Device Drivers

Alejandro Furfaro

27 de octubre de 2023

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Un anticipo de lo que viene...

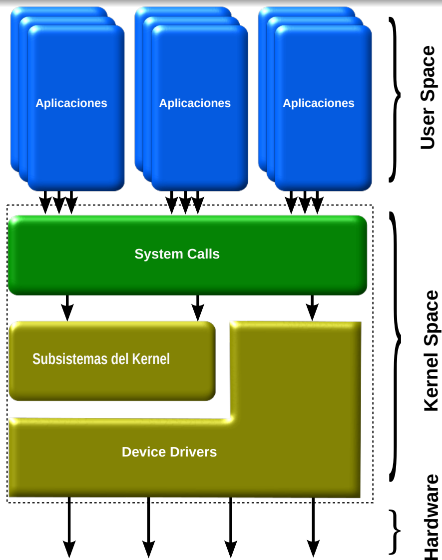
"We're back to the times when men where men and wrote their own device drivers..."

Linus Torvalds

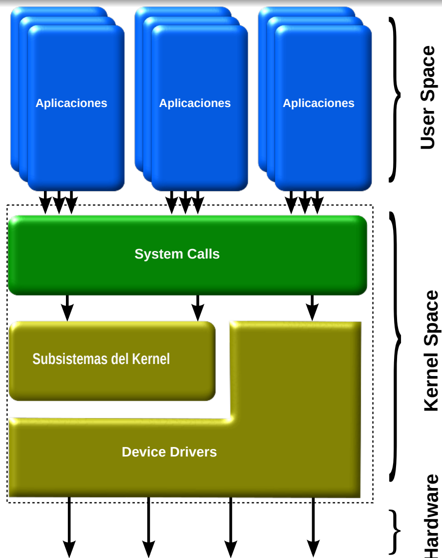
Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Ejecución en modo User

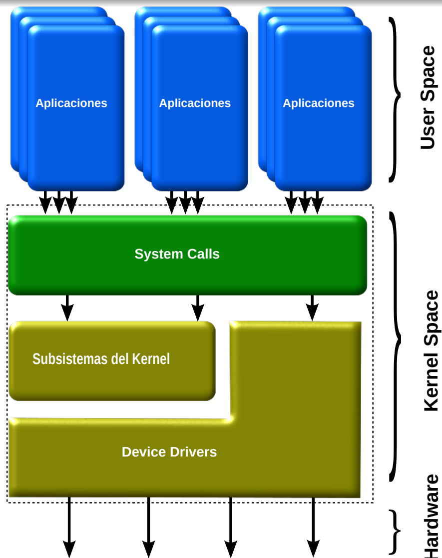


Ejecución en modo User



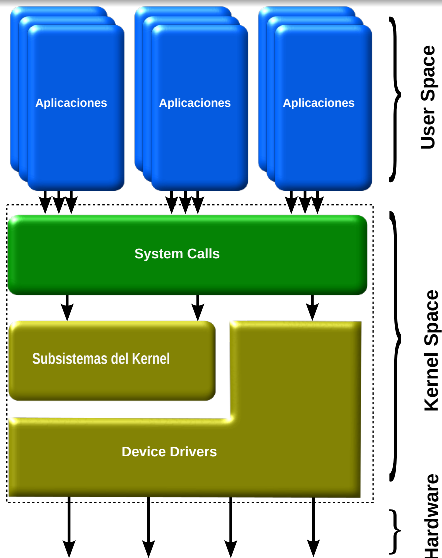
- Los programas ejecutan en forma de Procesos bien definidos.

Ejecución en modo User



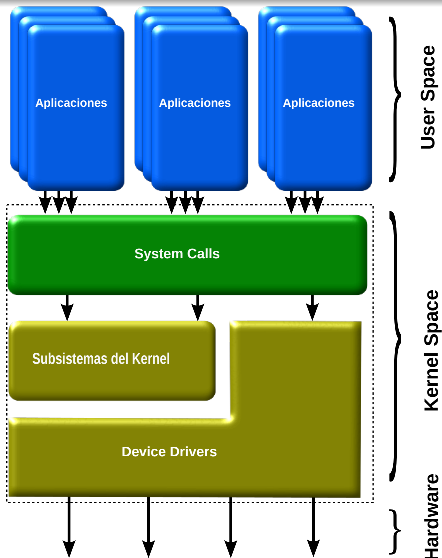
- Los programas ejecutan en forma de Procesos bien definidos.
- Cada proceso tiene su espacio de memoria VIRTUAL.(Memory Protection).

Ejecución en modo User



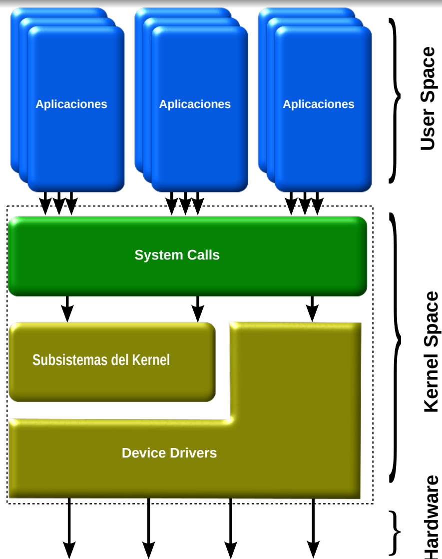
- Los programas ejecutan en forma de Procesos bien definidos.
- Cada proceso tiene su espacio de memoria VIRTUAL.(Memory Protection).
- Recursos Limitados.

Ejecución en modo User



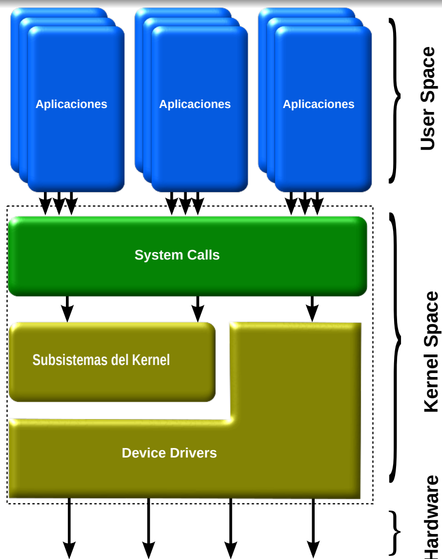
- Los programas ejecutan en forma de Procesos bien definidos.
- Cada proceso tiene su espacio de memoria VIRTUAL.(Memory Protection).
- Recursos Limitados.
- Debe realizar peticiones al kernel para el uso de HW.

Ejecución en modo User



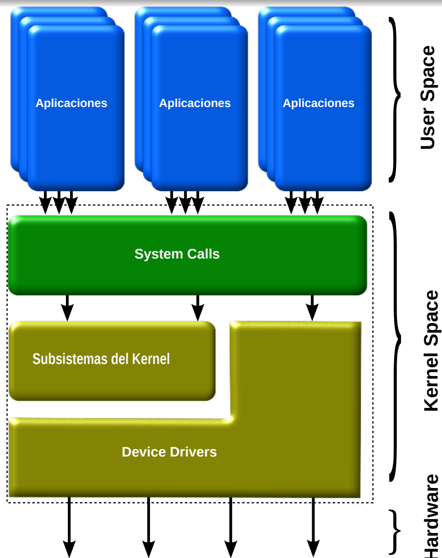
- Los programas ejecutan en forma de Procesos bien definidos.
- Cada proceso tiene su espacio de memoria VIRTUAL.(Memory Protection).
- Recursos Limitados.
- Debe realizar peticiones al kernel para el uso de HW.
- Modo Privilegiado. (Supervisor).

Ejecución en modo User



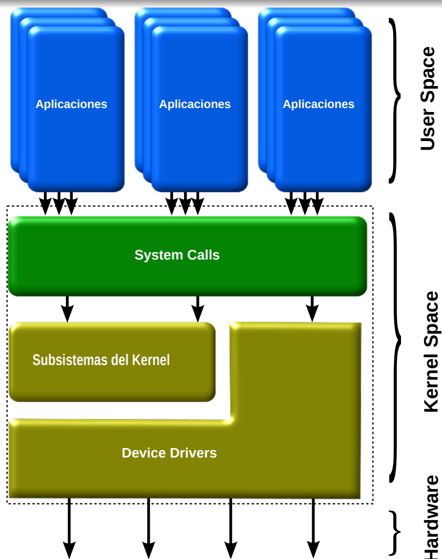
- Los programas ejecutan en forma de Procesos bien definidos.
- Cada proceso tiene su espacio de memoria VIRTUAL.(Memory Protection).
- Recursos Limitados.
- Debe realizar peticiones al kernel para el uso de HW.
- Modo Privilegiado. (Supervisor).
- Acceso irrestricto a dispositivos y memoria.

Ejecución en modo User



- Los programas ejecutan en forma de Procesos bien definidos.
- Cada proceso tiene su espacio de memoria VIRTUAL.(Memory Protection).
- Recursos Limitados.
- Debe realizar peticiones al kernel para el uso de HW.
- Modo Privilegiado. (Supervisor).
- Acceso irrestricto a dispositivos y memoria.
- Controla y administra procesos de User Space.

Ejecución en modo User



- Los programas ejecutan en forma de Procesos bien definidos.
- Cada proceso tiene su espacio de memoria VIRTUAL.(Memory Protection).
- Recursos Limitados.
- Debe realizar peticiones al kernel para el uso de HW.
- Modo Privilegiado. (Supervisor).
- Acceso irrestricto a dispositivos y memoria.
- Controla y administra procesos de User Space.
- Provee medios para solicitar recursos de HW desde User Space.

Kernel Monolítico vs. Micro Kernel

Kernel Monolítico vs. Micro Kernel

- Existen dos modelos para diseñar un kernel.

Kernel Monolítico vs. Micro Kernel

- Existen dos modelos para diseñar un kernel.

  1. Kernel Monolítico

Kernel Monolítico vs. Micro Kernel

- Existen dos modelos para diseñar un kernel.

  1. Kernel Monolítico
  2. Micro Kernel

Kernel Monolítico vs. Micro Kernel

- Existen dos modelos para diseñar un kernel.

  1. Kernel Monolítico
  2. Micro Kernel

- No hay un modelo mejor que otro en forma taxativa.

Kernel Monolítico vs. Micro Kernel

- Existen dos modelos para diseñar un kernel.
  1. Kernel Monolítico
  2. Micro Kernel
- No hay un modelo mejor que otro en forma taxativa.
- Cada uno tiene ventajas y desventajas relativas respecto del otro.

Kernel Monolítico

Kernel Monolítico

- El código completo del Sistema Operativo está ubicado en Kernel Space

Kernel Monolítico

- El código completo del Sistema Operativo está ubicado en Kernel Space
- Todo el sistema ejecuta en modo privilegiado.

Kernel Monolítico

- El código completo del Sistema Operativo está ubicado en Kernel Space
- Todo el sistema ejecuta en modo privilegiado.
- El overhead de comunicación de los diferentes módulos del kernel es mínimo ya que no se cambia el Nivel de Privilegio al llamar de un módulo al otro.

Kernel Monolítico

- El código completo del Sistema Operativo está ubicado en Kernel Space
- Todo el sistema ejecuta en modo privilegiado.
- El overhead de comunicación de los diferentes módulos del kernel es mínimo ya que no se cambia el Nivel de Privilegio al llamar de un módulo al otro.
- El tamaño del código del kernel es comparativamente muy grande.

Kernel Monolítico

- El código completo del Sistema Operativo está ubicado en Kernel Space
- Todo el sistema ejecuta en modo privilegiado.
- El overhead de comunicación de los diferentes módulos del kernel es mínimo ya que no se cambia el Nivel de Privilegio al llamar de un módulo al otro.
- El tamaño del código del kernel es comparativamente muy grande.
- Requiere mayor rigurosidad en el diseño ya que una falla en cualquiera de sus componentes estrella el sistema completo. Y al ser muy grande su código, la probabilidad de error se incrementa.

Micro Kernel

Micro Kernel

- Solo una mínima parte del código del Sistema Operativo está ubicado en Kernel Space, ejecutando en modo privilegiado.

Micro Kernel

- Solo una mínima parte del código del Sistema Operativo está ubicado en Kernel Space, ejecutando en modo privilegiado.
- El resto del Sistema Operativo ejecuta en User Space (En procesadores Intel puede ejecutar en un nivel de privilegio intermedio).

Micro Kernel

- Solo una mínima parte del código del Sistema Operativo está ubicado en Kernel Space, ejecutando en modo privilegiado.
- El resto del Sistema Operativo ejecuta en User Space (En procesadores Intel puede ejecutar en un nivel de privilegio intermedio).
- El overhead de comunicación de los diferentes módulos del kernel aumenta ya que se cambia el Nivel de Privilegio al llamar de un módulo al otro.

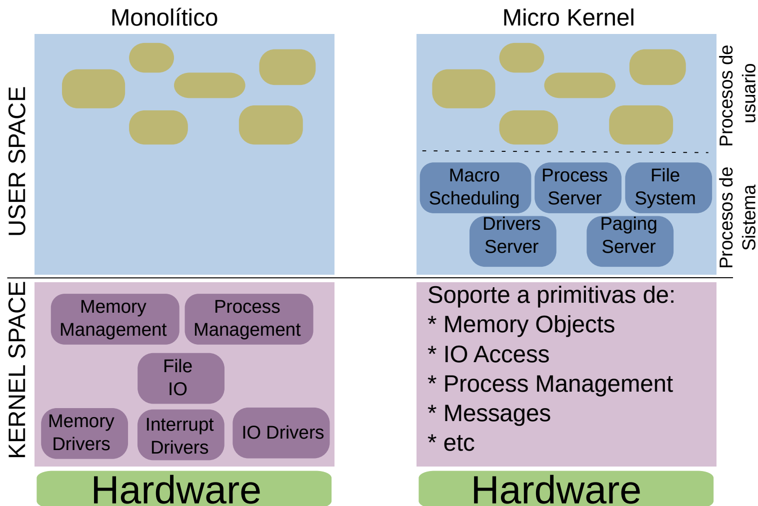
Micro Kernel

- Solo una mínima parte del código del Sistema Operativo está ubicado en Kernel Space, ejecutando en modo privilegiado.
- El resto del Sistema Operativo ejecuta en User Space (En procesadores Intel puede ejecutar en un nivel de privilegio intermedio).
- El overhead de comunicación de los diferentes módulos del kernel aumenta ya que se cambia el Nivel de Privilegio al llamar de un módulo al otro.
- El tamaño del código del kernel es comparativamente muy pequeño.

Micro Kernel

- Solo una mínima parte del código del Sistema Operativo está ubicado en Kernel Space, ejecutando en modo privilegiado.
- El resto del Sistema Operativo ejecuta en User Space (En procesadores Intel puede ejecutar en un nivel de privilegio intermedio).
- El overhead de comunicación de los diferentes módulos del kernel aumenta ya que se cambia el Nivel de Privilegio al llamar de un módulo al otro.
- El tamaño del código del kernel es comparativamente muy pequeño.
- En general debería ser mas robusto ya que el código privilegiado es comparativamente muy pequeño y el resto son procesos de kernel en User Space. Si falla uno de estos procesos el resto del Sistema Operativo continúa operando.

Kernel Monolítico vs. Micro Kernel



Linux adoptó el modelo Monolítico

Linux adoptó el modelo Monolítico

- Por ese motivo El kernel no tiene procesos.

Linux adoptó el modelo Monolítico

- Por ese motivo El kernel no tiene procesos.
- Los procesos operan en Modo User (USER Space) y pasan a ejecutar en Modo Kernel mediante dos mecanismos:

Linux adoptó el modelo Monolítico

- Por ese motivo El kernel no tiene procesos.
- Los procesos operan en Modo User (USER Space) y pasan a ejecutar en Modo Kernel mediante dos mecanismos:
  1. Invocando a una system call cualquiera lo cual lo llevará a un tramo de código de Kernel. Se llega a esta instancia invocando una interrupción de software. Puntualmente la INT 0x80.

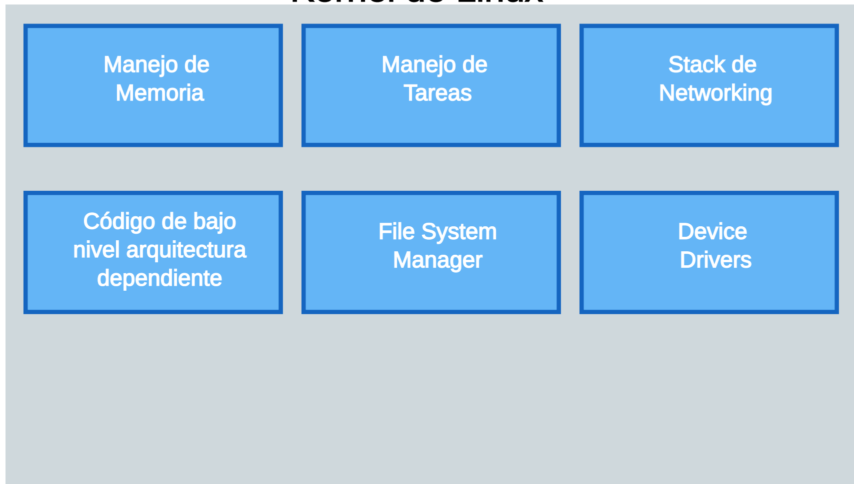
Linux adoptó el modelo Monolítico

- Por ese motivo El kernel no tiene procesos.
- Los procesos operan en Modo User (USER Space) y pasan a ejecutar en Modo Kernel mediante dos mecanismos:
    1. Invocando a una system call cualquiera lo cual lo llevará a un tramo de código de Kernel. Se llega a esta instancia invocando una interrupción de software. Puntualmente la INT 0x80.
    2. Cuando se produce una interrupción de hardware o una Excepción.

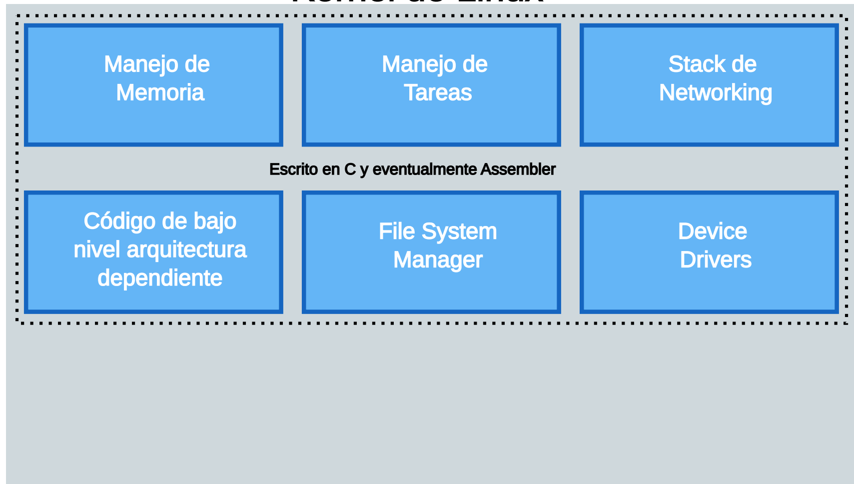
Linux adoptó el modelo Monolítico

- Por ese motivo El kernel no tiene procesos.
- Los procesos operan en Modo User (USER Space) y pasan a ejecutar en Modo Kernel mediante dos mecanismos:
  1. Invocando a una system call cualquiera lo cual lo llevará a un tramo de código de Kernel. Se llega a esta instancia invocando una interrupción de software. Puntualmente la INT 0x80.
  2. Cuando se produce una interrupción de hardware o una Excepción.
- ***En cualquier caso para el sistema operativo el proceso activo sigue siendo el proceso que invocó a INT 0x80 o que fue interrumpido por Hardware o Excepción.***

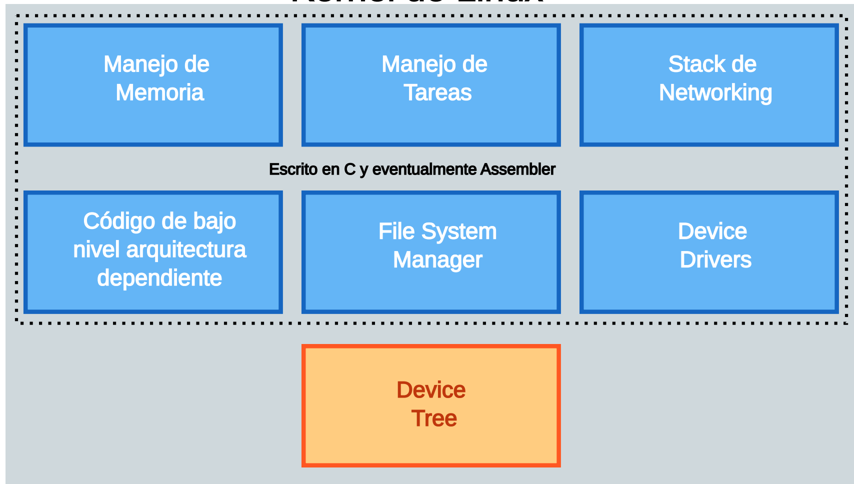
Principales componentes del kernel



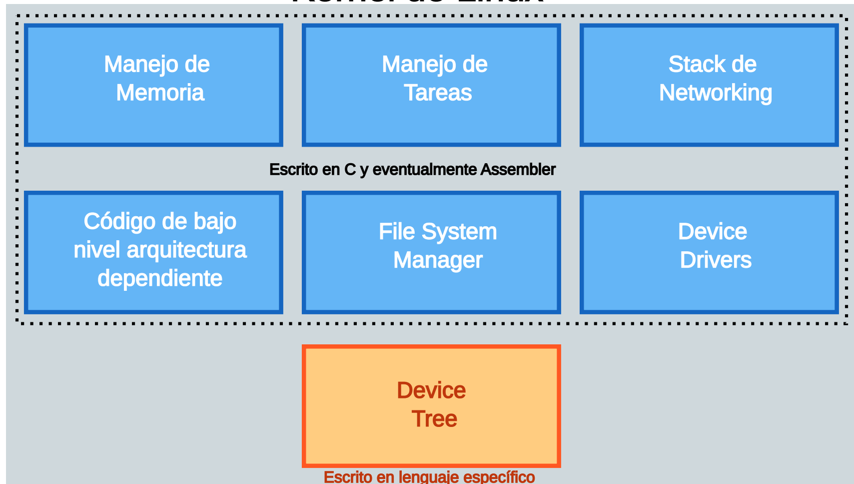
Principales componentes del kernel



Principales componentes del kernel



Principales componentes del kernel



Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

¿Que es un Device Driver?

¿Que es un Device Driver?

- Es código que se ejecuta en modo Kernel.

¿Que es un Device Driver?

- Es código que se ejecuta en modo Kernel.
- Se encarga de la mediación entre los dispositivos de hardware y capas del kernel de mas alto nivel o procesos de usuario.

¿Que es un Device Driver?

- Es código que se ejecuta en modo Kernel.
- Se encarga de la mediación entre los dispositivos de hardware y capas del kernel de mas alto nivel o procesos de usuario.
- Resuelve ***el mecanismo de acceso al hardware*** para obtener la información en crudo.

¿Que es un Device Driver?

- Es código que se ejecuta en modo Kernel.
- Se encarga de la mediación entre los dispositivos de hardware y capas del kernel de mas alto nivel o procesos de usuario.
- Resuelve ***el mecanismo de acceso al hardware*** para obtener la información en crudo.
- No se concentra en la ***política*** de manejo de la información. Este aspecto queda para capas superiores de software. Ej:

¿Que es un Device Driver?

- Es código que se ejecuta en modo Kernel.
- Se encarga de la mediación entre los dispositivos de hardware y capas del kernel de mas alto nivel o procesos de usuario.
- Resuelve ***el mecanismo de acceso al hardware*** para obtener la información en crudo.
- No se concentra en la ***política*** de manejo de la información. Este aspecto queda para capas superiores de software. Ej:
  - El Driver de disco accede a los detalles del controlador y muestra el disco hacia las capas superiores como un arreglo de bloques consecutivos (***mecanismo***).

¿Que es un Device Driver?

- Es código que se ejecuta en modo Kernel.
- Se encarga de la mediación entre los dispositivos de hardware y capas del kernel de mas alto nivel o procesos de usuario.
- Resuelve ***el mecanismo de acceso al hardware*** para obtener la información en crudo.
- No se concentra en la ***política*** de manejo de la información. Este aspecto queda para capas superiores de software. Ej:
  - El Driver de disco accede a los detalles del controlador y muestra el disco hacia las capas superiores como un arreglo de bloques consecutivos (***mecanismo***).
  - Luego File System Manager interpreta los requerimientos de los usuarios, y de acuerdo al formato estipulado por el File System bajo el que se formateó el disco (***política*** de manejo de información), suministra la información a la aplicación.

¿Como se incorpora al Kernel el código de un driver?

¿Como se incorpora al Kernel el código de un driver?

- En forma estática.

¿Como se incorpora al Kernel el código de un driver?

- En forma estática.
    - Como resultado el driver pasa a formar parte del bloque de código del kernel.

¿Como se incorpora al Kernel el código de un driver?

- En forma estática.
  - Como resultado el driver pasa a formar parte del bloque de código del kernel.
  - Es mas compacto.

¿Como se incorpora al Kernel el código de un driver?

- En forma estática.
  - Como resultado el driver pasa a formar parte del bloque de código del kernel.
  - Es mas compacto.
  - Requiere incluir los fuentes del driver en la compilación del kernel.

¿Como se incorpora al Kernel el código de un driver?

- En forma estática.
  - Como resultado el driver pasa a formar parte del bloque de código del kernel.
  - Es mas compacto.
  - Requiere incluir los fuentes del driver en la compilación del kernel.
- Escribiendo el driver como un kernel module que son linkeables en run-time.

¿Como se incorpora al Kernel el código de un driver?

- En forma estática.
    - Como resultado el driver pasa a formar parte del bloque de código del kernel.
    - Es mas compacto.
    - Requiere incluir los fuentes del driver en la compilación del kernel.
- Escribiendo el driver como un kernel module que son linkeables en run-time.
    - Mucho mas flexible.

¿Como se incorpora al Kernel el código de un driver?

- En forma estática.
  - Como resultado el driver pasa a formar parte del bloque de código del kernel.
  - Es mas compacto.
  - Requiere incluir los fuentes del driver en la compilación del kernel.
- Escribiendo el driver como un kernel module que son linkeables en run-time.
  - Mucho mas flexible.
  - La estructura modular es lo que utilizamos desde hace ya suficiente tiempo como para considerarlo el modo estándar de trabajar, por su flexibilidad especialmente útil en la fase de desarrollo.

Clasificación

1. Char Devices
2. Block Devices
3. Network Devices
4. Miscellaneous (Bus / Platform) Devices

Char Device

Char Device

- Se accede a través de un stream de bytes. (Idénticamente a un archivo estándar)

Char Device

- Se accede a través de un stream de bytes. (Idénticamente a un archivo estándar)
- Para acceder al mismo se utilizan las syscalls básicas: `read ()`, `write ()`, `open ()`, `close ()`, y `ioctl ()`.

Char Device

- Se accede a través de un stream de bytes. (Idénticamente a un archivo estándar)
- Para acceder al mismo se utilizan las syscalls básicas: `read ()`, `write ()`, `open ()`, `close ()`, y `ioctl ()`.
- Si bien acceden a los datos en forma secuencial, a diferencia de los archivos comunes, no nos podemos desplazar hacia atrás y hacia adelante mediante `lseek ()` por ejemplo.

Char Device

- Se accede a través de un stream de bytes. (Idénticamente a un archivo estándar)
- Para acceder al mismo se utilizan las syscalls básicas: `read ()`, `write ()`, `open ()`, `close ()`, y `ioctl ()`.
- Si bien acceden a los datos en forma secuencial, a diferencia de los archivos comunes, no nos podemos desplazar hacia atrás y hacia adelante mediante `lseek ()` por ejemplo.
- El acceso se hace a través de los nodos del File System donde están declarados (Ej: /dev/tty1).

Block Device

Block Device

- El kernel accede a los datos a través de bloques. (Típicamente 512 bytes).

Block Device

- El kernel accede a los datos a través de bloques. (Típicamente 512 bytes).
- Utilizados para dispositivos de almacenamiento.

Block Device

- El kernel accede a los datos a través de bloques. (Típicamente 512 bytes).
- Utilizados para dispositivos de almacenamiento.
- El acceso por parte del usuario se hace a través de un stream de bytes.

Block Device

- El kernel accede a los datos a través de bloques. (Típicamente 512 bytes).
- Utilizados para dispositivos de almacenamiento.
- El acceso por parte del usuario se hace a través de un stream de bytes.
- Para acceder al mismo se utilizan las syscalls básicas: `read ()`, `write ()`, `open ()`, `close ()`.

Block Device

- El kernel accede a los datos a través de bloques. (Típicamente 512 bytes).
- Utilizados para dispositivos de almacenamiento.
- El acceso por parte del usuario se hace a través de un stream de bytes.
- Para acceder al mismo se utilizan las syscalls básicas: `read ()`, `write ()`, `open ()`, `close ()`.
- El acceso se hace a través de los nodos del File System donde están declarados (Ej: /dev/sda1).

Network Interface

Network Interface

- Puede ser interfaz por HW o SW (Loopback).

Network Interface

- Puede ser interfaz por HW o SW (Loopback).
- Su objetivo es enviar y recibir paquetes de datos desde y hacia el gestor de redes del kernel.

Network Interface

- Puede ser interfaz por HW o SW (Loopback).
- Su objetivo es enviar y recibir paquetes de datos desde y hacia el gestor de redes del kernel.
- Solo maneja paquetes de datos. No tiene en cuenta los tipos de conexiones ni protocolos. (Lease: TCP / UDP , etc).

Network Interface

- Puede ser interfaz por HW o SW (Loopback).
- Su objetivo es enviar y recibir paquetes de datos desde y hacia el gestor de redes del kernel.
- Solo maneja paquetes de datos. No tiene en cuenta los tipos de conexiones ni protocolos. (Lease: TCP / UDP , etc).
- No son stream-oriented. Es decir no se acceden utilizando read, write, etc...

Miscelaneaous Devices

Miscelaneaous Devices

- En general esta categoría agrupa a cualquier dispositivo o subsistema cuyas características le impiden clasificar en alguna de las tres categorías anteriores.

Miscelaneaous Devices

- En general esta categoría agrupa a cualquier dispositivo o subsistema cuyas características le impiden clasificar en alguna de las tres categorías anteriores.
- Desde hace algún tiempo se refiere a esta categoría de drivers como ***platform devices***, en la que se agrupan los drivers de los controladores de buses, ya que no son específicos para un dispositivo de hahrdware sino para un subsistema al que se conectan diversos dispositivos de hardware.

Miscelaneaous Devices

- En general esta categoría agrupa a cualquier dispositivo o subsistema cuyas características le impiden clasificar en alguna de las tres categorías anteriores.
- Desde hace algún tiempo se refiere a esta categoría de drivers como ***platform devices***, en la que se agrupan los drivers de los controladores de buses, ya que no son específicos para un dispositivo de hahrdware sino para un subsistema al que se conectan diversos dispositivos de hardware.

PCI

Miscelaneaous Devices

- En general esta categoría agrupa a cualquier dispositivo o subsistema cuyas características le impiden clasificar en alguna de las tres categorías anteriores.
- Desde hace algún tiempo se refiere a esta categoría de drivers como ***platform devices***, en la que se agrupan los drivers de los controladores de buses, ya que no son específicos para un dispositivo de hahrdware sino para un subsistema al que se conectan diversos dispositivos de hardware.

PCI USB

Miscelaneaous Devices

- En general esta categoría agrupa a cualquier dispositivo o subsistema cuyas características le impiden clasificar en alguna de las tres categorías anteriores.
- Desde hace algún tiempo se refiere a esta categoría de drivers como ***platform devices***, en la que se agrupan los drivers de los controladores de buses, ya que no son específicos para un dispositivo de hahrdware sino para un subsistema al que se conectan diversos dispositivos de hardware.

PCI USB SCSI

Miscelaneaous Devices

- En general esta categoría agrupa a cualquier dispositivo o subsistema cuyas características le impiden clasificar en alguna de las tres categorías anteriores.
- Desde hace algún tiempo se refiere a esta categoría de drivers como ***platform devices***, en la que se agrupan los drivers de los controladores de buses, ya que no son específicos para un dispositivo de hahrdware sino para un subsistema al que se conectan diversos dispositivos de hardware.

PCI USB SCSI I2C

Miscelaneaous Devices

- En general esta categoría agrupa a cualquier dispositivo o subsistema cuyas características le impiden clasificar en alguna de las tres categorías anteriores.
- Desde hace algún tiempo se refiere a esta categoría de drivers como ***platform devices***, en la que se agrupan los drivers de los controladores de buses, ya que no son específicos para un dispositivo de hahrdware sino para un subsistema al que se conectan diversos dispositivos de hardware.

PCI USB SCSI I2C SPI

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Linkeable Kernel Modules (LKM)

Linkeable Kernel Modules (LKM)

- Escribir un device driver, es escribir código de kernel

Linkeable Kernel Modules (LKM)

- Escribir un device driver, es escribir código de kernel
- En modo kernel se dispone de un tipo especial de programa denominado Módulo (kernel module).

Linkeable Kernel Modules (LKM)

- Escribir un device driver, es escribir código de kernel
- En modo kernel se dispone de un tipo especial de programa denominado Módulo (kernel module).
- Una aplicación convencional realiza una tarea única del principio hasta el fin.

Linkeable Kernel Modules (LKM)

- Escribir un device driver, es escribir código de kernel
- En modo kernel se dispone de un tipo especial de programa denominado Módulo (kernel module).
- Una aplicación convencional realiza una tarea única del principio hasta el fin.
- Un módulo, en cambio, se registra a si mismo a fin de prestar servicios a futuro. Su función principal es efímera, pero queda “instalado” en el sistema.

Linkeable Kernel Modules (LKM)

- Escribir un device driver, es escribir código de kernel
- En modo kernel se dispone de un tipo especial de programa denominado Módulo (kernel module).
- Una aplicación convencional realiza una tarea única del principio hasta el fin.
- Un módulo, en cambio, se registra a si mismo a fin de prestar servicios a futuro. Su función principal es efímera, pero queda “instalado” en el sistema.
- Un módulo en definitiva se comporta como un plugin. Se carga cuando se lo necesita y se lo descarga cuando ya no se lo utiliza.

Linkeable Kernel Modules (LKM)

- Escribir un device driver, es escribir código de kernel
- En modo kernel se dispone de un tipo especial de programa denominado Módulo (kernel module).
- Una aplicación convencional realiza una tarea única del principio hasta el fin.
- Un módulo, en cambio, se registra a si mismo a fin de prestar servicios a futuro. Su función principal es efímera, pero queda "instalado" en el sistema.
- Un módulo en definitiva se comporta como un plugin. Se carga cuando se lo necesita y se lo descarga cuando ya no se lo utiliza.
- Solo que es un plugin del kernel.

Linkeable Kernel Modules (LKM)

- Escribir un device driver, es escribir código de kernel
- En modo kernel se dispone de un tipo especial de programa denominado Módulo (kernel module).
- Una aplicación convencional realiza una tarea única del principio hasta el fin.
- Un módulo, en cambio, se registra a si mismo a fin de prestar servicios a futuro. Su función principal es efímera, pero queda "instalado" en el sistema.
- Un módulo en definitiva se comporta como un plugin. Se carga cuando se lo necesita y se lo descarga cuando ya no se lo utiliza.
- Solo que es un plugin del kernel.
- Para que el kernel soporte el linkeo dinámico de módulos a su propio código debe ser compilado con la opción `CONFIG_MODULES=y`

Algunas características de los LKM

Algunas características de los LKM

- No se ejecutan secuencialmente.

Algunas características de los LKM

- No se ejecutan secuencialmente.
- No cuentan con un entry point (main).

Algunas características de los LKM

- No se ejecutan secuencialmente.
- No cuentan con un entry point (main).
- El módulo se registra en el kernel y espera requests.

Algunas características de los LKM

- No se ejecutan secuencialmente.
- No cuentan con un entry point (main).
- El módulo se registra en el kernel y espera requests.
- Todos los recursos solicitados por el kernel DEBEN ser correctamente eliminados al momento de quitar el módulo del kernel.

Algunas características de los LKM

- No se ejecutan secuencialmente.
- No cuentan con un entry point (main).
- El módulo se registra en el kernel y espera requests.
- Todos los recursos solicitados por el kernel DEBEN ser correctamente eliminados al momento de quitar el módulo del kernel.
- Pueden ser utilizados por mas de un cliente a la vez e interrumpidos.

Algunas características de los LKM

- No se ejecutan secuencialmente.
- No cuentan con un entry point (main).
- El módulo se registra en el kernel y espera requests.
- Todos los recursos solicitados por el kernel DEBEN ser correctamente eliminados al momento de quitar el módulo del kernel.
- Pueden ser utilizados por mas de un cliente a la vez e interrumpidos.
- Mas detalles acerca de como compilar un módulo: https://www.kernel.org/doc/Documentation/kbuild/modules.txt

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Requisitos para compilar y correr un módulo

Requisitos para compilar y correr un módulo

- Headers del Kernel instalados en el sistema.

Requisitos para compilar y correr un módulo

- Headers del Kernel instalados en el sistema.
- Para saber exactamente que versión de kernel tenemos instalada en el sistema:

Requisitos para compilar y correr un módulo

- Headers del Kernel instalados en el sistema.
- Para saber exactamente que versión de kernel tenemos instalada en el sistema:

```
1 ˜$ uname -r
2 4.15.0-58-generic
3 ˜$ sudo apt-get install linux-headers-‘uname -r‘
```

Requisitos para compilar y correr un módulo

- Headers del Kernel instalados en el sistema.
- Para saber exactamente que versión de kernel tenemos instalada en el sistema:

```
1 ˜$ uname −r
2 4.15.0−58−generic
3 ˜$ sudo apt−get install linux−headers−‘uname −r‘
```

- **make** (paquete binutils)

Requisitos para compilar y correr un módulo

- Headers del Kernel instalados en el sistema.
- Para saber exactamente que versión de kernel tenemos instalada en el sistema:

```
1 ~$ uname -r
2 4.15.0-58-generic
3 ~$ sudo apt-get install linux-headers-`uname -r`
```

- `make` (paquete binutils)
- `insmod` y `rmmod` para insertar y eliminar módulos del kernel.

Requisitos para compilar y correr un módulo

- Headers del Kernel instalados en el sistema.
- Para saber exactamente que versión de kernel tenemos instalada en el sistema:

```
1 ~$ uname -r
2 4.15.0-58-generic
3 ~$ sudo apt-get install linux-headers-`uname -r`
```

- **make** (paquete binutils)
- **insmod** y **rmmod** para insertar y eliminar módulos del kernel.
- El módulo debe ser re-compilado para cada versión de kernel.

Hola Mundo! Soy el Kernel

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("Alejandro Furfaro");
MODULE_VERSION("1.0");
MODULE_DESCRIPTION("HolaMundo LKM");

static int __init hello_init(void)
{
 printk(KERN_ALERT "Hola, mundo!. Soy el Kernel\n");
 return 0;
}
static void __exit hello_exit(void)
{
 printk(KERN_ALERT "Adiós, Mundo! \n");
}
module_init(hello_init);
module_exit(hello_exit);
```

printk

printk

`KERN_EMERG` Emergencia (el sistema está fuera de juego).

printk

KERN_EMERG	Emergencia (el sistema está fuera de juego).
KERN_ALERT	Problema serio (requiere acción inmediata).

printk

KERN_EMERG Emergencia (el sistema está fuera de juego).

KERN_ALERT Problema serio (requiere acción inmediata).

KERN_CRIT Condición crítica, normalmente asociada a falla de hardware o de software.

printk

KERN_EMERG	Emergencia (el sistema está fuera de juego).
KERN_ALERT	Problema serio (requiere acción inmediata).
KERN_CRIT	Condición crítica, normalmente asociada a falla de hardware o de software.
KERN_ERR	Utilizada para indicar condiciones de error, normalmente dificultades en el hardware.

printk

KERN_EMERG	Emergencia (el sistema está fuera de juego).
KERN_ALERT	Problema serio (requiere acción inmediata).
KERN_CRIT	Condición crítica, normalmente asociada a falla de hardware o de software.
KERN_ERR	Utilizada para indicar condiciones de error, normalmente dificultades en el hardware.
KERN_WARNING	Advertencia de situaciones problemáticas no tan serias.

printk

KERN_EMERG	Emergencia (el sistema está fuera de juego).
KERN_ALERT	Problema serio (requiere acción inmediata).
KERN_CRIT	Condición crítica, normalmente asociada a falla de hardware o de software.
KERN_ERR	Utilizada para indicar condiciones de error, normalmente dificultades en el hardware.
KERN_WARNING	Advertencia de situaciones problemáticas no tan serias.
KERN_NOTICE	Notificación de situaciones normales.

printk

KERN_EMERG	Emergencia (el sistema está fuera de juego).
KERN_ALERT	Problema serio (requiere acción inmediata).
KERN_CRIT	Condición crítica, normalmente asociada a falla de hardware o de software.
KERN_ERR	Utilizada para indicar condiciones de error, normalmente dificultades en el hardware.
KERN_WARNING	Advertencia de situaciones problemáticas no tan serias.
KERN_NOTICE	Notificación de situaciones normales.
KERN_INFO	Mensajes de información;utilizadas generalmente por los drivers.

printk

KERN_EMERG	Emergencia (el sistema está fuera de juego).
KERN_ALERT	Problema serio (requiere acción inmediata).
KERN_CRIT	Condición crítica, normalmente asociada a falla de hardware o de software.
KERN_ERR	Utilizada para indicar condiciones de error, normalmente dificultades en el hardware.
KERN_WARNING	Advertencia de situaciones problemáticas no tan serias.
KERN_NOTICE	Notificación de situaciones normales.
KERN_INFO	Mensajes de información;utilizadas generalmente por los drivers.
KERN_DEBUG	Mensajes para propósitos de debuging.

wrappers para printk

wrappers para printk

Para encapsular los detalles de `printk` y darle flexibilidad para su evolución futura, se han introducido una serie de wrappers, donde entre otros citamos:

wrappers para printk

Para encapsular los detalles de **printk** y darle flexibilidad para su evolución futura, se han introducido una serie de wrappers, donde entre otros citamos:

pr_info () Equivalente a utilizar **printk** con **KERN_INFO**.

wrappers para printk

Para encapsular los detalles de `printk` y darle flexibilidad para su evolución futura, se han introducido una serie de wrappers, donde entre otros citamos:

- `pr_info ()` Equivalente a utilizar `printk` con `KERN_INFO`.
- `pr_err ()` Equivalente a utilizar `printk` con `KERN_ERR`.

funciones de inicio y de finalización

funciones de inicio y de finalización

- Merece ser explicada la declaración de los prototipos de las funciones definidas por las macros `init_module ()` y `exit_module ()` :

funciones de inicio y de finalización

- Merece ser explicada la declaración de los prototipos de las funciones definidas por las macros `init_module ()` y `exit_module ()` :

```
1 static int __init hello_init(void)
2 static void __exit hello_exit(void)
```

funciones de inicio y de finalización

- Merece ser explicada la declaración de los prototipos de las funciones definidas por las macros **init_module ()** y **exit_module ()** :

```
1 static int __init hello_init(void)
2 static void __exit hello_exit(void)
```

- **__init** , y **__exit** , obran como atributos de las funciones definidas por las macros **init_module()** y **exit_module()** .

funciones de inicio y de finalización

- Merece ser explicada la declaración de los prototipos de las funciones definidas por las macros **init_module ()** y **exit_module ()** :

```
1 static int __init hello_init(void)
2 static void __exit hello_exit(void)
```

- **__init** , y **__exit** , obran como atributos de las funciones definidas por las macros **init_module()** y **exit_module()** .
- **__init** , y **__exit** , son macros definidas del siguiente modo en **include/linux/init** :

funciones de inicio y de finalización

- Merece ser explicada la declaración de los prototipos de las funciones definidas por las macros **init_module ()** y **exit_module ()** :

```
1 static int __init hello_init(void)
2 static void __exit hello_exit(void)
```

- **__init** , y **__exit** , obran como atributos de las funciones definidas por las macros **init_module()** y **exit_module()** .
- **__init** , y **__exit** , son macros definidas del siguiente modo en **include/linux/init** :

```
1 #define __init __section(.init.text)
2 #define __exit __section(.exit.text)
```

funciones de inicio y de finalización

- Merece ser explicada la declaración de los prototipos de las funciones definidas por las macros **init_module ()** y **exit_module ()** :

```
1 static int __init hello_init(void)
2 static void __exit hello_exit(void)
```

- **__init** , y **__exit** , obran como atributos de las funciones definidas por las macros **init_module()** y **exit_module()** .
- **__init** , y **__exit** , son macros definidas del siguiente modo en **include/linux/init** :

```
1 #define __init __section(.init.text)
2 #define __exit __section(.exit.text)
```

- Las funciones de inicialización y desisntalación se ubican en una sección exclusiva

ELF Header del módulo

```
alejandro@DarkSideOfTheMoon:~/work/facu/TDIII/ProgramasClase/lkm$ objdump -h hello.ko

hello.ko:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .note.gnu.build-id 00000024  0000000000000000  0000000000000000  00000040  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text         00000000  0000000000000000  0000000000000000  00000064  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .init.text    00000019  0000000000000000  0000000000000000  00000064  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  3 .exit.text    00000012  0000000000000000  0000000000000000  0000007d  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  4 .rodata.str1.8 00000051  0000000000000000  0000000000000000  00000090  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .modinfo      000000eb  0000000000000000  0000000000000000  000000e8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 __mcount_loc  00000008  0000000000000000  0000000000000000  000001d8  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
  7 .data         00000000  0000000000000000  0000000000000000  000001e0  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  8 .gnu.linkonce.this_module 00000340  0000000000000000  0000000000000000  00000200  2**6
                  CONTENTS, ALLOC, LOAD, RELOC, DATA, LINK_ONCE_DISCARD
  9 .bss          00000000  0000000000000000  0000000000000000  00000540  2**0
                  ALLOC
 10 .comment      00000058  0000000000000000  0000000000000000  00000540  2**0
                  CONTENTS, READONLY
 11 .note.GNU-stack 00000000  0000000000000000  0000000000000000  00000598  2**0
                  CONTENTS, READONLY
alejandro@DarkSideOfTheMoon:~/work/facu/TDIII/ProgramasClase/lkm$
```

Ya tenemos los headers, y demás ¿y ahora?

Ya tenemos los headers, y demás ¿y ahora?

- Documentación adicional del kernel.
  https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation

Ya tenemos los headers, y demás ¿y ahora?

- Documentación adicional del kernel.
  https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation

- Linux Device Drivers - 3er.Edición - Alessandro Rubini- Ed. O'rreily. "**El libro**" para aprender a programar drivers. A pesar de ser publicado en 1998, describe conceptos y temas muy bien desarrollados, y que siguen vigentes (Se amagó con una 4ta. edición anunciada para 2018, pero no hay nada nuevo aun).

Ya tenemos los headers, y demás ¿y ahora?

- Documentación adicional del kernel.
  https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation

- Linux Device Drivers - 3er.Edición - Alessandro Rubini- Ed. O'rreily. "**El libro**" para aprender a programar drivers. A pesar de ser publicado en 1998, describe conceptos y temas muy bien desarrollados, y que siguen vigentes (Se amagó con una 4ta. edición anunciada para 2018, pero no hay nada nuevo aun).

- Linux Device Drivers Development - Develop customized Drivers for embedded Linux - John Madieu - Ed. Packt Birmingham Bombai. Octubre de 2017. Actualizado y excelente libro. Aunque no es tan completo como el anterior

compilación

compilación

- Se utiliza un `Makefile` como el siguiente en el mismo directorio de trabajo

compilación

- Se utiliza un **Makefile** como el siguiente en el mismo directorio de trabajo .

```
1 obj-m += helloWorldModule.o
2
3 all:
4   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
5
6 clean:
7   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

compilación

- Se utiliza un **Makefile** como el siguiente en el mismo directorio de trabajo .

```
1 obj-m += helloWorldModule.o
2
3 all:
4   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
5
6 clean:
7   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

- Se compila con **make** .

¿Que hace este Makefile?

la sentencia `obj-<X>`

¿Que hace este Makefile?

la sentencia `obj-<X>`

En prácticamente todos los `Makefile` del kernel de Linux puede verse una línea con el comando `obj-<X>` .

¿Que hace este Makefile?

la sentencia `obj-<X>`

En prácticamente todos los `Makefile` del kernel de Linux puede verse una línea con el comando `obj-<X>`.
El argumento X puede valer `y`, `m`, `n`, o un caracter en blanco.

¿Que hace este Makefile?

la sentencia `obj-<X>`

En prácticamente todos los `Makefile` del kernel de Linux puede verse una línea con el comando `obj-<X>` .

El argumento X puede valer `y` , `m` , `n` , o un caracter en blanco.

`obj-m` Se genera el objeto como un módulo (tal es el caso de nuestro `Makefile`).

¿Que hace este Makefile?

la sentencia `obj-<X>`

En prácticamente todos los `Makefile` del kernel de Linux puede verse una línea con el comando `obj-<X>` .
El argumento X puede valer `y` , `m` , `n` , o un caracter en blanco.

- `obj-m` Se genera el objeto como un módulo (tal es el caso de nuestro `Makefile`).
- `obj-y` El objeto se linkea como parte del kernel (no es un módulo, forma parte del kernel monolítico

¿Que hace este Makefile?

la sentencia `obj-<X>`

En prácticamente todos los `Makefile` del kernel de Linux puede verse una línea con el comando `obj-<X>` .
El argumento X puede valer `y` , `m` , `n` , o un caracter en blanco.

- **`obj-m`** Se genera el objeto como un módulo (tal es el caso de nuestro `Makefile`).
- **`obj-y`** El objeto se linkea como parte del kernel (no es un módulo, forma parte del kernel monolítico
- **`obj-n`** No se generará ningún módulo.

¿Que hace este Makefile?

¿Que hace este Makefile?

- Para poder construir un módulo se dispone de un prebuilt del kernel.

¿Que hace este Makefile?

- Para poder construir un módulo se dispone de un prebuilt del kernel.
- Este está en el directorio `/lib/modules/`uname -r`/build`.

¿Que hace este Makefile?

- Para poder construir un módulo se dispone de un prebuilt del kernel.
- Este está en el directorio `/lib/modules/`uname -r`/build`.
- De modo que la línea:

¿Que hace este Makefile?

- Para poder construir un módulo se dispone de un prebuilt del kernel.
- Este está en el directorio `/lib/modules/`uname -r`/build`.
- De modo que la línea:

```
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

¿Que hace este Makefile?

- Para poder construir un módulo se dispone de un prebuilt del kernel.
- Este está en el directorio `/lib/modules/`uname -r`/build`.
- De modo que la línea:

```
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

- Utiliza este directorio para trabajar nuestro módulo.

¿Que hace este Makefile?

- Para poder construir un módulo se dispone de un prebuilt del kernel.
- Este está en el directorio `/lib/modules/`uname -r`/build`.
- De modo que la línea:

```
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

- Utiliza este directorio para trabajar nuestro módulo.
- La opción **`-C`** le indica a **`make`** , que ingrese al directorio antes de leer un **`Makefile`** o cualquier otra acción que se le indique.

¿Que hace este Makefile?

- Para poder construir un módulo se dispone de un prebuilt del kernel.
- Este está en el directorio `/lib/modules/`uname -r`/build`.
- De modo que la línea:

```
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

- Utiliza este directorio para trabajar nuestro módulo.
- La opción **`-C`** le indica a **`make`**, que ingrese al directorio antes de leer un **`Makefile`** o cualquier otra acción que se le indique.
- Con **`M=$(PWD)`** se le indica ala prebuilt del kernel el path en el que está el módulo que se desea construir.

¿Que hace este Makefile?

- Para poder construir un módulo se dispone de un prebuilt del kernel.
- Este está en el directorio `/lib/modules/`uname -r`/build`.
- De modo que la línea:

```
make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

- Utiliza este directorio para trabajar nuestro módulo.
- La opción **`-C`** le indica a **`make`**, que ingrese al directorio antes de leer un **`Makefile`** o cualquier otra acción que se le indique.
- Con **`M=$(PWD)`** se le indica ala prebuilt del kernel el path en el que está el módulo que se desea construir.
- **`modules`** indica que ejecute este target en caso de no encontrarse targets denominados **`all`** o **`default`**.

Instalación y remoción

1. Se utilizan los comandos `insmod` , `rmmod` , y `modprobe` .

Instalación y remoción

1. Se utilizan los comandos `insmod` , `rmmod` , y `modprobe` .
2. `insmod` o `modprobe` lo instalan.

Instalación y remoción

1. Se utilizan los comandos `insmod`, `rmmod`, y `modprobe`.
2. `insmod` o `modprobe` lo instalan.
3. `insmod` hace que se ejecute la función `module_init()`.

Instalación y remoción

1. Se utilizan los comandos `insmod`, `rmmod`, y `modprobe`.
2. `insmod` o `modprobe` lo instalan.
3. `insmod` hace que se ejecute la función `module_init()`.
4. `rmmod`, o `modprobe -r` hacen lo propio con `module_exit()`.

Instalación y remoción

1. Se utilizan los comandos `insmod`, `rmmod`, y `modprobe`.
2. `insmod` o `modprobe` lo instalan.
3. `insmod` hace que se ejecute la función `module_init()`.
4. `rmmod`, o `modprobe -r` hacen lo propio con `module_exit()`.
5. `module_init()` y `module_exit()` se comportan como función constructora o destructora.

Instalación y remoción

1. Se utilizan los comandos `insmod`, `rmmod`, y `modprobe`.
2. `insmod` o `modprobe` lo instalan.
3. `insmod` hace que se ejecute la función `module_init()`.
4. `rmmod`, o `modprobe -r` hacen lo propio con `module_exit()`.
5. `module_init()` y `module_exit()` se comportan como función constructora o destructora.
6. Instalar el módulo

Instalación y remoción

1. Se utilizan los comandos `insmod`, `rmmod`, y `modprobe`.
2. `insmod` o `modprobe` lo instalan.
3. `insmod` hace que se ejecute la función `module_init()`.
4. `rmmod`, o `modprobe -r` hacen lo propio con `module_exit()`.
5. `module_init()` y `module_exit()` se comportan como función constructora o destructora.
6. Instalar el módulo

```
$ insmod helloWorldModule.ko
```

Instalación y remoción

1. Se utilizan los comandos `insmod`, `rmmod`, y `modprobe`.
2. `insmod` o `modprobe` lo instalan.
3. `insmod` hace que se ejecute la función `module_init()`.
4. `rmmod`, o `modprobe -r` hacen lo propio con `module_exit()`.
5. `module_init()` y `module_exit()` se comportan como función constructora o destructora.
6. Instalar el módulo
```
$ insmod helloWorldModule.ko
```
7. Mirar su inserción en `/proc/modules`:

Instalación y remoción

1. Se utilizan los comandos **insmod**, **rmmod**, y **modprobe**.
2. **insmod** o **modprobe** lo instalan.
3. **insmod** hace que se ejecute la función **module_init()**.
4. **rmmod**, o **modprobe -r** hacen lo propio con **module_exit()**.
5. **module_init()** y **module_exit()** se comportan como función constructora o destructora.
6. Instalar el módulo
   ```
   $ insmod helloWorldModule.ko
   ```
7. Mirar su inserción en **/proc/modules**:
   ```
   $ grep hello /proc/modules
   ```

Instalación y remoción

1. Se utilizan los comandos `insmod`, `rmmod`, y `modprobe`.
2. `insmod` o `modprobe` lo instalan.
3. `insmod` hace que se ejecute la función `module_init()`.
4. `rmmod`, o `modprobe -r` hacen lo propio con `module_exit()`.
5. `module_init()` y `module_exit()` se comportan como función constructora o destructora.
6. Instalar el módulo
```
$ insmod helloWorldModule.ko
```
7. Mirar su inserción en `/proc/modules`:
```
$ grep hello /proc/modules
```
8. Remover el módulo.

Instalación y remoción

1. Se utilizan los comandos `insmod`, `rmmod`, y `modprobe`.
2. `insmod` o `modprobe` lo instalan.
3. `insmod` hace que se ejecute la función `module_init()`.
4. `rmmod`, o `modprobe -r` hacen lo propio con `module_exit()`.
5. `module_init()` y `module_exit()` se comportan como función constructora o destructora.
6. Instalar el módulo
```
$ insmod helloWorldModule.ko
```
7. Mirar su inserción en `/proc/modules`:
```
$ grep hello /proc/modules
```
8. Remover el módulo.
```
$ rmmod helloWorldModule.o
```

Instalación y remoción

1. Se utilizan los comandos `insmod`, `rmmod`, y `modprobe`.
2. `insmod` o `modprobe` lo instalan.
3. `insmod` hace que se ejecute la función `module_init()`.
4. `rmmod`, o `modprobe -r` hacen lo propio con `module_exit()`.
5. `module_init()` y `module_exit()` se comportan como función constructora o destructora.
6. Instalar el módulo
```
$ insmod helloWorldModule.ko
```
7. Mirar su inserción en `/proc/modules`:
```
$ grep hello /proc/modules
```
8. Remover el módulo.
```
$ rmmod helloWorldModule.o
```
9. Observar los resultados:

Instalación y remoción

1. Se utilizan los comandos **insmod**, **rmmod**, y **modprobe**.
2. **insmod** o **modprobe** lo instalan.
3. **insmod** hace que se ejecute la función **module_init()**.
4. **rmmod**, o **modprobe -r** hacen lo propio con **module_exit()**.
5. **module_init()** y **module_exit()** se comportan como función constructora o destructora.
6. Instalar el módulo

```
$ insmod helloWorldModule.ko
```

7. Mirar su inserción en **/proc/modules**:

```
$ grep hello /proc/modules
```

8. Remover el módulo.

```
$ rmmod helloWorldModule.o
```

9. Observar los resultado

```
$ dmesg | grep mundo
Hola, mundo!. Soy el Kernel
Adiós mundo!.
$
```

modprobe depmod

modprobe depmod

- En producción se prefiere instalar módulos con `modprobe` .

modprobe depmod

- En producción se prefiere instalar módulos con `modprobe` .
- `modprobe` evalúa las dependencias del módulo antes de instalarlo parseando el archivo `modules.dep` , y manejando de manera automática las dependencias del módulo antes de cargarlo tal como lo haría un gestor de paquetes.

modprobe depmod

- En producción se prefiere instalar módulos con `modprobe` .
- `modprobe` evalúa las dependencias del módulo antes de instalarlo parseando el archivo `modules.dep` , y manejando de manera automática las dependencias del módulo antes de cargarlo tal como lo haría un gestor de paquetes.
- Las dependencias se generan con otra utilidad llamada `depmod` .

modprobe depmod

- En producción se prefiere instalar módulos con `modprobe` .
- `modprobe` evalúa las dependencias del módulo antes de instalarlo parseando el archivo `modules.dep` , y manejando de manera automática las dependencias del módulo antes de cargarlo tal como lo haría un gestor de paquetes.
- Las dependencias se generan con otra utilidad llamada `depmod` .
- `depmod` además procesa los archivos del módulo, del que recolecta información relacionada con el hardware que el desarrollador debería incluir, como vendor ID y devices que soporta el driver. Con esa información de cada módulo arma el archivo `modules.alias` , en el directorio /lib/modules/<kernel-release >.

modprobe depmod

- En producción se prefiere instalar módulos con `modprobe`.
- `modprobe` evalúa las dependencias del módulo antes de instalarlo parseando el archivo `modules.dep`, y manejando de manera automática las dependencias del módulo antes de cargarlo tal como lo haría un gestor de paquetes.
- Las dependencias se generan con otra utilidad llamada `depmod`.
- `depmod` además procesa los archivos del módulo, del que recolecta información relacionada con el hardware que el desarrollador debería incluir, como vendor ID y devices que soporta el driver. Con esa información de cada módulo arma el archivo `modules.alias`, en el directorio /lib/modules/<kernel-release >.
- Para generar las dependencias, una opción es la siguiente

modprobe depmod

- En producción se prefiere instalar módulos con **modprobe** .
- **modprobe** evalúa las dependencias del módulo antes de instalarlo parseando el archivo **modules.dep** , y manejando de manera automática las dependencias del módulo antes de cargarlo tal como lo haría un gestor de paquetes.
- Las dependencias se generan con otra utilidad llamada **depmod** .
- **depmod** además procesa los archivos del módulo, del que recolecta información relacionada con el hardware que el desarrollador debería incluir, como vendor ID y devices que soporta el driver. Con esa información de cada módulo arma el archivo **modules.alias** , en el directorio /lib/modules/<kernel-release >.
- Para generar las dependencias, una opción es la siguiente

```
$ sudo ln -s /path/a/Nuestro/KernelModule /lib/modules/`uname -r`
$ sudo depmod -a
$ sudo modprobe KernelModule
```

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Kernel 2.4

El desarrollo de la versión 2.6 del kernel solucionó varias deficiencias del modesto modelo de drivers de la versión 2.4, a saber:

Kernel 2.4

El desarrollo de la versión 2.6 del kernel solucionó varias deficiencias del modesto modelo de drivers de la versión 2.4, a saber:

1. Falta de un método unificado para representar las relaciones entre los controladores y los dispositivos.

Kernel 2.4

El desarrollo de la versión 2.6 del kernel solucionó varias deficiencias del modesto modelo de drivers de la versión 2.4, a saber:

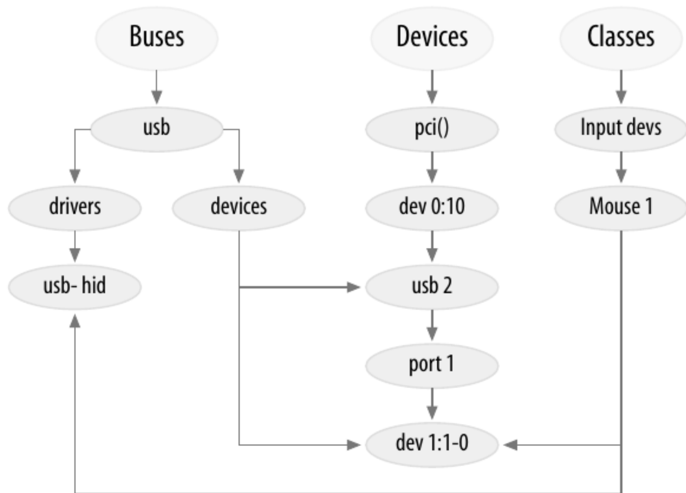
1. Falta de un método unificado para representar las relaciones entre los controladores y los dispositivos.
2. Falta de un mecanismo hotplug estándar.

Kernel 2.4

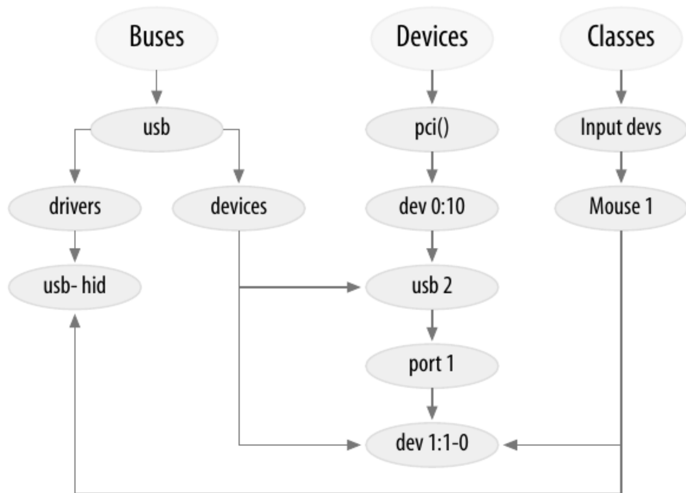
El desarrollo de la versión 2.6 del kernel solucionó varias deficiencias del modesto modelo de drivers de la versión 2.4, a saber:

1. Falta de un método unificado para representar las relaciones entre los controladores y los dispositivos.
2. Falta de un mecanismo hotplug estándar.
3. En la versión 2.4, el file system en el que se exportan los dispositivos de hardware (`procfs`) está plagado de información que no es de los dispositivos solamente sino también de procesos y otras entidades componentes del Sistema.

Estructura

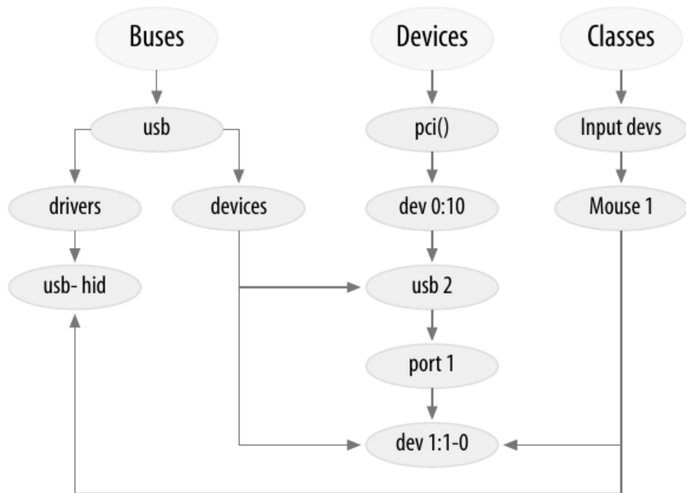


Estructura



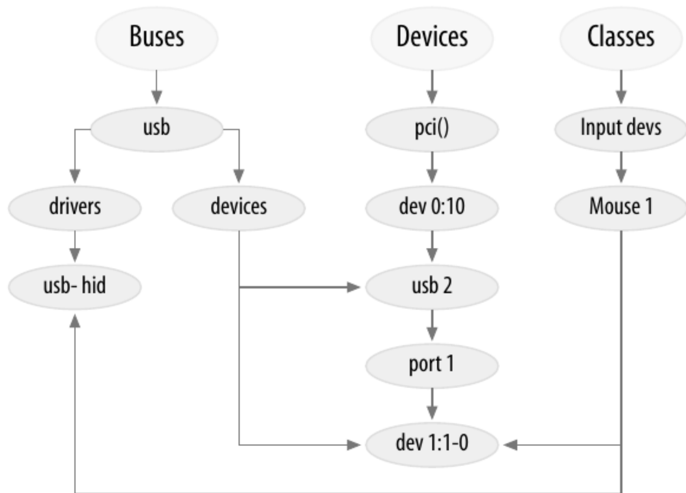
- Esta es una pequeña porción de la compleja estructura que conforma el Linux Driver Model.

Estructura



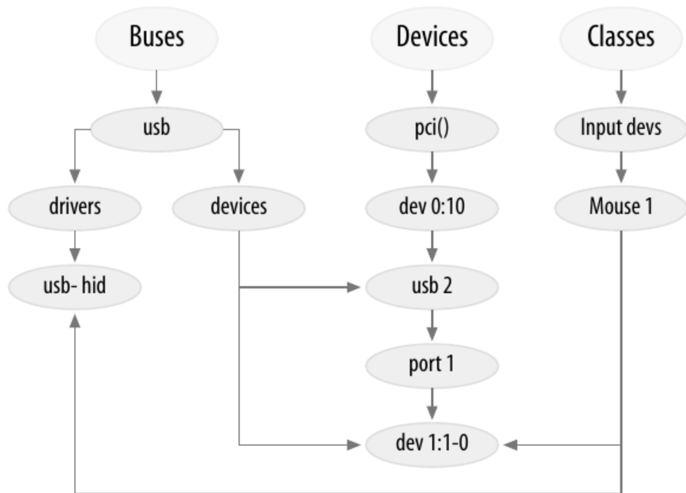
- Esta es una pequeña porción de la compleja estructura que conforma el Linux Driver Model.
- En el centro de la figura el core `Devices` muestra en este caso **como** un mouse se conecta al sistema.

Estructura



- Esta es una pequeña porción de la compleja estructura que conforma el Linux Driver Model.
- En el centro de la figura el core `Devices` muestra en este caso **como** un mouse se conecta al sistema.
- El árbol `Buses` permite conocer **que** esta conectado a cada bus.

Estructura



- Esta es una pequeña porción de la compleja estructura que conforma el Linux Driver Model.
- En el centro de la figura el core `Devices` muestra en este caso **como** un mouse se conecta al sistema.
- El árbol `Buses` permite conocer **que** esta conectado a cada bus.
- El árbol `Classes` mantiene la información de la función que provee cada dispositivo, sin importar como ni a donde esté conectado.

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Comunicación con el User Space: **sysfs**

Comunicación con el User Space: **sysfs**

`sysfs` es un File System virtual no persistente, que exporta hacia el espacio del usuario información sobre la estructura de los dispositivos y sus controladores del modelo de dispositivos del kernel, mediante sus `kobjects`

Cada `kobject` se muestra como un nodo de ese Fie System (archivo, directorio etc). Los archivos se denominan atributos y pueden escribirse para configurar parámetros o leerse para accederlos eventualmente.

Fue introducida por la versión de kernel 2.6.

Comunicación con el User Space: **sysfs**

`sysfs` es un File System virtual no persistente, que exporta hacia el espacio del usuario información sobre la estructura de los dispositivos y sus controladores del modelo de dispositivos del kernel, mediante sus `kobjects`

Cada `kobject` se muestra como un nodo de ese Fie System (archivo, directorio etc). Los archivos se denominan atributos y pueden escribirse para configurar parámetros o leerse para accederlos eventualmente.

Fue introducida por la versión de kernel 2.6.

```
alejandro@Meddle:~$ tree /sys -L 1
/sys
├── block
├── bus
├── class
├── dev
├── devices
├── firmware
├── fs
├── hypervisor
├── kernel
├── module
└── power

11 directories, 0 files
alejandro@Meddle:~$
```

Comunicación con el User Space: **sysfs**

`sysfs` es un File System virtual no persistente, que exporta hacia el espacio del usuario información sobre la estructura de los dispositivos y sus controladores del modelo de dispositivos del kernel, mediante sus `kobjects`

Cada `kobject` se muestra como un nodo de ese Fie System (archivo, directorio etc). Los archivos se denominan atributos y pueden escribirse para configurar parámetros o leerse para accederlos eventualmente.

Fue introducida por la versión de kernel 2.6.

```
alejandro@Meddle:~$ tree /sys -L 1
/sys
├── block
├── bus
├── class
├── dev
├── devices
├── firmware
├── fs
├── hypervisor
├── kernel
├── module
└── power

11 directories, 0 files
alejandro@Meddle:~$
```

Estructura sysfs

Comunicación con el User Space: **sysfs**

`sysfs` es un File System virtual no persistente, que exporta hacia el espacio del usuario información sobre la estructura de los dispositivos y sus controladores del modelo de dispositivos del kernel, mediante sus **`kobjects`**

Cada **`kobject`** se muestra como un nodo de ese Fie System (archivo, directorio etc). Los archivos se denominan atributos y pueden escribirse para configurar parámetros o leerse para accederlos eventualmente.

Fue introducida por la versión de kernel 2.6.

```
alejandro@Meddle:~$ tree /sys -L 1
/sys
├── block
├── bus
├── class
├── dev
├── devices
├── firmware
├── fs
├── hypervisor
├── kernel
├── module
└── power

11 directories, 0 files
alejandro@Meddle:~$
```

Estructura sysfs

`/sys/block` Contiene los dispositivos de bloques encontrados en el sistema.

Comunicación con el User Space: **sysfs**

`sysfs` es un File System virtual no persistente, que exporta hacia el espacio del usuario información sobre la estructura de los dispositivos y sus controladores del modelo de dispositivos del kernel, mediante sus `kobjects`
Cada `kobject` se muestra como un nodo de ese Fie System (archivo, directorio etc). Los archivos se denominan atributos y pueden escribirse para configurar parámetros o leerse para accederlos eventualmente.
Fue introducida por la versión de kernel 2.6.

```
alejandro@Meddle:~$ tree /sys -L 1
/sys
├── block
├── bus
├── class
├── dev
├── devices
├── firmware
├── fs
├── hypervisor
├── kernel
├── module
└── power

11 directories, 0 files
alejandro@Meddle:~$
```

Estructura sysfs

`/sys/block` Contiene los dispositivos de bloques encontrados en el sistema.

`/sys/bus` Contiene los tipos de buses del sistema. (Un subdirectorio por cada tipo de bus encontrado).

Comunicación con el User Space: **sysfs**

`sysfs` es un File System virtual no persistente, que exporta hacia el espacio del usuario información sobre la estructura de los dispositivos y sus controladores del modelo de dispositivos del kernel, mediante sus `kobjects`

Cada `kobject` se muestra como un nodo de ese Fie System (archivo, directorio etc). Los archivos se denominan atributos y pueden escribirse para configurar parámetros o leerse para accederlos eventualmente.

Fue introducida por la versión de kernel 2.6.

```
alejandro@Meddle:~$ tree /sys -L 1
/sys
├── block
├── bus
├── class
├── dev
├── devices
├── firmware
├── fs
├── hypervisor
├── kernel
├── module
└── power

11 directories, 0 files
alejandro@Meddle:~$
```

Estructura sysfs

`/sys/block`	Contiene los dispositivos de bloques encontrados en el sistema.
`/sys/bus`	Contiene los tipos de buses del sistema. (Un subdirectorio por cada tipo de bus encontrado).
`/sys/class`	Organiza en clases los dispositivos presentes en el sistema (Ej: input, sound, tty).

Comunicación con el User Space: sysfs

sysfs es un File System virtual no persistente, que exporta hacia el espacio del usuario información sobre la estructura de los dispositivos y sus controladores del modelo de dispositivos del kernel, mediante sus **kobjects**

Cada **kobject** se muestra como un nodo de ese Fie System (archivo, directorio etc). Los archivos se denominan atributos y pueden escribirse para configurar parámetros o leerse para accederlos eventualmente.

Fue introducida por la versión de kernel 2.6.

```
alejandro@Meddle:~$ tree /sys -L 1
/sys
├── block
├── bus
├── class
├── dev
├── devices
├── firmware
├── fs
├── hypervisor
├── kernel
├── module
└── power

11 directories, 0 files
alejandro@Meddle:~$
```

Estructura sysfs

/sys/devices Brinda una vista de la topología de los dispositivos.

Comunicación con el User Space: **sysfs**

`sysfs` es un File System virtual no persistente, que exporta hacia el espacio del usuario información sobre la estructura de los dispositivos y sus controladores del modelo de dispositivos del kernel, mediante sus `kobjects`

Cada `kobject` se muestra como un nodo de ese Fie System (archivo, directorio etc). Los archivos se denominan atributos y pueden escribirse para configurar parámetros o leerse para accederlos eventualmente.

Fue introducida por la versión de kernel 2.6.

```
alejandro@Meddle:~$ tree /sys -L 1
/sys
├── block
├── bus
├── class
├── dev
├── devices
├── firmware
├── fs
├── hypervisor
├── kernel
├── module
└── power

11 directories, 0 files
alejandro@Meddle:~$
```

Estructura sysfs

`/sys/devices` Brinda una vista de la topología de los dispositivos.

`/sys/dev` Brinda una vista “en crudo” de los dispositivos presentes en el sistema sin organizarlos jerárquicamente (cada uno es un link al real en `/sys/devices`).

Comunicación con el User Space: **sysfs**

`sysfs` es un File System virtual no persistente, que exporta hacia el espacio del usuario información sobre la estructura de los dispositivos y sus controladores del modelo de dispositivos del kernel, mediante sus `kobjects`

Cada `kobject` se muestra como un nodo de ese Fie System (archivo, directorio etc). Los archivos se denominan atributos y pueden escribirse para configurar parámetros o leerse para accederlos eventualmente.

Fue introducida por la versión de kernel 2.6.

```
alejandro@Meddle:~$ tree /sys -L 1
/sys
├── block
├── bus
├── class
├── dev
├── devices
├── firmware
├── fs
├── hypervisor
├── kernel
├── module
└── power

11 directories, 0 files
alejandro@Meddle:~$
```

Estructura sysfs

`/sys/firmware` Muestra una vista de árbol sistema-específica de los subsistemas de bajo nivel (ACPI,EFI,etc.)

Comunicación con el User Space: **sysfs**

`sysfs` es un File System virtual no persistente, que exporta hacia el espacio del usuario información sobre la estructura de los dispositivos y sus controladores del modelo de dispositivos del kernel, mediante sus `kobjects`

Cada `kobject` se muestra como un nodo de ese Fie System (archivo, directorio etc). Los archivos se denominan atributos y pueden escribirse para configurar parámetros o leerse para accederlos eventualmente.

Fue introducida por la versión de kernel 2.6.

```
alejandro@Meddle:~$ tree /sys -L 1
/sys
├── block
├── bus
├── class
├── dev
├── devices
├── firmware
├── fs
├── hypervisor
├── kernel
├── module
└── power

11 directories, 0 files
alejandro@Meddle:~$
```

Estructura sysfs

`/sys/firmware`	Muestra una vista de árbol sistema-específica de los subsistemas de bajo nivel (ACPI,EFI,etc.)
`/sys/kernel`	Opciones de configuración del kernel e información de estado.

Comunicación con el User Space: **sysfs**

`sysfs` es un File System virtual no persistente, que exporta hacia el espacio del usuario información sobre la estructura de los dispositivos y sus controladores del modelo de dispositivos del kernel, mediante sus `kobjects`
Cada `kobject` se muestra como un nodo de ese Fie System (archivo, directorio etc). Los archivos se denominan atributos y pueden escribirse para configurar parámetros o leerse para accederlos eventualmente.
Fue introducida por la versión de kernel 2.6.

```
alejandro@Meddle:~$ tree /sys -L 1
/sys
├── block
├── bus
├── class
├── dev
├── devices
├── firmware
├── fs
├── hypervisor
├── kernel
├── module
└── power

11 directories, 0 files
alejandro@Meddle:~$
```

Estructura sysfs

`/sys/firmware`	Muestra una vista de árbol sistema-específica de los subsistemas de bajo nivel (ACPI,EFI,etc.)
`/sys/kernel`	Opciones de configuración del kernel e información de estado.
`/sys/power`	Power Managment de los dispositivos.

Comunicación con el User Space: **sysfs**

sysfs es un File System virtual no persistente, que exporta hacia el espacio del usuario información sobre la estructura de los dispositivos y sus controladores del modelo de dispositivos del kernel, mediante sus **kobjects**
Cada **kobject** se muestra como un nodo de ese Fie System (archivo, directorio etc). Los archivos se denominan atributos y pueden escribirse para configurar parámetros o leerse para accederlos eventualmente.
Fue introducida por la versión de kernel 2.6.

```
alejandro@Meddle:~$ tree /sys -L 1
/sys
├── block
├── bus
├── class
├── dev
├── devices
├── firmware
├── fs
├── hypervisor
├── kernel
├── module
└── power

11 directories, 0 files
alejandro@Meddle:~$
```

Estructura sysfs

/sys/fs	Información relacionada con los filesystems.

Comunicación con el User Space: **sysfs**

`sysfs` es un File System virtual no persistente, que exporta hacia el espacio del usuario información sobre la estructura de los dispositivos y sus controladores del modelo de dispositivos del kernel, mediante sus `kobjects`
Cada `kobject` se muestra como un nodo de ese Fie System (archivo, directorio etc). Los archivos se denominan atributos y pueden escribirse para configurar parámetros o leerse para accederlos eventualmente.
Fue introducida por la versión de kernel 2.6.

```
alejandro@Meddle:~$ tree /sys -L 1
/sys
├── block
├── bus
├── class
├── dev
├── devices
├── firmware
├── fs
├── hypervisor
├── kernel
├── module
└── power

11 directories, 0 files
alejandro@Meddle:~$
```

Estructura sysfs

`/sys/fs`	Información relacionada con los filesystems.
`/sys/module`	Lista de los módulos cargados en el sistema.

Comunicación con el User Space: **sysfs**

`sysfs` es un File System virtual no persistente, que exporta hacia el espacio del usuario información sobre la estructura de los dispositivos y sus controladores del modelo de dispositivos del kernel, mediante sus `kobjects`
Cada `kobject` se muestra como un nodo de ese Fie System (archivo, directorio etc). Los archivos se denominan atributos y pueden escribirse para configurar parámetros o leerse para accederlos eventualmente.
Fue introducida por la versión de kernel 2.6.

```
alejandro@Meddle:~$ tree /sys -L 1
/sys
├── block
├── bus
├── class
├── dev
├── devices
├── firmware
├── fs
├── hypervisor
├── kernel
├── module
└── power

11 directories, 0 files
alejandro@Meddle:~$
```

Estructura sysfs

`/sys/fs`	Información relacionada con los filesystems.
`/sys/module`	Lista de los módulos cargados en el sistema.

Interacción entre UserSpace y HW

Interacción entre UserSpace y HW

✓El uso de dispositivos se hace a través del manejo de archivos presentes en sysfs.

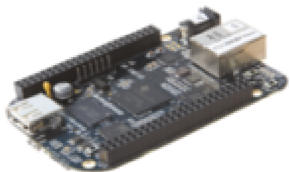
Interacción entre UserSpace y HW

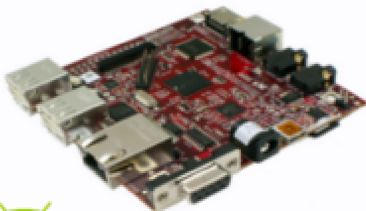
✓El uso de dispositivos se hace a través del manejo de archivos presentes en sysfs.
✓Al ser archivos, la interacción se hace a través de las syscalls de siempre: `open ()` , `read ()` , `write ()` , `close ()` .

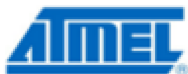
Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Plataformas ARM







Plataformas ARM mas difundidas y SO

Plataformas ARM mas difundidas y SO

- ARM plantea un escenario de amplia variedad de opciones para seleccionar el hardware mas adecuado a nuestra aplicación.

Plataformas ARM mas difundidas y SO

- ARM plantea un escenario de amplia variedad de opciones para seleccionar el hardware mas adecuado a nuestra aplicación.
- ARM provee el core bajo licencia y cada fabricante le agrega los dispositivos de entrada salida que considera apropiados, en las direcciones que le parecen adecuadas.

Plataformas ARM mas difundidas y SO

- ARM plantea un escenario de amplia variedad de opciones para seleccionar el hardware mas adecuado a nuestra aplicación.
- ARM provee el core bajo licencia y cada fabricante le agrega los dispositivos de entrada salida que considera apropiados, en las direcciones que le parecen adecuadas.
- Cada SoC termina alojando un set de dispositivos de E/S propios, mapeados en un espacio de direcciones de memoria propio, y con un layout de registros y demás recursos de hardware propio.

Plataformas ARM mas difundidas y SO

- ARM plantea un escenario de amplia variedad de opciones para seleccionar el hardware mas adecuado a nuestra aplicación.
- ARM provee el core bajo licencia y cada fabricante le agrega los dispositivos de entrada salida que considera apropiados, en las direcciones que le parecen adecuadas.
- Cada SoC termina alojando un set de dispositivos de E/S propios, mapeados en un espacio de direcciones de memoria propio, y con un layout de registros y demás recursos de hardware propio.
- Tenemos por eso una amplia variedad de SoCs para elegir la mas adecuada a nuestro proyecto.

Plataformas ARM mas difundidas y SO

- ARM plantea un escenario de amplia variedad de opciones para seleccionar el hardware mas adecuado a nuestra aplicación.
- ARM provee el core bajo licencia y cada fabricante le agrega los dispositivos de entrada salida que considera apropiados, en las direcciones que le parecen adecuadas.
- Cada SoC termina alojando un set de dispositivos de E/S propios, mapeados en un espacio de direcciones de memoria propio, y con un layout de registros y demás recursos de hardware propio.
- Tenemos por eso una amplia variedad de SoCs para elegir la mas adecuada a nuestro proyecto.
- Pero, ¿como puede un desarrollador construir un GPOS portable entre semejante diversidad de subsistemas de E/S?.

Plataformas ARM mas difundidas y SO

- ARM plantea un escenario de amplia variedad de opciones para seleccionar el hardware mas adecuado a nuestra aplicación.
- ARM provee el core bajo licencia y cada fabricante le agrega los dispositivos de entrada salida que considera apropiados, en las direcciones que le parecen adecuadas.
- Cada SoC termina alojando un set de dispositivos de E/S propios, mapeados en un espacio de direcciones de memoria propio, y con un layout de registros y demás recursos de hardware propio.
- Tenemos por eso una amplia variedad de SoCs para elegir la mas adecuada a nuestro proyecto.
- Pero, ¿como puede un desarrollador construir un GPOS portable entre semejante diversidad de subsistemas de E/S?.
- Este es el inconveniente cuando se decide portar a ARM un GPOS diseñado en un mundo donde todos los equipos son compatibles. (Los mismos periférios en las mismas direcciones).

Primer reacción de un “principiante”

Primer reacción de un “principiante”

*“Guys, this whole ARM thing is a f*ck’n pain in the ass!”*

Primer reacción de un “principiante”

*“Guys, this whole ARM thing is a f*ck’n pain in the ass!”*

Linus Torvalds

SoCs ARM vs. portabilidad del Kernel

SoCs ARM vs. portabilidad del Kernel

Todos los Cores de ARM tienen el mismo Register File y set de instrucciones.

SoCs ARM vs. portabilidad del Kernel

Todos los Cores de ARM tienen el mismo Register File y set de instrucciones.✓

SoCs ARM vs. portabilidad del Kernel

Todos los Cores de ARM tienen el mismo Register File y set de instrucciones.✓
Cada SoC combina un core con sus propios dispositivos de E/S. El core puede ser diferente o el mismo (por ejemplo un Cortex A15), pero los dispositivos de E/S cambian de un core a otro.

SoCs ARM vs. portabilidad del Kernel

Todos los Cores de ARM tienen el mismo Register File y set de instrucciones.✓
Cada SoC combina un core con sus propios dispositivos de E/S. El core puede ser diferente o el mismo (por ejemplo un Cortex A15), pero los dispositivos de E/S cambian de un core a otro. ☹

SoCs ARM vs. portabilidad del Kernel

Todos los Cores de ARM tienen el mismo Register File y set de instrucciones.✓

Cada SoC combina un core con sus propios dispositivos de E/S. El core puede ser diferente o el mismo (por ejemplo un Cortex A15), pero los dispositivos de E/S cambian de un core a otro. ☹

Cada SoC tiene su propio mapa de memoria.

SoCs ARM vs. portabilidad del Kernel

Todos los Cores de ARM tienen el mismo Register File y set de instrucciones.✓

Cada SoC combina un core con sus propios dispositivos de E/S. El core puede ser diferente o el mismo (por ejemplo un Cortex A15), pero los dispositivos de E/S cambian de un core a otro. ☹

Cada SoC tiene su propio mapa de memoria. ☹

SoCs ARM vs. portabilidad del Kernel

Todos los Cores de ARM tienen el mismo Register File y set de instrucciones.✓

Cada SoC combina un core con sus propios dispositivos de E/S. El core puede ser diferente o el mismo (por ejemplo un Cortex A15), pero los dispositivos de E/S cambian de un core a otro. ☹

Cada SoC tiene su propio mapa de memoria. ☹

A fin de cuentas, no hay un estándar definido, como lo hay para las arquitectura de las PCs.

SoCs ARM vs. portabilidad del Kernel

Todos los Cores de ARM tienen el mismo Register File y set de instrucciones.✓

Cada SoC combina un core con sus propios dispositivos de E/S. El core puede ser diferente o el mismo (por ejemplo un Cortex A15), pero los dispositivos de E/S cambian de un core a otro. ☹

Cada SoC tiene su propio mapa de memoria. ☹

A fin de cuentas, no hay un estándar definido, como lo hay para las arquitectura de las PCs. ☹

SoCs ARM vs. portabilidad del Kernel

Todos los Cores de ARM tienen el mismo Register File y set de instrucciones.✓
Cada SoC combina un core con sus propios dispositivos de E/S. El core puede ser diferente o el mismo (por ejemplo un Cortex A15), pero los dispositivos de E/S cambian de un core a otro. ☹
Cada SoC tiene su propio mapa de memoria. ☹
A fin de cuentas, no hay un estándar definido, como lo hay para las arquitectura de las PCs. ☹
No se puede compilar una sola imagen de kernel para ARM, como ocurre con la arquitectura x86. Esto es, una imagen binaria única (con módulos) que bootea en todas las PCs.

SoCs ARM vs. portabilidad del Kernel

Todos los Cores de ARM tienen el mismo Register File y set de instrucciones.✓

Cada SoC combina un core con sus propios dispositivos de E/S. El core puede ser diferente o el mismo (por ejemplo un Cortex A15), pero los dispositivos de E/S cambian de un core a otro. ☹

Cada SoC tiene su propio mapa de memoria. ☹

A fin de cuentas, no hay un estándar definido, como lo hay para las arquitectura de las PCs. ☹

No se puede compilar una sola imagen de kernel para ARM, como ocurre con la arquitectura x86. Esto es, una imagen binaria única (con módulos) que bootea en todas las PCs.☹

SoCs ARM vs. portabilidad del Kernel

Todos los Cores de ARM tienen el mismo Register File y set de instrucciones.✓

Cada SoC combina un core con sus propios dispositivos de E/S. El core puede ser diferente o el mismo (por ejemplo un Cortex A15), pero los dispositivos de E/S cambian de un core a otro. ☹

Cada SoC tiene su propio mapa de memoria. ☹

A fin de cuentas, no hay un estándar definido, como lo hay para las arquitectura de las PCs. ☹

No se puede compilar una sola imagen de kernel para ARM, como ocurre con la arquitectura x86. Esto es, una imagen binaria única (con módulos) que bootea en todas las PCs.☹

Es evidente que la cantidad de inconvenientes inclina la balanza. En estos casos, cualquier intento particular para un SoC no es escalable. Se requiere una reingeniería. De esto se trata el ***Device Tree***.

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Organización de un SoC ARM

Organización de un SoC ARM

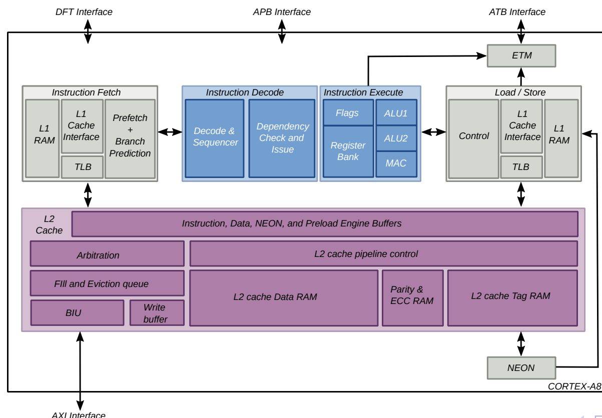
- ARM solo provee el core de procesamiento que se requiera.

Organización de un SoC ARM

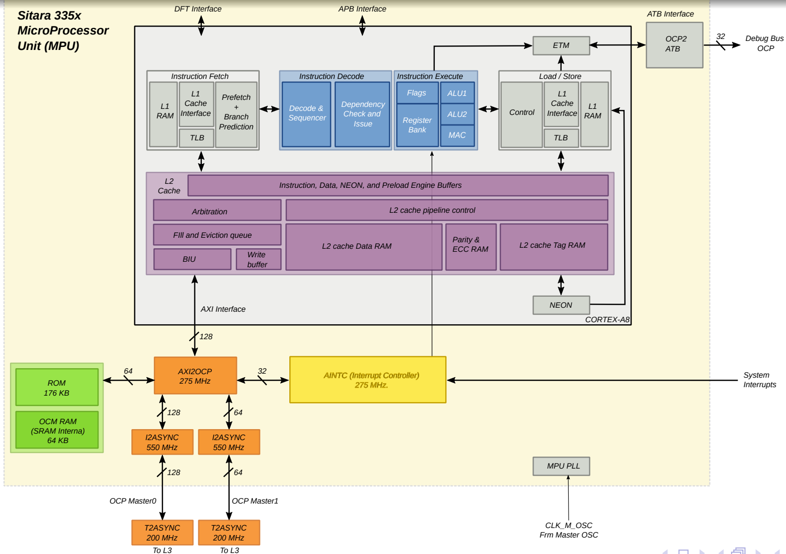
- ARM solo provee el core de procesamiento que se requiera.
- A los procesadores Cortex-A les agrega el subsistema Cache, y la interfaz denominada Advanced Trace Bus (ATB) para implementar debuging.

Organización de un SoC ARM

- ARM solo provee el core de procesamiento que se requiera.
- A los procesadores Cortex-A les agrega el subsistema Cache, y la interfaz denominada Advanced Trace Bus (ATB) para implementar debuging.



Organización de un SoC ARM: TI Sitara335x. MPU



Organización de un SoC ARM: TI Sitara335x. MPU

Organización de un SoC ARM: TI Sitara335x. MPU

- Cada fabricante (en el ejemplo TI), toma el core ARM, y lo completa. . . como mejor le parece.

Organización de un SoC ARM: TI Sitara335x. MPU

- Cada fabricante (en el ejemplo TI), toma el core ARM, y lo completa. . . como mejor le parece.
- Para cualquier Cortex-A, como vemos, queda a criterio del fabricante del SoC hasta el controlador de interrupciones.

Organización de un SoC ARM: TI Sitara335x. MPU

- Cada fabricante (en el ejemplo TI), toma el core ARM, y lo completa. . . como mejor le parece.
- Para cualquier Cortex-A, como vemos, queda a criterio del fabricante del SoC hasta el controlador de interrupciones.
- Esto quiere decir que cada SoC ARM basado en un Cortex-A, tendrá un sistema de Interrupciones de hardware diferente, y por lo tanto un driver diferente para su Controlador de interrupciones.

Organización de un SoC ARM: TI Sitara335x. MPU

- Cada fabricante (en el ejemplo TI), toma el core ARM, y lo completa. . . como mejor le parece.
- Para cualquier Cortex-A, como vemos, queda a criterio del fabricante del SoC hasta el controlador de interrupciones.
- Esto quiere decir que cada SoC ARM basado en un Cortex-A, tendrá un sistema de Interrupciones de hardware diferente, y por lo tanto un driver diferente para su Controlador de interrupciones.
- Imaginemos el resto del SoC. . . .

Organización de un SoC ARM: TI Sitara335x. MPU

- Cada fabricante (en el ejemplo TI), toma el core ARM, y lo completa. . . como mejor le parece.
- Para cualquier Cortex-A, como vemos, queda a criterio del fabricante del SoC hasta el controlador de interrupciones.
- Esto quiere decir que cada SoC ARM basado en un Cortex-A, tendrá un sistema de Interrupciones de hardware diferente, y por lo tanto un driver diferente para su Controlador de interrupciones.
- Imaginemos el resto del SoC. . . .
- El reset vector (programa de arranque) es también propietario de cada fabricante. En el ejemplo del slide anterior, para el TI Sitara 3358 se tiene una ROM de 176K y una pequeña RAM de 64K.

Organización de un SoC ARM: TI Sitara335x. MPU

- Cada fabricante (en el ejemplo TI), toma el core ARM, y lo completa. . . como mejor le parece.
- Para cualquier Cortex-A, como vemos, queda a criterio del fabricante del SoC hasta el controlador de interrupciones.
- Esto quiere decir que cada SoC ARM basado en un Cortex-A, tendrá un sistema de Interrupciones de hardware diferente, y por lo tanto un driver diferente para su Controlador de interrupciones.
- Imaginemos el resto del SoC. . . .
- El reset vector (programa de arranque) es también propietario de cada fabricante. En el ejemplo del slide anterior, para el TI Sitara 3358 se tiene una ROM de 176K y una pequeña RAM de 64K.
- En la ROM está el programa de arranque del sistema al que TI denomina Secure Boot. Ese programa se graba en fábrica en esa ROM, y está encriptado mediante algoritmos criptográficos basados en claves pública y privada.

Arranque de un SoC ARM

Arranque de un SoC ARM

- En general una vez que el firmware básico termina de inicializar los mínimos recursos de un board, lo que suele hacerse el cargar a un bootloader. U-Boot es uno de los mas difundidos

Arranque de un SoC ARM

- En general una vez que el firmware básico termina de inicializar los mínimos recursos de un board, lo que suele hacerse el cargar a un bootloader. U-Boot es uno de los mas difundidos
- Un bootoader en general debe proveer las siguientes funciones:

Arranque de un SoC ARM

- En general una vez que el firmware básico termina de inicializar los mínimos recursos de un board, lo que suele hacerse el cargar a un bootloader. U-Boot es uno de los mas difundidos
- Un bootoader en general debe proveer las siguientes funciones:

✓ Encontrar e inicializar la totalidad de memoria RAM que el kernel va a utilizar como almacenamiento volátil. Es 100 % dependiente de la plataforma. Puede tener la información de tamaño de la RAM preconfigurada y accederla o la puede determinar mediante un algoritmo.

Arranque de un SoC ARM

- En general una vez que el firmware básico termina de inicializar los mínimos recursos de un board, lo que suele hacerse el cargar a un bootloader. U-Boot es uno de los mas difundidos
- Un bootoader en general debe proveer las siguientes funciones:

✓ Encontrar e inicializar la totalidad de memoria RAM que el kernel va a utilizar como almacenamiento volátil. Es 100 % dependiente de la plataforma. Puede tener la información de tamaño de la RAM preconfigurada y accederla o la puede determinar mediante un algoritmo.

✓ Luego debe inicializar un puerto serie RS-232. El driver de puerto serie del kernel necesita establecer por allí una consola de Administración, que también tiene propósitos de terminal de log. Una alternativa es pasarle al kernel `console=<serial device>`

Arranque de un SoC ARM

- En general una vez que el firmware básico termina de inicializar los mínimos recursos de un board, lo que suele hacerse el cargar a un bootloader. U-Boot es uno de los mas difundidos
- Un bootoader en general debe proveer las siguientes funciones:

✓ Encontrar e inicializar la totalidad de memoria RAM que el kernel va a utilizar como almacenamiento volátil. Es 100 % dependiente de la plataforma. Puede tener la información de tamaño de la RAM preconfigurada y accederla o la puede determinar mediante un algoritmo.

✓ Luego debe inicializar un puerto serie RS-232. El driver de puerto serie del kernel necesita establecer por allí una consola de Administración, que también tiene propósitos de terminal de log. Una alternativa es pasarle al kernel `console=<serial device>`

- Lo que sigue depende de tener implementado o no el Device Tree.

Arranque de un SoC ARM sin usar Device Tree

Arranque de un SoC ARM sin usar Device Tree

- El bootloader carga en memoria y ejecuta la imagen del kernel que se encuentra en `/boot` . En el `/` hay un link denominado `/uImage` , `/zImage` , o `/vmlinuz` .

Arranque de un SoC ARM sin usar Device Tree

- El bootloader carga en memoria y ejecuta la imagen del kernel que se encuentra en `/boot` . En el `/` hay un link denominado `/uImage` , `/zImage` , o `/vmlinuz` .
- Si no hacemos nada, esa imagen del kernel debe contener la descripción completa del hardware.

Arranque de un SoC ARM sin usar Device Tree

- El bootloader carga en memoria y ejecuta la imagen del kernel que se encuentra en `/boot` . En el `/` hay un link denominado `/uImage` , `/zImage` , o `/vmlinuz` .
- Si no hacemos nada, esa imagen del kernel debe contener la descripción completa del hardware.
- El bootloader antes de ejecutar la imagen del kernel arma referencias a la descripción de hardware que debe estar dentro de la imagen del kernel

Arranque de un SoC ARM sin usar Device Tree

- El bootloader carga en memoria y ejecuta la imagen del kernel que se encuentra en `/boot`. En el `/` hay un link denominado `/uImage`, `/zImage`, o `/vmlinuz`.
- Si no hacemos nada, esa imagen del kernel debe contener la descripción completa del hardware.
- El bootloader antes de ejecutar la imagen del kernel arma referencias a la descripción de hardware que debe estar dentro de la imagen del kernel
- Le indica al kernel en que plataforma está booteando a través del registro `r1`.

Arranque de un SoC ARM sin usar Device Tree

- El bootloader carga en memoria y ejecuta la imagen del kernel que se encuentra en `/boot` . En el `/` hay un link denominado `/uImage` , `/zImage` , o `/vmlinuz` .
- Si no hacemos nada, esa imagen del kernel debe contener la descripción completa del hardware.
- El bootloader antes de ejecutar la imagen del kernel arma referencias a la descripción de hardware que debe estar dentro de la imagen del kernel
- Le indica al kernel en que plataforma está booteando a través del registro `r1` .
- El valor de `r1` sale de `/arch/arm/tools/mach-types`

Arranque de un SoC ARM sin usar Device Tree

- El bootloader carga en memoria y ejecuta la imagen del kernel que se encuentra en `/boot` . En el `/` hay un link denominado `/uImage` , `/zImage` , o `/vmlinuz` .
- Si no hacemos nada, esa imagen del kernel debe contener la descripción completa del hardware.
- El bootloader antes de ejecutar la imagen del kernel arma referencias a la descripción de hardware que debe estar dentro de la imagen del kernel
- Le indica al kernel en que plataforma está booteando a través del registro `r1` .
- El valor de `r1` sale de `/arch/arm/tools/mach-types`
- Para entender la dimensión del problema puede consultarse el listado de IDs ARM Machines en *http://www.arm.linux.org.uk/developer/machines/*

Arranque de un SoC ARM sin usar Device Tree

- El bootloader carga en memoria y ejecuta la imagen del kernel que se encuentra en `/boot` . En el `/` hay un link denominado `/uImage` , `/zImage` , o `/vmlinuz` .
- Si no hacemos nada, esa imagen del kernel debe contener la descripción completa del hardware.
- El bootloader antes de ejecutar la imagen del kernel arma referencias a la descripción de hardware que debe estar dentro de la imagen del kernel
- Le indica al kernel en que plataforma está booteando a través del registro `r1` .
- El valor de `r1` sale de `/arch/arm/tools/mach-types`
- Para entender la dimensión del problema puede consultarse el listado de IDs ARM Machines en *http://www.arm.linux.org.uk/developer/machines/*
- Carga información adicional denominada ATAGs. .

Arranque de un SoC ARM sin usar Device Tree

Arranque de un SoC ARM sin usar Device Tree

- El bootloader debe crear e inicializar la lista de etiquetas del kernel.

Arranque de un SoC ARM sin usar Device Tree

- El bootloader debe crear e inicializar la lista de etiquetas del kernel.
- Una lista etiquetada válida comienza con `ATAG_CORE` y termina con `ATAG_NONE`.

Arranque de un SoC ARM sin usar Device Tree

- El bootloader debe crear e inicializar la lista de etiquetas del kernel.
- Una lista etiquetada válida comienza con `ATAG_CORE` y termina con `ATAG_NONE`.
- La etiqueta `ATAG_CORE` puede estar vacía o no. Una etiqueta `ATAG_CORE` vacía tiene el campo de tamaño establecido en '2' (0x00000002).

Arranque de un SoC ARM sin usar Device Tree

- El bootloader debe crear e inicializar la lista de etiquetas del kernel.
- Una lista etiquetada válida comienza con `ATAG_CORE` y termina con `ATAG_NONE`.
- La etiqueta `ATAG_CORE` puede estar vacía o no. Una etiqueta `ATAG_CORE` vacía tiene el campo de tamaño establecido en '2' (0x00000002).
- `ATAG_NONE` debe establecerse el campo de tamaño a cero.

Arranque de un SoC ARM sin usar Device Tree

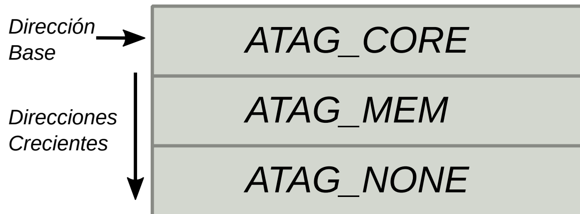
- El bootloader debe crear e inicializar la lista de etiquetas del kernel.
- Una lista etiquetada válida comienza con **ATAG_CORE** y termina con **ATAG_NONE** .
- La etiqueta **ATAG_CORE** puede estar vacía o no. Una etiqueta **ATAG_CORE** vacía tiene el campo de tamaño establecido en '2' (0x00000002).
- **ATAG_NONE** debe establecerse el campo de tamaño a cero.
- Se puede colocar cualquier número de etiquetas en la lista.

Arranque de un SoC ARM sin usar Device Tree

- El bootloader debe crear e inicializar la lista de etiquetas del kernel.
- Una lista etiquetada válida comienza con **ATAG_CORE** y termina con **ATAG_NONE** .
- La etiqueta **ATAG_CORE** puede estar vacía o no. Una etiqueta **ATAG_CORE** vacía tiene el campo de tamaño establecido en '2' (0x00000002).
- **ATAG_NONE** debe establecerse el campo de tamaño a cero.
- Se puede colocar cualquier número de etiquetas en la lista.
- El bootloader debe pasar como mínimo tamaño y ubicación de la memoria del sistema y ubicación del root file system.

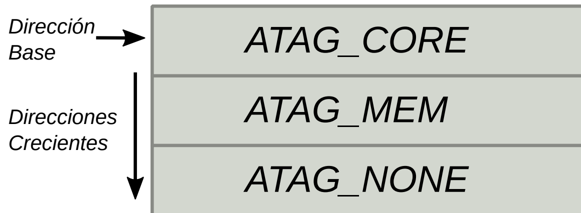
Arranque de un SoC ARM sin usar Device Tree

- El bootloader debe crear e inicializar la lista de etiquetas del kernel.
- Una lista etiquetada válida comienza con **ATAG_CORE** y termina con **ATAG_NONE** .
- La etiqueta **ATAG_CORE** puede estar vacía o no. Una etiqueta **ATAG_CORE** vacía tiene el campo de tamaño establecido en '2' (0x00000002).
- **ATAG_NONE** debe establecerse el campo de tamaño a cero.
- Se puede colocar cualquier número de etiquetas en la lista.
- El bootloader debe pasar como mínimo tamaño y ubicación de la memoria del sistema y ubicación del root file system.
- **ATAGs** Luce de este modo→:

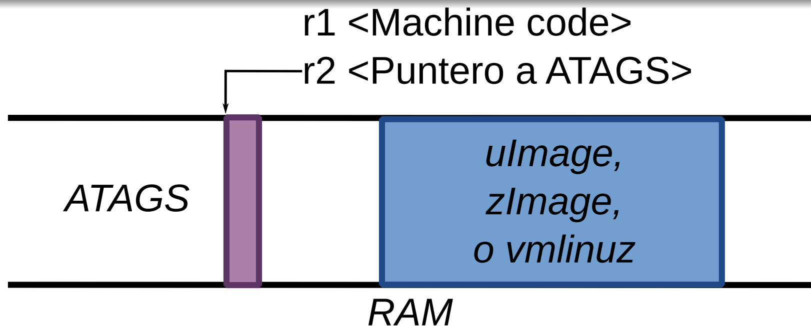


Arranque de un SoC ARM sin usar Device Tree

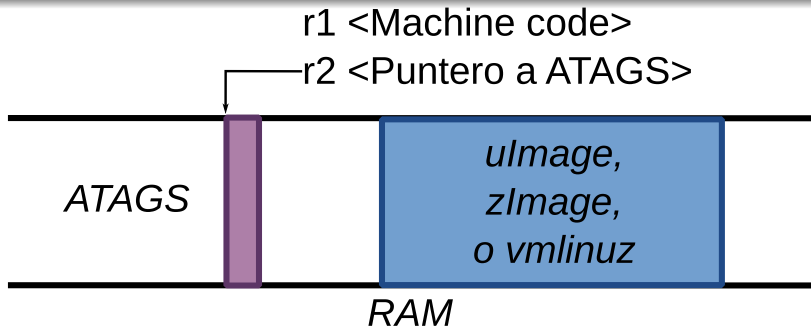
- El bootloader debe crear e inicializar la lista de etiquetas del kernel.
- Una lista etiquetada válida comienza con **ATAG_CORE** y termina con **ATAG_NONE** .
- La etiqueta **ATAG_CORE** puede estar vacía o no. Una etiqueta **ATAG_CORE** vacía tiene el campo de tamaño establecido en '2' (0x00000002).
- **ATAG_NONE** debe establecerse el campo de tamaño a cero.
- Se puede colocar cualquier número de etiquetas en la lista.
- El bootloader debe pasar como mínimo tamaño y ubicación de la memoria del sistema y ubicación del root file system.
- **ATAGs** Luce de este modo→:
- La dirección de inicio de **ATAG** se pasa por el registro **r2** .



Arranque de un SoC ARM sin usar Device Tree

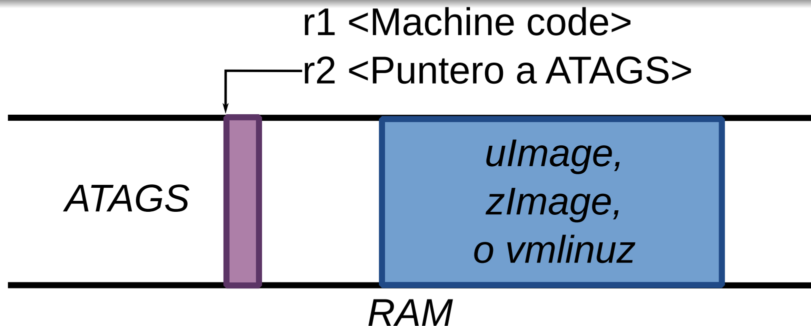


Arranque de un SoC ARM sin usar Device Tree



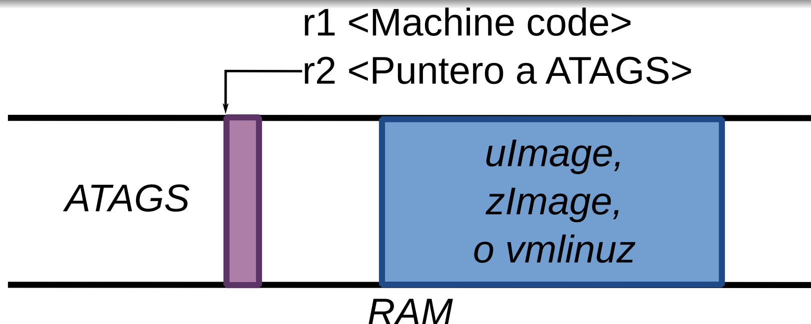
- La lista de etiquetas debe estar en memoria DRAM al momento del arranque del kernel

Arranque de un SoC ARM sin usar Device Tree



- La lista de etiquetas debe estar en memoria DRAM al momento del arranque del kernel
- Debe ser una zona de direcciones que no pueda ser escrita por el descompresor del kernel, ni por el init RAM Disk (`initd`).

Arranque de un SoC ARM sin usar Device Tree



- La lista de etiquetas debe estar en memoria DRAM al momento del arranque del kernel
- Debe ser una zona de direcciones que no pueda ser escrita por el descompresor del kernel, ni por el init RAM Disk (`initd`).
- Normalmente se ubica en los primeros 16 Kbytes del mapa de RAM para evitar este problema.

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Arranque con Device Tree

Arranque con Device Tree

- El kernel no contiene la descripción de hardware.

Arranque con Device Tree

- El kernel no contiene la descripción de hardware.
- La descripción del hardware se encuentra en un archivo binario denominado *Device Tree Blob*.

Arranque con Device Tree

- El kernel no contiene la descripción de hardware.
- La descripción del hardware se encuentra en un archivo binario denominado *Device Tree Blob*.
- El bootloader carga en memoria dos archivos binarios, uno con la imagen del kernel y otro con el Device Tree.

Arranque con Device Tree

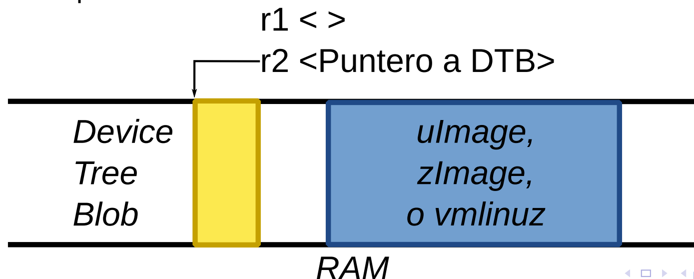
- El kernel no contiene la descripción de hardware.
- La descripción del hardware se encuentra en un archivo binario denominado *Device Tree Blob*.
- El bootloader carga en memoria dos archivos binarios, uno con la imagen del kernel y otro con el Device Tree.
- Luego inicializa `r2` con la dirección de inicio del Device Tree.

Arranque con Device Tree

- El kernel no contiene la descripción de hardware.
- La descripción del hardware se encuentra en un archivo binario denominado *Device Tree Blob*.
- El bootloader carga en memoria dos archivos binarios, uno con la imagen del kernel y otro con el Device Tree.
- Luego inicializa `r2` con la dirección de inicio del Device Tree.
- No existe el concepto de machine number. No se usa `r1` .

Arranque con Device Tree

- El kernel no contiene la descripción de hardware.
- La descripción del hardware se encuentra en un archivo binario denominado *Device Tree Blob*.
- El bootloader carga en memoria dos archivos binarios, uno con la imagen del kernel y otro con el Device Tree.
- Luego inicializa `r2` con la dirección de inicio del Device Tree.
- No existe el concepto de machine number. No se usa `r1`.



Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - **Funcionamiento del Device Tree**
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Introducción a Device Tree

Introducción a Device Tree

- El Device Tree es una estructura de datos descriptiva del hardware.

Introducción a Device Tree

- El Device Tree es una estructura de datos descriptiva del hardware.
- Dicha descripción en vez de estar incluida en los fuentes del Sistema Operativo, está en un archivo binario, que será accedido por el sistema operativo al inicializar el sistema.

Introducción a Device Tree

- El Device Tree es una estructura de datos descriptiva del hardware.
- Dicha descripción en vez de estar incluida en los fuentes del Sistema Operativo, está en un archivo binario, que será accedido por el sistema operativo al inicializar el sistema.
- El Sistema Operativo dispone de un conjunto de funciones que interpretan el contenido del Device Tree y generan las configuraciones necesarias en el sistema.

Introducción a Device Tree

- El Device Tree es una estructura de datos descriptiva del hardware.
- Dicha descripción en vez de estar incluida en los fuentes del Sistema Operativo, está en un archivo binario, que será accedido por el sistema operativo al inicializar el sistema.
- El Sistema Operativo dispone de un conjunto de funciones que interpretan el contenido del Device Tree y generan las configuraciones necesarias en el sistema.
- De este modo, el código fuente del Sistema Operativo es el mismo para cualquier modelo de sistema basado en ARM.

Introducción a Device Tree

- El Device Tree es una estructura de datos descriptiva del hardware.
- Dicha descripción en vez de estar incluida en los fuentes del Sistema Operativo, está en un archivo binario, que será accedido por el sistema operativo al inicializar el sistema.
- El Sistema Operativo dispone de un conjunto de funciones que interpretan el contenido del Device Tree y generan las configuraciones necesarias en el sistema.
- De este modo, el código fuente del Sistema Operativo es el mismo para cualquier modelo de sistema basado en ARM.
- Lo mismo ocurre con otras arquitecturas.

Introducción a Device Tree

- El Device Tree es una estructura de datos descriptiva del hardware.
- Dicha descripción en vez de estar incluida en los fuentes del Sistema Operativo, está en un archivo binario, que será accedido por el sistema operativo al inicializar el sistema.
- El Sistema Operativo dispone de un conjunto de funciones que interpretan el contenido del Device Tree y generan las configuraciones necesarias en el sistema.
- De este modo, el código fuente del Sistema Operativo es el mismo para cualquier modelo de sistema basado en ARM.
- Lo mismo ocurre con otras arquitecturas.
- Incluido Intel que (si bien es una arquitectura estándar), también pudo desarrollar versiones embedded basadas en procesadores x86, pero no compatibles con el mapa de direcciones de una PC, como Quarqz, Edison, y Joule. Si bien estos SoCs fueron oficialmente discontinuados vale la pena aclararlo de todos modos.

Device Tree, extensiones y tipos de archivos

Device Tree, extensiones y tipos de archivos

`.dts` : Device Tree Source File.

- Son parte del source de Linux. Están en: `arch/arm/boot/dts` .

Device Tree, extensiones y tipos de archivos

`.dts` : Device Tree Source File.

- Son parte del source de Linux. Están en: `arch/arm/boot/dts` .
- Existe un `dts` diferente por cada SoC y plataforma disponible.

Device Tree, extensiones y tipos de archivos

`.dts` : Device Tree Source File.

- Son parte del source de Linux. Están en: `arch/arm/boot/dts` .
- Existe un `dts` diferente por cada SoC y plataforma disponible.
- Los `dts` son compilados cuando se genera la imagen de kernel.

Device Tree, extensiones y tipos de archivos

`.dts` : Device Tree Source File.

- Son parte del source de Linux. Están en: `arch/arm/boot/dts` .
- Existe un `dts` diferente por cada SoC y plataforma disponible.
- Los `dts` son compilados cuando se genera la imagen de kernel.

`.dtb` : Device Tree Blob file.

- Resultado de la compilación de los archivos `.dts` .

Device Tree, extensiones y tipos de archivos

`.dts` : Device Tree Source File.

- Son parte del source de Linux. Están en: `arch/arm/boot/dts` .
- Existe un `dts` diferente por cada SoC y plataforma disponible.
- Los `dts` son compilados cuando se genera la imagen de kernel.

`.dtb` : Device Tree Blob file.

- Resultado de la compilación de los archivos `.dts` .
- Estos archivos están en el directorio `/boot` del SO.

Device Tree, extensiones y tipos de archivos

`.dts` : Device Tree Source File.

- Son parte del source de Linux. Están en: `arch/arm/boot/dts` .
- Existe un `dts` diferente por cada SoC y plataforma disponible.
- Los `dts` son compilados cuando se genera la imagen de kernel.

`.dtb` : Device Tree Blob file.

- Resultado de la compilación de los archivos `.dts` .
- Estos archivos están en el directorio `/boot` del SO.

`dtc` : Device Tree Compiler.

- Transforma el árbol especificado en el archivo `.dts` en un archivo binario `.dtb` que acompañará a la imagen del kernel

Device Tree, extensiones y tipos de archivos

`.dts` : Device Tree Source File.

- Son parte del source de Linux. Están en: `arch/arm/boot/dts` .
- Existe un `dts` diferente por cada SoC y plataforma disponible.
- Los `dts` son compilados cuando se genera la imagen de kernel.

`.dtb` : Device Tree Blob file.

- Resultado de la compilación de los archivos `.dts` .
- Estos archivos están en el directorio `/boot` del SO.

`dtc` : Device Tree Compiler.

- Transforma el árbol especificado en el archivo `.dts` en un archivo binario `.dtb` que acompañará a la imagen del kernel
- Se puede ver el device-tree en uso en `/proc/device-tree/` .

Introducción a Device Tree

Introducción a Device Tree

- Un device tree es una estructura de árbol compuesta por nodos, y para cada uno de estos propiedades

Introducción a Device Tree

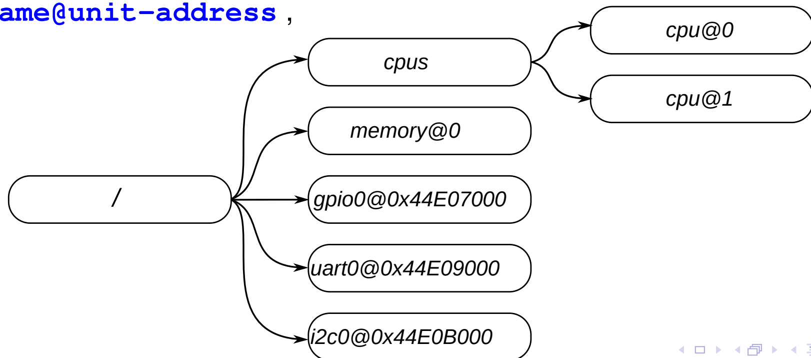
- Un device tree es una estructura de árbol compuesta por nodos, y para cada uno de estos propiedades
- Cada configuración tiene una estructura de nodos y propiedades que se van ramificando a medida que se describe el hardware.

Introducción a Device Tree

- Un device tree es una estructura de árbol compuesta por nodos, y para cada uno de estos propiedades
- Cada configuración tiene una estructura de nodos y propiedades que se van ramificando a medida que se describe el hardware.
- El inicio es `/` y a partir de allí se describen nodos con el formato `node-name@unit-address` ,

Introducción a Device Tree

- Un device tree es una estructura de árbol compuesta por nodos, y para cada uno de estos propiedades
- Cada configuración tiene una estructura de nodos y propiedades que se van ramificando a medida que se describe el hardware.
- El inicio es `/` y a partir de allí se describen nodos con el formato `node-name@unit-address` ,



Flattened Device Tree (FDT)

Flattened Device Tree (FDT)

- Es simplemente una estructura en la que se describe todo el hardware de la tarjeta.

Flattened Device Tree (FDT)

- Es simplemente una estructura en la que se describe todo el hardware de la tarjeta.
- Sintaxis genérica:

Flattened Device Tree (FDT)

- Es simplemente una estructura en la que se describe todo el hardware de la tarjeta.
- Sintaxis genérica:


```
/{
  nodo@0 {
```

- Inicia en / , y cada ítem relevante es un nodo.

Flattened Device Tree (FDT)

- Es simplemente una estructura en la que se describe todo el hardware de la tarjeta.
- Sintaxis genérica:

```
/{
  nodo@0 {
    propiedad string = "Cadena de texto";
    propiedad lista    = "string1", "string2";
    propiedad Datos byte = [0x24 0xA9 0x2B];
```


- Inicia en **/** , y cada ítem relevante es un nodo.
- Los nodos tienen propiedades a las que se le definen valores.

Flattened Device Tree (FDT)

- Es simplemente una estructura en la que se describe todo el hardware de la tarjeta.
- Sintaxis genérica:
- Inicia en / , y cada ítem relevante es un nodo.
- Los nodos tienen propiedades a las que se le definen valores.
- Puede haber nodos dentro de otros.

```
/{
  nodo@0 {
    propiedad string = "Cadena de texto";
    propiedad lista    = "string1", "string2";
    propiedad Datos byte = [0x24 0xA9 0x2B];

    nodo-child@0 {
        primer propiedad;
        segunda propiedad = <2>;
        referencia = <&node1>;
    };

    nodo-child@1 {
    };
};
```

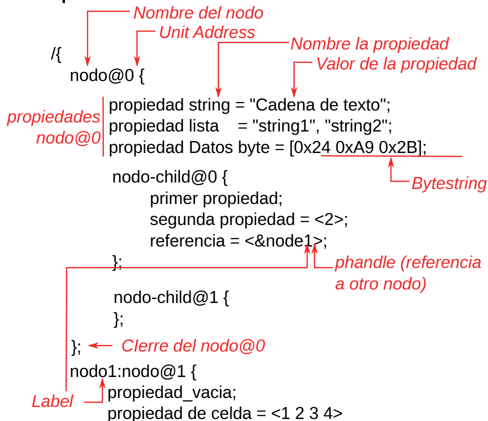
Flattened Device Tree (FDT)

- Es simplemente una estructura en la que se describe todo el hardware de la tarjeta.
- Sintaxis genérica:
- Inicia en / , y cada ítem relevante es un nodo.
- Los nodos tienen propiedades a las que se le definen valores.
- Puede haber nodos dentro de otros.
- La descripción de un nodo es entre llaves .

```
/{
  nodo@0 {
    propiedad string = "Cadena de texto";
    propiedad lista    = "string1", "string2";
    propiedad Datos byte = [0x24 0xA9 0x2B];
    nodo-child@0 {
        primer propiedad;
        segunda propiedad = <2>;
        referencia = <&node1>;
    };
    nodo-child@1 {
    };
  };
  nodo1:nodo@1 {
    propiedad_vacia;
    propiedad de celda = <1 2 3 4>
```

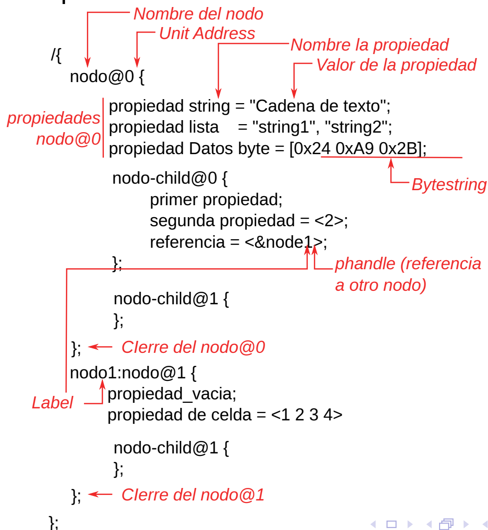
Flattened Device Tree (FDT)

- Es simplemente una estructura en la que se describe todo el hardware de la tarjeta.
- Sintaxis genérica:
- Inicia en / , y cada ítem relevante es un nodo.
- Los nodos tienen propiedades a las que se le definen valores.
- Puede haber nodos dentro de otros.
- La descripción de un nodo es entre llaves .
- Labels a ser referenciados desde cualquier punto.



Flattened Device Tree (FDT)

- Es simplemente una estructura en la que se describe todo el hardware de la tarjeta.
- Sintaxis genérica:
- Inicia en / , y cada ítem relevante es un nodo.
- Los nodos tienen propiedades a las que se le definen valores.
- Puede haber nodos dentro de otros.
- La descripción de un nodo es entre llaves .
- Labels a ser referenciados desde cualquier punto.



Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

/arch/arm/boot/dts/am335x-bone.dts

```
1 /* Copyright (C) 2012 Texas Instruments Incorporated - http://www.ti.com/
2  *
3  * This program is free software; you can redistribute it and/or modify
4  * it under the terms of the GNU General Public License version 2 as
5  * published by the Free Software Foundation.
6  */
7 /dts-v1/;
8 #include "am33xx.dtsi"
9 #include "am335x-bone-common.dtsi"
10
11 / {
12     model = "TI AM335x BeagleBone";
13     compatible = "ti,am335x-bone", "ti,am33xx";
14   };
15
16   &ldo3_reg {
17     regulator-min-microvolt = <1800000>;
18     regulator-max-microvolt = <3300000>;
19     regulator-always-on;
20   };
21
22   &mmc1 {
23     vmmc-supply = <&ldo3_reg>;
24   };
```

dentro de “am335x-bone-common.dtsi”

```
1  &am33xx_pinmux {
2      pinctrl-names = "default";
3      pinctrl-0 = <&clkout2_pin>;
4
5      user_leds_s0: user_leds_s0 {
6          pinctrl-single,pins = <
7              0x54 (PIN_OUTPUT_PULLDOWN | MUX_MODE7) /* gpmc_a5.gpio1_21 */
8              0x58 (PIN_OUTPUT_PULLUP | MUX_MODE7)   /* gpmc_a6.gpio1_22 */
9              0x5c (PIN_OUTPUT_PULLDOWN | MUX_MODE7) /* gpmc_a7.gpio1_23 */
10             0x60 (PIN_OUTPUT_PULLUP | MUX_MODE7)   /* gpmc_a8.gpio1_24 */
11         >;
12     };
13     i2c0_pins: pinmux_i2c0_pins {
14         pinctrl-single,pins = <
15             0x188 (PIN_INPUT_PULLUP | MUX_MODE0)   /* i2c0_sda.i2c0_sda */
16             0x18c (PIN_INPUT_PULLUP | MUX_MODE0)   /* i2c0_scl.i2c0_scl */
17         >;
18     };
19     i2c2_pins: pinmux_i2c2_pins {
20         pinctrl-single,pins = <
21             0x178 (PIN_INPUT_PULLUP | MUX_MODE3)   /* uart1_ctsn.i2c2_sda */
22             0x17c (PIN_INPUT_PULLUP | MUX_MODE3)   /* uart1_rtsn.i2c2_scl */
23         >;
24     };
```

Compilación

Compilación

- Estos archivos se compilan con el `dtc` , (*device tree compiler*) y se instalan dentro del dispositivo de almacenamiento de la Beagle Bone Black en el path `/boot/uboot/dtbs` .

Compilación

- Estos archivos se compilan con el `dtc` , (*device tree compiler*) y se instalan dentro del dispositivo de almacenamiento de la Beagle Bone Black en el path `/boot/uboot/dtbs` .
- Si se quiere modificar o agregar hardware hay que modificar los archivos fuentes y recompilarlos.

Compilación

- Estos archivos se compilan con el `dtc` , (*device tree compiler*) y se instalan dentro del dispositivo de almacenamiento de la Beagle Bone Black en el path `/boot/uboot/dtbs` .
- Si se quiere modificar o agregar hardware hay que modificar los archivos fuentes y recompilarlos.

```
root@beaglebone:/boot/uboot/dtbs# ls
am335x-bone.dtb        am335x-tester.dtb     omap3-evm.dtb          omap4-panda.dtb
am335x-boneblack.dtb   omap2420-h4.dtb       omap3-tobi.dtb         omap4-sdp.dtb
am335x-evm.dtb         omap3-beagle-xm.dtb   omap4-panda-a4.dtb     omap4-var-som.dtb
am335x-evmsk.dtb       omap3-beagle.dtb      omap4-panda-es.dtb     omap5-evm.dtb
root@beaglebone:/boot/uboot/dtbs#
```

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Motivación del Device Tree Overlay (DTO)

Motivación del Device Tree Overlay (DTO)

- La BeagleBone Black al igual que otras tarjetas de su tipo tienen una variedad de productos de terceras partes conectables a sus terminales para extender funcionalidades,

Motivación del Device Tree Overlay (DTO)

- La BeagleBone Black al igual que otras tarjetas de su tipo tienen una variedad de productos de terceras partes conectables a sus terminales para extender funcionalidades,
- Estos dispositivos se denominan *capes*.

Motivación del Device Tree Overlay (DTO)

- La BeagleBone Black al igual que otras tarjetas de su tipo tienen una variedad de productos de terceras partes conectables a sus terminales para extender funcionalidades,
- Estos dispositivos se denominan *capes*.
- En ocasiones los capes requieren multiplexar las diferentes funcionalidades de un puerto para cambiarla runtime, es decir, con Linux ejecutando en nuestro sistema.

Motivación del Device Tree Overlay (DTO)

- La BeagleBone Black al igual que otras tarjetas de su tipo tienen una variedad de productos de terceras partes conectables a sus terminales para extender funcionalidades,
- Estos dispositivos se denominan *capes*.
- En ocasiones los capes requieren multiplexar las diferentes funcionalidades de un puerto para cambiarla runtime, es decir, con Linux ejecutando en nuestro sistema.
- El Device Tree no soporta reconfiguración runtime. Linux arranca con un DT y opera en función de lo que encontró en él durante su inicialización.

Motivación del Device Tree Overlay (DTO)

- La BeagleBone Black al igual que otras tarjetas de su tipo tienen una variedad de productos de terceras partes conectables a sus terminales para extender funcionalidades,
- Estos dispositivos se denominan *capes*.
- En ocasiones los capes requieren multiplexar las diferentes funcionalidades de un puerto para cambiarla runtime, es decir, con Linux ejecutando en nuestro sistema.
- El Device Tree no soporta reconfiguración runtime. Linux arranca con un DT y opera en función de lo que encontró en él durante su inicialización.
- Cuando se utilizan capes necesitamos modificar el comportamiento de los periféricos desde el SO.

Motivación del Device Tree Overlay (DTO)

- La BeagleBone Black al igual que otras tarjetas de su tipo tienen una variedad de productos de terceras partes conectables a sus terminales para extender funcionalidades,
- Estos dispositivos se denominan *capes*.
- En ocasiones los capes requieren multiplexar las diferentes funcionalidades de un puerto para cambiarla runtime, es decir, con Linux ejecutando en nuestro sistema.
- El Device Tree no soporta reconfiguración runtime. Linux arranca con un DT y opera en función de lo que encontró en él durante su inicialización.
- Cuando se utilizan capes necesitamos modificar el comportamiento de los periféricos desde el SO.
- Para que las modificaciones se pueden hacer en tiempo de ejecución en modo usuario, se introduce el concepto de Device Tree Overlays (DTO)

Cape Manager

Cape Manager

- ***Capemgr*** fue diseñado con dos objetivos principales:

Cape Manager

- ***Capemgr*** fue diseñado con dos objetivos principales:
- Un board (la Beagle Bone Black por ejemplo) puede detectar al momento de bootear la presencia de hardware de expansión, cargar sus drivers, asignar recursos, como espacios de direccionamiento, líneas de requerimiento de interrupción, por ejemplo.

Cape Manager

- ***Capemgr*** fue diseñado con dos objetivos principales:
- Un board (la Beagle Bone Black por ejemplo) puede detectar al momento de bootear la presencia de hardware de expansión, cargar sus drivers, asignar recursos, como espacios de direccionamiento, líneas de requerimiento de interrupción, por ejemplo.
- Lógicamente esto requiere que el kernel soporte Device Tree.

Cape Manager

- ***Capemgr*** fue diseñado con dos objetivos principales:
- Un board (la Beagle Bone Black por ejemplo) puede detectar al momento de bootear la presencia de hardware de expansión, cargar sus drivers, asignar recursos, como espacios de direccionamiento, líneas de requerimiento de interrupción, por ejemplo.
- Lógicamente esto requiere que el kernel soporte Device Tree.
- Como ya se dijo, el Device Tree es una estructura estática, que no puede reconfigurarse ante la detección de un nuevo hardware

Cape Manager

- ***Capemgr*** fue diseñado con dos objetivos principales:
- Un board (la Beagle Bone Black por ejemplo) puede detectar al momento de bootear la presencia de hardware de expansión, cargar sus drivers, asignar recursos, como espacios de direccionamiento, líneas de requerimiento de interrupción, por ejemplo.
- Lógicamente esto requiere que el kernel soporte Device Tree.
- Como ya se dijo, el Device Tree es una estructura estática, que no puede reconfigurarse ante la detección de un nuevo hardware
- Las BeagleBone en general permiten usar ***capes*** fabricados por terceros y arrancarlos sin necesidad de recompilar el kernel.

Cape Manager

- ***Capemgr*** fue diseñado con dos objetivos principales:
- Un board (la Beagle Bone Black por ejemplo) puede detectar al momento de bootear la presencia de hardware de expansión, cargar sus drivers, asignar recursos, como espacios de direccionamiento, líneas de requerimiento de interrupción, por ejemplo.
- Lógicamente esto requiere que el kernel soporte Device Tree.
- Como ya se dijo, el Device Tree es una estructura estática, que no puede reconfigurarse ante la detección de un nuevo hardware
- Las BeagleBone en general permiten usar ***capes*** fabricados por terceros y arrancarlos sin necesidad de recompilar el kernel.
- La habilidad del ***Capemgr*** consiste en cargar y agregar en tiempo de ejecución durante el arranque del sistema, un archivo fragmento del Device Tree en forma de overlay.

Cape Manager

- ***Capemgr*** fue diseñado con dos objetivos principales:
- Un board (la Beagle Bone Black por ejemplo) puede detectar al momento de bootear la presencia de hardware de expansión, cargar sus drivers, asignar recursos, como espacios de direccionamiento, líneas de requerimiento de interrupción, por ejemplo.
- Lógicamente esto requiere que el kernel soporte Device Tree.
- Como ya se dijo, el Device Tree es una estructura estática, que no puede reconfigurarse ante la detección de un nuevo hardware
- Las BeagleBone en general permiten usar ***capes*** fabricados por terceros y arrancarlos sin necesidad de recompilar el kernel.
- La habilidad del ***Capemgr*** consiste en cargar y agregar en tiempo de ejecución durante el arranque del sistema, un archivo fragmento del Device Tree en forma de overlay.
- Para lograrlo, cada Cape que utilicemos en nuestra plataforma, debe tener un Device Tree Overlay que la describa.

Device Tree Overlay en BeagleBone Black

Device Tree Overlay en BeagleBone Black

- En la BBB el Cape Manager se encarga de cargar el Device Tree de cada Cape.

Device Tree Overlay en BeagleBone Black

- En la BBB el Cape Manager se encarga de cargar el Device Tree de cada Cape.
- Cada Cape provisto por el fabricante tiene una memoria EEPROM que contiene la información necesaria para que el Cape Manager cargue el fragmento de Device Tree correspondiente.

Device Tree Overlay en BeagleBone Black

- En la BBB el Cape Manager se encarga de cargar el Device Tree de cada Cape.
- Cada Cape provisto por el fabricante tiene una memoria EEPROM que contiene la información necesaria para que el Cape Manager cargue el fragmento de Device Tree correspondiente.
- Esta memoria EEPROM está conectada al Bus I2C #2 de la BBB.

Device Tree Overlay en BeagleBone Black

- En la BBB el Cape Manager se encarga de cargar el Device Tree de cada Cape.
- Cada Cape provisto por el fabricante tiene una memoria EEPROM que contiene la información necesaria para que el Cape Manager cargue el fragmento de Device Tree correspondiente.
- Esta memoria EEPROM está conectada al Bus I2C #2 de la BBB.
- El formato de datos en la EEPROM se describe en System Reference Manual (SRM), Capítulo 8 Cape Board Support. A la fecha en *https://raw.githubusercontent.com/CircuitCo/BeagleBone-Black/master/BBB_SRM.pdf*

Device Tree Overlay en BeagleBone Black

- En la BBB el Cape Manager se encarga de cargar el Device Tree de cada Cape.
- Cada Cape provisto por el fabricante tiene una memoria EEPROM que contiene la información necesaria para que el Cape Manager cargue el fragmento de Device Tree correspondiente.
- Esta memoria EEPROM está conectada al Bus I2C #2 de la BBB.
- El formato de datos en la EEPROM se describe en System Reference Manual (SRM), Capítulo 8 Cape Board Support. A la fecha en *https://raw.githubusercontent.com/CircuitCo/BeagleBone-Black/master/BBB_SRM.pdf*
- El campo Board Name de la EEPROM utilizado por ***Capemgr*** para determinar que fragmento de DTO debe cargar.

Device Tree Overlay en BeagleBone Black

- Hay dos manera de utilizar el campo Board Name:

Device Tree Overlay en BeagleBone Black

- Hay dos manera de utilizar el campo Board Name:

✓ Mapear el Board Name en el Source del DT Principal de la BeagleBone (`arch/arm/boot/dts/am335x-bone-common.dtsi`)

Device Tree Overlay en BeagleBone Black

- Hay dos manera de utilizar el campo Board Name:

✓ Mapear el Board Name en el Source del DT Principal de la BeagleBone (`arch/arm/boot/dts/am335x-bone-common.dtsi`)

✓ Usar un archivo overlay con un formato como este:

Device Tree Overlay en BeagleBone Black

- Hay dos manera de utilizar el campo Board Name:

✓ Mapear el Board Name en el Source del DT Principal de la BeagleBone (**arch/arm/boot/dts/am335x-bone-common.dtsi**)

✓ Usar un archivo overlay con un formato como este:

```
1          /* mrf24j40 cape */
2          cape@10 {
3                  part-number = "BB-BONE-MRF24J40";
4                  version@00A0 {
5                          version = "00A0";
6                          dtbo = "cape-bone-mrf24j40-00A0.dtbo";
7                  };
8          };
```

Device Tree Overlay en BeagleBone Black

- Hay dos manera de utilizar el campo Board Name:
- ✓ Mapear el Board Name en el Source del DT Principal de la BeagleBone (**`arch/arm/boot/dts/am335x-bone-common.dtsi`**)
- ✓ Usar un archivo overlay con un formato como este:

```
1          /* mrf24j40 cape */
2          cape@10 {
3                  part-number = "BB-BONE-MRF24J40";
4                  version@00A0 {
5                          version = "00A0";
6                          dtbo = "cape-bone-mrf24j40-00A0.dtbo";
7                  };
8          };
```

`BB-BONE-MRF24J40` es el campo Device Name y **`00A0`** el campo **`Version`** de la tabla de la EEPROM.
El ***Capmngr*** procesará el archivo **`/lib/firmware/cape-bone-mrf24j40-00A0.dtbo`**.

Device Tree Overlay en BeagleBone Black

```
alejandro@beaglebone:/sys/devices/bone_capemgr.9$ ls /lib/firmware/
ADAFRUIT-SPI0-00A0.dtbo      BB-BONE-SERL-03-00A1.dtbo     atmel_at76c502e-wpa.bin          bone_pwm_P9_42-00A0.dtbo           cape-universaln-00A0.dtbo
ADAFRUIT-SPI1-00A0.dtbo      BB-BONE-eMMC1-01-00A0.dtbo    atmel_at76c502e.bin              brcm                               carl9170-1.fw
ADAFRUIT-UART1-00A0.dtbo     BB-BONELT-BT-00A0.dtbo        atmel_at76c503-i3861.bin         cape-CBB-Serial-r01.dtbo           htc_9271.fw
ADAFRUIT-UART2-00A0.dtbo     BB-GPIOHELP-00A0.dtbo         atmel_at76c503-i3863.bin         cape-bebopr-R2.dtbo                lbtf_usb.bin
ADAFRUIT-UART4-00A0.dtbo     BB-I2C1-00A0.dtbo             atmel_at76c503-rfmd-0.90.2-140.bin  cape-bebopr-brdg-R2.dtbo        libertas
ADAFRUIT-UART5-00A0.dtbo     BB-I2C1A1-00A0.dtbo           atmel_at76c503-rfmd-acc.bin      cape-bebopr-ena-R2.dtbo            libertas_cs.fw
BB-ADC-00A0.dtbo             BB-SPIDEV0-00A0.dtbo          atmel_at76c503-rfmd.bin          cape-bone-2g-emmc1.dtbo            libertas_cs_helper.fw
BB-BONE-AUDI-01-00A0.dtbo    BB-SPIDEV1-00A0.dtbo          atmel_at76c504.bin               cape-bone-adafruit-lcd-00A0.dtbo   mwl8k
BB-BONE-BACON-00A0.dtbo      BB-SPIDEV1A1-00A0.dtbo        atmel_at76c504_2958-wpa.bin      cape-bone-argus-00A0.dtbo          rt2561.bin
BB-BONE-BACONE-00A0.dtbo     BB-UART1-00A0.dtbo            atmel_at76c504a_2958-wpa.bin     cape-bone-dvi-00A0.dtbo            rt2561s.bin
BB-BONE-BACONE2-00A0.dtbo    BB-UART2-00A0.dtbo            atmel_at76c504c-wpa.bin          cape-bone-dvi-00A1.dtbo            rt2661.bin
BB-BONE-CAM-VVDN-00A0.dtbo   BB-UART2-RTSCTS-00A0.dtbo     atmel_at76c505-rfmd.bin          cape-bone-dvi-00A2.dtbo            rt2860.bin
BB-BONE-CAM3-01-00A2.dtbo    BB-UART4-00A0.dtbo            atmel_at76c505-rfmd2958.bin      cape-bone-exptest-00A0.dtbo        rt2870.bin
BB-BONE-CRYPTO-00A0.dtbo     BB-UART4-RTSCTS-00A0.dtbo     atmel_at76c505a-rfmd2958.bin     cape-bone-geiger-00A0.dtbo         rt3070.bin
BB-BONE-GPEVT-00A0.dtbo      BB-UART5-00A0.dtbo            atmel_at76c506-wpa.bin           cape-bone-hexy-00A0.dtbo           rt3071.bin
BB-BONE-GPS-00A0.dtbo        CBB-Relay-00A0.dtbo           atmel_at76c506.bin               cape-bone-ibb-00A0.dtbo            rt3090.bin
BB-BONE-LCD4-01-00A0.dtbo    DNIL-AMPCAPE-1-00R1.dtbo      bone_eqep0-00A0.dtbo             cape-bone-iio-00A0.dtbo            rt73.bin
BB-BONE-LCD4-01-00A1.dtbo    LICENCE.atheros_firmware      bone_eqep1-00A0.dtbo             cape-bone-lcd3-00A0.dtbo           rtl_nic
BB-BONE-LCD7-01-00A2.dtbo    LICENCE.broadcom_bcm43xx      bone_eqep2-00A0.dtbo             cape-bone-lcd3-00A2.dtbo           rtlwifi
BB-BONE-LCD7-01-00A3.dtbo    LICENCE.rtlwifi_firmware.txt  bone_pwm_P8_13-00A0.dtbo         cape-bone-mrf24j40-00A0.dtbo       sd8385.bin
BB-BONE-LCD7-01-00A4.dtbo    LICENCE.ti-connectivity       bone_pwm_P8_19-00A0.dtbo         cape-bone-nixie-00A0.dtbo          sd8385_helper.bin
BB-BONE-LOGIBONE-00R1.dtbo   RTL8192E                      bone_pwm_P8_34-00A0.dtbo         cape-bone-pinmux-test-00A0.dtbo    sd8686.bin
BB-BONE-PRU-01-00A0.dtbo     RTL8192SU                     bone_pwm_P8_36-00A0.dtbo         cape-bone-proto-00A0.dtbo          sd8686_helper.bin
BB-BONE-PRU-02-00A0.dtbo     TT3201-001-01.dtbo            bone_pwm_P8_45-00A0.dtbo         cape-bone-replicape-00A2.dtbo      sd8688.bin
BB-BONE-PRU-03-00A0.dtbo     am335x-pm-firmware.bin        bone_pwm_P8_46-00A0.dtbo         cape-bone-replicape-00A3.dtbo      sd8688_helper.bin
BB-BONE-PRU-04-00A0.dtbo     am33xx_pwm-00A0.dtbo          bone_pwm_P9_14-00A0.dtbo         cape-bone-tester-00A0.dtbo         ti-connectivity
BB-BONE-PWMT-00A0.dtbo       atmel_at76c502-wpa.bin        bone_pwm_P9_16-00A0.dtbo         cape-bone-weather-00A0.dtbo        usb8388.bin
BB-BONE-RS232-00A0.dtbo      atmel_at76c502.bin            bone_pwm_P9_21-00A0.dtbo         cape-bone-weather-00B0.dtbo        zd1211
BB-BONE-RST-00A0.dtbo        atmel_at76c502_3com-wpa.bin   bone_pwm_P9_22-00A0.dtbo         cape-boneblack-hdmi-00A0.dtbo
BB-BONE-RST2-00A0.dtbo       atmel_at76c502_3com.bin       bone_pwm_P9_28-00A0.dtbo         cape-boneblack-hdmin-00A0.dtbo
BB-BONE-RTC-00A0.dtbo        atmel_at76c502d-wpa.bin       bone_pwm_P9_29-00A0.dtbo         cape-univ-emmc-00A0.dtbo
BB-BONE-SERL-01-00A1.dtbo    atmel_at76c502d.bin           bone_pwm_P9_31-00A0.dtbo         cape-universal-00A0.dtbo
```

Device Tree Overlay en BeagleBone Black

- Para cargar en tiempo de ejecución un Device Tree Overlay el Cape Manager provee una interfaz para esta función.

Device Tree Overlay en BeagleBone Black

- Para cargar en tiempo de ejecución un Device Tree Overlay el Cape Manager provee una interfaz para esta función.

  `/sys/devices/bone_capemgr.*/`

Device Tree Overlay en BeagleBone Black

- Para cargar en tiempo de ejecución un Device Tree Overlay el Cape Manager provee una interfaz para esta función.

/sys/devices/bone_capemgr.*/

```
alejandro@beaglebone:/sys/devices/bone_capemgr.9$ ls -ls
total 0
0 drwxr-xr-x 2 root root    0 May 15 02:23 baseboard
0 lrwxrwxrwx 1 root root    0 May 15 02:23 driver -> ../../bus/platform/drivers/bone-capemgr
0 -r--r--r-- 1 root root 4096 May 15 02:23 modalias
0 drwxr-xr-x 2 root root    0 May 15 02:23 power
0 drwxr-xr-x 2 root root    0 May 15 02:23 slot-4
0 drwxr-xr-x 2 root root    0 May 15 02:23 slot-5
0 -rw-r--r-- 1 root root 4096 Jan  1  2000 slots
0 lrwxrwxrwx 1 root root    0 Jan  1  2000 subsystem -> ../../bus/platform
0 -rw-r--r-- 1 root root 4096 Jan  1  2000 uevent
```

Device Tree Overlay en BeagleBone Black

- Para cargar en tiempo de ejecución un Device Tree Overlay el Cape Manager provee una interfaz para esta función.

/sys/devices/bone_capemgr.*/

```
alejandro@beaglebone:/sys/devices/bone_capemgr.9$ ls -ls
total 0
0 drwxr-xr-x 2 root root    0 May 15 02:23 baseboard
0 lrwxrwxrwx 1 root root    0 May 15 02:23 driver -> ../../bus/platform/drivers/bone-capemgr
0 -r--r--r-- 1 root root 4096 May 15 02:23 modalias
0 drwxr-xr-x 2 root root    0 May 15 02:23 power
0 drwxr-xr-x 2 root root    0 May 15 02:23 slot-4
0 drwxr-xr-x 2 root root    0 May 15 02:23 slot-5
0 -rw-r--r-- 1 root root 4096 Jan  1  2000 slots
0 lrwxrwxrwx 1 root root    0 Jan  1  2000 subsystem -> ../../bus/platform
0 -rw-r--r-- 1 root root 4096 Jan  1  2000 uevent
```

```
root@beaglebone:/sys/devices/bone_capemgr.9# cat slots
 0: 54:PF---
 1: 55:PF---
 2: 56:PF---
 3: 57:PF---
 4: ff:P-O-L Bone-LT-eMMC-2G,00A0,Texas Instrument,BB-BONE-EMMC-2G
 5: ff:P-O-L Bone-Black-HDMI,00A0,Texas Instrument,BB-BONELT-HDMI
root@beaglebone:/sys/devices/bone_capemgr.9#
```

Device Tree Overlay en BeagleBone Black

Los tres primeros Slots están reservados para asignación de EEPROM IDs. Los dos siguientes son ODTs que se cargan en tiempo de ejecución. El 4 corresponde a la memoria ROM EMMC de la que se carga el Sistema Operativo. Del 5 en adelante, designados para usuario.

Device Tree Overlay en BeagleBone Black

Los tres primeros Slots están reservados para asignación de EEPROM IDs. Los dos siguientes son ODTs que se cargan en tiempo de ejecución. El 4 corresponde a la memoria ROM EMMC de la que se carga el Sistema Operativo. Del 5 en adelante, designados para usuario.

- Para agregar nuestro Overlay a un slot disponible:

Device Tree Overlay en BeagleBone Black

Los tres primeros Slots están reservados para asignación de EEPROM IDs. Los dos siguientes son ODTs que se cargan en tiempo de ejecución. El 4 corresponde a la memoria ROM EMMC de la que se carga el Sistema Operativo. Del 5 en adelante, designados para usuario.

- Para agregar nuestro Overlay a un slot disponible:

```
echo OVERLAY_NAME>/sys/devices/bone_capemgr.9/slots
```

Device Tree Overlay en BeagleBone Black

Los tres primeros Slots están reservados para asignación de EEPROM IDs. Los dos siguientes son ODTs que se cargan en tiempo de ejecución. El 4 corresponde a la memoria ROM EMMC de la que se carga el Sistema Operativo. Del 5 en adelante, designados para usuario.

- Para agregar nuestro Overlay a un slot disponible:

```
echo OVERLAY_NAME>/sys/devices/bone_capemgr.9/slots
```

- Para el ejemplo `cape-bone-mrf24j40-00A0.dtbo` :

Device Tree Overlay en BeagleBone Black

Los tres primeros Slots están reservados para asignación de EEPROM IDs. Los dos siguientes son ODTs que se cargan en tiempo de ejecución. El 4 corresponde a la memoria ROM EMMC de la que se carga el Sistema Operativo. Del 5 en adelante, designados para usuario.

- Para agregar nuestro Overlay a un slot disponible:

```
echo OVERLAY_NAME>/sys/devices/bone_capemgr.9/slots
```

- Para el ejemplo `cape-bone-mrf24j40-00A0.dtbo` :

```
echo BB-BONE-MRF24J40:00A0 > /sys/devices/bone_capemgr.9/slots
```

Device Tree Overlay en BeagleBone Black

Los tres primeros Slots están reservados para asignación de EEPROM IDs. Los dos siguientes son ODTs que se cargan en tiempo de ejecución. El 4 corresponde a la memoria ROM EMMC de la que se carga el Sistema Operativo. Del 5 en adelante, designados para usuario.

- Para agregar nuestro Overlay a un slot disponible:

```
echo OVERLAY_NAME>/sys/devices/bone_capemgr.9/slots
```

- Para el ejemplo `cape-bone-mrf24j40-00A0.dtbo` :

```
echo BB-BONE-MRF24J40:00A0 > /sys/devices/bone_capemgr.9/slots
```

- Para eliminar el Overlay:

Device Tree Overlay en BeagleBone Black

Los tres primeros Slots están reservados para asignación de EEPROM IDs. Los dos siguientes son ODTs que se cargan en tiempo de ejecución. El 4 corresponde a la memoria ROM EMMC de la que se carga el Sistema Operativo. Del 5 en adelante, designados para usuario.

- Para agregar nuestro Overlay a un slot disponible:

```
echo OVERLAY_NAME>/sys/devices/bone_capemgr.9/slots
```

- Para el ejemplo `cape-bone-mrf24j40-00A0.dtbo` :

```
echo BB-BONE-MRF24J40:00A0 > /sys/devices/bone_capemgr.9/slots
```

- Para eliminar el Overlay:

```
echo NUM_SLOT > /sys/devices/bone_capemgr.9/slots
```

Device Tree Overlay en BeagleBone Black

Los tres primeros Slots están reservados para asignación de EEPROM IDs. Los dos siguientes son ODTs que se cargan en tiempo de ejecución. El 4 corresponde a la memoria ROM EMMC de la que se carga el Sistema Operativo. Del 5 en adelante, designados para usuario.

- Para agregar nuestro Overlay a un slot disponible:

```
echo OVERLAY_NAME>/sys/devices/bone_capemgr.9/slots
```

- Para el ejemplo `cape-bone-mrf24j40-00A0.dtbo` :

```
echo BB-BONE-MRF24J40:00A0 > /sys/devices/bone_capemgr.9/slots
```

- Para eliminar el Overlay:

```
echo NUM_SLOT > /sys/devices/bone_capemgr.9/slots
```

- Con esto tenemos nuestro hardware funcionando en los pines descriptos en el Overlay.

Ejemplo de Overlay

```
1  /* OUTPUT GPIO(mode7) 0x07 pulldown, 0x17 pullup, 0x?f no pullup/down */
2  /* INPUT GPIO(mode7) 0x27 pulldown, 0x37 pullup, 0x?f no pullup/down */
3  /dts-v1/;
4  /plugin/;
5  /{
6    compatible = "ti,beaglebone", "ti,beaglebone-black";
7    part-number = "EBB-GPIO-Example";
8    version = "00A0";
9    fragment@0 {
10     target = <&am33xx_pinmux>;
11     __overlay__ {
12       ebb_example: EBB_GPIO_Example {
13    pinctrl-single,pins = <
14      0x070 0x07 // P9_11 $28 GPIO0_30=30 Output Mode7 pulldown
15      0x074 0x37 // P9_13 $29 GPIO0_31=31 Input Mode7 pullup
16    >;
17         };
18     };
19   };
```

Ejemplo de Overlay

```
1   fragment@1 {
2     target = <&ocp>;
3       __overlay__ {
4    gpio_helper {
5      compatible = "gpio-of-helper";
6      status = "okay";
7      pinctrl-names = "default";
8      pinctrl-0 = <&ebb_example>;
9    };
10      };
11    };
12  };
```

Para compilar e instalar en el file system

```
1   $ dtc -O dtb -o EBB-GPIO-Example-00A0.dtbo -b 0 -@ EBB-GPIO-Example.dt
2   $ sudo cp EBB-GPIO-Example-00A0.dtbo /lib/firmware
```

BBB Capes: acerca de la EEPROM

BBB Capes: acerca de la EEPROM

- Cada Cape debe incorporar una EEPROM para su identificación y para especificarla expnsión de pines de resultar necesario.

BBB Capes: acerca de la EEPROM

- Cada Cape debe incorporar una EEPROM para su identificación y para especificarla expnsión de pines de resultar necesario.
- El único cape que puede no tener la EEPROM es el de prototyping.

BBB Capes: acerca de la EEPROM

- Cada Cape debe incorporar una EEPROM para su identificación y para especificarla expnsión de pines de resultar necesario.
- El único cape que puede no tener la EEPROM es el de prototyping.
- Es similar a la propia EEPROM de identificación que tienen incorporada las Beagle Bone (24LC32AT-I/OT 32Kbits interfaz i2c). La del board se conecta al bus i2c2.

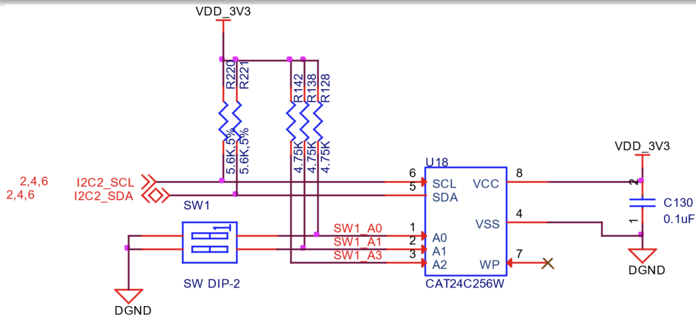
BBB Capes: acerca de la EEPROM

- Cada Cape debe incorporar una EEPROM para su identificación y para especificarla expnsión de pines de resultar necesario.
- El único cape que puede no tener la EEPROM es el de prototyping.
- Es similar a la propia EEPROM de identificación que tienen incorporada las Beagle Bone (24LC32AT-I/OT 32Kbits interfaz i2c). La del board se conecta al bus i2c2.
- En el caso de los capes se utiliza una CAT24C256 (256 Kbits interfaz i2c, buffer de escritura de 64bytes, y trabaja con las tres velocidades: 100KH.z, 400Khz., y 1MHz.).

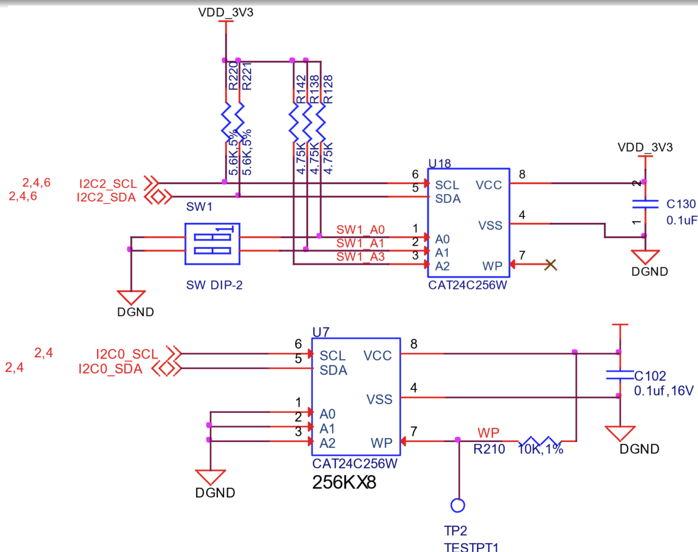
BBB Capes: acerca de la EEPROM

- Cada Cape debe incorporar una EEPROM para su identificación y para especificarla expnsión de pines de resultar necesario.
- El único cape que puede no tener la EEPROM es el de prototyping.
- Es similar a la propia EEPROM de identificación que tienen incorporada las Beagle Bone (24LC32AT-I/OT 32Kbits interfaz i2c). La del board se conecta al bus i2c2.
- En el caso de los capes se utiliza una CAT24C256 (256 Kbits interfaz i2c, buffer de escritura de 64bytes, y trabaja con las tres velocidades: 100KH.z, 400Khz., y 1MHz.).
- Hay dos conexionados posibles de acuerdo a si se emplea o no Protección contra escritura

BBB Capes: acerca de la EEPROM



BBB Capes: acerca de la EEPROM



BBB Capes: acerca de la EEPROM

Nombre	Offset	Tamaño	Descripción
Header	0	4	0xAA 0x55 0x33 0xEE
EEPROM Revision	4	2	Nro de Revisión de la EEPROM en ASCII = A1
Board Name	6	32	Nombre del board en ASCII
Version	38	4	Versión del código del Hardware en ASCII. El formato es puesto por el desarrolldor: 02.1...00A1...10A0
Manufacturer	42	16	ASCII del nombre del fabricante.
Part Number	58	16	Nro. de parte en ASCII. Lo pone el fabricante.
Number of Pins	74	2	Nro de pines usados por el cape, incluida alimentación. Nro. max.= 92. Formato hexa
Serial Number	76	12	String de 12 caracteres. Formato: WWYY&&&&nnnn. Donde: WW=semana del año de fabricación, YY = año de fabricación (dos dígitos), &&&& = Codigo de producción interno, lo fija el fabricante. nnnn = Nro. incremental de placa para la semana de producción WW.
Pin Usage	88	148	Dos bytes para cada uno de los 74 pines de los conectores de expansión. Bit 15: Uso del pin: 0=Unused by cape 1=Used by cape. Bits 14-13: Dirección del pin Pin 10=Output 01=Input 11=BDIR. Bits 12-7: Reservados (deben ser '0'). Bit 6: Slew Rate : 0=Fast 1=Slow. Bit 5: Rx Enable: 0=Disabled 1=Enabled. Bit 4: Pull Up/Down Select: 0=Pulldown 1=PullUp. Bit 3: Pull Up/Down enabled: 0=Enabled 1=Disabled. Bits 2-0: Mux Mode Selection. Mods 0 a 7
VDD_3V3B Current	236	2	Corriente max. en ma. Representación Hexa de un valor en decimal. 1500mA=0x05 0xDC 325mA=0x01 0x45
VDD_5V Current	238	2	Corriente max. en ma. Representación Hexa de un valor en decimal. 1500mA=0x05 0xDC 325mA=0x01 0x45
SYS_5V Current	240	2	Corriente max. en ma. Representación Hexa de un valor en decimal. 1500mA=0x05 0xDC 325mA=0x01 0x45
DC Supplied	242	2	Indica si el rail VDD_5V de la placa suministra tensión al cape, y el nivel de corriente. 000=No 1-0xFFFF es el valor decimal de corriente suministrado en formato hexa.
Available	244	32543	Espacio disponible para código o datos no volátiles necesarios para el fabricante del driver.Puede también tener presets para el driver.

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Char Devices

Char Devices

- Son dispositivos de E/S que intercambian datos con las aplicaciones un byte a la vez, enviándolos en forma de streams.

Char Devices

- Son dispositivos de E/S que intercambian datos con las aplicaciones un byte a la vez, enviándolos en forma de streams.
- Son los dispositivos mas básicos a nivel de fuentes en el kernel.

Char Devices

- Son dispositivos de E/S que intercambian datos con las aplicaciones un byte a la vez, enviándolos en forma de streams.
- Son los dispositivos mas básicos a nivel de fuentes en el kernel.
- Se los representa internamente mediante instancias de la estructura `struct cdev` definida en `include/linux/cdev.h` .

Char Devices

- Son dispositivos de E/S que intercambian datos con las aplicaciones un byte a la vez, enviándolos en forma de streams.
- Son los dispositivos mas básicos a nivel de fuentes en el kernel.
- Se los representa internamente mediante instancias de la estructura `struct cdev` definida en `include/linux/cdev.h` .

Char Devices

- Son dispositivos de E/S que intercambian datos con las aplicaciones un byte a la vez, enviándolos en forma de streams.
- Son los dispositivos mas básicos a nivel de fuentes en el kernel.
- Se los representa internamente mediante instancias de la estructura **`struct cdev`** definida en **`include/linux/cdev.h`**.

```
struct cdev {
  struct kobject kobj;
  struct module *owner;
  const struct file_operations *ops;
  struct list_head list;
  dev_t dev;
  unsigned int count;
};
```

- La función **`struct cdev * cdev_alloc (void);`** obtiene memoria para alojar una instancia de **`struct cdev`**

Char Devices

- Son dispositivos de E/S que intercambian datos con las aplicaciones un byte a la vez, enviándolos en forma de streams.
- Son los dispositivos mas básicos a nivel de fuentes en el kernel.
- Se los representa internamente mediante instancias de la estructura `struct cdev` definida en `include/linux/cdev.h`.

```
struct cdev {
  struct kobject kobj;
  struct module *owner;
  const struct file_operations *ops;
  struct list_head list;
  dev_t dev;
  unsigned int count;
};
```

- La función `struct cdev * cdev_alloc (void);` obtiene memoria para alojar una instancia de `struct cdev`
- Para eliminarla del kernel `void cdev_del(struct cdev *dev);`

Char Devices

- Exporta sus propiedades y funcionalidades por medio de un nodo del file system, en el directorio `/dev`. Para verlos `ls -l /dev`.

Char Devices

- Exporta sus propiedades y funcionalidades por medio de un nodo del file system, en el directorio `/dev`. Para verlos `ls -l /dev`.

```
crw-------    1 root root      10,  61 ago 10 19:57 network_latency
crw-------    1 root root      10,  60 ago 10 19:57 network_throughput
crw-rw-rw-    1 root root       1,   3 ago 10 19:57 null
crw-r-----    1 root kmem       1,   4 ago 10 19:57 port
crw-------    1 root root     108,   0 ago 10 19:57 ppp
crw-------    1 root root      10,   1 ago 10 19:57 psaux
crw-rw-rw-    1 root tty        5,   2 ago 11 09:44 ptmx
```

Char Devices

- Exporta sus propiedades y funcionalidades por medio de un nodo del file system, en el directorio `/dev`. Para verlos `ls -l /dev`.

```
crw-------   1 root root     10, 61 ago 10 19:57 network_latency
crw-------   1 root root     10, 60 ago 10 19:57 network_throughput
crw-rw-rw-   1 root root      1,  3 ago 10 19:57 null
crw-r-----   1 root kmem      1,  4 ago 10 19:57 port
crw-------   1 root root    108,  0 ago 10 19:57 ppp
crw-------   1 root root     10,  1 ago 10 19:57 psaux
crw-rw-rw-   1 root tty       5,  2 ago 11 09:44 ptmx
```

- La pregunta es ¿como que el kernel logra la abstracción?.

Char Devices

- Exporta sus propiedades y funcionalidades por medio de un nodo del file system, en el directorio `/dev`. Para verlos `ls -l /dev`.

```
crw-------    1 root root     10,  61 ago 10 19:57 network_latency
crw-------    1 root root     10,  60 ago 10 19:57 network_throughput
crw-rw-rw-    1 root root      1,   3 ago 10 19:57 null
crw-r-----    1 root kmem      1,   4 ago 10 19:57 port
crw-------    1 root root    108,   0 ago 10 19:57 ppp
crw-------    1 root root     10,   1 ago 10 19:57 psaux
crw-rw-rw-    1 root tty       5,   2 ago 11 09:44 ptmx
```

- La pregunta es ¿como que el kernel logra la abstracción?.
- La primer columna (resaltada en el gráfico es el tipo de nodo)

Char Devices

- Exporta sus propiedades y funcionalidades por medio de un nodo del file system, en el directorio `/dev`. Para verlos `ls -l /dev`.

```
crw-------   1 root root     10,  61 ago 10 19:57 network_latency
crw-------   1 root root     10,  60 ago 10 19:57 network_throughput
crw-rw-rw-   1 root root      1,   3 ago 10 19:57 null
crw-r-----   1 root kmem      1,   4 ago 10 19:57 port
crw-------   1 root root    108,   0 ago 10 19:57 ppp
crw-------   1 root root     10,   1 ago 10 19:57 psaux
crw-rw-rw-   1 root tty       5,   2 ago 11 09:44 ptmx
```

- La pregunta es ¿como que el kernel logra la abstracción?.
- La primer columna (resaltada en el gráfico es el tipo de nodo)
  - `c` Tipo de nodo dispositivo de caracter

Char Devices

- Exporta sus propiedades y funcionalidades por medio de un nodo del file system, en el directorio `/dev`. Para verlos `ls -l /dev`.

```
crw-------    1 root  root      10,  61 ago 10 19:57 network_latency
crw-------    1 root  root      10,  60 ago 10 19:57 network_throughput
crw-rw-rw-    1 root  root       1,   3 ago 10 19:57 null
crw-r-----    1 root  kmem       1,   4 ago 10 19:57 port
crw-------    1 root  root     108,   0 ago 10 19:57 ppp
crw-------    1 root  root      10,   1 ago 10 19:57 psaux
crw-rw-rw-    1 root  tty        5,   2 ago 11 09:44 ptmx
```

- La pregunta es ¿como que el kernel logra la abstracción?.
- La primer columna (resaltada en el gráfico es el tipo de nodo)
  - `c` Tipo de nodo dispositivo de caracter
  - `b` Tipo de nodo dispositivo de bloque

Char Devices

- Exporta sus propiedades y funcionalidades por medio de un nodo del file system, en el directorio `/dev`. Para verlos `ls -l /dev`.

```
crw-------    1 root  root      10,  61 ago 10 19:57 network_latency
crw-------    1 root  root      10,  60 ago 10 19:57 network_throughput
crw-rw-rw-    1 root  root       1,   3 ago 10 19:57 null
crw-r-----    1 root  kmem       1,   4 ago 10 19:57 port
crw-------    1 root  root     108,   0 ago 10 19:57 ppp
crw-------    1 root  root      10,   1 ago 10 19:57 psaux
crw-rw-rw-    1 root  tty        5,   2 ago 11 09:44 ptmx
```

- La pregunta es ¿como que el kernel logra la abstracción?.
- La primer columna (resaltada en el gráfico es el tipo de nodo)
  - `c` Tipo de nodo dispositivo de caracter
  - `b` Tipo de nodo dispositivo de bloque
  - `l` Tipo de nodo link simbólico

Char Devices

- Exporta sus propiedades y funcionalidades por medio de un nodo del file system, en el directorio `/dev`. Para verlos `ls -l /dev`.

```
crw-------    1 root root      10,  61 ago 10 19:57 network_latency
crw-------    1 root root      10,  60 ago 10 19:57 network_throughput
crw-rw-rw-    1 root root       1,   3 ago 10 19:57 null
crw-r-----    1 root kmem       1,   4 ago 10 19:57 port
crw-------    1 root root     108,   0 ago 10 19:57 ppp
crw-------    1 root root      10,   1 ago 10 19:57 psaux
crw-rw-rw-    1 root tty        5,   2 ago 11 09:44 ptmx
```

- La pregunta es ¿como que el kernel logra la abstracción?.
- La primer columna (resaltada en el gráfico es el tipo de nodo)
  - `c` Tipo de nodo dispositivo de caracter
  - `b` Tipo de nodo dispositivo de bloque
  - `l` Tipo de nodo link simbólico
  - `d` Tipo de nodo directorio

Char Devices

- Exporta sus propiedades y funcionalidades por medio de un nodo del file system, en el directorio `/dev`. Para verlos `ls -l /dev`.

```
crw-------    1 root root      10,  61 ago 10 19:57 network_latency
crw-------    1 root root      10,  60 ago 10 19:57 network_throughput
crw-rw-rw-    1 root root       1,   3 ago 10 19:57 null
crw-r-----    1 root kmem       1,   4 ago 10 19:57 port
crw-------    1 root root     108,   0 ago 10 19:57 ppp
crw-------    1 root root      10,   1 ago 10 19:57 psaux
crw-rw-rw-    1 root tty        5,   2 ago 11 09:44 ptmx
```

- La pregunta es ¿como que el kernel logra la abstracción?.
- La primer columna (resaltada en el gráfico es el tipo de nodo)
  - `c` Tipo de nodo dispositivo de caracter
  - `b` Tipo de nodo dispositivo de bloque
  - `l` Tipo de nodo link simbólico
  - `d` Tipo de nodo directorio
  - `s` Tipo de nodo socket

Char Devices

- Exporta sus propiedades y funcionalidades por medio de un nodo del file system, en el directorio `/dev`. Para verlos `ls -l /dev`.

```
crw-------  1 root root   10, 61 ago 10 19:57 network_latency
crw-------  1 root root   10, 60 ago 10 19:57 network_throughput
crw-rw-rw-  1 root root    1,  3 ago 10 19:57 null
crw-r-----  1 root kmem    1,  4 ago 10 19:57 port
crw-------  1 root root  108,  0 ago 10 19:57 ppp
crw-------  1 root root   10,  1 ago 10 19:57 psaux
crw-rw-rw-  1 root tty     5,  2 ago 11 09:44 ptmx
```

- La pregunta es ¿como que el kernel logra la abstracción?.
- La primer columna (resaltada en el gráfico es el tipo de nodo)
  - `c` Tipo de nodo dispositivo de caracter
  - `b` Tipo de nodo dispositivo de bloque
  - `l` Tipo de nodo link simbólico
  - `d` Tipo de nodo directorio
  - `s` Tipo de nodo socket
  - `p` Tipo de nodo pipe

Major and Minor Numbers

Major and Minor Numbers

```
crw--w----   1 root   tty   4,  0 oct 19 21:06 tty0
crw--w----   1 gdm    tty   4,  1 oct 19 21:06 tty1
crw--w----   1 root   tty   4, 10 oct 19 21:06 tty10
crw--w----   1 root   tty   4, 11 oct 19 21:06 tty11
crw--w----   1 root   tty   4, 12 oct 19 21:06 tty12
crw--w----   1 root   tty   4, 13 oct 19 21:06 tty13
crw--w----   1 root   tty   4, 14 oct 19 21:06 tty14
crw--w----   1 root   tty   4, 15 oct 19 21:06 tty15
crw--w----   1 root   tty   4, 16 oct 19 21:06 tty16
crw--w----   1 root   tty   4, 17 oct 19 21:06 tty17
crw--w----   1 root   tty   4, 18 oct 19 21:06 tty18
crw--w----   1 root   tty   4, 19 oct 19 21:06 tty19
```

Nº. Mayor

Nº. Menor

Major and Minor Numbers

```
crw--w----  1 root  tty   4,  0 oct 19 21:06 tty0
crw--w----  1 gdm   tty   4,  1 oct 19 21:06 tty1
crw--w----  1 root  tty   4, 10 oct 19 21:06 tty10
crw--w----  1 root  tty   4, 11 oct 19 21:06 tty11
crw--w----  1 root  tty   4, 12 oct 19 21:06 tty12
crw--w----  1 root  tty   4, 13 oct 19 21:06 tty13
crw--w----  1 root  tty   4, 14 oct 19 21:06 tty14
crw--w----  1 root  tty   4, 15 oct 19 21:06 tty15
crw--w----  1 root  tty   4, 16 oct 19 21:06 tty16
crw--w----  1 root  tty   4, 17 oct 19 21:06 tty17
crw--w----  1 root  tty   4, 18 oct 19 21:06 tty18
crw--w----  1 root  tty   4, 19 oct 19 21:06 tty19
```

Nº. Mayor

Nº. Menor

MAJOR NUMBER Identifica al driver asociado con el dispositivo y el “tipo” de dispositivo al que se hace referencia.

Major and Minor Numbers

```
crw--w----   1 root    tty    4,  0 oct 19 21:06 tty0
crw--w----   1 gdm     tty    4,  1 oct 19 21:06 tty1
crw--w----   1 root    tty    4, 10 oct 19 21:06 tty10
crw--w----   1 root    tty    4, 11 oct 19 21:06 tty11
crw--w----   1 root    tty    4, 12 oct 19 21:06 tty12
crw--w----   1 root    tty    4, 13 oct 19 21:06 tty13
crw--w----   1 root    tty    4, 14 oct 19 21:06 tty14
crw--w----   1 root    tty    4, 15 oct 19 21:06 tty15
crw--w----   1 root    tty    4, 16 oct 19 21:06 tty16
crw--w----   1 root    tty    4, 17 oct 19 21:06 tty17
crw--w----   1 root    tty    4, 18 oct 19 21:06 tty18
crw--w----   1 root    tty    4, 19 oct 19 21:06 tty19
```

Nº. Mayor

Nº. Menor

MAJOR NUMBER Identifica al driver asociado con el dispositivo y el “tipo” de dispositivo al que se hace referencia.

MINOR NUMBER Identifica al dispositivo en particular o a una determinada interfaz.

Major and Minor Numbers

```
crw--w----    1 root        tty        4,   0 oct 19 21:06 tty0
crw--w----    1 gdm         tty        4,   1 oct 19 21:06 tty1
crw--w----    1 root        tty        4,  10 oct 19 21:06 tty10
crw--w----    1 root        tty        4,  11 oct 19 21:06 tty11
crw--w----    1 root        tty        4,  12 oct 19 21:06 tty12
crw--w----    1 root        tty        4,  13 oct 19 21:06 tty13
crw--w----    1 root        tty        4,  14 oct 19 21:06 tty14
crw--w----    1 root        tty        4,  15 oct 19 21:06 tty15
crw--w----    1 root        tty        4,  16 oct 19 21:06 tty16
crw--w----    1 root        tty        4,  17 oct 19 21:06 tty17
crw--w----    1 root        tty        4,  18 oct 19 21:06 tty18
crw--w----    1 root        tty        4,  19 oct 19 21:06 tty19
```


MAJOR NUMBER Identifica al driver asociado con el dispositivo y el “tipo” de dispositivo al que se hace referencia.

MINOR NUMBER Identifica al dispositivo en particular o a una determinada interfaz.

Para mas información sobre Major & Minors:

https://www.kernel.org/doc/Documentation/devices.txt

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

dev_t type

dev_t type

- Si urgamos en `linux/types.h` , veremos un tipo `dev_t` type, que no es otra cosa que una redefinición de un `unit32` .

dev_t type

- Si urgamos en `linux/types.h`, veremos un tipo `dev_t` type, que no es otra cosa que una redefinición de un `unit32`.
- Utilizado para almacenar el Major y el Minor Number.

dev_t type

- Si urgamos en `linux/types.h`, veremos un tipo `dev_t` type, que no es otra cosa que una redefinición de un `unit32`.
- Utilizado para almacenar el Major y el Minor Number.
- El almacenamiento de ambos números dentro del tipo `dev_t`: 12 bits para el Major y 20 para el Minor.

dev_t type

- Si urgamos en `linux/types.h`, veremos un tipo `dev_t` type, que no es otra cosa que una redefinición de un `unit32`.
- Utilizado para almacenar el Major y el Minor Number.
- El almacenamiento de ambos números dentro del tipo `dev_t`: 12 bits para el Major y 20 para el Minor.
- Es deseable que éstos detalles sean transparentes al desarrollador y prevenir que cambios futuros dejen incompatible a nuestro driver.

dev_t type

- Si urgamos en `linux/types.h`, veremos un tipo `dev_t` type, que no es otra cosa que una redefinición de un `unit32`.
- Utilizado para almacenar el Major y el Minor Number.
- El almacenamiento de ambos números dentro del tipo `dev_t`: 12 bits para el Major y 20 para el Minor.
- Es deseable que éstos detalles sean transparentes al desarrollador y prevenir que cambios futuros dejen incompatible a nuestro driver.
- Por eso, para obtener el valor del Minor Number y el Major Number de un tipo `dev_t` se utilizan las dos macros siguientes:

```
MAJOR(dev_t dev);

MINOR(dev_t dev);
```

dev_t type

- Si urgamos en `linux/types.h`, veremos un tipo `dev_t` type, que no es otra cosa que una redefinición de un `unit32`.
- Utilizado para almacenar el Major y el Minor Number.
- El almacenamiento de ambos números dentro del tipo `dev_t`: 12 bits para el Major y 20 para el Minor.
- Es deseable que éstos detalles sean transparentes al desarrollador y prevenir que cambios futuros dejen incompatible a nuestro driver.
- Por eso, para obtener el valor del Minor Number y el Major Number de un tipo `dev_t` se utilizan las dos macros siguientes:

```
MAJOR(dev_t dev);

MINOR(dev_t dev);
```

- Estas macros se encuentran definidas en `linux/kdev_t`

Generación de un `dev_t`

Generación de un `dev_t`

- Para generar un tipo `dev_t` a partir de los valores major y minor ya obtenidos, también se dispone de una macro para evitar incompatibilidades a futuro:

Generación de un dev_t

- Para generar un tipo dev_t a partir de los valores major y minor ya obtenidos, también se dispone de una macro para evitar incompatibilidades a futuro:

```
MKDEV(int major, int minor);
```

Generación de un `dev_t`

- Para generar un tipo `dev_t` a partir de los valores major y minor ya obtenidos, también se dispone de una macro para evitar incompatibilidades a futuro:

```
MKDEV(int major, int minor);
```

- También se encuentra en `linux/kdev_t.h`

Generación de un `dev_t`

- Para generar un tipo `dev_t` a partir de los valores major y minor ya obtenidos, también se dispone de una macro para evitar incompatibilidades a futuro:

```
MKDEV(int major, int minor);
```

- También se encuentra en `linux/kdev_t.h`

```
1 #define MINORBITS 20
2 #define MINORMASK ((1U << MINORBITS) - 1)
3 #define MAJOR(dev) ((unsigned int) ((dev) >> MINORBITS))
4 #define MINOR(dev) ((unsigned int) ((dev) & MINORMASK))
5 #define MKDEV(ma,mi) (((ma) << MINORBITS) (mi))
```

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - **Funciones para gestión de Major y Minor Numbers**
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Reservar y Liberar Device Numbers

Reservar y Liberar Device Numbers

- Para reservar major y minor numbers para el dispositivo es recomendable hacerlo en forma dinámica:

Reservar y Liberar Device Numbers

- Para reservar major y minor numbers para el dispositivo es recomendable hacerlo en forma dinámica:

```
int alloc_chrdev_region (dev_t *dev, unsigned int firstminor,
    unsigned int count, char *name);
```

Reservar y Liberar Device Numbers

- Para reservar major y minor numbers para el dispositivo es recomendable hacerlo en forma dinámica:

```
int alloc_chrdev_region (dev_t *dev, unsigned int firstminor,
    unsigned int count, char *name);
```

Donde

- **dev_t *dev** : Estructura con MAJOR y primer MINOR reservado por el kernel.

Reservar y Liberar Device Numbers

- Para reservar major y minor numbers para el dispositivo es recomendable hacerlo en forma dinámica:

```
int alloc_chrdev_region (dev_t *dev, unsigned int firstminor,
    unsigned int count, char *name);
```

Donde

- **dev_t *dev** : Estructura con MAJOR y primer MINOR reservado por el kernel.
- **unsigned int firstminor** : Primer MINOR solicitado

Reservar y Liberar Device Numbers

- Para reservar major y minor numbers para el dispositivo es recomendable hacerlo en forma dinámica:

```
int alloc_chrdev_region (dev_t *dev, unsigned int firstminor,
    unsigned int count, char *name);
```

Donde

- `dev_t *dev` : Estructura con MAJOR y primer MINOR reservado por el kernel.
- `unsigned int firstminor` : Primer MINOR solicitado
- `unsigned int count` : Cantidad de números menores solicitados

Reservar y Liberar Device Numbers

- Para reservar major y minor numbers para el dispositivo es recomendable hacerlo en forma dinámica:

```
int alloc_chrdev_region (dev_t *dev, unsigned int firstminor,
    unsigned int count, char *name);
```

Donde

- **dev_t *dev** : Estructura con MAJOR y primer MINOR reservado por el kernel.
- **unsigned int firstminor** : Primer MINOR solicitado
- **unsigned int count** : Cantidad de números menores solicitados
- **char *name** : Nombre del device que estará asociado con este rango.

Reservar y Liberar Device Numbers

- Para reservar major y minor numbers para el dispositivo es recomendable hacerlo en forma dinámica:

```
int alloc_chrdev_region (dev_t *dev, unsigned int firstminor,
    unsigned int count, char *name);
```

Donde

- `dev_t *dev` : Estructura con MAJOR y primer MINOR reservado por el kernel.
- `unsigned int firstminor` : Primer MINOR solicitado
- `unsigned int count` : Cantidad de números menores solicitados
- `char *name` : Nombre del device que estará asociado con este rango.
- Retorna 0 si terminó correctamente o un valor negativo en caso de error.

Reservar y Liberar Device Numbers

Reservar y Liberar Device Numbers

- Para liberar major y minor numbers del dispositivo también se dispone de una función específica del kernel:

Reservar y Liberar Device Numbers

- Para liberar major y minor numbers del dispositivo también se dispone de una función específica del kernel:

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

Reservar y Liberar Device Numbers

- Para liberar major y minor numbers del dispositivo también se dispone de una función específica del kernel:

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

Donde

- `dev_t first` : Estructura con MAJOR y primer MINOR reservado por el kernel.

Reservar y Liberar Device Numbers

- Para liberar major y minor numbers del dispositivo también se dispone de una función específica del kernel:

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

Donde

- `dev_t first` : Estructura con MAJOR y primer MINOR reservado por el kernel.
- `unsigned int count` : Cantidad de números menores oportunamente asignados al device.

Asignación dinámica de Major number

Asignación dinámica de Major number

- Una de las primeras cosas que debe hacer un driver al inicializarse durante su instalación es obtener uno o mas números de dispositivo.

Asignación dinámica de Major number

- Una de las primeras cosas que debe hacer un driver al inicializarse durante su instalación es obtener uno o mas números de dispositivo.
- `alloc_chrdev_region` , deja que el kernel seleccione un número mayor disponible y lo asigne a nuestro device.

Asignación dinámica de Major number

- Una de las primeras cosas que debe hacer un driver al inicializarse durante su instalación es obtener uno o mas números de dispositivo.
- `alloc_chrdev_region` , deja que el kernel seleccione un número mayor disponible y lo asigne a nuestro device.
- `register_chrdev_region` , es una alternativa estática en donde en lugar de pasar un puntero a `dev_t` vacío se lo pasamos inicializado.

Asignación dinámica de Major number

- Una de las primeras cosas que debe hacer un driver al inicializarse durante su instalación es obtener uno o mas números de dispositivo.
- `alloc_chrdev_region` , deja que el kernel seleccione un número mayor disponible y lo asigne a nuestro device.
- `register_chrdev_region` , es una alternativa estática en donde en lugar de pasar un puntero a `dev_t` vacío se lo pasamos inicializado.
- Ventaja: Podemos en el mismo código de inicialización crear el device file en `/dev` .

Asignación dinámica de Major number

- Una de las primeras cosas que debe hacer un driver al inicializarse durante su instalación es obtener uno o mas números de dispositivo.
- `alloc_chrdev_region` , deja que el kernel seleccione un número mayor disponible y lo asigne a nuestro device.
- `register_chrdev_region` , es una alternativa estática en donde en lugar de pasar un puntero a `dev_t` vacío se lo pasamos inicializado.
- Ventaja: Podemos en el mismo código de inicialización crear el device file en `/dev` .
- Desventaja: Probablemente se colisione con un device pre instalado que ya tenga ese número.

Asignación dinámica de Major number

- `/proc/devices` contiene los devices y su Major number

```
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
....
 14 dsp
 ...
180 usb
189 usb_device
```

Asignación dinámica de Major number

- `/proc/devices` contiene los devices y su Major number

```
189 usb_device
...
...
Block devices:
  8 sd
 11 sr
 65 sd
 66 sd
 67 sd
 68 sd
...
134 sd
135 sd
253 nd_blk
254 btt
259 blkext
```

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Registro de un char device en el kernel

[1] La desarrollaremos en algunos slides

Registro de un char device en el kernel

- Como el char device gira en torno de la estructura `cdev`, para registrarlo es necesario inicializarla y agregarla a la estructura de char devices del kernel.

[1] La desarrollaremos en algunos slides

Registro de un char device en el kernel

- Como el char device gira en torno de la estructura `cdev`, para registrarlo es necesario inicializarla y agregarla a la estructura de char devices del kernel.
- Los miembros de esta estructura se pueden inicializar a mano, aunque también se dispone de funciones del kernel para resolverlo.

[1] La desarrollaremos en algunos slides

Registro de un char device en el kernel

- Como el char device gira en torno de la estructura `cdev`, para registrarlo es necesario inicializarla y agregarla a la estructura de char devices del kernel.
- Los miembros de esta estructura se pueden inicializar a mano, aunque también se dispone de funciones del kernel para resolverlo.
- Por ejemplo, una vez definida la estructura `file_operations` [1] su puntero se puede asignar al miembro de la estructura `cdev`, o se puede utilizar la siguiente función:

[1] La desarrollaremos en algunos slides

Registro de un char device en el kernel

- Como el char device gira en torno de la estructura **cdev**, para registrarlo es necesario inicializarla y agregarla a la estructura de char devices del kernel.
- Los miembros de esta estructura se pueden inicializar a mano, aunque también se dispone de funciones del kernel para resolverlo.
- Por ejemplo, una vez definida la estructura **file_operations** [1] su puntero se puede asignar al miembro de la estructura **cdev**, o se puede utilizar la siguiente función:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

[1] La desarrollaremos en algunos slides

Registro de un char device en el kernel

- Como el char device gira en torno de la estructura `cdev`, para registrarlo es necesario inicializarla y agregarla a la estructura de char devices del kernel.
- Los miembros de esta estructura se pueden inicializar a mano, aunque también se dispone de funciones del kernel para resolverlo.
- Por ejemplo, una vez definida la estructura `file_operations` [1] su puntero se puede asignar al miembro de la estructura `cdev`, o se puede utilizar la siguiente función:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

- Donde `struct cdev *cdev` es el puntero a la estructura `cdev`.

[1] La desarrollaremos en algunos slides

Registro de un char device en el kernel

- Como el char device gira en torno de la estructura **cdev** , para registrarlo es necesario inicializarla y agregarla a la estructura de char devices del kernel.
- Los miembros de esta estructura se pueden inicializar a mano, aunque también se dispone de funciones del kernel para resolverlo.
- Por ejemplo, una vez definida la estructura **file_operations** [1] su puntero se puede asignar al miembro de la estructura **cdev** , o se puede utilizar la siguiente función:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

- Donde **struct cdev *cdev** es el puntero a la estructura **cdev** .
- y **struct file_operations *fops** es el puntero a la estructura **file_operations** .

[1] La desarrollaremos en algunos slides

Registro de un char device en el kernel

- Para registrar `cdev` en el kernel utilizamos la función

Registro de un char device en el kernel

- Para registrar **cdev** en el kernel utilizamos la función

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

Registro de un char device en el kernel

- Para registrar `cdev` en el kernel utilizamos la función

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

- Donde `struct cdev *cdev` es el puntero a `cdev`.

Registro de un char device en el kernel

- Para registrar `cdev` en el kernel utilizamos la función

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

- Donde `struct cdev *cdev` es el puntero a `cdev`.
- `dev_t num` contiene los Minor & Major numbers.

Registro de un char device en el kernel

- Para registrar `cdev` en el kernel utilizamos la función

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

- Donde `struct cdev *cdev` es el puntero a `cdev`.
- `dev_t num` contiene los Minor & Major numbers.
- y `unsigned int count` es la cantidad de Minors.

Registro de un char device en el kernel

- Una vez creado el device mediante `alloc_chrdev_region ()` , nos resta generar la clase de dispositivo.

Registro de un char device en el kernel

- Una vez creado el device mediante `alloc_chrdev_region ()` , nos resta generar la clase de dispositivo.
- Se utiliza al función del kernel `class_create()`

Registro de un char device en el kernel

- Una vez creado el device mediante `alloc_chrdev_region ()` , nos resta generar la clase de dispositivo.
- Se utiliza al función del kernel `class_create()`
- Esta generará el atributo clase requerido en el Linux Drivers Model.

Registro de un char device en el kernel

- Una vez creado el device mediante `alloc_chrdev_region ()` , nos resta generar la clase de dispositivo.
- Se utiliza al función del kernel `class_create()`
- Esta generará el atributo clase requerido en el Linux Drivers Model.
- Se reflejará en uno de los virtual file system `/sys/class/`

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Estructura File Operations

Estructura File Operations

- La estructura File Operations es la pieza clave de un driver, ya que vincula las llamadas de user space (`open ()` , `read ()` , `write ()` , `close ()` , `ioctl ()`) con las funcionalidades del driver / módulo.

Estructura File Operations

- La estructura File Operations es la pieza clave de un driver, ya que vincula las llamadas de user space (`open ()` , `read ()` , `write ()` , `close ()` , `ioctl ()`) con las funcionalidades del driver / módulo.
- Está compuesta por un set de punteros a función.

Estructura File Operations

- La estructura File Operations es la pieza clave de un driver, ya que vincula las llamadas de user space (`open ()` , `read ()` , `write ()` , `close ()` , `ioctl ()`) con las funcionalidades del driver / módulo.
- Está compuesta por un set de punteros a función.
- La estructura está definida como: `file_operations` , en `linux/fs.h`

Estructura File Operations

- La estructura File Operations es la pieza clave de un driver, ya que vincula las llamadas de user space (`open ()` , `read ()` , `write ()` , `close ()` , `ioctl ()`) con las funcionalidades del driver / módulo.
- Está compuesta por un set de punteros a función.
- La estructura está definida como: `file_operations` , en `linux/fs.h`
- Cada nodo `/dev` tiene asociado una estructura de este tipo, que le permite exponer sus funcionalidades y características particulares a las aplicaciones.

Estructura File Operations

```
struct file_operations {
 struct module *owner;
 loff_t (*llseek) (struct file *, loff_t , int);
 ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
 ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
 ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
 ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
 int (*iterate) (struct file *, struct dir_context *);
 int (*iterate_shared) (struct file *, struct dir_context *);
 unsigned int (*poll) (struct file *, struct poll_table_struct *);
 long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
 long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
 int (*mmap) (struct file *, struct vm_area_struct *);
 int (*flush) (struct file *, fl_owner_t id);
 int (*open) (struct inode *, struct file *);
 int (*release) (struct inode *, struct file *);
 int (*fsync) (struct file *, loff_t, loff_t, int datasync);
 int (*aio_fsync) (struct kiocb *, int datasync);
 int (*fasync) (int, struct file *, int);
 int (*lock) (struct file *, int, struct file_lock *);
 ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
 unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
 int (*check_flags)(int);
 int (*flock) (struct file *, int, struct file_lock *);
 ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
 ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
 int (*setlease)(struct file *, long, struct file_lock **, void **);
 long (*fallocate)(struct file *file, int mode, loff_t offset,loff_t len);
```

Estructura File Operations

```
 void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
  unsigned (*mmap_capabilities)(struct file *);
#endif
 ssize_t (*copy_file_range)(struct file *, loff_t , struct file *,loff_t , size_t , unsigned int);
 int (*clone_file_range)(struct file *, loff_t , struct file *, loff_t ,u64);
 ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *,u64);
};
```

Estructura File Operations

```
 void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
  unsigned (*mmap_capabilities)(struct file *);
#endif
 ssize_t (*copy_file_range)(struct file *, loff_t , struct file *,loff_t , size_t , unsigned int);
 int (*clone_file_range)(struct file *, loff_t , struct file *, loff_t ,u64);
 ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *,u64);
};
```

Estructura File Operations

```
 void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
  unsigned (*mmap_capabilities)(struct file *);
#endif
 ssize_t (*copy_file_range)(struct file *, loff_t , struct file *,loff_t , size_t , unsigned int);
 int (*clone_file_range)(struct file *, loff_t , struct file *, loff_t ,u64);
 ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *,u64);
};
```

Para bajar un poco a detalle nos concentraremos en nuestras 5 funciones de interés.

Llenando file_operations

```
/* Declaro las file operations*/
static struct file_operations mydev_ops = {
    .owner   = THIS_MODULE,
    .read    = mydev_read,
    .write   = mydev_write,
    .open    = mydev_open,
    .release = mydev_close,
};
```

Llenando file_operations

```
/* Declaro las file operations*/
static struct file_operations mydev_ops = {
    .owner   = THIS_MODULE,
    .read    = mydev_read,
    .write   = mydev_write,
    .open    = mydev_open,
    .release = mydev_close,
};
```

- Solo se definen los métodos que implementará nuestro driver.

Llenando file_operations

```
/* Declaro las file operations*/
static struct file_operations mydev_ops = {
    .owner    = THIS_MODULE,
    .read     = mydev_read,
    .write    = mydev_write,
    .open     = mydev_open,
    .release  = mydev_close,
};
```

- Solo se definen los métodos que implementará nuestro driver.
- El resto el kernel los asume `NULL`

Llenando file_operations

```
/* Declaro las file operations*/
static struct file_operations mydev_ops = {
    .owner   = THIS_MODULE,
    .read    = mydev_read,
    .write   = mydev_write,
    .open    = mydev_open,
    .release = mydev_close,
};
```

- Solo se definen los métodos que implementará nuestro driver.
- El resto el kernel los asume `NULL`
- Cada nombre en `file_operations` será el de la función de nuestro módulo que implementará el método (Ej: `mydev_read`, será la función a la que el kernel invocará cuando un proceso ejecute `read ()`.

Primeros pasos en la inicialización

```
#define MENOR 0
#define CANT_DISP 1
static dev_t   dispo;

int mydev_init (void)
{
...
    /*Instanciamos nuestro dispositivo*/
    /*Pedimos numero mayor de forma dinámica*/
    mydev_cdev = cdev_alloc();
    if (  (alloc_chrdev_region(&dispo, MENOR, CANT_DISP, "td3_mydev")) < 0)
    {
        printk(KERN_ALERT "td3_mydev: Error no es posible asignar numero mayor\n");
        return -EBUSY;
    }
    printk(KERN_ALERT "td3_mydev: Numero mayor asignado: % x\n", MAJOR(dispo));
    /*Registramos el dispositivo en el sistema*/
    mydev_cdev->ops = &i2c_ops;
    mydev_cdev->owner = THIS_MODULE;
    mydev_cdev->dev = dispo;
    if ((cdev_add (mydev_cdev, dispo, CANT_DISP)) < 0)
    {
        unregister_chrdev_region(dispo, CANT_DISP);
        printk(KERN_ALERT "td3_mydev: No es posible registrar el dispositivo\n");
        return -EBUSY;
    }
    ....
```

Método open

```
int (*open) (struct inode *inode, struct file *filp);
```

Método open

```
int (*open) (struct inode *inode, struct file *filp);
```

- Como sabemos, es la primer operación realizada sobre el archivo, o device en este caso.

Método open

```
int (*open) (struct inode *inode, struct file *filp);
```

- Como sabemos, es la primer operación realizada sobre el archivo, o device en este caso.
- Podemos obviar su implementación en el driver (no es recomendable, pero se puede), y de todos modos el kernel resuelve por default su tarea: genera una instancia de apertura y devuelve el file descriptor.

Método open

```
int (*open) (struct inode *inode, struct file *filp);
```

- Como sabemos, es la primer operación realizada sobre el archivo, o device en este caso.
- Podemos obviar su implementación en el driver (no es recomendable, pero se puede), y de todos modos el kernel resuelve por default su tarea: genera una instancia de apertura y devuelve el file descriptor.
- Si no se usa se define como **`NULL`** en **`file_operations`** (como corresponde con puntero que no se usa), y el kernel abre el device pero no le notifica nada al driver.

Método open

```
int (*open) (struct inode *inode, struct file *filp);
```

- Como sabemos, es la primer operación realizada sobre el archivo, o device en este caso.
- Podemos obviar su implementación en el driver (no es recomendable, pero se puede), y de todos modos el kernel resuelve por default su tarea: genera una instancia de apertura y devuelve el file descriptor.
- Si no se usa se define como `NULL` en `file_operations` (como corresponde con puntero que no se usa), y el kernel abre el device pero no le notifica nada al driver.
- Trabaja con dos estructuras fundamentales para el manejo del File System.

Método open

```
int (*open) (struct inode *inode, struct file *filp);
```

- Como sabemos, es la primer operación realizada sobre el archivo, o device en este caso.
- Podemos obviar su implementación en el driver (no es recomendable, pero se puede), y de todos modos el kernel resuelve por default su tarea: genera una instancia de apertura y devuelve el file descriptor.
- Si no se usa se define como **`NULL`** en **`file_operations`** (como corresponde con puntero que no se usa), y el kernel abre el device pero no le notifica nada al driver.
- Trabaja con dos estructuras fundamentales para el manejo del File System.

✓ **`struct inode`**

Método open

```
int (*open) (struct inode *inode, struct file *filp);
```

- Como sabemos, es la primer operación realizada sobre el archivo, o device en este caso.
- Podemos obviar su implementación en el driver (no es recomendable, pero se puede), y de todos modos el kernel resuelve por default su tarea: genera una instancia de apertura y devuelve el file descriptor.
- Si no se usa se define como `NULL` en `file_operations` (como corresponde con puntero que no se usa), y el kernel abre el device pero no le notifica nada al driver.
- Trabaja con dos estructuras fundamentales para el manejo del File System.

✓ `struct inode`

✓ `struct file`

Estructura inode

Estructura inode

- Estructura utilizada por el kenel para representar los nodos del file system.

Estructura inode

- Estructura utilizada por el kenel para representar los nodos del file system.
- Contiene información del archivo relevante exclusivamente para el Sistema Operativo (como el tipo de archivo, es decir, pipe, char device, directorio, etc). Por ejemplo:

Estructura inode

- Estructura utilizada por el kenel para representar los nodos del file system.
- Contiene información del archivo relevante exclusivamente para el Sistema Operativo (como el tipo de archivo, es decir, pipe, char device, directorio, etc). Por ejemplo:
- `dev_t i_rdev` que contiene el device number.

Estructura inode

- Estructura utilizada por el kenel para representar los nodos del file system.
- Contiene información del archivo relevante exclusivamente para el Sistema Operativo (como el tipo de archivo, es decir, pipe, char device, directorio, etc). Por ejemplo:
- `dev_t i_rdev` que contiene el device number.
- `struct cdev * i_cdev` Estructura que representa un char device. Este campo apunta a una estructura `cdev` solo cuando el nodo representa un char device. De lo contrario es `NULL` .*Esta estructura es la que declaramos al instalar el módulo.*

Estructura inode

- Estructura utilizada por el kenel para representar los nodos del file system.
- Contiene información del archivo relevante exclusivamente para el Sistema Operativo (como el tipo de archivo, es decir, pipe, char device, directorio, etc). Por ejemplo:
- `dev_t i_rdev` que contiene el device number.
- `struct cdev * i_cdev` Estructura que representa un char device. Este campo apunta a una estructura `cdev` solo cuando el nodo representa un char device. De lo contrario es `NULL` .*Esta estructura es la que declaramos al instalar el módulo.*

Nota: Para obtener el MAJOR / MINOR de una estructura inode, están disponibles las siguientes funciones:

```
unsigned int iminor(struct inode *inode);
int imajor(struct inode *inode);
```

Estructura inode

```
struct inode {
    umode_t         i_mode;
    unsigned short  i_opflags;
    kuid_t          i_uid;
    kgid_t          i_gid;
    unsigned int    i_flags;

#ifdef CONFIG_FS_POSIX_ACL
    struct posix_acl  *i_acl;
    struct posix_acl  *i_default_acl;
#endif

    const struct inode_operations  *i_op;
    struct super_block             *i_sb;
    struct address_space           *i_mapping;

#ifdef CONFIG_SECURITY
    void  *i_security;
#endif
    /* Stat data, not accessed from path walking */
    unsigned long     i_ino;
    /*
     * Filesystems may only read i_nlink directly.  They shall use the
     * following functions for modification:
     *
     *    (set|clear|inc|drop)_nlink
     *    inode_(inc|dec)_link_count
     */
    union {
        const unsigned int     i_nlink;
        unsigned int           __i_nlink;
    };
```

Estructura inode

```
    dev_t                  i_rdev;
    loff_t                 i_size;  /* tamaño de archivo en bytes*/
    struct timespec64      i_atime;
    struct timespec64      i_mtime;
    struct timespec64      i_ctime;
    spinlock_t             i_lock; /* i_blocks, i_bytes, maybe i_size */
    unsigned short         i_bytes;
    u8                     i_blkbits;
    u8                     i_write_hint;
    blkcnt_t               i_blocks;

#ifdef __NEED_I_SIZE_ORDERED
    seqcount_t             i_size_seqcount;
#endif

    /* Misc */
    unsigned long          i_state;
    struct rw_semaphore    i_rwsem;

    unsigned long          dirtied_when;  /* jiffies of first dirtying */
    unsigned long          dirtied_time_when;

    struct hlist_node i_hash;
    struct list_head  i_io_list;  /* backing dev IO list */
#ifdef CONFIG_CGROUP_WRITEBACK
    struct bdi_writeback *i_wb;       /* the associated cgroup wb */

    /* foreign inode detection, see wbc_detach_inode() */
    int              i_wb_frn_winner;
    u16              i_wb_frn_avg_time;
    u16              i_wb_frn_history;
#endif
```

Estructura inode

```
    struct list_head   i_lru;       /* inode LRU list */
    struct list_head   i_sb_list;
    struct list_head   i_wb_list;   /* backing dev writeback list */
    union {
        struct hlist_head i_dentry;
        struct rcu_head      i_rcu;
    };
    atomic64_t      i_version;
    atomic64_t      i_sequence; /* see futex */
    atomic_t     i_count;
    atomic_t     i_dio_count;
    atomic_t     i_writecount;
#if defined(CONFIG_IMA) || defined(CONFIG_FILE_LOCKING)
    atomic_t     i_readcount; /* struct files open RO */
#endif
    union {
        const struct file_operations  *i_fop;  /* former ->i_op->default_file_ops */
        void (*free_inode)(struct inode *);
    };
     struct  file_lock_context   *i_flctx;
     struct  address_space        i_data;
     struct  list_head            i_devices;
     union {
          struct pipe_inode_info  *i_pipe;
          struct block_device     *i_bdev;
          struct cdev             *i_cdev;
          char                    *i_link;
          unsigned                 i_dir_seq;
    };

    __u32        i_generation;
```

Estructura inode

```
#ifdef CONFIG_FSNOTIFY
    __u32         i_fsnotify_mask; /* all events this inode cares about */
    struct fsnotify_mark_connector __rcu   *i_fsnotify_marks;
#endif

#ifdef CONFIG_FS_ENCRYPTION
    struct fscrypt_info  *i_crypt_info;
#endif

#ifdef CONFIG_FS_VERITY
    struct fsverity_info *i_verity_info;
#endif

    void          *i_private; /* fs or device private pointer */
} __randomize_layout;
```

Estructura File

Estructura File

- Es una estructura de mas alto nivel que `struct inode`, definida al igual que ésta en `linux/fs.h` que representa en el kernel una instancia de un archivo abierto. No obstante descansa sobre esta otra estructura ya que la tiene referenciada.

Estructura File

- Es una estructura de mas alto nivel que `struct inode` , definida al igual que ésta en `linux/fs.h` que representa en el kernel una instancia de un archivo abierto. No obstante descansa sobre esta otra estructura ya que la tiene referenciada.
- No es especifica del desarrollo de device drivers. Cada archivo abierto tiene en el kernel una instancia de esta estructura.

Estructura File

- Es una estructura de mas alto nivel que `struct inode` , definida al igual que ésta en `linux/fs.h` que representa en el kernel una instancia de un archivo abierto. No obstante descansa sobre esta otra estructura ya que la tiene referenciada.
- No es especifica del desarrollo de device drivers. Cada archivo abierto tiene en el kernel una instancia de esta estructura.
- La estructura se instancia cuando un proceso llama a `open ()` . Cuando se cierra el archivo, el kernel la libera.

Estructura File

- Es una estructura de mas alto nivel que `struct inode` , definida al igual que ésta en `linux/fs.h` que representa en el kernel una instancia de un archivo abierto. No obstante descansa sobre esta otra estructura ya que la tiene referenciada.
- No es especifica del desarrollo de device drivers. Cada archivo abierto tiene en el kernel una instancia de esta estructura.
- La estructura se instancia cuando un proceso llama a `open ()` . Cuando se cierra el archivo, el kernel la libera.
- La nomenclatura utilizada en el kernel para la instancia de la estructura es `filp` (file pointer).

Estructura File

- Es una estructura de mas alto nivel que `struct inode`, definida al igual que ésta en `linux/fs.h` que representa en el kernel una instancia de un archivo abierto. No obstante descansa sobre esta otra estructura ya que la tiene referenciada.
- No es especifica del desarrollo de device drivers. Cada archivo abierto tiene en el kernel una instancia de esta estructura.
- La estructura se instancia cuando un proceso llama a `open ()`. Cuando se cierra el archivo, el kernel la libera.
- La nomenclatura utilizada en el kernel para la instancia de la estructura es `filp` (file pointer).
- Pueden existir muchas instancias de la estructura file para representar un mismo nodo. (Múltiples llamadas a `open ()` para el mismo nodo.)

Estructura file

```
struct file {
   union {
      struct llist_node  fu_llist;
      struct rcu_head    fu_rcuhead;
   } f_u;
   struct path     f_path;
   struct inode    *f_inode;    /* cached value */
   const struct file_operations  *f_op;
/* Protects f_ep_links, f_flags.
 * Must not be taken from IRQ context.  */
   spinlock_t             f_lock;
   enum rw_hint           f_write_hint;
   atomic_long_t          f_count;
   unsigned int           f_flags;
   fmode_t                f_mode;
   struct mutex           f_pos_lock;
   loff_t                 f_pos;
   struct fown_struct     f_owner;
   const struct cred      *f_cred;
   struct file_ra_state   f_ra;
   u64          f_version;
#ifdef CONFIG_SECURITY
   void      *f_security;
#endif
   void      *private_data;    /* needed for tty driver, and maybe others */
#ifdef CONFIG_EPOLL /* Used by fs/eventpoll.c to link all the hooks to this file */
   struct list_head  f_ep_links;
   struct list_head  f_tfile_llink;
#endif /* #ifdef CONFIG_EPOLL */
   struct address_space *f_mapping;
   errseq_t     f_wb_err;
   errseq_t     f_sb_err; /* for syncfs */
} __randomize_layout  __attribute__((aligned(4)));
```

Estructura File

Estructura File

- Algunos campos importantes:

Estructura File

- Algunos campos importantes:
- `mode_t f_mode` Especifica si el archivo tiene permisos de lectura, escritura o ambos.

Estructura File

- Algunos campos importantes:
- `mode_t f_mode` Especifica si el archivo tiene permisos de lectura, escritura o ambos.
- `loff_t f_pos` Contiene la posición de lectura / escritura actual.

Estructura File

- Algunos campos importantes:
- `mode_t f_mode` Especifica si el archivo tiene permisos de lectura, escritura o ambos.
- `loff_t f_pos` Contiene la posición de lectura / escritura actual.
- `unsigned int f_flags`

Estructura File

- Algunos campos importantes:
- `mode_t f_mode` Especifica si el archivo tiene permisos de lectura, escritura o ambos.
- `loff_t f_pos` Contiene la posición de lectura / escritura actual.
- `unsigned int f_flags`
- `struct file_operations *f_op` Puntero a `file_operations` .

Método release

```
int (*release) (struct inode *inode, struct file *filp);
```

Método release

```
int (*release) (struct inode *inode, struct file *filp);
```

- Se invoca cuando se desea liberar la estructura.

Método release

```
int (*release) (struct inode *inode, struct file *filp);
```

- Se invoca cuando se desea liberar la estructura.
- Igual que `open ()` puede ser `NULL`.

Método release

```
int (*release) (struct inode *inode, struct file *filp);
```

- Se invoca cuando se desea liberar la estructura.
- Igual que `open ()` puede ser `NULL`.
- `release ()` no se invoca cada vez que un proceso llama a `close ()`. Si una estructura file se comparte (como resultado de `fork ()` o `dup ()`), `release ()` se invoca cuando todas las copias ejecutan `close ()`.

Método write

```
ssize_t (*write) (struct file *filp, const char __user *buf, size_t
count, loff_t *pos);
```

Método write

```
ssize_t (*write) (struct file *filp, const char __user *buf, size_t
count, loff_t *pos);
```

- Recibe un puntero a la **struct file** creada por el kernel cuando el proceso hizo **open ()**.

Método write

```
ssize_t (*write) (struct file *filp, const char __user *buf, size_t
count, loff_t *pos);
```

- Recibe un puntero a la `struct file` creada por el kernel cuando el proceso hizo `open ()`.
- Escribe a partir de la posición definida en `pos` del archivo o device definido en `filp` la cantidad de datos especificada por `count`.

Método write

```
ssize_t (*write) (struct file *filp, const char __user *buf, size_t
count, loff_t *pos);
```

- Recibe un puntero a la `struct file` creada por el kernel cuando el proceso hizo `open ()`.
- Escribe a partir de la posición definida en `pos` del archivo o device definido en `filp` la cantidad de datos especificada por `count`.
- Si este puntero a este método en `file_operations` es `NULL`, la system call `write ()` retorna `-EINVAL` al programa que la invoca.

Método write

```
ssize_t (*write) (struct file *filp, const char __user *buf, size_t
count, loff_t *pos);
```

- Recibe un puntero a la `struct file` creada por el kernel cuando el proceso hizo `open ()`.
- Escribe a partir de la posición definida en `pos` del archivo o device definido en `filp` la cantidad de datos especificada por `count`.
- Si este puntero a este método en `file_operations` es `NULL`, la system call `write ()` retorna `-EINVAL` al programa que la invoca.
- Un valor de retorno, no negativo, es el número de bytes escritos.

Método write: Procedimento general

Método write: Procedimento general

- El requerimiento viene desde el User Space y puede tener mal especificado el tamaño. Si el dispositivo exporta algún espacio de direcciones (eeprom, o I/O memory mapped, por ejemplo).

Método write: Procedimento general

- El requerimiento viene desde el User Space y puede tener mal especificado el tamaño. Si el dispositivo exporta algún espacio de direcciones (eeprom, o I/O memory mapped, por ejemplo).
- Ajusta `count` para evitar que un remanente de bytes se pase del límite de tamaño del file o device. Al igual que el paso aterior no es mandatorio y vale en las mismas condiciones.

Método write: Procedimento general

- El requerimiento viene desde el User Space y puede tener mal especificado el tamaño. Si el dispositivo exporta algún espacio de direcciones (eeprom, o I/O memory mapped, por ejemplo).
- Ajusta `count` para evitar que un remanente de bytes se pase del límite de tamaño del file o device. Al igual que el paso aterior no es mandatorio y vale en las mismas condiciones.
- Determinar la posición desde la que se debe comenzar a escribir. Relevante si el device escribe en un area de direccionamiento incremental.

Método write: Procedimento general

- El requerimiento viene desde el User Space y puede tener mal especificado el tamaño. Si el dispositivo exporta algún espacio de direcciones (eeprom, o I/O memory mapped, por ejemplo).
- Ajusta `count` para evitar que un remanente de bytes se pase del límite de tamaño del file o device. Al igual que el paso aterior no es mandatorio y vale en las mismas condiciones.
- Determinar la posición desde la que se debe comenzar a escribir. Relevante si el device escribe en un area de direccionamiento incremental.
- Copiar los datos desde el User Space a un buffer de Kernel Space y si falla retornar error.

Método write: Procedimento general

- El requerimiento viene desde el User Space y puede tener mal especificado el tamaño. Si el dispositivo exporta algún espacio de direcciones (eeprom, o I/O memory mapped, por ejemplo).
- Ajusta `count` para evitar que un remanente de bytes se pase del límite de tamaño del file o device. Al igual que el paso aterior no es mandatorio y vale en las mismas condiciones.
- Determinar la posición desde la que se debe comenzar a escribir. Relevante si el device escribe en un area de direccionamiento incremental.
- Copiar los datos desde el User Space a un buffer de Kernel Space y si falla retornar error.
- Escribir los datos en el device y si falla retornar error

Método write: Procedimento general

- El requerimiento viene desde el User Space y puede tener mal especificado el tamaño. Si el dispositivo exporta algún espacio de direcciones (eeprom, o I/O memory mapped, por ejemplo).
- Ajusta `count` para evitar que un remanente de bytes se pase del límite de tamaño del file o device. Al igual que el paso aterior no es mandatorio y vale en las mismas condiciones.
- Determinar la posición desde la que se debe comenzar a escribir. Relevante si el device escribe en un area de direccionamiento incremental.
- Copiar los datos desde el User Space a un buffer de Kernel Space y si falla retornar error.
- Escribir los datos en el device y si falla retornar error
- Actualizar el contador de posición del archivo de acuerdo con la cantidad de bytes leídos, y retornar el número de bytes copiados.

Método write: Procedimento general

```
if ( *pos >= *inode->i_size ) return -EINVAL;
if (*pos + count > *inode->i_size) /* Idem */
    count = inode->i_size - (*pos); /* Ajusta cantidad al remanente del file*/
void *from = pos_to_address (*pos); /* convierte pos en una address válida*/
if ( copy_from_user(dev->buffer, buf, count) ) /*Devuelve 0 si no hubo error*/
    return -EFAULT;
write_error = device_write(dev->buffer, count);
if ( write_error )
    return -EFAULT;
*pos += count; /* Actualiza el puntero interno del file.*/
return count;  /* Retorna cantidad de bytes leídos.*/
```

Método write: Procedimento general

```
if ( *pos >= *inode->i_size ) return -EINVAL;
if (*pos + count > *inode->i_size) /* Idem */
    count = inode->i_size - (*pos); /* Ajusta cantidad al remanente del file*/
void *from = pos_to_address (*pos); /* convierte pos en una address válida*/
if ( copy_from_user(dev->buffer, buf, count) ) /*Devuelve 0 si no hubo error*/
    return -EFAULT;
write_error = device_write(dev->buffer, count);
if ( write_error )
    return -EFAULT;
*pos += count; /* Actualiza el puntero interno del file.*/
return count;  /* Retorna cantidad de bytes leídos.*/
```

Los pasos descriptos tienen caracter general. Cuando desarrollamos un device driver determinado debemos analizar sus características particulares y aplicar cada paso solo si corresponde y en función de sus datos particulares.

Método read

```
ssize_t (*read) (struct file *filp, char __user *buf, size_t count,
loff_t * pos);
```

Método read

```
ssize_t (*read) (struct file *filp, char __user *buf, size_t count,
loff_t * pos);
```

- Recibe un puntero a la `struct file` creada por el kernel cuando el proceso hizo `open ()`.

Método read

```
ssize_t (*read) (struct file *filp, char __user *buf, size_t count,
loff_t * pos);
```

- Recibe un puntero a la `struct file` creada por el kernel cuando el proceso hizo `open ()`.
- Lee datos desde un archivo o device, y los almacena en la dirección del User Space apuntada por `buf`

Método read

```
ssize_t (*read) (struct file *filp, char __user *buf, size_t count,
loff_t * pos);
```

- Recibe un puntero a la **struct file** creada por el kernel cuando el proceso hizo **open ()**.
- Lee datos desde un archivo o device, y los almacena en la dirección del User Space apuntada por **buf**
- **count** le indica la cantidad de bytes a copiar en **buf**.

Método read

```
ssize_t (*read) (struct file *filp, char __user *buf, size_t count,
loff_t * pos);
```

- Recibe un puntero a la `struct file` creada por el kernel cuando el proceso hizo `open ()`.
- Lee datos desde un archivo o device, y los almacena en la dirección del User Space apuntada por `buf`
- `count` le indica la cantidad de bytes a copiar en `buf`.
- `pos` es el valor actual del puntero interno del stream. Si se actualiza deberá guardarlo en la estructura file antes de regresar

Método read

```
ssize_t (*read) (struct file *filp, char __user *buf, size_t count,
loff_t * pos);
```

- Recibe un puntero a la **struct file** creada por el kernel cuando el proceso hizo **open ()** .
- Lee datos desde un archivo o device, y los almacena en la dirección del User Space apuntada por **buf**
- **count** le indica la cantidad de bytes a copiar en **buf** .
- **pos** es el valor actual del puntero interno del stream. Si se actualiza deberá guardarlo en la estructura file antes de regresar
- Un puntero **NULL** en esta posición de **file_operations** hace que la system call **read ()** sobre este device devuelva - **EINVAL** (“Invalid argument”).

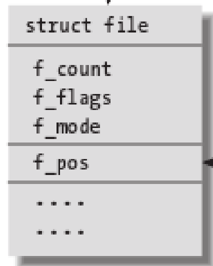
Método read

```
ssize_t (*read) (struct file *filp, char __user *buf, size_t count,
loff_t * pos);
```

- Recibe un puntero a la `struct file` creada por el kernel cuando el proceso hizo `open ()`.
- Lee datos desde un archivo o device, y los almacena en la dirección del User Space apuntada por `buf`
- `count` le indica la cantidad de bytes a copiar en `buf`.
- `pos` es el valor actual del puntero interno del stream. Si se actualiza deberá guardarlo en la estructura file antes de regresar
- Un puntero `NULL` en esta posición de `file_operations` hace que la system call `read ()` sobre este device devuelva -`EINVAL` ("Invalid argument").
- Un valor de retorno no negativo representa el número de bytes leídos, de modo que es lo que debemos retornar desde el nuestra función del driver.

Relación de read() con los recursos del kernel

```
ssize_t dev_read(struct file *file, char *buf, size_t count, loff_t *ppos);
```



Kernel Space
(nonswappable)

User Space
(swappable)

Método read: Procedimento general

Método read: Procedimento general

- Prevenir lecturas mas allá del tamaño del archivo. En tal caso retornar end-of-file:

Método read: Procedimento general

- Prevenir lecturas mas allá del tamaño del archivo. En tal caso retornar end-of-file:
- El número de bytes a retornar en `buf`, no puede ir mas allá del final del archivo. Debe detectar esta situación y ajustar el valor del argumento `count`.

Método read: Procedimento general

- Prevenir lecturas mas allá del tamaño del archivo. En tal caso retornar end-of-file:
- El número de bytes a retornar en `buf` , no puede ir mas allá del final del archivo. Debe detectar esta situación y ajustar el valor del argumento `count` .
- Determinar la posición desde la que se debe comenzar a leer.

Método read: Procedimento general

- Prevenir lecturas mas allá del tamaño del archivo. En tal caso retornar end-of-file:
- El número de bytes a retornar en `buf` , no puede ir mas allá del final del archivo. Debe detectar esta situación y ajustar el valor del argumento `count` .
- Determinar la posición desde la que se debe comenzar a leer.
- Copiar los datos al buffer del User Space y si falla retornar error.

Método read: Procedimento general

- Prevenir lecturas mas allá del tamaño del archivo. En tal caso retornar end-of-file:
- El número de bytes a retornar en `buf` , no puede ir mas allá del final del archivo. Debe detectar esta situación y ajustar el valor del argumento `count` .
- Determinar la posición desde la que se debe comenzar a leer.
- Copiar los datos al buffer del User Space y si falla retornar error.
- Actualizar el contador de posición del archivo de acuerdo con la cantidad de bytes leídos, y retornar el número de bytes copiados.

Método read: Procedimento general

- Prevenir lecturas mas allá del tamaño del archivo. En tal caso retornar end-of-file:
- El número de bytes a retornar en **buf** , no puede ir mas allá del final del archivo. Debe detectar esta situación y ajustar el valor del argumento **count** .
- Determinar la posición desde la que se debe comenzar a leer.
- Copiar los datos al buffer del User Space y si falla retornar error.
- Actualizar el contador de posición del archivo de acuerdo con la cantidad de bytes leídos, y retornar el número de bytes copiados.

```
 if (*pos >= *inode->i_size) /*Tamaño es i_size de inode. Si es I/O ignorar */
     return 0; /* Retorna EOF en realidad*/
 if (*pos + count > *inode->i_size) /* Idem */
     count = inode->i_size - (*pos); /* Ajusta cantidad al remanente del file*/
void *from = pos_to_address (*pos); /* convierte pos en una address válida*/
 if ( copy_to_user(buf, from, count) ) /*Devuelve 0 si no hubo error*/
     return -EFAULT;
*pos += count; /* Actualiza el puntero interno del file.*/
return count;  /* Retorna cantidad de bytes leídos.*/
```

Método ioctl

```
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```

Método ioctl

```
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```

- La system call **ioctl ()** envía comandos device específicos: **reset** , **shutdown** , **configure** , por ejemplo.

Método ioctl

```
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
```

- La system call `ioctl ()` envía comandos device específicos: `reset` , `shutdown` , `configure` , por ejemplo.
- El kernel generalmente procesa `ioctl ()` por medio del método definido en `file_operations` .

Método ioctl

```
int (*ioctl) (struct inode *, struct file *, unsigned int , unsigned
long);
```

- La system call `ioctl ()` envía comandos device específicos: `reset` , `shutdown` , `configure` , por ejemplo.
- El kernel generalmente procesa `ioctl ()` por medio del método definido en `file_operations` .
- Si no hay un método `ioctl ()` , la system call retorna error para cualquier requerimiento no predefinido (`-ENOTTY` , “No such ioctl for device”).

Método ioctl

```
int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned
long);
```

- La system call `ioctl ()` envía comandos device específicos: `reset`, `shutdown`, `configure`, por ejemplo.
- El kernel generalmente procesa `ioctl ()` por medio del método definido en `file_operations`.
- Si no hay un método `ioctl ()`, la system call retorna error para cualquier requerimiento no predefinido (`-ENOTTY`, “No such ioctl for device”).
- La implementación de `ioctl ()` es device específica.

Manejo de memoria

Manejo de memoria

- A pesar de tratarse de software que ejecuta en Modo Kernel, cada vez que se necesita memoria hay que solicitarla al memory manager.

Manejo de memoria

- A pesar de tratarse de software que ejecuta en Modo Kernel, cada vez que se necesita memoria hay que solicitarla al memory manager.
- La función que usamos en modo kernel es `kmalloc` para solicitarla y `kfree` para devolverla

Manejo de memoria

- A pesar de tratarse de software que ejecuta en Modo Kernel, cada vez que se necesita memoria hay que solicitarla al memory manager.
- La función que usamos en modo kernel es **`kmalloc`** para solicitarla y **`kfree`** para devolverla

```
void * kmalloc(size t_size, int flags);
void kfree ( const void * objp);
```

Manejo de memoria

- A pesar de tratarse de software que ejecuta en Modo Kernel, cada vez que se necesita memoria hay que solicitarla al memory manager.
- La función que usamos en modo kernel es **kmalloc** para solicitarla y **kfree** para devolverla

```
void * kmalloc(size t_size, int flags);
void kfree ( const void * objp);
```

- **kmalloc** devuelve un puntero a memoria de Kernel Space

Manejo de memoria

- A pesar de tratarse de software que ejecuta en Modo Kernel, cada vez que se necesita memoria hay que solicitarla al memory manager.
- La función que usamos en modo kernel es **kmalloc** para solicitarla y **kfree** para devolverla

```
void * kmalloc(size t_size, int flags);
void kfree ( const void * objp);
```

- **kmalloc** devuelve un puntero a memoria de Kernel Space
- El argumento **flags** puede valer **GFP_KERNEL**, para entornos en donde no se requiere ejecución atómica (puede ponerse en estado sleep el proceso invocante), o **GFP_ATOMIC** cuando requiere ese tipo de ejecución (interrupt handlers).

Manejo de memoria

- Si necesitamos grandes cantidades de memoria, se dispone de funciones para obtener páginas completas.

Manejo de memoria

- Si necesitamos grandes cantidades de memoria, se dispone de funciones para obtener páginas completas.

```
unsigned long get_zeroed_page(unsigned int flags);
unsigned long __get_free_page(unsigned int flags);
void free_page(unsigned long addr);
unsigned long __get_free_page(unsigned int flags, unsigned int order);
void free_pages(unsigned long addr, unsigned long order);
```

Manejo de memoria

- Si necesitamos grandes cantidades de memoria, se dispone de funciones para obtener páginas completas.

```
unsigned long get_zeroed_page(unsigned int flags);
unsigned long __get_free_page(unsigned int flags);
void free_page(unsigned long addr);
unsigned long __get_free_page(unsigned int flags, unsigned int order);
void free_pages(unsigned long addr, unsigned long order);
```

- **`get_zeroed_page`**, es como **`__get_free_page`**, solo que además devuelve la página inicializada con '0's.

Manejo de memoria

- Si necesitamos grandes cantidades de memoria, se dispone de funciones para obtener páginas completas.

```
unsigned long get_zeroed_page(unsigned int flags);
unsigned long __get_free_page(unsigned int flags);
void free_page(unsigned long addr);
unsigned long __get_free_page(unsigned int flags, unsigned int order);
void free_pages(unsigned long addr, unsigned long order);
```

- **`get_zeroed_page`**, es como **`__get_free_page`**, solo que además devuelve la página inicializada con '0's.
- **`order`** es $log_2 n$ siendo n la cantidad de páginas que queremos allocar. Si no hay tal cantidad de páginas contiguas la función falla.

Copia de memoria

- Las mencionamos en los métodos **`read`** y **`write`**.

Copia de memoria

- Las mencionamos en los métodos **read** y **write**.

```
unsigned long copy_to_user(void __user *to, const void *from,
unsigned long count);
unsigned long copy_from_user(void *to, const void __user *from,
unsigned long count);
```

Copia de memoria

- Las mencionamos en los métodos **read** y **write**.

```
unsigned long copy_to_user(void __user *to, const void *from,
unsigned long count);
unsigned long copy_from_user(void *to, const void __user *from,
unsigned long count);
```

- Son como **memcpy** solo que el argumento origen está en kernel space y el destino en user space, y visceversa, respectivamente.

Copia de memoria

- Las mencionamos en los métodos **read** y **write**.

```
unsigned long copy_to_user(void __user *to, const void *from,
unsigned long count);
unsigned long copy_from_user(void *to, const void __user *from,
unsigned long count);
```

- Son como **memcpy** solo que el argumento origen está en kernel space y el destino en user space, y visceversa, respectivamente.
- Chequean que el puntero de memoria de cada argumento esté en el espacio correcto.

Copia de memoria

- Las mencionamos en los métodos **read** y **write**.

```
unsigned long copy_to_user(void __user *to, const void *from,
unsigned long count);
unsigned long copy_from_user(void *to, const void __user *from,
unsigned long count);
```

- Son como **memcpy** solo que el argumento origen está en kernel space y el destino en user space, y visceversa, respectivamente.
- Chequean que el puntero de memoria de cada argumento esté en el espacio correcto.
- Las versiones de estas funciones que comienzan con __ no realizan este chequeo.

Verificando punteros

Verificando punteros

- En un kernel module, utilizar un puntero inválido a un área de memoria, constituye un error grosero.

Verificando punteros

- En un kernel module, utilizar un puntero inválido a un área de memoria, constituye un error grosero.

- Es de esperar que los punteros a memoria del kernel space estén correctamente inicializados. ¿Hace falta justificar esta afirmación?

Verificando punteros

- En un kernel module, utilizar un puntero inválido a un área de memoria, constituye un error grosero.
- Es de esperar que los punteros a memoria del kernel space estén correctamente inicializados. ¿Hace falta justificar esta afirmación?
- ¿Podemos tener la misma seguridad con respecto a un puntero recibido desde la aplicación invocante? (User Space)

Verificando punteros

- En un kernel module, utilizar un puntero inválido a un área de memoria, constituye un error grosero.
- Es de esperar que los punteros a memoria del kernel space estén correctamente inicializados. ¿Hace falta justificar esta afirmación?
- ¿Podemos tener la misma seguridad con respecto a un puntero recibido desde la aplicación invocante? (User Space)
- Necesitamos un chequeo de integridad para los punteros del User Space.

Verificando punteros

Verificando punteros

```
access_ok (type, addr, size);
```

Verificando punteros

```
access_ok (type, addr, size);
```

- `type` Puede ser `VERIFY_READ`, o `VERIFY_WRITE`. Especifica el tipo de acceso del bloque de memoria:

Verificando punteros

```
access_ok (type, addr, size);
```

- **type** Puede ser **VERIFY_READ** , o **VERIFY_WRITE** . Especifica el tipo de acceso del bloque de memoria:
- **addr** Puntero a verificar

Verificando punteros

```
access_ok (type, addr, size);
```

- **type** Puede ser **VERIFY_READ**, o **VERIFY_WRITE**. Especifica el tipo de acceso del bloque de memoria:
- **addr** Puntero a verificar
- **size** Tamaño del bloque apuntado

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. **Cuestiones adicionales asociadas al dispositivo**
   - **Platform drivers**
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

¿Que es un platform driver?

¿Que es un platform driver?

- En sistemas embebidos los dispositivos no están usualmente conectados a buses con capacidad de enumeración, sino que están conectados dentro del SoC.

¿Que es un platform driver?

- En sistemas embebidos los dispositivos no están usualmente conectados a buses con capacidad de enumeración, sino que están conectados dentro del SoC.
- No pueden desconectarse, y el Sistema Operativo debe recoconocerlos y determinar sus capacidades antes de utilizarlos .

¿Que es un platform driver?

- En sistemas embebidos los dispositivos no están usualmente conectados a buses con capacidad de enumeración, sino que están conectados dentro del SoC.
- No pueden desconectarse, y el Sistema Operativo debe recoconocerlos y determinar sus capacidades antes de utilizarlos .
- Los bus tales como I2C, SPI, USB, I2S, SATA, (y hasta PCI en ocasiones) forman habitualmente parte del SoC, y son dispositivos de hardware. Se los denomina controladores.

¿Que es un platform driver?

- En sistemas embebidos los dispositivos no están usualmente conectados a buses con capacidad de enumeración, sino que están conectados dentro del SoC.
- No pueden desconectarse, y el Sistema Operativo debe recoconocerlos y determinar sus capacidades antes de utilizarlos .
- Los bus tales como I2C, SPI, USB, I2S, SATA, (y hasta PCI en ocasiones) forman habitualmente parte del SoC, y son dispositivos de hardware. Se los denomina controladores.
- Por estar embebidos en el SoC, no pueden ser removidos, y no son “auto-discovered”. Por tal motivo se denominan platform devices.

¿Que es un platform driver?

- En sistemas embebidos los dispositivos no están usualmente conectados a buses con capacidad de enumeración, sino que están conectados dentro del SoC.
- No pueden desconectarse, y el Sistema Operativo debe recoconocerlos y determinar sus capacidades antes de utilizarlos .
- Los bus tales como I2C, SPI, USB, I2S, SATA, (y hasta PCI en ocasiones) forman habitualmente parte del SoC, y son dispositivos de hardware. Se los denomina controladores.
- Por estar embebidos en el SoC, no pueden ser removidos, y no son “auto-discovered”. Por tal motivo se denominan platform devices.
- Desde el punto de vista del kernel estos dispositivos son el root de un bus, es decir una plataforma para conectar otros dispositivos, internos o externos al SoC.

¿Que es un platform driver?

- En sistemas embebidos los dispositivos no están usualmente conectados a buses con capacidad de enumeración, sino que están conectados dentro del SoC.
- No pueden desconectarse, y el Sistema Operativo debe recoconocerlos y determinar sus capacidades antes de utilizarlos .
- Los bus tales como I2C, SPI, USB, I2S, SATA, (y hasta PCI en ocasiones) forman habitualmente parte del SoC, y son dispositivos de hardware. Se los denomina controladores.
- Por estar embebidos en el SoC, no pueden ser removidos, y no son “auto-discovered”. Por tal motivo se denominan platform devices.
- Desde el punto de vista del kernel estos dispositivos son el root de un bus, es decir una plataforma para conectar otros dispositivos, internos o externos al SoC.
- Lo primero que se requiere es un driver de plataforma con un nombre fijo el cual manejará los dispositivos que se conecten al bus.

Platform drivers

Platform drivers

- Para estos dispositivos existe este tipo de driver que permite describir el dispositivo de manera estática en el Device Tree.

Platform drivers

- Para estos dispositivos existe este tipo de driver que permite describir el dispositivo de manera estática en el Device Tree.
- La estructura `platform_driver` junto con su función de registro, llamada `platform_driver_register`, son el eje.

Platform drivers

- Para estos dispositivos existe este tipo de driver que permite describir el dispositivo de manera estática en el Device Tree.
- La estructura **platform_driver** junto con su función de registro, llamada **platform_driver_register** , son el eje.

```
static struct platform_driver mypdrv = {
    .probe  = my_pdrv_probe,
    .remove = my_pdrv_remove,
    .driver = {
        .name   = "my_platform_driver",
        .owner  = THIS_MODULE,
    },
};
```

Platform drivers

- Para estos dispositivos existe este tipo de driver que permite describir el dispositivo de manera estática en el Device Tree.
- La estructura `platform_driver` junto con su función de registro, llamada `platform_driver_register`, son el eje.

```
static struct platform_driver mypdrv = {
    .probe  = my_pdrv_probe,
    .remove = my_pdrv_remove,
    .driver = {
        .name   = "my_platform_driver",
        .owner  = THIS_MODULE,
    },
};
```

- `probe()` Es la función que se ejecuta cuando luego de ser reconocido un dispositivo conectado a la plataforma requiere su driver. Su prototipo generico es `static int my_pdrv_probe(struct platform_device *pdev)`

Platform drivers

- `remove()` Es la función que se invoca cuando no se necesita mas el dispositivo

  ```
  static int my_pdrv_remove(struct platform_device *pdev)
  ```

Platform drivers

- `remove()` Es la función que se invoca cuando no se necesita mas el dispositivo `static int my_pdrv_remove(struct platform_device *pdev)`
- `struct device_driver` Desribe al driver en sí mismo (nombre, owner, entre otros atributos).

Platform drivers

- `remove()` Es la función que se invoca cuando no se necesita mas el dispositivo
  `static int my_pdrv_remove(struct platform_device *pdev)`
- `struct device_driver` Desribe al driver en sí mismo (nombre, owner, entre otros atributos).
- Luego de setear adecuadamente los miembros de esta estructura hay que registrarla en el kernel. Para ello hay dos posibilidades

Platform drivers

- `remove()` Es la función que se invoca cuando no se necesita mas el dispositivo
  `static int my_pdrv_remove(struct platform_device *pdev)`
- `struct device_driver` Desribe al driver en sí mismo (nombre, owner, entre otros atributos).
- Luego de setear adecuadamente los miembros de esta estructura hay que registrarla en el kernel. Para ello hay dos posibilidades
- `platform_driver_register()` Inserta al driver en la lista de drivers mantenida por el kernel, y se invoca `probe()` cada vez que se detecta un dispositivo que coincide con este driver.

Platform drivers

- `remove()` Es la función que se invoca cuando no se necesita mas el dispositivo `static int my_pdrv_remove(struct platform_device *pdev)`
- `struct device_driver` Desribe al driver en sí mismo (nombre, owner, entre otros atributos).
- Luego de setear adecuadamente los miembros de esta estructura hay que registrarla en el kernel. Para ello hay dos posibilidades
- `platform_driver_register()` Inserta al driver en la lista de drivers mantenida por el kernel, y se invoca `probe()` cada vez que se detecta un dispositivo que coincide con este driver.
- `platform_driver_probe()` No registra al driver en el sistema. Corre un loop de búsqueda. Llama a `probe()` si encuentra una coincidencia, sino ignora al driver. Remueve de la memoria la sección `__init` del driver, bajando el overhead de memoria, e impidiendo posteriores llamadas a `probe()`.

```
ret = platform_driver_probe(&mypdrv, my_pdrv_probe);
```

Platform drivers

- Usamos `platform_driver_probe()` cuando estamos 100 % seguros que el device esta presente en el sistema.

Platform drivers

- Usamos **platform_driver_probe()** cuando estamos 100 % seguros que el device esta presente en el sistema.

```
1  #include <linux/module.h>
2  #include <linux/kernel.h>
3  #include <linux/init.h>
4  #include <linux/platform_device.h>
5
6  static int my_pdrv_probe (struct \
7      platform_device *pdev){
8      pr_info("Hola! device probed!\n");
9      return 0;
10 }
11 static void my_pdrv_remove(struct \
12     platform_device *pdev){
13     pr_info("Hasta la vista baby!\n");
14 }
15 static struct platform_driver mypdrv = {
16     .probe   = my_pdrv_probe,
17     .remove  = my_pdrv_remove,
18     .driver  = {
19         .name    = "my_platform_driver",
20         .owner   = THIS_MODULE,
21     },
22 };
23 static int __init my_pdrv_init(void) x{
24     pr_info("Hola Mundo!\n");
25     platform_driver_register(&mypdrv);
26     return 0;
27 }
28 static void __exit my_pdrv_remove (void) {
29     pr_info("Hasta luego!\n");
30     platform_driver_unregister(&mypdriver);
31 }
32 module_init(my_pdrv_init);
33 module_exit(my_pdrv_remove);
```

Platform drivers

Si como en este caso solo vamos a registrar y desregistrar el módulo con `module_init` y `module_exit` , podemos reemplazarlas por la macro `module_platform_driver` para registrar nuestro driver en la plataforma core

Platform drivers

Si como en este caso solo vamos a registrar y desregistrar el módulo con **module_init** y **module_exit** , podemos reemplazarlas por la macro **module_platform_driver** para registrar nuestro driver en la plataforma core

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/platform_device.h>
#define module_platform_driver(
    __platform_driver) \
module_driver(__platform_driver,
    platform_driver_register, \
platform_driver_unregister)

static int my_pdrv_probe (struct
    platform_device *pdev){
    pr_info("Hola! device probed!\n");
    return 0;
}
static void my_pdrv_remove(struct
    platform_device *pdev){
    pr_info("Hasta la vista baby!\n");
}
static struct platform_driver mypdrv = {
    .probe  = my_pdrv_probe,
    .remove = my_pdrv_remove,
    .driver = {
        .name   = "my_platform_driver",
        .owner  = THIS_MODULE,
    },
};
module_platform_driver(my_driver);
```

Platform devices

Platform devices

- Una vez implementado el driver, es necesario indicarle al kernel cuales son los dispositivos que necesitan ese driver

Platform devices

- Una vez implementado el driver, es necesario indicarle al kernel cuales son los dispositivos que necesitan ese driver
- Un dispositivo de plataforma se representa en el kernel mediante la estructura `struct platform_device`

Platform devices

- Una vez implementado el driver, es necesario indicarle al kernel cuales son los dispositivos que necesitan ese driver
- Un dispositivo de plataforma se representa en el kernel mediante la estructura **struct platform_device**

```
struct platform_device
{
    const char *name;
    u32 id;
    struct device dev;
    u32 num_resources;
    struct resource *resource;
};
```

Platform devices

- Una vez implementado el driver, es necesario indicarle al kernel cuales son los dispositivos que necesitan ese driver
- Un dispositivo de plataforma se representa en el kernel mediante la estructura **struct platform_device**

```
struct platform_device
{
    const char *name;
    u32 id;
    struct device dev;
    u32 num_resources;
    struct resource *resource;
};
```

- Esta estructura está relacionada con la estructura **platform_driver** mostrada anteriormente.

Platform devices

- Una vez implementado el driver, es necesario indicarle al kernel cuales son los dispositivos que necesitan ese driver
- Un dispositivo de plataforma se representa en el kernel mediante la estructura **struct platform_device**

```
struct platform_device
{
    const char *name;
    u32 id;
    struct device dev;
    u32 num_resources;
    struct resource *resource;
};
```

```
static struct platform_driver mypdrv = {
    .probe  = my_pdrv_probe,
    .remove = my_pdrv_remove,
    .driver = {
        .name   = "my_platform_driver",
        .owner  = THIS_MODULE,
    },
};
```

- Esta estructura está relacionada con la estructura **platform_driver** mostrada anteriormente.

Platform devices

- Una vez implementado el driver, es necesario indicarle al kernel cuales son los dispositivos que necesitan ese driver
- Un dispositivo de plataforma se representa en el kernel mediante la estructura **struct platform_device**

```
struct platform_device
{
    const char *name;
    u32 id;
    struct device dev;
    u32 num_resources;
    struct resource *resource;
};
```

```
static struct platform_driver mypdrv = {
    .probe   = my_pdrv_probe,
    .remove  = my_pdrv_remove,
    .driver  = {
        .name    = "my_platform_driver",
        .owner   = THIS_MODULE,
    },
};
```

- Esta estructura está relacionada con la estructura **platform_driver** mostrada anteriormente.
- La clave para que el sistema operativo encuentre los dispositivos correspondientes al driver es necesario que ambas estructuras tengan el mismo nombre en **.name**.

Provisión de Dispositivos

Provisión de Dispositivos

Cuando los dispositivos no son hot-plugable, el kernel no tiene forma de conocer sus capacidades, ni que necesita cada device para trabajar apropiadamente.

Por lo tanto es necesario proveerle al kernel la información de los recursos que necesita cada dispositivo: IRQs, canales de DMA, espacio de direccionamiento, direcciones de E/S, Buses, o lo que corresponda.

Además debe informársele acerca de que recursos debe proveerse al dispositivo durante su operación: estructuras de datos privadas que eventualmente pueda necesitar.

Provisión de Dispositivos

- El método de provisión de esta información es el Device Tree.

Provisión de Dispositivos

- El método de provisión de esta información es el Device Tree.
- El Device Tree es un archivo de Descripción de Hardware cuyo formato es similar a una estructura de árbol, en la que cada dispositivo se corresponde con un nodo, y sus datos, recursos, y configuraciones se representan como propiedades del nodo.

Provisión de Dispositivos

- El método de provisión de esta información es el Device Tree.
- El Device Tree es un archivo de Descripción de Hardware cuyo formato es similar a una estructura de árbol, en la que cada dispositivo se corresponde con un nodo, y sus datos, recursos, y configuraciones se representan como propiedades del nodo.
- De este modo al hacer una modificación en el hardware solo necesitamos modificar el Device Tree en el archivo DTS (Device Tree Source).

Provisión de Dispositivos

- El método de provisión de esta información es el Device Tree.
- El Device Tree es un archivo de Descripción de Hardware cuyo formato es similar a una estructura de árbol, en la que cada dispositivo se corresponde con un nodo, y sus datos, recursos, y configuraciones se representan como propiedades del nodo.
- De este modo al hacer una modificación en el hardware solo necesitamos modificar el Device Tree en el archivo DTS (Device Tree Source).
- Nos proponemos explicar como se enlaza desde los fuentes de un módulo de kernel que implementa un platform device driver con el Device Tree.

Provisión de Dispositivos

- El método de provisión de esta información es el Device Tree.
- El Device Tree es un archivo de Descripción de Hardware cuyo formato es similar a una estructura de árbol, en la que cada dispositivo se corresponde con un nodo, y sus datos, recursos, y configuraciones se representan como propiedades del nodo.
- De este modo al hacer una modificación en el hardware solo necesitamos modificar el Device Tree en el archivo DTS (Device Tree Source).
- Nos proponemos explicar como se enlaza desde los fuentes de un módulo de kernel que implementa un platform device driver con el Device Tree.
- Antes de hacerlo es necesario es necesario describir el proceso de Provisión de devices.

Device Drivers y Bus Matching

- Los platform devices matchean con sus dispositivos mediante comparación de cadenas de texto almacenadas en miembros de las diferentes estructuras que describen el hardware.

Device Drivers y Bus Matching

- Los platform devices matchean con sus dispositivos mediante comparación de cadenas de texto almacenadas en miembros de las diferentes estructuras que describen el hardware.
- No obstante, previamente a cualquier matching, el S.O. ejecuta la función:

```
platform_match (struct device *dev, struct device_driver *drv).
```

Device Drivers y Bus Matching

- Los platform devices matchean con sus dispositivos mediante comparación de cadenas de texto almacenadas en miembros de las diferentes estructuras que describen el hardware.
- No obstante, previamente a cualquier matching, el S.O. ejecuta la función:

```
platform_match (struct device *dev, struct device_driver *drv).
```

- De acuerdo con el Linux Drivers Model el elemento Bus es uno de los puntos mas importantes. Cada controlador de bus debe mantener la lista de drivers y la de los dispositivos conectados a él.

Device Drivers y Bus Matching

- Los platform devices matchean con sus dispositivos mediante comparación de cadenas de texto almacenadas en miembros de las diferentes estructuras que describen el hardware.
- No obstante, previamente a cualquier matching, el S.O. ejecuta la función:

```
platform_match (struct device *dev, struct device_driver *drv).
```

- De acuerdo con el Linux Drivers Model el elemento Bus es uno de los puntos mas importantes. Cada controlador de bus debe mantener la lista de drivers y la de los dispositivos conectados a él.
- Cada Driver de Bus se responsabiliza de matchear devices y drivers.

Device Drivers y Bus Matching

- Los platform devices matchean con sus dispositivos mediante comparación de cadenas de texto almacenadas en miembros de las diferentes estructuras que describen el hardware.
- No obstante, previamente a cualquier matching, el S.O. ejecuta la función:

```
platform_match (struct device *dev, struct device_driver *drv).
```

- De acuerdo con el Linux Drivers Model el elemento Bus es uno de los puntos mas importantes. Cada controlador de bus debe mantener la lista de drivers y la de los dispositivos conectados a él.
- Cada Driver de Bus se responsabiliza de matchear devices y drivers.
- Cada vez que conectamos un device o agregamos un driver el Bus driver inicia un procedimiento denominado matching loop.

Provisión de Dispositivos

- Cuando registramos un nuevo dispositivo en un bus utilizando las funciones provistas en el core del bus, el kernel invoca a la función match del core que se registró con el driver del bus.

Provisión de Dispositivos

- Cuando registramos un nuevo dispositivo en un bus utilizando las funciones provistas en el core del bus, el kernel invoca a la función match del core que se registró con el driver del bus.
- Es decir dispara el matching loop.

Provisión de Dispositivos

- Cuando registramos un nuevo dispositivo en un bus utilizando las funciones provistas en el core del bus, el kernel invoca a la función match del core que se registró con el driver del bus.
- Es decir dispara el matching loop.
- Esta función verifica si hay registrado en el bus algún driver que coincida con el dispositivo registrado.

Provisión de Dispositivos

- Cuando registramos un nuevo dispositivo en un bus utilizando las funciones provistas en el core del bus, el kernel invoca a la función match del core que se registró con el driver del bus.
- Es decir dispara el matching loop.
- Esta función verifica si hay registrado en el bus algún driver que coincida con el dispositivo registrado.
- Si no encuentra coincidencia, no hace nada.

Provisión de Dispositivos

- Cuando registramos un nuevo dispositivo en un bus utilizando las funciones provistas en el core del bus, el kernel invoca a la función match del core que se registró con el driver del bus.
- Es decir dispara el matching loop.
- Esta función verifica si hay registrado en el bus algún driver que coincida con el dispositivo registrado.
- Si no encuentra coincidencia, no hace nada.
- Si la encuentra, el kernel la notifica al device manager (**`udev`** / **`mdev`**), mediante un mecanismo de notificación denominado *netlink socket*

Provisión de Dispositivos

- Cuando registramos un nuevo dispositivo en un bus utilizando las funciones provistas en el core del bus, el kernel invoca a la función match del core que se registró con el driver del bus.
- Es decir dispara el matching loop.
- Esta función verifica si hay registrado en el bus algún driver que coincida con el dispositivo registrado.
- Si no encuentra coincidencia, no hace nada.
- Si la encuentra, el kernel la notifica al device manager (**`udev`** / **`mdev`**), mediante un mecanismo de notificación denominado *netlink socket*
- El device manager carga el driver (en caso que no esté ya cargado).

Provisión de Dispositivos

- Cuando registramos un nuevo dispositivo en un bus utilizando las funciones provistas en el core del bus, el kernel invoca a la función match del core que se registró con el driver del bus.
- Es decir dispara el matching loop.
- Esta función verifica si hay registrado en el bus algún driver que coincida con el dispositivo registrado.
- Si no encuentra coincidencia, no hace nada.
- Si la encuentra, el kernel la notifica al device manager (`udev` / `mdev`), mediante un mecanismo de notificación denominado *netlink socket*
- El device manager carga el driver (en caso que no esté ya cargado).
- Se ejecuta inmediatamente la función `probe()`

Provisión de Dispositivos

- Cuando registramos un nuevo dispositivo en un bus utilizando las funciones provistas en el core del bus, el kernel invoca a la función match del core que se registró con el driver del bus.
- Es decir dispara el matching loop.
- Esta función verifica si hay registrado en el bus algún driver que coincida con el dispositivo registrado.
- Si no encuentra coincidencia, no hace nada.
- Si la encuentra, el kernel la notifica al device manager (`udev` / `mdev`), mediante un mecanismo de notificación denominado *netlink socket*
- El device manager carga el driver (en caso que no esté ya cargado).
- Se ejecuta inmediatamente la función `probe()`
- Este *matching loop* inicia cada vez que se registra un device o un driver.

Provisión de Dispositivos

- Cuando registramos un nuevo dispositivo en un bus utilizando las funciones provistas en el core del bus, el kernel invoca a la función match del core que se registró con el driver del bus.
- Es decir dispara el matching loop.
- Esta función verifica si hay registrado en el bus algún driver que coincida con el dispositivo registrado.
- Si no encuentra coincidencia, no hace nada.
- Si la encuentra, el kernel la notifica al device manager (`udev` / `mdev`), mediante un mecanismo de notificación denominado *netlink socket*
- El device manager carga el driver (en caso que no esté ya cargado).
- Se ejecuta inmediatamente la función `probe()`
- Este *matching loop* inicia cada vez que se registra un device o un driver.
- El resumen de lo dicho hasta aqui se presenta a continuación

Provisión de Dispositivos

Lista de drivers I2C	Lista de drivers USB			Lista de drivers de Plataforma
driver n	driver m			gpio_reset driver
driver ...	driver ...			led_gpio driver
driver ...	driver ...			....
driver 2	driver 2			gpio_keyboard matrix
driver 1	driver 1	....	....	driver foo
I2C bus driver	USB bus driver	....	....	Platform Bus

Physical BUS (I2C, SPI, PCI, USB, SATA, MDIO, ...)

Provisión de Dispositivos

- Cada driver registrado y cada dispositivo se establece sobre un bus.

Provisión de Dispositivos

- Cada driver registrado y cada dispositivo se establece sobre un bus.
- De este modo se forma un árbol de dispositivos, con nodos padres y nodos child.

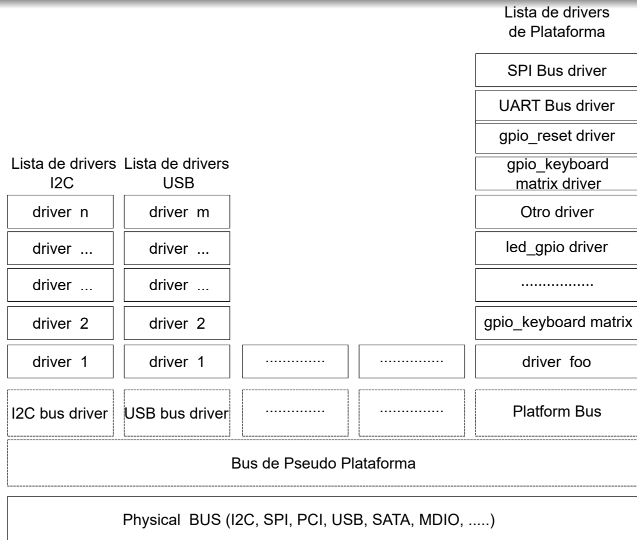
Provisión de Dispositivos

- Cada driver registrado y cada dispositivo se establece sobre un bus.
- De este modo se forma un árbol de dispositivos, con nodos padres y nodos child.
- Esto último incluye Buses. Por ejemplo, generalmente USB es un child de PCI, en tanto que los buses MDIO (Management Data Input/Output), tales como los dispositivos serie, o algunas variantes del estándar IEEE 802.3 pueden ser a su vez dependientes de otros nodos (incluso children de USB)

Provisión de Dispositivos

- Cada driver registrado y cada dispositivo se establece sobre un bus.
- De este modo se forma un árbol de dispositivos, con nodos padres y nodos child.
- Esto último incluye Buses. Por ejemplo, generalmente USB es un child de PCI, en tanto que los buses MDIO (Management Data Input/Output), tales como los dispositivos serie, o algunas variantes del estándar IEEE 802.3 pueden ser a su vez dependientes de otros nodos (incluso children de USB)
- Cuando un módulo registra un driver cuya función `platform_driver_probe()` está definida, el kernel recorre esta la tabla de dispositivos registrados en la plataforma y si lo encuentra invoca a la función `probe()` con los datos de la plataforma.

Provisión de Dispositivos



Como trabaja el matching plataform driver - device

- En tiempo de compilación del driver, el proceso de build extrae su información y la incorpora a un archivo de texto `/lib/modules/`uname -r`/modules.alias`

Como trabaja el matching plataform driver - device

- En tiempo de compilación del driver, el proceso de build extrae su información y la incorpora a un archivo de texto `/lib/modules/`uname -r`/modules.alias`
- El miembro `.name` de la estructura `platform_device` debe coincidir con el nombre del módulo, o no será detectado en el matching loop. Sino, hay que utilizar la macro `MODULE_ALIAS` para asignar otro nombre al módulo.

Como trabaja el matching plataform driver - device

- En tiempo de compilación del driver, el proceso de build extrae su información y la incorpora a un archivo de texto `/lib/modules/`uname -r`/modules.alias`
- El miembro `.name` de la estructura `platform_device` debe coincidir con el nombre del módulo, o no será detectado en el matching loop. Sino, hay que utilizar la macro `MODULE_ALIAS` para asignar otro nombre al módulo.
- Para exponer el ID de esta tabla que describe todos los dispositivos que soporta, el platform driver utiliza la siguiente macro:

Como trabaja el matching plataform driver - device

- En tiempo de compilación del driver, el proceso de build extrae su información y la incorpora a un archivo de texto **/lib/modules/`uname -r`/modules.alias**
- El miembro **.name** de la estructura **platform_device** debe coincidir con el nombre del módulo, o no será detectado en el matching loop. Sino, hay que utilizar la macro **MODULE_ALIAS** para asignar otro nombre al módulo.
- Para exponer el ID de esta tabla que describe todos los dispositivos que soporta, el platform driver utiliza la siguiente macro:

```
#define MODULE_DEVICE_TABLE(type, name)
```

Como trabaja el matching plataform driver - device

- En tiempo de compilación del driver, el proceso de build extrae su información y la incorpora a un archivo de texto **/lib/modules/`uname -r`/modules.alias**
- El miembro **.name** de la estructura **platform_device** debe coincidir con el nombre del módulo, o no será detectado en el matching loop. Sino, hay que utilizar la macro **MODULE_ALIAS** para asignar otro nombre al módulo.
- Para exponer el ID de esta tabla que describe todos los dispositivos que soporta, el platform driver utiliza la siguiente macro:

```
#define MODULE_DEVICE_TABLE(type, name)
```

- **type** : En **include/linux/mod_devicetable.h** , se definen los tipos de plataforma. Este campo es el nombre definido allí (i2c, spi, usb, of, pci, acpi, etc.).

Como trabaja el matching plataform driver - device

- En tiempo de compilación del driver, el proceso de build extrae su información y la incorpora a un archivo de texto `/lib/modules/`uname -r`/modules.alias`
- El miembro `.name` de la estructura `platform_device` debe coincidir con el nombre del módulo, o no será detectado en el matching loop. Sino, hay que utilizar la macro `MODULE_ALIAS` para asignar otro nombre al módulo.
- Para exponer el ID de esta tabla que describe todos los dispositivos que soporta, el platform driver utiliza la siguiente macro:

```
#define MODULE_DEVICE_TABLE(type, name)
```

- `type` : En `include/linux/mod_devicetable.h` , se definen los tipos de plataforma. Este campo es el nombre definido allí (i2c, spi, usb, of, pci, acpi, etc.).
- `name` : puntero a `xxx_device_id` . xxx = nombre del platform device

Como trabaja el matching plataform driver - device

- Hay cuatro mecanismos de matching.

Como trabaja el matching plataform driver - device

- Hay cuatro mecanismos de matching.

✓ OF style. Basado en el device tree.

Como trabaja el matching plataform driver - device

- Hay cuatro mecanismos de matching.
- ✓ OF style. Basado en el device tree.
- ✓ ACPI match. Advanced Configuration and Power Interface

Como trabaja el matching plataform driver - device

- Hay cuatro mecanismos de matching.
- ✓ OF style. Basado en el device tree.
- ✓ ACPI match. Advanced Configuration and Power Interface
- ✓ ID Table Matching

Como trabaja el matching plataform driver - device

- Hay cuatro mecanismos de matching.
- ✓ OF style. Basado en el device tree.
- ✓ ACPI match. Advanced Configuration and Power Interface
- ✓ ID Table Matching
- ✓ Per device-specific data on ID table matching

Como trabaja el matching plataform driver - device

- Hay cuatro mecanismos de matching.

✓ OF style. Basado en el device tree.

✓ ACPI match. Advanced Configuration and Power Interface

✓ ID Table Matching

✓ Per device-specific data on ID table matching

```
struct device_driver {
    const           char * name;
    struct          bus_type * bus;
    struct          module * owner;
    const           char * mod_name;
    bool            suppress_bind_attrs;
    enum            probe_type probe_type;
    const struct    of_device_id * of_match_table;
    const struct    acpi_device_id * acpi_match_table;
    int (* probe)   (struct device *dev);
    int (* remove)  (struct device *dev);
    void (* shutdown) (struct device *dev);
    int (* suspend) (struct device *dev, pm_message_t state);
    int (* resume)  (struct device *dev);
    const struc     attribute_group ** groups;
    const struct    dev_pm_ops * pm;
    struct          driver_private * p;
};
```

Como trabaja el matching plataform driver - device

```
1 static int platform_match(struct device *dev, struct device_driver *drv)
2 {
3     struct platform_device *pdev = to_platform_device(dev);
4     struct platform_driver *pdrv = to_platform_driver(drv);
5 /* When driver_override is set, only bind to the matching driver */
6     if (pdev->driver_override)
7         return !strcmp(pdev->driver_override, drv->name);
8 /* Attempt an OF style match first */
9     if (of_driver_match_device(dev, drv))
10        return 1;
11 /* Then try ACPI style match */
12    if (acpi_driver_match_device(dev, drv))
13        return 1;
14 /* Then try to match against the id table */
15    if (pdrv->id_table)
16        return platform_match_id(pdrv->id_table, pdev) != NULL;
17 /* fall-back to driver name match */
18    return (strcmp(pdev->name, drv->name) == 0);
19 }
```

Como trabaja el matching plataform driver - device

```
1  static int platform_match(struct device *dev, struct device_driver *drv)
2  {
3      struct platform_device *pdev = to_platform_device(dev);
4      struct platform_driver *pdrv = to_platform_driver(drv);
5  /* When driver_override is set, only bind to the matching driver */
6      if (pdev->driver_override)
7          return !strcmp(pdev->driver_override, drv->name);
8  /* Attempt an OF style match first */
9      if (of_driver_match_device(dev, drv))
10         return 1;
11 /* Then try ACPI style match */
12     if (acpi_driver_match_device(dev, drv))
13         return 1;
14 /* Then try to match against the id table */
15     if (pdrv->id_table)
16         return platform_match_id(pdrv->id_table, pdev) != NULL;
17 /* fall-back to driver name match */
18     return (strcmp(pdev->name, drv->name) == 0);
19 }
```

Si corresponde a Open Firmware o a ACPI, **`struct device_driver`** tiene el puntero a **`device_id`**, sino, la clave del matching loop pasa a ser la función de la línea 16, **`platform_match_id`**.

ID table matching

ID table matching

- Se basa en la estructura `struct device_id` .

ID table matching

- Se basa en la estructura `struct device_id`.
- Definidas en `include/linux/mod_devicetable.h`

ID table matching

- Se basa en la estructura `struct device_id`.
- Definidas en `include/linux/mod_devicetable.h`
- Para encontrar la estructura correcta necesitamos agregar como prefijo el tipo de device (**i2c**, **spi**, **usb**, **pci**, etc.)

ID table matching

- Se basa en la estructura `struct device_id`.
- Definidas en `include/linux/mod_devicetable.h`
- Para encontrar la estructura correcta necesitamos agregar como prefijo el tipo de device (**i2c**, **spi**, **usb**, **pci**, etc.)
- Si no depende de un bus el prefijo es ***platform***.

ID table matching

- Se basa en la estructura `struct device_id`.
- Definidas en `include/linux/mod_devicetable.h`
- Para encontrar la estructura correcta necesitamos agregar como prefijo el tipo de device (**i2c**, **spi**, **usb**, **pci**, etc.)
- Si no depende de un bus el prefijo es ***platform***.
- La forma típica de una estructura de éstas es la siguiente:

ID table matching

- Se basa en la estructura `struct device_id`.
- Definidas en `include/linux/mod_devicetable.h`
- Para encontrar la estructura correcta necesitamos agregar como prefijo el tipo de device (**i2c**, **spi**, **usb**, **pci**, etc.)
- Si no depende de un bus el prefijo es ***platform***.
- La forma típica de una estructura de éstas es la siguiente:

```
struct platform_device_id {
    __u8 id[PLATFORM_NAME_LEN];
    kernel_ulong_t driver_data;
};
```

ID table matching

- Se basa en la estructura `struct device_id`.
- Definidas en `include/linux/mod_devicetable.h`
- Para encontrar la estructura correcta necesitamos agregar como prefijo el tipo de device (**i2c**, **spi**, **usb**, **pci**, etc.)
- Si no depende de un bus el prefijo es ***platform***.
- La forma típica de una estructura de éstas es la siguiente:

```
struct platform_device_id {
    __u8 id[PLATFORM_NAME_LEN];
    kernel_ulong_t driver_data;
};
```

- El miembro `id` es un arreglo que contiene una cadena de texto correspondiente al nombre del driver.

ID table matching

- Se basa en la estructura `struct device_id`.
- Definidas en `include/linux/mod_devicetable.h`
- Para encontrar la estructura correcta necesitamos agregar como prefijo el tipo de device (**i2c**, **spi**, **usb**, **pci**, etc.)
- Si no depende de un bus el prefijo es ***platform***.
- La forma típica de una estructura de éstas es la siguiente:

```
struct platform_device_id {
    __u8 id[PLATFORM_NAME_LEN];
    kernel_ulong_t driver_data;
};
```

- El miembro `id` es un arreglo que contiene una cadena de texto correspondiente al nombre del driver.
- El tipo `__un` es un conjunto de typedefs específicos de Linux, no portables. el doble `_` indica que se trata de una definición no estándar.

ID table matching

ID table matching

- El tipo `kernel_ulong_t` es algun tipo de dato particular del driver, que puede ser un puntero.

ID table matching

- El tipo **kernel_ulong_t** es algun tipo de dato particular del driver, que puede ser un puntero.
- En particular en **drivers/tty/serial/imx.c** está el armado de esta estructura para el serial-imx.

```
static const struct platform_device_id imx_uart_devtype[] = {
	{
		.name = "imx1-uart",
		.driver_data = (kernel_ulong_t) &imx_uart_devdata[IMX1_UART],
	}, {
		.name = "imx21-uart",
		.driver_data = (kernel_ulong_t) &imx_uart_devdata[IMX21_UART],
	}, {
		.name = "imx53-uart",
		.driver_data = (kernel_ulong_t) &imx_uart_devdata[IMX53_UART],
	}, {
		.name = "imx6q-uart",
		.driver_data = (kernel_ulong_t) &imx_uart_devdata[IMX6Q_UART],
	}, {
		/* sentinel */
	}
};
```

Función `platform_match_idg`

- El procedimiento termina siendo una comparación de cadenas de texto.

Función `platform_match_idg`

- El procedimiento termina siendo una comparación de cadenas de texto.
- La clave está en la siguiente función:

Función `platform_match_idg`

- El procedimiento termina siendo una comparación de cadenas de texto.
- La clave está en la siguiente función:

```
static const struct platform_device_id *platform_match_id(
            const struct platform_device_id *devid, struct platform_device *pdev)
{
    while (devid->id[0]) {
        if (strcmp(pdev->id, devid->id) == 0) {
            pdev->id_entry = devid;
            return devid;
        }
        devid++;
    }
    return NULL;
}
```

OF Match. En el drevice tree

OF Match. En el drevice tree

- Utiliza la propiedad `compatible` del Device Tree que matchee con la entrada homónima de `of_match_table` definida en la estructra driver.

OF Match. En el drevice tree

- Utiliza la propiedad `compatible` del Device Tree que matchee con la entrada homónima de `of_match_table` definida en la estructra driver.

```
/*Arma arreglo de estructuras para i2c. Cada elemento tiene una propiedad "compatible"*/
static const struct of_device_id i2c_of_match[] = {
    { .compatible = "td3,omap4-i2c" },
    {},
};
MODULE_DEVICE_TABLE(of, i2c_of_match);
static struct platform_driver i2c_driver = {
    .probe        = i2c_probe,
    .remove       = i2c_remove,
    .driver       = {
        .name             = "td3_i2c",
        .of_match_table = of_match_ptr(i2c_of_match),/*Apunta a string compatible*/
    },
};
```

OF Match. En el drevice tree

- Utiliza la propiedad **compatible** del Device Tree que matchee con la entrada homónima de **of_match_table** definida en la estructra driver.

```
/*Arma arreglo de estructuras para i2c. Cada elemento tiene una propiedad "compatible"*/
static const struct of_device_id i2c_of_match[] = {
    { .compatible = "td3,omap4-i2c" },
    {},
};
MODULE_DEVICE_TABLE(of, i2c_of_match);
static struct platform_driver i2c_driver = {
    .probe      = i2c_probe,
    .remove     = i2c_remove,
    .driver     = {
        .name           = "td3_i2c",
        .of_match_table = of_match_ptr(i2c_of_match),/*Apunta a string compatible*/
    },
};
```

- Este bloque va en la función que definida para ejecutarse al instalar el módulo es decir en **module_init(i2c_init);** .

OF Match. En el drevice tree

- Cuando la función `module_init` ejecuta `MODULE_DEVICE_TABLE` declara la tabla y el kernel busca en el Device Tree la string de la propiedad `.compatible`

```
i2c_td3@4819c000 {
        compatible = "td3,omap4-i2c";
        #address-cells = <0x1>;
        #size-cells = <0x0>;
        ti,hwmods = "i2c3";
        reg = <0x4819c000 0x1000>;
        interrupts = <0x1e>;
        status = "okay";
        pinctrl-names = "default";
        pinctrl-0 = <0x34>;
        clock-frequency = <0x186a0>;
        cape_eeprom0@54 {
            compatible = "at,24c256";
            reg = <0x54>;
            #address-cells = <0x1>;
            #size-cells = <0x1>;
            cape_data@0 {
                reg = <0x0 0x100>;
            };
        };
```

OF Match. En el drevice tree

- Cuando la función `module_init` ejecuta `MODULE_DEVICE_TABLE` declara la tabla y el kernel busca en el Device Tree la string de la propiedad `.compatible`
- Si la encuentra, instancia una estructura `platform_device` la inicializa con la información del Device Tree e invoca a la función `probe()`, con un puntero a `platform_device` como argumento.

```
i2c_td3@4819c000 {
        compatible = "td3,omap4-i2c";
        #address-cells = <0x1>;
        #size-cells = <0x0>;
        ti,hwmods = "i2c3";
        reg = <0x4819c000 0x1000>;
        interrupts = <0x1e>;
        status = "okay";
        pinctrl-names = "default";
        pinctrl-0 = <0x34>;
        clock-frequency = <0x186a0>;
        cape_eeprom0@54 {
            compatible = "at,24c256";
            reg = <0x54>;
            #address-cells = <0x1>;
            #size-cells = <0x1>;
            cape_data@0 {
                reg = <0x0 0x100>;
            };
        };
```

OF Match. En el drevice tree

- Cuando la función `module_init` ejecuta `MODULE_DEVICE_TABLE` declara la tabla y el kernel busca en el Device Tree la string de la propiedad `.compatible`
- Si la encuentra, instancia una estructura `platform_device` la inicializa con la información del Device Tree e invoca a la función `probe()`, con un puntero a `platform_device` como argumento.
- La función `probe` la obtiene de `struct platform_driver` definida en el listado del slide anterior.

```
i2c_td3@4819c000 {
        compatible = "td3,omap4-i2c";
        #address-cells = <0x1>;
        #size-cells = <0x0>;
        ti,hwmods = "i2c3";
        reg = <0x4819c000 0x1000>;
        interrupts = <0x1e>;
        status = "okay";
        pinctrl-names = "default";
        pinctrl-0 = <0x34>;
        clock-frequency = <0x186a0>;
        cape_eeprom0@54 {
            compatible = "at,24c256";
            reg = <0x54>;
            #address-cells = <0x1>;
            #size-cells = <0x1>;
            cape_data@0 {
                reg = <0x0 0x100>;
            };
        };
```

OF Match. En el drevice tree

- Cuando la función **module_init** ejecuta **MODULE_DEVICE_TABLE** declara la tabla y el kernel busca en el Device Tree la string de la propiedad **.compatible**
- Si la encuentra, instancia una estructura **platform_device** la inicializa con la información del Device Tree e invoca a la función **probe()**, con un puntero a **platform_device** como argumento.
- La función **probe** la obtiene de **struct platform_driver** definida en el listado del slide anterior.

```
i2c_td3@4819c000 {
        compatible = "td3,omap4-i2c";
        #address-cells = <0x1>;
        #size-cells = <0x0>;
        ti,hwmods = "i2c3";
        reg = <0x4819c000 0x1000>;
        interrupts = <0x1e>;
        status = "okay";
        pinctrl-names = "default";
        pinctrl-0 = <0x34>;
        clock-frequency = <0x186a0>;
        cape_eeprom0@54 {
            compatible = "at,24c256";
            reg = <0x54>;
            #address-cells = <0x1>;
            #size-cells = <0x1>;
            cape_data@0 {
                reg = <0x0 0x100>;
            };
        };
```

- El cuerpo de la función **probe ()** hay que escribirlo en nuestro platform device y el él inicializar el resto de los recursos necesarios

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Procesador SITARA AM335x

Block Name	Start_address (hex)	End_address (hex)	Size	Description
USBSS	0x4740_0000	0x4740_0FFF	32KB	USB Subsystem Registers
USB0	0x4740_1000	0x4740_12FF		USB0 Controller Registers
USB0_PHY	0x4740_1300	0x4740_13FF		USB0 PHY Registers
USB0 Core	0x4740_1400	0x4740_17FF		USB0 Core Registers
USB1	0x4740_1800	0x4740_1AFF		USB1 Controller Registers
USB1_PHY	0x4740_1B00	0x4740_1BFF		USB1 PHY Registers
USB1 Core	0x4740_1C00	0x4740_1FFF		USB1 Core Registers
USB CPPI DMA Controller	0x4740_2000	0x4740_2FFF		USB CPPI DMA Controller Registers
USB CPPI DMA Scheduler	0x4740_3000	0x4740_3FFF		USB CPPI DMA Scheduler Registers
USB Queue Manager	0x4740_4000	0x4740_7FFF		USB Queue Manager Registers
Reserved	0x4740_8000	0x477F_FFFF	4MB-32KB	Reserved
Reserved	0x4780_0000	0x4780_FFFF	64KB	Reserved
MMCHS2	0x4781_0000	0x4781_FFFF	64KB	MMCHS2
Reserved	0x4782_0000	0x47BF_FFFF	4MB-128KB	Reserved
Reserved	0x47C0_0000	0x47FF_FFFF	4MB	Reserved
L4_PER	0x4800_0000	0x48FF_FFFF	16MB	L4 Peripheral (see L4_PER table)
TPCC (EDMA3CC)	0x4900_0000	0x490F_FFFF	1MB	EDMA3 Channel Controller Registers
Reserved	0x4910_0000	0x497F_FFFF	7MB	Reserved
TPTC0 (EDMA3TC0)	0x4980_0000	0x498F_FFFF	1MB	EDMA3 Transfer Controller 0 Registers

ARM no dispone espacio de direccionamiento separado para E/S. Los registros de todos sus devices están mapeados en memoria.

Registros del SITARA AM335x

- Esto sugiere un problema para la Unidad de Paginación ya que la dirección física está impuesta por el hardware.

Registros del SITARA AM335x

- Esto sugiere un problema para la Unidad de Paginación ya que la dirección física está impuesta por el hardware.
- Normalmente el problema es inverso. Dada una dirección lógica el SO busca una dirección física disponible y en base a esta arma las tablas de traducción en la Unidad de Paginación.

Registros del SITARA AM335x

- Esto sugiere un problema para la Unidad de Paginación ya que la dirección física está impuesta por el hardware.
- Normalmente el problema es inverso. Dada una dirección lógica el SO busca una dirección física disponible y en base a esta arma las tablas de traducción en la Unidad de Paginación.
- Aqui es dada la direcón física, asignar una dirección lógica de manera unívoca (en el mismo proceso no pueden coexistir dos ítemes con idéntica dirección virtual.

Registros del SITARA AM335x

- Esto sugiere un problema para la Unidad de Paginación ya que la dirección física está impuesta por el hardware.
- Normalmente el problema es inverso. Dada una dirección lógica el SO busca una dirección física disponible y en base a esta arma las tablas de traducción en la Unidad de Paginación.
- Aqui es dada la direcón física, asignar una dirección lógica de manera unívoca (en el mismo proceso no pueden coexistir dos ítemes con idéntica dirección virtual.
- Para esto se deben utilizar las siguientes funciones:

Registros del SITARA AM335x

- Esto sugiere un problema para la Unidad de Paginación ya que la dirección física está impuesta por el hardware.
- Normalmente el problema es inverso. Dada una dirección lógica el SO busca una dirección física disponible y en base a esta arma las tablas de traducción en la Unidad de Paginación.
- Aqui es dada la direcón física, asignar una dirección lógica de manera unívoca (en el mismo proceso no pueden coexistir dos ítemes con idéntica dirección virtual.
- Para esto se deben utilizar las siguientes funciones:

```
void * ioremap(unsigned long offset, unsigned long size);
void iounmap(void * addr);
```

Registros del SITARA AM335x

- Esto sugiere un problema para la Unidad de Paginación ya que la dirección física está impuesta por el hardware.
- Normalmente el problema es inverso. Dada una dirección lógica el SO busca una dirección física disponible y en base a esta arma las tablas de traducción en la Unidad de Paginación.
- Aqui es dada la direcón física, asignar una dirección lógica de manera unívoca (en el mismo proceso no pueden coexistir dos ítemes con idéntica dirección virtual.
- Para esto se deben utilizar las siguientes funciones:

```
void * ioremap(unsigned long offset, unsigned long size);
void iounmap(void * addr);
```

- Estas funciones devuelven un puntero a la zona de memoria física que se quiso mapear.

Registros del SITARA AM335x

- Esto sugiere un problema para la Unidad de Paginación ya que la dirección física está impuesta por el hardware.
- Normalmente el problema es inverso. Dada una dirección lógica el SO busca una dirección física disponible y en base a esta arma las tablas de traducción en la Unidad de Paginación.
- Aqui es dada la direcón física, asignar una dirección lógica de manera unívoca (en el mismo proceso no pueden coexistir dos ítemes con idéntica dirección virtual.
- Para esto se deben utilizar las siguientes funciones:

```
void * ioremap(unsigned long offset, unsigned long size);
void iounmap(void * addr);
```

- Estas funciones devuelven un puntero a la zona de memoria física que se quiso mapear.
- Por lo general no es identity mapping.

Registros del SITARA AM335x

- A continuación una alternativa similar a la anterior, pero que en vez de especficar la dirección física, la busca en el Device Tree:

Registros del SITARA AM335x

- A continuación una alternativa similar a la anterior, pero que en vez de especficar la dirección física, la busca en el Device Tree:

```
void iomem * of_iomap(struct device node *device, int index);
void iounmap(void * addr);
```

Registros del SITARA AM335x

- A continuación una alternativa similar a la anterior, pero que en vez de especficar la dirección física, la busca en el Device Tree:

```
void iomem * of_iomap(struct device node *device, int index);
void iounmap(void * addr);
```

- Se pasa (`(pdev->dev).of_node`) como argumento, que es un puntero, uno de los elementos de la estructura que se recibe por referencia en la función `probe ()` .

Registros del SITARA AM335x

- A continuación una alternativa similar a la anterior, pero que en vez de especficar la dirección física, la busca en el Device Tree:

```
void iomem * of_iomap(struct device node *device, int index);
void iounmap(void * addr);
```

- Se pasa (**(pdev->dev).of_node**) como argumento, que es un puntero, uno de los elementos de la estructura que se recibe por referencia en la función **probe ()** .
- Para “desmapear”, la función es la misma de siempre.

Registros del SITARA AM335x

- A continuación una alternativa similar a la anterior, pero que en vez de especficar la dirección física, la busca en el Device Tree:

```
void iomem * of_iomap(struct device node *device, int index);
void iounmap(void * addr);
```

- Se pasa (`(pdev->dev).of_node`) como argumento, que es un puntero, uno de los elementos de la estructura que se recibe por referencia en la función `probe ()` .
- Para “desmapear”, la función es la misma de siempre.
- Por otra parte, para leer y escribir en los registros se utilizan las siguientes funciones:

Registros del SITARA AM335x

- A continuación una alternativa similar a la anterior, pero que en vez de especficar la dirección física, la busca en el Device Tree:

```
void iomem * of_iomap(struct device node *device, int index);
void iounmap(void * addr);
```

- Se pasa (**`(pdev->dev).of_node`**) como argumento, que es un puntero, uno de los elementos de la estructura que se recibe por referencia en la función **`probe ()`** .
- Para “desmapear”, la función es la misma de siempre.
- Por otra parte, para leer y escribir en los registros se utilizan las siguientes funciones:

```
unsigned int ioread32(void *addr);
void iowrite32(u32 value, void *addr);
```

Registros del SITARA AM335x

- A continuación una alternativa similar a la anterior, pero que en vez de especficar la dirección física, la busca en el Device Tree:

```
void iomem * of_iomap(struct device node *device, int index);
void iounmap(void * addr);
```

- Se pasa (`(pdev->dev).of_node`) como argumento, que es un puntero, uno de los elementos de la estructura que se recibe por referencia en la función `probe ()` .
- Para "desmapear", la función es la misma de siempre.
- Por otra parte, para leer y escribir en los registros se utilizan las siguientes funciones:

```
unsigned int ioread32(void *addr);
void iowrite32(u32 value, void *addr);
```

- Aunque existen versiones para 16 y 8 bits, conviene emplear las de 32 para asegurar la portabilidad del driver y la compatibilidad de arquitectura del procesador.

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Interrupciones - Generalidades

- Al principio el kernel de Linux asociaba cada dispositivo a la línea de interrupción a la que estaba conectado.

Interrupciones - Generalidades

- Al principio el kernel de Linux asociaba cada dispositivo a la línea de interrupción a la que estaba conectado.
- Al tener que portar a sistemas embebidos se enfrentaron algunos cambios.

Interrupciones - Generalidades

- Al principio el kernel de Linux asociaba cada dispositivo a la línea de interrupción a la que estaba conectado.
- Al tener que portar a sistemas embebidos se enfrentaron algunos cambios.
- Por ejemplo: los puertos GPIOs podían ser definidos como controladores de interrupciones en cascada.

Interrupciones - Generalidades

- Al principio el kernel de Linux asociaba cada dispositivo a la línea de interrupción a la que estaba conectado.
- Al tener que portar a sistemas embebidos se enfrentaron algunos cambios.
- Por ejemplo: los puertos GPIOs podían ser definidos como controladores de interrupciones en cascada.
- Para solucionar esto el kernel de Linux adquirió capacidad para manejar dispositivos que pueden ser definidos como controladores de interrupciones.

Interrupciones - Generalidades

- Al principio el kernel de Linux asociaba cada dispositivo a la línea de interrupción a la que estaba conectado.
- Al tener que portar a sistemas embebidos se enfrentaron algunos cambios.
- Por ejemplo: los puertos GPIOs podían ser definidos como controladores de interrupciones en cascada.
- Para solucionar esto el kernel de Linux adquirió capacidad para manejar dispositivos que pueden ser definidos como controladores de interrupciones.
- Es así como a cada interrupción se le asigna un `VIRQ` (virtual IRQ number), el cual usualmente no coincide con la línea de interrupción.

Obteniendo una VIRQ

- La línea de interrupción se define en el device tree, y en la función **`probe`** se pide un **`VIRQ`** para esa interrupción.

Obteniendo una VIRQ

- La línea de interrupción se define en el device tree, y en la función `probe` se pide un `VIRQ` para esa interrupción.
- A través de la función:

Obteniendo una VIRQ

- La línea de interrupción se define en el device tree, y en la función **`probe`** se pide un **`VIRQ`** para esa interrupción.
- A través de la función:

```
int platform_get_irq(struct platform_device *, unsigned int);
```

Obteniendo una VIRQ

- La línea de interrupción se define en el device tree, y en la función **probe** se pide un **VIRQ** para esa interrupción.
- A través de la función:

```
int platform_get_irq(struct platform_device *, unsigned int);
```

- se le pasa el puntero recibido en la función **probe**, y un índice, por si hay más de una INT.

Obteniendo una VIRQ

- La línea de interrupción se define en el device tree, y en la función **`probe`** se pide un **`VIRQ`** para esa interrupción.
- A través de la función:

```
int platform_get_irq(struct platform_device *, unsigned int);
```

- se le pasa el puntero recibido en la función **`probe`**, y un índice, por si hay más de una INT.
- Usualmente este parámetro es 0. Devuelve en **`VIRQ`**

Interrupt Handlers

- Corren en un contexto específico para interrupciones.

Interrupt Handlers

- Corren en un contexto específico para interrupciones.
- Deben respetar un prototipo establecido por el kernel.

Interrupt Handlers

- Corren en un contexto específico para interrupciones.
- Deben respetar un prototipo establecido por el kernel.
- El kernel tiene mecanismos para dividir las interrupciones en Top Halves & Bottom Halves.

Interrupt Handlers

- Corren en un contexto específico para interrupciones.
- Deben respetar un prototipo establecido por el kernel.
- El kernel tiene mecanismos para dividir las interrupciones en Top Halves & Bottom Halves.
- Para registrar un INT Handler en el Kenel

Interrupt Handlers

- Corren en un contexto específico para interrupciones.
- Deben respetar un prototipo establecido por el kernel.
- El kernel tiene mecanismos para dividir las interrupciones en Top Halves & Bottom Halves.
- Para registrar un INT Handler en el Kenel

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
const char *name, void *dev);
```

Interrupt Handlers

- Corren en un contexto específico para interrupciones.
- Deben respetar un prototipo establecido por el kernel.
- El kernel tiene mecanismos para dividir las interrupciones en Top Halves & Bottom Halves.
- Para registrar un INT Handler en el Kenel

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
const char *name, void *dev);
```

- **irq** Número de IRQ a alocar.

Interrupt Handlers

- Corren en un contexto específico para interrupciones.
- Deben respetar un prototipo establecido por el kernel.
- El kernel tiene mecanismos para dividir las interrupciones en Top Halves & Bottom Halves.
- Para registrar un INT Handler en el Kenel

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
const char *name, void *dev);
```

- **`irq`** Número de IRQ a alocar.
- **`handler`** Puntero al IRQ Handler
  (**`typedef irqreturn_t (*irq_handler_t)(int, void *);`**

Interrupt Handlers

- Corren en un contexto específico para interrupciones.
- Deben respetar un prototipo establecido por el kernel.
- El kernel tiene mecanismos para dividir las interrupciones en Top Halves & Bottom Halves.
- Para registrar un INT Handler en el Kenel

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
const char *name, void *dev);
```

- `irq` Número de IRQ a alocar.
- `handler` Puntero al IRQ Handler
  (`typedef irqreturn_t (*irq_handler_t)(int, void *);`
- `flags` Flags correspondientes al comportamiento del IRQ.
  (Ver: `<linux/interrupt.h>`)

Interrupt Handlers

- Corren en un contexto específico para interrupciones.
- Deben respetar un prototipo establecido por el kernel.
- El kernel tiene mecanismos para dividir las interrupciones en Top Halves & Bottom Halves.
- Para registrar un INT Handler en el Kenel

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
const char *name, void *dev);
```

- **irq** Número de IRQ a alocar.
- **handler** Puntero al IRQ Handler
  (**typedef irqreturn_t (*irq_handler_t)(int, void *);**
- **flags** Flags correspondientes al comportamiento del IRQ.
  (Ver: **<linux/interrupt.h>**)
- ***name** Nombre que será exportado en **/proc/** .

Interrupt Handlers

- Corren en un contexto específico para interrupciones.
- Deben respetar un prototipo establecido por el kernel.
- El kernel tiene mecanismos para dividir las interrupciones en Top Halves & Bottom Halves.
- Para registrar un INT Handler en el Kenel

```
int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
const char *name, void *dev);
```

- **irq** Número de IRQ a alocar.
- **handler** Puntero al IRQ Handler (**typedef irqreturn_t (*irq_handler_t)(int, void *);**
- **flags** Flags correspondientes al comportamiento del IRQ. (Ver: **<linux/interrupt.h>**)
- ***name** Nombre que será exportado en **/proc/** .
- ***dev** Identifica al dispositivo cuando se comparten IRQs.

Interrupt Handlers

- Para eliminar un INT Handler del Kernel:

Interrupt Handlers

- Para eliminar un INT Handler del Kernel:

```
void free_irq(unsigned int irq, void *dev);
```

Interrupt Handlers

- Para eliminar un INT Handler del Kernel:

```
void free_irq(unsigned int irq, void *dev);
```

- Prototipo del Handler

Interrupt Handlers

- Para eliminar un INT Handler del Kernel:

```
void free_irq(unsigned int irq, void *dev);
```

- Prototipo del Handler

```
static irqreturn_t intr_handler(int irq, void *dev_id, struct pt_regs *regs);
```

Interrupt Handlers

- Para eliminar un INT Handler del Kernel:

```
void free_irq(unsigned int irq, void *dev);
```

- Prototipo del Handler

```
static irqreturn_t intr_handler(int irq, void *dev_id, struct pt_regs *regs);
```

- **int irq** Número de IRQ.

Interrupt Handlers

- Para eliminar un INT Handler del Kernel:

```
void free_irq(unsigned int irq, void *dev);
```

- Prototipo del Handler

```
static irqreturn_t intr_handler(int irq, void *dev_id, struct pt_regs *regs);
```

- **int irq** Número de IRQ.
- **void *dev_id** Puntero al device ID para identificar el dispositivo en caso de tener shared Interrupts.

Interrupt Handlers

- Para eliminar un INT Handler del Kernel:

```
void free_irq(unsigned int irq, void *dev);
```

- Prototipo del Handler

```
static irqreturn_t intr_handler(int irq, void *dev_id, struct pt_regs *regs);
```

- **`int irq`** Número de IRQ.
- **`void *dev_id`** Puntero al device ID para identificar el dispositivo en caso de tener shared Interrupts.
- **`struct pt_regs *regs`** Puntero a estructura que contiene el estado de los registros del procesador previo a la interrupción.

Interrupt Handlers

- Para eliminar un INT Handler del Kernel:

```
void free_irq(unsigned int irq, void *dev);
```

- Prototipo del Handler

```
static irqreturn_t intr_handler(int irq, void *dev_id, struct pt_regs *regs);
```

- **`int irq`** Número de IRQ.
- **`void *dev_id`** Puntero al device ID para identificar el dispositivo en caso de tener shared Interrupts.
- **`struct pt_regs *regs`** Puntero a estructura que contiene el estado de los registros del procesador previo a la interrupción.
- Retorna: **`IRQ_NONE`** ó **`IRQ_HANDLED`** .

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Sleep y demoras

Sleep y demoras

- En casos que el driver tenga que esperar un lapso de tiempo, se deberá evaluar si se le da curso a la espera.

Sleep y demoras

- En casos que el driver tenga que esperar un lapso de tiempo, se deberá evaluar si se le da curso a la espera.
  1. `Sleep` : durante el tiempo de espera el procesador duerme. Esta es la opción más común. Es importante remarcar que nunca se puede poner a dormir a un procesador en contexto atómico

Sleep y demoras

- En casos que el driver tenga que esperar un lapso de tiempo, se deberá evaluar si se le da curso a la espera.
  1. `Sleep` : durante el tiempo de espera el procesador duerme. Esta es la opción más común. Es importante remarcar que nunca se puede poner a dormir a un procesador en contexto atómico
  2. `Software o Hardware loop` : durante el tiempo de espera el procesador espera recorriendo un loop. *Este tipo de demoras deben usarse en contexto atómico y con mucha precaución*.

Sleep y demoras

- En casos que el driver tenga que esperar un lapso de tiempo, se deberá evaluar si se le da curso a la espera.
  1. `Sleep` : durante el tiempo de espera el procesador duerme. Esta es la opción más común. Es importante remarcar que nunca se puede poner a dormir a un procesador en contexto atómico
  2. `Software o Hardware loop` : durante el tiempo de espera el procesador espera recorriendo un loop. *Este tipo de demoras deben usarse en contexto atómico y con mucha precaución*.
- Para poner a dormir un proceso se debe utilizar alguna de las siguientes funciones:

Sleep y demoras

- En casos que el driver tenga que esperar un lapso de tiempo, se deberá evaluar si se le da curso a la espera.
  1. **`Sleep`** : durante el tiempo de espera el procesador duerme. Esta es la opción más común. Es importante remarcar que nunca se puede poner a dormir a un procesador en contexto atómico
  2. **`Software o Hardware loop`** : durante el tiempo de espera el procesador espera recorriendo un loop. *Este tipo de demoras deben usarse en contexto atómico y con mucha precaución*.
- Para poner a dormir un proceso se debe utilizar alguna de las siguientes funciones:

```
wait_event_interruptible(queue, condition);
void wake_up_interruptible(wait_queue_head_t *queue);
```

Sleep y demoras

- En casos que el driver tenga que esperar un lapso de tiempo, se deberá evaluar si se le da curso a la espera.
  1. **`Sleep`** : durante el tiempo de espera el procesador duerme. Esta es la opción más común. Es importante remarcar que nunca se puede poner a dormir a un procesador en contexto atómico
  2. **`Software o Hardware loop`** : durante el tiempo de espera el procesador espera recorriendo un loop. *Este tipo de demoras deben usarse en contexto atómico y con mucha precaución*.
- Para poner a dormir un proceso se debe utilizar alguna de las siguientes funciones:

```
wait_event_interruptible(queue, condition);
void wake_up_interruptible(wait_queue_head_t *queue);
```

- La Primer función pone a dormir el proceso, la segunda lo despierta. Se le debe pasar una estructura de tipo **`wait_queue_head_t`**

Sleep y demoras

- En casos que el driver tenga que esperar un lapso de tiempo, se deberá evaluar si se le da curso a la espera.
    1. **`Sleep`** : durante el tiempo de espera el procesador duerme. Esta es la opción más común. Es importante remarcar que nunca se puede poner a dormir a un procesador en contexto atómico
    2. **`Software o Hardware loop`** : durante el tiempo de espera el procesador espera recorriendo un loop. *Este tipo de demoras deben usarse en contexto atómico y con mucha precaución*.
- Para poner a dormir un proceso se debe utilizar alguna de las siguientes funciones:

```
wait_event_interruptible(queue, condition);
void wake_up_interruptible(wait_queue_head_t *queue);
```

- La Primer función pone a dormir el proceso, la segunda lo despierta. Se le debe pasar una estructura de tipo **`wait_queue_head_t`**
- Al despertar el proceso se debe asegurar que la condición sea verdadera.

Sleep y demoras

```
static DECLARE_WAIT_QUEUE_HEAD (mi_queue);
```

Sleep y demoras

```
static DECLARE_WAIT_QUEUE_HEAD (mi_queue);
```

- Existe otra versión de **wait_event** que pone al proceso en estado **TASK_UNINTERRUPTIBLE** .

Sleep y demoras

```
static DECLARE_WAIT_QUEUE_HEAD (mi_queue);
```

- Existe otra versión de **wait_event** que pone al proceso en estado **TASK_UNINTERRUPTIBLE**.
- En general, se recomienda evitar su uso siempre que sea posible ya que puede derivar en deadlocks.

Sleep y demoras

```
static DECLARE_WAIT_QUEUE_HEAD (mi_queue);
```

- Existe otra versión de **wait_event** que pone al proceso en estado **TASK_UNINTERRUPTIBLE**.
- En general, se recomienda evitar su uso siempre que sea posible ya que puede derivar en deadlocks.
- Sin embargo es particularmente útil cuando vamos a escribir en un dispositivo, ya que asegura que la transferencia de todos los datos se realizará en forma atómica.

Sleep y demoras

```
static DECLARE_WAIT_QUEUE_HEAD (mi_queue);
```

- Existe otra versión de **wait_event** que pone al proceso en estado **TASK_UNINTERRUPTIBLE** .
- En general, se recomienda evitar su uso siempre que sea posible ya que puede derivar en deadlocks.
- Sin embargo es particularmente útil cuando vamos a escribir en un dispositivo, ya que asegura que la transferencia de todos los datos se realizará en forma atómica.
- Esto último es particularmente útil en dispositivos requieren transferencias de bloques de datos.

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

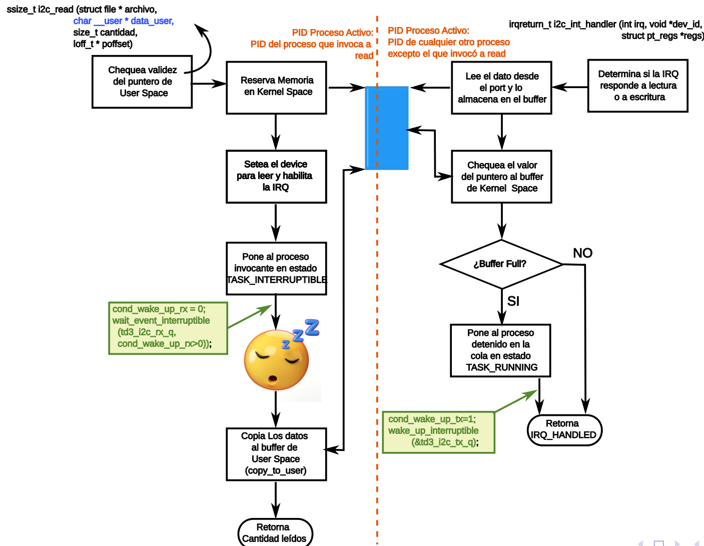
Donde se bloquea el proceso invocante?

```
/* Definimos cola de espera para lectura */
static DECLARE_WAIT_QUEUE_HEAD (td3_i2c_rx_q);

ssize_t i2c_read (struct file * archivo, char __user * data_user, size_t
cantidad, loff_t * poffset)
{
  /*Se pide memoria, chequean los punteros de User Space,
  y dem|\color{Gray}\'a|s inicializaciones*/
  cond_wake_up_rx = 0;  /*Variable de condici|\color{Gray}\'o|n*/
  /*Se prepara al device para el acceso a los datos*/

  wait_event_interruptible (td3_i2c_rx_q,cond_wake_up_rx>0)
  /* El proceso queda TASK_INTERRUPTIBLE*/

  /*Cuando se debloquea, se analiza si el buffer se ha
  completado y en tal caso lo copia a modo user y regresa*/
}
```

Donde y cuando se lo desbloquea?

```
irqreturn_t i2c_int_handler (int irq, void *dev_id, struct pt_regs *regs)
{
  /*Determina si la interrupci|\color{Gray}\'o|n obedece a lectura,
  o a escritura*/
  /*Si es de lectura lee el dato desde el device*/
  Puntero_Buffer_KernelSpace = ioread32 (register_Address);
  Puntero_Buffer_KernelSpace++;
  if (Puntero_Buffer_KernelSpace == Buffer_KernelSpace_size)
  {
   /*configura el hardware para dejar de adquirir datos*/
    cond_wake_up_rx=1;
    wake_up_interruptible(&td3_i2c_rx_q);  /*El proceso pasa a TASK_RUNNING*/
  }
  return IRQ_HANDLED
}
```

Secuencia completa



Observaciones

Observaciones

- El proceso que invoca a read es suspendido en su ejecución hasta que se adquieran los datos

Observaciones

- El proceso que invoca a read es suspendido en su ejecución hasta que se adquieran los datos
- Esto ocurre siempre (aun cuando leemos un archivo de datos en disco)

Observaciones

- El proceso que invoca a read es suspendido en su ejecución hasta que se adquieran los datos
- Esto ocurre siempre (aun cuando leemos un archivo de datos en disco)
- El proceso invocante es retirado de la lista de ejecución del scheduler

Observaciones

- El proceso que invoca a read es suspendido en su ejecución hasta que se adquieran los datos
- Esto ocurre siempre (aun cuando leemos un archivo de datos en disco)
- El proceso invocante es retirado de la lista de ejecución del scheduler
- El handler de la interrupción corre entonces en el contexto de cualquier otro proceso, excepto del invocante, el que nunca será schedulado hasta ser despertado

Observaciones

- El proceso que invoca a read es suspendido en su ejecución hasta que se adquieran los datos
- Esto ocurre siempre (aun cuando leemos un archivo de datos en disco)
- El proceso invocante es retirado de la lista de ejecución del scheduler
- El handler de la interrupción corre entonces en el contexto de cualquier otro proceso, excepto del invocante, el que nunca será schedulado hasta ser despertado
- Justamente es el handler de la interrupción quien pone el proceso invocante en estado de ejecución recién cuando verifica que se han adquirido la totalidad de los datos demandada por aquel.

Temario

1. Conceptos Básicos de acceso al Hardware
   - Conceptos previos
   - Devices Drivers
2. Módulos de Kernel
   - Introducción
   - HowTo
3. Linux Driver Model
   - Motivación
   - Recursos relacionados
4. Linux Device Drivers en plataformas ARM
   - Cuando lo diverso se trasformó en un problema
5. Device Tree
   - Punto de partida
   - Introducción del Device Tree
   - Funcionamiento del Device Tree
   - Device Tree en Beagle Bone Black
   - Device Tree Overlay
6. Char Devices
   - Conceptos iniciales
   - Representación interna de Números de device. Estructuras y Macros
   - Funciones para gestión de Major y Minor Numbers
   - Registro del char dev
7. Implementando POSIX (everything is a file)
   - Implementación de las funciones del driver
8. Cuestiones adicionales asociadas al dispositivo
   - Platform drivers
   - Acceso a Registros
   - Manejo de Interrupciones
   - Sleeping Demoras y demás ...
   - Donde bloquear y desbloquear
9. Resumen de un char dev
   - Pasos recursos y funciones

Capacidades y estructuras

Capacidades y estructuras

1. En principio necesitamos declarar las estructuras y datos necesarios. Conviene utilizar el modificador `static`. Resumiendo lo ya visto, sería:

```
static struct file_operations mydev_ops = {
    .owner       = THIS_MODULE,
    .read        = mydev_read,
    .write       = mydev_write,
    .open        = mydev_open,
    .release     = mydev_close,
};
static const struct of_device_id mydev_of_match[] = {
    { .compatible = "td3,omap4-mydev" },
    {},
};
MODULE_DEVICE_TABLE(of, i2c_of_match);
static struct platform_driver mydev_driver = {
     .probe                  = mydev_probe,
     .remove                 = mydev_remove,
     .driver                 = {
          .name              = "td3_mydev",
          .of_match_table    = of_match_ptr(mydev_of_match),
    },
};
```

Capacidades y estructuras

2. Declaración de todas las estructuras necesarias para insertar al device en l estructura jerárquica del LDM.
   Las estructuras **class** y **dev** cumplen esa función

```
static dev_t    dispo;
static struct   cdev * mydev_cdev;
static struct   class * pdev_class;
static struct   device * pdev_dev;
static struct   mydev_data_t mydev_data;


static int cond_wake_up_rx=0;
static int cond_wake_up_tx=0;
/*Wait queues (suponen un dispositivo que bloquea al recibir y al transmitir)*/
static DECLARE_WAIT_QUEUE_HEAD (mydev_rx_q);
static DECLARE_WAIT_QUEUE_HEAD (mydev_tx_q);
```

Función `module_init`

3. Inicializar `struct cdev` utilizando como ya se mostró las funciones `cdev_alloc()` y `alloc_chrdev_region()`

Función `module_init`

3. Inicializar `struct cdev` utilizando como ya se mostró las funciones `cdev_alloc()` y `alloc_chrdev_region()`
4. Registrar el device en el sistema con `cdev_add()` (ya visto).

Función `module_init`

3. Inicializar `struct cdev` utilizando como ya se mostró las funciones `cdev_alloc()` y `alloc_chrdev_region()`
4. Registrar el device en el sistema con `cdev_add()` (ya visto).
5. Registrar la clase y el device en el kernel

```
pdev_class = class_create(THIS_MODULE, "myclassdev");
if (IS_ERR(pdev_class))
{
    cdev_del (mydev_cdev);
    unregister_chrdev_region(dispo, CANT_DISP);
    return PTR_ERR(pdev_class);
}
/*Creo el device dentro de la clase*/
pdev_dev = device_create(pdev_class, NULL, dispo, NULL,"td3_mydev");
if (IS_ERR(pdev_dev))
{
    class_destroy(pdev_class);
    cdev_del (mydev_cdev);
    unregister_chrdev_region(dispo, CANT_DISP);
    return PTR_ERR(pdev_dev);
}
```

Función `module_init`

3. Inicializar `struct cdev` utilizando como ya se mostró las funciones `cdev_alloc()` y `alloc_chrdev_region()`
4. Registrar el device en el sistema con `cdev_add()` (ya visto).
5. Registrar la clase y el device en el kernel

```
pdev_class = class_create(THIS_MODULE, "myclassdev");
if (IS_ERR(pdev_class))
{
   cdev_del (mydev_cdev);
   unregister_chrdev_region(dispo, CANT_DISP);
   return PTR_ERR(pdev_class);
}
/*Creo el device dentro de la clase*/
pdev_dev = device_create(pdev_class, NULL, dispo, NULL,"td3_mydev");
if (IS_ERR(pdev_dev))
{
   class_destroy(pdev_class);
   cdev_del (mydev_cdev);
   unregister_chrdev_region(dispo, CANT_DISP);
   return PTR_ERR(pdev_dev);
}
```

6. Registrar el platform device: `platform_driver_register()`

Notas para `module_init`

Las funciones del kernel que retornan un tipo `int` por lo general si fallan retornan valores negativos. Las que retornan punteros devuelven `NULL` cuando fallan.
Es necesario entonces evaluar su valor de retorno y obrar en consecuencia.
Para ello en cada manejador de error debe deshacerse todo lo hecho previamente en la instalación.
En el caso de funciones que retornan punteros el kernel provee tres funciones auxiliares:
`void *ERR_PTR(long error);` Retorna el valor de error actual como puntero.
`long IS_ERR(const void *ptr);` Chequea si el valor retornado es un puntero error.
`long PTR_ERR(const void *ptr);` Retorna el puntero erróneo actual como código de error

Función `module_exit`

- No hay mucho para decir en esta sección

Función `module_exit`

- No hay mucho para decir en esta sección
- Simplemente en el orden inverso al del registro en `module_init` , se debe desregistrar, platform device, el device, la clase, destruir la estructura `cdev` , y finalmente devolver los `MAJOR` and —codeMINOR Numbers.

Función `module_exit`

- No hay mucho para decir en esta sección
- Simplemente en el orden inverso al del registro en `module_init` , se debe desregistrar, platform device, el device, la clase, destruir la estructura `cdev` , y finalmente devolver los `MAJOR` and —codeMINOR Numbers.
- No es mas que invocar a las mismas funciones que utilizamos durante la inicialización en los handlers de error.

Función `module_exit`

- No hay mucho para decir en esta sección
- Simplemente en el orden inverso al del registro en `module_init` , se debe desregistrar, platform device, el device, la clase, destruir la estructura `cdev` , y finalmente devolver los `MAJOR` and —codeMINOR Numbers.
- No es mas que invocar a las mismas funciones que utilizamos durante la inicialización en los handlers de error.
- Solo `platform_driver_unregister (&mydriver_driver);` que invocada al principio desregistra el platform device.

Funciones `probe` y `remove`

1. Obtener la `VIRQ`

Funciones `probe` y `remove`

1. Obtener la `VIRQ`
2. Instalar el handler de interrupción

Funciones `probe` y `remove`

1. Obtener la `VIRQ`
2. Instalar el handler de interrupción
3. Especificar el área física de direccionamiento del device invocando a `ioremap()`

Funciones `probe` y `remove`

1. Obtener la `VIRQ`
2. Instalar el handler de interrupción
3. Especificar el área física de direccionamiento del device invocando a `ioremap()`
4. Configurar algún aspecto general de la plataforma a la que está conectado dispositivo de E/S por medio de `(ioread ()` e `iowrite()`

Funciones `probe` y `remove`

1. Obtener la `VIRQ`
2. Instalar el handler de interrupción
3. Especificar el área física de direccionamiento del device invocando a `ioremap()`
4. Configurar algún aspecto general de la plataforma a la que está conectado dispositivo de E/S por medio de `(ioread ()` e `iowrite()`
5. En la función `remove()` se efectúa la devolución de la IRQ y el desmapeo (`iounmap()`) de los espacios físicos de direccionamiento asignados en `prove()`

Funciones `open` y `release`

1. `open()` Inicializa en detalle el dispositivo.

Funciones `open` y `release`

1. `open()` Inicializa en detalle el dispositivo.
2. Inicializa partes del SoC relacionadas con el dispositivo necesarias para su operación adecuada (pinmux por ejemplo en el Sitara 3358).

Funciones `open` y `release`

1. `open()` Inicializa en detalle el dispositivo.
2. Inicializa partes del SoC relacionadas con el dispositivo necesarias para su operación adecuada (pinmux por ejemplo en el Sitara 3358).
3. Reservar memoria si corresponde (al menos un buffer mínimo como para empezar la operación

Funciones `open` y `release`

1. `open()` Inicializa en detalle el dispositivo.
2. Inicializa partes del SoC relacionadas con el dispositivo necesarias para su operación adecuada (pinmux por ejemplo en el Sitara 3358).
3. Reservar memoria si corresponde (al menos un buffer mínimo como para empezar la operación
4. En la función `release()` se efectúa la devolución de la memoria reservada en `open()`

Funciones `open` y `release`

1. `open()` Inicializa en detalle el dispositivo.
2. Inicializa partes del SoC relacionadas con el dispositivo necesarias para su operación adecuada (pinmux por ejemplo en el Sitara 3358).
3. Reservar memoria si corresponde (al menos un buffer mínimo como para empezar la operación
4. En la función `release()` se efectúa la devolución de la memoria reservada en `open()`
5. Ambas funciones si todo está correcto devuelven 0. El kernel completará el retorno al programa de User Space con los valores adecuados.

¿Preguntas?