

Anexo A

PLATAFORMA DE DESARROLLO ECBOT

A.1. Introducción

En esta sección se realizará una explicación detallada del proceso de adaptación de Linux a la familia de plataformas ECB_AT91, este proceso es aplicable a otros dispositivos que trabajen con procesadores soportados por la distribución de Linux. Adicionalmente, se explicará de forma detallada el funcionamiento de este sistema operativo, el proceso de arranque y su puesta en marcha, así como las principales distribuciones, aplicaciones y librerías disponibles para el desarrollo de aplicaciones. Esta y otra información recolectada durante más de tres años permite que la industria y la academia desarrollen aplicaciones comerciales utilizando herramientas de diseño modernas ¹.

A.2. Herramientas abiertas para diseño de sistemas embebidos

A.2.1. Herramientas de Desarrollo

En este apartado se describirán las herramientas abiertas necesarias para el desarrollo de aplicaciones software en sistemas sistemas digitales. Todas las aplicaciones mencionadas a continuación hacen parte de la cadena de herramientas GNU, que son parte de los recursos suministrados por la comunidad de software libre.

GNU binutils

Colección de utilidades para archivos binarios y están compuestas por:

- **addr2line** Convierte direcciones de un programa en nombres de archivos y números de línea. Dada una dirección y un ejecutable, usa la información de depuración en el ejecutable para determinar que

¹A lo largo de este capítulo se utilizarán cuadros de texto donde se listará código fuente en color gris, instrucciones ejecutadas en el computador personal en color verde y operaciones que se ejecutan en el sistema embebido en color amarillo

nombre de archivo y número de línea está asociado con la dirección dada.

- **ar** Esta utilidad crea, modifica y extrae desde ficheros; un fichero es una colección de otros archivos en una estructura que hace posible obtener los archivos individuales.
- **as** Utilidad que compila la salida del compilador de C (GCC).
- **c++filt** Este programa realiza un mapeo inverso: Decodifica nombres de bajo-nivel en nombres a nivel de usuario, de tal forma que el *linker* pueda mantener estas funciones sobrecargadas (overloaded) “from clashing”.
- **gasp** GNU Assembler Macro Preprocessor
- **ld** El *linker* GNU combina un número de objetos y ficheros, re-localiza sus datos y los relaciona con referencias. Normalmente el último paso en la construcción de un nuevo programa es el llamado a *ld*.
- **nm** Realiza un listado de símbolos de archivos tipo objeto.
- **objcopy** Copia los contenidos de un archivo tipo objeto a otro. *objcopy* utiliza la librería GNU BFD para leer y escribir el archivo tipo objeto. Permite escribir el archivo destino en un formato diferente al del archivo fuente.
- **objdump** Despliega información sobre archivos tipo objeto.
- **ranlib** Genera un índice de contenidos de un fichero, y lo almacena en él.
- **readelf** Interpreta encabezados de un archivo ELF.
- **size** Lista el tamaño de las secciones y el tamaño total de un archivo tipo objeto.
- **strings** Imprime las secuencias de caracteres imprimibles de al menos 4 caracteres de longitud.
- **strip** Elimina todos los símbolos de un archivo tipo objeto.

Compilador

El *GNU Compiler Collection* normalmente llamado GCC, es un grupo de compiladores de lenguajes de programación producido por el proyecto GNU. Es el compilador estándar para el software libre, de los sistemas operativos basados en Unix y algunos propietarios como Mac OS de Apple. Soporta los lenguajes ADA, C, C++, Fortran, Java, Objective-C, Objective-C++ para las arquitecturas Alpha, ARM, Atmel AVR, Blackfin, H8/300, System/370, System/390, IA-32 (x86), x86-64, IA-64 i.e. the “Itanium”, Motorola 68000, Motorola 88000, MIPS, PA-RISC, PDP-11, PowerPC, SuperH, SPARC, VAX, Renesas R8C/M16C/M32C y MorphoSys. Gracias a esto puede considerarse como una herramienta universal para el desarrollo de sistemas embebidos, el código escrito en una plataforma (en un lenguaje de alto nivel) puede ser implementado en otra sin mayores cambios, esto elimina la dependencia entre el código fuente y el procesador (re-utilización de código), lo que no es posible cuando se utiliza el lenguaje ensamblador.

GNU Debugger

El depurador oficial de GNU (GDB) al igual que GCC, soporta múltiples lenguajes y plataformas; permite monitorear y modificar las variables internas del programa y hacer llamado a funciones de forma independiente a la ejecución normal del mismo. Además, permite establecer sesiones remotas utilizando el puerto serie o TCP/IP. Aunque GDB es una aplicación que se ejecuta en consola de comandos, se han desarrollado varios front-ends como DDD o GDB/Insight.

Librerías C

Es necesario contar con las librerías standard de C: `stdio`, `stdlib`, `math`, etc; las más utilizadas en sistemas embebidos son:

- **glibc** Es la librería C oficial del proyecto GNU; el principal inconveniente al trabajar con esta librería en sistemas embebidos es que genera ejecutables de mayor tamaño que los generados a partir de otras librerías, lo cual no la hace muy atractiva para este tipo de aplicaciones.
- **uClibc** Es una librería diseñada especialmente para sistemas embebidos, es mucho más pequeña que **glibc**.
- **newlib** Al igual que **uClibc**, está diseñada para sistemas embebidos. El típico “Hello, world!” ocupa menos de 30k en un entorno basado en newlib, mientras que en uno basado en glibc, puede ocupar 380k.
- **diet libc** Es una versión de *libc* optimizada en tamaño, puede ser utilizada para crear ejecutables estáticamente enlazados para Linux en plataformas alpha, arm, hppa, ia64, i386, mips, s390, sparc, sparc64, ppc y x86_64.

A.2.2. Herramientas CAD

Para la elaboración de las cajas que contendrán las tarjetas electrónicas y servirán de soporte físico a los dispositivos electrónicos, se utilizará la herramienta *QCAD* disponible para los sistemas operativos más utilizados; con esta aplicación se puede generar la forma de la placa de circuito impreso con los orificios necesarios para ajustarse a la caja y a la aplicación; este archivo puede ser exportado a la herramienta *KICAD* lo que facilita la colocación de componentes.

A.2.3. Software de ayuda al diseño

Para la realización de las actividades descritas anteriormente se utilizará software abierto, que pueda ser descargado libremente sin restricción alguna; esto, con el fin de proporcionar al estudiante herramientas software legales que le permitan adquirir las habilidades descritas en el capítulo ?? y puedan ser utilizadas en su ejercicio profesional. Se debe tener presente que además de proporcionar a los estudiantes los conocimientos necesarios en su profesión, también se deben hacer respetar los derechos de autor y la propiedad intelectual de las herramientas software comerciales; en muchas universidades se tolera el uso de software ilegal por parte de los estudiantes, debido a que es necesario para el desarrollo del curso y/o el establecimiento no cuenta con recursos para adquirir las licencias suficientes. Aunque el uso de este software conseguido de forma ilegal es muy loable, no deja de ser una violación a las leyes que protegen los derechos de sus creadores; adicionalmente, formar a estudiantes con herramientas costosas los vuelve dependientes de ellas, lo que requerirá de grandes inversiones en software cuando ellos formen sus propias empresas, lo que puede desalentar este tipo de iniciativas. Por lo anterior, se hizo una revisión y prueba de las herramientas abiertas disponibles que pueden ser utilizadas en los cursos de diseño digital y que proporcionan la funcionalidad requerida en esta propuesta académica.

Análisis y simulación de circuitos

En la mayoría de centros de formación se utilizan herramientas que utilizan el simulador SPICE (Simulation Program with Integrated Circuits Emphasis) el cual ha demostrado ser una herramienta muy

útil en diferentes áreas (electrónica analógica, electrónica digital, diseño VLSI, electrónica de potencia, entre otras) ya que proporciona diferentes formas y niveles de simulación; permitiendo el uso de modelos para representar de la mejor forma posible un dispositivo real. SPICE fue desarrollado por el departamento de ingeniería eléctrica y ciencias de la computación de la universidad de Berkeley ² y su código fuente es distribuido bajo la licencia BSD (Berkeley Software Distribution), la última versión disponible es la 3f5, gracias a esto se han desarrollado una serie de herramientas que utilizan este potente simulador y proporcionan una interfaz amigable para el usuario.

En esta propuesta se utiliza *QUCS* (Quite Universal Circuit Simulator) desarrollado en QT y permite ser ejecutado en cual sistema operativo. QUCS permite realizar análisis AC, DC, transitorio, de balance armónico y soporta dispositivos lineales, no lineales, micro-cintas, coplanares, líneas de transmisión y representados por modelos SPICE. En la figura A.1 se muestra la captura de esta aplicación.

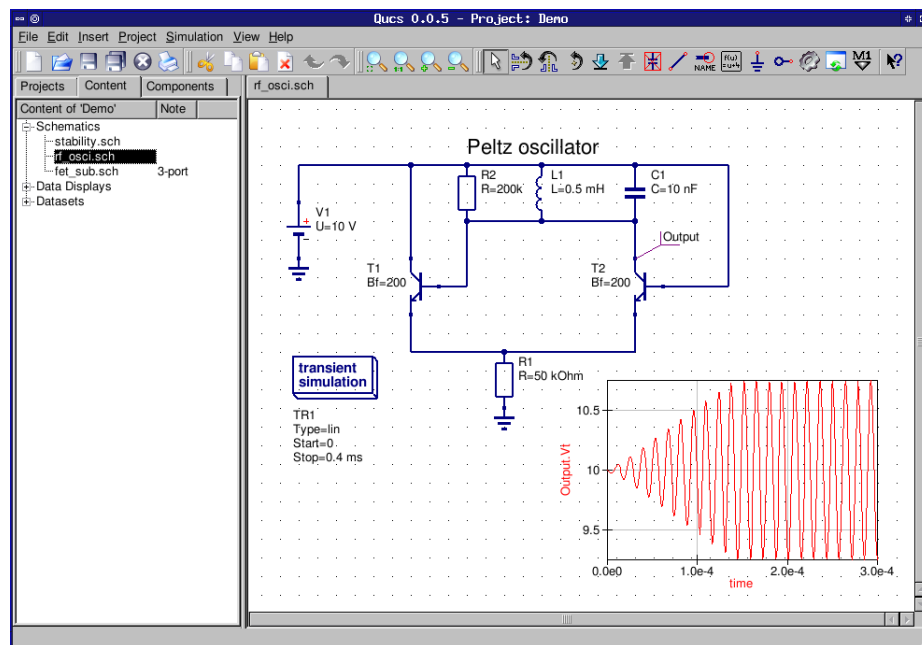


Figura A.1: Simulador QUCS

Síntesis, simulación y verificación digital

Para la síntesis digital a partir de lenguajes de descripción de hardware se utilizan las herramientas gratuitas suministradas por los fabricantes de FPGAs, *webpack* de Xilinx y *Quartus* de Altera; debido a que la estructura interna de las FPGAs solo la conocen los fabricantes³, es obligatorio utilizar sus herramientas para obtener el archivo de configuración.

Para la simulación de sistemas digitales que utilizan como entrada de diseño lenguajes de descripción de hardware existen los simuladores *ICARUS* para verilog y *GHD* para vhd; los dos pueden ser utilizados

²<http://http://bwrc.eecs.berkeley.edu/Classes/IcBook/SPICE/>

³Es posible obtener información valiosa de sus patentes por ejemplo: <http://www.freepatentsonline.com/6301693.pdf>, con este tipo de información varios proyectos buscan generar herramientas abiertas de síntesis

para realizar simulaciones funcionales, post síntesis o post place & route (trabajando en conjunto con las herramientas de los fabricantes) y ambos soportan el formato de salida VCD (definido junto con el lenguaje de descripción de hardware verilog por el estándar IEEE 1364-2001). Adicionalmente, estas herramientas pueden ser utilizadas en los sistemas operativos más utilizados.

Como herramienta de simulación se utilizará *GTKWAVE*, la cual acepta como entrada archivos en formato *VCD* y puede ser ejecutada en MAC, Linux y Windows. *GTKWAVE* realiza un manejo adecuado de la jerarquía del sistema bajo análisis, permitiendo observar todas las señales de los diferentes módulos que componen la jerarquía superior, lo que es muy útil en este tipo de simulaciones; en la figura A.2 se puede observar una captura de esta herramienta.

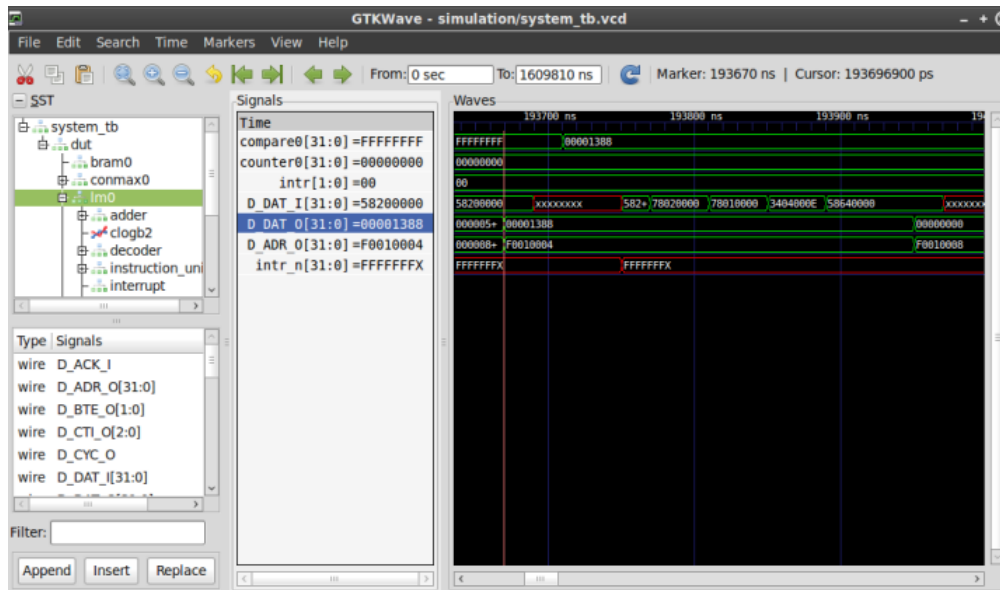


Figura A.2: Visualizador de formas de onda *GTKWAVE*

Una vez configurada la FPGA, se utiliza la herramienta *URJTAG* para verificar el correcto funcionamiento del sistema implementado en la FPGA, *URJTAG* proporciona una capa de abstracción de hardware que permite el manejo del puerto JTAG de cualquier dispositivo, proporcionando funciones de alto nivel para la aplicación de las funciones JTAG (IDCODE, INTEST, EXTEST, BYPASS, SAMPLE/PRELOAD) permitiendo la aplicación de vectores de prueba al núcleo lógico de la FPGA y la captura de la respuesta a estos estímulos; estas pruebas se realizan a baja frecuencia. *URJTAG* soporta varias interfaces físicas para control de las señales del puerto JTAG (TDI, TDO, TMS y TCK) las cuales pueden ser conectadas a diferentes puertos de un computador, o como en este caso a un puerto virtual creado en el procesador MIPS.

Herramientas para la configuración de PLDs

Aunque con la plataforma de desarrollo SIE no es necesario utilizar herramientas ni unidades de programación adicionales para configurar su FPGA; el procesador de SIE ejecuta dos aplicaciones que son utilizadas para realizar esta función y pueden ser ejecutadas en computadores personales: *URJTAG* y *XC3SPROG*, las dos funcionan de forma similar, utilizan dispositivos conectados al puerto paralelo

(conexión directa) o USB (basados en el protocolo MPSSE del chip FT2232) del computador; para ejecutar las instrucciones extendidas de la FPGA CFG_OUT, CFG_IN, JSTART y JPROGRAM (hasta el momento solo han sido probadas con las FPGAs de Xilinx).

Diseño de placas de circuito impreso

Este tipo de herramientas han sido las más difíciles de conseguir ya que al comienzo de este estudio no existía una herramienta que permitiera realizar de forma fácil la elaboración del esquemático, asignación de *footprints*, distribución y localización de componentes en el layout, ruteo de las señales, y verificación de reglas de diseño tanto en el esquemático como en el layout. En la actualidad, el proyecto *KICAD* ha alcanzado un nivel adecuado para ser utilizado en el desarrollo de aplicaciones comercializables y fue utilizado en todos los proyectos académicos durante la aplicación del presente plan de estudios; está formado por 5 aplicaciones: el editor de esquemáticos *Eeschema*; el editor de circuitos impresos *Pcbnew*; el visor de archivos gerber *Gerbview*; la utilidad para asignar footprints *Cvpcb* y el manejador de proyectos *Kicad*.

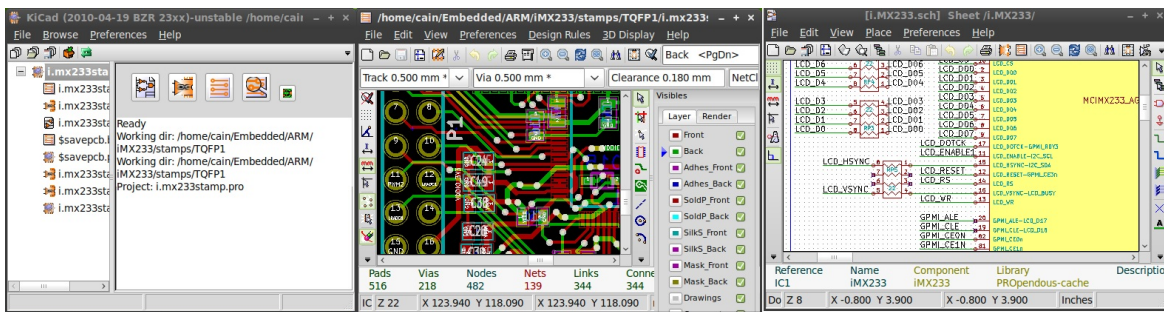


Figura A.3: Herramienta para la elaboración de circuitos impresos *KICAD*

A.2.4. Discusión

Las aplicaciones mencionadas en esta sección han sido probadas por estudiantes de las universidades: UNAL, ULA, UIS y por la empresa emQbit, por esto se puede afirmar que este grupo de herramientas soporta todo el proceso de diseño de un sistema embebido; su carácter abierto y gratuito permite ahorrar mucho dinero si se compara con el requerido para comprar herramientas propietarias; el único inconveniente de las herramientas abiertas es que en algunos casos su interfaz no es tan amigable y no cuentan con las funcionalidades de su contraparte comercial. Una ventaja adicional es la no utilización de software ilegal, práctica que ha sido tolerada por la academia durante muchos años; adicionalmente, se le enseña al estudiante una alternativa que le permite ahorrar costos de inversión en el software necesario para comenzar una nueva empresa; con lo que este dinero se puede destinar a compra de equipo necesario para el desarrollo de la actividad.

Aunque este grupo de herramientas pueden ejecutarse en los sistemas operativos mas populares (Linux, Mac OS y Windows) se prefiere el uso de Linux ya que es un sistema operativo gratuito y no es necesario pagar ningún tipo de licencia, lo que reduce aún más la inversión en software.

A.3. Métodos de arranque

Como se vio anteriormente, la mayoría de los SoC utilizados en aplicaciones modernas no poseen memorias programables en su interior. Todo SoC debe ser programado para que pueda ejecutar una determinada tarea; este programa debe estar almacenado en una memoria no volátil externa y debe estar en el formato requerido por el procesador. Normalmente los SoCs proporcionan varios caminos (habilitando diferentes periféricos) para hacer esto. Un programa de inicialización (*boot program*) contenido en una pequeña ROM del SoC se encarga de configurar, y revisar ciertos periféricos en búsqueda de un ejecutable válido, una vez lo encuentra, lo copia a la memoria RAM interna y lo ejecuta desde allí. No sobra mencionar que este ejecutable debe estar enlazado de tal forma que todas sus secciones se encuentren en el espacio de la memoria RAM interna (0x0 después del REMAP⁴).

El AT91RM9200 posee una memoria interna SRAM de 16 kbytes. Después del *reset* esta memoria esta disponible en la posición 0x200000; después del remap esta memoria se puede acceder en la posición 0x0. Algunos fabricantes, en la etapa de producción graban en las memorias no volátiles las aplicaciones definitivas, y sueldan en la placa de circuito impreso los dispositivos programados, esto es muy conveniente cuando se trabaja con grandes cantidades ya que ahorra tiempo en el montaje de los dispositivos.

El programa de inicialización del AT91RM9200 (se ejecuta si el pin BMS se encuentra en un valor lógico alto) busca una secuencia de 8 vectores de excepción válidos en la DataFlash conectada al puerto SPI, en una EEPROM conectada a la interfaz I2C o en una memoria de 8 bits conectada a la interfaz de bus externo (EBI). Estos vectores son instrucciones LDR o Bbranch, a excepción de la sexta instrucción (posición 14 a 17) que contiene información sobre el tamaño de la imagen (en bytes) a descargar y el tipo de dispositivo *DataFlash*. Si la secuencia es encontrada, el código es almacenado en la memoria SRAM interna y se realiza un *remap* (con lo que la memoria interna SRAM es accesible en la posición 0x0 ver Figura A.4).

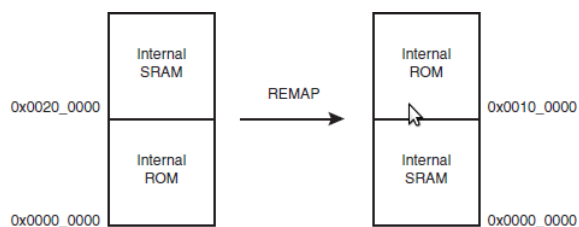


Figura A.4: Diagrama de flujo del programa de inicialización del SoC AT91RM9200

Si no se encuentra esta secuencia de vectores, se inicializa un programa que configura el puerto serial de depuración (DBGU) y el puerto USB Device. Quedando en espera de la descarga de una aplicación a través del protocolo DFU (Digital Firmware Upgrade) por el puerto USB o con el protocolo XMODEM en el puerto serial DBGU (115200,N81). La figura A.5 muestra el diagrama de flujo del programa de inicialización del SoC AT91RM9200.

El programa descargado a la memoria SRAM interna debe: programar una memoria no volátil (DataFlash SPI para la familia de plataformas ECB_AT91); proporcionar un canal de comunicación que permita descargar ejecutables más grandes, e inicializar el controlador de memoria SDRAM para que almacene temporalmente el ejecutable a grabar, esto debido a que la memoria interna del AT91RM9200 es de 16kBytes. Más adelante hablaremos detalladamente de la aplicación que realiza estas funciones.

⁴Los procesadores ARM pueden intercambiar el sitio de la memoria RAM interna y la memoria no volátil

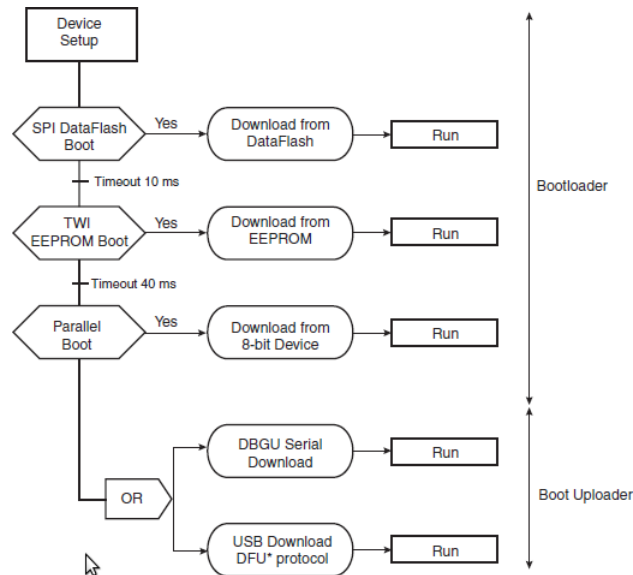


Figura A.5: Diagrama de flujo del programa de inicialización del SoC AT91RM9200

A.3.1. Interfaz JTAG

A mediados de los 70s, la estructura de pruebas para tarjetas de circuito impreso (PCB, Printed Circuit Boards) se basaba en el uso de la técnica “bed-of-nails”. Este método utilizaba un dispositivo que contenía una gran cantidad de puntos de prueba, que permitían el acceso a dispositivos en la tarjeta a través de puntos colocados en la capa de cobre para dicho fin. Las pruebas se realizaban en dos fases: con el circuito apagado y con el circuito funcionando. Con la aparición de los dispositivos de montaje superficial se empezó a colocar dispositivos en las dos caras de la tarjeta, y se redujeron de forma considerable las dimensiones de los dispositivos, disminuyendo la distancia física entre las interconexiones (0.4 - 1mm), dificultando el proceso de pruebas tradicional.

A mediados de los 80s un grupo de ingenieros de pruebas miembros de compañías electrónicas europeas se reunieron para examinar el problema y buscar posibles soluciones. Este grupo se auto-denominó JETAG (Joint European Test Action Group). El método de solución propuesto por ellos estaba basado en el concepto de un registro de corrimiento serial colocado alrededor de la frontera dispositivo, de aquí el nombre “Boundary Scan”. Después el grupo se asoció a compañías norteamericanas y la “E” de “European” desapareció del nombre de la organización convirtiéndose en JTAG (Join Test Action Group).

Arquitectura BOUNDARY SCAN

A cada señal de entrada o salida se le adiciona un elemento de memoria multi-propósito llamado “Boundary Scan Cell” (BSC). Las celdas conectadas a los pines de entrada reciben el nombre de “Celdas de entrada”, y las que están conectadas a los pines de salida “Celdas de salida”. En la Figura A.6 se muestra esta arquitectura.

Las BSC se configuran en un registro de corrimiento de entrada y salida paralela. Una carga paralela de los registros (captura) ocasiona que los valores de las señales aplicadas a los pines del dispositivo pasen

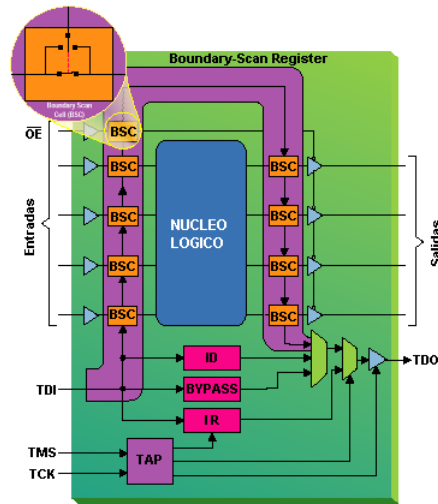


Figura A.6: Arquitectura Boundary Scan

a las celdas de entrada y que opcionalmente los valores de las señales internas del dispositivo pasen a las celdas de salida. Una descarga paralela (Actualización) ocasiona que los valores presentes en las celdas de salida pasen a los pines del dispositivo, y opcionalmente los valores almacenados en las celdas de entrada pasen al interior del dispositivo.

Los datos pueden ser corridos a través del registro de corrimiento de forma serial, empezando por un pin dedicado TDI (Test Data In) y terminando en un pin de salida dedicado llamado TDO (Test Data Out). La señal de reloj se proporciona por un pin externo TCLK (Test Clock) y el modo de operación se controla por la señal TMS (Test Mode Select). Los elementos del Boundary Scan no afectan el funcionamiento del dispositivo. Y son independientes del núcleo lógico del mismo.

Instrucciones JTAG

El Standard IEEE 1149.1 describe tres instrucciones obligatorias: Bypass, Sample/Preload, y Extest [1].

- **BYPASS** Esta instrucción permite que el chip permanezca en un modo funcional, hace que el registro de Bypass se coloque entre TDI y TDO; permitiendo la transferencia serial de datos a través del circuito integrado desde TDI hacia TDO sin afectar la operación. La codificación en binario para esta instrucción debe ser con todos sus bits en uno.
- **SAMPLE/PRELOAD** Esta instrucción selecciona coloca el registro Boundary-Scan entre los terminales TDI y TDO. Durante esta instrucción, se puede acceder al registro Boundary-Scan y obtener una muestra de los datos de entrada y salida del chip a través de la operación *Data Scan*. Esta instrucción también se utiliza para precargar los datos de prueba en el registro Boundary-Scan, antes de ejecutar la instrucción EXTEST. La codificación de esta instrucción la define el fabricante.
- **EXTEST** Esta instrucción coloca al circuito integrado en modo de test externo (pruebas de interconexión) y conecta el registro Boundary-Scan entre TDI y TDO. Las señales que salen del

circuito son cargadas en el registro boundary-scan en el flanco de bajada de TCK del estado Capture-DR; las señales de entrada al dispositivo son cargadas al registro boundary-scan durante el flanco de bajada de TCK dl estado Update-DR (ver Figura A.7). La codificación para esta instrucción está definida con todos sus bits en cero.

- **INTEST** La instrucción INTEST (opcional) selecciona el registro boundary-scan, pero es utilizado para capturar las señales que salen del núcleo lógico del dispositivo, y para aplicar valores conocidos a las señales de entrada del núcleo. La codificación para esta señal es asignada por el diseñador.

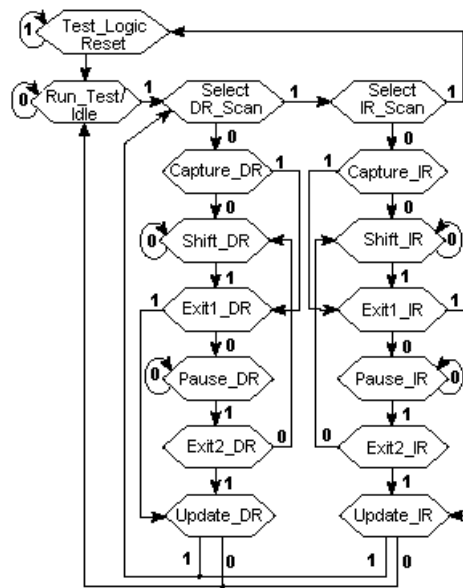


Figura A.7: Arquitectura Boundary Scan

A.4. Flujo de diseño software

En la figura A.8 se ilustra la secuencia de pasos que se realizan desde la creación de un archivo de texto que posee el código fuente de una aplicación hasta su implementación en la tarjeta de desarrollo. Los pasos necesarios para generar un ejecutable para un sistema embebido son:

1. **Escritura del código fuente:** Creación del código fuente en cualquier editor de archivos de texto.
2. **Compilación:** Utilizando GCC se compila el código fuente; el compilador busca en los encabezados (*headers .h*) de las librerías la definición de una determinada función, pero no busca el segmento de código donde está implementada. (por ejemplo el *printf* en el archivo *stdio.h*). Como resultado de este paso se obtiene un archivo binario tipo objeto que incluye el código necesario para realizar la funcionalidad deseada utilizando en conjunto de instrucciones del procesador.
3. **Enlazado:** En esta etapa se realizan dos tareas:

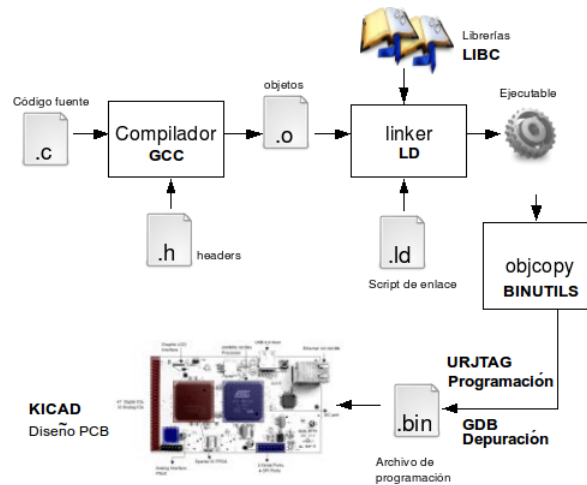


Figura A.8: Flujo de diseño SW utilizando la cadena de herramientas GNU

- a) Se enlazan los archivos tipo objeto del proyecto junto con las librerías, si una determinada función no es definida por ninguna de las librerías pasadas como parámetro al enlazador (*linker*), este generará un error y no se generará el ejecutable.
 - b) Se definen la posiciones físicas de las secciones del ejecutable (tipo ELF), esto se realiza a través de un *script de enlazado* que define de forma explícita su localización.
4. **Extracción del archivo de programación** En algunas aplicaciones es necesario extraer únicamente las secciones que residen en los medios de almacenamiento no volátil y eliminar las demás secciones del ejecutable. Esto se realiza con la herramienta *objcopy*, la cual, permite generar archivos en la mayoría de los formatos soportados por los programadores de memorias y procesadores, como por ejemplo S19 e Intel Hex. Adicionalmente se puede generar un archivo binario que contiene las instrucciones en lenguaje del procesador, y pueden ser descargadas directamente a la memoria de la plataforma.
5. **Descarga del programa.** Dependiendo de la plataforma, existen varios métodos para descargar el archivo de programación:
 - a) Utilizando un *loader*: El *loader* es una aplicación que reside en un medio de almacenamiento no volátil y permite la descarga de archivos utilizando el puerto serie o una interfaz de red a una memoria no volátil externa.
 - b) Utilizando el puerto JTAG: El puerto JTAG (Joint Test Action Group) proporciona una interfaz capaz de controlar los registros internos del procesador, y de esta forma, acceder a las memorias de la plataforma y ejecutar un programa residente en una posición de memoria determinada.
6. **Depuración** Una vez se descarga la aplicación a la plataforma es necesario someterla a una serie de pruebas, para verificar su correcto funcionamiento. Esto se puede realizar con el depurador GNU (GDB) y una interfaz de comunicación que puede ser un puerto serie, USB o un adaptador de red.

Make

Como vimos anteriormente, es necesario realizar una serie de pasos para poder descargar una aplicación a una plataforma embebida. Debido a que las herramientas GNU solo poseen entrada por consola, es necesario ejecutar una serie de comandos cada vez que se realiza un cambio en el código fuente, lo que resulta poco práctico durante la etapa de desarrollo. Para realizar este proceso de forma automática, se creó la herramienta *make*, la cual recibe como entrada un archivo con una serie de instrucciones que normalmente lleva el nombre de *Makefile* o *makefile*. La herramienta *make* ejecuta los comandos necesarios para realizar la compilación, depuración, o programación, indicados en el archivo *Makefile* o *makefile*. Un ejemplo de este tipo de archivo se muestra a continuación:

```

1 SHELL = /bin/sh
2
3 basetoolsdir = /home/cain/Embedded/ARM/iMX233/toolchain/arm-2008q3
4 bindir = ${basetoolsdir}/bin
5 libdir = ${basetoolsdir}/lib/gcc/arm-none-linux-gnueabi/4.3.2
6
7 CC = arm-none-linux-gnueabi-gcc
8 AS = arm-none-linux-gnueabi-as
9 LD = arm-none-linux-gnueabi-ld
10 OBJCOPY = arm-none-linux-gnueabi-objcopy
11
12 CFLAGS = -mcpu=arm920t -I. -Wall
13 LDFLAGS = -L${libdir} -lgcc
14
15 OBJS = \
16     main.o          \
17     debug_io.o       \
18     at91rm9200_lowlevel.o \
19     p_string.o
20
21 ASFILES = arm_init.o
22
23 LIBS=${libdir}/libgcc.a
24
25 all: hello_world
26
27 hello_world: ${OBJS} ${ASFILES}
28     ${LD} -e 0 -o hello_world.elf -T linker.cfg ${ASFILES} ${OBJS} ${LDFLAGS}
29     ${OBJCOPY} -O binary hello_world.elf hello_world.bin
30
31 clean:
32     rm -f *.o *~ hello_world.*
33
34 .c.o:
35     $(CC) $(CFLAGS) -c $< -o $@
36
37 .S.o:
38     $(CC) $(AFLAGS) -c $< -o $@

```

En las líneas 3-5 se definen algunas variables globales que serán utilizadas a lo largo del archivo; en las líneas 7 - 10 se definen las herramientas de compilación a utilizar: los compiladores de C (CC), de assembler (AS); el enlazador (LD) y la utilidad objcopy. A partir de la línea 15 se definen los objetos que forman parte del proyecto, en este caso: *main.o*, *debug_io.o*, *at91rm9200_lowlevel.o* y *p_string.o*; en la línea 21 se definen los archivos en lenguaje ensamblador, para este ejemplo *arm_init.o*. Las líneas 12 y 13 definen dos variables especiales que el compilador de C (CFLAGS) y el enlazador (LDFLAGS) utilizarán como parámetros .

En las líneas 25, 27 y 31 aparecen unas etiquetas de la forma: *nombre*: esta es la forma de definir reglas y permiten ejecutar de forma independiente el conjunto de instrucciones asociadas a ellas, por ejemplo, si se ejecuta el comando:

```
make clean
```

make ejecutará:

```
rm -f *.o * hello_world.*
```

Observemos los comandos asociados a la etiqueta *hello_world*: En la misma línea aparecen *\$OBJS* *\$ASFILES*, estos parámetros reciben el nombre de dependencias y le indican a la herramienta *make* que antes de ejecutar los comandos asociados a esta etiqueta, debe realizar las acciones necesarias para generar las dependencias; es decir: *main.o*, *debug_io.o*, *at91rm9200_lowlevel.o*, *p_string.o* (*\$OBJS*) y *arm_init.o* (*\$ASFILES*).

En las líneas 34 y 35 se le indica a la herramienta *make* la regla para generar un archivo *.o* a partir de un archivo *.c*; *make* aplicará esta regla a cada uno de los elementos que conforman la variable *\$OBJS*. *\$<* Es el nombre del primer pre-requisito (*.c*) y *\$@* es el nombre del destino (*.o*).

En las líneas 36 y 37 se le indica a la herramienta *make* la regla para generar un archivo *.o* a partir de un archivo *.s*; *make* aplicará esta regla a cada uno de los elementos que conforman la variable *\$OBJS*.

En la línea 28 se realiza el proceso de enlazado; al *linker* (*ld*) se le pasan los parámetros:

- **-e 0**: Punto de entrada, utilice 0 como símbolo para el inicio de ejecución.
- **-o hello_world.elf**: Nombre del archivo de salida *hello_world*
- **-T linker.cfg**: Utilice el archivo de enlace *linker.cfg* para definir las direcciones de las secciones del ejecutable.
- **\$ASFILES \$OBJS \$LDFLAGS**: Lista de objetos y librerías para crear el ejecutable.

En la línea 29 se utiliza la herramienta *objcopy* para generar un archivo binario (*-O binary*) con la información necesaria para cargar en una memoria no volátil. Esto se explicará con mayor detalle más adelante.

Al ejecutar el comando: *make hello_world*, *make* realizará las siguientes operaciones:

```
arm-none-linux-gnueabi-gcc -mcpu=arm920t -I. -Wall -c main.c -o main.o
arm-none-linux-gnueabi-gcc -mcpu=arm920t -I. -Wall -c debug_io.c -o debug_io.o
arm-none-linux-gnueabi-gcc -mcpu=arm920t -I. -Wall -c at91rm9200_lowlevel.c
-o at91rm9200_lowlevel.o
arm-none-linux-gnueabi-gcc -mcpu=arm920t -I. -Wall -c p_string.c -o p_string.o
arm-none-linux-gnueabi-as -o arm_init.o arm_init.s
arm-none-linux-gnueabi-ld -e 0 -o hello_world.elf -T linker.cfg arm_init.o main.o
debug_io.o at91rm9200_lowlevel.o p_string.o -lgcc
arm-none-linux-gnueabi-objcopy -O binary hello_world.elf hello_world.bin
```

El formato ELF

El formato ELF (*Executable and Linkable Format*) Es un estándar para objetos, librerías y ejecutables y es el formato que generan las herramientas GNU. Como puede verse en la figura A.9 un ejecutable *ELF* está compuesto por las secciones (*link view*) o segmentos (*execution view*). Si un programador está interesado en obtener información de secciones sobre tablas de símbolos, código ejecutable específico o información de enlazado dinámico debe utilizar *link view*. Pero si busca información sobre segmentos, como por ejemplo, la localización de los segmentos *text* o *data* debe utilizar *execution view*. El encabezado describe el layout del archivo, proporcionando información de la forma de acceder a las secciones [2].

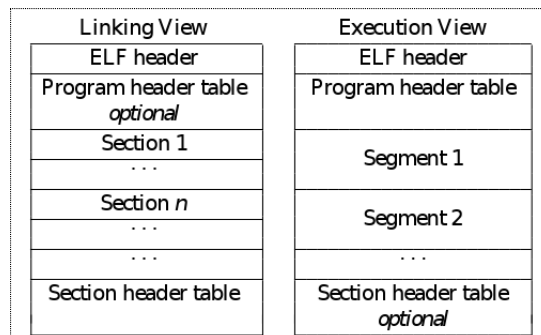


Figura A.9: Formato ELF

Las secciones pueden almacenar código ejecutable, datos, información de enlazado dinámico, datos de depuración, tablas de símbolos, comentarios, tablas de cadenas, y notas. Las secciones más importantes son:

- **.bss** Datos no inicializados. (RAM)
- **.comment** Información de la versión.
- **.data** y **.data1** Datos inicializados. (RAM)
- **.debug** Información para depuración simbólica.
- **.dynamic** Información sobre enlace dinámico
- **.dynstr** Strings necesarios para el enlace dinámico
- **.dynsym** Tabla de símbolos utilizada para enlace dinámico.
- **.fini** Código de terminación de proceso.
- **.init** Código de inicialización de proceso.
- **.line** Información de número de línea para depuración simbólica.
- **.rodata** y **.rodata1** Datos de solo-lectura (ROM)
- **.shstrtab** Nombres de secciones.
- **.symtab** Tabla de símbolos.

■ **.text** Instrucciones ejecutables (ROM)

Para aclarar el contenido de cada una de estas secciones, consideremos la siguiente aplicación sencilla:

```
#include <stdio.h>

int global;
int global_1 = 1;

int main(void)
{
    int i;                // Variable no inicializada
    int j = 2;            // Variable inicializada
    for(i=0; i<10; i++){
        printf("Printing %d\n", i*j);    // Caracteres constantes
        j = j + 1;
        global = i;
        global_1 = i+j;
    }
    return 0;
}
```

Generemos el objeto compilándolo con el siguiente comando: *arm-none-linux-gnueabi-gcc -c hello.c*

Examinemos que tipo de secciones tiene este ejecutable *arm-none-linux-gnueabi-readelf -S hello.o*

Section Headers :										
[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.text	PROGBITS	00000000	000034	00009c	00	AX	0	0	4
[2]	.rel.text	REL	00000000	000484	000020	08		9	1	4
[3]	.data	PROGBITS	00000000	0000d0	000004	00	WA	0	0	4
[4]	.bss	NOBITS	00000000	0000d4	000000	00	WA	0	0	1
[5]	.rodata	PROGBITS	00000000	0000d4	000010	00	A	0	0	4
[6]	.comment	PROGBITS	00000000	0000e4	00004d	00		0	0	1
[7]	.ARM.attributes	ARM_ATTRIBUTES	00000000	000131	00002e	00		0	0	1
[8]	.shstrtab	STRTAB	00000000	00015f	000051	00		0	0	1
[9]	.symtab	SYMTAB	00000000	000368	0000f0	10		10	11	4
[10]	.strtab	STRTAB	00000000	000458	00002b	00		0	0	1

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)

I (info), L (link order), G (group), x (unknown)

O (extra OS processing required) o (OS specific), p (processor specific)

La sección *.text*, como se dijo anteriormente contiene las instrucciones ejecutables, por esta razón se marca como ejecutable “X” en la columna *Flg*. Es posible ver las instrucciones que se ejecutan en esta sección ejecutando:

arm-none-linux-gnueabi-objdump -d -j .text hello.o

```
00000000 <main>:
0:  e92d4800      stmdb    sp!, {fp, lr}
4:  e28db004      add     fp, sp, #4      ; 0x4
8:  e24dd008      sub     sp, sp, #8      ; 0x8
c:  e3a03002      mov     r3, #2          ; 0x2
10: e50b3008      str     r3, [fp, #-8]
14: e3a03000      mov     r3, #0          ; 0x0
18: e50b300c      str     r3, [fp, #-12]
```

```

1c: ea000013      b       70 <main+0x70>
20: e51b200c      ldr     r2, [fp, #-12]
24: e51b3008      ldr     r3, [fp, #-8]
28: e0030392      mul     r3, r2, r3
2c: e59f005c      ldr     r0, [pc, #92] ; 90 <.text+0x90>
30: e1a01003      mov     r1, r3
34: ebfffffe      bl      0 <printf>
38: e51b3008      ldr     r3, [fp, #-8]
3c: e2833001      add     r3, r3, #1 ; 0x1
40: e50b3008      str     r3, [fp, #-8]
44: e59f2048      ldr     r2, [pc, #72] ; 94 <.text+0x94>
48: e51b300c      ldr     r3, [fp, #-12]
4c: e5823000      str     r3, [r2]
50: e51b200c      ldr     r2, [fp, #-12]
54: e51b3008      ldr     r3, [fp, #-8]
58: e0822003      add     r2, r2, r3
5c: e59f3034      ldr     r3, [pc, #52] ; 98 <.text+0x98>
60: e5832000      str     r2, [r3]
64: e51b300c      ldr     r3, [fp, #-12]
68: e2833001      add     r3, r3, #1 ; 0x1
6c: e50b300c      str     r3, [fp, #-12]
70: e51b300c      ldr     r3, [fp, #-12]
74: e3530009      cmp     r3, #9 ; 0x9
78: daffffe8      ble     20 <main+0x20>
7c: e3a03000      mov     r3, #0 ; 0x0
80: e1a00003      mov     r0, r3
84: e24bd004      sub     sp, fp, #4 ; 0x4
88: e8bd4800      ldmia   sp!, {fp, lr}
8c: e12fff1e      bx      lr

```

La sección `.data` mantiene las variables inicializadas, y contiene:

`arm-none-linux-gnueabi-objdump -d -j .data hello.o`

```

00000000 <global_1>:
0: 01 00 00 00

```

Como vemos, la sección `.data` contiene únicamente el valor de inicialización de la variable `global_1` (1) y no muestra información acerca de la variable `j`, esto se debe a que la información está en el `stack` del proceso. Si observamos el contenido de la sección `.text` observamos que esta variable es asignada en tiempo de ejecución, en la línea `0c`: se ve la asignación de esta variable:

```

0c: e3a03002      mov     r3, #2 ; 0x2
10: e50b3008      str     r3, [fp, #-8]

```

La sección `.bss` mantiene la información de las variables no inicializadas. En Linux todas las variables no inicializadas, se inicializan en cero:

`arm-none-linux-gnueabi-objdump -d -j .bss hello`

```

000145c4 <global>:
145c4: 00000000

```

La sección `.rodata` contiene los datos que no cambian durante la ejecución del programa, es decir, los de solo lectura, si examinamos esta sección obtenemos:

`hexdump -C hello.o -- grep -i 000000d0` (la sección `.rodata` comienza en la posición de memoria `0xd4`)

000000d0	01 00 00 00 50 72 69 6e 74 69 6e 67 20 25 64 0aPrinting %d.
000000e0	00 00 00 00 00 47 43 43 3a 20 28 43 6f 64 65 53GCC: (CodeS

Observamos que en el archivo se almacena la cadena de caracteres *Printing %d* a la cual no se modifica durante la ejecución del programa.

Linker Script

Como vimos anteriormente, el *linker* es el encargado de agrupar todos los archivos objeto *.o*, y las librerías necesarias para crear el ejecutable, este *linker* permite definir donde serán ubicados los diferentes segmentos del archivo ELF, por medio de un archivo de enlace *linker script*. De esta forma podemos ajustar el ejecutable a plataformas con diferentes configuraciones de memoria. Esto brinda un grado mayor de flexibilidad de la cadena de herramientas GNU. Cuando se dispone de un sistema operativo como Linux no es necesario definir este archivo para los ejecutables, ya que el sistema operativo se encarga de guardar las secciones en el lugar indicado; sin embargo, es necesario tenerlo presente ya que como veremos más adelante existe un momento en el que el sistema operativo no ha sido cargado en la plataforma y las aplicaciones que se ejecuten deben proporcionar esta información. A continuación se muestra un ejemplo de este archivo:

```
/* identify the Entry Point (_vec_reset is defined in file crt.s) */
ENTRY(_vec_reset)

/* specify the memory areas */
MEMORY
{
    flash : ORIGIN = 0,          LENGTH = 256K    /* FLASH EPROM          */
    ram   : ORIGIN = 0x00200000, LENGTH = 64K     /* static RAM area      */
}

/* define a global symbol _stack_end */
_stack_end = 0x20FFFC;

/* now define the output sections */
SECTIONS
{
    . = 0;                /* set location counter to address zero */
    .text :               /* collect all sections that should go into FLASH after startup */
    {
        *(.text)          /* all .text sections (code) */
        *(.rodata)        /* all .rodata sections (constants, strings, etc.) */
        *(.rodata*)       /* all .rodata* sections (constants, strings, etc.) */
        *(.glue_7)        /* all .glue_7 sections (no idea what these are) */
        *(.glue_7t)       /* all .glue_7t sections (no idea what these are) */
        _etext = .;       /* define a global symbol _etext just after the last code byte */
    } >flash              /* put all the above into FLASH */

    .data :               /* collect all initialized .data sections that go into RAM */
    {
        _data = .;        /* create a global symbol marking the start of the .data section */
        *(.data)          /* all .data sections */
        _edata = .;       /* define a global symbol marking the end of the .data section */
    } >ram AT >flash      /* put all the above into RAM (but load the LMA initializer copy
                           into FLASH) */

    .bss :               /* collect all uninitialized .bss sections that go into RAM */
    {
        _bss_start = .;   /* define a global symbol marking the start of the .bss section */
    }
```

```

    *(.bss)          /* all .bss sections */
} >ram             /* put all the above in RAM (it will be cleared in the startup code*/
. = ALIGN(4);       /* advance location counter to the next 32-bit boundary */
_bss_end = .;       /* define a global symbol marking the end of the .bss section */
}
_end = .;           /* define a global symbol marking the end of application RAM */

```

En las primeras líneas del archivo aparece la declaración de las memorias de la plataforma; en este ejemplo, tenemos una memoria RAM de 64kB que comienza en la posición de memoria 0x00200000 y una memoria flash de 256k que comienza en la posición 0x0. A continuación se definen las secciones y el lugar donde serán almacenadas; en este caso, las secciones *.text* (código ejecutable) y *.rodata* (datos de solo lectura) se almacenan en una memoria no volátil la flash. Cuando el sistema sea energizado el procesador ejecutará el código almacenado en su memoria no volátil. Las secciones *.data* (variables inicializadas) y *.bss* (variables no inicializadas) se almacenarán en la memoria volátil RAM, ya que el acceso a las memorias no volátiles son más lentas y tienen ciclos de lectura/escritura finitos.

En algunos SoCs no se dispone de una memoria no volátil, por lo que es necesario que la aplicación sea cargada por completo en la RAM. Algunos desarrolladores prefieren almacenar y ejecutar sus aplicaciones en las memorias volátiles durante la etapa de desarrollo, debido a que la programación de las memorias no volátiles toman mucho más tiempo. Obviamente una vez finalizada la etapa de desarrollo las aplicaciones deben ser almacenadas en memorias no volátiles.

A.5. Dispositivos semiconductores

En esta sección se realizará una breve descripción de los dispositivos semiconductores más utilizados para la implementación de dispositivos digitales.

SoC

La Figura A.10 muestra la arquitectura de un SoC actual, específicamente del AT91RM920 de Atmel. En este diagrama podemos observar el núcleo central un procesador ARM920T de 180MHz y los periféricos asociados a él. En la actualidad podemos encontrar una gran variedad de SoC diseñados para diferentes aplicaciones: multimedia, comunicaciones, asistentes digitales; los periféricos incluidos en cada SoC buscan minimizar el número de componentes externos, y de esta forma reducir los costos. Este SoC en particular fue uno de los primeros que diseño ATMEL y está enfocado a tareas en las que se requiere una conexión de red. Dentro de los periféricos encontramos:

- Controlador para memorias: NAND flash, DataFlash, SDRAM, SD/MMC
- Puerto USB 2.0 host.
- Puerto I2C
- Interfaz Ethernet 10/100.
- Interfaz high speed USB 2.0
- 4 Puertos SPI.
- 2 puertos seriales (RS232).

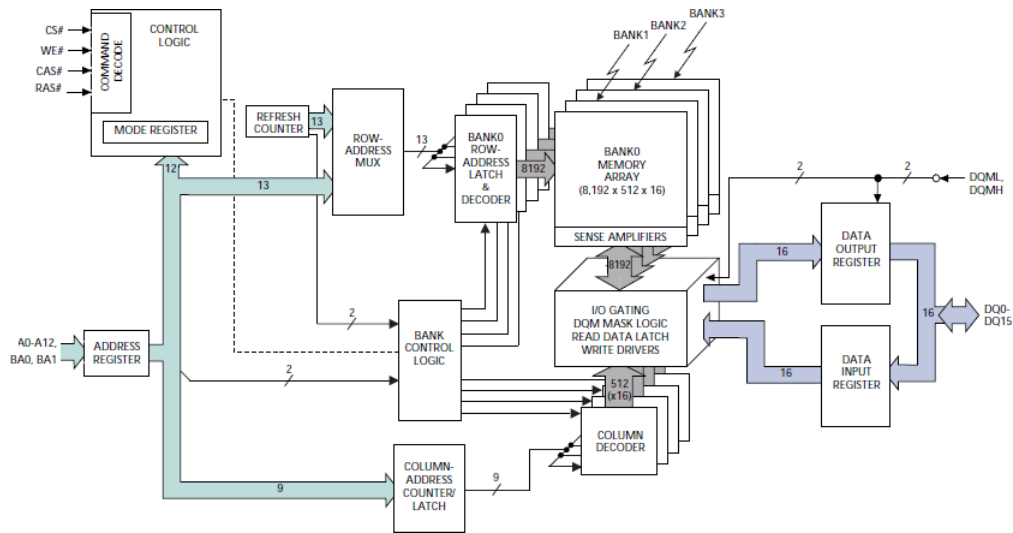


Figura A.11: Diagrama de Bloques de una memoria SDRAM fuente: Hoja de Especificaciones MT48LC16M16, Micron Technology

Un ejemplo simplificado de una operación de lectura es el siguiente: Una posición de memoria se determina colocando la dirección de la fila y la de la columna en las líneas de dirección de fila y columna respectivamente, un tiempo después el dato almacenado aparecerá en el bus de datos. El procesador coloca la dirección de la fila en el bus de direcciones y después activa la señal *RAS* (Row Access Strobe). Después de un retardo de tiempo predeterminado para permitir que el circuito de la SDRAM capture la dirección de la fila, el procesador coloca la dirección de la columna en el bus de direcciones y activa la señal *CAS* (Column Access Strobe). Una celda de memoria SDRAM esta compuesta por un transistor y un condensador; el transistor suministra la carga y el condensador almacena el estado de cada celda, esta carga en el condensador desaparece con el tiempo, razón por la cual es necesario recargar estos condensadores periódicamente, este proceso recibe el nombre de *Refresco*. Un ciclo de refresco es un ciclo especial en el que no se escribe ni se lee información, solo se recargan los condensadores para mantener la información. El periférico que controla la SDRAM está encargado de garantizar los ciclos de refresco de acuerdo con los requerimientos de la SDRAM [3].

Memorias No Volátiles

Las memorias no volátiles almacenan por largos períodos de tiempo información necesaria para la operación de un Sistema Embebido, pueden ser vistos como discos duros de estado sólido; existen dos tipos de memoria las memorias NOR y las NAND; las dos poseen la capacidad de ser escritas y borradas utilizando por software, con lo que no es necesario utilizar programadores externos y pueden ser modificadas una vez instaladas en el circuito integrado. Una desventaja de estas memorias es que los tiempos de escritura y borrado son muy largos en comparación con los requeridos por las memorias RAM.

Las memorias NOR poseen buses de datos y dirección, con lo que es posible acceder de forma fácil a cada byte almacenado en ella. Los bits datos pueden ser cambiados de 0 a 1 por software un byte a la vez, sin embargo, para cambiar un bit de 1 a 0 es necesario borrar una serie de unidades de borrado que reciben el nombre de bloques, lo que permite reducir el tiempo de borrado de la memoria. Debido a que el borrado

y escritura de una memoria ROM se puede realizar utilizando el control software (ver Figura A.12) no es necesario contar con un periférico especializado para su manejo.

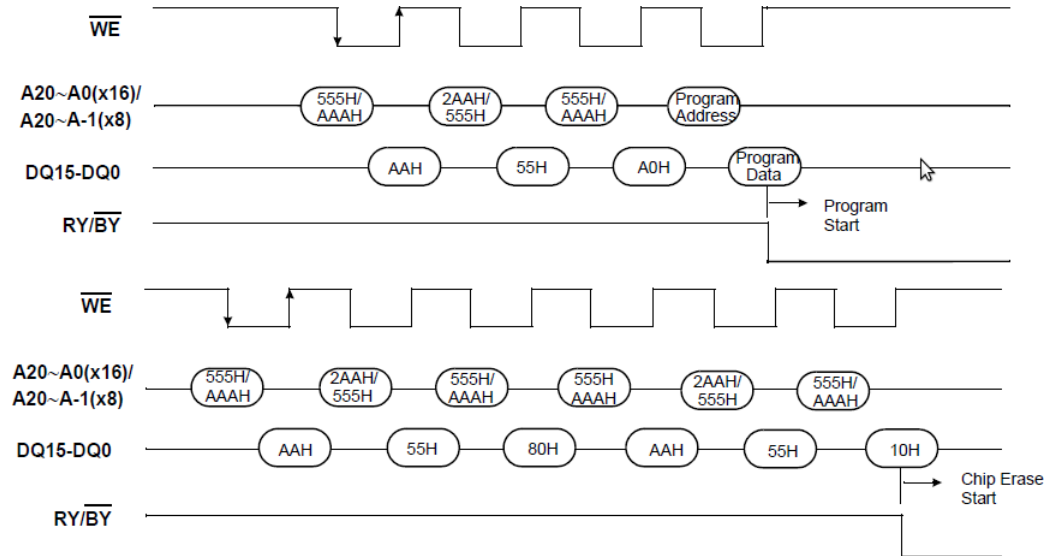


Figura A.12: Ciclos de escritura y borrado de una memoria flash NOR

Las memorias NOR son utilizadas en aplicaciones donde se necesiten altas velocidades de lectura y baja densidad, debido a que los tiempos de escritura y lectura son muy grandes se utilizan como memorias ROM. Las memorias NAND disminuyen los tiempos de escritura y aumentan la capacidad de almacenamiento, ideales para aplicaciones donde se requiera almacenamiento de información. Adicionalmente las memorias NAND consumen menos potencia que las memorias NOR, por esta razón este tipo de memorias son utilizadas en casi todos los dispositivos de almacenamiento modernos como las memorias SD y USB, las que integran una memoria NAND con un circuito encargado de controlarlas e implementar el protocolo de comunicación. A diferencia de las flash tipo NOR, los dispositivos NAND se acceden de forma serial utilizando interfaces complejas; su operación se asemeja a un disco duro tradicional. Se accede a la información utilizando bloques (más pequeños que los bloques NOR). Los ciclos de escritura de las flash NAND son mayores en un orden de magnitud que los de las memorias NOR.

Un problema al momento de trabajar con las memorias tipo NAND es que requieren el uso de un *manejo de bloques defectuosos*, esto es necesario ya que las celdas de memoria pueden dañarse de forma espontánea durante la operación normal. Debido a esto, se debe tener un determinado número de bloques que se encarguen de almacenar tablas de mapeo para manejar los bloques defectuosos; o puede hacerse un chequeo en cada inicialización del sistema de toda la memoria para actualizar esta lista de sectores defectuosos. El algoritmo de ECC (Error-Correcting Code) debe ser capaz de corregir errores tan pequeños como un bit de cada 2048 bits, hasta 22 bits de cada 2048. Este algoritmo es capaz de detectar bloques defectuosos en la fase de programación, comparando la información almacenada con la que debe ser almacenada (verificación), si encuentra un error marca el bloque como defectuoso y utiliza un bloque sin defectos para almacenar la información.

La tabla A.1 resume las principales características de los diferentes tipos de memoria flash.

	SLC NAND	MLC NAND	MLC NOR
Densidad	512Mbits - 4GBits	1Gbits - 16GBits	16Mbits - 1Gbit
Velocidad de Lectura	24MB/s	18.6MB/s	103MB/s
Velocidad de escritura	8 MB/s	2.4MB/s	0,47MB/s
Tiempo de borrado	2ms	2ms	900ms
Interfaz	Acceso Indirecto	Acceso Indirecto	Acceso Aleatorio
Aplicación	Almacenamiento	Almacenamiento	Solo lectura

Cuadro A.1: Cuadro de comparación de las memorias flash NAND y NOR

Adicionalmente, se encuentran disponibles las memorias *DATAFLASH*, estos dispositivos son básicamente una memoria flash tipo NOR con una interfaz SPI, permite una velocidad de lectura de hasta 66MHz utilizando solamente 4 pines para la comunicación con el procesador.

A.6. Depuración del core ARM [4]

Todos los núcleos ARM7 y ARM9 poseen soporte de depuración en modo *halt*, lo que permite detener por completo el núcleo. Durante este estado es posible modificar y capturar las señales del núcleo, permitiendo cambiar y examinar el estado del sistema. En este estado, la fuente de reloj del núcleo es el reloj de depuración (DCLK) que es generado por la lógica de depuración. Los núcleos ARM7 y ARM9 implementan un controlador compatible con JTAG, con dos cadena boundary-scan alrededor de las señales del núcleo; una con las señales del núcleo, para pruebas del dispositivo y la otra es un sub-set de la primera con señales importantes para la depuración. La Figura A.13 muestra el orden de las señales en las cadenas para los núcleos ARM7TDMI y ARM9TDMI. Para propósitos de depuración es suficiente la cadena 1. Esta cadena puede ser utilizada en modo *INTTEST*, el que permite capturar las señales del núcleo y aplicar vectores de prueba al mismo, o en modo *EXTTEST*, permitiendo la salida y entrada de información hacia y desde exterior del núcleo respectivamente.



Figura A.13: Cadena Boundary Scan 1

Las señales $D[0:31]$ del ARM7TDMI están conectadas al bus de datos del núcleo y se utiliza para capturar instrucciones o lectura/escritura de información; la señal *BREAKPT* se utiliza para indicar que la

instrucción debe ejecutarse a la velocidad del sistema, esto es, utilizando el reloj MCLK en lugar de DCLK.

Las señales $ID[0:31]$ del ARM9TDMI están conectadas al bus de instrucciones y se utilizan para capturar instrucciones, las señales $DD[31:0]$ están conectadas al bus de datos bi-direccional y se utilizan para leer o escribir información.

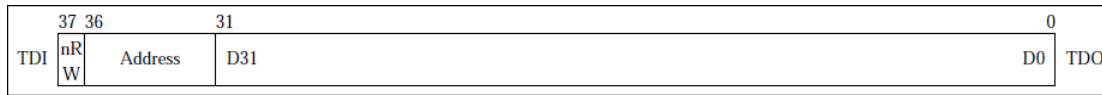


Figura A.14: Cadena Boundary Scan 2 (Embedded ICE)

Los ARM7 y ARM9 poseen un módulo *ICE* (In Circuit Emulator) que reemplaza el microcontrolador con una variación que posee facilidades para la depuración hardware. El emulador es conectado a un computador que ejecuta el software de depuración. Esto permite realizar depuración activa y pasiva, dando un punto de vista no intrusivo del flujo del programa. La Figura A.14 muestra la cadena scan ICE, que es la misma para los núcleos ARM7 y ARM9, esta formada por 32 bits de datos, 5 bits de direcciones y una bandera para diferenciar entre lectura (nRW bajo) y escritura. Se puede acceder a las características ICE a través de registros, cuya dirección es colocada en el bus de direcciones.

A.6.1. Proyecto OpenOCD

El proyecto *OpenOCD* (Open On-Chip Debugger ⁵) permite la programación de la memoria flash interna de algunos procesadores ARM7TDMI, y la depuración de procesadores ARM7 y ARM9 utilizando el módulo ICE. Este proyecto se ha convertido en el más popular dentro del grupo de desarrolladores de sistemas Embebidos. En [4] se puede encontrar el funcionamiento interno de esta herramienta. La figura A.15 muestra el principio de funcionamiento de la herramienta *OpenOCD*.

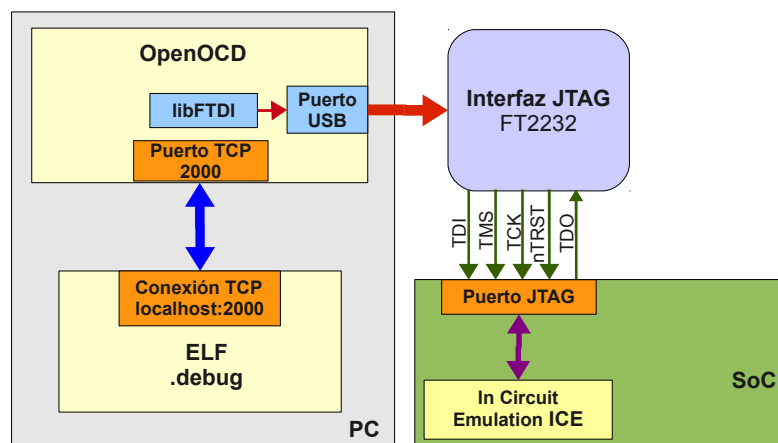


Figura A.15: Principio de funcionamiento del proyecto OpenOCD

La herramienta de depuración *GDB* utiliza la información suministrada por las secciones *debug*... del ejecutable *ELF*. Esta información es generada al agregar **-g** a los parámetros pasados al compilador

⁵<http://openocd.berlios.de/web/>

(CFLAGS); proporcionando la información necesaria para relacionar símbolos y código lo que permite su depuración. A continuación se muestra una parte de la salida del comando *arm-none-linux-gnueabi-objdump -S -j .debug_info hello* el cual muestra el contenido de la sección *debug_info*:

```
int main(void) {
    160: 0001e502  andeq lr, r1, r2, lsl #10
    164: 24230200  strcs r0, [r3], #-512
    168: 00029f08  andeq r9, r2, r8, lsl #30

    char    l_char;

    DebugPrint("\n\rEntry: main\n\r");
    16c: 025e0200  subseq r0, lr, #0 ; 0x0
    170: 00000025  andeq r0, r0, r5, lsr #32
```

Con esta información, *GDB* utiliza el protocolo serial (Remote Serial Protocol - RSP) para ejecutar una operación determinada; estas instrucciones son enviadas a un puerto TCP (2000 en este ejemplo). Por ejemplo, para leer dos bytes desde la posición de memoria *0x4015bc*, *GDB* enviará el paquete: **\$m4015bc,2#5a** por el puerto 2000; *m* indica que se debe leer la memoria, todo paquete *RSP* comienza con el signo \$ y termina con el signo # seguido por dos bytes que representan el *checksum*.

Estos paquetes enviados por *GDB* son recibidos por un programa que se encuentra en ejecución, esperando por una comunicación RSP en el mismo puerto. *OpenOCD* establece la comunicación con *GDB* y convierte los comandos *RSP* a una serie de instrucciones JTAG que implementan la operación requerida. Estas instrucciones JTAG son enviadas a una interfaz especial basada en un dispositivo de la compañía *FTDI* que permite controlar el puerto JTAG a través de una interfaz USB. *OpenOCD* utiliza la librería abierta *libftdi* para controlar esta interfaz.

Programación de memorias Flash

Algunos SoC no poseen programas de inicialización que permitan la descarga de aplicaciones ya sea a las memorias externas o a la interna. *OpenOCD* puede ser utilizado en estos casos para cargar las aplicaciones en la RAM interna; sin embargo, existen procesadores en los que no se conocen las especificaciones del ICE y en en otros este módulo no existe. En estos casos se utiliza la interfaz JTAG en modo *EXTEST* para controlar directamente los pines a los que se encuentran conectadas las memorias a los SoCs. El proyecto *UrJTAG*,⁶ y el proyecto *OpenOCD* permiten la programación de diversas memorias flash, utilizando los pines del dispositivo. Adicionalmente, *UrJTAG* permite la creación de diferentes interfaces para conectarse con las memorias, estas interfaces reciben el nombre de buses y pueden ser creadas e incluidas en el código original de forma fácil.

A.7. El sistema Operativo Linux

Hello everybody out there using minix - I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones.

Con este correo enviado al foro de discusión comp.os.minix, Linus Torvalds, un estudiante de la Universidad de Helsinki en Finlandia introduce Linux el 25 de Agosto de 1991. A principios de los 90, el sistema operativo Unix (desarrollado en *The Bell Labs* a principios de los 60s), tenía una solida posición en el mercado de servidores, y estaba muy bien posicionado en las Universidades, por lo que un gran número

⁶<http://urjtag.sourceforge.net/>

de estudiantes trabajaban a diario con él y muchos de ellos deseaban poder utilizarlo en sus computadores personales; sin embargo, Unix era un producto comercial muy caro y solo podía ejecutarse en costosos servidores. La figura A.16 muestra el desarrollo previo a la creación de la primera versión de Linux hasta el día de hoy.

Una de las características más importantes de Linux es que desde su creación, fue pensado como un sistema operativo gratuito y de libre distribución. Esta característica ha permitido que programadores a lo largo del mundo puedan manipular el código fuente para eliminar errores y para aumentar sus capacidades. Sin embargo, el crédito de las distribuciones que conocemos hoy (Debian, Ubuntu, Suse, etc) no solo se debe a Torvalds, ya que Linux es solo el kernel del sistema operativo. En 1983, Richard Stallman funda en proyecto GNU, el cual proporciona una parte esencial de los sistemas Linux. A principios de los 90s, GNU había producido una serie de herramientas como librerías, compiladores, editores de texto, Shells, etc. Estas herramientas fueron utilizadas por Torvalds para escribir el elemento que le hacía falta al proyecto GNU para completar su sistema operativo: el kernel.

Desde el lanzamiento de la primera versión de Linux, cientos, miles, cientos de miles y millones de programadores se han puesto en la tarea de convertirlo en un sistema operativo robusto, amigable y actualizado; tan pronto como se desarrolla una nueva pieza de hardware existen cientos de programadores trabajando en crear el soporte para Linux.

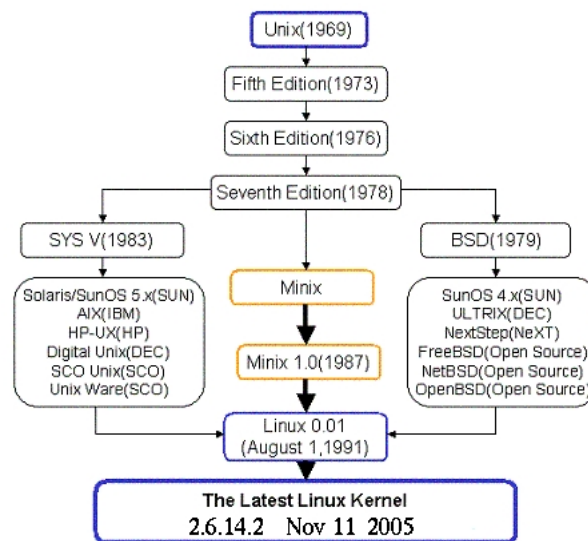


Figura A.16: Linux: Historia Fuente.

Porqué Linux

Existen varias motivaciones para escoger Linux frente a un sistema operativo tradicional para sistemas embebidos [5]:

- **Calidad Y confiabilidad del código:** Aunque estas medidas son subjetivas, miden el nivel de confianza en el código, que compromete software como el kernel y las aplicaciones proporcionadas por las distribuciones.

- **Disponibilidad de Código:** Todo el código fuente de las aplicaciones, del sistema operativo y de las herramientas de compilación se encuentran disponibles sin ninguna restricción. Existen varios tipos de licencias, siendo las más populares la GNU General Public License (GPL) y la BSD (Berkeley Software Distribution), esta última permite la distribución de binarios sin el código fuente.
- **Soporte de hardware:** Linux soporta una gran variedad de dispositivos y plataformas, a pesar de que muchos fabricantes no proporcionan soporte para Linux, la comunidad trabaja arduamente en incluir el nuevo hardware en las nuevas distribuciones de Linux. Linux en la actualidad se puede ejecutar en docenas de diferentes arquitecturas hardware, lo cual lo convierte en el Sistema Operativo más portable.
- **Protocolos de comunicación y estándares de software:** Linux proporciona muchos protocolos de comunicación, lo que permite su fácil integración a plataformas de trabajo existentes.
- **Disponibilidad de herramientas:** La variedad de herramientas disponibles para Linux lo hacen muy versátil. Existen grandes comunidades como SourceForge⁷ o Freshmeat⁸ que permiten a miles de desarrolladores compartir sus trabajos y es posible que en ellas los usuarios encuentren una aplicación que cumpla con sus necesidades.
- **Soporte de la comunidad:** Esta es la principal fortaleza de Linux. Millones de usuarios alrededor del mundo pueden encontrar errores en alguna aplicación y desarrolladores miembros de la comunidad (en algunos casos los creadores de las aplicaciones) arreglarán este problema y difundirán su solución. El mejor sitio para intercambio de información son las listas de correo de soporte y desarrollo.
- **Licencia:** Al crear una aplicación bajo alguna de las licencias usuales en Linux, no implica perder la propiedad intelectual ni los derechos de autor de la misma.
- **Independencia del vendedor:** No existe solo un distribuidor del sistema operativo GNU-Linux, ya que las licencias de Linux garantizan igualdad a los distribuidores. Algunos vendedores proporcionan aplicaciones adicionales que no son libres y por lo tanto no se encuentran disponibles con otros vendedores. Esto debe tenerse en cuenta en el momento de elegir la distribución a utilizar.
- **Costo:** Muchas de las herramientas de desarrollo y componentes de Linux son gratuitos, y no requieren el pago por ser incluidos en productos comerciales.

A.7.1. Arquitectura de Linux [6] [7]

Linux (Linus' Minix) es un clon del sistema operativo Unix para PC con procesador Intel 386. Linux tomó dos características muy importantes de Unix: sistema multitarea y multiusuario, lo cual fue implementado posteriormente por los sistemas operativos Windows y MacOS. Con estas características es posible ejecutar tareas de forma independiente, y transparente para el/los usuarios.

Linux está compuesto por cinco sub-módulos [7]: El programador (Scheduler), el manejador de memoria, el sistema de archivos virtual, la interfaz de red y la comunicación entre procesos (IPC) (figura A.17).

Como puede observarse en la figura A.17 el kernel de Linux posee sub-módulos que son independientes de la arquitectura y otros que deben ser escritos para el procesador utilizado, esto hace que Linux sea un sistema operativo portable. En la actualidad existe una gran variedad de arquitecturas soportadas por Linux, entre las que se encuentran: alpha arm26 frv i386 m32r m68knommu parisc ppc64

⁷<http://www.sourceforge.net>

⁸<http://www.freshmeat.net>

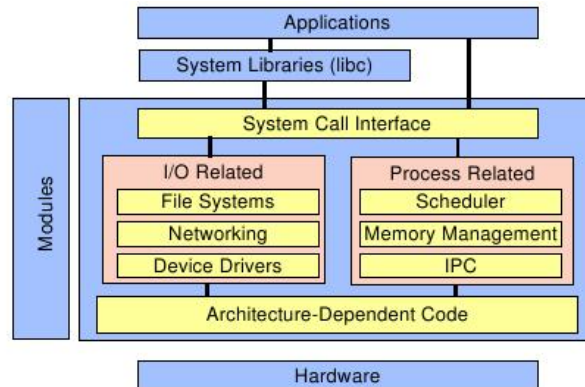


Figura A.17: Estructura del kernel de Linux

sh sparc um x86_64 arm cris h8300 ia64 m68k mips ppc s390 sh64 sparc64 v850. Las funciones de estos componentes se describen a continuación [6]:

Programador de procesos (Scheduler)

Es el corazón del sistema operativo Linux; es encargado de realizar las siguientes tareas:

- Permitir a los procesos hacer copias de si mismos.
- Determinar que procesos pueden acceder a la CPU y efectuar la transferencia entre los procesos en ejecución.
- Recibir interrupciones y llevarlas al subsistema del kernel adecuado.
- Enviar señales a los procesos de usuario.
- Manejar el timer físico.
- Liberar los recursos de los procesos cuando estos finalizan su ejecución.

El programador de procesos proporciona dos interfaces: Una con recursos limitados para los procesos de usuario y una interfaz amplia para el resto del kernel. Debido a que el programador utiliza una interrupción del timer que se presenta cada 10 ms, el cambio de estado del programador⁹ se realiza cada 10 ms. Esto debe ser tenido en cuenta a la hora de realizar procesos que manejen dispositivos hardware veloces.

Manejador de Memoria

En el momento en que se energiza la CPU, esta solo ve 1 MByte de memoria física (incluyendo las ROMs). El código de inicialización del sistema operativo (OS) debe activar el modo protegido del procesador, de tal forma que se pueda acceder a la memoria extendida (incluyendo la memoria de los

⁹Un proceso puede estar en uno de los siguientes estados: ejecución, retornando de un llamado de sistema, procesando una rutina de interrupción, procesando un llamado del sistema, listo, en espera

dispositivos). Finalmente, el OS habilita la memoria virtual para dar la ilusión de un espacio de memoria de 4 GB. El manejador de memoria proporciona los siguientes servicios (ver figuras A.18 y A.19):

- **Gran espacio de memoria.** Los procesos usuario, pueden referenciar más memoria que la existente físicamente.
- **Protección** La memoria de un proceso es privada y no puede ser leída o modificada por otro proceso. Adicionalmente evita que los procesos sobrescriban datos de solo lectura.
- **Mapeo de memoria** Los usuarios pueden mapear un archivo en un área de memoria virtual y acceder a el como si fuera una memoria.
- **Acceso transparente a la memoria física** lo que asegura un buen desempeño del sistema.
- **Memoria compartida**

Al igual que el programador, el manejador de memoria proporciona dos niveles de acceso a memoria diferentes a nivel de usuario y a nivel de kernel.

- Nivel de Usuario
 - *malloc()* / *free()*. Asigna o libera memoria para que sea utilizada por un proceso.
 - *mmap()* / *munmap()* / *msync()* / *mremap()* Mapea archivos en regiones de memoria virtual.
 - *mprotect* Cambia la protección sobre una región de una memoria virtual.
 - *mlock()* / *mlockall()* / *munlock()* / *munlockall()* Rutinas que permiten al super-usuario prevenir el intercambio de memoria.
 - *swapon()* / *swapoff()* Rutinas que le permiten al super-usuario agregar o eliminar archivos *swap* en el sistema.
- Nivel de kernel
 - *kmalloc()* / *kfree()* Asigna o libera memoria para que sea utilizada por estructuras de datos del kernel.
 - *verify_area()* Verifica que una región de la memoria de usuario ha sido mapeada con los permisos necesarios.
 - *get_free_page()* / *free_page()* Asigna y libera páginas de memoria física.

Comunicación Entre Procesos (IPC)

El mecanismo IPC de Linux posibilita la ejecución *concurrente* de procesos, permitiendo compartir recursos y la sincronización e intercambio de datos entre ellos. Linux proporciona los siguientes mecanismos:

- **Señales** Mensajes asíncronos enviados a los procesos.
- **Listas de espera** Proporciona mecanismos para colocar a dormir a los procesos mientras esperan que una operación se complete, o un recurso se libere.

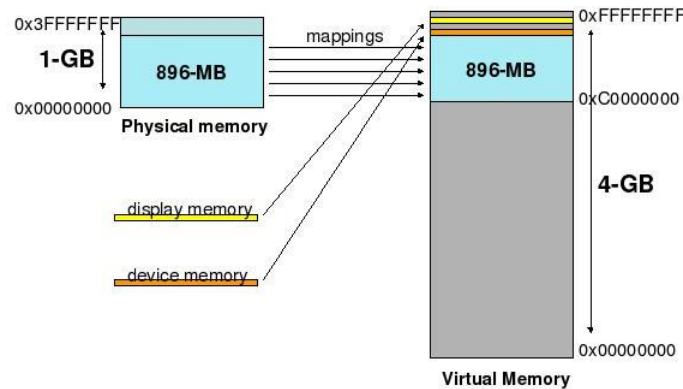


Figura A.18: Mapeo de memoria del Kernel.

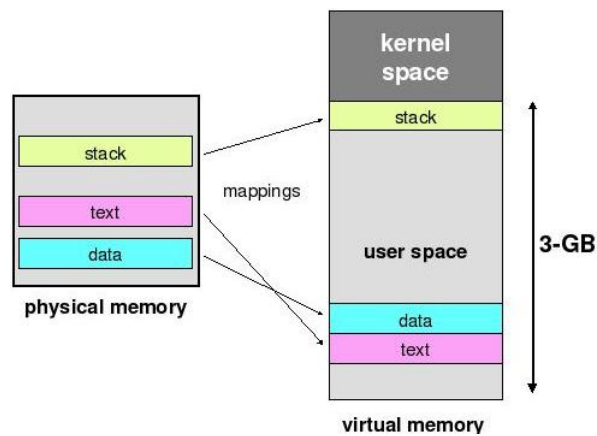


Figura A.19: Mapeo de memoria para una aplicación

- **Bloqueo de archivos** Permite a un proceso declarar una región de un archivo como solo lectura para los demás procesos.
- **Conductos (pipe)** Permite transferencias bi-direccionales entre dos procesos.
- **System V**
 - **Semáforos** Una implementación del modelo clásico del semáforo.
 - **Lista de Mensajes** Secuencia de bytes, con un tipo asociado, los mensajes son escritos a una lista de mensajes y pueden obtenerse leyendo esta lista.
 - **Memoria Compartida** Mecanismo por medio del cual varios procesos tienen acceso a la misma región de memoria física.
- **Sockets del dominio Unix** Mecanismo de transferencia de datos orientada a la conexión.

Interfaces de Red

Este sistema proporciona conectividad entre máquinas, y un modelo de comunicación por sockets. Se proporcionan dos modelos de implementación de sockets: BSD e INET. Además, proporciona dos protocolos de transporte con diferentes modelos de comunicación y calidad de servicio: El poco confiable protocolo UDP (*User Datagram Protocol*) y el confiable TCP (*Transmission Control Protocol*), este último garantiza el envío de los datos y que los paquetes serán entregados en el mismo orden en que fueron enviados. Se proporcionan tres tipos diferentes de conexión: SLIP (Serial), PLIP (paralela) y ethernet.

Sistema de archivo virtual

El sistema de archivos de Linux cumple con las siguientes tareas:

- **Controlar múltiples dispositivos hardware**
- **Manejar sistemas de archivos lógicos**
- **Soporta diferentes formatos ejecutables** Por ejemplo a.out, ELF, java)
- **Homogeneidad** Proporciona una interfaz común a todos los dispositivos lógicos o físicos.
- **Desempeño**
- **Seguridad**
- **Confiabilidad**

Drivers de Dispositivos

La capa que controla los dispositivos es responsable de presentar una interfaz común a todos los dispositivos físicos. El kernel de Linux tiene 3 tipos de controladores de dispositivo: Caracter (acceso secuencial), bloque (acceso en múltiplos de tamaño de bloque) y red. Ejemplos de controladores secuenciales son el modem, el mouse; ejemplos de controladores tipo bloque son los dispositivos de almacenamiento masivo como discos duros, memorias SD. Los controladores de dispositivos soportan las operaciones de archivo, y pueden ser tratados como tal.

Sistema de Archivos lógico

Aunque es posible acceder a dispositivos físicos a través de un archivo de dispositivo, es común acceder a dispositivos tipo bloque utilizando un sistema de archivos lógico, el que puede ser montado en un punto del sistema de archivos virtual.

Para dar soporte al sistema de archivos virtual, Linux utiliza el concepto de *inodes*. Un inode representa un archivo sobre un dispositivo tipo bloque; el inode es virtual en el sentido que contiene operaciones que son implementadas de forma diferente dependiendo del sistema lógico y del sistema físico donde reside el archivo. La interfaz inode hace que todos los archivos se vean igual a otros subsistemas Linux. El inode se utiliza como una posición de almacenamiento para la información relacionada con un archivo abierto en el disco. El inode almacena los buffers asociados, la longitud total del archivo en bloques, y el mapeo entre el offset del archivo y los bloques del dispositivo.

Módulos

La mayor funcionalidad del sistema de archivos virtual se encuentra disponible en forma de módulos cargados dinámicamente. Esta configuración dinámica permite a los usuarios de Linux compilar un kernel tan pequeño como sea posible, mientras permite cargar el controlador del dispositivo y módulos del sistema de archivos solo si son necesarios durante una sesión. Esto es útil en el caso de los dispositivos que se pueden conectar en caliente.

A.8. Portando Linux a la plataforma ECBOT y ECB_AT91

Como vimos anteriormente el kernel de Linux es el corazón del sistema operativo GNU/Linux y es el encargado de manejar directamente el hardware asociado a una determinada plataforma; por lo tanto, es necesario que este sea capaz de manejar todos los periféricos asociados a esta y proporcione caminos para controlarlos. En esta sección se describirá el proceso que debe realizarse para hacer que Linux se ejecute de forma correcta en una plataforma y que permita controlar sus diferentes componentes hardware (este proceso recibe el nombre de *porting*).

A.8.1. El Kernel de Linux

El código fuente se encuentra dividido entre el código específico a una arquitectura y el código común. El código específico de cada arquitectura lo encontramos en los subdirectorios *arch* y *include/asm-xxx*, en ellos podemos encontrar los archivos para las arquitecturas: alpha, blackfin, h8300, m68k, parisc, s390, sparc, v850, arm, cris, ia64, m68knommu, powerpc, sh, sparc64, x86, avr32, frv, m32r, mips, ppc, sh64, um, xtensa. A continuación se listan los directorios que hacen parte del código fuente de Linux.

arch	fs	lib	net	block
usr	include	crypto	ipc	scripts
Documentation	mm	security	drivers	kernel
sound				

Dentro de cada arquitectura soportada por Linux (*arch/xxx/*) encontramos los siguientes directorios:

- **kernel** Código del núcleo del kernel.
- **mm** Código para el manejo de memoria.
- **lib** Librería de funciones internas, optimizadas para la arquitectura (backtrace, memcpy, funciones I/O, bit-twiddling etc);
- **nwfpe** Implementaciones de punto flotante.
- **boot** Sitio donde reside la imagen del kernel una vez compilada, este directorio contiene herramientas para generar imágenes comprimidas.
- **tools** Contiene scripts para la auto-generación de archivos, también contiene el archivo *mach-types* que contiene la lista de las máquinas (plataformas) registradas.
- **configs** Contiene el archivo de configuración para cada plataforma.

Si deseamos dar soporte a una determinada plataforma debemos trabajar en el directorio que contiene la arquitectura del procesador a utilizar, en nuestro caso trabajaremos con la arquitectura *arm*. En el código fuente de Linux se encuentran muchas plataformas que utilizan las diferentes arquitecturas, los archivos de configuración de estas pueden ser utilizados para crear una propia. En nuestro caso tomamos como referencia la plataforma *Atmel AT91RM9200-EK* diseñada por ATMEL. Los archivos de configuración de esta plataforma se encuentra en: *arch/arm/mach-at91/board-ek.c*; dentro de la arquitectura *arm* existen varias sub-arquitecturas que corresponden a las diferentes familias de SoC; El AT91RM9200 hace parte de la familia de SoCs AT91 de ATMEL.

Existen dos acciones que se deben realizar para dar soporte de nuestra plataforma al kernel de Linux, la primera es adaptar un archivo de configuración de una plataforma existente y la otra es registrar la nueva plataforma. Cada plataforma ARM es identificada por un *machine ID*, el primer paso en el proceso de *port* es obtener un ID, lo cuál se puede hacer en línea: <http://www.arm.linux.org.uk/developer/machines/>, mientras que se asigna un nuevo número puede asignarse uno que no este utilizado en el archivo: *arch/arm/tools/mach-types*. La entrada correspondiente a la familia de plataformas ECBAT91 es:

ecbat91	MACHECBAT91	ECBAT91	1072
---------	-------------	---------	------

Con este ID (o uno temporal) se debe crear la siguiente entrada en el archivo */arch/arm/mach-at91/Kconfig* b/*arch/arm/mach-at91/Kconfig*:

```
config MACHECBAT91
    bool "emQbit ECB_AT91 SBC"
    depends on ARCH_AT91RM9200
    help
        Select this if you are using emQbit's ECB_AT91 board.
        <http://wiki.emqbit.com/free-ecb-at91>
```

La que permite seleccionar nuestra plataforma desde el programa de configuración de Linux¹⁰(Ver Figura A.20)

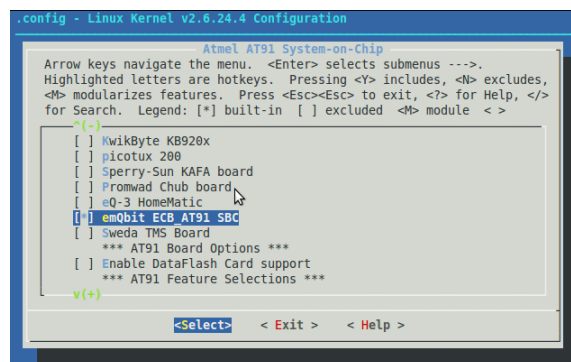


Figura A.20: Herramienta de configuración de Linux mostrando la plataforma ECB_AT91

Al seleccionar nuestra plataforma el programa de configuración creará un archivo de configuración *.config*¹¹ localizado en la raíz del código fuente, este archivo refleja las selecciones realizadas para configurar la imagen del kernel; en este caso específico se agregará la línea:

¹⁰este programa se ejecuta con el comando: *make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-*

¹¹los archivos que comienzan con "." están ocultos


```
CONFIG_MACH_ECBAT91=y
```

Como veremos más adelante, la aplicación encargada de cargar la imagen del kernel de Linux le pasa a este el número de identificación de la plataforma, si este número no es el mismo que el kernel tiene registrado se generará un error, para evitar esto debemos incluir las siguientes líneas en el archivo *arch/arm/boot/compressed/head-at91rm9200.S*

```
@ emQbit ECB_AT91 : 1072
mov    r3 ,      #(MACH_TYPE_ECBAT91 & 0xff)
orr     r3 , r3 , #(MACH_TYPE_ECBAT91 & 0xff00)
cmp     r7 , r3
beq     99f
```

El archivo */arch/arm/mach-at91/Makefile* *b/arch/arm/mach-at91/Makefile* contiene una relación entre la plataforma seleccionada y el archivo de compilación a utilizar, por lo que debemos incluir las siguientes líneas:

```
obj-$(CONFIG_MACH_ECBAT91) += board-ecbat91.o
```

Con esto asignamos el archivo de configuración *board-ecbat91.c* a la plataforma *MACH_ECBAT91*.

Archivo de configuración de la plataforma ECB_AT91

En esta sección realizaremos una descripción del archivo de configuración de la familia de plataformas *ECB_AT91* (*board-ecbat91.c*) y se hará una explicación de cada uno de los parámetros de este archivo.

Como todo archivo escrito en C al comienzo se declaran los encabezados que contienen las funciones utilizadas, en nuestro caso:

```
1 #include <linux/types.h>
2 #include <linux/init.h>
3 #include <linux/mm.h>
4 #include <linux/module.h>
5 #include <linux/dma-mapping.h>
6 #include <linux/platform_device.h>
7 #include <linux/spi/spi.h>
8 #include <linux/spi/flash.h>
9 #include <linux/leds.h>
10
11 #include <asm/hardware.h>
12 #include <asm/setup.h>
13 #include <asm/mach-types.h>
14 #include <asm/irq.h>
15
16 #include <asm/mach/arch.h>
17 #include <asm/mach/map.h>
18 #include <asm/mach/irq.h>
19
20 #include <asm/arch/board.h>
21 #include <asm/arch/at91rm9200_mc.h>
22 #include <asm/arch/gpio.h>
23 #include <linux/gpio_keys.h>
24 #include <linux/input.h>
25
26 #include "generic.h"
```

```

27 #include <asm/arch/at91_pio.h>
28
29 #include <linux/w1-gpio.h>

```

Cada arquitectura (CPU) posee un archivo donde se declaran los diferentes periféricos que pueden ser utilizados: *arch/arm/mach-at91/at91rm9200_devices.c* en el caso del procesador AT91RM9200; En él podemos encontrar soporte para: USB Host, USB Device, Ethernet, Compact Flash / PCMCIA, MMC / SD, NAND / SmartMedia, TWI (i2c), SPI, Timer/Counter blocks, RTC, Watchdog, SSC – Synchronous Serial Controller, UART. Este archivo, proporciona funciones que permiten incluir y configurar los diferentes controladores asociados al SoC; adicionalmente, realiza las operaciones necesarias para poder utilizarlo; como por ejemplo, definir que el control de un determinado pin este a cargo del periférico.

El primer dispositivo declarado en el archivo de configuración es el puerto serial (UART), este puerto es vital ya que es el único medio de comunicación que tenemos con nuestra plataforma.

```

static struct at91_uart_config __initdata ecb_at91uart_config = {
    .console_tty    = 0,                /* ttyS0 */
    .nr_tty         = 2,
    .tty_map        = { 4, 0, -1, -1, -1 } /* ttyS0, ..., ttyS4 */
};

```

Los parámetros de configuración de la *UART* se fijan utilizando la estructura *at91_uart_config* declarada en el archivo *include/asm-arm/arch/board.h*, la variable *console_tty* define el número del dispositivo *tty* asociado a la consola serial, *ttyS0* en nuestro caso, la variable *nr_tty* define el número de interfaces seriales disponibles ¹², *ttyS0* y *ttyS1* en nuestro caso; *tty_map* realiza la correspondencia entre los dispositivos *tty* y las *UART* disponibles en el SoC, para este ejemplo asocia *ttyS0* a la *UART4* y *ttyS1* a la *UART0*

```

static void __init ecb_at91map_io(void)
{
    /* Initialize processor: 18.432 MHz crystal */
    at91rm9200_initialize(18432000, AT91RM9200_PQFP);

    /* Setup the LEDs */
    at91_init_leds(AT91_PIN_PB20, AT91_PIN_PB20);

    /* Setup the serial ports and console */
    at91_init_serial(&ecb_at91uart_config);
}

```

La función *at91rm9200_initialize* declarada en *arch/arm/mach-at91/at91rm9200.c* se encarga, como su nombre lo indica, de inicializar el procesador AT91RM9200, específicamente se encarga de configurar, el reloj interno del procesador (con la función *at91_clock_init*), registra los osciladores configurables PCK0 a PCK3 (utilizando la función *at91rm9200_register_clocks*) y finalmente habilita el soporte para los pines de entrada/salida de propósito general *GPIOs* (utilizando la función *at91_gpio_init*). Adicionalmente informa que se está utilizando el empaquetado TQFP208 (Este chip viene en dos versiones TQFP y BGA).

Las plataformas que carecen de dispositivos de visualización (pantalla, LCD), reflejan la actividad de procesos importantes en el estado de diodos emisores de luz LEDs, Linux permite visualizar la actividad de la CPU y el estado de un timer interno (*at91_init_leds(cpu_led, timer_led)*). Nuestra plataforma utiliza un LED conectado al pin PB20 para estas indicaciones.

La función *at91_init_serial(&ecb_at91uart_config)*, como su nombre lo indica inicializa las interfaces seriales utilizando la estructura *at91_init_serial* explicada anteriormente.

¹²El AT91RM9200 tiene 4 USARTS y una UART para depuración

```
static void __init ecb_at91init_irq(void)
{
    at91rm9200_init_interrupts(NULL);
}
```

La función *at91rm9200_init_interrupts* inicializa el controlador de interrupciones del AT91RM9200 y permite que estas sean atendidas.

```
static struct at91_usbh_data __initdata ecb_at91usbh_data = {
    .ports = 1,
};
```

La estructura *at91_usbh_data* fija el número de puertos USB host a 1 (El encapsulado TQFP solo tiene un puerto USB host, la versión BGA tiene 2).

```
static struct at91_mmc_data __initdata ecb_at91mmc_data = {
    .slot_b = 0,
    .wire4 = 1,
};
```

El SoC AT91RM9200 puede manejar dos memorias SD, la variable *slot_b* determina cual se usa (*slot a* en nuestro caso); la variable *wire4* selecciona el número de líneas de datos entre 1 y 4 (*wire4 = 1*).

```
#if defined(CONFIG.MTD_DATAFLASH)
static struct mtd_partition __initdata my_flash0_partitions[] =
{
    {
        .name = "Darrel-loader",           / 0x0
        .offset = 0,
        .size = 12 * 1056,                  // 12672 bytes
    },
    {
        .name = "uboot",                   // 0x3180
        .offset = MTDPART.OFS_NXTBLK,
        .size = 118 * 1056,
    },
    {
        .name = "kernel",                  // 0x21840
        .offset = MTDPART.OFS_NXTBLK,
        .size = 1534 * 1056, /* 1619904 bytes */
    },
    {
        .name = "filesystem",
        .offset = MTDPART.OFS_NXTBLK,
        .size = MTDPART.SIZ_FULL,
    }
};

static struct flash_platform_data __initdata my_flash0_platform = {
    .name = "Removable flash card",
    .parts = my_flash0_partitions,
    .nr_parts = ARRAY_SIZE(my_flash0_partitions)
};
#endif
```

Muchos dispositivos flash están divididos en secciones que reciben el nombre de particiones; y sin equivalentes a las que se realizan en un disco duro. El subsistema *MTD* (Memory Technology Device) proporciona soporte para estas particiones Flash cuando es seleccionada con la herramienta de

configuración del kernel (`CONFIG_MTD_DATAFLASH = y`). La familia de plataformas `ECB_AT91` tiene un dispositivo DataFlash serial de 16 Mbits (2Mbytes); en este dispositivo creamos 4 particiones en las que se almacenarán: el loader; el u-boot; la imagen del kernel y el sistema de archivos (En el dispositivo flash actual no hay suficiente espacio para este último). Cada partición se define con una estructura *mtd_partition* formada por 3 variables: *name*, *offset* (Dirección inicial de la partición) y *.size*.

```
static struct spi_board_info __initdata ecb_at91spi_devices[] = {
    {
        /* DataFlash chip */
        .modalias      = "mtd_dataflash",
        .chip_select    = 0,
        .max_speed_hz   = 10 * 1000 * 1000,
        .bus_num        = 0,
        .platform_data  = &my_flash0_platform ,
    },
    {
        /* User accessible spi - cs1 (250KHz) */
        .modalias      = "spi-cs1",
        .chip_select    = 1,
        .max_speed_hz   = 250 * 1000,
    },
    {
        /* User accessible spi - cs2 (1MHz) */
        .modalias      = "spi-cs2",
        .chip_select    = 2,
        .max_speed_hz   = 1 * 1000 * 1000,
    },
    {
        /* User accessible spi - cs3 (10MHz) */
        .modalias      = "spi-cs3",
        .chip_select    = 3,
        .max_speed_hz   = 10 * 1000 * 1000,
    },
};
```

La estructura *spi_board_info* define los dispositivos SPI conectados al SoC, esta formada por la variable *modalias*, *chip_select* y *max_speed_hz*. Nuestro dispositivo DataFlash se controla utilizando un puerto SPI, por esto se define la variable *platform_data = &my_flash0_platform*

```
static void __init ecb_at91board_init(void)
{
    /* Serial */
    at91_add_device_serial();

    /* USB Host */
    at91_add_device_usbh(&ecb_at91usbh_data);

    /* I2C */
    at91_add_device_i2c(NULL, 0);

    /* MMC */
    at91_add_device_mmc(0, &ecb_at91mmc_data);

    /* SPI */
    at91_add_device_spi(ecb_at91spi_devices, ARRAY_SIZE(ecb_at91spi_devices));

    /* Programmable Clock 1 */
    at91_set_B_periph(AT91_PIN_PA4, 0);
}
```

Una vez configurados los periféricos utilizados en la plataforma debemos agregar estos dispositivos utilizando las funciones correspondientes *at91_add_device_(serial, usbh, i2c, mmc, spi)* estas funciones hacen un llamado a la función *platform_device_register* la que adiciona el dispositivo a nivel de plataforma.

```

MACHINE_START(ECBAT91, "emQbit's ECB_AT91")
/* Maintainer: emQbit.com */
    .phys_io      = AT91_BASE_SYS,
    .io_pg_offst  = (AT91_VA_BASE_SYS >> 18) & 0xfffc,
    .boot_params  = AT91_SDRAM_BASE + 0x100,
    .timer        = &at91rm9200_timer,
    .map_io       = ecb_at91map_io,
    .init_irq     = ecb_at91init_irq,
    .init_machine = ecb_at91board_init,
MACHINE_END

```

Archivo de Configuración del kernel

Finalmente debemos incluir un archivo de configuración para el kernel (*arch/arm/configs/ecbat91_defconfig*) el cual contiene la configuración inicial que da soporte de forma correcta a todos los periféricos de la plataforma.

A.8.2. Imagen del kernel

En esta sección se realizará una descripción de los pasos necesarios para compilar la imagen del kernel de Linux para la familia de plataforma ECB_AT91; después se realizará un análisis de la imagen, indicando su composición.

Compilación de la Imagen del kernel

El primer paso obvio es obtener la imagen del kernel, la que obtenemos del sitio oficial ftp.kernel.org

```

wget http://ftp.kernel.org/pub/linux/kernel/v2.6/linux-2.6.24.4.tar.bz2
tar xjf linux-2.6.24.4.tar.bz2
cd linux-2.6.24.4

```

A continuación se debe descargar un parche que mantienen los desarrolladores de la familia de SoCs de ATMEL, este parche modifica algunos archivos de la distribución oficial de Linux para dar soporte completo y actualizado a los procesadores ATMEL. Los cambios que se deben realizar en el código fuente, mencionados anteriormente fueron enviados a los encargados de mantener estas actualizaciones, por lo que no es necesario modificar el código fuente para dar soporte a la familia de plataformas ECB_AT91.

```

wget http://maxim.org.za/AT91RM9200/2.6/2.6.24-at91.patch.gz
zcat 2.6.24-at91.patch.gz | patch -p1

```

Finalmente, debemos compilar el kernel utilizando la cadena de herramientas GNU, el resultado de estas compilaciones son ejecutables para una arquitectura ARM, este proceso recibe el nombre de compilación cruzada. El primer paso para el proceso de compilación es hacer visibles los ejecutables de la cadena de herramientas; esto se hace adicionando la ruta donde se encuentran instalados a la variable de entorno *PATH*.

```

export export PATH=$PATH:/home/at91/arm-2007q1/bin/
alias crossmake='make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-'

```

El alias *crossmake* hace que cada vez que se escriba esta palabra el sistema lo remplace por *make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-*, lo que ahorra tiempo al momento de pasar los parámetros a la herramienta *make*; estos parámetros determinan:

1. *ARCH=arm*: Define la arquitectura *arm*.
2. *CROSS_COMPILE=arm-none-linux-gnueabi-*: Indica que se realizará una compilación cruzada y que los ejecutables de la cadena de herramientas (*gcc*, *c++*, *ld*, *objcopy*, *etc*) comienzan con el prefijo *arm-none-linux-gnueabi-* (por lo que el nombre del ejecutable del compilador de C será *arm-none-linux-gnueabi-gcc*)

Una vez declaradas estas variables de entorno debemos generar el archivo oculto *.config*¹³ ejecutando el siguiente comando:

```
crossmake ecbat91_defconfig
o
make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- ecbat91_defconfig
```

Finalmente, damos inicio al proceso de compilación:

```
crossmake -j2 (o make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- -j2)
```

Si no ocurre ningún error, al finalizar debemos observar lo siguiente:

```
LD      vmlinux
SYSMAP  System.map
SYSMAP  .tmp_System.map
OBJCOPY arch/arm/boot/Image
Building modules, stage 2.
MODPOST 1 modules
Kernel: arch/arm/boot/Image is ready
AS      arch/arm/boot/compressed/head.o
CC      drivers/scsi/scsi_wait_scan.mod.o
GZIP     arch/arm/boot/compressed/piggy.gz
LD [M]  drivers/scsi/scsi_wait_scan.ko
CC      arch/arm/boot/compressed/misc.o
AS      arch/arm/boot/compressed/head-at91rm9200.o
AS      arch/arm/boot/compressed/piggy.o
LD      arch/arm/boot/compressed/vmlinux
OBJCOPY arch/arm/boot/zImage
Kernel: arch/arm/boot/zImage is ready
```

y el contenido de los siguientes directorios debe ser algo como:

```
$ ls arch/arm/boot/
bootp compressed Image install.sh Makefile zImage

$ ls vmlinux*
vmlinux vmlinux.o
```

Componentes de la Imagen del kernel

Como nuestro interés en este punto es entender la estructura de la imagen del kernel podemos indicarle a nuestra herramienta de compilación que nos muestre más información del proceso de compilación, esto se hace ejecutando el comando:

¹³este archivo es utilizado por las herramientas de compilación para determinar que soporte fue incluido en el kernel

```
crossmake -j2 V=1
```

Al finalizar el proceso de compilación obtenemos:

```
arm-none-linux-gnueabi-ld -EL -p --no-undefined -X -o vmlinux -T arch/arm/kernel/vmlinux.lds
arch/arm/kernel/head.o          arch/arm/kernel/init_task.o \
init/built-in.o                 \
--start-group                   \
usr/built-in.o                  arch/arm/kernel/built-in.o \
arch/arm/mm/built-in.o          arch/arm/common/built-in.o \
arch/arm/mach-at91/built-in.o   arch/arm/nwfpe/built-in.o  \
kernel/built-in.o              mm/built-in.o                    \
fs/built-in.o                  ipc/built-in.o                 \
security/built-in.o            crypto/built-in.o              \
block/built-in.o               arch/arm/lib/lib.a           \
lib/lib.a                      arch/arm/lib/built-in.o     \
lib/built-in.o                 drivers/built-in.o          \
sound/built-in.o               net/built-in.o              \
--end-group                     \
.tmp_kallsyms2.o
```

En la primera línea del listado anterior (*arm-none-linux-gnueabi-ld -EL -p --no-undefined -X -o vmlinux -T arch/arm/kernel/vmlinux.lds*) se realiza el proceso de enlace del archivo *vmlinux*, utilizando el script de enlace *vmlinux.lds*, incluyendo los objetos (archivos *.o* indicados en la lista. Nótese que el primer archivo enlazado es *head.o*, el cual se genera a partir de *arch/arm/kernel/head.S*, y contiene rutinas de inicialización del kernel. Este proceso se explicará detalladamente más adelante. El siguiente objeto es *init_task.o*, el cual establece estructuras de datos e hilos que utilizará el kernel. A continuación se enlazan una serie de objetos con el nombre *built-in.o*, la ruta del archivo indica la función que realiza el objeto; por ejemplo, el objeto *arch/arm/mm/built-in.o* realiza operaciones de manejo de memoria específicas a la arquitectura ARM. En la Figura A.21 se muestran los componentes de la imagen *vmlinux* y su tamaño para una compilación en particular que nos da una idea de la relación de tamaños.

Cada uno de estos componentes de la imagen del kernel *vmlinux* realizan las siguientes funciones [3]:

1. **arch/arm/kernel/head.o** Código de inicialización del kernel dependiente de la arquitectura.
2. **init_task.o** Hilos y estructuras iniciales utilizadas por el kernel.
3. **init/built-in.o** Código de inicialización del kernel.
4. **usr/built-in.o** Imagen interna *initramfs*.
5. **arch/arm/kernel/built-in.o** Código del kernel específico de la arquitectura.
6. **arch/arm/mm/built-in.o** Código de manejo de memoria específico de la arquitectura.
7. **arch/arm/common/built-in.o** Código genérico específico de la arquitectura.
8. **arch/arm/mach-at91/built-in.o** Código de inicialización específico de la plataforma.
9. **arch/arm/nwfpe/built-in.o** Emulación de punto flotante específico de la arquitectura.
10. **kernel/built-in.o** Código de componentes comunes del kernel.
11. **mm/built-in.o** Código de componentes comunes de manejo de memoria.

2.3K	arch/arm/kernel/head.o
13K	arch/arm/kernel/init_task.o
25K	init/built-in.o
8	usr/built-in.o
87K	arch/arm/kernel/built-in.o
44K	arch/arm/mm/built-in.o
7.4K	arch/arm/common/built-in.o
35K	arch/arm/mach-at91/built-in.o
35K	arch/arm/nwfpe/built-in.o
451K	kernel/built-in.o
255K	mm/built-in.o
1.4M	fs/built-in.o
39K	ipc/built-in.o
4.5K	security/built-in.o
4.5K	crypto/built-in.o
94K	block/built-in.o
44K	arch/arm/lib/lib.a
84K	lib/lib.a
1.1M	drivers/built-in.o
8	sound/built-in.o
1.2M	net/built-in.o

Figura A.21: Componentes de la imagen del kernel de Linux vmlinux

12. **ipc/built-in.o** Comunicación inter-proceso.
13. **security/built-in.o** Componentes de seguridad de Linux.
14. **lib/lib.a** Diferentes funciones auxiliares.
15. **arch/arm/lib/lib.a** Diferentes funciones auxiliares, específicas de la arquitectura.
16. **lib/built-in.o** Funciones auxiliares comunes del kernel.
17. **drivers/built-in.o** Drivers internos o drivers no cargables.
18. **sound/built-in.o** Drivers de sonido.
19. **net/built-in.o** Red de Linux.
20. **.tmp_kallsyms2.o** Tabla de símbolos.

La imagen *vmlinux* generada tiene un tamaño relativamente grande, lo que la hace inconveniente para ser almacenada en un dispositivo Flash, por esta razón se acostumbra comprimirla. El primer paso para reducir el tamaño del ejecutable ELF *vmlinux* es eliminar la información de depuración (notas y comentarios).


```
$ crossmake -j3 vmlinux (4.4 MBytes)
$ arm-none-linux-gnueabi-objcopy -O binary -R .note -R .comment -S vmlinux linux.bin (3.4 MBytes)
$ gzip -c -9 linux.bin > linux.bin.gz (1.6M)
```

A continuación se comprime el archivo resultante utilizando la herramienta *gzip*, como resultado obtenemos un archivo de la tercera parte del archivo original.

A.9. Inicialización del kernel

Como se mencionó anteriormente, el SoC AT91RM9200, posee un programa residente en una pequeña ROM interna que revisa los controladores de memorias flash en busca de un programa válido. En la familia de plataforma ECB_AT91 se utiliza la memoria DataFlash para almacenar: el bootloader, el loader de Linux (u-boot) y la imagen del kernel. La Figura A.22 indica la secuencia de ejecución de aplicaciones cuando se energiza nuestra plataforma.

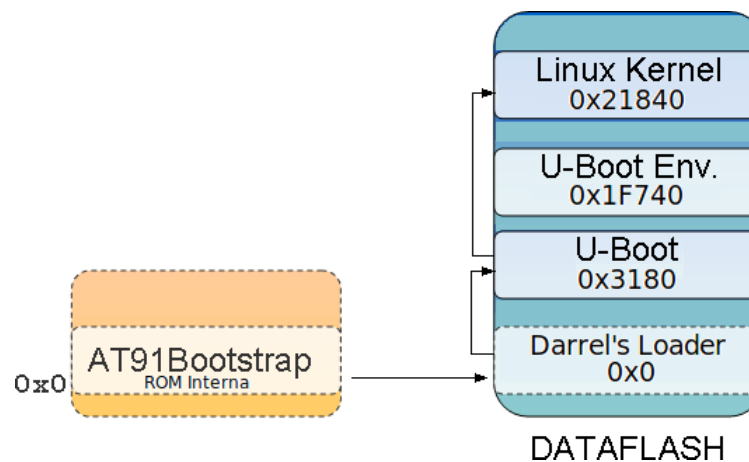


Figura A.22: Flujo de ejecución en la inicialización de plataforma ECB_AT91

La primera aplicación en ejecutarse es el loader de Darrel ¹⁴. Debido a que la RAM interna del AT91RM9200 es de tan solo 16kBytes, es necesario utilizar otra aplicación para poder cargar el *loader* de Linux *U-boot* en la memoria SDRAM externa (con capacidad de de 32Mbytes). Recordemos que inicialmente el SoC no posee ninguna aplicación válida en la memoria DataFlash, por lo que el programa interno de inicialización dará comienzo a una comunicación *Xmodem*, para que se descargue una aplicación a la memoria RAM interna utilizando el puerto de depuración serial a una velocidad de 115200 baudios.

A.9.1. Darrel's Loader

Esta aplicación permite configurar la memoria externa SDRAM, configurar el puerto serie, implementar un protocolo *xmodem* que permita transferir aplicaciones a la memoria SDRAM eterna,

¹⁴Originalmente creado por Darrel Harmon: <http://dlharmon.com/>

controlar la memoria DataFlash y almacenar aplicaciones en ella. El loader de Darrel está basado en u-boot, es una versión reducida de este y permite únicamente las operaciones mencionadas anteriormente, lo que resulta en un archivo adecuado para ser almacenado en la memoria RAM interna (9.3 kBytes).

Es interesante analizar esta aplicación, ya que esta se ejecuta sin ningún soporte de sistema operativo, lo que lo hace ideal para plataformas en las que no se puede ejecutar Linux. Su código fuente lo podemos descargar utilizando el siguiente comando:

```
$svn co http://svn.arhuaco.org/svn/src/emqbit/ECB-AT91-V2/darrell-loader/
```

La estructura del código fuente se muestra en el siguiente listado:

```
|-- include
|   |-- asm
|       |-- arch
|       |-- proc
|       |-- proc-armv
|   |-- linux
|       |-- byteorder
|       |-- mtd
|-- src
```

Antes de estudiar el código fuente demos un vistazo al archivo *Makefile* para ver el proceso de compilación. De ella podemos extraer las partes más interesantes: Los objetos que hacen parte del ejecutable, el proceso de enlazado, y la creación del archivo a descargar en la plataforma:

```
AOBJS      = src/start.o    //ASSEMBLER OBJECTS
COBJS      = src/board.o src/serial.o src/xmodem.o src/dataflash.o src/div0.o \
             src/interrupts.o
LDSCRIPT   := u-boot.lds
LDFLAGS    += -Bstatic -T $(LDSCRIPT) -Ttext $(TEXT_BASE)
OBJCFLAGS  += -mcpu=armv7t -mfloat-abi=hard -mthumb -mthumb-interwork
TEXT_BASE  = 0x00000000

loader:     $(AOBJS) $(COBJS) $(LDSCRIPT)
            $(LD) $(LDFLAGS) $(AOBJS) $(COBJS) \
                -start-group $(PLATFORM_LIBS) -end-group \
                -Map loader.map -o loader

loader.bin: loader
            $(OBJCOPY) ${OBJCFLAGS} -O binary $< $@
```

Como podemos observar, esta aplicación está compuesta por el código en ensamblador *src/start.S* y el código en C *src/board.c src/serial.c src/xmodem.c src/dataflash.c src/div0.c src/interrupts.c*. Para generar el ejecutable *loader* (en formato ELF), encadenamos los objetos AOBJS y COBJS utilizando el script de enlazado *u-boot.lds*. A continuación se muestra el comando ejecutado por las herramientas de compilación al crear el ejecutable *loader*

```
arm-elf-ld -Bstatic -T u-boot.lds -Ttext 0x00000000
src/start.o src/board.o src/serial.o src/xmodem.o src/dataflash.o src/div0.o
src/interrupts.o
--start-group
--no-warn-mismatch -L /home/at91/gnutools/arm-elf/bin/./lib/gcc-lib/arm-elf/3.2.1 -lgcc
--end-group -Map loader.map
-o loader
```

A continuación se muestra el contenido del archivo *u-boot.lds*

```

OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
/*OUTPUT_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")*/
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0x00000000;

    . = ALIGN(4);
    .text :
    {
        src / start.o    (.text)
        *(.text)
    }

    . = ALIGN(4);
    .rodata : { *(.rodata) }

    . = ALIGN(4);
    .data : { *(.data) }

    . = ALIGN(4);
    .got : { *(.got) }

    . = ALIGN(4);
    __bss_start = .;
    .bss : { *(.bss) }
    _end = .;
}

```

Este archivo indica que la primera dirección del ejecutable es la 0x0 y que la primera instrucción en ejecutarse (definida por *ENTRY*) se encuentra en el símbolo *_start* definida en el archivo *start.S*; por otro lado, este script de enlazado nos informa que las secciones *.text* *.rodata* *.data* *.got* y *.bss* serán incluidas en el ejecutable y se encuentran en la misma región de memoria (lo que era de esperarse ya que solo tenemos la memoria interna RAM).

crt0.S (startup.S)

Cuando se crea un ejecutable para ser utilizado en una plataforma específica, es necesario incluir el archivo *crt0.S* (*C runtime startup code*). Este archivo contiene el punto de entrada *_start* y está encargado de las siguientes funciones:

1. Inicialización de las pilas (*stacks*).
2. Copiar el contenido de la sección *.data* (datos inicializados) de la memoria no volátil.
3. Inicializar la sección *.bss*
4. Hacer el llamado al punto de entrada *main* (*start_armboot* para el Darrel's loader)

Y esta compuesto por:

1. **_vectors** Tabla de vectores de excepciones. Deben estar colocados en la dirección 0x0000.
2. **reset_handler** Esta función maneja el reset , y es el punto de entrada principal de un ejecutable. Realiza operaciones de inicialización específica de la arquitectura.

3. **undef_handler** Es el manejador de las instrucciones no definidas.
4. **swi_handler** Manejador de las interrupciones software.
5. **pabort_handler** Manejador de las excepciones *prefetch abort*.
6. **dabort_handler** Manejador de las excepciones *data abort*.
7. **irq_handler** Manejador de las IRQ (Interrupt Request).
8. **fiq_handler** Manejador de las FIQ (Fast Interrupt Request).

El archivo *cr0.S* es escrito en lenguaje ensamblador, y es fuertemente dependiente de la arquitectura, solo programadores expertos con un conocimiento del lenguaje ensamblador pueden escribirlo o modificarlo; sin embargo, los fabricantes proporcionan ejemplos de este tipo de archivos para que pueden ser utilizados sin tener que estudiar profundamente la arquitectura y el lenguaje ensamblador de la misma.

Una vez se han ejecutado las operaciones mencionadas anteriormente se hace un llamado al punto de entrada *start_armboot* (*ldr pc, _start_armboot*)

serial.c, xmpdem.c, dataflash.c, div0.c, interrupts.c

- **serial.c:** Inicializa y configura el puerto serial de depuración (DBGU) a una velocidad de 115200 baudios, 8 bits de datos, paridad par. Adicionalmente proporciona las funciones:
 - *putc*: Transmite un caracter por la interfaz serial de depuración.
 - *puts*: Transmite una cadena de caracteres.
 - *getc*: Recibe un caracter por la interfaz serial de depuración.
- **xmodem.c:** Implementa la recepción serial utilizando el protocolo *xmodem*.
- **dataflash.c:** Inicializa el puerto SPI, y proporciona funciones de alto nivel para la escritura de la DataFlash.
 - *write_dataflash(addr_dest, addr_src, size)*
- **div0.c:** Remplazo (dummy) del manejador división por cero de GNU/Linux.
- **interrupts.c:** Proporciona funciones para el manejo de interrupciones e implementación de retardos.

board.c

El archivo *board.c* proporciona el punto de entrada *start_armboot* y realiza las siguientes operaciones:

- Configuración del controlador de manejo de potencia (PMC).
- Hace un llamado a la inicialización del puerto serie de depuración *serial_init()*
- Configura la SDRAM externa *try_configure_sdram*
- Despliega un menú de funciones.
- Realiza las operaciones del menú.

Ejecución del loader

Una vez se carga el archivo *loader.bin*, el que resulta de las secciones *text* y *rodata* del ejecutable tipo ELF:

```
arm-elf-objcopy --gap-fill=0xff -O binary loader loader.bin
```

El programa desplegará el siguiente menú:

```
Darrell's loader - Thanks to the u-boot project
Version 1.0. Build Jul 29 2007 12:04:04
RAM:32MB

1: Upload Darrell's loader to Dataflash
2: Upload u-boot to Dataflash
3: Upload Kernel to Dataflash
4: Start u-boot
5: Upload Filesystem image
6: Memory test
```

La opción 1. del menú permite almacenar el archivo *loader.bin* en la memoria DataFlash, una vez hecho esto, el programa de inicialización almacenado en la memoria ROM interna lo ejecutará cada vez que se reinicie la plataforma.

Las opciones 2 y 3 son similares a la 1, solo que se cargan las aplicaciones en diferentes direcciones de memoria, observemos el código fuente que implementa estas opciones:

```
else if(key == '2'){
    puts("Please transfer u-boot.bin via Xmodem\n\0");
    len = rxmodem((char *)0x20000000);
    AT91F_DataflashInit ();
    dataflash_print_info ();
    if(write_dataflash(DATAFLASH_UBOOT_BASE, 0x20000000, len))
        puts("Dataflash write successful\n");
    dispmenu = 1;
}
```

Acá vemos como que al elegir la opción 2, se despliega un mensaje indicándole al usuario que transmita el archivo *u-boot.bin* utilizando el protocolo *xmodem*, después se ejecuta la función *rxmodem* la que recibe los datos por el serial y lo almacena en la SDRAM externa (dirección 0x20000000) y finalmente se almacena esta información en la dirección *DATAFLASH_UBOOT_BASE* de la memoria Dataflash.

La opción 4. del menú transfiere la ejecución al cargador de Linux *U-boot*:

```
else if(key == '4' || ((scans > 300000) && autoboot)){
    if(AT91F_DataflashInit()){
        dataflash_print_info();
        if(read_dataflash(DATAFLASH_UBOOT_BASE, 0x1C000, (char *)0x20700000)){
            puts("Dataflash read successful: Starting U-boot\n");
            asm("ldr pc, =0x20700000");
        }
    }
}
```

En esta opción se copian 0x1C000 bytes del contenido de la memoria DataFlash comenzando en la posición *DATAFLASH_UBOOT_BASE* a la dirección de memoria 0x20700000 (SDRAM externa) y luego se carga el contador de programa con la dirección 0x20700000, con lo que se inicia la ejecución del *U-boot*;

es importante hacer notar que el programa debe ser enlazado para que las secciones estén en la posición de memoria donde serán ejecutadas (0x20700000) y no en la dirección donde son almacenadas.

A.9.2. U-boot

U-boot es un *bootloader* que permite cargar archivos utilizando una gran variedad de periféricos como: puerto serie, memoria SD, memorias flash paralelas, seriales, NAND, NOR, o puerto ethernet. Es capaz de iniciar una variedad de tipos de archivos, además de un formato especial propio que almacena información sobre el tipo de sistema operativo, la dirección de carga, el punto de entrada, verificación de integridad via CRC, tipos de compresión, y textos descriptivos. La siguiente información es desplegada cuando *u-boot* inicializa una imagen del kernel de Linux.

```
## Booting image at c0021840 ...
Image Name:      Linux Kernel Image
Image Type:      ARM Linux Kernel Image (gzip compressed)
Data Size:       1550369 Bytes = 1.5 MB
Load Address:    20008000
Entry Point:     20008000
Verifying Checksum ... OK
Uncompressing Kernel Image ... OK
```

Para crear una imagen con el formato *U-boot* basta con ejecutar el siguiente comando:

```
$ mkimage -A arm -O linux -T kernel -C gzip -a 0x20008000 -e 0x20008000 \
-n "Linux Kernel Image" -d linux.bin.gz ecb_at91.img
```

Acá definimos que nuestra arquitectura es *ARM*, el sistema operativo es *Linux*, que se trata de una imagen tipo *Kernel*, se utilizó *gzip* como método de compresión, la dirección de carga y punto de entrada (son iguales para kernels superiores al 2.3.X) es *0x20008000*, el nombre de la imagen es *Linux Kernel Image*, *linux.bin.gz* es el archivo de entrada y *ecb_at91.img* el archivo de salida. Esta información es almacenada en el encabezado de la imagen y puede ser leído utilizando el comando:

```
$mkimage -l /home/at91/binaries/ecb_at91.img

Image Name:      Linux Kernel Image
Created:         Fri Jun 26 09:26:03 2009
Image Type:      ARM Linux Kernel Image (gzip compressed)
Data Size:       1550369 Bytes = 1514.03 kB = 1.48 MB
Load Address:    0x20008000
Entry Point:     0x20008000
```

El siguiente listado muestra la estructura del encabezado definida por *u-boot*, en ella podemos observar sus componentes y el tamaño de cada uno de ellos.

```
typedef struct image_header {
    uint32_t      ih_magic;      /* Image Header Magic Number */
    uint32_t      ih_hcrc;      /* Image Header CRC Checksum */
    uint32_t      ih_time;      /* Image Creation Timestamp */
    uint32_t      ih_size;      /* Image Data Size */
    uint32_t      ih_load;      /* Data Load Address */
    uint32_t      ih_ep;        /* Entry Point Address */
    uint32_t      ih_dcrc;      /* Image Data CRC Checksum */
    uint8_t       ih_os;        /* Operating System */
    uint8_t       ih_arch;      /* CPU architecture */
    uint8_t       ih_type;      /* Image Type */
    uint8_t       ih_comp;      /* Compression Type */
}
```

```
uint8_t      ih_name[IH_NMLEN];      /* Image Name      */
} image_header_t;
```

A.9.3. Portando U-boot a la familia de plataformas ECB_AT91

Es necesario modificar varios archivos para que *u-boot* soporte la familia de plataformas ECB_AT91:

Makefile

Se le debe indicar a las herramientas de compilación la nueva plataforma, adicionando las siguientes líneas.

```
ecb_at91_config      :      unconfig
    @$(MKCONFIG) $(@: _config=) arm arm920t ecb_at91 NULL at91rm9200
```

MAKEALL

Se debe agregar la siguiente entrada en el archivo MAKEALL:

```
LIST_ARM9="      \
-----
    at91rm9200dk cmc_pu2 ecb_at91
```

board/ecb_at91/Makefile

En el directorio *board/ecb_at91* se alojan los archivos que dan soporte a la nueva arquitectura, el archivo de reglas de configuración se muestra a continuación:

```
include $(TOPDIR)/config.mk

LIB      = $(obj)lib$(BOARD).a

COBJS    := $(BOARD).o at45.o flash.o

SRCS     := $(SOBJS:.o=.S) $(COBJS:.o=.c)
OBJS     := $(addprefix $(obj),$(COBJS))
SOBJS    := $(addprefix $(obj),$(SOBJS))

$(LIB): $(obj).depend $(OBJS) $(SOBJS)
    $(AR) $(ARFLAGS) $@ $(OBJS) $(SOBJS)

clean:
    rm -f $(SOBJS) $(OBJS)

distclean:
    clean
    rm -f $(LIB) core *.bak .depend

# defines $(obj).depend target
include $(SRCTREE)/rules.mk
sinclude $(obj).depend
```

Este es el formato utilizado para todas las plataformas, la parte importante se encuentra en la definición de la variable *COBJS*; en la que se incluyen los archivos: *ecb_at91.o*, archivo de la plataforma; *at45.o*, soporte a las memorias DataFlash AT45 y *flash.o*, soporte genérico para dispositivos flash¹⁵.

board/ecb_at91/board.c

Este archivo contiene toda la configuración específica de la plataforma, y como se puede ver en el siguiente listado, fija el número de la plataforma a: *MACH_TYPE_ECBAT91* (1072 definida en *include/asm-arm/mach-types.h*) y los parámetros del boot en: *PHYS_SDRAM + 0x100*; (*PHYS_SDRAM* = 0x20000000 está definido en *include/configs/ecb_at91.h*). Adicionalmente, inicializa la SDRAM, la interfaz de red y la memoria DataFlash.

```
#include <common.h>
#include <asm/arch/AT91RM9200.h>
#include <at91rm9200_net.h>
#include <lxt972.h>

DECLARE_GLOBAL_DATA_PTR;

/*
 * Miscellaneous platform dependant initialisations
 */

void lowlevel_init(void)
{
    /* Required by assembly functions — do nothing */
}

int board_init (void)
{
    /* Enable Ctrlc */
    console_init_f ();

    /* arch number of ECB-AT91 board */
    gd->bd->bi_arch_number = MACH_TYPE_ECBAT91;

    /* adress of boot parameters */
    gd->bd->bi_boot_params = PHYS_SDRAM + 0x100;

    return 0;
}

int dram_init (void)
{
    gd->bd->bi_dram[0].start = PHYS_SDRAM;
    gd->bd->bi_dram[0].size = PHYS_SDRAM_SIZE;
    return 0;
}

#ifdef CONFIG_DRIVER_ETHER
#ifdef CONFIG_COMMANDS & CFG_CMD_NET

/*
 * Name:
 *     at91rm9200_GetPhyInterface
 * Description:

```

¹⁵Estos archivos pueden ser descargados de: http://svn.arhuaco.org/svn/src/emqbit/ECB-AT91_V2/u-boot/u-boot-1.1.6-ecbat91.patch


```

*   Initialise the interface functions to the PHY
* Arguments:
*   None
* Return value:
*   None
*/

void at91rm9200_GetPhyInterface(AT91PS_PhyOps p_phyops)
{
    p_phyops->Init = lxt972_InitPhy;
    p_phyops->IsPhyConnected = lxt972_IsPhyConnected;
    p_phyops->GetLinkSpeed = lxt972_GetLinkSpeed;
    p_phyops->AutoNegotiate = lxt972_AutoNegotiate;
}

#endif /* CONFIG_COMMANDS & CFG_CMD_NET */
#endif /* CONFIG_DRIVER_ETHER */

#ifdef CONFIG_HAS_DATAFLASH
#include <dataflash.h>

void AT91F_DataflashMapInit(void)
{
    static int cs[][CFG_MAX_DATAFLASH_BANKS] = {
        /* Logical address, CS */
        {CFG_DATAFLASH_LOGIC_ADDR_CS0, 0},
    };

    static dataflash_protect_t area_list[NB_DATAFLASH_AREA] = {
        /* define the dataflash offsets */
        {DATAFLASH_LOADER_BASE /* 0 */, DATAFLASH_UBOOT_BASE - 1,
         FLAG_PROTECT_SET, "Darrell loader"},
        {DATAFLASH_UBOOT_BASE, DATAFLASH_ENV_UBOOT_BASE - 1,
         FLAG_PROTECT_SET, "U-boot"},
        {DATAFLASH_ENV_UBOOT_BASE, DATAFLASH_KERNEL_BASE - 1,
         FLAG_PROTECT_CLEAR, "Environment"},
        {DATAFLASH_KERNEL_BASE, DATAFLASH_FILESYSTEM_BASE - 1,
         FLAG_PROTECT_CLEAR, "Kernel"},
        {DATAFLASH_FILESYSTEM_BASE, 0x1ffff, FLAG_PROTECT_SET, "Filesystem"},
    };

    AT91F_MapInit(cs, area_list);
}

#endif

```

include/configs/ecb_at91.h

Este archivo contiene variables que son utilizadas para la inicialización de la plataforma, algunas de ellas definen el valor de registros de configuración de periféricos como: el controlador del reloj del sistema, controlador de memorias SDRAM y memorias Flash. A continuación se muestra un segmento de este archivo, en el que se define la arquitectura:

```

/* ARM asynchronous clock */
#define AT91C_MAIN_CLOCK      18000000
#define AT91C_MASTER_CLOCK   60000000

#define AT91_SLOW_CLOCK       32768    /* slow clock */

```

```
#define CONFIG_ARM920T      1      /* This is an ARM920T Core */
#define CONFIG_AT91RM9200   1      /* It's an Atmel AT91RM9200 SoC */
#define CONFIG_ECB_AT91     1      /* on an AT91RM9200DK Board */
#define CONFIG_USE_IRQ      1      /* we don't need IRQ/FIQ stuff */
#define USE_920T_MMU        1

#define CONFIG_CMDLINE_TAG   1      /* enable passing of ATAGs */
#define CONFIG_SETUP_MEMORY_TAGS 1
#define CONFIG_INITRD_TAG    1

#ifndef CONFIG_SKIP_LOWLEVEL_INIT
#define CFG_LONGHELP
```

Este archivo también incluye el valor predeterminado de las variables de entorno utilizadas por *u-boot*:

```
#define CONFIG_BOOTARGS      "mem=32M root=/dev/mmcblk0p1 rootfstype=ext3 console=ttyS0,
                             115200n8 rootdelay=1"
#define CONFIG_ETHADDR      00:00:00:00:00:5b
#define CONFIG_NETMASK      255.255.255.0
#define CONFIG_IPADDR       192.168.0.135
#define CONFIG_SERVERIP     192.168.0.128
#define CONFIG_BOOTDELAY     2
#define CONFIG_BOOTCOMMAND   "bootm C0021840"
#define CONFIG_BOOTFILE      "ecb_at91.img"
#define CONFIG_ROOTPATH      "/home/at91/rootfs"
#define CONFIG_LOADADDR      0x20200000
```

La variable *bootargs* son parámetros pasados al kernel en su inicialización, y en este ejemplo fija la memoria RAM en 32 Mbytes, indica que el sistema de archivos se encuentra en el dispositivo */dev/mmcblk0p1* y utiliza el sistema de archivos *ext3*, la consola es el dispositivo serial */dev/ttyS0* configurado a una velocidad de 115200 baudios, bit de paridad y 8 bits de datos y que debe esperar 1 segundo para montar el sistema de archivos.

Las variables *ethaddr*, *netmask*, *ipaddr*, *serverip* configuran la interfaz de red y las direcciones IP de la plataforma y de un servidor de donde puede descargarse la imagen del kernel. La variable *bootdelay* fija el número de segundos que esperará u-boot para ejecutar el comando almacenado en *bootcmd*, este conteo puede detenerse para interactuar con u-boot. El comando *bootm C0021840* (*bootcmd*) es utilizado para iniciar la imagen almacenada en la dirección de memoria 0xC0021840 (dirección donde almacena el loader de Darrell la imagen del kernel), *bootm* utiliza la información almacenada en el encabezado de la imagen para utilizar el método de descompresión adecuado, almacenarlo en la dirección requerida y pasarle los parámetros almacenados en *bootcmd*.

```
#define CONFIG_COMMANDS      \
    ((CONFIG_CMD_DFL | CFG_CMD_NET | CFG_CMD_PING | CFG_CMD_DHCP) & \
     ~(CFG_CMD_BDI | CFG_CMD_FPGA | CFG_CMD_MISC))

/* Remember that you must have the same mapping in the Darrell loader */
#define NB_DATAFLASH_AREA    5 /* protected areas (4 + u-boot env) */
#define DATAFLASH_MAX_PAGESIZE 1056
#define DATAFLASH_LOADER_BASE (0*DATAFLASH_MAX_PAGESIZE)
#define DATAFLASH_UBOOT_BASE  (12*DATAFLASH_MAX_PAGESIZE)
#define DATAFLASH_ENV_UBOOT_BASE (122*DATAFLASH_MAX_PAGESIZE)
#define DATAFLASH_KERNEL_BASE (130*DATAFLASH_MAX_PAGESIZE)
#define DATAFLASH_FILESYSTEM_BASE (1664*DATAFLASH_MAX_PAGESIZE)
```

Compilación de U-boot en la familia de plataformas ECB_AT91

A continuación se indican los pasos necesarios para generar el archivo *u-boot.bin* que será almacenado en la memoria DataFlash por el loader de Darrel.

El primer paso es obviamente descargar el código fuente de la versión 1.1.6 de <http://sourceforge.net/projects/u-boot/>, después debemos descargar el patch que da soporte a la familia de plataformas ECB_AT91:

```
$ wget http://svn.arhuaco.org/svn/src/emqbit/ECB_AT91_V2/u-boot/u-boot-1.1.6-ecbat91.patch
$ tar xjf u-boot-1.1.6.tar.bz2
$ cd u-boot-1.1.6
```

Aplicamos el patch:

```
$ cat ../u-boot-1.1.6-ecbat91.patch | patch -p1
```

Configuramos y generamos las herramientas utilizadas por *u-boot* (entre ellas *mkimage*).

```
$ make ecb_at91_config
Configuring for ecb_at91 board... (Este es un comentario)
$ make tools
```

Por último compilamos *u-boot*:

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
o
crossmake (si el alias crossmake está definido)
```

Si el proceso se siguió correctamente y no se presentan errores al final del proceso obtenemos el mensaje:

```
arm-none-linux-gnueabi-objcopy --gap-fill=0xff -O srec u-boot u-boot.srec
arm-none-linux-gnueabi-objcopy --gap-fill=0xff -O binary u-boot u-boot.bin
```

Lo que nos indica que se generó exitosamente el archivo *u-boot.bin*; el que puede ser descargado a la memoria Dataflash de la plataforma utilizando la opción 2 del menú del loader de Darrel (*Upload u-boot to Dataflash*) utilizando el protocolo xmodem.

Podemos verificar que el archivo fue generado y almacenado en la Dataflash al ejecutar la opción número 4 del menú del loader de Darrel (*Start u-boot*), como se mencionó anteriormente esta opción copia el archivo *u-boot* desde la DataFlash a la memoria SDRAM y es ejecutado desde allí, con lo que se desplegará el siguiente mensaje:

```
Dataflash read successful: Starting U-boot
U-Boot 1.1.6 (Jul 29 2007 - 12:12:38)

DRAM: 32 MB
Atmel: Flash: 0 kB
DataFlash: AT45DB161
Nb pages: 4096
Page Size: 528
Size= 2162688 bytes
Logical address: 0xC0000000
Area 0: C0000000 to C000317F (RO) Darrell loader
Area 1: C0003180 to C001F73F (RO) U-boot
Area 2: C001F740 to C002183F Environment
Area 3: C0021840 to C01ACFFF Kernel
Area 4: C01AD000 to C020FFFF (RO) Filesystem
```

```
In:    serial
Out:   serial
Err:   serial
PHY not connected!!
Hit any key to stop autoboot:  2
```

Si se presiona cualquier tecla antes de que el contador llegue a 0, podemos interactuar con *u-boot*, si ejecutamos el comando *print* (Despliega en pantalla las variables de entorno definidas):

```
bootcmd=bootm C0021840
bootdelay=2
baudrate=115200
ethaddr=00:00:00:00:00:5b
ipaddr=192.168.0.135
serverip=192.168.0.128
rootpath="/home/at91/rootfs"
netmask=255.255.255.0
bootfile="ecb_at91.img"
loadaddr=0x20200000
bootargs=mem=32M root=/dev/mmcblk0p1 rootfstype=ext3 console=ttyS0,115200n8 rootdelay=1
stdin=serial
stdout=serial
stderr=serial
Environment size: 345/8188
```

Podemos cambiar el valor predeterminado de la variable *bootdelay* a 1:

```
ecb_at91> setenv bootdelay 1
```

Y almacenamos los cambios realizados en una sección de la flash reservada para este fin con el comando:

```
ecb_at91> save
Saving Environment to dataflash...
```

s

Podemos generar una nueva variable de entorno y almacenarla en la DataFlash:

```
ecb_at91> setenv nfsargs=mem=32M console=ttyS0,115200n8 root=/dev/nfs
nfsroot=192.168.0.128:/home
/at91/rootfs,timeo=200,retrans=500 ip=::: eth0:on
ecb_at91> save
```

A.9.4. Almacenamiento de la imagen del kernel

Una vez creada la imagen del kernel (*ecb_at91.img*) con el formato de *U-boot* debemos probar su correcto funcionamiento; esto lo podemos hacer de dos formas: Almacenándola directamente en una memoria no volátil o cargándola en la memoria RAM y ejecutándola desde allí.

Almacenamiento en la memoria DataFlash

Cuando almacenamos la imagen del kernel de Linux a un medio de almacenamiento no volátil, debemos tener presente que los ciclos de borrado y escritura de este toman un tiempo mucho mayor que

en el caso de las memorias no volátiles, por esto, se recomienda esta opción cuando ya se cuente con una imagen estable o cuando no existan otros medios (como en el caso de la plataforma ECBOT).

Inicialmente debemos ejecutar el loader de Darrel, esto se hace presionando el pulsador de *reset* disponible en todas las plataformas de la familia *ECB_AT91*. Inmediatamente después de observar el menú del loader debemos oprimir cualquier tecla para interrumpir la ejecución automática del *u-boot*.

Seleccionando la opción del menú: 3: *Upload linux to Dataflash*, podemos iniciar la transferencia de la imagen a la memoria SDRAM de nuestra plataforma:

```
Please transfer linux via Xmodem
Receiving Xmodem transfer
```

Cuando aparezca este mensaje se debe transmitir el archivo *ecb_at91.img* utilizando el protocolo *xmodem*. Unos minutos después la transferencia finaliza; sin embargo, debemos esperar a que la información sea almacenada en la memoria DataFlash, mientras se completa la escritura la consola no mostrará ninguna actividad, eso es normal y no se debe reiniciar la placa. Una vez finalizada la escritura observaremos el mensaje:

```
Dataflash write successful
```

Almacenamiento en la memoria RAM

El proceso de grabación en la memoria DataFlash puede tomar alrededor de 6 minutos, por lo que no es conveniente utilizarlo cuando se está tratando de crear una imagen propia o se están realizando cambios a la misma. Cuando necesitamos modificar esta imagen ya sea porque queremos hacerlo nosotros mismos o porque deseamos una versión de kernel más moderna, es preferible utilizar un método de transferencia más rápido.

La plataforma *ECB_AT91* posee una interfaz de red que puede ser controlada por *u-boot*. Utilizando el protocolo *tftp U-boot* puede descargar la imagen desde un servidor a la memoria SDRAM y ejecutarla desde allí, ese proceso se realiza en segundos, facilitando de esta forma el proceso de desarrollo. A continuación se describen los pasos que deben seguirse para realizar esta operación:

Primero debemos instalar y configurar el servidor *tftp* en el computador donde se tiene las herramientas de desarrollo:

```
$ aptitude install tftpd tftp.
```

Se debe agregar la siguiente línea al archivo */etc/inetd.conf*

```
tftp dgram udp wait nobody /usr/sbin/tcpd /usr/sbin/in.tftpd /srv/tftp
```

Debe asegurarse que el protocolo *tftp* utiliza el puerto *UDP 69*

```
$ cat /etc/services | grep tftp
tftp          69/udp
```

Ahora creamos el directorio */srv/tftp*¹⁶ y colocamos la imagen en él:

```
$ mkdir /srv/tftp /
$ chown myuser. /srv/tftp /
$ cp ecb_at91.img /srv/tftp /
```

¹⁶Este directorio tiene restricciones de seguridad, debe tener permisos de super-usuario para hacer esto

Para verificar la correcta configuración del servidor, podemos utilizar un cliente *tftp*:

```
$ cd /tmp/
$ tftp localhost # from the server
tftp> get ecb_at91.img
Received 1319525 bytes in 0.2 seconds
tftp> quit
```

Ahora se deben configurar algunas variables de entorno en nuestra plataforma para indicarle a *u-boot* la dirección *IP*, el nombre y la ubicación de la imagen del kernel. Estas variables deben ser modificadas para que contengan los siguientes valores:

```
loadaddr=0x20200000
bootdelay=1
bootfile="ecb_at91.img"
fileaddr=20200000
gatewayip=192.168.0.1
netmask=255.255.255.0
serverip=192.168.0.128
```

Estas variables fijan la dirección *IP* de: la plataforma a *192.168.0.2*, del gateway a *192.168.0.1*, la del servidor *tftp* a *192.168.0.128*¹⁷. Adicionalmente define el nombre de la imagen del kernel a *ecb_at91.img*. Una vez configurado *U-boot* podemos descargar la imagen a la dirección de memoria *0x20200000*:

```
ecb_at91 >tftp
TFTP from server 192.168.0.1; our IP address is 192.168.0.2
Filename 'ecb_at91.img'.
Load address: 0x20200000
Loading: #####
#####
#####
#####
#####
done
Bytes transferred = 1409031 (158007 hex)
ecb_at91 >
```

A.9.5. Inicialización del Kernel

En general los cargadores de Linux como el *u-boot* realizan las siguientes funciones:

- **Configurar e inicializar la RAM**
- **Inicializar un puerto serial**
- **Detectar el tipo de máquina:** El boot loader debe proporcionar un valor *MACH_TYPE_XXX* al kernel, como vimos anteriormente tanto *u-boot* como *Linux* fijan este valor a 1072.
- **Configurar la *kernel tagged list*:** El boot loader debe crear e inicializar una estructura llamada *kernel tagged list*, la que comienza con *ATAG_CORE* y finaliza con *ATAG_NONE*. El tag *ATAG_CORE* puede o no estar desocupado, si se encuentra desocupado debe fijar el campo *size* en '2'. El campo *size* del tag *ATAG_NONE* debe fijarse en 0. Se pueden definir cualquier número de tags, pero se deben definir por lo menos el tamaño y la localización de la memoria del sistema, y la localización del sistema de

¹⁷dirección IP del PC donde están las herramientas de desarrollo

archivos. Esta *tagged list* debe ser almacenada en RAM, en una región que no pueda ser modificada por el programa que descomprime el kernel o por el programa *initrd*. Se recomienda colocarlo en los primeros 16kBytes de la RAM.

- Hacer un llamado a la imagen del kernel: Existen dos formas de hacer este llamado, directamente desde la flash o en cualquier posición de la RAM. Los dos métodos deben cumplir las siguientes condiciones:
 - Desactiva los dispositivos que tienen capacidad de DMA, de tal forma que la memoria no se corrompa.
 - Fijar los registros de la CPU: $r0 = 0$, $r1 = \text{tipo de máquina}$, $r2 = \text{dirección física de la tagged list en RAM}$.
 - Modo de la CPU: Deshabilitar todas las interrupciones (IRQs y FIQs) y colocar a la CPU en modo SVC.
 - Caches, MMU: Debe estar desactivada la MMU, La cache de instrucciones puede estar activada o desactivada, la cache de datos debe estar desactivada.

Llamado a la Imagen del kernel

Como mencionamos anteriormente, *u-boot* ejecuta las instrucciones almacenadas en la variable de entorno *bootcmd*; que para la familia de plataformas ECB-AT91 almacena el comando *bootm C0021840*, esta instrucción le indica a *u-boot* que ejecute el comando *bootm* con una imagen almacenada en la posición de memoria *0xC0021840*, en la que (como mencionamos anteriormente (Figura A.22)) se almacena la imagen del kernel. El código que implementa el comando *bootm* se encuentra en el archivo *common/cmd_bootm.c*; analizando este archivo podemos descubrir el proceso que realiza *u-boot* al hacer el llamado a la imagen del kernel (la que almacenamos utilizando la opción 3 del loader de Darrel). La función *do_bootm* realiza las siguientes operaciones:

- Verificar la existencia de un número mágico en los primeros 4 bytes de la imagen (0x27051956). Si no se encuentra este número se desplegará el mensaje: *Bad Magic Number*
- Verifica la integridad del encabezado de la imagen. De no pasar esta prueba se mostrará el mensaje: *Bad Header Checksum*.
- Imprime el encabezado de la imagen.
- Calcula el CRC del archivo almacenado y lo compara con el almacenado en la cabecera de la imagen. Si no se supera esta prueba, se desplegará el mensaje: *Bad Data CRC*
- Comprueba que la arquitectura está soportada por *u-boot*.
- Descomprime la imagen almacenada en la dirección *load_address* (la cual se pasa como parámetro en el momento de la creación de la imagen).
- Transferir el control a Linux en la función *do_bootm_linux* *U-boot es un loader que permite trabajar con: LYNXOS, RTEMS, VXWORKS, QNX, ARTOS, NETBSD*

La función *do_bootm_linux* hace el llamado a la imagen del kernel utilizando el siguiente comando:

```
(*kernel) (kbd, initrd_start, initrd_end, cmd_start, cmd_end);
```

en donde:

- *kbd* Información de la plataforma de desarrollo:

```
typedef struct bd_info {
    int          bi_baudrate;    /* serial baudrate */
    unsigned long bi_ip_addr;    /* IP Address */
    unsigned char bi_enetaddr[6]; /* Ethernet address */
    struct environment_s *bi_env;
    ulong        bi_arch_number; /* id for this board */
    ulong        bi_boot_params; /* boot params */
    struct       /* RAM configuration */
    {
        ulong start;
        ulong size;
    } bi_dram[CONFIG_NR_DRAM_BANKS];
} bd_t; \end{itemize}
```

- *initrd_start* - *initrd_end*: Linux permite que el sistema de archivos sea almacenado en la memoria RAM, el sistema es almacenado en algún medio no volátil y después es descomprimido en la RAM, esto acelera la ejecución ya que como se mencionó anteriormente, el acceso a las memorias volátiles es mucho menor. *initrd_start* - *initrd_end* indican el inicio y fin de este archivo
- *cmd_start* - *cmd_end*: Posición de memoria donde se almacenan los parámetros pasados al kernel (*mem=32M root=/dev/mmcblk0p2 rootfstype=ext3 console=ttyS0,115200n8 rootdelay=1*)

En este punto termina el trabajo de *u-boot* y el control es pasado al kernel. Como pudimos darnos cuenta lo más atractivo de *u-boot* es su capacidad para manejar diferentes dispositivos de almacenamiento no volátiles como memorias flash, memorias SD y su capacidad para manejar interfaces de red y permitir utilizarlas para al carga de imágenes del kernel.

Punto de Entrada del Kernel *head.o*

Como puede verse en A.21 el primer archivo encadenado en la imagen del kernel es *arch/arm/kernel/head.o*, y corresponde al punto de entrada del kernel de Linux, este archivo ejecuta las siguientes funciones:

1. Verificar que la arquitectura y el procesador sean válidos. Si el procesador no es válido se generará un error y en la consola aparecerá una “p”, si la plataforma no corresponde se genera un error y se imprimirá una “a” en la consola.
2. Se genera una estructura de datos (*page table*) que almacena el mapeo entre las direcciones de memoria virtual y la memoria física. Antes de pasar el control al kernel, el procesador corre en modo *real*, en el que las direcciones corresponden a direcciones reales de los dispositivos conectados físicamente al procesador.
3. Activa la unidad de manejo de memoria (MMU) del procesador. Cuando se activa la MMU el esquema de memoria físico se reemplaza por un direccionamiento virtual determinado por los desarrolladores del kernel.
4. Establece un limitado mecanismo de detección y reporte de errores.
5. Hace un llamado a la función *start_kernel* en *init/main.c*


```

setup_arch(&command_line);
setup_command_line(command_line);
sched_init();
preempt_disable();
page_alloc_init();
console_init();
mem_init();
kmem_cache_init();
setup_per_cpu_pageset();
numa_policy_init();
calibrate_delay();
pidmap_init();
pgtable_cache_init();
prio_tree_init();
anon_vma_init();
fork_init(num_physpages);
proc_caches_init();
buffer_init();
unnamed_dev_init();
key_init();
security_init();
vfs_caches_init(num_physpages);
radix_tree_init();
signals_init();

page_writeback_init();
proc_root_init();
cgroup_init();
cpuset_init();
taskstats_init_early();
delayacct_init();
acpi_early_init();
schedule();
preempt_disable();

```

En los últimos pasos en el proceso de arranque de Linux, se libera la memoria que será utilizada por los procesos de inicialización, abre un dispositivo que permita la interacción con el usuario `/dev/console` (consola serial en nuestro caso) y ejecuta el primer proceso en espacio de usuario `init`. El siguiente listado muestra el código que implementa esta última fase del proceso de arranque.

```

static int noinline init_post(void)
{
    free_initmem();
    unlock_kernel();
    mark_rodata_ro();
    system_state = SYSTEM_RUNNING;
    numa_default_policy();

    if (sys_open((const char __user *) "/dev/console", ORDWR, 0) < 0)
        printk(KERN_WARNING "Warning: unable to open an initial console.\n");

    (void) sys_dup(0);
    (void) sys_dup(0);

    if (ramdisk_execute_command) {
        run_init_process(ramdisk_execute_command);
        printk(KERN_WARNING "Failed to execute %s\n",
                ramdisk_execute_command);
    }

    /*

```

```

    * We try each of these until one succeeds.
    *
    * The Bourne shell can be used instead of init if we are
    * trying to recover a really broken machine.
    */
    if (execute_command) {
        run_init_process(execute_command);
        printk(KERN_WARNING "Failed to execute %s. Attempting "
                        "defaults...\n", execute_command);
    }
    run_init_process("/sbin/init");
    run_init_process("/etc/init");
    run_init_process("/bin/init");
    run_init_process("/bin/sh");

    panic("No init found. Try passing init= option to kernel.");
}

```

Como podemos observar el último paso consiste en el llamado a un archivo en espacio de usuario llamado *init* o *sh*, en la siguiente subsección se describirán las acciones que se realizan cuando se ejecuta este archivo. De no encontrarse se desplegará el mensaje *No init found. Try passing init= option to kernel.* y la plataforma pasará a un estado de inactividad.

A.10. Inicialización del Sistema

En esta sección describiremos el proceso de inicialización de la plataforma embebida, en la sección anterior se estudió la inicialización del kernel de Linux. En esta sección se realizará una descripción del sistema de archivos que contiene aplicaciones que Linux requiere para inicializar servicios como los de red y la consola, cargar drivers (*módulos*) de dispositivos y montar sistemas de archivos adicionales.

A.10.1. Sistema de Archivos

Anteriormente hemos hecho referencia a la localización de la raíz del sistema de archivos (*root*), e indicamos que está se encuentra en una memoria no volátil. Esta memoria debe estar formateada en alguno de los siguientes sistemas de archivos:

- **Second Extended File System (ext2)** : Este sistema de archivos utiliza bloques como unidad de almacenamiento básico, inodes como medio para mantener un seguimiento de archivos y objetos de sistema, grupos de bloques para dividir lógicamente el disco en secciones más manejables, directorios para proporcionar una organización jerárquica de archivos, bloques y mapas de bits (*bitmap*) de bloques e inodes para mantener un seguimiento de bloques e inodes asignados, y superbloques para definir los parámetros del sistema de archivos y su estado general. Adicionalmente posee la capacidad de crear enlaces simbólicos, un tipo especial de archivo que contiene la referencia a otro archivo o directorio.
- **Third Extended File System (ext3)**: *ext3* es una extensión del sistema de archivos *ext2* con capacidades de *journaling*. El *Journaling* es utilizado para seguir cambios de archivos y tiene como propósito asegurar que las transacciones sean procesadas de forma adecuada; adicionalmente permite arreglar daños en el sistema de archivos originados por una falla en la fuente de alimentación de la plataforma.

- **ReiserFS:** Este sistema de archivos al igual que *ext3* utiliza *journaling*. Fué creado con el fin de aumentar el desempeño frente al sistema *ext2*, es un sistema eficiente en espacio, y mejora el manejo de grandes directorios.
- **Journalling File Flash System 2 (JFFS2):** Sistema creado para trabajar con dispositivos Flash, los cuales son utilizados ampliamente en aplicaciones embebidas.
- **Compressed ROM file system (cramfs):** Sistema de solo lectura, es utilizado cuando se dispone de una pequeña memoria flash NOR. El máximo tamaño de cramfs es de 256MB. Los archivos en este sistema de archivos se encuentran comprimidos.
- **Network File System:** Permite montar particiones de disco o directorios de sistemas remotos como un sistema de archivos local, esto permite compartir recursos como unidades de CDs, DVDs u otro medio de almacenamiento masivo. Por otro lado, reduce el tiempo de desarrollo ya que no es necesario transferir archivos entre el sitio donde se encuentran las herramientas de desarrollo y la plataforma.
- **Pseudo File System** Este sistema de archivos es utilizado por Linux para representar el estado actual del kernel. Se encuentra montado en el directorio */proc*, y dentro de él podemos encontrar información detallada del hardware del sistema. Adicionalmente algunos archivos pueden ser manipulados para informar al kernel cambios en la configuración. Este sistema de archivos es virtual y es constantemente actualizado por el kernel. Los archivos */proc/cpuinfo*, */proc/interrupt*, */proc/devices*, */proc/mounts* proporcionan información sobre los dispositivos hardware de la plataforma

Estructura del Sistema de Archivos

Todas las distribuciones de Linux se basan en el estándar *Filesystem Hierarchy Standard* utilizado en los sistemas operativos UNIX. Este estándar permite que los programas y los usuarios conozcan de antemano la localización de los archivos instalados. Los siguientes directorios o links simbólicos son de uso obligatorio:

1. **bin** Ejecutables esenciales.
2. **boot** Archivos estáticos del boot loader.
3. **dev** Archivos de dispositivos.
4. **etc** Configuración específica del host.
5. **lib** Librerías esenciales y módulos de kernel.
6. **media** Punto de montaje para dispositivos removibles.
7. **mnt** Punto de montaje temporal.
8. **opt**
9. **sbin** Ejecutables esenciales del sistema.
10. **srv** Datos de servicios suministrados por el sistema.
11. **tmp** Archivos temporales.
12. **usr** Segunda jerarquía.
13. **var** Datos variables.

A.10.2. Primer Programa en Espacio de Usuario *init*

Como vimos anteriormente, la primera aplicación en espacio de usuario que ejecuta el kernel es */sbin/init*, todos los procesos que no sean del kernel son generados de forma directa o indirecta por él y es responsable de la inicialización de los scripts y terminales. Su papel más importante es generar procesos adicionales bajo la dirección de un archivo de configuración especial */etc/inittab*

Modos de operación

Existen dos modos de operación en Linux: Modo usuario simple y Multi-usuario, en el primero solo se activa una línea de comandos y el único usuario que puede utilizarla es el super-usuario *root*; es utilizado para sistemas en mantenimiento y normalmente se le asigna el nivel de ejecución 1. En este nivel de ejecución, no existen procesos demonios¹⁸ en ejecución, y la interfaz de red no está configurada[8].

El modo multi-usuario es el modo normal de ejecución del sistema Linux; cuando se inicia en este modo, se ejecutan los siguientes procesos:

- Se revisa el estado del sistema de archivos con *fsck*.
- Se monta el sistema de archivos.
- *init* analiza el archivo */etc/inittab* y
 - Determina el nivel de ejecución
 - Ejecuta los scripts asociados con este nivel de ejecución.
- Inicializa los demonios.
- Permite el acceso a usuarios.

Niveles de ejecución

Un nivel de ejecución puede entenderse como un estado del sistema. Hoy día, la mayoría de las distribuciones utilizan los siguientes niveles de ejecución[8]:

1. 0. Cierre o detención del sistema (*halt*).
2. 1. Modo usuario simple para configuración del sistema y mantenimiento.
3. 2. Modo multi-usuario sin red remota.
4. 3. Modo multi-usuario con red. Este es el modo de operación normal de un usuario de un sistema sin capacidades gráficas.
5. 4. No utilizado - Definido por el usuario.
6. 5. Modo multi-usuario con interfaz gráfica.
7. 6. Re-inicialización del sistema (*reboot*).

El nivel de ejecución puede ser cambiado por el super-usuario (*root*) en cualquier momento utilizando el comando *init n*, donde *n* es el nivel de ejecución deseado.

¹⁸Proceso que se ejecuta de forma discreta sin intervención del usuario y es activado por la ocurrencia de una condición específica

El Archivo */etc/inittab*

Como se mencionó anteriormente el programa *init* está encargado de montar el sistema de archivos y de analizar el archivo */etc/inittab*. Este archivo contiene:

- Una entrada para el nivel de ejecución por defecto. Nivel de ejecución en que inicia el sistema a menos que especifique otra cosa en el boot loader.
- Entradas que deben ser ejecutadas en todos o en un específico nivel de ejecución, su sintaxis es: *id:runlevels:action:process [arguments]*
 - *id* Cualquier cosa.
 - *runlevels* Puede ser un número o lista de números de 0 a 6.
 - *action* Acción a tomar:
 - *respawn* El proceso debe ser re-iniciado una vez finalice.
 - *wait start* Ejecuta el proceso cuando se ingresa al nivel de ejecución y espera por su terminación.
 - *bootwait* El proceso debe ser ejecutado durante la inicialización del sistema.
 - *initdefault* Especifica el nivel de ejecución al que se ingresa después de la inicialización del sistema.
 - *sysinit* El proceso debe ejecutarse durante la inicialización del sistema. Debe ejecutarse antes de cualquier entrada *boot* o *bootinit*
 - *process* Programa o script a ser ejecutado.

En el siguiente listado se muestra un archivo *inittab* típico:

```
# The default runlevel.
id:5:initdefault:

# Boot-time system configuration/initialization script.
# This is run first except when booting in emergency (-b) mode.
si::sysinit:/etc/init.d/rcS

# What to do in single-user mode.
~~:S:wait:/sbin/sulogin

# /etc/init.d executes the S and K scripts upon change
# of runlevel.

10:0:wait:/etc/init.d/rc 0
11:1:wait:/etc/init.d/rc 1
12:2:wait:/etc/init.d/rc 2
13:3:wait:/etc/init.d/rc 3
14:4:wait:/etc/init.d/rc 4
15:5:wait:/etc/init.d/rc 5
16:6:wait:/etc/init.d/rc 6
# Normally not reached, but fallthrough in case of emergency.
z6:6:respawn:/sbin/sulogin
S:2345:respawn:/sbin/getty 115200 ttyS0
# /sbin/getty invocations for the runlevels.

1:2345:respawn:/sbin/getty 38400 tty1
```

En este archivo se define el nivel de ejecución 5 como el nivel por defecto. El primer script en ejecutarse es `/etc/init.d/rcS` (ya que su acción es del tipo *sysinit*). Luego se ingresa al nivel de ejecución 5 y se ejecuta el script `/etc/init.d/rc` pasándole el argumento “5” y espera hasta que el script se complete. `/etc/init.d/rc` ejecuta los scripts localizados en directorios individuales para cada nivel: `/etc/rcX.d` (X un entero de 0 a 6); el nombre de los archivos localizados en estos directorios deben comenzar con el carácter “S” (para iniciar procesos) o “K” (para “matar” procesos), y dos caracteres numéricos: `S[0-9][0-9]`, `S[0-9][0-9]`. Un script típico de inicialización del demonio del servidor web *cherokee* se muestra en el siguiente listado (`/etc/rc5.d/S91cherokee`):

```
#!/bin/sh
DAEMON=/usr/sbin/cherokee
CONFIG=/etc/cherokee/cherokee.conf
PIDFILE=/var/run/cherokee.pid
NAME="cherokee"
DESC="Cherokee http server"

test -r /etc/default/cherokee && . /etc/default/cherokee
test -x "$DAEMON" || exit 0
test ! -r "$CONFIG" && exit 0

case "$1" in
    start)
        echo "Starting $DESC: "
        start-stop-daemon --oknodo -S -x $DAEMON -- -b -C $CONFIG
        ;;

    stop)
        echo "Stopping $DESC:"
        start-stop-daemon -K -p $PIDFILE
        ;;

    restart)
        $0 stop >/dev/null 2>&1
        $0 start
        ;;

    *)
        echo "Usage: $0 {start|stop|restart}"
        exit 0
        ;;
esac
```

Como podemos ver existen tres parámetros que podemos pasar al script: *start*, *stop* y *restart*, cuyas acciones son iniciar, detener y reiniciar el demonio respectivamente. El directorio `/etc/init.d` contiene todos los scripts utilizados en los diferentes niveles de ejecución, el nombre de ellos en este directorio no incluyen los caracteres `S[0-9][0-9]` o `K[0-9][0-9]`.

La línea `S:2345:respawn:/sbin/getty 115200 ttyS0` del archivo *inittab* inicia una consola por el puerto serial `/dev/ttyS0` con una velocidad de 115200 baudios, cuando el nivel de ejecución sea 2, 3, 4 o 5. Finalmente se crea una terminal virtual para los niveles de ejecución multi-usuario.

Cuando el super-usuario cambia el nivel de ejecución con el comando *init*, los procesos únicos al nivel anterior son terminados y los procesos únicos del nuevo nivel son iniciados.

A.10.3. Distribuciones Linux

Aunque es posible construir el sistema de archivos desde cero, no es nada práctico ya que es una tarea tediosa que requiere cierto nivel de experiencia. En la actualidad, existen varias distribuciones que realizan estas tareas por nosotros, dentro de las más utilizadas se encuentran:

Busybox

Diseñado para optimizar el tamaño y rendimiento de aplicaciones embebidas, BusyBox ¹⁹ combina en un solo ejecutable más de 70 utilidades estándares UNIX, en sus versiones ligeras. BusyBox es considerada la navaja suiza de los sistemas embebidos, dado que permite sustituir la gran mayoría de utilidades que se suelen localizar en los paquetes GNU fileutils, shellutils, findutils, textutils, modutils, grep, gzip, tar, etc.

Busybox hace parte de la mayoría de distribuciones de Linux para sistemas embebidos y en la actualidad proporciona las siguientes funciones:

addgroup, adduser, adjtimex, ar, arping, ash, awk, basename, bunzip2, busybox, bzip, cal, cat, chgrp, chmod, chown, chroot, chvt, clear, cmp, cp, cpio, crond, crontab, cut, date, dc, dd, deallocvt, delgroup, deluser, df, dirname, dmesg, dos2unix, dpkg, dpkg-deb, du, dumpkmap, dumpleases, echo, egrep, env, expr, false, fbset, fdflush, fdisk, fgrep, find, fold, free, freeramdisk, fsck.minix, ftpget, ftpput, getopt, getty, grep, gunzip, gzip, halt, head, hexdump, hostid, hostname, httpd, hwclock, id, ifconfig, ifdown, ifup, init, ip, ipaddr, ipcalc, iplink, iproute, iptunnel, kill, killall, klogd, last, length, linuxrc, ln, loadfont, loadkmap, logger, login, logname, logread, losetup, ls, makedevs, md5sum, msg, mkdir, mkfifo, mkfs.minix, mknod, mkswap, mktmp, more, mount, mt, mv, nameif, nc, netstat, nslookup, od, openvt, passwd, patch, pidof, ping, ping6, pivot_root, poweroff, printf, ps, pwd, rdate, readlink, realpath, reboot, renice, reset, rm, rmdir, route, rpm, rpm2cpio, run-parts, sed, setkeycodes, sh, sha1sum, sleep, sort, start-stop-daemon, strings, stty, su, sulogin, swapoff, swapon, sync, syslogd, tail, tar, tee, telnet, telnetd, test, tftp, time, top, touch, tr, traceroute, true, tty, udhcpc, udhcpd, umount, uname, uncompress, uniq, unix2dos, unzip, uptime, usleep, uudecode, uuencode, vi, vlock, watch, watchdog, wc, wget, which, who, whoami, xargs, yes, zcat.

Buildroot

Buildroot²⁰ Es un grupo de *Makefiles* y *patches* que facilita la generación de la cadena de herramientas y el sistema de archivos para un sistema embebido que usa Linux. Posee una interfaz que permite realizar de forma fácil la configuración; utiliza busybox para generar las utilidades básicas de Linux y permite adaptar software adicional de forma fácil ²¹.

Openembedded

Al igual que Buildroot, el proyecto *openembedded* proporciona un entorno que permite generar la cadena de herramientas y el sistema de archivos para un sistema embebido, utiliza busybox y permite la creación de archivos que permiten compilar aplicaciones que no se incluyen en la distribución original. Adicionalmente *openembedded* crea archivos instaladores con un formato derivado del proyecto *handhelds* ²² *ipk*, lo que permite la instalación de paquetes de forma similar a la distribución debian.

¹⁹<http://www.busybox.net/>

²⁰<http://buildroot.uclibc.org/>

²¹http://buildroot.uclibc.org/buildroot.html#add_software

²²<http://handhelds.org>

La información necesaria para generar una distribución utilizando las herramientas de *Openembedded* se encuentra en http://linuxencaja.com/wiki/ECBOT#Generación_de_la_imagen_para_la_memoria_Flash.

A.11. Módulos del kernel

Los módulos son pequeñas piezas de código que pueden ser cargadas y descargadas en el kernel en el momento que sea necesario. Ellos extienden la funcionalidad del kernel, sin la necesidad de reiniciar el sistema y recompilar el kernel. Por ejemplo, un tipo de módulo es el controlador de dispositivo, el cual permite al kernel acceder a un dispositivo hardware conectado al sistema. Este tipo de módulos serán estudiados en esta sección.

Existen tres tipos de dispositivos en Linux [9]:

- **Caracter:** Puede accederse de forma similar a un archivo; este tipo de dispositivos permite por lo menos las operaciones *open*, *close*, *read*. Ejemplos de este tipo de dispositivo son el puerto serie (*/dev/ttyS0*) y la consola (*/dev/console*)
- **Bloque:** Este tipo de dispositivo puede hospedar un sistema de archivos; normalmente realiza operaciones de bloques de datos únicamente; un ejemplo de este tipo de dispositivo es el disco duro (*/dev/hda*).
- **Red:** Toda transacción de red se realiza a través de una interfaz, esto es, un dispositivo hardware (*/dev/eth0*) o software (*loopback*) capaz de intercambiar datos con otros hosts.

A.11.1. Ejemplo de un driver tipo caracter

Recuerde que una aplicación en el área de usuario, no puede acceder directamente al área del kernel; los dispositivos se acceden a través de archivos de dispositivos, localizados en */dev* (ver figura A.23), A continuación se muestra la salida del comando *ls -l /dev/*

brw-rw—	1	root	disk	3,	0	Nov 27	hda
brw-rw—	1	root	disk	3,	1	Nov 27	hda1
brw-rw—	1	root	disk	3,	2	Nov 27	hda2
crw-rw—	1	root	uucp	4,	64	Nov 27	ttyS0
crw-rw—	1	root	uucp	4,	65	Nov 27	ttyS1

Los archivos tipo caracter están identificados por una “c” en la primera columna, mientras que los dispositivos tipo bloque por una “b”. Podemos observar que existen dos números (5ta y 6ta columna) que identifican al driver, el número de la 5ta columna recibe el nombre de *major number* y el de la sexta *minor number*; estos números son utilizados por el sistema operativo para determinar el dispositivo y el driver que deben ser utilizados ante una solicitud a nivel de usuario.

El *major number* identifica la clase o grupo del dispositivo, mientras que el *minor number* se utiliza para identificar sub-dispositivos (Ver Figura A.23).

El kernel de Linux permite que los drivers compartan el número mayor, como el caso del disco duro, *hda* posee dos particiones *hda1* y *hda2*, las que son manejadas por el mismo driver, pero se asigna un número menor único a cada una; lo mismo sucede con el puerto serie.

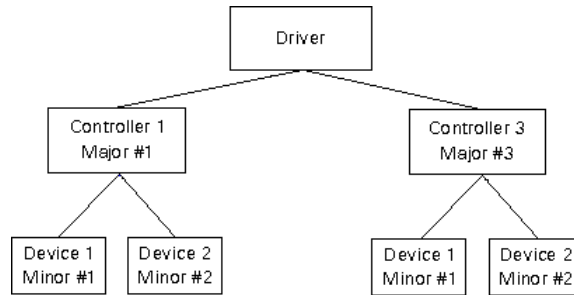


Figura A.23: Números *major* y *minor* de un driver

Implementación del driver de un LED

A continuación se realizará la descripción de un driver tipo caracter para un dispositivo muy sencillo, un LED. Este ejemplo realiza las siguientes operaciones:

- *init*: Se ejecuta cuando se carga el módulo, el LED se encenderá.
- *open*: Se ejecuta cuando se abre el archivo que realiza la interfaz con el driver en una operación de lectura o escritura. El LED parpadeará 5 veces y quedará encendido.
- *release*: Se ejecuta cuando se cierra el archivo que hace la interfaz con el módulo después de una operación de lectura o escritura. El LED se apagará.
- *exit*: Se ejecuta cuando se descarga el módulo, el LED se apagará.

Existen dos funciones que deben estar presentes en todo tipo de módulo, estas son: *module_init* y *module_exit* las cuales se ejecutan cuando se carga y descarga el módulo respectivamente.

```

76 struct file_operations fops = {
77     .open    = device_open ,
78     .release = device_release ,
79 };
80
81
82 static int __init blink_init(void)
83 {
84     printk(KERN_INFO "BLINK module is Up.\n");
85
86     Major = register_chrdev(0, DEVICE_NAME, &fops);
87
88
89     if (Major < 0) {
90         printk(KERN_ALERT "Registering char device failed with %d\n", Major);
91         return Major;
92     }
93
94     printk(KERN_ALERT "I was assigned major number %d. To talk to\n", Major);
95     printk(KERN_ALERT "the driver, create a dev file with\n");
96     printk(KERN_ALERT "'mknod /dev/%s c %d 0'.\n", DEVICE_NAME, Major);
97
98     at91_set_gpio_output(LED, 1);    /* Let the LED's pin as output*/
99
  
```

```

100     at91_set_gpio_value(LED, 1);    /* Turn LED ON */
101
102     return 0;
103 }
104
105 static void __exit blink_exit(void)
106 {
107     int ret;
108
109     at91_set_gpio_value(LED, 0);
110
111     ret = unregister_chrdev(Major, DEVICE_NAME);
112
113     if ( ret < 0 )
114         printk( KERN_ALERT "Error in unregister_chrdev: %d\n", ret );
115
116     printk( KERN_INFO "BLINK driver is down...\n" );
117 }
118
119 module_init(blink_init);
120 module_exit(blink_exit);
121
122 MODULE_LICENSE("GPL");
123 MODULE_AUTHOR("Carlos Camargo <cicamargoba@gmail.com>");
124 MODULE_DESCRIPTION("BLINKER LED driver");
125 MODULE_VERSION("1:0.1");

```

Las funciones *module_init* *module_exit* deben ser declaradas como *static* ya que no serán visibles fuera del archivo. Como puede observarse en la línea 119 se hace la definición de las funciones que deben ejecutarse al cargar y descargar el módulo en nuestro caso *blink_init* y *blink_exit* respectivamente. La información sobre el módulo aparece en las líneas 122-125.

En la línea 76 se define la estructura de operaciones de archivo del módulo; cada campo de la estructura corresponde a la dirección de una función definida por el driver para manejar una solicitud determinada. En nuestro caso existen las funciones *open* y *release*, las cuales serán definidas más adelante.

Como se puede ver en la figura A.23 es necesario que el kernel sepa que driver está encargado del dispositivo, esto es, el *major number* del driver que lo maneja. Por esto, lo primero que se debe hacer es obtener este número. En la función *blink_init* (línea 86) podemos observar la forma en que se realiza el registro de nuestro dispositivo. La función *register_chrdev* retorna el número mayor asignado de forma dinámica, esto es recomendable ya que si se fijara un número de forma arbitraria, podría causar conflictos con otros dispositivos. De esta forma, al cargar el módulo con el comando: *insmod blinker.ko*, el LED se encenderá y aparecerá el siguiente mensaje en la consola:

```

BLINK module is Up.
I was assigned major number 253. To talk to
the driver, create a dev file with
'mknod /dev/blink c 253 0'.

```

Con lo anterior, nuestro dispositivo es registrado y se le asigna el número 253 como *major number*. En el archivo */proc/devices* aparecen los dispositivos que están siendo utilizados por el kernel, este archivo debe contener una entrada para blink de la forma: *253 blink*.

En la función *blink_exit* se realiza la liberación del dispositivo (la función *unregister_chrdev*, línea 111), (la que se ejecuta cuando se lanza el comando: *rmmod blinker.ko*), se apaga el LED y se imprime en la consola el mensaje:

```

BLINK driver is down...

```

Como se mencionó anteriormente es posible manejar un archivo tipo caracter como si fuera un archivo de texto, por lo tanto, es posible adicionar funciones a las acciones de abrir, cerrar, escribir en o leer del dispositivo.

```

40 static int device_open(struct inode *inode, struct file *file)
41 {
42
43     unsigned int i;
44     printk( KERN_INFO "Open BLINKER\n" );
45     if (is_device_open)
46         return -EBUSY;
47
48     is_device_open = 1;
49
50
51     for( i=0; i<5; i++ ){
52         at91_set_gpio_value(LED, 0);
53         mdelay(0x0080);
54         at91_set_gpio_value(LED, 1);
55         mdelay(0x0080);
56     }
57
58     try_module_get(THIS_MODULE);
59
60     return SUCCESS;
61 }
62
63 static int device_release(struct inode *inode, struct file *file)
64 {
65     is_device_open = 0;
66
67     at91_set_gpio_value(LED, 0);
68
69     module_put(THIS_MODULE);
70
71     printk( KERN_INFO "Close BLINKER\n" );
72
73     return 0;
74 }

```

La función *device_open* (línea 40) se ejecuta cada vez que el archivo de dispositivo */dev/blink* es abierto, esto sucede en operaciones de lectura o escritura. Es decir si se utilizan los siguientes comandos:

```

more /dev/blink
cat file > /dev/blink
cp file /dev/blink

```

La función *device_release* (línea 63) se ejecuta cada vez que se cierra el archivo de dispositivo */dev/blink*, después de una operación de lectura o escritura.

Con cada acceso de lectura o escritura el LED se parpadeará 5 veces y en la consola se despliega el mensaje:

```

Open BLINKER
Close BLINKER

```

A.12. Interfaz con Periféricos dedicados

Es común, que algunas aplicaciones requieran ciertos periféricos especiales que les permitan cumplir las restricciones temporales, es decir, se deben crear tareas Hardware que ayuden a las tareas software a cumplir con las restricciones de diseño. Para esto es necesario implementar algunas funciones en periféricos externos al procesador. Si la tarea no se encuentra implementada en un dispositivo comercial es necesario implementarlas en un dispositivo lógico programable (PLD) o en un circuito integrado de aplicación específica (ASIC). En esta sección realizaremos una explicación detallada del proceso de comunicación entre el procesador y un periférico implementado en una FPGA.

A.12.1. Comunicación Procesador - Periférico

La figura A.24 muestra una arquitectura básica para la comunicación de tareas Hardware - Software; en ella podemos observar que el procesador maneja tres buses:

- Bus de Datos: Bus bidireccional por donde se realiza el intercambio de información.
- Bus de Direcciones: Bus controlado por el procesador y es utilizado para direccionar un determinado periférico o una determinada funcionalidad del mismo.
- Bus de control: Señales necesarias para indicarle a los periféricos el tipo de comunicación (lectura o escritura).

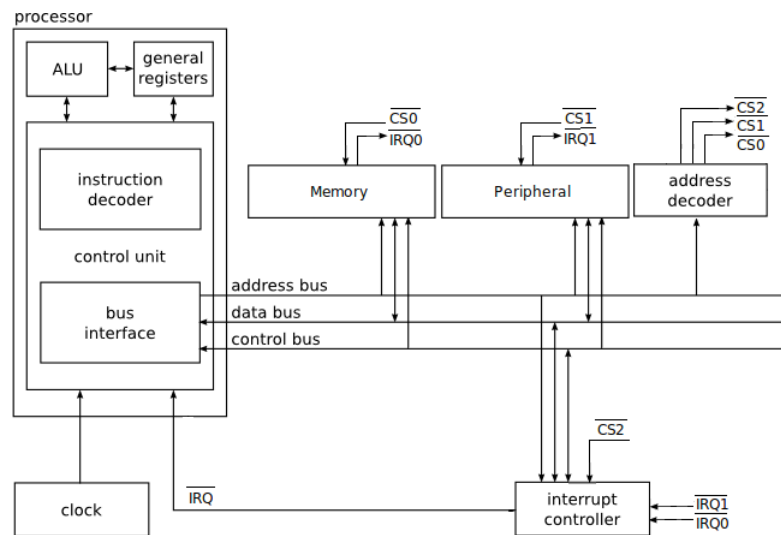


Figura A.24: Arquitectura básica hardware/software

Todos los periféricos que requieren intercambio de información con el procesador comparten el mismo bus de datos, por lo que es necesario que mientras el procesador no se comuniquen con ellos permanezcan en estado de alta impedancia; esto es necesario para evitar corto - circuitos originados por diferentes niveles lógicos en el bus. Por lo tanto, las comunicaciones siempre son iniciadas por el procesador y se selecciona uno y solo un periférico a la vez. El decodificador de direcciones es el encargado de habilitar

un determinado periférico ante una solicitud del procesador (mediante una dirección de memoria), esto lo hace activando la señal \overline{CSX} , cuando esta señal se encuentra en estado lógico alto el periférico coloca su bus de datos en alta impedancia; si se encuentra en estado lógico bajo, el periférico escribe o lee el bus de datos, dependiendo de la activación de las señales del bus de control RD y WR.

El decodificador de direcciones, como su nombre lo indica utiliza como entradas el bus de direcciones y activa solo una señal de selección de Chip (\overline{CSX}), basándose en un rango de direcciones asignado a cada periférico, este rango de direcciones no debe traslaparse para asegurar que solo un chip es seleccionado. Este rango de direcciones que se asigna a cada dispositivo que puede ser accedido por la unidad de procesamiento recibe el nombre de *mapa de memoria* y puede variar para cada plataforma.

Cuando la unidad de procesamiento necesite comunicarse con un determinado periférico, colocará en el bus de direcciones un valor que se encuentre en el rango de direcciones asignado para ese periférico; esto hace que el decodificador de direcciones active la señal de selección adecuada para informarle al periférico que el procesador va a iniciar una transferencia de información. La Figura A.25 muestra el diagrama de tiempos para un ciclo de lectura y escritura del procesador.

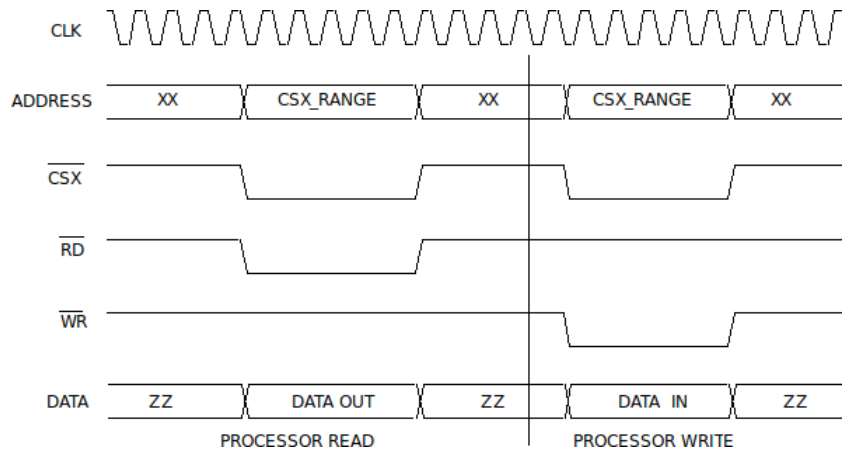


Figura A.25: Ciclo de lectura y escritura para la arquitectura de la Figura A.24

De lo anterior podemos concluir que un periférico es visto por el procesador como una posición de memoria y las transacciones las inicia únicamente el procesador.

Implementación de Periféricos en una FPGA

Es importante tener en cuenta los siguientes puntos cuando se implemente una tarea Hardware en una FPGA:

- La frecuencia del reloj de la FPGA es mucho mayor que la de las señales del bus de control; por lo que es necesario asegurarse que cada vez que el procesador realiza una solicitud de lectura o escritura, la señal de activación cambia del estado lógico alto al bajo; si solo se tiene en cuenta el estado bajo de la señal \overline{CSX} el periférico puede ejecutar la tarea varias veces en el mismo ciclo de activación, lo que puede llevar a resultados incorrectos.
- La fase de los relojes del procesador y la FPGA no es la misma, por lo que es necesario sincronizar las señales del procesador con el reloj de la FPGA; si esto no se hace las señales fuera de fase pueden

originar un estado de meta-estabilidad en los Flip-Flops internos de la FPGA y por lo tanto el malfuncionamiento del sistema.

- El bus de datos es bidireccional, por lo que es necesario que la FPGA lo coloque en alta impedancia cuando no se esté habilitando un periférico.
- La FPGA no permite implementar buffers tri-estado internamente por lo que es necesario separar los buses de entrada y salida a cada periférico. El bus de entrada es común a todos los periféricos mientras que es necesario utilizar un esquema de multiplexación entre los buses de salida.

La figura A.26 muestra el diagrama de bloques de la interfaz necesaria para poder comunicar un grupo de periféricos o tareas Hardware con el bus de datos, dirección y control de un procesador. El bloque *SYNC* se encarga de sincronizar las señales provenientes de la FPGA con el reloj interno de la misma.

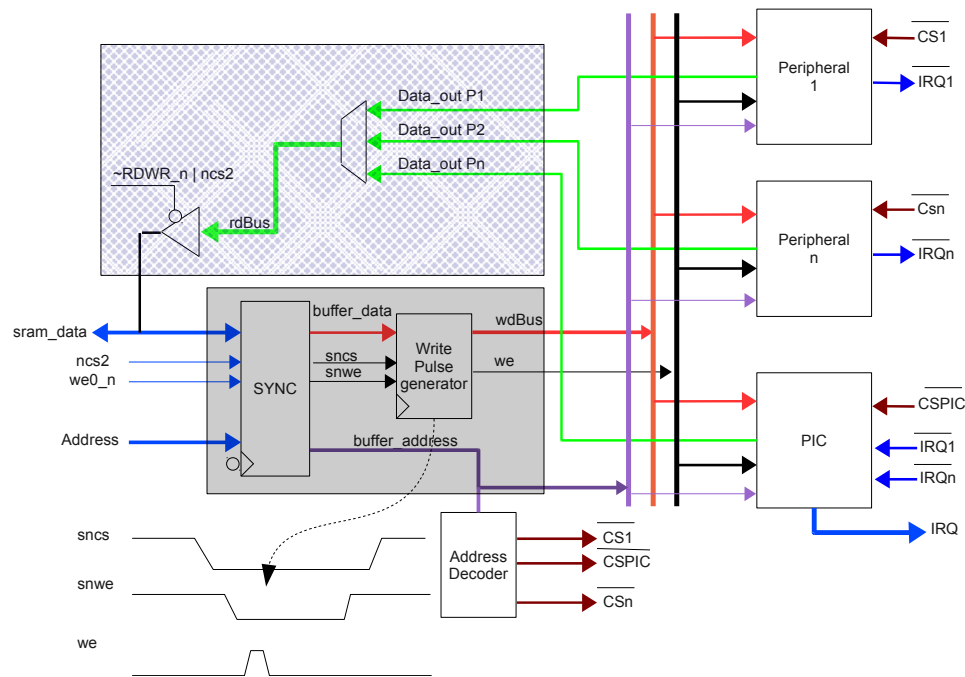


Figura A.26: Diagrama de Bloques para la comunicación de tareas HW-SW A.24

El módulo *Write Pulse generator* genera un pulso cuando las señales *sncs* y *snwe* son activadas como se indica en la parte inferior izquierda de la figura A.24. El decodificador de direcciones selecciona un determinado periférico dependiendo del rango especificado para cada uno. Como mencionamos anteriormente, las FPGAs no permiten implementar buffers tri-estado internamente, por lo que cada periférico debe tener un bus de entrada y uno de salida de datos, los buses de entrada de datos son comunes, mientras que los de salida deben ser manejados por un dispositivo de salida, el cual puede ser un multiplexor controlado por el decodificador de direcciones o una compuerta OR, para este último caso es necesario que los periféricos coloquen el bus de datos en "0" cuando no estén seleccionados. Finalmente es necesario colocar un buffer tri-estado en los pines de la FPGA, este buffer está controlado por las señales *RDWRn* y *ncs2* y solo se activa cuando el procesador inicia una operación de lectura.

Ejemplo de periférico

La figura A.27 muestra un ejemplo sencillo de un periférico que implementa una comunicación en ambas vías con el procesador. En cada uno de sus componentes se indica su implementación utilizando el lenguaje de descripción de hardware *verilog*. El periférico instancia una memoria RAM de Xilinx utilizando sus celdas propietarias, de tal forma que visto por el procesador, este periférico no es mas que un espacio de lectura/escritura.

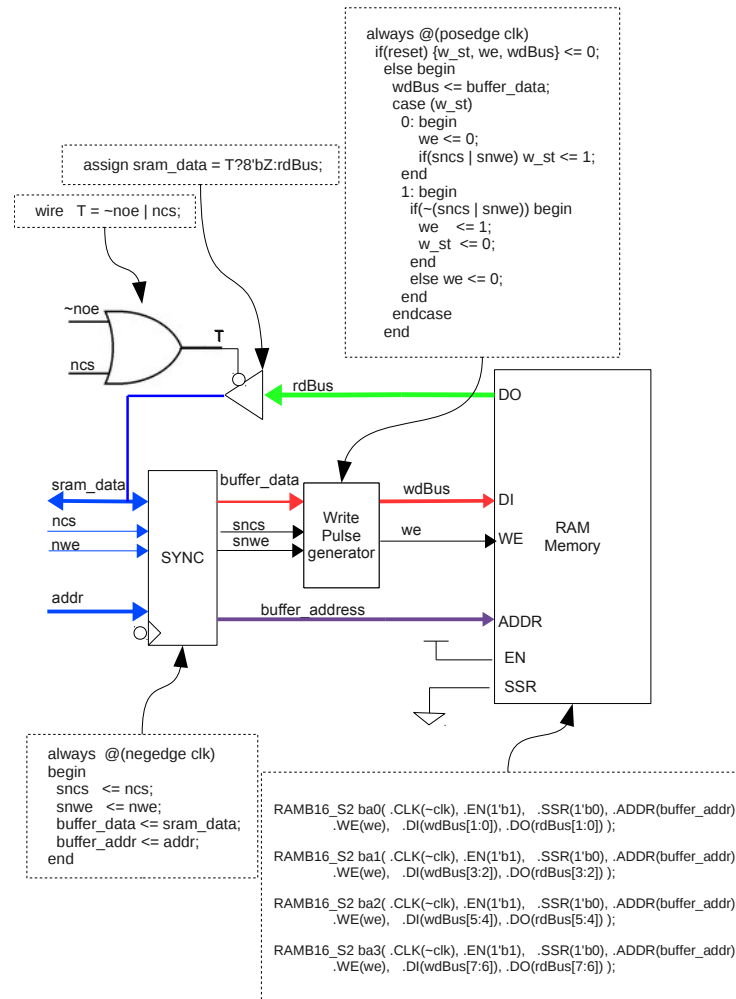


Figura A.27: Diagrama de bloques de un periférico sencillo

Programa en Espacio de Usuario para la comunicación

El kernel de Linux proporciona una interfaz que permite a una aplicación *mapear* un archivo en memoria virtual; creando una correspondencia uno a uno entre las direcciones de memoria y el contenido del archivo. Linux implementa la llamada de sistema *mmap()* para *mapear* objetos en memoria virtual.[10]

```
#include <sys/mman.h>

void * mmap (void *addr,
             size_t len,
             int prot,
             int flags,
             int fd,
             off_t offset);
```

Un llamado a *mmap()* le pide al kernel hacer un mapeo en la memoria de *len* bytes del objeto representado por el descriptor de archivo *fd*, comenzando a *offset* bytes dentro del archivo. Si se especifica *addr*, se utiliza este valor como la dirección inicial en la memoria. Los permisos de acceso son determinados por *prot*; PROT_READ habilita la lectura, PROT_WRITE habilita escritura y PROT_EXEC habilita la ejecución. El argumento *flags* describe el tipo de mapeo, y algunos elementos de su comportamiento y puede tomar los valores:

- MAP_FIXED: hace que *addr* sea un requerimiento, si el kernel es incapaz de hacer el mapeo en esta dirección el llamado falla, si los parámetros de dirección y longitud traslapan un mapeo existente se descartan las áreas que se traslapan y se reemplazan por el nuevo mapeo.
- MAP_PRIVATE: Establece que el mapeo no es compartido. Los cambios realizados a la memoria por este proceso no se reflejan en el archivo actual o en el mapeo de los otros procesos.
- MAP_SHARED: Comparte el mapeo con otros procesos que usan el mismo archivo. Escribir en el mapeo equivale a escribir en el archivo.

En la figura A.28 se muestra el concepto del mapeo de memoria; en ella, se mapean *0x20000000* bytes de dirección física comenzando desde la posición *0x40000000*; la variable *base* retorna la posición dentro del espacio de memoria virtual donde fue asignada la memoria física.

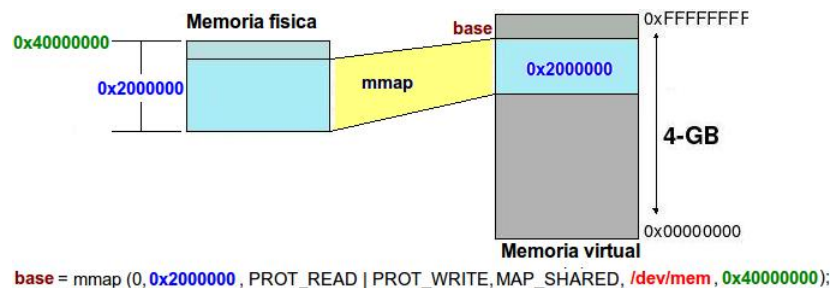


Figura A.28: Mapeo de memoria.

A continuación se lista el código en C que implementa el mapeo de la dirección de memoria asignado al periférico de la figura A.27.

```
#define MAP_SIZE 0x20000001 // ECBOT USE A11 to A25
#define MAP_MASK (MAP_SIZE - 1)

int fd;
unsigned long i, j;
void *base;
volatile unsigned short *virt_addr;
```



```

io_map(0xFFFFF7C);           // Configure CS3 as 16 bit Memory and 0 Wait States
off_t address = 0x40000000;   // CS3 Base Address

if ((fd = open ("/dev/mem", ORDWR | O_SYNC)) == -1){
    printf ("Cannot open /dev/mem.\n");
    return -1;
}
printf ("/dev/mem opened.\n");

base = mmap (0, MAP_SIZE, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, address);
if (base == (void *) -1){
    printf ("Cannot mmap.\n");
    return -1;
}
printf ("Memory mapped at address %p.\n", base);

virt_addr = base;

```

En este ejemplo utilizamos el llamado *mmap* para hacer un mapeo de la dirección de memoria correspondiente al CS3 (0x40000000). El descriptor del archivo utilizado es el dispositivo */dev/mem* el cual permite operaciones de lectura y escritura a la memoria virtual. Adicionalmente permitimos operaciones de lectura/escritura (PROT_READ — PROT_WRITE) y permitimos el acceso a otros procesos. Finalmente, podemos usar la variable *virt_addr* para escribir en cualquier posición de memoria desde 0x40000000 hasta 0x40000000 + MAP_SIZE. El siguiente listado muestra un ejemplo de la forma de hacer estas operaciones.

```

printf("Writing Memory..\n");
for (i = 0 ; i < 32; i++){
    virt_addr[i<<10] = i*3;           // ECBOT use A11 to A25
}

printf("Reading Memory..\n");
for (i = 0 ; i < 32; i++)
{
    j = virt_addr[i<<10];
    printf(" %X = %X\n", i, j );
}
if (munmap (base, MAP_SIZE) == -1)
{
    printf ("Cannot munmap.\n");
    return -1;
}

```

A.12.2. Comunicación Periférico - Procesador

Cuando un periférico requiere atención por parte de la CPU debido a que terminó de realizar un proceso o porque requiere nuevas instrucciones para seguir operando, o un evento originado por un usuario necesita ser atendido; se debe iniciar un proceso para que la unidad de procesamiento se comunique con él. Como se mencionó anteriormente, la unidad de procesamiento está encargada de forma exclusiva de iniciar las operaciones de lectura/escritura con los periféricos; por esta razón, el periférico utiliza una señal (IRQ) para informarle al procesador que requiere ser atendido. Algunas arquitecturas poseen un mecanismo que permite el acceso a la memoria por parte de los periféricos sin utilizar el procesador, esto permite que periféricos de diferentes velocidades se comuniquen sin someter al procesador a una carga excesiva. En esta sección hablaremos de la forma de definir las interrupciones en un driver de Linux.

Manejo de Interrupciones

A continuación se describirá la forma de manejar las interrupciones utilizando un driver de Linux. Analicemos la función *qem_init*

```
static int __init qem_init(void)
{
    int res;
    printk(KERN_INFO "FPGA module is Up.\n");

    Major = register_chrdev(0, DEVICE_NAME, &fops);

    if (Major < 0) {
        return Major;
    }

    /* Set up the FPGA irq line */
    at91_set_gpio_input(FPGA_IRQ_PIN, 0);
    at91_set_deglitch(FPGA_IRQ_PIN, 1);

    res = request_irq(FPGA_IRQ_PIN, irq_handler, IRQF_DISABLED, "FPGA - IRQ", NULL);

    set_irq_type(FPGA_IRQ, IRQT_RISING);

    ioremap(FPGA_BASE, 0x4000);

    return 0;
}
```

Esta rutina es similar a la presentada anteriormente, solo se agrega un par de comandos para definir un pin del procesador como entrada, para poder utilizarlo como señal IRQ, en la línea:

```
res = request_irq(FPGA_IRQ_PIN, irq_handler, IRQF_DISABLED, "FPGA - IRQ", NULL);
set_irq_type(FPGA_IRQ, IRQT_RISING);
```

Se hace un llamado a la función *request_irq* que asigna recursos a la interrupción, habilita el manejador y la línea de interrupción. En nuestro caso define el pin *FPGA_IRQ_PIN* como la línea de interrupción, la rutina *irq_handler* será ejecutada cuando se presente una interrupción, el flag *IRQF_DISABLED* deshabilita las interrupciones locales mientras se procesa, el nombre del dispositivo que realiza la interrupción es *FPGA - IRQ*.

La función *ioremap(offset, size)* permite que la memoria de la CPU se pueda acceder con las funciones *readb/ readw/ readl/ writeb/ writew/ write*, utilizando la variable *ioaddress*. Finalmente se define el tipo de interrupción *FPGA_IRQ* como *IRQT_RISING*.

La función que se ejecuta cuando se presenta la interrupción, se define en el siguiente listado:

```
irqreturn_t irq_handler(int irq, void *dev_id, struct pt_regs *regs)
{
    if (irq_enabled)
    {
        interrupt_counter++;
        printk(KERN_INFO "interrupt_counter=%d\n", interrupt_counter);
        wake_up_interruptible(&wq);
    }
    return IRQ_HANDLED;
}
```

Cada vez que se produce una interrupción y si la variable global *irq_enabled* es igual a 1, se aumenta en 1 el valor de *interrupt_counter*, se imprime su valor y el de un registro interno del periférico.

En este driver utilizaremos la función *device_read* para enviar información a un programa en espacio de usuario.

```
static ssize_t device_read(struct file *filp, /* see include/linux/fs.h */
                           char *buffer, /* buffer to fill with data */
                           size_t count, /* length of the buffer */
                           loff_t * offset)
{
    if(irq_enabled){
        wait_event_interruptible(wq, interrupt_counter!=0);
        copy_to_user ( buffer, &interrupt_counter, sizeof(interrupt_counter) );
        interrupt_counter=0;
    }
    else{
        interrupt_counter = -1;
        copy_to_user ( buffer, &interrupt_counter, sizeof(interrupt_counter) );
    }
    return sizeof(interrupt_counter);
}
```

Cuando se realice una operación de lectura desde espacio de usuario, el proceso quedará bloqueado por la función *wait_event_interruptible* hasta que la rutina de atención a la interrupción ejecute la función *wake_up_interruptible*, pero es necesario que se cumpla la condición evaluada por *wait_event_interruptible* para que se ejecute la tarea. Para este ejemplo:

```
wait\_event\_interruptible(wq, interrupt\_counter!=0);
```

Por lo que *irq_handler* debe hacer:

```
interrupt_counter++;
wake_up_interruptible(&wq);
```

Si no se hace esto el proceso nunca se despertará y el proceso de lectura quedará bloqueado. En la línea:

```
copy_to_user ( buffer, &interrupt_counter, sizeof(interrupt_counter) );
copy_to_user ( to, from, long n);
```

Se utiliza la función *copy_to_user* para intercambiar información con el programa que se ejecuta en espacio de usuario. En este caso se copia a *char *buffer*, la variable *interrupt_counter*. Existe una función análoga que recibe información proveniente del espacio de usuario *copy_from_user*. En la Figura A.29 se muestran estas operaciones.

Inicialmente la variable *irq_enabled* tiene un valor de 0, para cambiar su valor se utiliza la función *device_write*, la que permite pasar información al driver desde espacio de usuario. A continuación se muestra esta función:

```
static ssize_t
device_write(struct file *filp, const char *buff, size_t count, loff_t * off)
{
    const char cmd = buff[0];

    if(cmd=='Q')
    {
        /* Reset the FPGA */
    }
}
```

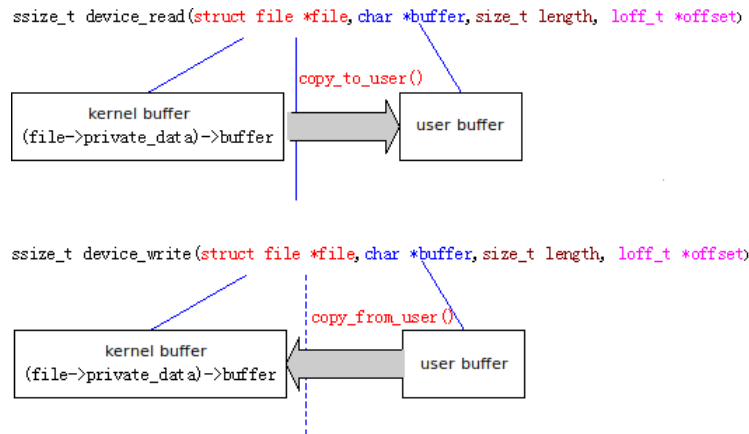


Figura A.29: Intercambio de información entre los espacios de kernel y usuario

```

irq_enabled      = 1;
interrupt_counter = 0;
printk(KERN_INFO "FPGA irq_enabled...\n");
}
else
{
    if (cmd=='S'){
        irq_enabled=0;
        printk(KERN_INFO "FPGA irq disabled.\n");
    }
}
return 1;
}

```

la información que el usuario escribe en el driver se encuentra disponible en la variable *buff*, esta función permite escribir cualquier cantidad de información, el tamaño de la información escrita desde espacio de usuario es almacenado en la variable *count*. En este ejemplo, si se escribe el caracter “Q” *irq_enabled* será igual a **1** y si se escribe “S” *irq_enabled* será igual a **0**.

En la función *qem_exit* se liberan los recursos de la interrupción y la variable *ioaddress*.

```

static void __exit qem_exit(void)
{
    int ret;
    /*Tho order for free_irq, iounmap & unregister is very important */
    free_irq(FPGA_IRQ_PIN, NULL);
    iounmap(ioaddress);
    unregister_chrdev(Major, DEVICE_NAME);

    printk(KERN_INFO "FPGA driver is down...\n");
}

```

Comunicación con el driver desde espacio de usuario

Aunque es posible leer y escribir la información que suministra un driver desde la consola utilizando las aplicaciones *cat* (lectura) y *echo* (escritura); es importante conocer la forma de comunicación desde

aplicaciones propias en espacio de usuario, a continuación se muestra como puede hacerse esto:

```

1 int main(int argc, char **argv) {
2     int         fileNum, bytes;
3     unsigned char buf[40];
4     size_t      nbytes;
5     ssize_t     bytes_read;
6
7     if(argc != 2){
8         fprintf(stderr, "\nUsage: %s enable|disable|read \n", argv[0]);
9     }
10    nbytes = sizeof(int);
11    fileNum = open("/dev/fpga", ORDWR);
12    if (fileNum < 0) {
13        printf(" Unable to open device\n");
14        exit(1);
15    }
16    printf("Device opened successfully \n");
17    if(!strcmp(argv[1], "enable"))
18        write(fileNum, "Q", 1);
19    if(!strcmp(argv[1], "disable"))
20        write(fileNum, "S", 1);
21    if(!strcmp(argv[1], "read")){
22        read(fileNum, buf, nbytes);
23        printf("Interrupts = %d \n", *((int*)(buf)));
24    }
25    if( (strcmp(argv[1], "read") != 0) & (strcmp(argv[1], "disable") != 0) &
26        (strcmp(argv[1], "enable") != 0) )
27        fprintf(stderr, "\nUsage: %s enable|disable|read \n", argv[0]);
28    close(fileNum);
29    return (0);
30 }

```

En la línea XX se abre el archivo de dispositivo */dev/fpga*, con las funciones utilizadas para manejo de archivos (*open*, línea 11). y se utilizan la funciones *read*(línea 22) para leer la información suministrada por el driver o la función *write* (líneas 18,20) para escribir en él; dependiendo del parámetro pasado al ejecutable (*argv[1]*, líneas 17, 19, 21).

Encuestas realizadas a los estudiantes indican que: con este grupo de herramientas se pueden realizar todas las actividades propuestas en este programa académico; que son más difíciles de manejar que las herramientas comerciales; pero que se prefiere el uso de alternativas abiertas al de software ilegal; en el pasado no les importaba utilizar métodos ilegales para licenciar aplicaciones software, pero que ahora van a buscar primero las alternativas libres antes de recurrir al uso de software ilegal; el uso de Linux hace que se sientan identificados con él, llegando al punto de realizar un cambio total de sistema operativo. Lo anterior demuestra que los estudiantes se adaptan fácilmente a nuevas herramientas y que ellos prefieren el uso de software abierto a software ilegal, por lo que el constante uso de herramientas comerciales se puede atribuir a los docentes que no están interesados en buscar alternativas abiertas, ya sea porque cuentan con licencias, porque no les interesa que se utilice software ilegal o porque no quieren aprender a manejar nuevas herramientas.

Bibliografía

- [1] Texas Instruments. IEEE Std 1149.1 (JTAG) Testability. *1997 Semiconductor Group*, 1996.
- [2] Michael L. Haungs. The Executable and Linking Format (ELF). <http://www.cs.ucdavis.edu/haungs/paper/node1.html>, 21 September 1998.
- [3] C. Hallinan. *Embedded Linux Premiere A Practical Real-World Approach*. Prentice Hall, 18 September 2006.
- [4] Dominic Rath. *Open On-Chip Debugger*. PhD thesis, University of Applied Sciences Augsburg, 2005.
- [5] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, and Philippe Gerum. *Building Embedded Linux Systems*. O'REILLY, 2008.
- [6] I. Bowman, S. Siddiqi, and M. Tanuan. Concrete Architecture of the Linux Kernel. <http://docs.huihoo.com/linux/kernel/a2/index.html>, 12 February 1998.
- [7] I. Bowman. Conceptual Architecture of the Linux Kernel. <http://docs.huihoo.com/linux/kernel/a1/>, 1998.
- [8] J Feldman. The Linux startup process. *The Linux Expertise Center, Hewlett-Packard Company*.
- [9] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly, 2005.
- [10] R Love. *Linux System Programming*. O'Reilly Media, Inc., 2007.