

Sistemas Críticos

Tema 1:
Selección y configuración de un sistema
operativo

Lección 1:
Introducción a *Embedded Linux*



Jesús González Peñalver

Contenidos

Tema 1: Selección y configuración de un sistema operativo

Introducción

Fundamentos de Linux

Selección de la plataforma y prerequisitos del sistema

Diseño de una plataforma de ejecución mínima

Construcción del *kernel* de *Linux*

Construcción del *Device Tree Blob*

Necesidad de un *Root File System*

Construcción de un *Root File System*

Generación del *First Stage Boot Loader*

Construcción de *U-Boot*

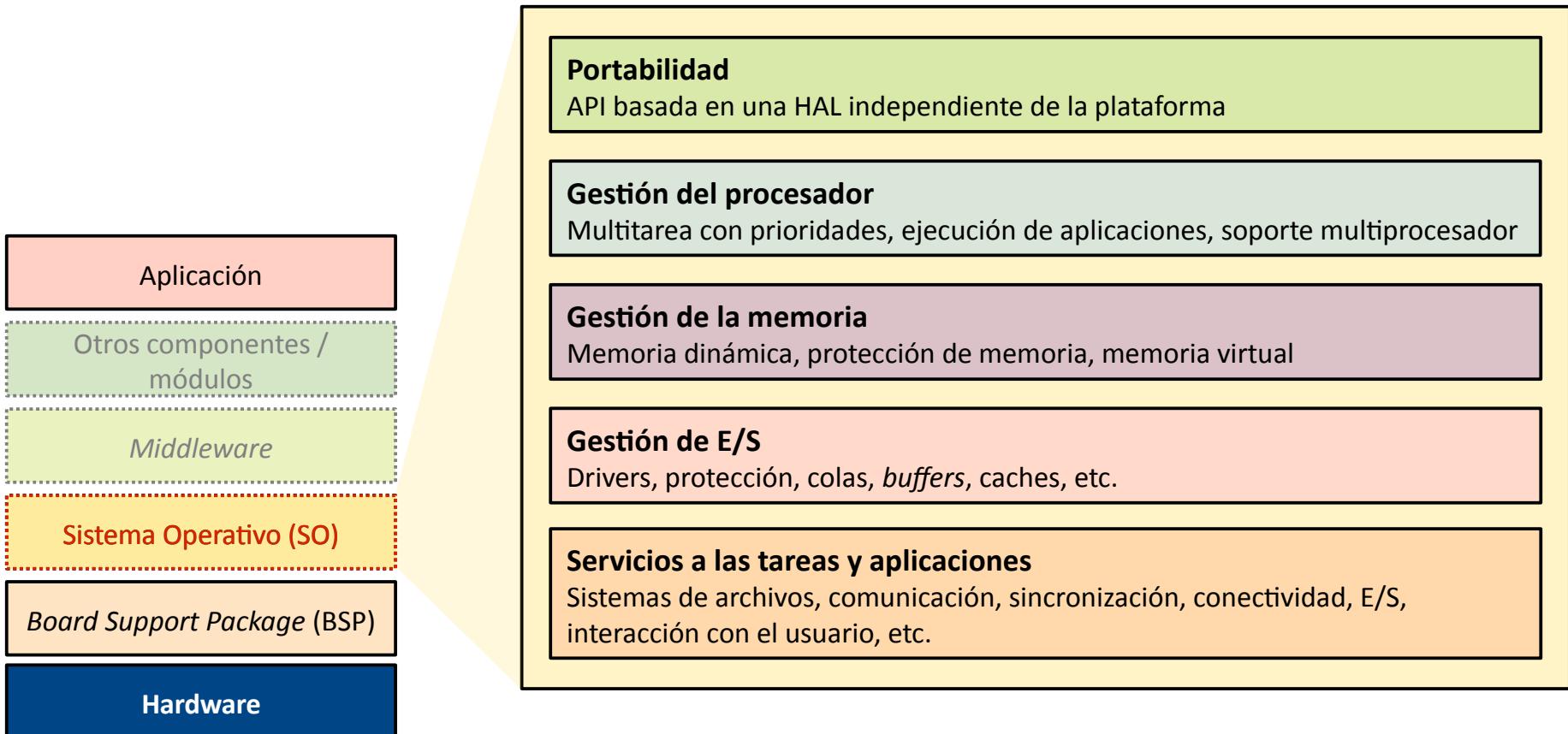
Preparación de la imagen de arranque

Necesidad de un Sistema Operativo

Algunos sistemas empotrados necesitan más servicios de los que proporciona el BSP

Se debe seleccionar un SO adecuado para el sistema (RTOS o GPOS)

El SO se configura para soportar sólo los servicios necesarios



Real Time Operating Systems (RTOS)

Objetivo: Reaccionar ante ciertos estímulos dentro de un intervalo de tiempo determinado

Características:

Fiabilidad: Funcionamiento correcto en el tiempo

Determinismo: Comportamiento predecible en el tiempo

Rendimiento: Dadas las restricciones en potencia de cálculo

Compacidad: Dadas las restricciones en la cantidad de memoria disponible

Escalabilidad: Capacidad de adaptación a los requerimientos de cada aplicación
(desde un DVD hasta un avión)

Componentes fundamentales:

Gestión de tareas (cambio de tarea, gestión de la pila de cada tarea, ...)

Planificación de tareas (determinista y con prioridades)

Intercomunicación de tareas (*buffers*, *mailboxes*, colas de mensajes, ...)

Sincronización de tareas (regiones críticas, semáforos, eventos, ...)

Gestión de memoria y E/S (protección → **MPU**, memoria dinámica, ...)

Aplicaciones: Multimedia, automoción, telecomunicaciones, dispositivos médicos, ...

Uso de un RTOS

RTOS



Código fuente



Scripts de configuración

Opcional

Port

BSP

A veces puede ir incluido en el RTOS

Configuración y construcción

Un RTOS es una biblioteca que proporciona los servicios que necesita nuestra aplicación

Aplicación



Código fuente



Bibliotecas de terceros

RTOS



Ficheros de cabecera



Biblioteca



Script de enlazado

BSP, Procesos, comunicación, sincronización, etc.

Enlazador

Firmware

Dependiendo de la plataforma y del RTOS puede que ya esté portado a nuestra plataforma o que tengamos que portarlo

General Purpose Operating Systems (GPOS)

Objetivo: Permitir la ejecución de múltiples aplicaciones / *widgets*

Características:

Memoria virtual → **MMU** (muchos más procesos que un RTOS, más escalable)

Soporte multiprocesador, red, sistemas de ficheros, seguridad, multilengua, etc.

Múltiples lenguajes de programación (Java, Python, Php, Perl, Ruby, TCL, etc.)

Navegador Web, multimedia e interfaz al usuario ya integradas

Interacción multimodal (Ventanas, 3D, multimedia, voz, gestual, etc.)

Conectividad: TCP/IP, Wi-Fi, *Bluetooth*, 3G, 4G, NFC, ...

Disponibilidad de muchos más drivers y bibliotecas de código

Múltiples distribuidores de software (*open source, markets*, etc.)

Demanda de más memoria y más potencia de cálculo

Ejemplos:

Google Android, Apple iOS, Nokia Symbian, Microsoft Windows 7, Montavista Linux

Aplicaciones:

Smartphones, Tablets, Smart TVs, e-Book readers, routers, Portable Media Players (PMP), ...

Uso de un GPOS

A veces puede ir incluido en el GPOS

BSP

Opcional

Port

GPOS



Scripts de configuración

Código fuente

Las aplicaciones se instalan por el usuario una vez que el GPOS (firmware) está instalado en el sistema

Firmware

Configuración y construcción

Instalación

Aparecen los mercados de aplicaciones

Origen 1



Origen 2



Origen n

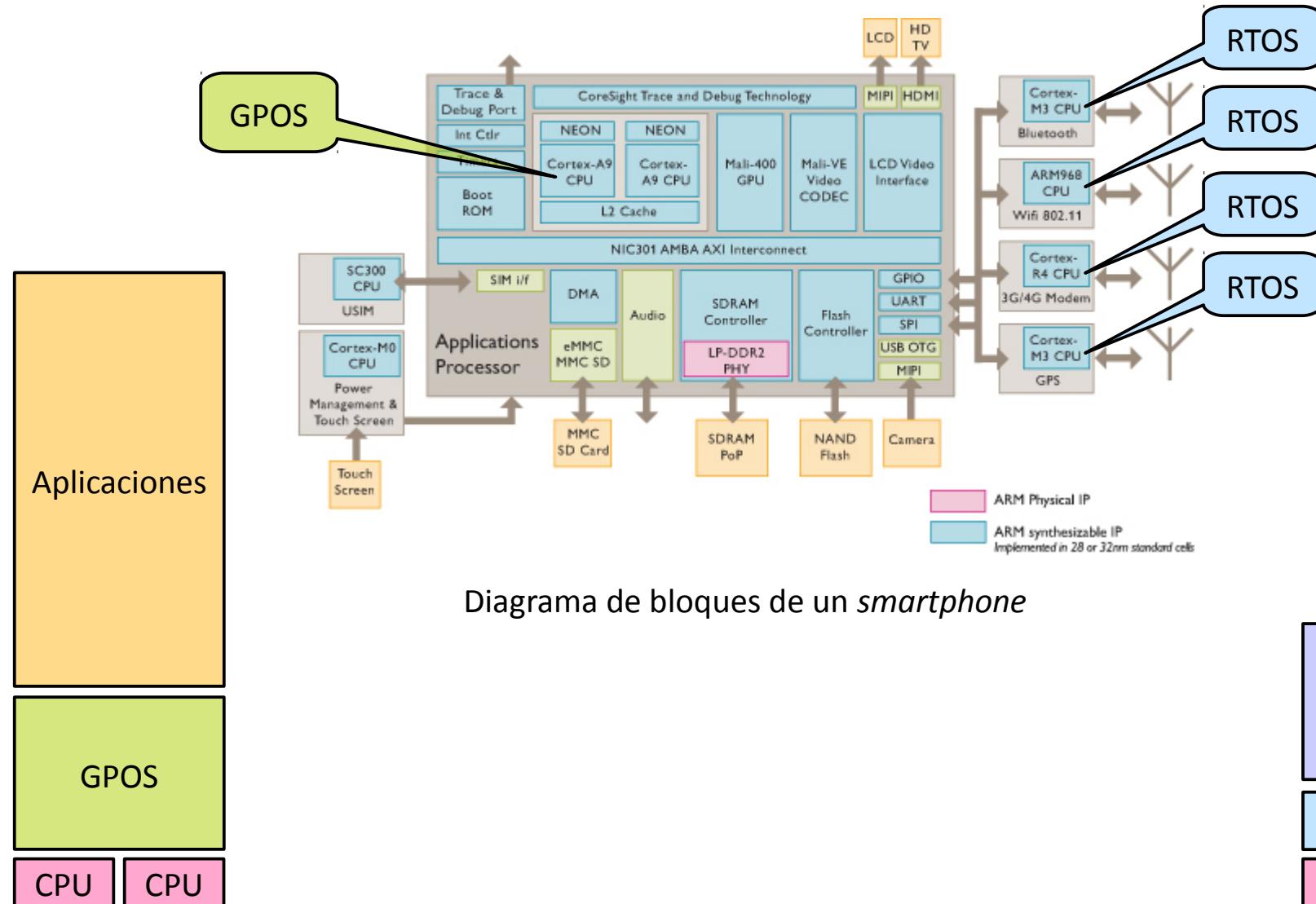


Aplicaciones

Aplicaciones

Aplicaciones

Uso combinado de RTOS y GPOS



Fuente: ARM Markets. Smartphones

<http://www.arm.com/markets/mobile/smartphones.php>

¿Por qué *Embedded Linux*?

Ventajas del open-source

Reutilización de componentes ya existentes (protocolos, drivers, bibliotecas, etc.)

Bajo coste (nos entramos en el desarrollo de nuestra aplicación)

Control total (podemos cambiar cualquier aspecto SW del sistema → tenemos las fuentes)

Calidad (la mayoría del código abierto está muy extendido y depurado)

Flexibilidad (podemos testar varias implementaciones de cara a evaluar la mejor)

Soporte (desarrolladores, comunidad, empresas)

Interoperabilidad e infraestructura de SO

Red, sistemas de archivos, soporte de dispositivos

Gestión de procesos y hebras, interrupciones, etc.

Familiaridad de los desarrolladores

Herramientas estándar

Runtime de C estándar

Las aplicaciones pueden ser prototipadas en el PC

Escalabilidad

Deeply embedded → *single board computers* → *desktop* → *server* → *cluster*

Características de la plataforma

Procesador

La mayoría de procesadores de 32 y 64 bits (*X86, ARM, MIPS, PPC, MicroBlaze, ...*)

Preferiblemente con MMU

RAM

Mínimo 8-32MB, dependiendo de la aplicación

Almacenamiento

Mínimo 4MB

Soporte de dispositivos flash (NAND, NOR) y de bloques (SD/MMC, eMMC)

Comunicaciones

Soporte de la mayoría de buses (I2C, SPI, CAN, USB, ...)

Soporte de red (Ethernet, Wi-Fi, *Bluetooth*, IP, TCP, UDP, *firewalls, routing*, ...)

Soporte de SoCs basados en ARM

Para la mayoría de fabricantes (*TI, Freescale, ST, Ericsson, Atmel*, etc.)

Ejemplos de sistemas con *Embedded Linux*



Set-top box



Router



NAS



Smart TV



TPV



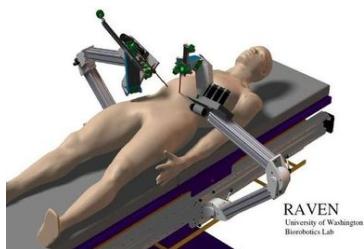
Drones



Robots



Aspiradora



Cirujía



Cortadora láser



Aero-generadores



Ordeño automático

Contenidos

Tema 1: Selección y configuración de un sistema operativo

Introducción

[Fundamentos de Linux](#)

Selección de la plataforma y prerequisitos del sistema

Diseño de una plataforma de ejecución mínima

Construcción del *kernel* de *Linux*

Construcción del *Device Tree Blob*

Necesidad de un *Root File System*

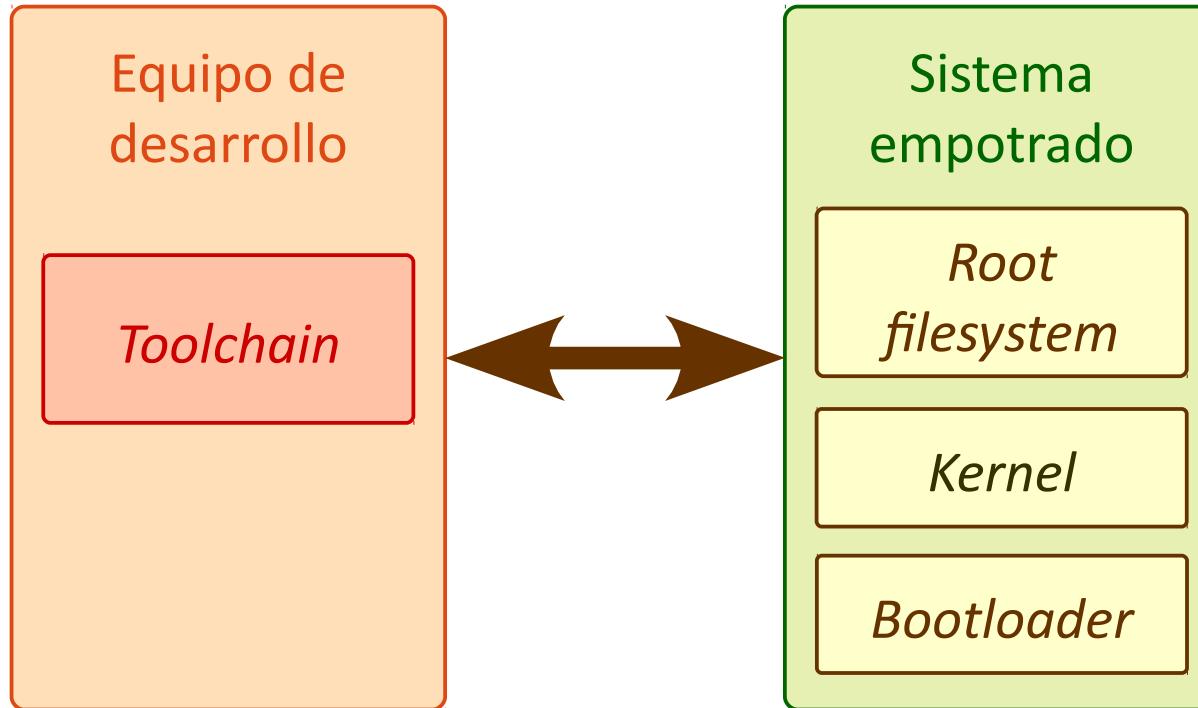
Construcción de un *Root File System*

Generación del *First Stage Boot Loader*

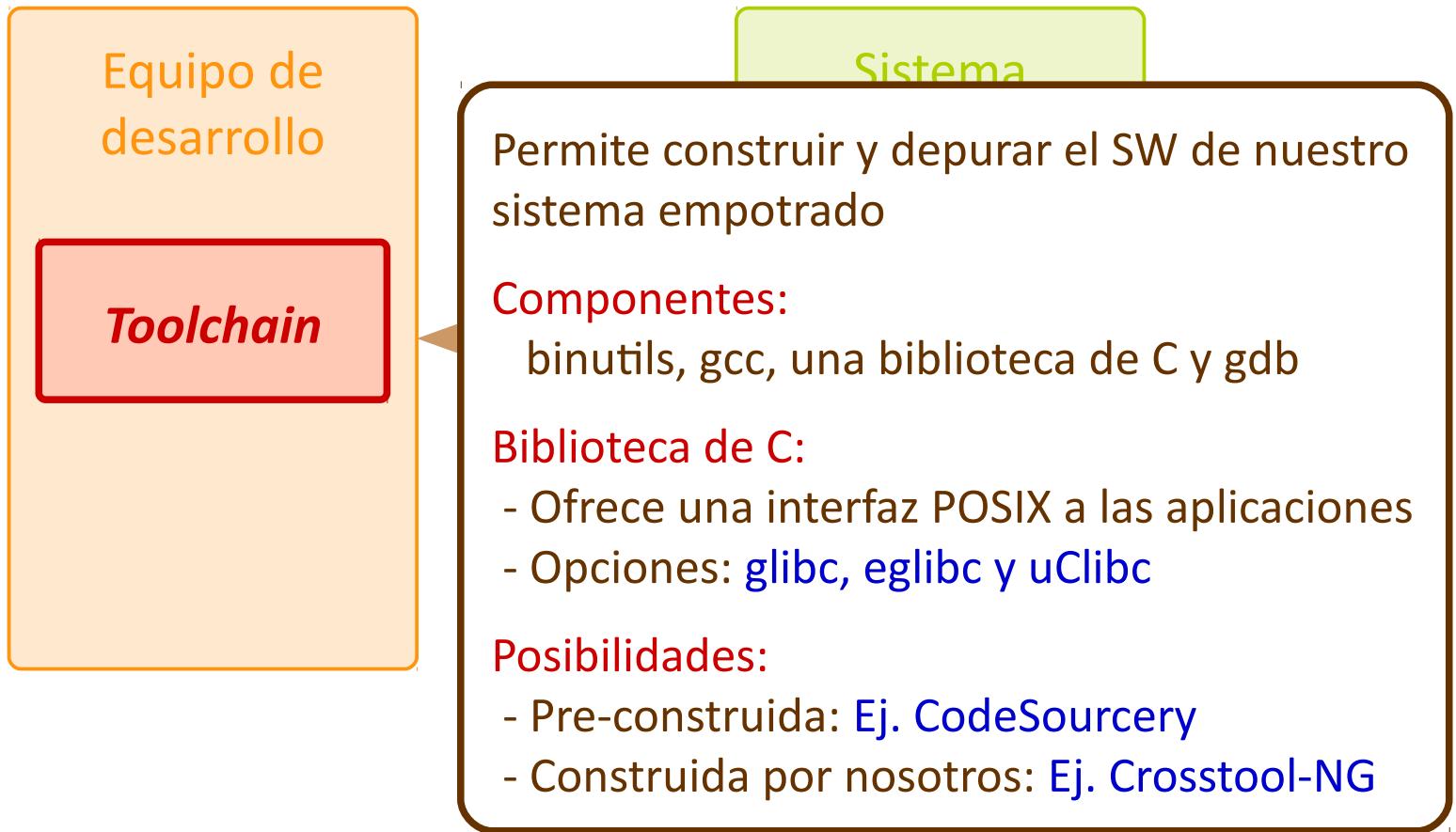
Construcción de *U-Boot*

Preparación de la imagen de arranque

Componentes principales



La *toolchain* de desarrollo



Proceso de carga del *kernel*

Inicializa el sistema, carga la imagen del kernel en RAM y la ejecuta

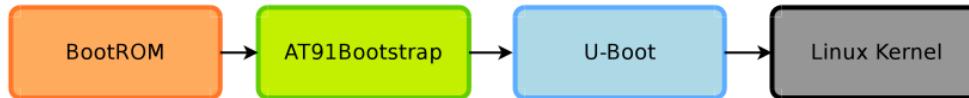
Pasos:

1. El cargador de la ROM carga el *first stage bootloader* de una flash, SD, ...
2. El *first stage bootloader* inicializa el controlador de memoria, los dispositivos críticos, y carga el *second stage bootloader*. No permite interacción y suele ser dependiente del fabricante de la plataforma
3. El *second stage bootloader* es el que carga la imagen del kernel en RAM. Además permite interacción a través de *shell*, manipular dispositivos, red, etc.
Normalmente es *open-source*

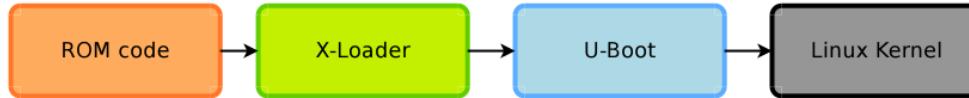


Alternativas para el cargador

Atmel AT91



Texas Instruments OMAP3



Posibilidades para la 2^a etapa:

- *U-Boot*:

Soporta muchos plataformas

Muchas características (red, USB, SD, ...)

- *Barebox*:

Diseño más limpio

Soporta menos HW

Sistema
empotrado

*Root
filesystem*

Kernel

Bootloader

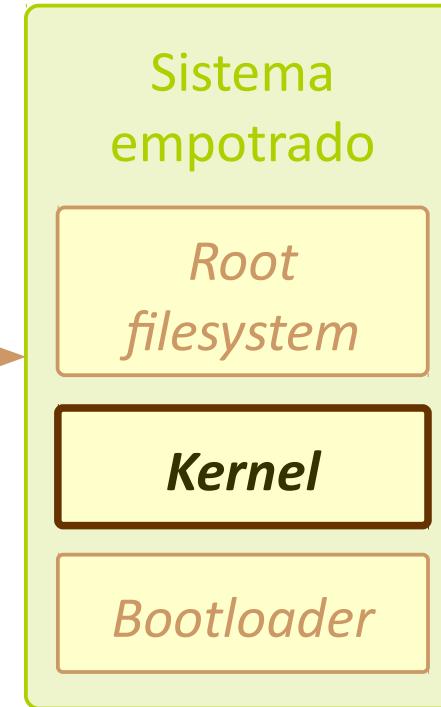
El *kernel* de Linux

Servicios a las aplicaciones:

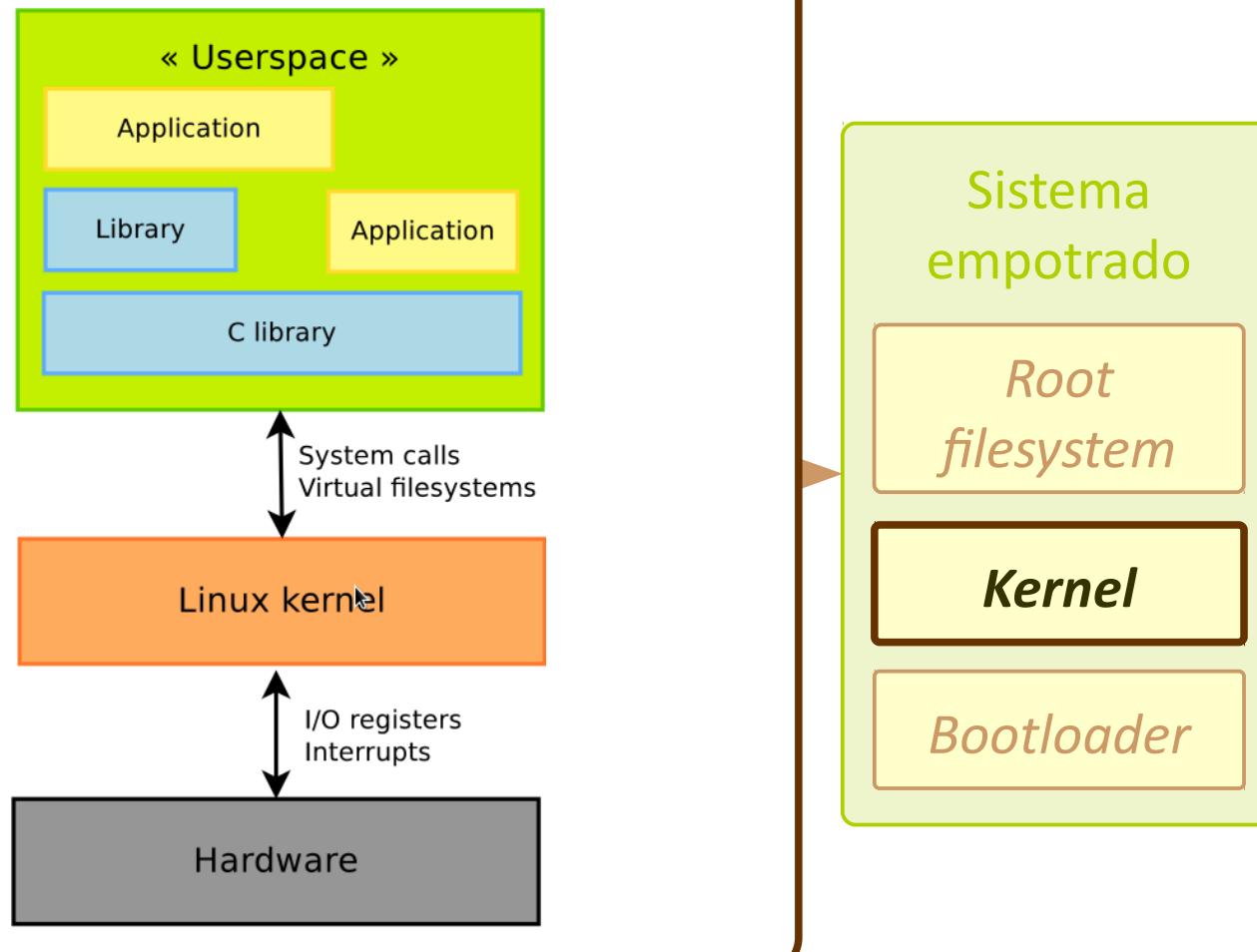
- Gestión de procesos
- Gestión de memoria
- Comunicación/sincronización de procesos
- Drivers (red, almacenamiento, gráficos, sonido, *timers*, GPIO, etc.)
- Sistemas de archivos
- Protocolos y servicios de red
- Gestión de energía

Se puede configurar para seleccionar sólo aquellos servicios que necesitemos para nuestra aplicación

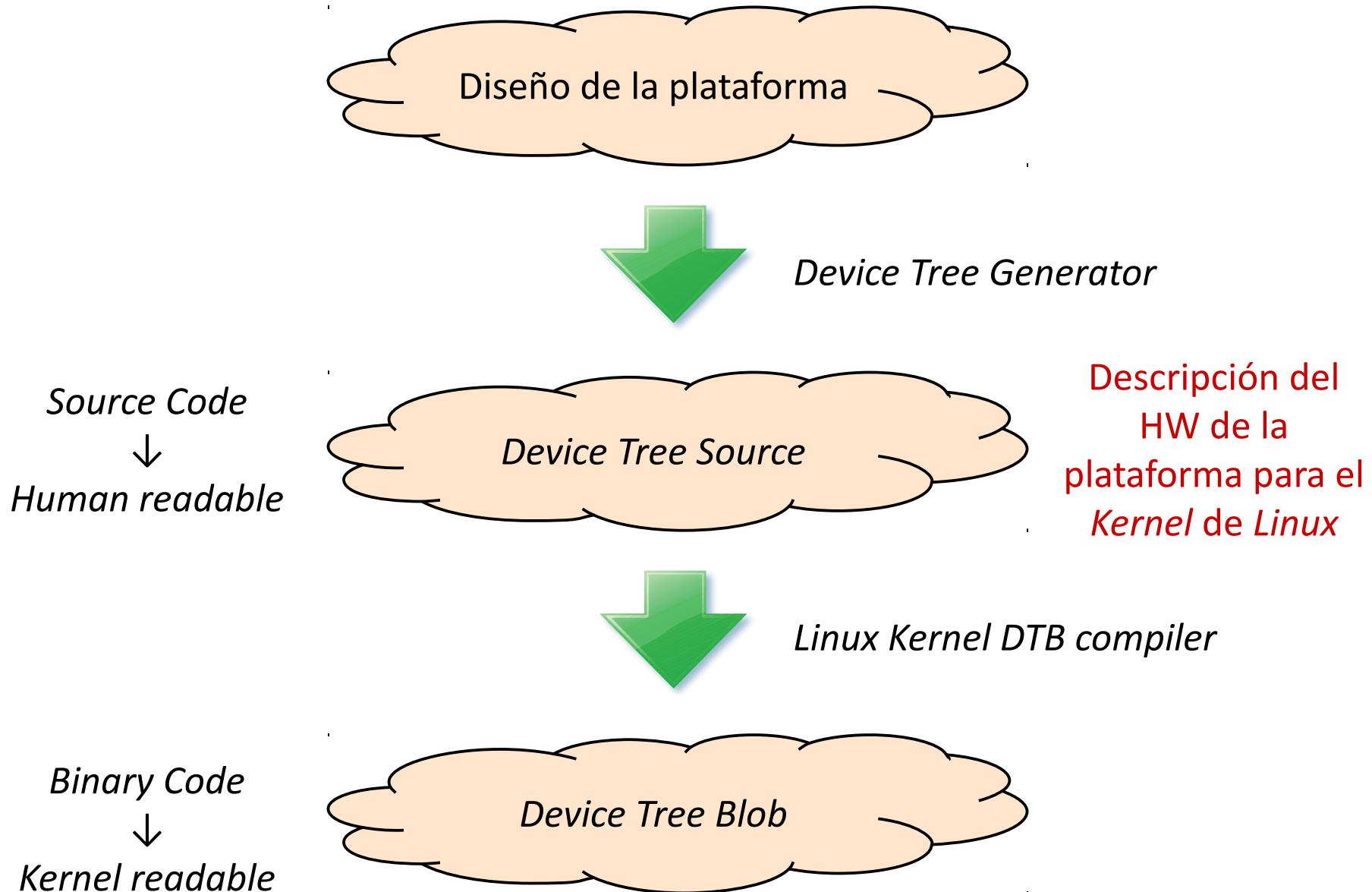
Una vez construido, su imagen ocupa de 1 a 3 MB



Uso del *kernel*



El Device Tree



El sistema de archivos raíz

Contiene todos los ficheros necesarios para que el sistema funcione correctamente

Sistemas de archivos soportados:

- Dispositivos de bloques: ext2, ext3, ext4
- Dispositivos flash: jffs2, ubifs
- RAM: initramfs, ramdisk
- Red: nfs, cifs
- Compatibilidad: vtat, ntfs

Contenido mínimo:

- Proceso init (padre de todos los procesos)
- Una shell (vía UART → administración)
- Una biblioteca de C (API POSIX para apps)
- Device files (gestión de dispositivos)
- Estructura típica (/bin, /etc, /lib, ...)
- FS virtuales (proc, sysfs)



Busybox

Busybox

Contiene todas las órdenes típicas de Linux (cp, mv, sh, ps, ...) en un solo ejecutable

El *RootFS* se rellena con enlaces simbólicos a dicho ejecutable

El comportamiento de *Busybox* dependerá del parámetro `argv[0]`, el nombre y ruta del enlace simbólico que se haya usado para invocarlo

Ahorra mucho espacio en el *RootFS*

Simplifica la creación del *RootFS*

Sistema empotrado

Root filesystem

Kernel

Bootloader

Funcionamiento de *Busybox*

Contenido del directorio /bin

addgroup	df	hush	lzop	pipe_progress	sync
adduser	dmesg	ionice	makemime	printenv	tar
busybox	dnsdomainname	ip	mkdir	ps	touch
cat	dumpkmap	ipaddr	mknod	pwd	true
catv	echo	ipcalc	mktemp	reformime	umount
chattr	ed	iplink	more	rm	uname
chgrp	egrep	iproute	mount	rmdir	uncompress
chmod	false	iprule	mountpoint	rpm	usleep
chown	fdflush	iptunnel	msh	run-parts	vi
cp	fgrep	kill	mt	scriptreplay	watch
cpio	fsync	linux32	mv	sed	zcat
cttyhack	getopt	linux64	netstat	setarch	
date	grep	ln	nice	sleep	
dd	gunzip	login	pidof	stat	
delgroup	gzip	ls	ping	stty	
deluser	hostname	lsattr	ping6	su	

Todos los archivos son enlaces simbólicos a *Busybox*

Integración del sistema

Distribuciones de *Linux*

Debian, Ubuntu, etc., ofrecen distribuciones para sistemas empotrados

Ventajas: Todo está pre-compilado. Sólo hay que seleccionar los paquetes adecuados

Inconvenientes: *RootFS* demasiado grande, código no optimizado para la plataforma

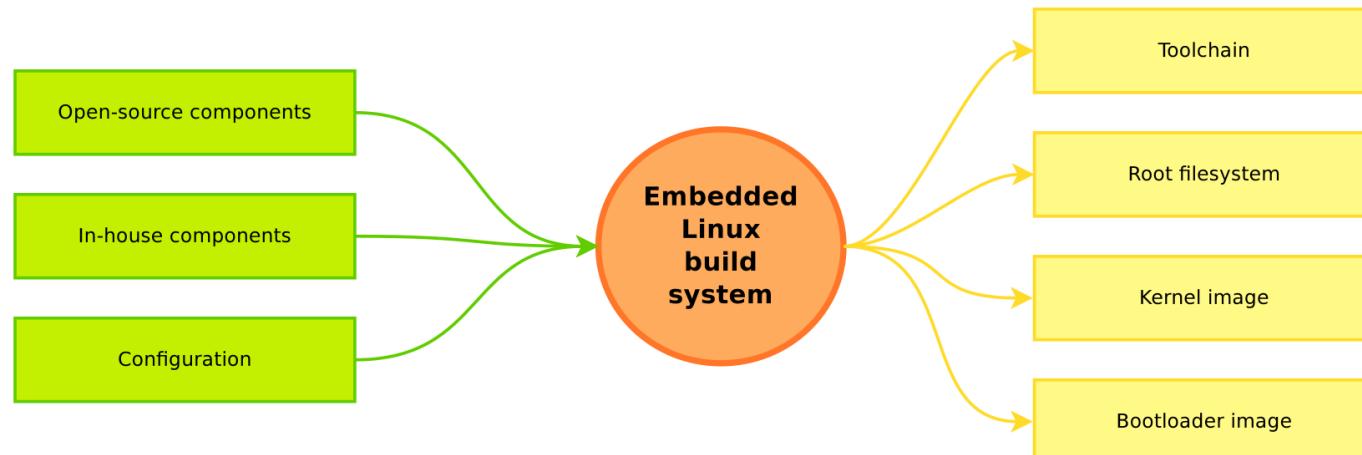
Sistemas de construcción de *Linux* empotrado

Construyen todo el sistema (incluido el *RootFS*) desde las fuentes

Ventajas: Control total sobre el sistema. Reproducibilidad

Inconvenientes: Altos conocimientos técnicos y largos tiempos de compilación

Alternativas: *Buildroot*, *OpenEmbedded*, *Yocto*, ...



Lecturas recomendadas

Sistemas operativos:

- Q. Li, C. Yao. *Real-Time Concepts for Embedded Systems*. CMP Books, 2003. Capítulo 4
- P. A. Laplante. *Real-Time Design and Analysis, 3^a edición*. Willey, 2004. Capítulo 3
- J. Labrose *et al.* *Embedded Software. Know it All*. Newnes, 2008. Capítulo 3
- P. N. Leroux. *RTOS versus GPOS: What is best for embedded development?* Embedded Computing Design, Enero 2005. <http://embedded-computing.com/pdfs/QNX.Jan05.pdf>
- D. Addison. *Embedded Linux applications: An overview*. IBM DeveloperWorks, Agosto 2001. <http://www.ibm.com/developerworks/linux/library/l-embl/index.html>
- B. Japenga. *Why Use Linux for Real-Time Embedded Systems*. <http://www.microtoolsinc.com/Why%20Use%20Embedded%20Linux%20for%20Real%20Time%20Embedded%20Systems%20Rev%20A.pdf>

Embedded Linux:

- T. Petazzoni. *Embedded Linux Introduction*. Free Electrons, 2011.
<http://free-electrons.com/pub/conferences/2011/limoges-clermont/presentation.pdf>
- T. Petazzoni. *Device Tree for Dummies*. Free Electrons, 2013.
<http://free-electrons.com/pub/conferences/2013/elce/petazzoni-device-tree-dummies/petazzoni-device-tree-dummies.pdf>

Lecturas recomendadas

Toolchains de compilación cruzada:

Mentor Graphics. *Sourcery Tools*. <http://www.mentor.com/embedded-software/codesourcery>

Linaro. *ToolChain*. <https://wiki.linaro.org/WorkingGroups/ToolChain>

Crosstool-NG.org. *Crosstool-NG*. <http://www.crosstool-ng.org/>

Boot loaders:

DENX. *U-Boot*. <http://www.denx.de/wiki/U-Boot>

Barebox.org. *Barebox*. <http://www.barebox.org/>

Kernel de Linux:

Linux Kernel Organization, Inc. *The Linux Kernel Archives*. <https://www.kernel.org/>

Root filesystem:

Denys Vlasenko. *BusyBox*. <http://www.busybox.net/>

Sistemas de construcción de Linux empotrado:

Peter Korsgaard. *Buildroot*. <http://www.buildroot.org/>

OpenEmbedded.org. *OpenEmbedded*. <http://www.openembedded.org/>

Linux Foundation. *Yocto Project*. <https://www.yoctoproject.org/>