ya desarrolladas.

La utilización de la Raspberry Pi como plataforma base ofrece una serie de ventajas, incluyendo su tamaño compacto, bajo costo y flexibilidad para la integración de diversos componentes adicionales. Además, el uso de Python como lenguaje de programación principal y Linux como sistema operativo proporciona una base sólida y bien establecida para el desarrollo de aplicaciones de visión por computadora y procesamiento de imágenes.

OBJETIVOS

Iniciales:

Con el fin de realizar el procesado de la imagen y la identificación de matrículas de manera óptima, se impusieron los siguientes objetivos a cumplir, teniendo cada uno de ellos una complejidad mayor.

1. Desarrollo del código de detección de imagen mediante pytesseract y opencv en un PC.
2. Configuración y puesta en funcionamiento de la RaspBerry Pi. Detección de matrículas mediante la cámara a partir del código desarrollado en el PC. (Mínimo entregable)
3. Migración a la PYNQ-Z2.
4. Programación de la lógica programable presente en la FPGA Zynq-7020 de la PYNQ-Z2 para realizar el procesamiento imagen lo más rápido posible.

Debido a diferentes obstáculos y retos encontrados durante el desarrollo del proyecto, se ha optado por perfeccionar al máximo el mínimo entregable, apartando un poco la migración a la PYNQ-Z2.

Una modificación respecto a los objetivos iniciales es el cambio de plataforma en la que se iba a realizar la migración. La documentación sobre la PYNQ-Z2 es muy escasa y no se encontró ninguna bibliografía en la que se diseñase un sistema operativo Linux para integrar en la PYNQ-Z2. Ya que se disponía también de una placa de desarrollo ZYBO, se optó por cambiar a esta nueva herramienta.

En la ZYBO, aunque no se consiguió la migración del sistema de detección, se busco la integración de un sistema Linux en la placa.

MATERIALES Y RECURSOS

Para la realización del proyecto, se utilizaron los siguientes elementos:

Figura 1. Material utilizado

- Cámara Raspberry Pi
- Raspberry Pi 4B
- 4 cables
- Tarjeta Micro SD
- Sensor de ultrasonidos HC-SR04
- Cable Ethernet
- Contenedor del sistema de resina

TAREAS Y PLANIFICACIÓN

En el presente apartado se desglosan las distintas tareas que han tomado lugar en el desarrollo del proyecto, así como su planificación.

Tarea 1: Código en PC

En esta tarea se desarrolla el código en PC de forma que se detecten matrículas a partir de imágenes obtenidas de internet con las librerías OpenCV y PyTesseract de Python.

Tarea 2: Configuración Raspberry Pi

Seleccionar y grabar la versión del sistema operativo Raspbian adecuada en la Raspberry Pi que funcione con las versiones de las librerías y paquetes utilizadas para el código desarrollado en el PC. Esto es muy importante, porque dependiendo de la versión Raspbian a instalar, librerías esenciales como "numpy", no aparecen actualizadas o no se pueden actualizar.

Tarea 3: Implementación del reconocimiento de texto en tiempo real en la Raspberry Pi

Mediante la "PiCamera", capturar imágenes en tiempo real, procesarlas mediante el código de Python, y obtener un resultado coherente. Para ello, el código de Python llama mediante subprocess a un script de bash que lanza la cámara y realiza la captura.

Tarea 4: Verificación y test

Se realiza un testeo del sistema de captura para obtener una probabilidad de acierto.

Tarea 5: Incorporación de sensor ultrasonido HC-SR04

Se incorpora un sensor HC-SR04 para establecer a qué distancia se debería lanzar el código de reconocimiento de matrículas, simulando un escenario real. Para ello, es necesario acondicionar el entorno de medida con las condiciones de distancia, iluminación y altura adecuadas.

Tarea 6: Diseño e impresión contenedor 3D

Mediante AutoCad, se diseña un contenedor en 3D que alojará el sistema compuesto de la Raspberry Pi, la protoboard y el sensor

Tarea 7: Implementar Linux en la Zybo

Por último, se diseña un sistema operativo Linux, con sus correspondientes cargadores de arranque (FSBL, U-Boot), kernel de Linux, bitstream y Device Tree en la Zybo. Se profundizará más en esta tarea en el apartado "Diseño de sistema Linux para la Zybo".

En la siguiente sección, se mostrará el procedimiento seguido para la detección de matrículas en la raspberry.

ALGORITMO DE DETECCIÓN

A continuación, se describe el algoritmo planteado para la identificación y reconocimiento de matrículas, aplicable tanto a la Raspberry Pi como a la Zybo.

- **"Detección_con_sensor.py"**: es un script que lanza la detección de objetos a menos de 50 cm, que es la distancia estimada a la que la cámara es capaz de tomar la mejor foto para el postprocesamiento. Este script ejecuta un while True infinito que no parará hasta que se cancele la ejecución mediante atajos de teclado.
- **"Detecta_final.py":** es un script de Python que es lanzado por "Detección_con_sensor.py" mediante el uso de la biblioteca "subprocess" cuando detecta el objeto cercano. Al no ser posible acceder a la cámara directamente desde Python por medio de ninguna librería, se planteó lanzar la captura de la imagen mediante un código Bash. "Detecta_final.py" lanza el script de bash mediante la biblioteca "subprocess". En este código es donde se realiza el procesamiento de la imagen y se extrae la matrícula detectada. La matrícula la almacena en un fichero de texto que actúa como base de datos.
- **"Captura_imagen.sh"**: es un script de Bash que abre la interfaz de la cámara y toma una captura. Esta captura la almacena en un directorio, a partir del cuál "Detecta_final.py" accederá para procesar la imagen. Todas las capturas se guardan con el mismo nombre, ya que es innecesario almacenar cada una de las capturas que se están tomando. De esta manera cada captura se sobreescribe sobre la anterior.
- **"Directorio_capturas"**: es un directorio en el que se almacena la foto tomada por el script de Bash y al que se accede desde "Detecta_final.py" para obtener la captura realizada.
- **"Base_de_datos.txt"**: es un archivo de texto que actúa como base de datos y contiene la fecha y hora en la que se detectó una matrícula y la matrícula asociada.

Todos estos códigos se encuentran en el repositorio del proyecto en GitHub: https://github.com/Linzpe/SIM-Sistema-de-Identificacion-de-Matriculas

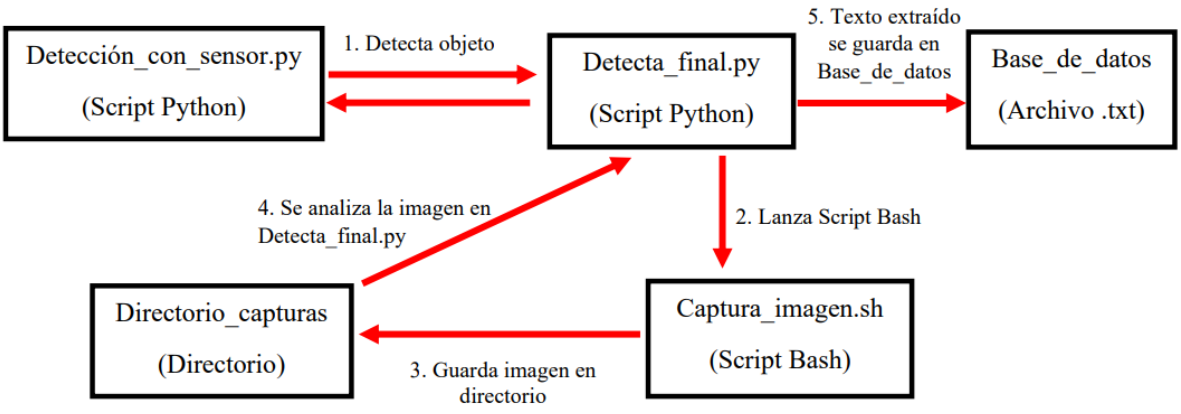

Figura 2. Esquema de bloques del algoritmo de detección.

PROCESAMIENTO DE LA IMAGEN

Es interesante mostrar cómo se ha realizado el procesamiento de la imagen para la obtención de los contornos y la extracción del texto de la matrícula.

En primer lugar, introducimos la captura de la imagen que ha captado la cámara y hacemos un ‘resize’ que lo que hace es redimensionar la imagen obteniendo una resolución de 620x480.

```
img = cv2.imread('matricula.jpg',cv2.IMREAD_COLOR)

img = cv2.resize(img, (620,480) )
```

Figura 3. código para leer y redimensionar la imagen.

Figura 4. Imagen redimensionada.

El segundo paso es hacer una conversión a escala de grises y aplicamos un filtro bilateral para reducir el ruido de la imagen.

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) #convert to grey scale
gray = cv2.bilateralFilter(gray, 11, 17, 17) #Blur to reduce noise
```

Figura 5. Conversión y reducción de ruido

Figura 6. Imagen en escala de grises y con reduccion de ruido.

Con esta parte del código encontramos los contornos de la imagen, seleccionando los diez más grandes e inicializando una variable que se va a encargar de almacenar la placa de la matrícula.

```
edged = cv2.Canny(gray, 30, 200) #Perform Edge detection

# find contours in the edged image, keep only the largest
# ones, and initialize our screen contour
cnts = cv2.findContours(edged.copy(), cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
cnts = imutils.grab_contours(cnts)
cnts = sorted(cnts, key = cv2.contourArea, reverse = True)[:10]
screenCnt = None
```

Figura 7. Elección de 10 contornos.

Figura 8. Contornos seleccionados.

En esta siguiente parte hacemos una iteración de las partes que ha seleccionado y se queda con la que más se ajusta al rectángulo que buscamos.

```
for c in cnts:
    # approximate the contour
    peri = cv2.arcLength(c, True)
    approx = cv2.approxPolyDP(c, 0.018 * peri, True)

    # if our approximated contour has four points, then
    # we can assume that we have found our screen
    if len(approx) == 4:
        screenCnt = approx
        break
```

Figura 9. Elección del rectángulo

Figura 10. Elección de la matrícula.

La siguiente parte nos hace un bucle que nos dice si no hay contorno o si hay contorno, nos crea una máscara de la placa y la recorta guardándola en la variable 'cropped' en tonos grises.

```
if screenCnt is None:
    detected = 0
    print =("No contour detected")
else:
    detected = 1

if detected == 1:
    cv2.drawContours(img, [screenCnt], -1, (0, 255, 0), 3)

# Masking the part other than the number plate
mask = np.zeros(gray.shape,np.uint8)
new_image = cv2.drawContours(mask,[screenCnt],0,255,-1,)
new_image = cv2.bitwise_and(img,img,mask=mask)

# Now crop
(x, y) = np.where(mask == 255)
(topx, topy) = (np.min(x), np.min(y))
(bottomx, bottomy) = (np.max(x), np.max(y))
Cropped = gray[topx:bottomx+1, topy:bottomy+1]
```

Figura 11. Indicación de si hay contorno.

En esta última parte lee los números y letras que encuentra en el contorno de la placa y nos lo va a mostrar por pantalla. La parte que hace la búsqueda de los caractereres es la biblioteca tesseract-OCR. La configuración –psm 11 nos indica que la imagen solamente contiene un bloque de caracteres de texto y números.

```
#Read the number plate
text = pytesseract.image_to_string(Cropped, config='--psm 11')
print("Detected Number is:",text)

cv2.imshow('image',img)
cv2.imshow('Cropped',Cropped)
```

Figura 12. Muestra por pantalla matrícula.

PRESUPUESTO

En esta parte vamos a mostrar una tabla con los siguientes elementos utilizados en el proyecto y el precio de cada uno:

OBJETOS UTILIZADOS	PRECIO
RASPBERRY PI 4 MODEL B	59,99 €
SENSOR HC-SR4	3,49 €
CÁMARA RASPBERRY PI 4	5,84 €
RESISTENCIA (2k y 1.2k)	2,00 €
PLACA PROTOBOARD	3 €
CABLES	2 €
CABLE ETHERNET DE 3M	7,99 €
CAJA 3D (300 ml)	69,00 €
TOTAL	153,31 €

Figura 13. Componentes y precio.

También hay que tener en cuenta los recursos humanos utilizados en el proyecto contabilizando que cada hora se paga a 30€ según la ITU:

RECURSOS HUMANOS UTILIZADOS	HORAS	PRECIO
INGENIERO 1	30	900,00 €
INGENIERO 2	30	900,00 €
INGENIERO 3	30	900,00 €
TOTAL		2.700,00 €

Figura 14. Presupuesto recursos humanos.

DISEÑO CAJA 3D

Para cumplir con el mínimo entregable de la manera más satisfactoria posible, se decidió diseñar e imprimir en 3D con una impresora de estereolitografía un contenedor con las aperturas necesarias para el sensor, la cámara, el cable Ethernet y la alimentación. La impresora utilizada fue la FormLabs 3+:

Figura 15. Impresora 3D

A continuación se muestran tanto el boceto realizado con las medidas tomadas, como el diseño en Autocad en 3D, como el resultado obtenido. No se proporcionan los planos de secciones y cotas de las vistas “planta, alzado y perfil” por la falta de tiempo para aprender dicho método en AutoCad.

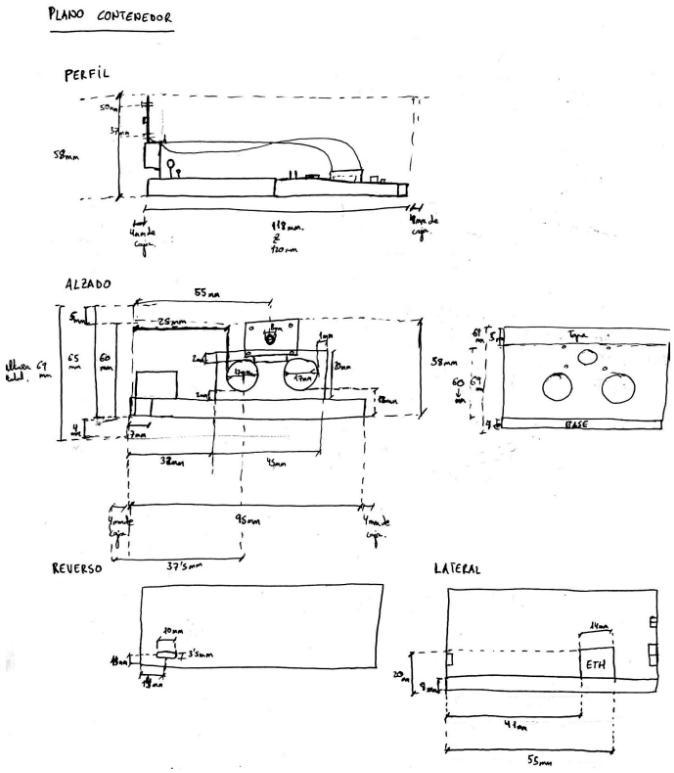

Figura 16. Boceto

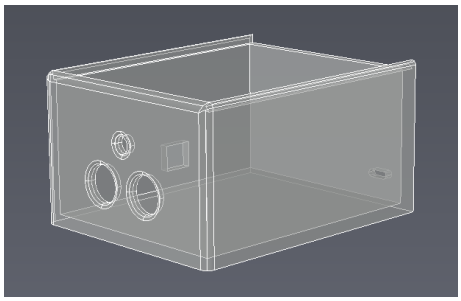

Figura 17. Diseño en AutoCad

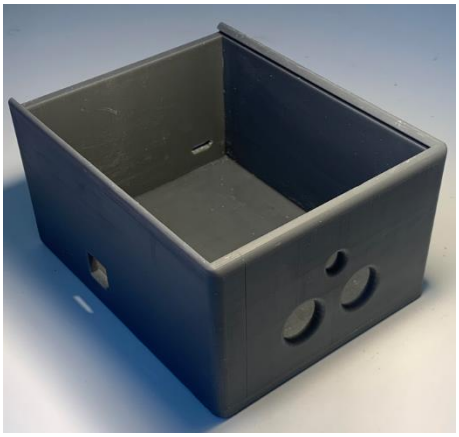

Figura 18. Resultado obtenido

MIGRACIÓN A ZYBO

Como se comentó en la sección "Objetivos", debido a la poca documentación encontrada acerca de la PYNQ-Z2 y haber sido diseñada principalmente para trabajar con la imagen de Linux que ya tiene cargada por defecto la tarjeta Micro SD que trae incorporada, se ha optado por migrar a la Zybo.

La Zybo es una placa de desarrollo diseñada por Digilent Inc. Se destaca por ser una plataforma versátil que combina una FPGA con un procesador ARM en un solo dispositivo, también conocido como SoC. En particular, utiliza un SoC de la serie Zynq-7000 de Xilinx.

Para lograr implementar este proyecto en la Zybo, es necesario acondicionar la herramienta para que sea capaz de ejecutar los scripts previamente presentados. Para ello, se debe diseñar un sistema operativo Linux que permita instalar las dependencias necesarias para la implementación de los paquetes necesarios de Python. Por otro lado, habría que averiguar qué driver y qué tipo de cámara sería la conveniente utilizar en la Zybo.

Construir SO Linux para la Zybo

Para construir un sistema operativo Linux que funcione en la placa de desarrollo escogida, es necesario customizarlo. Para ello, se sigue el siguiente esquema donde:

- **BSD**: Block System Design, proporciona una base funcional desde la cual los desarrolladores pueden empezar a construir y personalizar su propio sistema.
- **Bitstream**: es la configuración física de la FPGA que implementa la lógica definida en el Block Design.
- **FSBL**: First Stage Boot Loader, es un programa que se carga en la memoria del sistema durante su inicialización. Su función es inicializar el HW, configurar los periféricos y cargar el U-Boot.
- **U-Boot**: es un gestor de arranque universal utilizado en sistemas embebidos. Se compila para la arquitectura específica de la plataforma y se configura para cargar el Kernel de Linux desde el sistema de archivos.
- **Buildroot**: se utiliza para compilar el Kernel de Linux junto con los controladores de dispositivos necesarios. Se utiliza el Device Tree.
- **Device Tree**: es una descripción del HW de la plataforma que proporciona información al Kernel de Linux sobre la configuración del HW específica.
- **FAT32**: (File Allocation Table) es un sistema de archivos desarrollado por Microsoft, tiene una compatibilidad excelente con dispositivos externos y con SOs como Linux, Windows o macOS.
- **EXT4**: (Fourth Extended Filesystem) es un sistema de archivos de tipo journaling (registro de transacciones) utilizado principalmente en sistemas operativos basados en Linux.
- **Buildroot**: Buildroot es una herramienta de código abierto utilizada para facilitar la creación de sistemas Linux embebidos personalizados. Permite a los desarrolladores crear imágenes de sistemas embebidos completos.

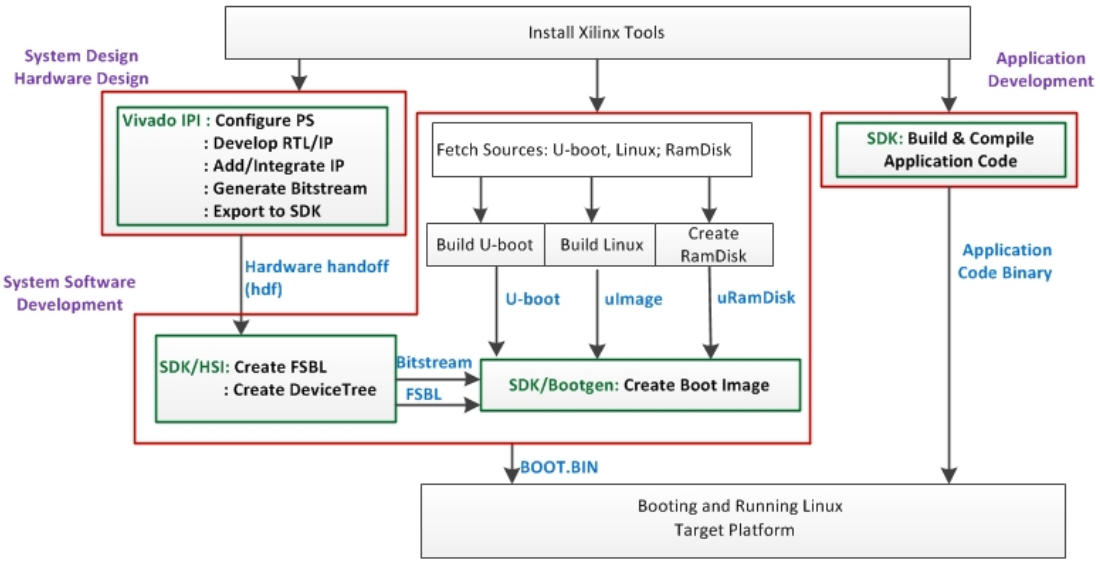

Figura 19. Esquema de customización del sistema operativo

Tras la generación de todos los archivos necesarios para la generación de un sistema operativo customizado para la plataforma el siguiente paso es formatear una tarjeta SD para incorporar el BOOT.bin (bootloader), el Devicetree.dtb, el kernel de Linux (uImage) y la distribución, si la hubiese. En el caso de la figura siguiente, se considera una distribución Ubuntu-Linaro.

Para el formateo de la SD, se puede utilizar GParted, o la herramienta de Ubuntu fdisk.

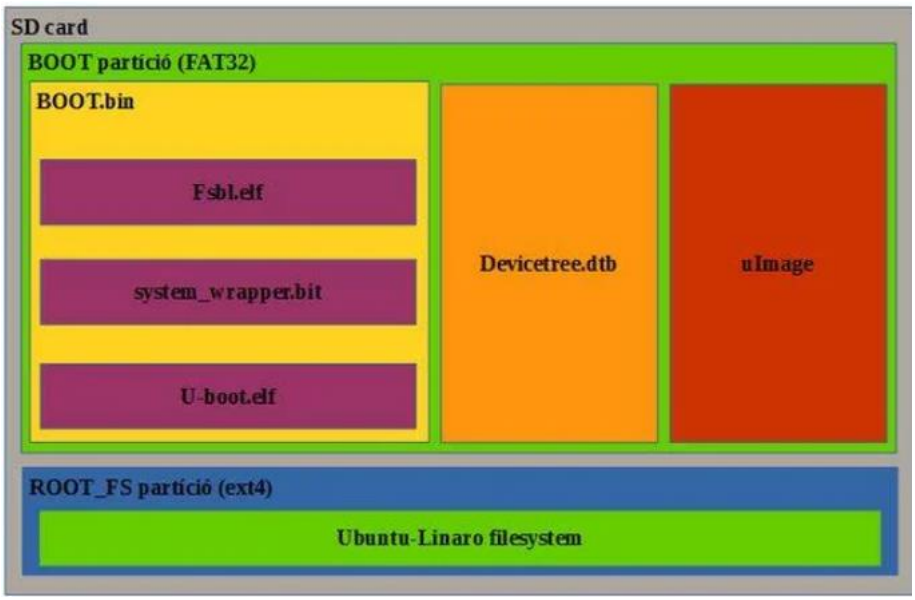

Figura 20. Partición tarjeta SD

OBSTÁCULOS ENFRENTADOS

En esta sección se presentan los obstáculos encontrados y enfrentados a lo largo de la realización del proyecto, siendo la gran parte de ellos superados.

- En el proceso de traspasar el código que se implementó en el PC que detectaba matriculas correctamente a la raspberry hubo inconvenientes. Esto se debió a las librerías que se podían descargar en la raspberry. No todas las que se pueden descargar en el PC las soporta, por ello, se tuvo que adaptar el código que teníamos para que fuesen compatibles tanto librerías como dependencias en la raspberry.
- Otra cosa que nos retrasó fue el no saber que en la raspberry, los pines no corresponden a los pines GPIO. Esto nos pasó por inexperiencia usando la raspberry.
- Con respecto a la raspberry, los mayores problemas surgieron el día de grabación del funcionamiento de nuestra solución SIM. Entre el día que se comprobó que todo funcionase armónicamente (13/02) y el dia de la grabación (15/02) transcurrieron dos días.
    1. En estos dos días, hubo un cambio en las URLs de los repositorios de la Raspberry. Esto hizo que el día de la grabación no funcionase inicialmente la cámara. Una vez se consiguió arreglar este problema mediante la ejecución de los comandos “apt-get update”, “apt-get upgrade”, surgió un nuevo inconveniente.
    2. No era posible conectarse a la Raspberry mediante RealVNC porque al reiniciar la misma para que los cambios de los comandos tuviesen efecto, cambió la resolución de la pantalla. Es decir, ya no teníamos acceso a una GUI a través del ordenador. Esto lo conseguimos arreglar descubriendo que podíamos conectarnos a través de RealVNC si también estaba conectada por HDMI a un monitor, ya que este es el que fijaba la resolución de la pantalla. Cuando ya conseguimos arreglar el problema de la resolución ajustándola a 1720x800, el script “Detección_con_sensor.py” no se ejecutaba.
    3. Se obtuvo un error que alertaba de que no se estaba ejecutando el código en una Raspberry Pi al intentar usar funciones de la biblioteca RPi.GPIO. Tras buscar información y conseguir arreglarlo, nos saltó el error de que solo nos dejaba acceder a los pines GPIO si éramos ‘root’, y para conseguir esto también hubo que hacer una búsqueda exhaustiva por foros de internet. Al final, para hacer el video final se tardó 7 horas, en vez de 20 minutos que era lo que se tenía programado.

- A la hora de crear un proyecto en Vivado, es necesario asegurarse que ninguno de los bloques IP se encuentren en un directorio con más caracteres que los limitados. De esta manera, se obtuvieron para 3 proyectos descargados de GitHub errores en la creación del BSD ya que Vivado no era capaz de encontrar los IP blocks en la ruta con máxima longitud especificada. Cuando se alojó el proyecto en C:/, ya funcionó.

- Las distintas versiones de Vivado, por su parte, al actualizar la arquitectura, producían cambios en el fichero de restricciones .xdc y planteaban errores.
- Una gran dificultad fue encontrar repositorios que alojasen material específico para la Zybo, y no para la Zybo Z7, ya que Digilent declara la plataforma Zybo como obsoleta.

CONCLUSIONES

Atendiendo al trabajo realizado y presentado en este documento, se puede concluir que se han satisfazido completamente los requisitos mínimos, presentando un producto mínimo entregable, portátil y vendible con un funcionamiento adecuado y concorde a lo especificado en los objetivos.

No obstante, es cierto que no se han podido satisfacer los objetivos de la migración del sistema de identificación a la plataforma Zybo.

BIBLIOGRAFÍA

https://www.instructables.com/Booting-Linux-on-the-ZYBO/ç

https://www.instructables.com/Embedded-Linux-Tutorial-Zybo/

https://github.com/Digilent/u-boot-digilent/tree/master/board/xilinx/zynq/zynq-zybo

https://github.com/Digilent/digilent-xdc/blob/master/Zybo-Master.xdc

https://digilent.com/reference/programmable-logic/zybo/start

https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842369/Build+Linux+for+Zynq-7000+AP+SoC+using+Buildroot?f=print

https://buildroot.org/downloads/manual/manual.html

https://www.amd.com/en/search/site-search.html#q=Linux%20with%20buildroot