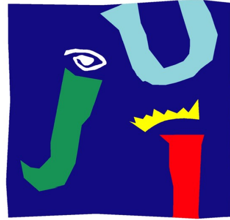


GRADO EN INGENIERÍA INFORMÁTICA

TRABAJO FINAL DE GRADO

Desarrollo con Yocto de un Sistema Operativo Linux Personalizado

Autor:
Carlos MOLLÓN BOU

Supervisor:
Ángel Iván CASTELL ROVIRA

Tutor académico:
María Asunción CASTAÑO ÁLVAREZ

Fecha de lectura: 19 de julio 2018
Curso académico: 2017/2018

Agradecimientos

A mi familia, por apoyarme en todos los momentos y decisiones de mi vida.

A mi pareja, por aguantarme durante tantas horas de dedicación exclusiva al estudio.

A mis amigos, por estar conmigo cuando lo he necesitado.

A la empresa Nayar Systems, por el buen trato y acogida durante la realización de las prácticas.

A Iván Castell Rovira, supervisor en prácticas, por transmitirme todos los conocimientos posibles y la buena relación conmigo.

A María Asunción Castaño Álvarez, tutora de prácticas, por guiarme en la redacción de esta memoria.

// Resumen

Esta memoria describe el proceso seguido para crear una imagen del sistema operativo Linux personalizada que se utilizará en sistemas embebidos, que será fácilmente exportable a varias plataformas. El trabajo se enmarca dentro del proyecto Advertisim de la empresa Nayar Systems.

La imagen se genera con Yocto siguiendo una serie de pasos explicados con detalle en la memoria. Crear dicha imagen es una tarea laboriosa y costosa, pero a largo plazo es un aspecto positivo para la empresa ya que evitará tener que desarrollar un nuevo sistema operativo para cada placa embebida nueva que reciban y también supondrá mayor facilidad en el mantenimiento de los sistemas operativos utilizados.

// Palabras clave

Yocto, Linux, imagen de sistema operativo personalizado, sistema embebido, OpenEmbedded core.

Summary

This paper describes the process to create a customized Linux operating system image for embedded systems, which is portable to different architectures. The work is part of the project Advertisim carried out by the company Nayar System.

The image is generated using Yocto and following a set of steps that are detailed in the report. The creation of this image is a laborious task, but it is a long-term asset for the company, taking into account that it will avoid having to develop a new system for each new embedded board. In addition, it also yields an easier maintienance of the operating systems.

Keywords

Yocto, Linux, custom operating system image, embedded system, OpenEmbedded core.

Índice

Capítulo 1 Introducción....15
1.1 Contexto y motivación del proyecto....15
1.2 Objetivos del proyecto....16
1.3 Estructura de la memoria....17
Capítulo 2 Descripción del proyecto....19
2.1 Características del proyecto....19
2.2 Punto de partida del proyecto....20
2.3 Ventajas del proyecto....20
2.4 Descripción del hardware utilizado....21
2.5 Descripción del software utilizado....23
2.6 Estado del arte....34
Capítulo 3 Planificación del proyecto....37
3.1 Planificación inicial....37
3.2 Planificación final....39
3.3 Diferencias entre la planificación inicial y la final....42
3.4 Costes del proyecto....43
Capítulo 4 Análisis, requisitos y diseño del sistema....45
4.1 Análisis del sistema....45
4.2 Requisitos del sistema....45
4.3 Diseño del sistema....46
Capítulo 5 Desarrollo....47
5.1 Documentación y estudio....47
5.2 Instalación de Yocto....48
5.3 Añadir las capas necesarias como submódulos....48
5.4 Crear, configurar y modificar las recetas necesarias....49
5.5 Personalizar la capa *bsp*....52
5.6 Personalizar la capa *distro*....53
5.7 Personalizar la imagen de inicio (*splash-image*)....54
5.8 Añadir los módulos necesarios al núcleo Linux....55
5.9 Crear el repositorio de paquetes....56
5.10 Publicar el repositorio de paquetes....57

5.11 Automatizar la versión para el nombre de los paquetes..60
5.12 Firmado de Paquetes..61
5.13 Configurar el uso del repositorio de paquetes..63
5.14 Generar las *toolchains* externas...63
Capítulo 6 Pruebas..65
6.1 Arrancar el dispositivo *Iwill* con la imagen generada..65
6.2 Audio y vídeo...66
6.3 Aceleración gráfica hardware..66
6.4 Creación del repositorio y *dnf*...67
6.5 Gpg y firmado de paquetes..68
Capítulo 7 Problemas y soluciones...71
7.1 Arranque de la imagen...71
7.2 Paquetes *rpm* delta..71
7.3 Librería *libb64*...72
7.4 Librería *alsa-lib*...73
7.5 Compilación...74
7.6 Claves gpg en la máquina Yocto..74
7.7 Golang..74
7.8 Reinicio del dispositivo *Iwill*...75
7.9 Configurar el sistema de ficheros en modo *journal*...76
Capítulo 8 Conclusiones...77
Bibliografía...79

Índice de figuras

Figura 1. Advertisim....................16
Figura 2. Escenario en producción del sistema Advertisim....................20
Figura 3. Placa del dispositivo Iwill....................22
Figura 4. Adaptador SATA/SSD– usb 3.0....................22
Figura 5. Módulo 3G sobre el adaptador USB....................23
Figura 6. Módulo Wi-Fi....................23
Figura 7. Estructura de directorios principales de Yocto....................24
Figura 8. Ejemplo del código de la receta para el paquete *advertisim-viewer*....................26
Figura 9. Estructura de directorios del paquete *advertisim-viewer*....................27
Figura 10. Listado de capas utilizadas en el proyecto....................29
Figura 11. Fragmento de código de una receta que utiliza la herencia....................30
Figura 12. Esquema de entradas y salidas de bitbake....................31
Figura 13. Fragmento de código que aplica un parche en Yocto....................33
Figura 14. Diagrama de Gantt con la planificación definitiva del proyecto....................41
Figura 15. Información sobre la capa *meta-intel*....................49
Figura 16. Receta de la librería *spandsp*....................51
Figura 17. Receta de *wpa-supplicant*....................52
Figura 18. Receta de configuración del núcleo Linux....................53
Figura 19. Receta de configuración de la capa *distro*....................54
Figura 20. Unit file que ejecuta la imagen *splash*....................55
Figura 21 Configuración del núcleo Linux....................56
Figura 22. Configuración de Nginx para publicar el repositorio de paquetes....................58
Figura 23. Código que permite la descarga del último *commit*....................59
Figura 24. Código que extrae la versión del *tag*....................60
Figura 25. Configuración de acceso al repositorio....................62
Figura 26. Código *qml* utilizado para probar la aceleración hardware....................66
Figura 27. Comprobación de la firma de un paquete....................70
Figura 28. Receta de *libb64*....................73
Figura 29. Configuración para que *go-dep* se ejecute en máquina nativa....................75

Índice de tablas

Tabla 1 Comparación entre *repo* y *git submodules*..32
Tabla 2 Planificación inicial...38
Tabla 3 Planificación final..39
Tabla 4 Coste total aproximado del proyecto..43

Capítulo 1

Introducción

En este capítulo se comentarán cuáles han sido las principales motivaciones por parte de la empresa para desarrollar este proyecto, los objetivos que se han tenido presentes para realizarlo y cuál será la estructura de esta memoria.

1.1 Contexto y motivación del proyecto

El proyecto del cual trata esta memoria se ha realizado en la empresa Nayar Systems, ubicada en el edificio CEEI (Centro Europeo de Empresas Innovadoras) de Castellón, sito en ronda Circunvalación, 188 12003 Castellón de la Plana. La empresa se dedicada al sector de las telecomunicaciones, principalmente orientadas al mundo de los ascensores.

En la fecha de creación de esta memoria la empresa cuenta con 39 trabajadores. Está dividida en 6 departamentos: comercial, administración, garaje, informática, soporte técnico e ingeniería. El departamento de informática, a su vez, está dividido en varios grupos de trabajo: Advertisim, GSR y 72 horas. Durante mi estancia en prácticas formé parte del grupo de trabajo Advertisim.

Actualmente, esta empresa está desarrollando 5 proyectos:

- N4M: Este proyecto consiste en una red privada VPN (Virtual Private Network) cuyo objetivo es comunicar de forma eficiente y segura los distintos dispositivos instalados.
- Nexus: Sistema de comunicaciones utilizado para interconectar todos sus sistemas.
- GSR (*GSM* (*Global System for Mobile communications*) *Smart Router*): Se trata de un sistema embebido conectado a internet que utilizan para control remoto de ascensores (llamadas de emergencia, control de presencia, telemetría, etc.).
- 72h: Proyecto que consiste en realizar llamadas cada 72 horas a los ascensores para verificar que funciona correctamente la línea telefónica.
- Advertisim: Se trata de una pantalla controlada por un dispositivo Linux embebido que permite mostrar información como mensajes publicitarios, vídeos, imágenes o *widgets* (noticias, el tiempo, etc). Se instala en ascensores, espacios comunes de cadenas hoteleras, salas de reuniones, restaurantes o *halls* de manera que sus huéspedes tengan a su alcance todo tipo de información interesante. A continuación, en la Figura 1 se puede ver el producto Advertisim, donde se aprecia cómo muestra una imagen, la planta en la que se encuentra el ascensor, el tiempo, los kilos y nº de personas, etc.

Figura 1: Advertisim.

Con ello han conseguido crear un conjunto de sistemas embebidos que interactúan entre sí y permiten controlar las maniobras del ascensor y el sistema de llamadas telefónicas del mismo, verificar el funcionamiento de la línea telefónica, mostrar información diversa a través de la pantalla instalada en él y crear una conexión VPN privada que conectará el ascensor con un servidor de contenido propio de la empresa.

El proyecto consiste en crear un sistema Linux embebido para el proyecto Advertisim, el cual incluirá todas las herramientas necesarias para llevar a cabo su cometido: herramientas propias desarrolladas por la empresa y algunas externas.

Hasta la fecha de inicio de este proyecto, para cada plataforma en concreto (ARM, MIPS, etc.), instalaban el sistema operativo junto a las aplicaciones y acto seguido creaban una imagen del sistema operativo completo para poder instalarla en otros dispositivos cuya plataforma fuese la misma. Es por ello que el proyecto surge de la necesidad de crear un único sistema operativo que incluya todas estas aplicaciones y que, además, sea fácilmente exportable a las diferentes arquitecturas que se puedan utilizar en los sistemas embebidos. Esto permitirá a la empresa disminuir el tiempo empleado y una menor carga de trabajo, como se verá en el apartado 2.3, ya que no tendrán que desarrollar en un futuro nuevos sistemas operativos ni instalar las aplicaciones de nuevo, pues utilizarán el mismo sistema operativo para todos los proyectos. Además del ahorro de tiempo, facilitará el mantenimiento de los sistemas, ya que todos los dispositivos tendrán el mismo sistema operativo y, además, estos dispositivos serán más fáciles de conocer por todos los miembros de la empresa.

1.2 Objetivos del proyecto

El objetivo principal del proyecto es crear un sistema operativo Linux personalizado para una arquitectura *x86_64*, que sea adaptable fácilmente a las diferentes plataformas (ARM, MIPS, etc.) que la empresa pueda utilizar en los sistemas embebidos y que, además, sea capaz de ejecutar todo el software propio desarrollado por la empresa.

Este objetivo principal puede ser descompuesto en un subconjunto de objetivos:

- Generar un sistema de ficheros que incluya todo lo necesario. Esto implicará:
  - Generar un *BSP (Board Support Packages)* para la arquitectura *x86_64* y que así el sistema operativo resultante pueda ejecutarse en plataformas x86_64.
  - Generar una capa para el proyecto Advertisim donde se incluya el software del proyecto Advertisim. Con ello se conseguirá que en la imagen resultante esté añadido dicho software.
  - Generar las demás capas de software que deberá incluir la imagen, tanto más software propio de la empresa como software externo.
  - Generar la imagen del sistema operativo capaz de ejecutar el proyecto Advertisim sobre la plataforma *x86_64*.
- Alojar todo el proyecto en un repositorio git.

Como objetivos opcionales se pueden establecer los siguientes:

- Generar un *BSP* para la arquitectura ARM y que así el sistema operativo resultante pueda ejecutarse en plataformas ARM.
- Generar la imagen del sistema operativo capaz de ejecutar el proyecto Advertisim sobre la plataforma ARM.

Además de los objetivos iniciales citados en este punto, durante el desarrollo del proyecto surgieron algunos objetivos más:

- Crear un repositorio de paquetes *rpm* propio para la gestión de los paquetes personalizados.
- Firmar los paquetes propios.
- Integrar una herramienta de gestión de paquetes *rpm* que permita hacer consultas al repositorio de paquetes propio, instalar aplicaciones, actualizarlas y borrarlas.

1.3 Estructura de la memoria

En el segundo capítulo se realiza una descripción general del proyecto. Esto incluye una definición de cuáles serán las características principales de dicho proyecto, el punto desde el que se parte, cuáles son las ventajas que le aportará este a la empresa y la descripción del hardware y software utilizado.

En el tercer capítulo puede verse la planificación inicial del proyecto y la que realmente se ha seguido. También se hace una comparativa entre ambas destacando las diferencias más significativas.

El cuarto capítulo plantea los requisitos mínimos que deberá cumplir el proyecto y un análisis general del sistema Advertisim.

El quinto capítulo describe el proceso de desarrollo que se ha llevado a cabo para conseguir el objetivo del proyecto.

El sexto capítulo muestra cuáles han sido las pruebas realizadas para comprobar que el sistema funciona bien y trabaja de la forma adecuada.

En el séptimo capítulo se detallan algunos de los problemas que han surgido durante el desarrollo del proyecto y las soluciones que se han aplicado para poder solventarlos.

Y por último, en el octavo capítulo se presentan las conclusiones. Se aporta una valoración del cumplimiento de objetivos, se citan algunas extensiones del proyecto y se da una opinión personal.

Capítulo 2

Descripción del proyecto

En este capítulo se comentarán las características que tendrá el proyecto, el punto desde el que parte, se listará el software que incluirá el sistema operativo resultante y se comentará el hardware y software utilizado.

2.1 Características del proyecto

Como se ha comentado en el capítulo anterior, el objetivo principal del proyecto es crear un sistema operativo Linux completamente funcional, que permita ejecutar las aplicaciones propias de la empresa y que sea capaz de ejecutarse sobre un sistema embebido. Además, deberá ser fácilmente adaptable a cualquier plataforma.

A continuación detallaré los componentes software que se incluirán en el proyecto :

- *G3d*: Demonio desarrollado por la empresa y utilizado para dar servicio al módulo 3G.
- *N4m*: Al igual que g*3d*, es un demonio propio que se utiliza como cliente de la VPN.
- Pppc: Demonio utilizado para establecer una conexión de tipo PPP (Point-to-Point Protocol) contra el servidor.
- *Advertisim-daemon*: Demonio propio de la empresa que forma parte del proyecto Advertisim. Es el encargado de gestionar el contenido que se mostrará por pantalla a través del demonio *advertisim-viewer.*
- *Advertisim-viewer:* Como *advertisim-daemon*, es un demonio propio de la empresa englobado dentro del proyecto Advertisim. Se encarga de reproducir todo el contenido gestionado por *advertisim-daemon.*
- *Gstreamer*: Es un *framework* que permite crear aplicaciones que sean capaces de reproducir contenido multimedia en *streaming* y permite añadir distintos *códecs* y funcionalidad a través de los *plugins*.
- *Openssh:* Servidor *SSH* (*Secure SHell*) que permitirá crear una conexión contra el sistema utilizando dicho protocolo y cifrando el tráfico que pasa a través de ella.
- *Pppd*: Este demonio se utiliza para establecer y mantener un *link* o túnel con otro sistema.
- *Alsa sound:* Es un *framework* y parte del núcleo que proporciona funcionalidad de sonido y MIDI (*Musical Instrument Digital Interface*) al sistema operativo Linux. Se utilizará para gestionar las diferentes tarjetas de sonido del dispositivo.
- Librería *qt5*: *Framework* de desarrollo que permite crear interfaces gráficas en las

aplicaciones. Será necesaria esta librería ya que *advertisim-viewer* la utiliza para montar todo el entorno gráfico mostrado al usuario por pantalla.

2.2 Punto de partida del proyecto

Al inicio de este proyecto, la empresa Nayar Systems ya había comercializado el sistema Advertisim. Este sistema está compuesto por una placa hardware de terceros sobre la cual se ejecuta un sistema operativo Linux. En él se instalan las aplicaciones propias de la empresa (*advertisim-daemon*, *advertisim-viewer, etc.*) y algunas externas necesarias para hacer funcionar el proyecto Advertisim correctamente. Su cometido es mostrar por pantalla el contenido deseado (noticias, mensajes publicitarios, vídeos, imágenes, etc.), que será gestionado por las aplicaciones *advertisim-daemon* y *advertisim-viewer* y que puede ser controlado remotamente.

En la Figura 2, se representa el escenario de producción del sistema Advertisim. Como se puede ver, por una parte están los diferentes dispositivos Advertisim conectados a la red de Nayar a través de una VPN (nube). Y por el otro lado se ven los distintos usuarios que gestionan los contenidos que serán mostrados en los dispositivos Advertisim a través del Advertisim Manager, el cual también está conectado a la red de la empresa.

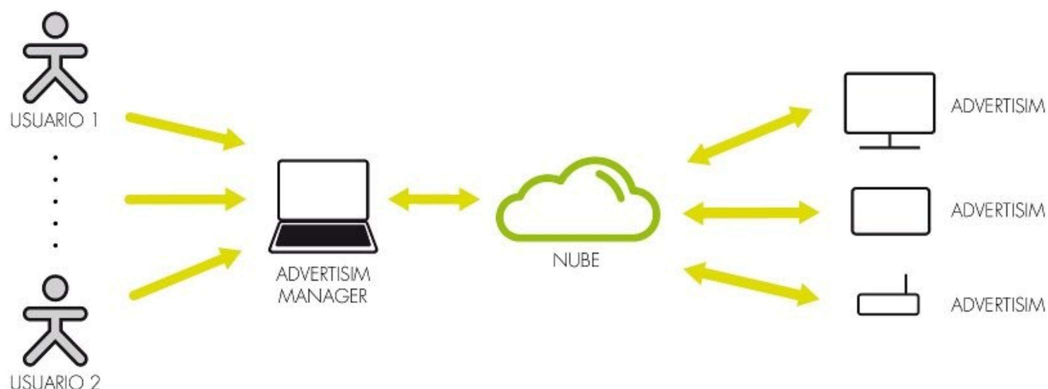


Figura 2. Escenario en producción del sistema Advertisim.
Imagen extraída de Nayar Systems [1]

En este proyecto se va a tratar de crear una imagen del sistema operativo que incluya todo el software propio de la empresa, el cual está almacenado en el repositorio *git Bitbucket* (https://bitbucket.org), y el software externo necesario para hacer funcionar Advertisim en cualquier escenario. Para ello se utilizará una herramienta llamada “Yocto”.

2.3 Ventajas del proyecto

A continuación se citarán las ventajas que ofrecerá este proyecto a la empresa:

- Ahorro de tiempo: Una vez está creada la imagen con todo lo necesario para una placa hardware utilizada en el sistema Advertisim, su instalación en más placas de ese

modelo en concreto será inmediata, sin tener que perder tiempo configurando cada una de ellas manualmente.

- Facilidad en el mantenimiento: El mantenimiento de los dispositivos Advertisim será mucho más fácil para los miembros de la empresa ya que todos serán iguales (mismo sistema operativo, misma configuración, mismo sistema de ficheros, mismas aplicaciones, etc.).
- Facilidad de exportación a otras arquitecturas: Gracias a la herramienta Yocto y a sus capas o "*layers*", teniendo la imagen creada para una arquitectura, dicha imagen es fácilmente exportable a otras modificando únicamente la capa *bsp*, la cual se explicará en el apartado *2.5*.
- Mayor flexibilidad en la gestión de paquetes: Esta ventaja viene dada por la herramienta *dnf*. Antiguamente, los dispositivos embebidos de la empresa utilizaban la herramienta *opkg* en lugar de *dnf*. El principal avance para la empresa es que con *dnf* pueden instalar una versión concreta de un paquete, lo que es muy útil cuando se actualiza el parque de dispositivos cliente y a posteriori se comprueba que ha habido un error y dicha actualización no funciona correctamente. Con esta herramienta se puede volver a instalar una versión anterior del paquete mientras que con *opkg* no se podía hacer.
- Reutilización de trabajo: Gran parte del trabajo de este proyecto se podrá reutilizar en un futuro para construir sistemas operativos personalizados para otros sistemas embebidos.

2.4 Descripción del hardware utilizado

En este apartado se va a especificar el hardware utilizado durante todo el proyecto y sus características:

- Ordenador de usuario: Equipo con Ubuntu 16.04-LTS utilizado para conectarse vía *SSH* a una máquina de Google en la que se compilará todo el proyecto. Este equipo es un ordenador de sobremesa sin ninguna característica en especial. Cuenta con 500 GB de disco duro, 4 GB de memoria RAM *(Random Access Memory)*, procesador de 4 núcleos, al menos 1 puerto USB *(Universal Serial Bus)* versión 3.0 y conexión a la red e internet.
- Máquina de Google (de compilación): Nombrada durante todo el proyecto como "máquina Yocto". Se trata de una máquina virtualizada en los servidores de Google con Debian 9.3 (stretch) como sistema operativo. Se incluye en hardware aunque sea una máquina virtual ya que es utilizada por sus características técnicas o físicas. Se accede a ella vía *SSH*. En esta máquina se hace realmente todo el trabajo del proyecto, pues será donde se instalará Yocto, se compilará el proyecto, se generarán las imágenes del sistema operativo, se ejecutarán las recetas, etc. Las características técnicas de dicha máquina son: procesador Intel Xeon CPU *(Central Processing Unit)* @ 2.60 GHz, 300 GB de disco duro y 6 GB de RAM.
- Dispositivo *Iwill*: Este es el dispositivo final para el cual se creará la imagen personalizada. La placa base que utiliza es el modelo ITX-N29 y las características técnicas son las siguientes: procesador Intel celeron J1900 2.0 GHz, 4 GB de RAM, y 120 GB de disco duro SSD (*Solid-State Drive*).
  Los puertos de conexión disponibles son: 2 USB 3.0, 2 USB 2.0, HDMI (*High-Definition Multimedia Interface*), VGA (*Video Graphics Array*), ethernet, audio

externo, micrófono y puerto serie.

En la Figura 3 se puede ver un dispositivo *Iwill*. El que se ha utilizado en las prácticas estaba cubierto por una carcasa para evitar golpes y mejorar su estética.

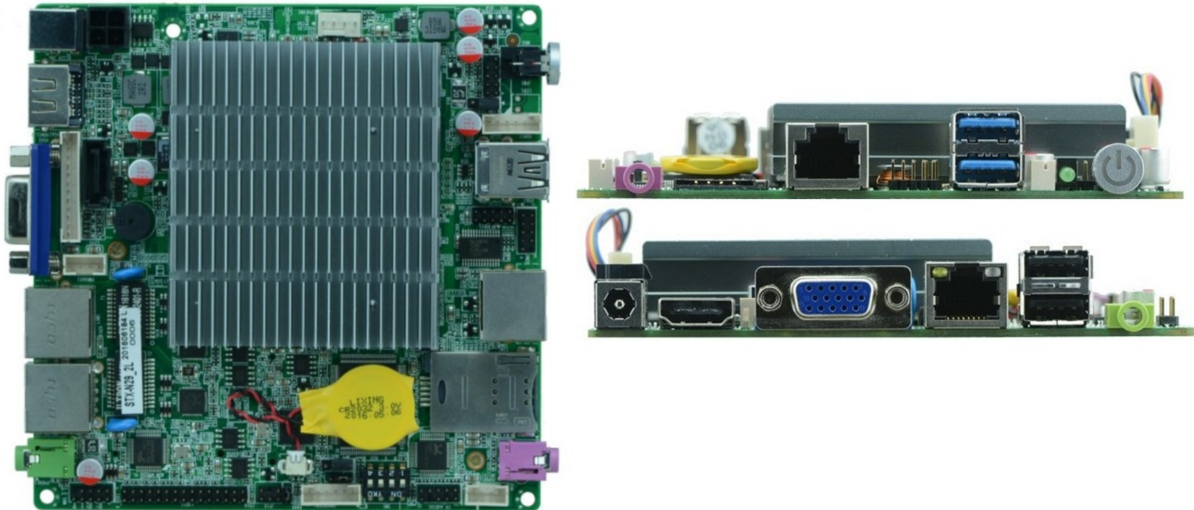


Figura 3. Placa del dispositivo *Iwill*.
Imagen extraída del manual de usuario de la placa ITX-N29

- Adaptador disco duro SATA (*Serial Advanced Technology Attachment*) / SSD a USB 3.0: Este adaptador se ha utilizado para volcar la imagen generada con Yocto directamente sobre el disco duro SSD del dispositivo *Iwill*. En la Figura 4 se puede ver una imagen de uno de estos adaptadores.

Figura 4. Adaptador SATA/SSD– usb 3.0.

- Módulo 3G: El módulo 3G se utiliza para conectar el dispositivo *Iwill* a internet a través de datos móviles. El módulo está diseñado para ser utilizado en ordenadores portátiles, por lo que para poder usarlo en el dispositivo *Iwill* se ha necesitado un adaptador para poder conectarlo por USB. Los drivers para su funcionamiento ya están cargados en la imagen del sistema operativo. El modelo en concreto de este módulo es *Dell 3G WWAN Card DW5550 2XGNJ*. En la Figura 5 se puede ver el módulo montado en el adaptador USB.

Figura 5. Módulo 3G sobre el adaptador USB.

- Adaptador USB WI-FI: Este adaptador se utiliza para que el dispositivo *Iwill* pueda conectarse a una red Wi-Fi. Se conecta por USB y al igual que con el módulo 3G, los drivers ya están cargados en la imagen del sistema operativo. En la Figura 6 se puede ver una imagen de este módulo.

Figura 6. Módulo Wi-Fi.

2.5 Descripción del software utilizado

A continuación se va a describir el software que se ha utilizado en el proyecto y su finalidad.

- Yocto

  Tal y como se describe en la propia web del proyecto Yocto [2], este es un proyecto de colaboración de software libre escrito en python que permite a los desarrolladores crear sistemas operativos para productos embebidos de manera independiente de la arquitectura.

  Su funcionamiento se basa en un sistema de capas, lo que permitirá añadir funcionalidad a una imagen a base de añadir las capas necesarias. También permite que colaboren empresas diseñando capas específicas y la personalización de capas por parte de los desarrolladores.

  Detrás de este proyecto hay empresas colaboradoras tan importantes como Intel, DELL, LG, NXP (Next eXPerience), AMD *(Advanced Micro Devices)*, FreeScale, etc. No solo colaboran con fondos económicos, sino que además mantienen distintas capas.

  Yocto tiene una estructura de directorios bien definida, la cual hay que conocer para poder trabajar. En la Figura 7 se ven algunos de los directorios de Yocto que más se han utilizado en el proyecto.

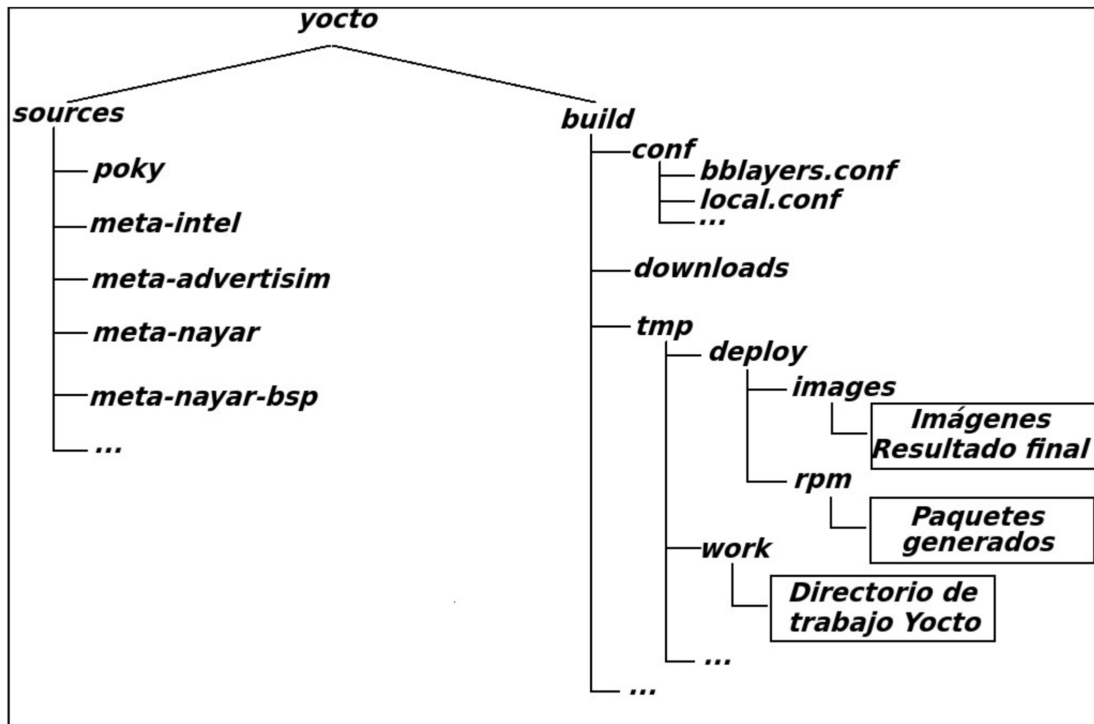


Figura 7. Estructura de directorios principales de Yocto.

En la parte de la izquierda de la Figura 7 (directorio *sources*), se pueden ver varios directorios cuyo nombre empieza por *meta* y otro más llamado *poky*. *Poky* contiene un conjunto de herramientas y metadatos base utilizados para que cualquier desarrollador pueda empezar a construir su propio sistema operativo. Todos los demás directorios hacen referencia a las distintas capas que tiene el proyecto.

En la parte de la derecha de la imagen se muestran los directorios Yocto que más se han utilizado para desarrollar el proyecto. Parten del directorio *build*, el cual está compuesto por los siguientes directorios:

- *conf*: Contiene los ficheros de configuración de Yocto. Los dos ficheros más importantes son *local.conf* y *bblayers.conf*.
  - En *local.conf* se modifican parámetros de configuración tales como el nombre de la distribución (*DISTRO*), la plataforma (*MACHINE*) para la cual se realiza la compilación, el tipo de paquetes (*PACKAGE_CLASSES*) que utilizará la distribución (*rpm*, *ipk o deb*), el número de núcleos del procesador que se utilizarán para compilar (*BB_NUMBER_THREADS*), etc. Cualquier otra variable que queramos declarar para todo el proyecto se puede especificar aquí.
  - Por otro lado, en el fichero *bblayers.conf* se declaran las capas que se utilizarán en el proyecto y el directorio raíz donde se encuentran (*root path*).
- *tmp*: Dentro de este directorio hay varios subdirectorios. Los dos más importantes son los que aparecen en la Figura 7 y serán descritos a continuación:
  - D*eploy:* Directorio donde se almacenarán los paquetes e imágenes construidos por Yocto, es decir, los paquetes *rpm* de aplicaciones generados

a partir de las recetas definidas, las cuales serán descritas a continuación, y las imágenes resultantes con el sistema operativo funcional y todo el software especificado.

- *Work:* Directorio de trabajo de Yocto. Aquí será donde se descarguen y descompriman los archivos fuente para generar los paquetes *rpm* definidos en sus correspondientes recetas, las cuales serán descritas a continuación. En él también se colocarán los ficheros antes de ser empaquetados. Además, aquí se almacena un registro de actividad (*log*) con la lista de tareas que Yocto ha realizado para poder generar los paquetes y, en caso de fallo, poder ver cómo y por qué ha fallado.

➢ *downloads*: Directorio donde se descargan todos los códigos fuente de los distintos proyectos que componen un sistema operativo Linux, por ejemplo el núcleo. Todo este código fuente será compilado y utilizado por Yocto para generar la imagen.

A la hora de trabajar con Yocto es necesario conocer algunos componentes o términos que serán descritos a continuación:

◆ <u>Recetas</u>

Las recetas son archivos de instrucciones o tareas que tiene que seguir Yocto para poder construir los paquetes de aplicaciones y generar la imagen final. Se distinguen de otros ficheros porque tienen la extensión *.bb o .bbappend*. La diferencia entre estas dos extensiones de fichero es que, según el estándar Yocto, las recetas deben tener la extensión *.bb* y los ficheros de modificación de recetas la extensión *.bbappend*.

Las recetas pueden ser divididas en dos tipos:

➢ Recetas de paquetes: Son las recetas que contienen las instrucciones para generar un paquete.

➢ Recetas de imagen: Contienen las instrucciones para generar una imagen. En ella se pueden especificar los paquetes que se instalarán en la imagen, tamaño del sistema de ficheros, características que deberá incorporar la imagen, etc.

En la Figura 8 aparece una captura de pantalla en la que se muestra una receta creada en el proyecto. Como se puede apreciar en ella, las instrucciones que se le pasan a Yocto a través de las recetas se configuran en forma de variables. A continuación se describirá cada una de las partes de código que aparecen en dicha figura, que están numeradas a la derecha de esta.

La parte número 1 se corresponde con la descripción de la receta y el tipo de licencia que utilizará. En este ejemplo concreto se modifican las siguientes variables:

➢ *SUMMARY*: Variable utilizada para añadir una descripción corta como metadato en el paquete generado.

➢ *DESCRIPTION*: Variable utilizada para añadir una descripción como metadato en el paquete generado. Se diferencia de la anterior en que será la descripción utilizada por los gestores de paquetes.

➢ *LICENSE*: Variable utilizada para especificar el tipo de licencia del código fuente utilizado por esta receta.

```
SUMMARY = "Advertisim-viewer"
DESCRIPTION = "Software to show multimedia content"
LICENSE = "CLOSED"

DEPENDS = "qtbase qttools-native qtdeclarative qtwebsockets qtmultimedia"

# Depends on gles2 enabled and that's not default configuration
EXCLUDE_FROM_WORLD = "1"

INHIBIT_PACKAGE_DEBUG_SPLIT = "1"

SRC_URI = "git://git@bitbucket.org/nayar/advertisim_viewer;protocol=ssh;branch=master"
SRCREV = "${AUTOREV}"

S = "${WORKDIR}/git"

do_install() {
    install -d ${D}/opt/bin
    install -d ${D}${bindir}
    install -m 0755 ${B}/advertisim_viewer ${D}${bindir}
    ln -sf /usr/bin/advertisim_viewer ${D}/opt/bin/advertisim_viewer
}

do_package_prepend () {
    import subprocess
    wdir = d.getVar('S')
    os.chdir(wdir)
    ownvers = subprocess.getoutput('git describe')
    d.setVar('PKGV', ownvers)
}

FILES_${PN}_append = " /opt/bin"

FILES_${PN} += "${bindir}/advertisim_viewer"
FILES_${PN}-dbg += "${bindir}/.debug/advertisim_viewer"

RDEPENDS_${PN} = "packagegroup-nayar-advertisim-qt5"
```

Figura 8. Ejemplo del código de la receta para el paquete *advertisim-viewer*.

En la parte número 2 de la Figura 8 se especifican cuáles serán las dependencias de la receta. Esto se utiliza para indicar a Yocto que antes de empezar a compilar esta receta, haya compilado todas las que aparecen en la lista de la variable DEPENDS, es decir, se crea una dependencia de compilación. En este ejemplo concreto, antes de compilar la receta del advertisim-viewer, Yocto debe compilar las recetas *qtbase, qttools-native, qtdeclarative, qtwebsockets y qtmultimedia*.

En la parte número 3 se indica el lugar desde donde se descargarán los archivos fuente, se le indica a Yocto qué revisión del código fuente del repositorio debe descargar y la ruta del directorio de trabajo donde se descargarán los fuentes. En este ejemplo concreto se han modificado las siguientes variables:

➢ *SRC_URI*: Utilizada para indicar a Yocto que el código fuente se descargará desde la rama *master* de un repositorio alojado en *bitbucket* dentro del *path nayar/advertisim_viewer* utilizando el protocolo *SSH*.
➢ *SRCREV*: En esta variable se le indica la revisión del código fuente del repositorio que se descargará Yocto. En este caso, al darle el valor *${AUTOREV}* se le está indicando que se descargue la última.

- *S*: Se utiliza para especificarle a Yocto cuál será el directorio de trabajo donde se descargarán los fuentes.

En la parte número 4 de la Figura 8 se pueden observar dos bloques de código. El primero de ellos, escrito en lenguaje *shell-script*, hace referencia a la tarea *do_install*. Es un código que se utiliza para crear la estructura de ficheros que tendrá el paquete resultante. En este caso concreto se crean dos directorios (*/opt/bin* y */usr/bin*) con el comando *install -d.* Una vez creados, con *install -m* se copia el fichero *advertisim-viewer* a */usr/bin.* Acto seguido se crea un enlace simbólico sobre este fichero y se almacena en */opt/bin*. Por lo tanto, el paquete generado tendrá la estructura de directorios que aparece en la Figura 9:

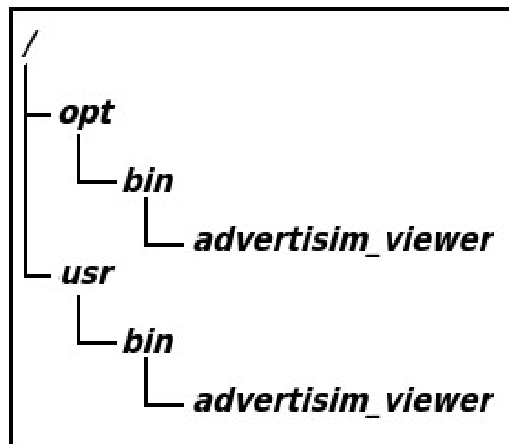


Figura 9. Estructura de directorios del paquete *advertisim-viewer*.

El bloque que hace referencia a la tarea *do_package_prepend* se ejecuta justo antes de que Yocto genere el paquete. Este bloque es código Python, y lo que hace es ejecutar el comando *git describe* (para obtener el último *tag* creado en el repositorio *git*) para colocar el resultado en la variable *PKGV*, que es la variable que utiliza Yocto para saber la versión del paquete y añadirla al nombre del mismo. Con esto conseguimos que la versión del paquete dependa del último *tag* creado en el repositorio *git*, lo cual se explicará con más detalle en el apartado 5.11.

En la parte número 5 de la receta se utiliza la variable *FILES*. Con ello se indica a Yocto que en el paquete *advertisim-viewer.rpm* generado se debe integrar el directorio */opt/bin* y el fichero */usr/bin/advertisim_viewer*. Como se puede observar en este código, hay dos formas de añadir contenido a un paquete, utilizando *FILES_${PN}_append* y usando *FILES_${PN}* +=. Se puede ver también la variable *FILES_${PN}-dbg* que es utilizada para añadir el fichero *advertisim_viewer* alojado en el directorio oculto */opt/bin/.debug* a un paquete rpm distinto del anterior. Con todo esto la receta creará dos paquetes, el paquete *advertisim-viewer.rpm* y el paquete *advertisim-viewer-dbg.rpm*.

Por último, en la parte número 6 se puede ver el uso de la variable *RDEPENDS*. Esta variable se utiliza para indicar a Yocto que antes de que genere el paquete de esta receta, deben estar construidos los paquetes que aparecen aquí. Además, se

genera una dependencia en tiempo de ejecución entre el paquete generado por esta receta y los paquetes incluidos en *RDEPENDS*, lo que quiere decir que a la hora de instalar ese paquete deben estar instalados primero los demás.
Packagegroup-nayar-advertisim-qt5 es un conjunto de paquetes definido en una receta, gracias a ello Yocto sabe qué paquetes forman dicho grupo.
En este caso concreto, Yocto primero deberá construir todo ese conjunto de paquetes para después construir el paquete de esta receta.
Al crearse la dependencia en tiempo de ejecución, en un entorno de producción, antes de poder instalar el paquete generado por esta receta, se deberá instalar el conjunto de paquetes especificado en dicha variable.

- Compilador cruzado

El proyecto Yocto provee los fuentes de un compilador *gcc*, el cual se configura y compila para hacerlo cruzado. Se dice que es cruzado porque es capaz de compilar software en una máquina nativa (llamada *host*) que será ejecutado en otra máquina destino (llamada *target*). Por ejemplo, Yocto podría generar un compilador *arm-linux-gc*c que compila software sobre una plataforma *x86_64* pero que genera código para una plataforma destino ARM. En el caso concreto de este proyecto, la máquina nativa es la máquina Yocto y la máquina destino el dispositivo *Iwill*.
Esta parte de Yocto es la que permite que, una vez creado un proyecto que genera una imagen para una plataforma determinada, sea fácilmente exportable a cualquier otra plataforma. Este compilador se configura con la información de las capas *bsp*, descritas a continuación.

- Capas

Una capa se podría definir como un conjunto de recetas. Son utilizadas para poder hacer el proyecto más modular. Por ejemplo, puede haber una capa que permita añadir un interfaz gráfico a la distribución, de modo que simplemente añadiendo y configurando dicha capa al proyecto se consigue que automáticamente disponga de ese interfaz.
Yocto dispone de un conjunto de capas [3] a disposición de cualquier desarrollador, nombrado de aquí en adelante repositorio de capas de OpenEmbedded.
Existen diferentes clases de capas. A continuación expongo las utilizadas en este proyecto:

- Capas base: Estas son las capas *openembedded-core* y *meta-oe*. Contienen lo mínimo para poder generar una imagen que funcione: un núcleo mínimo, sistema de ficheros mínimo, etc.
- Capas BSP (Board Support Package): Son capas que se encargan de definir la plataforma hardware destino para la cual se van a generar los paquetes y la imagen. Esto quiere decir que en ella se describirá la información necesaria para que Yocto pueda preparar el compilador cruzado. También incluyen una receta que se encarga de descargar y compilar el núcleo Linux, una receta para modificar el *bootloader* y recetas para configurar la información sobre otros dispositivos hardware que se conectan al dispositivo destino.
  En la documentación del proyecto Yocto [4], se pueden encontrar varios ejemplos de capas de este tipo: *meta-intel, meta-raspberrypi, meta-arduino*, etc. Como se puede intuir, los propios nombres indican que son capas para

ciertas plataformas (Intel, raspberry pi, arduino, etc.).
Generalmente, estas capas incluyen la receta para compilar y generar una imagen con una versión en concreto del núcleo Linux. Esto lo hacen los desarrolladores porque han probado su hardware con esa versión en concreto y saben que con ella funciona correctamente.

- Capas software: Estas capas están formadas por recetas que se encargan de generar ciertos paquetes y añadirlos a la imagen. Se les suele dar un nombre relacionado con el tipo de software que incluyen sus recetas. Por ejemplo, la capa del repositorio de capas de OpenEmbedded *meta-office* contiene recetas para generar paquetes como *LibreOffice*. Con la capa *meta-python* se puede integrar python a la imagen, etc.
- Capas de distribución: Son capas que contienen recetas orientadas a personalizar la distribución creada. Ejemplos de dicha personalización pueden ser ponerle una imagen de inicio, instalar un conjunto de software determinado y propio de la distribución, etc.

Un aspecto importante a la hora de trabajar con las capas es la prioridad de cada una de ellas. Puede darse el caso de que haya una receta duplicada en más de una capa. Esto generaría un error ya que Yocto no sabría qué receta de las duplicadas utilizar. La solución que tiene es utilizar la receta que esté almacenada en la capa con mayor prioridad. Cuanto mayor sea el número de prioridad, mayor prioridad tendrá la capa. En la Figura 10 se puede apreciar el nombre, ruta y prioridad de todas las capas que se utilizaron en el proyecto. Se observa que la capa con mayor prioridad es la capa *workspace*.

```
cmollon@yocto:~/yocto/build-intel$ bitbake-layers show-layers
NOTE: Starting bitbake server...
layer                 path                                              priority
==========================================================================
meta                  /home/cmollon/yocto/sources/poky/meta      5
meta-poky             /home/cmollon/yocto/sources/poky/meta-poky  5
meta-yocto-bsp        /home/cmollon/yocto/sources/poky/meta-yocto-bsp  5
meta-nayar            /home/cmollon/yocto/sources/meta-nayar     10
meta-intel            /home/cmollon/yocto/sources/meta-intel     5
meta-iot-cloud        /home/cmollon/yocto/sources/meta-iot-cloud  10
meta-oe               /home/cmollon/yocto/sources/meta-oe/meta-oe  6
meta-networking       /home/cmollon/yocto/sources/meta-oe/meta-networking  5
meta-python           /home/cmollon/yocto/sources/meta-oe/meta-python  7
meta-advertisim       /home/cmollon/yocto/sources/meta-advertisim  8
meta-nayar-bsp        /home/cmollon/yocto/sources/meta-nayar-bsp  10
meta-qt5              /home/cmollon/yocto/sources/meta-qt5       7
meta-gsr              /home/cmollon/yocto/sources/meta-gsr       10
workspace             /home/cmollon/yocto/build-intel/workspace  99
cmollon@yocto:~/yocto/build-intel$
```

Figura 10. Listado de capas utilizadas en el proyecto.

Por defecto, al empezar a trabajar con Yocto se incluyen las capas *meta, meta-poky, meta-yocto-bsp y workspace*.
Las capas *meta-nayar, meta-advertisim, meta-nayar-bsp y meta-gsr* fueron creadas y personalizadas durante el desarrollo del proyecto, mientras que las demás están dentro del proyecto Yocto y son mantenidas por empresas y particulares que colaboran con el proyecto Yocto.
Meta-advertisim contiene recetas para software del proyecto Advertisim, las recetas de *meta-nayar* son para software que puede ser común a otros proyectos de la empresa Nayar, la capa *bsp meta-nayar-bsp* contiene recetas para la configuración del dispositivo *Iwill* y en *meta-gsr* se engloban las recetas que permiten instalar el software del proyecto GSR.

- Clases

  Son archivos escritos en python que abstraen funcionalidad común y permiten utilizarla en las recetas. Son fácilmente identificables por la extensión *.bbclass*. Para utilizarlas en una receta simplemente hay que asegurarse que dicha receta herede la clase.

  Un ejemplo de clase puede ser *autotools.class*. Esta clase define un conjunto de tareas (configurar, compilar, etc.) que permiten compilar y generar paquetes que hayan sido creados con *Autotools*. *Autotools* es una suite de herramientas de programación diseñada para crear las instrucciones necesarias que se deben ejecutar para compilar el código fuente de una aplicación de forma automática y que sea exportable a otros sistemas Unix.

  En la Figura 11 se muestra un fragmento de código de una receta en la que se puede observar cómo se hereda de las clases *autotools* y *pkgconfig*. Esto permite que esta receta cuente con toda la lógica que hay en dichas clases.

```
#Recipe for lib spandsp
LICENSE = "GPLv2 & LGPLv2.1"
LIC_FILES_CHKSUM = "file://COPYING;md5=8791c23ddf418deb5be264cffb5fa6bc \
                    file://debian/copyright;md5=72bfe191182c473bf79b5371353be3bc"

SRC_URI = "http://gsr-repo.n4m.zone/repo/external/libs/spandsp-${PV}.tar.gz \
        file://rpl_malloc.patch"
SRC_URI[md5sum] = "897d839516a6d4edb20397d4757a7ca3"
SRC_URI[sha256sum] = "cc053ac67e8ac4bb992f258fd94f275a7872df959f6a87763965feabfdcc9465"

inherit autotools pkgconfig
```

Herencia

Figura 11. Fragmento de código de una receta que utiliza la herencia.

- Bitbake

  Bitbake es una herramienta dentro del proyecto Yocto que permite ejecutar tareas escritas en Python o guiones del shell de forma concurrente y resolviendo las dependencias que hay entre ellas de forma automática.

  Se encarga de leer e interpretar los metadatos aportados por las recetas, ficheros de configuración y clases para ejecutar las tareas declaradas en ellos, obtener el código fuente a compilar y sincronizar las tareas necesarias para poder llegar a generar tanto paquetes como imágenes.

  En la Figura 12 se puede ver una representación de *bitbake* como un horno, el cual toma como entradas distintos tipos de metadatos (recetas, configuraciones, clases, etc.) y código fuente obtenido de repositorios como pueden ser *bitbucket* o *github* para poder “cocinar” o generar imágenes que puedan utilizarse en sistemas embebidos.

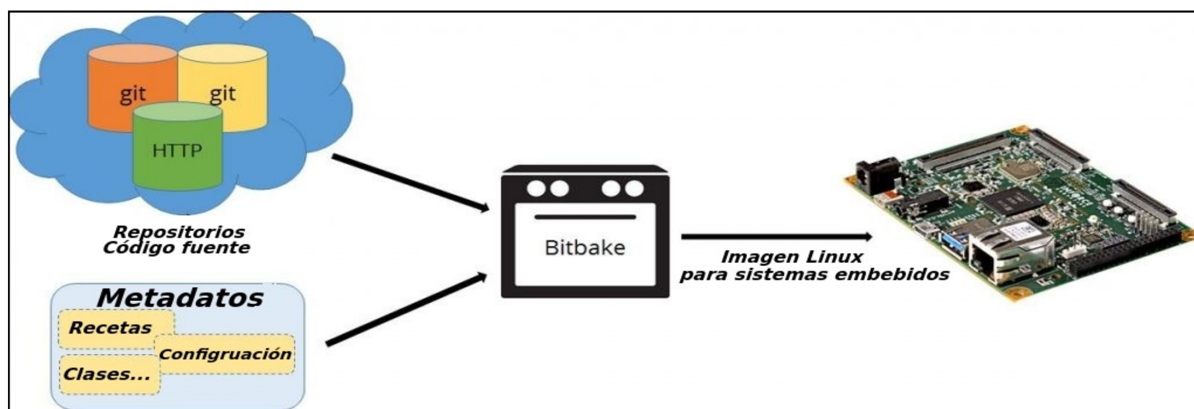


Figura 12. Esquema de entradas y salidas de bitbake.
Imagen extraída de Inforcecomputing [5]

A continuación se expondrán algunos de los comandos *bitbake* que más se han utilizado a lo largo del proyecto y una explicación de cada uno:

- *bitbake <nombre_receta>*: Mediante este comando se ejecutará *bitbake* sobre la receta señalada. Si la receta es de paquete, se compilará el código y se generará el paquete. Y si la receta es de imagen se generará la imagen. Todo ello siguiendo las instrucciones de la receta.
- *bitbake <nombre_receta> -c cleanall*: Borra los archivos que se hayan podido generar de anteriores ejecuciones de "*bitbake nombre_receta*". Se utiliza cuando se quiere ejecutar una receta como si no hubiera sido ejecutada nunca antes.
- *bitbake <nombre_receta> -e*: Muestra el valor de todas las variables que intervienen en la ejecución de la receta.

- <u>OpenEmbedded</u>

  OpenEmbedded [6] es un *framework* que provee un conjunto de metadatos formado por las capas básicas, recetas y clases necesarias para poder crear una distribución de Linux para sistemas embebidos, es decir, ofrece parte de los metadatos que la herramienta *bitbake* utiliza como entrada. Además de esto ofrece un entorno de compilación (*gcc*) que configurándolo debidamente permite convertirlo en un compilador cruzado. Para generar el compilador cruzado adecuado para la plataforma en concreto se utilizan las capas *bsp* descritas previamente. Este *framework* es utilizado y mantenido por el proyecto Yocto.

- <u>Poky</u>

  Es la herramienta que proporciona Yocto para poder generar sistemas operativos Linux propios. Engloba las herramientas citadas anteriormente que permiten crear una distribución de Linux propia: *OpenEmbedded* y *bitbake*.

• Repositorios git

Los repositorios son lugares de almacenamiento para organizar y centralizar ficheros. Pueden ser locales o remotos. Por otro lado, g*it* es un sistema de control de versiones que fue creado por Linus Torvalds. Si se juntan ambas herramientas (repositorios y *git*)

se obtienen los repositorios *git*. A lo largo de todo el desarrollo se han utilizado los repositorios *git* de *bitbucket* y *github*. Además, el control de versiones ha permitido que mientras yo trabajaba con el proyecto, mi supervisor pudiera ir viendo mis avances aun estando en localizaciones distintas. La parte del proyecto que se ha almacenado en el repositorio ha sido el directorio *sources* que se muestra en la Figura 7.

Git dispone de una herramienta denominada *tag* (*git tag*) con la que ya trabajaba anteriormente la empresa. Se utiliza para crear etiquetas en puntos importantes de la historia de un proyecto software, por ejemplo, para marcar números de versión en un estado concreto del código desarrollado. Durante el desarrollo de este proyecto se ha utilizado el valor de los *tags* tal y como se verá con mayor detenimiento en el apartado 5.11.

Muchas de las capas están alojadas y disponibles en el repositorio de capas de OpenEmbedded (*meta-intel, meta-oe, etc.*), por lo cual no hace falta almacenarlas en un repositorio propio. Esto obligaba a trabajar con varios repositorios, pero para este proyecto se necesita una forma sencilla de gestionarlos todos juntos. Para ello se valoró la utilización de dos herramientas, las cuales serán descritas a continuación:

- Submódulos de *git*: es una herramienta *git* que se utiliza para incluir otros repositorios externos dentro de un repositorio. Un submódulo se podría ver como un puntero. Todos los repositorios externos que estén apuntados por un submódulo serán clonados cuando se clone el repositorio principal.
- *Repo* de Google: es un script escrito en python y desarrollado por Google para trabajar con el proyecto Android. Trabaja sobre *git* y también permite incluir otros repositorios externos dentro de un repositorio.

Como se puede observar en la Tabla 1, es mayor el número de ventajas que ofrecen los submódulos de *git* y menor el número de inconvenientes, por lo que en este proyecto se decidió utilizar estos en lugar de la herramienta *repo*.

	Ventajas	Inconvenientes
Repo	Desarrollada por Google.	Hace falta instalación. Poca documentación. Menos estandarizado.
Submódulo	Viene incorporada con *git*. Bastante documentación.	Dificultad de mantenimiento a la hora de modificar submódulos.

Tabla 1. Comparación entre *repo* y *git submodules.*

- Diff y Patch

Estas herramientas se utilizan para crear parches en una aplicación.

Diff se utiliza para comparar dos archivos y generar las diferencias entre ambos. En este proyecto se ha usado para comparar dos versiones del código fuente de una aplicación y generar un parche, que realmente es un fichero con las diferencias entre los códigos fuente de ambas versiones.

La herramienta *patch* se utiliza en Linux para aplicar un parche. Lo que hace realmente es aplicar la diferencia de los códigos fuente de ambas versiones.

Los parches son creados manualmente con la herramienta *diff*. La forma de aplicarlos

directamente sobre un código fuente sería utilizando la herramienta *patch*. Yocto los aplica de forma automática, y para ello simplemente hay que declararlo en la variable *SRC_URI* de la receta. En la Figura 13 se puede observar en un cuadrado de color rojo cómo se le indica a la receta que existe un parche que deberá aplicar al código fuente antes de compilarlo.

```
SRC_URI += "git://github.com/carlos/calculadora.git;protocol=https;branch=testing; \
       file://parche.patch;striplevel=0 \
       file://advertisimos.hddimg \
       "
```

Figura 13. Fragmento de código que aplica un parche en Yocto.

- Meld

  Herramienta que se utiliza para comparar de forma visual dos archivos o directorios.

- GNU toolkit

  Bajo este nombre se engloban algunas herramientas de Unix con las que se ha trabajado durante todo el proyecto. Son herramientas que suelen estar incorporadas en cualquier distribución de Linux y algunas de ellas son: find, grep, sed y dd.

- Createrepo

  Utilidad de Unix que sirve para crear un repositorios de paquetes rpm a partir de un conjunto de paquetes.
  Ha sido utilizado para crear un repositorio de paquetes propio con los paquetes generados por Yocto, aunque más adelante se vio que no era necesaria esta herramienta y se dejó de utilizar.

- Nginx

  Nginx es un servidor web multiplataforma.
  Forma parte de este proyecto puesto que el repositorio de paquetes rpm se publicó en un servidor web Nginx.

- Rsync

  Programa que permite sincronizar ficheros y directorios ubicados en dos puntos diferentes. Se utiliza por su eficiencia durante la sincronización ya que únicamente transfiere la diferencia entre ambos extremos, que es justo lo necesario para que estén en el mismo estado.
  Ha sido utilizado para sincronizar el directorio donde Yocto deposita todos los paquetes rpm con el directorio donde el servidor web almacena todos los paquetes rpm que se sirven.

- GPG (GNU Privacy Guard)

  Software que permite cifrar y firmar datos y comunicaciones. También sirve como gestor de claves asimétricas.

En el actual proyecto se ha utilizado para generar un par de claves (pública y privada). Estas claves han sido utilizadas para poder firmar los paquetes generados por Yocto y poder verificar que los paquetes han sido firmados con la clave correcta a la hora de instalarlos.

- SSH-keygen

  Herramienta cuya utilidad es generar claves RSA (Rivest, Shamir y Adleman) para autenticación en conexiones SSH.
  Para poder conectarnos a la máquina remota donde se ejecutaba el entorno Yocto ha sido necesario hacerlo mediante el protocolo de administración remota SSH. Para que la autenticación se realice de forma segura en el servidor se han utilizado las claves RSA generadas con esta herramienta.

- RPM (RPM Package Manager)

  Herramienta de administración y gestión de paquetes rpm.
  Ha sido utilizada para hacer consultas sobre los paquetes generados por Yocto (versión del paquete, ficheros del paquete, dependencias, etc.).

- DNF (Dandified Yum)

  DNF es la actualización de Yum, una herramienta de gestión de paquetes para sistemas Unix basados en paquetes rpm. Se ha añadido a la imagen para poder administrar los paquetes instalados, instalar nuevos, actualizarlos, bajarles la versión, etc.
  Esta aplicación, además de haberse utilizado, ha sido incorporada a la imagen para disponer de ella.

2.6 Estado del arte

Este proyecto se ha desarrollado para ser utilizado en el marco del producto Advertisim de la empresa Nayar Systems, aunque podrá ser exportable a otros productos que utilicen sistemas embebidos, sea cual sea su ámbito de funcionamiento (tiendas, restaurantes, salas de congresos, etc.). Antes de la realización de este proyecto, la empresa ya comercializaba dicho producto. El nuevo Advertisim funcionará y tendrá las mismas características que antes, pero este proyecto le aportará las ventajas ya citadas en el apartado 2.3.

Haciendo un estudio de mercado se han encontrado dos productos con similares características a Advertisim. Los dispositivos son los siguientes:

- El modelo LT-line de la marca Flexypage.

  Es un modelo de pantalla que permite, al igual que Advertisim, mostrar contenido multimedia por pantalla, conectar con el ascensor y mostrar datos de la planta en la que se encuentra, la planta destino, la carga en Kg. que tiene, sentido del ascensor etc., conectarse a una red, leer dispositivos USB, memorias microSD, cuenta con entradas y salidas digitales, etc
  Hay varios tamaños, igual que sucede con Advertisim. A diferencia de Advertisim, no

tiene posibilidad de conexión por wifi y 3G.
Se puede encontrar más información sobre este producto en la web de Flexypage [7].

- El modelo XmediaLite de la marca Microlift.
  Al igual que el anterior, es un sistema embebido conectado a una pantalla que se utiliza para mostrar contenido multimedia y las maniobras e información de un ascensor. También está disponible en varios tamaños de pantalla. Es un dispositivo que utiliza un sistema operativo Linux como Advertisim y cuenta con salidas de hdmi, vga y audio.
  Se puede encontrar más información sobre este producto en la web de Microlift [8].

Como se ha podido observar en la descripción de estos productos, ambos siguen la misma línea que Advertisim, un sistema embebido con la lógica suficiente para controlar lo que se reproduce por una pantalla mostrando información del ascensor y contenido multimedia bajo demanda.

Capítulo 3

Planificación del proyecto

3.1 Planificación inicial

En la Tabla 2 se expone la planificación del proyecto realizada a priori. En ella se especifican las distintas tareas que se planificaron desde un principio.

Número	Tarea	Tiempo (h.)	Dependencias
1	Formación previa.	100	
1.1	Formación de herramientas varias.	51	
1.1.1	Entender proceso de arranque de Linux.	5	
1.1.2	Configurar y generar *toolchains* y *cross-platform toolchains*.	10	
1.1.3	Sistemas de compilación (*make* y *autotools*).	8	
1.1.4	Sistemas de paquetes (*rpm*, *deb* e *ipk*).	10	
1.1.5	Empaquetado de imágenes para generar *rootfs*.	2	
1.1.6	Gestión de repositorios *git*.	8	
1.1.7	*Busybox*.	8	
1.2	Formación de Yocto.	49	
1.2.1	*Bitbake*.	15	
1.2.2	Capas y capas *BSP*, recetas y clases.	15	
1.2.3	Recetas y clases.	10	
1.2.4	Estructura de directorios que utiliza Yocto.	9	
2	Desarrollo del proyecto para la arquitectura *x86_64*.	176	1
2.1	Generar una capa *BSP*.	40	
2.2	Generar *rootfs_minimal* utilizando la capa *BSP*.	12	2.1
2.3	Crear una capa personalizada y añadirla.	112	
2.3.1	Crear las recetas necesarias (*iptables*, *qt5*, *golang* y *gstreamer*).	40	
2.4	Generar el *rootfs* con la capa añadida.	4	2.3
2.4.1	Generar el *rootfs*.	2	
2.4.2	Probar el *rootfs*.	2	2.4.1
2.5	Alojar la capa en un repositorio *git* y automatizar su gestión con *repo*.	8	2.3
3	Configuraciones varias para el sistema operativo de la imagen.	15	2
3.1	Modificar recetas para realizar las configuraciones.	13	
3.2	Crear la imagen.	2	3.1
4	Pruebas de la imagen final.	9	3.2
	Duración total del proyecto.	300	

Tabla 2: Planificación inicial.

3.2 Planificación final

En la Tabla 3 se puede ver la planificación final del proyecto.

Número	Tarea	Tiempo (h.)	Dependencias
1	Formación previa.	66	
1.1	Formación de herramientas varias.	44	
1.1.1	Entender proceso de arranque de Linux.	2	
1.1.2	Configurar y generar *toolchains* y *cross-platform toolchains*.	7	
1.1.3	Sistemas de compilación (*make* y *autotools*).	3	
1.1.4	Sistemas de paquetes (*rpm*, *deb* e *ipk*).	3	
1.1.5	Gestionar repositorios *git*.	3	
1.1.6	Generar librerías estáticas y dinámicas.	3	
1.1.7	Crear aplicaciones.	7	
1.1.7.1	Crear la aplicación de prueba calculadora.	3	
1.1.7.2	Crear parches sobre el código fuente de calculadora.	3	
1.1.7.3	Aplicar parches a calculadora.	1	
1.1.8	Empaquetado de imágenes para generar *rootfs*.	2	
1.1.8.1	Crear partición con fichero *dd*.	1	
1.1.8.2	Montar y desmontar la partición creada en el punto anterior.	1	
1.1.9	*Busybox*.	3	
1.1.10	Herramienta *repo* y submódulos de *git*.	6	
1.1.11	Configurar y compilar el núcleo Linux.	5	
1.2	Formación de herramientas Yocto.	22	
1.2.1	*Bitbake*.	7	
1.2.1.1	Documentación sobre *bitbake* y Yocto [9].	3	
1.2.1.2	Crear ejemplo con *bitbake* (Hola mundo).	5	
1.2.2	Capas (capas base, capas software, capas *bsp* y capas *distro*), recetas y clases.	7	

Tabla 3(I). Planificación final.

Número	Tarea	Tiempo (h.)	Dependencias
2	Preparación del entorno de trabajo.	7	1
2.1	Generar las claves RSA y compartir la clave pública para poder acceder por *SSH*.	1	
2.2	Instalar el entorno Yocto.	2	2.1
2.3	Pruebas.	4	2.2
2.3.1	Pruebas con capas.	2	1.2.3
2.3.2	Pruebas con recetas.	2	1.2.4
3	Desarrollo del proyecto.	136	
3.1	Formación durante el desarrollo.	18	
3.1.1	GPG y firmado de paquetes.	7	
3.1.2	Versionado automático y creación de repositorio.	7	
3.1.3	*Systemd*.	4	
3.2	Crear el proyecto y almacenarlo en un repositorio.	3	2
3.3	Añadir las capas externas necesarias como submódulos.	6	3.2
3.4	Crear la capa *BSP* y añadirla como submódulo.	10	3.2
3.5	Crear capa personalizada *meta-advertisim* y añadirla como submódulo.	2	3.2
3.6	Crear y modificar recetas (*advertisim-viewer, qt5, iptables, advertisim-daemon, advertisim-static, gstreamer, systemd, golang, dnf , etc*).	70	3.5
3.7	Crear un repositorio de paquetes.	23	1.1.12
3.7.1	Versionado automático de paquetes.	12	3.1.2
3.7.2	Firmado de paquetes.	11	3.1.1
3.8	Paquetes *delta*.	4	3.7
4	Configurar el sistema operativo de la imagen.	41	
4.1	Configurar el sistema operativo resultante mediante recetas (cambio contraseña root, imagen *splash* inicio, fuentes tipográficas, añadir y configurar *systemd*, configurar software de red, etc).	40	3.5, 3.1.3
4.2	Generar de la imagen.	1	4.1
5	Testeo y pruebas.	50	
5.1	Realizar pruebas con recetas.	20	3.6
5.2	Realizar pruebas de la imagen.	30	3.10
	Duración total del proyecto.	300	

Tabla 3(II). Planificación final.

En la Figura 14 puede verse el diagrama de Gantt basado en la planificación final del proyecto.

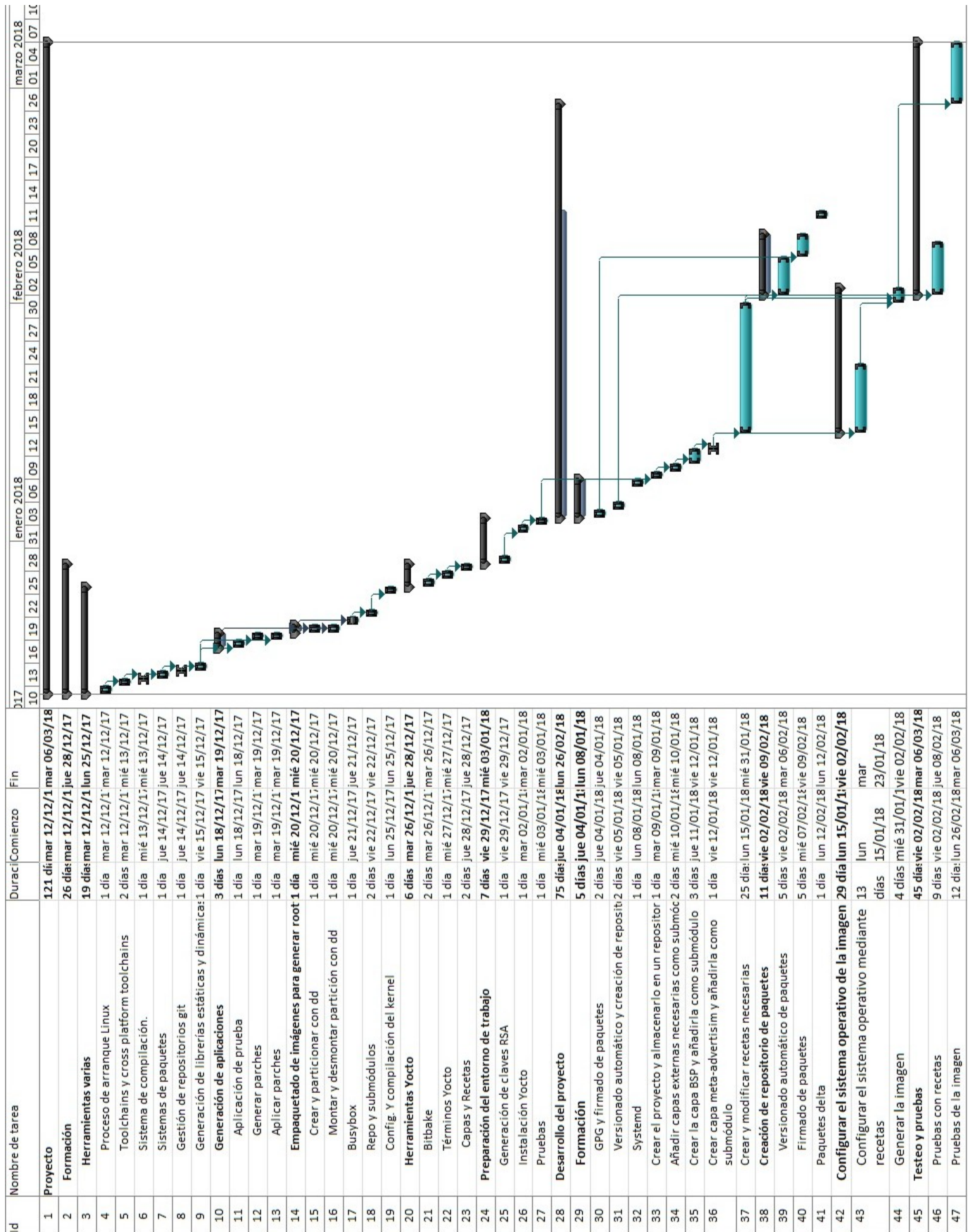


Figura 14. Diagrama de Gantt con la planificación definitiva del proyecto.

3.3 Diferencias entre la planificación inicial y la final

Durante la realización del proyecto se ha intentado seguir la línea que se marcó en un principio con la planificación inicial. Algunas tareas han surgido durante el desarrollo del proyecto, es por eso que no aparecen en la planificación inicial.

Durante el periodo de formación previo al desarrollo del proyecto se establecieron una serie de conocimientos que debía adquirir antes de empezar a desarrollarlo. Algunos de estos conocimientos han sido:

- Generación de librerías estáticas y dinámicas. Algunas de las recetas se crearon para compilar software que utilizaba librerías estáticas, por lo tanto se tuvo que hacer formación al respecto.
- Generación de aplicaciones para poder crear y aplicar parches. En muchas de las recetas creadas se aplican parches de forma automática. Para entender ese automatismo se tuvieron que adquirir conocimientos sobre cómo crear parches de aplicaciones (*diff*) y la forma en que se aplican (*patch*).
- Repo y submódulos de *git*. Como se ha comentado anteriormente, se tuvo que hacer una comparación entre estas dos herramientas. Para hacer esta comparación se tuvo que emplear tiempo en formación sobre ellas que no estaba planificado. En la planificación inicial se puede ver que en un principio se pretendía utilizar *repo*, pero tras la comparación de ambas herramientas se utilizó submódulos de *git* como se puede ver en la planificación final.
- Configuración y compilación del núcleo de Linux. Puesto que hay recetas que permiten configurar el núcleo de Linux, se dedicó tiempo en aprender cómo se hace dicha configuración.

El tiempo invertido en la preparación del entorno de trabajo tampoco se tuvo en cuenta a la hora de hacer la planificación inicial. Como se puede observar en la planificación final, se le han dedicado 7 horas.

Durante el desarrollo de trabajo es donde más ha variado en el tiempo estimado. Algunas de las tareas que han sido añadidas a este periodo del proyecto son:

- En un principio se pensó añadir recetas para *qt5*, *golang*, *gstreamer* e *iptables*. Durante el desarrollo del proyecto se ha visto la necesidad de añadir otras muchas más (recetas para *systemd*, *dnf*, recetas para las librerías utilizadas por las aplicaciones del proyecto Advertisim, *advertisim-viewer, advertisim-daemon*, etc). Algunas de ellas han sido fáciles de añadir puesto que ya estaban creadas en capas de Yocto y otras han sido creadas de forma manual.
- Otra de las variaciones en la planificación se debe a la configuración previa que se le ha hecho al sistema operativo (tarea 4 de la tabla 3). En un principio no se había contado con esta tarea, ha ido surgiendo conforme avanzaban el proyecto y las necesidades de la empresa.
- También durante esta fase del proyecto se pensó montar un repositorio de paquetes propio que fuera accesible desde los dispositivos. Es una tarea que también ha sido incorporada y en la que se ha dedicado tiempo de documentación y desarrollo.
- Versionado automático de los paquetes que genera Yocto. Se ha incorporado para que las versiones de los paquetes creados por Yocto dependan de si ha sido modificada o

no la receta que los genera y del *tag* que haya en el repositorio *git*.

- *Gpg* y firmado de paquetes. Esta tarea se añadió como sugerencia de la empresa ya que así pueden saber si un paquete en concreto ha sido firmado con la clave de la empresa. Esta tarea también ha necesitado de un periodo de documentación y desarrollo.
- Tareas de investigación y documentación. Al utilizar herramientas que en un principio no se tuvieron en cuenta se ha debido dedicar más tiempo a tareas de documentación e investigación (1.1.6, 1.1.7, 1.1.10 y 1.1.11 de la Tabla 3).
- Paquetes *delta*: Esta tarea se empezó a desarrollar pero se abandonó finalmente. En el apartado 7.2 se explicará qué son los paquetes delta y el motivo por el cual no se utilizaron.

Como puede observarse al comparar Tabla 2 y Tabla 3 también se ve un desvío del número de horas dedicadas a testeo y pruebas. En el desarrollo del proyecto se dedicó a esta fase 50 horas en lugar de las 9 previstas inicialmente para asegurar el correcto funcionamiento del proyecto.

3.4 Costes del proyecto

En este apartado se va a especificar el coste monetario aproximado que ha tenido el desarrollo de este proyecto:

- Mano de obra: Referido al tiempo empleado por el estudiante durante el desarrollo del proyecto. 1616 €/mes / 160 horas/mes = 10,1 €/hora [10].
- Máquina Yocto: Coste del alquiler de la máquina virtual utilizada para compilar y generar las imágenes. Precio obtenido de la tarifa de Google [11].
- Hardware: Coste del hardware. Va incluido el dispositivo *Iwill*, tarjeta wifi usb, tarjeta 3G usb y cables de conexión.
- Todo ello asciende a un total de: 3030 + 269,82 + 270 = 4139,82 €

En la Tabla 4 puede verse el desglose del coste total del proyecto.

Tipo	Cantidad	Precio	Total
Mano de obra	300 horas	10,1 € / hora	3030 €
Máquina Yocto (Google)	3 meses	89,94 € / mes	269,82 €
Hardware	1 pack	270 € / pack	270 €
		Total	**3569,82 €**

Tabla 4. Coste aproximado del proyecto.

Capítulo 4

Análisis, requisitos y diseño del sistema

En este capítulo se va a hacer un análisis del sistema y a especificar cuáles han sido los requisitos que se han seguido para desarrollar el proyecto.

4.1 Análisis del sistema

El dispositivo Advertisim, como se ha podido observar en la Figura 3, es un sistema embebido que cuenta con salida de vídeo (VGA y HDMI), salida y entrada de audio, puertos USB, puerto serie y ethernet. A través del puerto USB se podrá conectar una tarjeta de red wifi que actuará como cliente para conectarse a una red inalámbrica y/o un módem 4G para disponer de conexión a internet. Al contar con un puerto ethernet también da la posibilidad de conectarse a una red cableada.

Este dispositivo se utilizará para reproducir todo tipo de contenido multimedia. Este contenido estará alojado en un servidor de la red corporativa de la empresa, con lo cual tendrá que ser transmitido a través de una VPN. Dicho contenido será gestionado por los clientes a través de una plataforma online llamada *Advertisim Manager.*

4.2 Requisitos del sistema

A continuación se expondrán una serie de requisitos que se han tenido en cuenta a lo largo del proyecto. Algunos de ellos han surgido durante el desarrollo.

- Hardware: El sistema operativo resultante de este proyecto debe poder ejecutarse sin problemas sobre un dispositivo con las siguientes características o superiores:
  - Placa base ITX-N29.
  - Procesador Intel celeron 2.0 Ghz.
  - 4GB de Memoria RAM.
  - 120 GB de disco duro.
- Utilización de paquetes *rpm*: El sistema ha de ser capaz de trabajar con los paquetes *rpm*. Esto incluye poder instalar, visualizar, eliminar y modificar este tipo de paquetes.
- Utilizar un repositorio de paquetes *rpm* personalizado: Este requisito hace referencia a la configuración de la imagen para que utilice como repositorio principal uno propio de la empresa donde ellos mismos colgarán los paquetes que crean necesarios.
- Utilización de *systemd*: Utilizar como gestor del sistema y servicios *systemd* en lugar de los scripts de inicio *sysv*. Con *systemd* se consigue ahorrar tiempo en el arranque

del sistema operativo y mayor control a la hora de gestionar los servicios del sistema.
- Versionado automático de paquetes: Todos los paquetes deben llevar automáticamente su número de versión incluido en el nombre y asignado por Yocto. Además de ello el número de *release* del paquete en cuestión cambiará según se hagan cambios en la receta.
- Versionado de los paquetes propios: Los paquetes propios de la empresa deben llevar en el nombre del paquete la versión especificada en el *git* (*tag* de *git*).
- Paquetes firmados: Todos los paquetes que sean generados por Yocto deberán estar firmados con la clave RSA de la empresa.
- Reproducción de contenido multimedia: El sistema operativo ha de ser capaz de reproducir contenido multimedia. Ello implica que pueda mostrar vídeo y audio por las correspondientes salidas.
- Conexión a la red: Deberá disponer de conexión a la red para poder recibir el contenido multimedia.

4.3 Diseño del sistema

En esta memoria no ha sido necesaria la fase de diseño del proyecto, se ha tratado de desarrollo de código interno y no ha sido necesaria la creación de interfaces gráficas. El diseño del sistema ya estaba hecho por parte de la empresa, por lo que no ha sido necesario tomar ninguna decisión al respecto.

Capítulo 5

Desarrollo

En este capítulo se van a describir las tareas más significativas llevadas a cabo durante el desarrollo del proyecto así como los pasos a realizar para definir su correcto funcionamiento.

De forma general las tareas necesarias para lograr el desarrollo del proyecto son las siguientes:

- Preparar el entorno Yocto.
- Añadir la estructura de directorios del entorno Yocto a un sistema de control de versiones como puede ser git y alojarlo en un repositorio.
- Crear y añadir la capa BSP específica para la arquitectura en cuestión.
- Añadir las capas que ofrece Yocto que sean necesarias.
- Crear una capa donde añadir las recetas creadas y/o modificadas.
- Crear y/o modificar las recetas necesarias para poder incluir los paquetes deseados.
- Generar la imagen.
- Crear el repositorio de paquetes.

Cabe mencionar dos libros que han sido utilizados durante todo el desarrollo del proyecto para distintas tareas: *Embedded Linux Projects Using Yocto Project Cookbook* [12] y *Embedded linux development with Yocto project* [13].

5.1 Documentación y estudio

La primera fase del proyecto se dedicó a documentación y estudio de diversas herramientas. Todas estas herramientas han sido descritas en el apartado 2.5.

También cabe destacar que esta fase del desarrollo ha estado presente durante todo el proyecto ya que se añadieron herramientas nuevas y requisitos nuevos. Esto obligó a estudiar dichas herramientas así como a encontrar la manera de solucionar o conseguir cumplir con estos requisitos.

5.2 Instalación de Yocto

En esta fase se pretende poner en marcha el entorno de trabajo de Yocto. Esto se hizo siguiendo el manual de Yocto [14]. Este entorno se instaló en la máquina Yocto citada en el apartado 2.4.

Los pasos seguidos para ello son:

1) Ejecución del comando mostrado en la siguiente línea para instalar en la máquina Yocto toda la lista de paquetes necesarios para preparar el entorno de trabajo de Yocto:

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib build-essential chrpath socat cpio python python3 python3-pip python3-pexpect xz-utils iputils-ping libsdl1.2-dev sterm debianutils
```

2) Conseguir una copia del repositorio *poky*. Para ello se utiliza la herramienta "*git*" ejecutando el siguiente comando:

```
$ git clone git://git.yoctoproject.org/poky
```

3) Crear una rama *git* local. Esto se consigue ejecutando el siguiente comando:

```
$ git checkout -b rocko*
```

4) Añadir un repositorio remoto para poder guardar el proyecto en él. Con esto se consigue poder alojar el proyecto en un repositorio remoto. Para ello se utiliza el siguiente comando:

```
$ git remote add [nombre_repo] [url_repo]
```

5.3 Añadir las capas necesarias como submódulos

Durante esta fase del proyecto se han añadido las distintas capas que se utilizarán a lo largo del desarrollo. Estas capas son *meta-oe, meta-intel, meta-iot-cloud y meta-qt5*.

Esta fase se ha mezclado cronológicamente con la siguiente (descrita en el apartado 5.4) ya que a medida que se iba necesitando añadir recetas, se añadían las capas correspondientes que las incluían.

A continuación se detallarán los pasos seguidos para añadir la capa *meta-intel* como ejemplo.

1) Visitar el repositorio de capas de *OpenEmbedded*. En la Figura 15 se puede ver la información de la capa *meta-intel* que aparece en dicho repositorio. Incluye una breve

---

* En el momento de la redacción de esta memoria (mayo 2018), la versión del proyecto Yocto era la 2.4.2 (Rocko), posiblemente en un futuro cambie esta versión, por lo cual el comando especificado en este paso no será el mismo.

descripción de la misma, el tipo de capa que es (*BSP*) y la dirección del repositorio *git* utilizada para clonar su contenido. Para poder clonarla es necesario copiar la *url git* que aparece (*git://git.yoctoproject.org/meta-intel*).

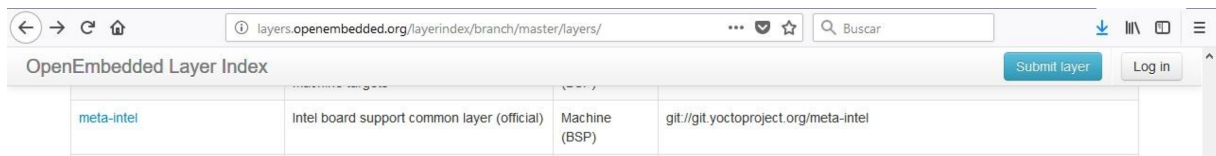


Figura 15. Información sobre la capa *meta-intel*.

2) Una vez copiado el enlace donde está alojada la capa se añade como submódulo con la herramienta *git submodule*. El comando utilizado para añadir la capa *meta-intel* sería:

```
$ git submodule add -b rocko --name meta-intel git://git.yoctoproject.org/meta-intel sources/meta-intel
```

   Con esto se consigue añadir esta capa como un submódulo de *git* al proyecto. Esto se hace para que al clonar el proyecto de este desarrollo desde el repositorio *git*, automáticamente también se clone esta capa. Así, no hace falta hacerlo de forma manual cada vez que se clone el proyecto principal.

3) Ahora solo falta indicarle a Yocto que dicha capa existe y dónde la puede encontrar. Para ello se ha utilizado el comando *bitbake-layers* como se muestra a continuación:

```
$ bitbake-layers add-layer /ruta-al-directorio-yocto/sources/meta-intel
```

4) Hay que asegurarse que la capa ha sido añadida correctamente ejecutando el siguiente comando:

```
$ bitbake-layers show-layers
```

Los pasos citados en esta sección han de seguirse para cada una de las capas externas que se quieran añadir al proyecto.

5.4 Crear, configurar y modificar las recetas necesarias.

Como se ha visto en el capítulo 2, para instalar aplicaciones en la imagen a través de Yocto es necesario crear o modificar primero las recetas necesarias. Algunas de las recetas han sido creadas desde cero y otras que ya existían solo ha sido necesario modificarlas.

Cuando se decide añadir una aplicación a la distribución que se quiere crear lo primero que se debe hacer es comprobar si la receta de esta aplicación ya ha sido creada por algún desarrollador, ya que, si es así, se va a ahorrar trabajo. Para ello se debe consultar el repositorio de capas OpenEmbedded. Si ya existe, hay que clonar la capa en la que esté la receta de la misma manera que se ha explicado en el apartado 5.3 . En el caso en que se quiera modificar alguna receta, se podrá hacer siguiendo el ejemplo que se citará más adelante en el apartado "Modificación de una receta ya creada".

A la hora de crear recetas existen tres opciones:

- Crearlas manualmente desde cero.
- Utilizar el comando *“devtools add”* ofrecido por Yocto.
- Utilizar el comando *“recipetool create”* ofrecido por Yocto.

Tras probar las tres opciones, se decidió descartar la forma manual porque no era la forma más sencilla de trabajar. Entre los dos comandos citados se decidió utilizar *“devtools add”*. El motivo fue porque permite más opciones que *recipetool create*. Se puede, por ejemplo, modificar o borrar una receta, mientras que con *recipetool* solo se pueden crear, para modificarlas se debe hacer a mano.

A continuación se mostrarán los pasos necesarios para crear una receta concreta desde cero, la receta de la librería *spandsp*, partiendo de la *url* donde está alojado el código fuente original de dicha librería.

Creación de una receta desde cero

1) Se necesita saber la *url* donde está alojado el código fuente de la librería en cuestión.

2) Ejecutar el comando *devtools add* junto con el nombre y la *url* donde está almacenado el código fuente de la librería.

   ```
   $ devtools add spandsp https://www.soft-switch.org/downloads/spandsp/spandsp-0.0.6pre21.tgz
   ```

   Con esto la receta será creada en una capa intermedia, llamada *Workspace*, que utiliza Yocto para estos casos. En este estado, la receta podrá ser modificada (cambiar el nombre, modificar la *url* de descarga, etc.)

3) Una vez que se ha comprobado que la configuración de la receta es la adecuada (nombre, *url* de descarga, etc.) hay que ejecutar el siguiente comando:

   ```
   $ devtools finish spandsp <ruta_al_directorio_donde_se/almacenará>
   ```

   siendo `ruta_al_directorio_donde_se_almacenará` la ruta a la capa donde se almacenará la receta.

4) En este punto, el archivo de la receta con extensión *.bb* se habrá creado en la ruta especificada en el paso anterior. Ahora solo quedará modificar la receta para que actúe según convenga. En la Figura 16 se puede observar el código de esta receta.

```
1  #Recipe for lib spandsp
2  LICENSE = "GPLv2 & LGPLv2.1"
3  LIC_FILES_CHKSUM = "file://COPYING;md5=8791c23ddf418deb5be264cffb5fa6bc \
4                      file://debian/copyright;md5=72bfe191182c473bf79b5371353be3bc"
5
6  SRC_URI = "https://www.soft-switch.org/downloads/spandsp/spandsp-0.0.6pre21.tgz \
7      file://rpl_malloc.patch"
8  SRC_URI[md5sum] = "897d839516a6d4edb20397d4757a7ca3"
9  SRC_URI[sha256sum] = "cc053ac67e8ac4bb992f258fd94f275a7872df959f6a87763965feabfdcc9465"
10
11 inherit autotools pkgconfig
12
13 do_configure_append() {
14     # Se borra este fichero porque si no daba problemas a la hora de compilar la receta
15     rm config.log
16 }
17
18 # En el do_install_append se incluyen una serie de ficheros header que Yocto no incluye automáticamente
19 do_install_append() {
20     install -d ${D}${includedir}/spandsp
21     install -m0644 ${WORKDIR}/${P}/spandsp-sim/*.h ${D}${includedir}
22     install -m0644 ${WORKDIR}/${P}/spandsp-sim/spandsp/*.h ${D}${includedir}/spandsp
23 }
24
25 DEPENDS = "tiff"
26
27 CFLAGS_append = "-fPIC -std=gnu99 -DSPANDSP_USE_FIXED_POINT=1"
28 EXTRA_OECONF = "--enable-fixed-point"
```

Figura 16. Receta de la librería *spandsp.*

Se puede dar la situación de que ya exista una receta y solo se necesite modificar el comportamiento de la misma. Para estos casos, el procedimiento a seguir según el estándar de Yocto es algo más sencillo, tal y como puede verse a continuación.

Modificación de una receta ya creada

1) Crear un fichero llamado igual que la receta original pero cuya extensión sea *.bbappend.*

2) Crear la misma ruta de directorios que hay en la capa donde está la receta original en la capa donde queramos guardar la modificación de la receta. Es decir, si la receta original se almacena, por ejemplo, en */ruta-al-directorio-yocto/sources/meta-intel/recipes-connectivity\wpa-supplicant*, la receta que modificará el comportamiento de esta receta original se alojará, por ejemplo, en */ruta-al-directorio-yocto/sources/meta-advertisim/recipes-connectivity/wpa-supplicant.*
   En la Figura 17 se puede ver el ejemplo de la receta *wpa-supplicant.bbappend*, alojada en la capa *meta-advertisim*, que modifica el comportamiento de la receta original *wpa-supplicant*, alojada en la capa *meta-oe*. Como se puede observar, la receta es muy corta ya que *l*as recetas de modificación solo contienen los parámetros de configuración específicos que se desean modificar. El resto de configuración la contiene la receta original.
   Wpa-supplicant es un cliente para poder acceder a redes wifi con protección WPA (Wi-Fi Protected Access) y WPA2. Lo que hace esta receta es configurar la aplicación *wpa-supplicant* para que utilice la librería *openssl* ya que por defecto está configurada para que utilice la librería *gnutls*.

```
1   PACKAGECONFIG ??= "openssl"
```

Figura 17. Receta wpa-supplicant.

A continuación se citan una serie de recetas de librerías y aplicaciones que han sido añadidas durante el desarrollo de este proyecto: *alsa-lib (libasound2), libb64, curl, eudev, libnl, libsndfile, tiff, zlib, libev, libsocketcan, spandsp, mbedtls, mbedx509, mbedcrypto, advertisim-daemon, advertisim-viewer, advertisim-static, bind, n4mc, networkmanager, openssh, ppp, wpa-supplicant, nayar-lang-conf, systemd, dnf, glide, iptables, gstreamer, vpn4mc, qtbase, qtimageformats, qtmultimedia, dfu-programmer, dnsmasq, usbtool, etc.*

Del listado anterior de recetas añadidas algunas han sido creadas desde cero (*.bb*) y otras son modificaciones de recetas ya creadas (*.bbappend*).

El día de la defensa de esta memoria, se podrá consultar el código de todas las recetas creadas y modificadas durante el desarrollo del proyecto. Forman un total de 48 recetas compuestas por 1493 líneas de código. Por acuerdo con la empresa, no se incluyen en esta memoria para garantizar la confidencialidad de dicho código.

5.5 Personalizar la capa *bsp*

Antes de poder personalizar la capa *bsp* hay que clonar desde el repositorio de capas de Yocto la capa *bsp* que más se adapte a la arquitectura para la cual se va a personalizar. En este proyecto, el dispositivo *Iwill* funciona con un procesador Intel, como se ha especificado en el apartado 2.4, por lo tanto, antes de crear la capa *meta-nayar-bsp* se tuvo que clonar la capa *meta-intel*, tal y como se especificó en el apartado 5.3. Esta última capa es oficial y ha sido desarrollada y mantenida por los desarrolladores de Intel.

Para una mayor optimización de la configuración del sistema operativo con el dispositivo embebido donde se quiera instalar, se debe crear y personalizar la capa *bsp*. Como ya se explicó en el apartado 2.5, las capas *bsp* son capas cuyas recetas configuran la información del núcleo y el compilador cruzado para que la imagen funcione correctamente en la plataforma deseada.

Para crear una capa *bsp* se debe ejecutar el siguiente comando:

```
$ yocto-bsp create <bsp-name> <karch>
```

donde `<bsp-name>` será el nombre que se le dará a la capa *bsp* y `<karch>` será la arquitectura del dispositivo. Con ello se ejecuta la opción *create* del script *yocto-bsp*.

La lista de arquitecturas disponibles se puede averiguar con el comando:

```
$ yocto-bsp list karch
```

Al ejecutar el comando anterior en este proyecto las arquitecturas disponibles fueron: *powerpc, x86_64, i386, arm, quemu, mips y mips64*. El resultado de este comando depende de

la capa *bsp* que se esté utilizando, en este caso concreto, al trabajar con la capa *meta-intel*, se pueden ver todas las arquitecturas soportadas por los procesadores Intel. Para este proyecto se ha utilizado la arquitectura *x86_64* ya que el dispositivo *Iwill* utiliza esta arquitectura. Por lo tanto, el comando concreto ejecutado en este proyecto para crear la capa *bsp* fue:

```
$ yocto-bsp create nayar-bsp x86_64
```

Una vez creada y personalizada la capa *bsp*, se debe integrar en el proyecto y en Yocto tal y como se especifica en el apartado 5.3.

En la Figura 18 se presenta un ejemplo de una receta de la capa *meta-nayar-bsp* en la que se puede ver cómo se configura la versión del núcleo Linux que utilizará la imagen y los archivos mediante los cuales se configurará el núcleo.

```
FILESEXTRAPATHS_prepend := "${THISDIR}/${PN}:"

SUMMARY = "Linux kernel for iwill boards"
LINUX_VERSION = "4.14.18"
COMPATIBLE_MACHINE_iwill = "iwill"

inherit kernel

SRC_URI += " \
    file://can.cfg \
    file://wifi.cfg \
    file://audio-codecs.cfg \
"
```

Figura 18. Receta de configuración del núcleo Linux.

5.6 Personalizar la capa *distro*.

En la capa *distro* se modifican aspectos relativos a la distribución Linux personalizada. Estos aspectos pueden ser: scripts de inicio que utilizará la distribución, zona horaria por defecto, formato de paquetes que se utilizará, modificación de la contraseña de root, servidor de administración remota que se utilizará, etc.

```
require conf/distro/poky.conf

DISTRO = "advos"
DISTRO_NAME = "Distro de Nayar Systems para el proyecto Advertisim"
DISTRO_VERSION = "1.0"
DISTRO_FEATURES_append = " alsa systemd wifi"
DISTRO_FEATURES_remove = " x11 wayland pulseaudio vulkan"

# We want a rpm based distro
PACKAGE_CLASSES ?= "package_rpm"

# We want a full featured systemd image
VIRTUAL-RUNTIME_dev_manager = "systemd"
VIRTUAL-RUNTIME_init_manager = "systemd"
VIRTUAL-RUNTIME_initscripts = ""
DISTRO_FEATURES_BACKFILL_CONSIDERED = "sysvinit"
# This trick removes syslog completely (sysvinit)
VIRTUAL-RUNTIME_syslog ?= ""

# Trick to remove busybox completely
VIRTUAL-RUNTIME_base-utils = ""
VIRTUAL-RUNTIME_login_manager = "shadow"

PREFERRED_PROVIDER_ssh = "openssh"
PREFERRED_PROVIDER_sshd = "openssh"

# Use image level user/group configuration to set '1234' as root password.
INHERIT += "extrausers"
EXTRA_USERS_PARAMS = "usermod -P 1234 root;"

# Sane default locales for images
GLIBC_GENERATE_LOCALES ?= "es_ES.UTF-8 en_US.UTF-8"
IMAGE_LINGUAS ?= "es-es en-us"
DEFAULT_TIMEZONE = "Europe/Madrid"
```

Nombre de la distribución

Características a eliminar y añadir a la distribución

Formato de paquetes que se utilizará

Eliminar scripts de inicio sysv y activar systemd

Eliminar busybox

Configurar openssh como servidor SSH

Establecer password al root

Configurar zona horaria y codificación de fuente UTF-8

Figura 19. Receta de configuración de la capa *distro*.

En la Figura 19 se observa una imagen con parte del código del archivo de configuración que se ha utilizado en el proyecto para configurar la capa *distro*. En ella aparecen ejemplos de código para darle un nombre a la distribución, añadir y quitar características a la distribución (wifi, x11, *wayland*, etc), especificar que trabajará con formato de paquetes *rpm*, eliminar *sysv* e implementar *systemd*, eliminar *Busybox*, configurar *openssh* como servidor de administración remota *SSH*, establecer la contraseña del usuario root y configurar la zona horaria y la codificación de las fuentes. Dicho fichero se encuentra en el siguiente directorio: */ruta_al_directorio_yocto/sources/meta-advertisim/conf/distro/advertisim.conf.*

5.7 Personalizar la imagen de inicio (*splash-image*)

La imagen de inicio hace referencia a la imagen que se muestra por pantalla durante el arranque del sistema operativo. En este apartado se describirá el proceso que se siguió para personalizar esta imagen.

Lo que se hizo fue crear un fichero *unit service* de *systemd* que se ejecuta al inicio del sistema operativo y que muestra una imagen a través de un *framebuffer*. Para añadirla a la imagen del sistema operativo se creó el fichero *unit* correspondiente, el cual se copia en el directorio */lib/systemd/system* del dispositivo *Iwill*. Para copiar este fichero, así como para copiar la imagen que se mostrará al inicio, se ha utilizado una receta. En la Figura 20 se puede ver el código del fichero *unit service*.
De forma general, un fichero *unit service* es un fichero que le indica a *systemd* qué tiene que hacer para poder arrancar, parar, chequear, etc. el servicio, aplicación, función, etc. especificado en el nombre de dicho fichero.

```
[Unit]
Description=Splash screen
DefaultDependencies=no
After=local-fs.target
[Service]
ExecStart=/usr/bin/fbi -d /dev/fb0 --noverbose -a /opt/bootimage.jpg
StandardInput=tty
StandardOutput=tty[Install]
WantedBy=sysinit.target
```

Figura 20. ***Unit file*** **que ejecuta la imagen de inicio** ***splash.***

5.8 Añadir los módulos necesarios al núcleo Linux

Los módulos del núcleo de Linux son fragmentos de código que permiten extender la funcionalidad del núcleo. Este concepto viene derivado de que el núcleo es modular, lo que permite añadirle funcionalidad o quitarle según sea necesario.

En esta fase del proyecto se ha modificado el núcleo de Linux para añadir algunos módulos. Con ellos se consigue que la distribución pueda utilizar ciertos chipsets de tarjetas wifi usb, tenga soporte CAN y para incluir algunos códecs de audio.

Para modificar los módulos del núcleo se ha seguido el procedimiento descrito en el manual de Yocto [15]:

1) Primero es conveniente crear una copia de la configuración actual del núcleo y almacenarla en un archivo en el directorio work (ver Figura 7). Para ello se debe ejecutar el siguiente comando:

   ```
   $ bitbake nombre_receta_núcleo -c kernel_configme -f
   ```

2) Acto seguido se ejecuta el siguiente comando:

   ```
   $ bitbake nombre_receta_núcleo -c menuconfig
   ```

   Con ello se abre un menú de configuración del núcleo donde se pueden marcar y desmarcar las opciones necesarias. El menú no es muy intuitivo pero permite utilizar un buscador que facilita la tarea. Una vez se hayan hecho todas las modificaciones oportunas, se debe guardar la configuración desde el menú y se creará automáticamente un nuevo archivo con la configuración del núcleo después de modificarla. En la Figura 21 se muestra un ejemplo del menú de configuración del núcleo Linux.

3) El tercer paso es generar un fichero de configuración del núcleo con la diferencia entre el fichero creado en el paso 1 y el fichero creado en el paso 2. Para ello se debe ejecutar el siguiente comando, el cual detecta automáticamente los ficheros creados en los dos pasos anteriores:

   ```
   $ bitbake nombre_receta_núcleo -c diffconfig
   ```

4) Finalmente se hace referencia al fichero generado en el paso 3 en la variable SRC_URI de la receta de configuración del núcleo Linux, tal y como se muestra en la

Figura 18, para que se aplique la configuración deseada en el núcleo. En dicha figura se puede apreciar cómo en esta variable se incluyen los ficheros *can.cfg*, *wifi.cfg* y *audio-codecs.cfg*. Para poder crear cada uno de estos ficheros se han repetido los pasos citados en este punto.

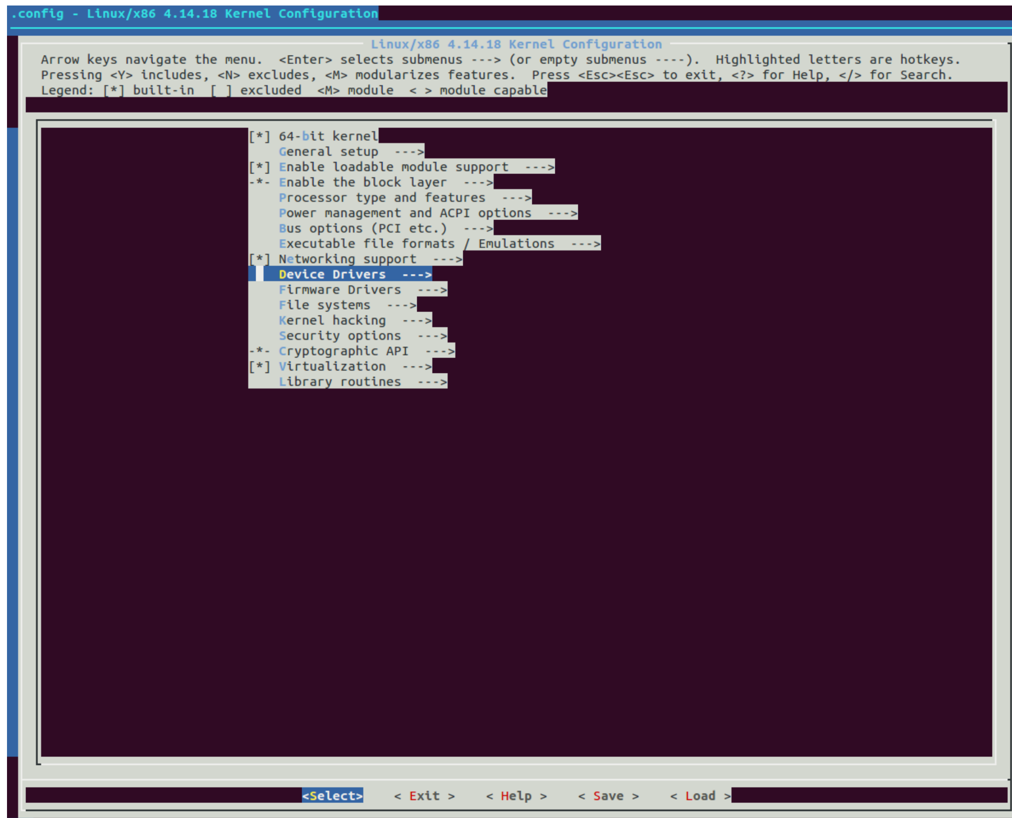


Figura 21. Menú de configuración del núcleo de Linux.

5.9 Crear el repositorio de paquetes

Como se ha comentado en el apartado 3.3, una tarea que surgió durante el desarrollo del proyecto fue montar un repositorio de paquetes. En este repositorio se almacenan todos los paquetes generados por Yocto con la idea de que estén disponibles para poder ser instalados desde cualquier dispositivo que utilice la imagen desarrollada en este proyecto.

Todos estos paquetes (los generados al lanzar a compila la imagen y los generados al compilar paquetes) tienen en común que, una vez generados, se guardan en el directorio */ruta-al-directorio-yocto/build/tmp/deploy/rpm*, como puede verse en la Figura 7. Posteriormente, cuando se lanza la compilación de una imagen con Yocto, se descarga, compila y empaqueta todo el software fuente necesario para poder montar tanto el sistema operativo como el sistema de ficheros que incluirá la imagen. Cuando se lanza a compilar un paquete con Yocto, se descarga su código fuente, se compila y se empaqueta.

Téngase en cuenta que la máquina en la que se compilan y generan todos los paquetes (máquina Yocto) no será la misma que la máquina donde se creará el repositorio de paquetes (nombrada de aquí en adelante máquina *repo_pub*). Se podría hacer en la misma máquina,

pero en el desarrollo de este proyecto se ha utilizado otra máquina distinta. El motivo principal es que no tiene sentido que la máquina Yocto esté en la misma red que los dispositivos que necesitarán poder acceder al repositorio. Por lo tanto, en este caso se conectarán los distintos dispositivos con la máquina *repo_pub* a través de una red VPN.

A continuación se expondrán los pasos que se han seguido para montar dicho repositorio:

1) Crear un directorio en la máquina *repo_pub* para poder copiar en él los paquetes *rpm* generados en la máquina Yocto. Para ello se debe ejecutar el siguiente comando:

```
$ mkdir -p /var/www/yocto/rpm
```

2) Regenerar el índice de paquetes en la máquina Yocto. Con ello se crean o actualizan los metadatos que se utilizarán en el repositorio de paquetes. Estos metadatos contienen la lista de paquetes disponibles para servir en el repositorio e información acerca de ellos. Para ello en la máquina Yocto hay que ejecutar el siguiente comando:

```
$ bitbake package-index
```

Package-index es una receta que permite reconstruir el índice de paquetes de Yocto.

3) Sincronizar el directorio de la máquina Yocto donde están almacenados todos los paquetes *rpm* con el directorio creado en el paso anterior, es decir, copiar todos los paquetes desde la máquina Yocto a la máquina *repo_pub*. El comando a ejecutar será el siguiente:

```
rsync  -avzhie  "ssh"  <yocto-user>@máquina_yocto:/ruta-al-directorio-yocto/build/tmp/deploy/rpm/*    /var/www/yocto/rpm/
```

5.10 Publicar el repositorio de paquetes

Para que el repositorio de paquetes pueda ser accesible desde cualquier dispositivo Advertisim que tenga conexión con *repo_pub*, este debe ser publicado en un servidor web. Para ello es necesario que *repo_pub* disponga del software necesario para montar un servidor web. En el desarrollo de este proyecto se ha utilizado el servidor *Nginx* por su facilidad de uso.

Los pasos para poder publicar el repositorio de paquetes son los siguientes:

1) Instalar el servidor web en *repo_pub*. Para ello se debe ejecutar el siguiente comando:

```
$ apt-get install nginx
```

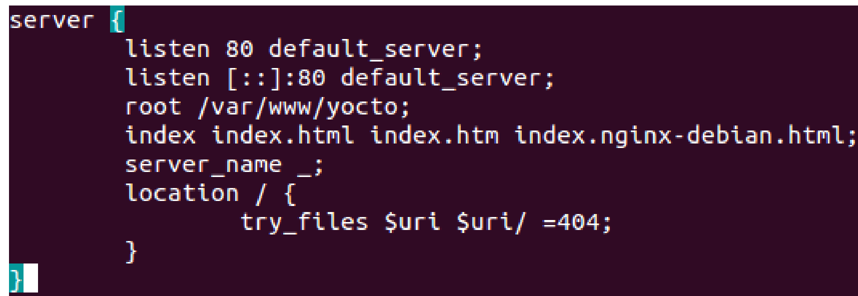
2) Asegurarse que el firewall de *repo_pub* permita conexiones con *Nginx*. Para ello se debe ejecutar el comando:

```
$ ufw allow 'Nginx HTTP (Hypertext Transfer Protocol)'
```

3) En *repo_pub*, copiar la configuración por defecto de *Nginx* para poder modificarla después . Esto se consigue ejecutando el comando:

```
$ cp /etc/nginx/sites-available/default /etc/nginx/sites-available/yocto-repo
```

4) Editar el fichero creado en el paso anterior cambiando dentro del apartado "*server*" la opción "*root*". Se hace así para que Nginx conozca el directorio donde están almacenados los paquetes que servirá el repositorio. El apartado *server* de dicho fichero quedará como se muestra en la Figura 22.



```
server {
        listen 80 default_server;
        listen [::]:80 default_server;
        root /var/www/yocto;
        index index.html index.htm index.nginx-debian.html;
        server_name _;
        location / {
                try_files $uri $uri/ =404;
        }
}
```

Figura 22. Configuración de *Nginx* para publicar el repositorio de paquetes.

5) Averiguar el usuario que ejecuta el servicio *nginx* con el comando:

```
$ cat /etc/nginx/nginx.conf | grep user
```

El usuario utilizado en este proyecto ha sido *www-data.*

6) Cambiar el propietario del directorio indicado en la opción *root* en el paso 4 para que sea el usuario y grupo averiguado en el paso anterior.

```
$ chown -R www-data:www-data  /var/www
```

7) Configurar el bit SGID del directorio citado en el punto anterior de tal forma que todos los ficheros y directorios creados en él posteriormente sean propiedad del grupo *www-data*.

```
$ chmod 2770 /var/www
```

8) Indicar a Nginx que sirva el nuevo sitio creado en el paso 4. Para ello ejecutar el comando:

```
$ ln -s /etc/nginx/sites-available/yocto-repo /etc/nginx/sites-enabled/
```

9) Reiniciar el servidor web. En el caso de *repo_pub*, como utiliza *systemd*, el comando a utilizar para ello sería:

```
$ systemctl reload nginx.service
```

Una vez seguidos estos pasos, el repositorio ya será accesible y cualquier dispositivo con conexión a *repo_pub* que cuente con una herramienta de gestión de paquetes *rpm* podrá instalar los paquetes disponibles en dicho repositorio.

5.11 Automatizar la versión para el nombre de los paquetes.

Esta fase hace referencia al proceso que se ha seguido para conseguir que el número de versión que Yocto le da a los paquetes sea automático y no haya que especificarlo manualmente. Esto se ve reflejado en los paquetes, pues en el nombre de cada uno se especifica su versión. Se ha hecho así en las recetas que fueron creadas desde cero.

En un principio, esto se hacía de forma manual añadiendo al nombre de la receta la versión del paquete siguiendo el formato <nombrereceta_versión>. Al hacer esto, el nombre del paquete generado llevaba el número de versión especificado en el nombre de la receta. Por ejemplo, para una receta llamada "*receta_1.0.0.bb*" se generará un paquete llamado "*receta_1.0.0.rpm*". Tras investigar se descubrió cómo añadir el número de versión al nombre del paquete de forma totalmente automática, es decir, sin tener que especificarlo manualmente en el nombre de la receta. A continuación se expondrán los pasos necesarios para conseguirlo:

1) Configurar las recetas para que, a la hora de descargar el código fuente a compilar desde el repositorio, se descargue siempre la última versión del código que haya sido subida al repositorio, es decir, el código que haya sido subido en el último *commit*.

   1.1) Esto se hace modificando el valor de la variable *SRCREV* en las recetas. En esta variable se especifica el código *hash* de un *commit*, así Yocto siempre descargará el código fuente perteneciente a ese *commit*. Como se puede apreciar en la Figura 23, a esta variable se le ha dado el valor *"${AUTOREV}"*, lo que indicará que se descargue siempre el código fuente del último *commit.* También se puede ver comentado el valor antiguo que tenía esta variable.

   1.2) Además de ello, también se debe especificar en la variable *SRC_URI* la rama del repositorio desde la cual se descargará el código fuente. En la Figura 23 se señala dicha rama en un cuadrado rojo. En este caso, el código fuente se descargará de la rama *master*.

```
SRC_URI = "git://${GO_IMPORT};protocol=http;user=${NAYAR_GIT_USER}:${NAYAR_GIT_PASS};\
               branch=master;destsuffix=${PN}-${PV}/src/${GO_IMPORT}"
#SRCREV = "7e3dff7bfe29ef09bf0d14c6894c51310e017387"
SRCREV = "${AUTOREV}"
```

Figura 23. Código que permite la descarga del último *commit*.

2) Configurar las recetas para que le añadan el número de versión al nombre dependiendo del *tag* del último *commit*. Esto solo se ha hecho sobre las recetas *advertisim-viewer* y *advertisim-daemon* porque son las aplicaciones propias desarrolladas por la empresa y así se puede garantizar que todos los *commits* tienen asociado un *tag* indicando el número de la versión de la aplicación (la empresa lo hace por norma). Lo que se hace en este paso es cambiar el valor de la variable *PKGV*, que es la variable que almacena la versión que se le añadirá al nombre del paquete. En la Figura 24 se puede ver el

código python escrito en la receta de *advertisim_viewer* que consigue el cometido.

```
do_package_prepend () {
        import subprocess
        ese = d.getVar('S') #S es ${S}
        os.chdir(ese)
        versionpropia = subprocess.getoutput('git describe')
        d.setVar('PKGV', versionpropia)
}
```

Figura 24. Código que extrae el número de versión del *tag*.

3) Activar el servidor PR (*Packet Revision*). Esto permitirá aumentar el número de revisión de un paquete. Cada vez que se genera un paquete con Yocto y está activado este servidor, se comprueba si la receta ha sido modificada desde la última vez que se creó este paquete y, si es así, se incrementa el número de la revisión. Este número va especificado en el nombre del paquete como el número de versión. Para hacer funcionar este servidor hay que añadir en el fichero de configuración de la capa *distro, comentado* en el apartado 5.6, la siguiente línea:
   PRSERV_HOST = "localhost:0"
   Con ello se indica a Yocto que la propia máquina Yocto (*localhost*) hará de servidor PR.

5.12 Firmado de Paquetes

Uno de los requisitos que surgieron durante el desarrollo del proyecto fue el firmado de los paquetes. La empresa lo propuso para asegurarse que todos los paquetes que instalaran cuando la imagen estuviera ya en producción fueran paquetes generados por ellos mismos o por Yocto.

Para firmar los paquetes es necesario disponer de un par de claves (pública y privada) RSA generadas con GPG. La clave privada será utilizada para firmar los paquetes y la pública para comprobar la firma de los mismos.

Los pasos seguidos fueron los siguientes, tal y como se indica en el libro "*Embedded Linux Development Using Yocto Project Cookbook*" [16]:

1) Crear las claves.
   1.1) Instalar la herramienta *GnuPG*. Para ello ejecutar el comando:

   ```
   $ apt-get install gnupg
   ```

   1.2) Una vez instalada la herramienta ya se pueden crear las claves y para ello se debe de ejecutar el comando:

   ```
   $ gpg —gen-key
   ```

La ejecución de este comando abrirá un menú interactivo en el que se solicitarán los siguientes datos:

<*real_name*>*:* Identificador para el par de claves.
<*email*>*:* Dirección de correo electrónico.
<*passphrase*>: Contraseña para proteger la clave privada.

2) Integración de las claves con Yocto.

2.1) Añadir las siguientes líneas al fichero de configuración de la capa *distro,* como se ha visto en el apartado 5.6:

INHERIT += "sign_rpm"
RPM_GPG_NAME = "<real_name>"
RPM_GPG_PASSPHRASE = "<passphrase>"

Se puede observar en las tres líneas de código expuestas en este punto que para la firma de paquetes es necesario heredar de la clase *sign_rpm*, indicar en la variable *RPM_GPG_NAME* el identificador que se le ha dado en el paso anterior a las claves y especificar en la variable *RPM_GPG_PASSPHRASE* la contraseña que protege a la clave privada.

2.2) Ejecutar el comando:

```
$ bitbake package-index -c cleanall
```

Con este comando se borra el estado de *package-index* y todos los ficheros que se han podido generar si en alguna ocasión se ha ejecutado el comando "*bitbake package-index*".

2.3) Finalmente, se debe regenerar el índice de paquetes. Además se revisarán todos los paquetes y los que no estén firmados serán firmados. Para ello hay que ejecutar el siguiente comando:

```
$ bitbake package-index
```

5.13 Configurar el uso del repositorio de paquetes

Como se ha comentado en el capítulo 2, *dnf* es la herramienta de gestión de paquetes para sistemas Unix que utilizan el formato de paquetes *rpm*. Esta herramienta será incluida en la imagen y, además, se debe configurar para permitir el acceso al repositorio de paquetes de la empresa.

En los apartados 5.9 y 5.10 se ha visto cómo crear y publicar un repositorio de paquetes. Para que los dispositivos en los que se instale la imagen resultante de este proyecto sean capaces de utilizar dicho repositorio y comprobar la firma de los paquetes, se debe dejar configurada la herramienta *dnf*. Para ello, hay que modificar la configuración de la

distribución tal y como se ha indicado en el apartado 5.6 e indicarle la *url* donde se alojará el repositorio de paquetes.

La configuración de repositorios de esta herramienta reside en el fichero */etc/yum.repos.d*. Al modificar la configuración de la capa *distro*, más concretamente al añadirle la variable "*PACKAGE_FEED_URIS*" al fichero */ruta-al-directorio-yocto/sources/meta-capa-distro/conf/archivo_configuracion_capa_distro.conf*, se indica a Yocto cómo configurar dicha herramienta. En este caso en concreto, al fichero de configuración de la capa *distro* se le ha añadido el código mostrado en la Figura 25.

```
# PACKAGE_FEED variables
INHERIT += "sign_package_feed"
PACKAGE_FEED_URIS = "<url_repo_pub>"
PACKAGE_FEED_GPG_NAME = "<real_name>"
PACKAGE_FEED_GPG_PASSPHRASE_FILE = "<url_fichero_contraseña>"
```

Figura 25. Configuración de acceso al repositorio.

A continuación se detallarán las variables que aparecen en dicha figura:

- INHERIT: como se ha comentado en apartados anteriores se utiliza para heredar la funcionalidad de una clase. En este caso en concreto se hereda la funcionalidad de la clase *sign_package_feed*.
- PACKAGE_FEED_URIS: Con esta variable se proporciona a Yocto la *url* donde está publicado el repositorio de paquetes.
- PACKAGE_FEED_GPG_NAME: Se utiliza para indicar a Yocto el identificador que se le ha dado a las claves (*real_name*) creadas en el apartado 5.12.
- PACKAGE_FEED_GPG_PASSPHRASE_FILE: Variable utilizada para indicar a Yocto dónde encontrar un fichero cuyo contenido sea la contraseña de la clave privada. Este fichero se debe tratar con cuidado, pues es inseguro guardar la contraseña de las claves en un fichero de texto plano. Debería ser protegido aplicándole los adecuados permisos.

El contenido de */etc/yum.repos.d* permite configurar la herramienta de gestión de paquetes, en este caso dnf. El contenido de este fichero, generado por Yocto tras modificar el fichero de configuración de la capa *distro*, será el siguiente:

```
[oe-remote-repo]
name=OE Remote Repo: <nombre_imagen>_repo
baseurl=<url_repo_pub>
repo_gpgcheck=1
gpgkey=file://etc/pki/packagefeed-gpg/PACKAGEFEED_GPG_KEY
```

A continuación se explica con detalle el contenido de este fichero:

- En la primera línea se indica, entre corchetes, el nombre con el que la herramienta de gestión de paquetes reconocerá al repositorio.
- La variable *name* se utiliza para nombrar al repositorio. Este nombre será el que mostrará la herramienta de gestión de paquetes cuando se le pregunte la lista de repositorios disponibles como se verá en el apartado 6.4.
- La variable *baseurl* especifica la *url* donde se aloja el repositorio de paquetes.

- La variable *repo_gpgcheck* con valor 1 indica que se debe comprobar la firma de todos los paquetes a instalar desde este repositorio.
- La variable *gpgkey* especifica la ruta donde se almacena la clave pública que se utilizará para comprobar la firma de los paquetes.

5.14 Generar las t*oolchains* externas

Una *toolchain* es un conjunto de herramientas que se utilizan para desarrollar y crear un producto software complejo. Todas las *toolchain* (al menos las basadas en el GNU-gcc) incluyen un compilador, un *linker* para enlazar las librerías, librerías y un *debugger* para probar el código. Al integrarlas en Yocto se permite crear recetas para software escrito en un lenguaje determinado y poder así generar los paquetes para añadir dicho software a la imagen. Gracias a estas *toolchains,* Yocto genera los binarios compatibles con la plataforma destino, en este caso el dispositivo *Iwill,* es decir, son compilados en la máquina Yocto pero válidos para el dispositivo *Iwill,* aunque sean plataformas distintas (compilación cruzada*)*. Yocto provee algunas *toolchains* externas, de las cuales durante el desarrollo de este proyecto se han necesitado integrar tres:

- La *toolchain* de *Go*: Utilizada para compilar y generar binarios de software escrito en lenguaje *Go*.
- La *toolchain* para *C*: Utilizada para compilar y generar binarios de software escrito en lenguaje *C*.
- La *toolchain* para *QT5*: Utilizada para compilar y generar binarios de software escrito en lenguaje *QT (*ver apartado 2.1).

A continuación se describirán los pasos seguidos para instalar la *toolchain* de *C.* Para instalar el resto de *toolchains* el procedimiento es similar.

1) Ejecutar el siguiente comando genérico:

```
$ bitbake <nombre_receta_toolchain>
```

Para este caso en concreto se ejecutó:

```
$ bitbake meta-toolchain
```

Yocto generará un archivo .sh que se utilizará para integrar la *toolchain* en Yocto y la almacenará en *yocto/build/tmp/deploy/sdk/.*

2) Ejecutar el archivo *.sh* generado en el paso anterior para instalar la *toolchain.* En este caso:

```
$ ./yocto/build/tmp/deploy/sdk/advos-glibc-x86_64-meta-toolchain-
corei7-64-toolchain-2.4.2.sh
```

3) Una vez instalada, ya se podrá cargar el entorno de compilación cuando sea necesario. Para ello se ejecuta:

```
$ ./opt/advos/2.4.2/environment-setup-corei7-64-poky-linux
```

Capítulo 6

Pruebas

Durante el desarrollo del proyecto se han hecho varias pruebas para comprobar que el funcionamiento de las diferentes tareas era el esperado. A continuación se expondrán algunas de estas pruebas.

6.1 Arrancar el dispositivo *Iwill* con la imagen generada

Esta ha sido la principal prueba y la que más veces se ha hecho durante el desarrollo del proyecto. Se trata de volcar la imagen generada con Yocto en un dispositivo de almacenamiento y arrancar el sistema desde él. En un principio como dispositivo de almacenamiento se utilizaba una memoria usb, hasta que la empresa adquirió el adaptador disco duro sata/ssd a usb 3.0 comentado en el apartado 2.4 de esta memoria para poder volcar la imagen directamente al disco duro ssd del dispositivo *Iwill*.

Para conseguir arrancar la imagen desde el disco duro se seguían los siguientes pasos:

1) Generar la imagen con el siguiente comando:

```
$ bitbake <nombre_de_la_receta_de_la_imagen>
```

2) Copiar la imagen generada en la máquina Yocto a un ordenador local. Para ello utilizábamos el comando *scp*, que sirve para hacer copias entre ordenadores remotos sobre el protocolo *SSH*. Un ejemplo de utilización de este comando es el siguiente:

```
$ scp <usuario_remoto>@<ip_pc_remoto>:/ruta/origen   /ruta/destino
```

3) Desmontar el disco duro del dispositivo *Iwill* y montarlo sobre el adaptador disco duro sata/ssd a usb 3.0.

4) Limpiar la tabla de particiones del disco duro con el siguiente comando:

```
$ sudo dd if=/dev/zero of=/dev/<dispositivo> bs=1M count=512
```

5) Volcar la imagen en el disco duro con el siguiente comando:

```
$ sudo dd if=nombre_imagen of=/dev/dispositivo
```

6) Desmontar el disco duro del adaptador disco duro sata/ssd a usb 3.0 y volver a

montarlo sobre el dispositivo *Iwill*.

7) Arrancar el dispositivo Iwill desde su disco duro.

6.2 Audio y vídeo

Uno de los requisitos del proyecto es que el dispositivo *Iwill* sea capaz de ejecutar contenido multimedia. Esto lleva a que se deba probar tanto el audio como el vídeo. Se hicieron pruebas que sirvieron para determinar que el dispositivo *Iwill* es capaz de reproducir vídeo tanto por la salida HDMI como por la VGA, y que también es capaz de reproducir audio tanto por HDMI como a través del jack 3,5mm.

6.3 Aceleración gráfica hardware

Para probar que la aceleración gráfica por hardware en el dispositivo *Iwill* funciona correctamente se ha utilizado la herramienta *qmlscene*. Esta herramienta permite ejecutar archivos *qml*, que son archivos escritos en lenguaje QML. Este lenguaje se utiliza sobre todo para diseñar aplicaciones con interfaces gráficas.

Los pasos seguidos para probar la aceleración por hardware después de crear el fichero *qml* han sido los siguientes:

1) Instalar el paquete *qtdeclarative-tools* en la imagen. Esto se puede hacer añadiendo en la variable *IMAGE_INSTALL* de la receta de la imagen dicho paquete.

2) Arrancar la imagen generada en el dispositivo *Iwill* tal y como se ha descrito en el apartado 6.1.

3) Crear el fichero *qml* que se utilizará para hacer la comprobación. En la Figura 26 se puede observar el código del fichero *qml* utilizado para hacer la prueba, denominado prueba.qml. Al ejecutar este fichero con *qmlscene* desde el dispositivo *Iwill* se abre un navegador web mostrando la dirección especificada, en este caso *http://www.google.es.*

```
$ cat prueba.qml

import QtQuick 2.0
import QtQuick.Window 2.0
import QtWebEngine 1.0
Window {
    width: 1024
    height: 750
    visible: true
    WebEngineView {
        anchors.fill: parent
        url: "http://www.google.es"
    }
}
```

Figura 26. Código *qml* utilizado para probar la aceleración hardware.

4) Ejecutar en el dispositivo *Iwill* el fichero *qml* creado en el paso anterior. Si se abre y se ve bien la página web se puede dar por correcto el funcionamiento de la aceleración gráfica por hardware. Para poder ejecutarlo se debe lanzar el siguiente comando:

```
$ qmlscene prueba.qml -platform eglfs
```

También se repitió la misma prueba apuntando a las direcciones web *http://helloracer.com/webgl/* y *https://www.webdesignerdepot.com/2014/05/8-simple-css3-transitions-that-will-wow-your-users/* en lugar de a la de Google. Estas *url* muestran animaciones gráficas realizadas con suavidad. Mientras eran mostradas por pantalla se comprobó que la carga de la CPU era inferior al 3%, lo que permite verificar el correcto funcionamiento de la aceleración gráfica por hardware. Para comprobar la carga de la CPU se ha utilizado la herramienta *htop*.

6.4 Creación del repositorio y *dnf*

También se tuvo que investigar y hacer pruebas para poner en marcha el repositorio de paquetes y utilizar la herramienta *dnf* para gestionarlo.

La parte del repositorio se dividió en dos pruebas:

- Montar el repositorio en una máquina local y probarlo desde ella. Esta tarea se hizo inicialmente utilizando el comando *createrepo* descrito en el apartado 2.5. Sin embargo, más adelante se vio que no hacía falta utilizar este comando para crear un repositorio puesto que Yocto genera los paquetes indexados y listos para ser servidos en un repositorio.
- La otra fue publicarlo en una máquina remota (*repo_pub*) y poder usarlo desde un dispositivo *Iwill*.

A continuación se expone una serie de pruebas que se hicieron con *dnf* sobre el repositorio y que permitieron decidir que finalmente se instalaría esta herramienta en la imagen como gestor de paquetes:

- Bajar la versión de un paquete:

```
$ dnf  downgrade <nombre_paquete>
```

- Listar todas las versiones de un paquete disponibles en el repositorio:

```
$ dnf --showduplicates list <nombre_paquete>
```

- Listar todos los repositorios existentes para dnf:

```
$ dnf repolist all
```

Esto permitió averiguar cuando se configuró dnf si realmente se añadía el repositorio personalizado o no.

- Instalar un paquete:

  ```
  $ dnf install <nombre_paquete>
  ```

  Además, de esta forma se puede forzar a instalar una versión en concreto de un paquete.

- Desinstalar un paquete instalado:

  ```
  $ dnf remove <nombre_paquete>
  ```

- Actualizar un paquete:

  ```
  $ dnf update <nombre_paquete>
  ```

- Actualizar todos los paquetes del sistema.

  ```
  $ dnf upgrade
  ```

- Borrar la caché dnf:

  ```
  $ dnf clean all
  ```

  Esta no era una prueba programada, sino más bien una necesidad, ya que al añadir paquetes nuevos al repositorio y actualizar los índices, hasta que no se borra la caché dnf no es capaz de verlos.

Además de las pruebas mencionadas, cuando se consiguió firmar los paquetes se hicieron algunas más. Desde el dispositivo Iwill, con la opción repo_gpgcheck del fichero /etc/yum.repos.d activada tal y como se puede ver en el apartado 5.13, pero, sin importar la clave pública, se llevaron a cabo las siguientes pruebas:

- Añadir un paquete sin firmar al repositorio y probar a instalarlo con dnf. El resultado fue que no se pudo instalar porque al activar repo_gpgcheck, todos los paquetes que se instalan desde ese repositorio deben ir firmados.
- Añadir un paquete firmado al repositorio y probar a instalarlo con dnf. El resultado fu negativo también porque no se había importado al dispositivo Iwill la clave pública.
- Importar la clave pública al dispositivo Iwill con el comando:

  ```
  $ rpm --import <clave_publica>
  ```

  En este estado, tras probar a instalarlo de nuevo, el resultado fue positivo. El paquete está firmado y, además, la clave pública pareja de la clave privada con la cual se firmó el paquete está presente en el dispositivo Iwill.

6.5 GPG y firmado de paquetes

Según el apartado de requisitos, todos los paquetes generados por Yocto deben estar firmados con la clave privada de la empresa. Para conseguir esta tarea se tuvo que emplear

tiempo en investigar sobre las claves *gpg* y cómo firmar paquetes con Yocto.

Como se ha comentado en el apartado 5.11, para crear el par de claves pública y privada se utilizó la herramienta *GnuPG*. Sobre este aspecto se estuvieron haciendo las siguientes pruebas de uso de esta herramienta:

- Generar el par de claves:

```
$ gpg –gen-key
```

- Listar las claves disponibles:

```
$ gpg –list-keys
```

- Hacer una copia de seguridad de las claves:
  - Clave privada:

    ```
    $ gpg --output <fichero_backup.prv> --armor –export-secret-key
    ```

  - Clave pública:

    ```
    $ gpg --output <fichero_backup.pub> --armor --export <id_clave_publica>
    ```

- Restaurar las claves:
  - Clave privada:

    ```
    $ gpg --allow-secret-key-import --import <fichero-backup.prv>
    ```

  - Clave pública:

    ```
    $ gpg --import <nombre_fichero_backup.pub>
    ```

Una vez se consiguió generar las claves, estas se utilizaron para firmar todos los paquetes generados. Para lograr esto se siguieron los pasos mencionados en el apartado 5.11. Una vez se consiguió que Yocto firmase los paquetes, para comprobar que funcionaba correctamente se analizó la información sobre dichos paquetes. En la Figura 27, se muestra una imagen en la que se puede apreciar la comprobación llevada a cabo en uno de los paquetes. Se ha encuadrado en color rojo la línea que demuestra que el paquete está firmado. Si no lo estuviera, la característica “signature” aparecería como “(none)”. Como se puede observar en la Figura 27, esta comprobación se hacía ejecutando el siguiente comando:

```
$ rpm -qpi <nombre_paquete>
```

```
cmollon@yocto:~/yocto/build-intel/tmp/deploy/rpm/corei7_64$ rpm -qpi calculadora-1.0-r0.8.corei7_64.rpm
Name        : calculadora
Version     : 1.0
Release     : r0.8
Architecture: corei7_64
Install Date: (not installed)
Group       : base
Size        : 31477598
License     : CLOSED
Signature   : RSA/SHA256, mar 27 mar 2018 15:17:47 CEST, Key ID 76d169f54af44f90
Source RPM  : calculadora-1.0-r0.8.src.rpm
Build Date  : mar 27 mar 2018 15:17:44 CEST
Build Host  : yocto.c.net4machines.internal
Relocations : (not relocatable)
Packager    : Poky <poky@yoctoproject.org>
Summary     : calculadora version 1.0-r0
Description :
calculadora version 1.0-r0.
cmollon@yocto:~/yocto/build-intel/tmp/deploy/rpm/corei7_64$
```

Figura 27. Comprobación de la firma de un paquete.

Capítulo 7

Problemas y soluciones

En este capítulo se expondrán algunos de los problemas que han surgido durante el desarrollo del proyecto y qué soluciones se aplicaron para resolverlos. Cabe mencionar que algunos de los problemas surgidos son debidos a falta de actualizaciones de software. Es probable que sean resueltos más adelante por parte de los desarrolladores.

7.1 Arranque de la imagen

Cuando se probó la primera imagen generada por Yocto ocurrió el primero de los problemas. Al tratar de arrancar dicha imagen desde una memoria USB en el dispositivo Iwill, se quedaba la pantalla en negro y con un cursor en la esquina superior izquierda parpadeando. Tras investigar sobre el caso en internet y comentarlo con compañeros de la empresa se llegó a la conclusión de que el error lo estaba generando el APIC (Advanced Programmable Interrupt Controller). Esta es una característica utilizada en los microprocesadores más modernos que se encarga de generar y asignar las interrupciones a los diferentes dispositivos del sistema.

Para solucionar el tema se incorporó a la línea de comandos que configuran el núcleo del sistema operativo durante el arranque la opción noapic. Lo que hace esta función es deshabilitar el APIC, con lo cual las interrupciones son gestionadas por el PIC (Programmable Interrupt Controller), que es la versión anterior al APIC. Se hizo la prueba añadiéndolo de forma manual y el sistema arrancó perfectamente. Aún así, la solución necesitaba que se hiciese de forma automática, y para ello, se necesitaba añadir dicha opción a través de Yocto. Finalmente se descubrió que había que crear un script en la capa bsp del proyecto que se encargase de pasarle dicha opción al núcleo.

El fichero creado fue el siguiente: /ruta-al-directorio-yocto/sources/meta-nayar-bsp/scripts/lib/wic/canned-wks/mkefidisk.wks. Y su contenido: bootloader --ptable gpt --timeout=5 --append="rootfstype=ext4 intel_idle.max_cstate=1 console=ttyS0,115200 console=tty0 rootflags=data=journal noapic"

7.2 Paquetes *rpm delta*

Los paquetes rpm delta contienen la diferencia entre la versión de una aplicación instalada en un equipo y la siguiente versión de la misma aplicación. Fueron introducidos por

Fedora en el año 2009 con la idea de no generar tanto tráfico de red a la hora de actualizar las aplicaciones, pensando en los lugares que no disponían de banda ancha.

Se ha intentado poner en marcha el uso de este tipo de paquetes pensando que muchos de los dispositivos Advertisim que se utilizan están conectados a través de una tarifas de datos móviles.

Tras probar a implementarlo se vio que hay un bug en la herramienta dnf que no permite trabajar con este tipo de paquetes. Se probó a actualizar dnf y libdnf a la última versión pero seguía sin funcionar. Finalmente se decidió abandonar este tema, esperando que los desarrolladores de dnf lo solucionen.

7.3 Librería *libb64*

Esta librería es utilizada por los demonios g3d y n4m, comentados en el apartado 2.1, y por lo tanto es necesario añadirla a la imagen. A continuación se expondrán los problemas generados por esta librería y sus soluciones.

El primero de ellos fue que el nombre con el que Yocto trataba este paquete era libb64-staticdev pese a que la receta se llamaba libb64. Esto generó confusión puesto que en principio se creía que el nombre del paquete dependía del nombre de la receta. Tras investigar al respecto se descubrió que el nombre que Yocto le da al paquete depende de las variables FILES y PACKAGES de la receta. Este problema influía a la hora de indicarle a Yocto que debe incluir este paquete en la imagen. Para solucionar el problema había dos opciones:

1. Modificar las variables PACKAGES y FILES para cambiar el nombre del paquete generado.
2. Cambiar en la receta de la imagen el nombre del paquete y poner el nombre que le da Yocto.

De las dos soluciones, se decidió utilizar la segunda ya que había que modificar menos código y así también se utilizaba el nombre que Yocto le da al paquete. Por tanto, la solución final fue poner en la receta de la imagen como nombre del paquete: “libb64-staticdev”.

El segundo de los problemas fue que Yocto añade una dependencia a libb64-staticdev de libb64-dev, por lo tanto, se instalaba automáticamente también libb64-dev. Este último paquete no era necesario instalarlo. Para ello la solución encontrada fue romper la dependencia entre ambos. En la Figura 28 se puede ver enmarcado en color rojo las dos líneas que se añadieron para ello.

Y el último problema fue que, en lugar de generar la librería estática, se necesitaba que se generase un paquete con la librería dinámica. Para ello, en el bloque de código do_install de la receta de libb64 se añadieron al paquete los ficheros .so creados tras compilar el código fuente de la librería. Además de ello, se configuró la receta para que el nombre del paquete que incluye la librería dinámica fuese libb64 y no libb64-dev. En la Figura 28 se puede ver enmarcado en color naranja el código relativo a este problema.

```
LICENSE = "GPLv2 & LGPLv2.1"
LIC_FILES_CHKSUM = "file://LICENSE;md5=ce551aad762074c7ab618a0e07a8dca3"

SRC_URI = "http:                    /repo/external/libs/libb64-${PV}.src.zip \
        file://patch_dynamic_library.patch"
SRC_URI[md5sum] = "a609809408327117e2c643bed91b76c5"
SRC_URI[sha256sum] = "343d8d61c5cbe3d3407394f16a5390c06f8ff907bd8d614c16546310b689bfd3"

# Las dos siguientes lineas se utilizan para romper las dependencias del paquete staticdev y dev
RDEPENDS_${PN}-staticdev = ""
RDEPENDS_${PN}-dev = ""

# Las 2 lineas siguientes permiten que no se incluyan las librerias dinamicas en el
# paquete libb64-dev_1.2-r0_amd64.deb y que se incluyan en el paquete libb64-1_1.2-r0_amd64.deb
FILES_${PN}-dev_remove = " ${libdir}/lib*.so"
FILES_${PN} += "${libdir}/*.so"

CFLAGS_append = "-fPIC"

do_configure[noexec] = "1"

do_install () {
        oe_runmake all
        install -d ${D}${libdir}
        install -d ${D}${includedir}/b64
        install -m644 ${S}/src/libb64.so ${D}${libdir}
        install -m644 ${S}/include/b64/* ${D}${includedir}/b64
}
```

Figura 28. Receta de *libb64.*

7.4 Librería *alsa-lib*

Alsa-lib es una librería que se utiliza para gestionar los drivers de sonido ALSA (Advanced Linux Sound Architecture) del núcleo Linux. La empresa la utiliza para gestionar el audio y las salidas de audio del dispositivo Iwill y que así se pueda reproducir el sonido.

El primero de los problemas con la librería alsa-lib es similar al primer problema de la librería libb64 citado anteriormente. Pese a llamarse la receta alsa-lib, Yocto le da el nombre de libasound2. Esto es debido a que cuando se compila la librería, se generan los ficheros binarios en un directorio llamado libasound2, y por ello, Yocto genera un paquete llamado así. En un principio se añadió en la receta de la imagen el paquete con el nombre alsa-lib, pero tras advertir el error se cambió y se puso libasound2.

El segundo de los problemas fue que Yocto, a la hora de crear la imagen, no encontraba el paquete libasound2. Esto es porque, aunque se genere el paquete llamado libasound2, hay que especificar en la receta que se creará un paquete llamado así. Para solucionar esto se añadió a la receta de alsa-lib la siguiente línea: RPROVIDES_${PN} = “libasound2”
Con ella le indicamos a la Yocto que la receta alsa-lib generará el paquete llamado libasound2 y así a la hora de generar la imagen sabrá dónde encontrarlo.

7.5 Compilación

Cuando se pedía a Yocto generar más de una imagen a la vez surgía un problema. Más concretamente, cuando compilaba el código de la librería gráfica QT5. El problema radicaba en que se consumía toda la memoria RAM de la que la máquina Yocto disponía. Para solucionar este tema se propusieron dos opciones:

1. Aumentar la cantidad de memoria RAM en la máquina Yocto.
2. No crear más de una imagen al mismo tiempo.

Se decidió utilizar la primera opción ya que en ocasiones se necesitaban generar dos imágenes al mismo tiempo para probar dos modificaciones de forma más rápida.

7.6 Claves *gpg* en la máquina Yocto

La generación de las claves gpg sobre la máquina virtual Yocto tardaba demasiado tiempo. Esto es debido a que una máquina virtual no crea suficiente aleatoriedad para generar las claves de forma rápida. Esta aleatoriedad se mide con la entropía. A mayor entropía, mayor aleatoriedad de los datos. Para aumentar la entropía, una técnica que se utiliza es generar los números aleatorios a partir de las interrupciones hardware. Por lo tanto, al no tener acceso al hardware las máquinas virtuales, esta entropía es menor.

Como solución a este problema se puede instalar un paquete llamado rng-tools, que permite incrementar la entropía. En este caso en concreto, lo que hace es utilizar el fichero /dev/urandom para ayudar a generar los números aleatorios. Este fichero genera constantemente contenido semi-aleatorio. A continuación se detallan los pasos seguidos para poder generar las claves con mayor rapidez:

1) Instalar el paquete rng-tools con el siguiente comando:

```
$  apt-get install rng-tools
```

2) Editar el fichero de configuración de *rng-tools* para indicarle que utilice también el fichero */dev/urandom*:

```
$ echo HRNGDEVICE=/dev/urandom >> /etc/default/rng-tools
```

3) Reiniciar el demonio de *rng-tools*:

```
$ /etc/init.d/rng-tools restart
```

7.7 Golang

La aplicación *advertisim-daemon* está escrita en lenguaje Go y para su compilación se utiliza la herramienta *go-dep*. Para poder compilar la aplicación se deben ejecutar los siguientes comandos: *“dep init” y “dep ensure”*. Hacer esto en una máquina local es tan fácil

como ejecutar los dos comandos anteriores. Pero en un entorno Yocto, que compila aplicaciones para otras plataformas (compilación cruzada), hay que indicarle que la herramienta *go-dep* será utilizada por la propia máquina Yocto (máquina nativa) y que el binario resultante debe funcionar en el dispositivo *Iwill (*máquina destino*).*

El problema fue que en un principio, por desconocimiento, en la receta de *advertisim-daemon* se indicaba que hiciera *"dep init"* y *"dep ensure"* sin especificar que lo debía ejecutar la máquina nativa. Al probarlo falló la compilación de la receta. El problema se resolvió al especificar en dicha receta que *go-dep* se ejecutase en la máquina nativa y, además, añadir *go-dep* como dependencia de *advertisim-daemon* para así asegurar que esté instalado a la hora de compilar.

En la Figura 29 se puede ver encuadrado en color rojo cómo se especifica que la herramienta *go-dep* que se utilizará para compilar la aplicación será nativa (*go-dep-native*) y que, además, es una dependencia, es decir, que se ejecutará en la máquina nativa (máquina Yocto) y que debe estar instalada en dicha máquina antes de compilar la aplicación. En el segundo cuadro rojo se puede ver cómo se usa la herramienta *go-dep* para compilar la aplicación (*dep init* y *dep ensure*).

```
inherit go systemd
DEPENDS = "go-dep-native mercurial-native"

CGO_ENABLED = "1"
GO_LINKSHARED = ""

do_compile_prepend() {
    rm -f ${WORKDIR}/build/src/${GO_IMPORT}/Gopkg.toml
    rm -f ${WORKDIR}/build/src/${GO_IMPORT}/Gopkg.lock
    cd ${WORKDIR}/build/src/${GO_IMPORT}
    dep init
    dep ensure
}
```

Figura 29. Configuración para que *go-dep* se ejecute en máquina nativa.

Además de ello se tuvo que modificar la receta *go-dep* para que ese comando pueda ser utilizado en la máquina Yocto (nativa) y compilar el software. A continuación se muestra la línea que se tuvo que añadir para modificar la receta.

```
$ cat modificacion_receta_go_dep_%.bbappend
BBCLASSEXTEND = "native nativesdk" # Otavio Salvador patch
```

7.8 Reinicio del dispositivo *Iwill*

Durante el desarrollo del proyecto se detectó que el dispositivo *Iwill* se reiniciaba ocasionalmente cuando estaba reproduciendo vídeos. Tras investigar de dónde venía el problema, utilizando el método ensayo-error se descubrió que estaba relacionado con la aceleración de vídeo. Como prueba, se desinstalaron las librerías que permiten dicha aceleración para que la CPU hiciese todo el trabajo de aceleración gráfica. Con ello se pudo observar que el dispositivo Iwill dejaba de reiniciarse al reproducir vídeos. Esto no se puede

considerar una solución ya que se carga al procesador de tareas que no debe realizar, pero ayudó a determinar dónde seguir buscando para encontrar la solución. Se investigó más a fondo y se llegó a la conclusión de que, cuando el procesador trabajaba a pleno rendimiento codificando y decodificando vídeo, el dispositivo *Iwill* no se reiniciaba, lo que condujo a la deducción de que el motivo de reinicio venía producido por el procesador. Investigando más sobre la causa se vio que el procesador puede ser configurado para trabajar en varios estados (*C-states*), y que cambiando este estado del procesador de forma manual se dejaban de producir los reinicios.

Los *C-States* son configuraciones del procesador para ahorrar energía cuando este no necesita trabajar a pleno rendimiento. En un principio, el dispositivo *Iwill* estaba configurado para que el cambio de *C-State* fuese gestionado por el sistema operativo de forma dinámica. Por lo tanto, cuando el procesador no tiene carga de trabajo, el sistema lo pone en estado de bajo consumo y cuando sí tiene carga le cambia el estado.

La documentación consultada para resolverlo fue *"Controlling Processor C-State Usage in Linux"* [17]. La solución a la que se optó fue forzar al núcleo a que el procesador solo pueda estar en dos *C-States* en los que se sabe que nunca entrará en bajo consumo. Para ello se le pasa como parámetro al núcleo durante el arranque con la opción "*intel_idle.max_cstate=1*" de la misma forma que se pasa la opción *noapic* vista en el apartado 7.1.

La configuración que se ha quedado finalmente es que el procesador solo puede trabajar en los *C-States* menores, es decir, en los de mayor consumo (C-state 0 y C-state 1). Con ello se consigue que nunca entre en estados de bajo consumo, por lo que el dispositivo Iwill no se reiniciará y la aceleración de vídeo siempre se hará a través de las librerías correspondientes.

7.9 Configurar el sistema de ficheros en modo journal

Ante un corte de suministro eléctrico puede ocurrir que la información que se debía escribir en disco no esté disponible cuando se reinicie el dispositivo. Para minimizar la pérdida de datos en estas ocasiones, se decidió que el sistema de ficheros se montase en modo *journal*. Lo que se pretende con esto es que cada cambio realizado en el sistema de ficheros quede registrado en un fichero *"journal"* automáticamente. Este fichero almacena un registro de todas las operaciones quese realizan sobre el sistema de ficheros. Así se consigue que ante un corte de suministro eléctrico se pueda dejar el sistema como debe.

Para añadir esta configuración a la imagen se agrega el parámetro "*rootflags=data=journal*" *al script* utilizado para configurar el núcleo nombrado en el apartado 7.1.

Capítulo 8

Conclusiones

En este proyecto se ha generado un sistema de ficheros que engloba lo necesario para que funcione correctamente el sistema operativo de un sistema embebido para el proyecto Advertisim de la empresa Nayar Systems. Dicho sistema incluye la capa *bsp* válida para la arquitectura *x86_64*, la capa para el proyecto Advertisim donde se incluye todo el software propio de la empresa y algunas capas más para completar el software necesario de la imagen. Para ello se han desarrollado 1493 líneas de código distribuidas en 48 recetas. Todo ha sido incluido en un repositorio *git*. Los objetivos principales planteados al inicio del proyecto han sido cumplidos en su totalidad, e incluso se han llevado a cabo nuevas funcionalidades surgidas durante el desarrollo del proyecto. Los objetivos opcionales no se han podido abordar por falta de tiempo.

La imagen del sistema operativo resultado de este proyecto es capaz de arrancar y tener un funcionamiento estable mostrando por pantalla y reproduciendo continuamente a través de los altavoces contenido multimedia en el dispositivo *Iwill*, compuesto por el hardware detallado en el apartado 2.4. Es capaz de gestionar paquetes *rpm* mediante un gestor de paquetes (*dnf*), acceder a un repositorio de paquetes propio de la empresa a través de una red privada virtual (VPN) y, además, solo instalar, si así se desea, paquetes que hayan sido firmados con la clave privada de la empresa, con lo cual se dota al sistema de mayor seguridad. Utiliza *systemd* como gestor del sistema y servicios. Este sistema es la actualización de los antiguos scripts de inicio de *SystemV* y permite un arranque más rápido. Además de ello se automatiza la tarea de añadir en el nombre del paquete la versión actual del mismo, con lo cual al gestionar los paquetes se puede saber en todo momento la versión actual de la aplicación instalada en el sistema y la próxima o anterior versión. Consecuentemente, el proyecto cumple con los requisitos hardware y software citados en el apartado 4.2.

La imagen permitirá a la empresa una disminución de tiempo y de carga de trabajo al no tener que volver a crear un nuevo sistema operativo para cada sistema embebido a desarrollar, pues todos estos sistemas embebidos utilizarán el mismo sistema operativo. También facilitará la tarea de mantenimiento y, además, todos los miembros de la empresa podrán conocer fácilmente el sistema operativo, pues todos los dispositivos de la empresa lo utilizarán.

Para poder exportar el resultado del proyecto y que funcione en otras arquitecturas se debería crear y/o clonar la o las capas *bsp* necesarias tal y como se explica en el apartado 5.5. Tras ello, al compilar la imagen, se configura automáticamente el compilador cruzado obteniendo los datos de la o las nuevas capas *bsp,* lo que permitirá generar una imagen válida

y funcional para la nueva arquitectura.

El funcionamiento del sistema actual se podría mejorar con las siguientes tareas:

- Añadir el funcionamiento de paquetes *delta* citado en el apartado 7.2.
- Montar el repositorio de paquetes *rpm* sobre *Google cloud* y servido a través de *HTTPS (HTTP Secure)* para mayor seguridad.

El hecho de poder crear un sistema operativo personalizado puede ser extensible a cualquier otro proyecto con sistemas embebidos en los que se necesite que el sistema tenga una determinada configuración. Pienso que tendría cabida en la industria 4.0, adaptación de *smart cities*, etc. en definitiva, en cualquier entorno en el cual se utilicen muchos sistemas embebidos con la misma configuración y que realicen las mismas o similares tareas. Además, un factor que yo veo muy positivo es que dicho sistema tiene lo justo y necesario para poder trabajar de forma normal, sin aplicaciones ni configuraciones innecesarias, lo que lo hace más liviano y eficaz.

El desarrollo de este proyecto me ha permitido adentrarme en el mundo de los sistemas embebidos y conseguir un aprendizaje a fondo del sistema operativo Linux. Además, pienso que mis competencias de trabajo en equipo han aumentado, pues he estado trabajando en el proyecto con el equipo de I+D+I de la empresa.

Me ha gustado poder poner en práctica todo lo aprendido en la fase de documentación. Pero lo que más me alegra es haber sido capaz de personalizar un sistema operativo Linux acorde a lo requerido por la empresa y que, además, incluya sus propias aplicaciones. Esto, antes de contactar con la empresa para poder hacer las prácticas, no sabía ni que se podía hacer.

Bibliografía

[1] Advertisim. http://www.advertisim.com/ - [Última visita Junio 2018]

[2] Yocto Project. https://www.yoctoproject.org/ [Última visita Mayo 2018]

[3] Capas Yocto. http://layers.openembedded.org/layerindex/branch/master/layers/ [Última visita Junio 2018]

[4] Yocto docs. https://www.yoctoproject.org/docs/ [Última visita Junio 2018]

[5] Imagen de bitbake. http://www.inforcecomputing.com/blog/open-embedded-inforce-platforms/ [Última visita Mayo 2018]

[6] OpenEmbedded project. http://www.openembedded.org/wiki/Main_Page [Última visita Mayo 2018]

[7] Flexypage LT-line. https://flexypage.de/en/lt-line-displays [Última visita Junio 2018]

[8] Microlift XmediaLite. http://www.microlift.es/producos.html [Última visita Junio 2018]

[9] Yocto project terms. https://www.yoctoproject.org/docs/1.4.2/dev-manual/dev-manual.html#yocto-project-terms [Última visita Mayo 2018]

[10] Información salarial mano de obra. https://www.indeed.es/salaries/Analista-programador/a-Salaries?period=monthly [Última visita Julio 2018]

[11] Información del coste según la tarifa de Google. https://cloud.google.com/products/calculator/#id=f11b231c-ad1a-4718-a5f1-159d886e4c3f [Última visita Junio 2018]

[12] González A., 2015, *Embedded Linux Projects Using Yocto Project Cookbook, first edition,* Birmingham, United Kingdom: Packt Publishing Ltd.

[13] Salvador O., 2014, Embedded linux development with Yocto project, Birmingham, United Kingdom: Packt Publishing Ltd.

[14] Yocto Project Quick Start. https://www.yoctoproject.org/docs/2.4.2/yocto-project-qs/yocto-project-qs.html [Última visita Mayo 2018]

[15] Kernel Development Manual. https://www.yoctoproject.org/docs/1.6.1/kernel-

dev/kernel-dev.html#generating-configuration-files [Última visita Mayo 2018]

[16] George G., 2018, Embedded Linux Development Using Yocto Project Cookbook, second edition, Birmingham, United Kingdom: Packt Publishing Ltd.

[17] Hayes S., 2013, Controlling Processor C-State Usage in Linux, http://en.community.dell.com/cfs-file/__key/telligent-evolution-components-attachments/13-4491-00-00-20-22-77-64/Controlling_5F00_Processor_5F00_C_2D00_State_5F00_Usage_5F00_in_5F00_Linux_5F00_v1.1_5F00_Nov2013.pdf [Última visita Mayo 2018]