UNIVERSIDAD NACIONAL DE COLOMBIA

Proyecto de Investigación:

Sistemas Embebidos: Teoría, Implementación y Metodologías de Diseño

Carlos Iván Camargo Bareño

16 de julio de 2009

INFORME FINAL
Sistemas Embebidos: Teoría, Implementación y Metodologías de Diseño
AUTHOR: C. Camargo
E-MAIL: cicamargoba@unal.edu.co

Published by the Universidad Nacional de Colombia

Índice general

Capítulo 1

Introducción

La industria de los semiconductores ha crecido velozmente durante los últimos años, su campo de acción se ha extendido a casi todas las actividades del ser humano (Entretenimiento, salud, seguridad, transporte, educación, etc); los tiempos de los procesos de diseño son cada vez mas cortos, lo cual require herramientas Hardware, Software y metodologías de diseño que ayuden a cumplir con las exigencias impuestas al sistema.

Esta *invasión* digital ha sido posible gracias a la industria de los semiconductores y a las empresas desarrolladoras de software, las primeras haciendo uso de un altísimo grado de integración ponen a disposición de los diseñadores *Systems On Chip* (SoC) en los cuales se integran procesadores de 32 bits con una gran variedad de periféricos tales como: Controladores de dispositivos de red (cableada e inalámbricos), controladores de video, procesadores aritméticos, controladores de memorias (Flash, SDRAM, DDR, USB, SD), codecs de audio, controladores de touch screen, etc; lo cual permite la implementación de aplicaciones completas dentro de uno de estos SoCs.

Por otro lado, las empresas desarrolladoras de SW crean herramientas de programación que permiten manejar toda la capacidad de estos SoCs. colocando a disposición de los diseñadores: compiladores, simuladores, emuladores, librerías, Sistemas Operativos y drivers. Lo cual permite realizar desarrollos complejos en tiempos cortos.

En la actualidad existe un gran número de sistemas operativos (OS) disponibles tanto comerciales como *open source*. La figura 1.1 muestra la utilización actual de OS comerciales en aplicaciones embebidas; si sumamos los sistemas operativos basados en linux se obtiene un valor del 19.3 % lo cual hace ganador al sistema operativo de libre distribución [3]. La figura 1.2 muestra un cuadro comparativo de la utilización de herramientas comerciales y linux; de nuevo se observa que linux es el preferido por los diseñadores. Este resultado es interesante ya que uno de los supuestos puntos débiles del software de libre distribución es el soporte, lo cual no lo hace tan agradable a la hora de realizar aplicaciones comerciales, sin embargo, esto es solo un mito, ya que gracias a que el código fuente está disponible, es posible comprender perfectamente su funcionamiento, lo cual no sucede con el software comercial; además, existen muchos foros de diseñadores y desarrolladores que se encargan de responder las inquietudes, estos foros almacenan todos los mensajes recibidos e incluyen herramientas de búsqueda para poder consultarlos, por regla general de estos foros, se debe buscar primero en estos archivos históricos, muy seguramente alguien más preguntó lo mismo antes que nosotros.

El caracter gratutito de las herramientas de libre distribución no significa, ni mucho menos, mala calidad, todo lo contrario, existen muchas personas que escudriñan su código fuente en búsqueda de posibles

errores, y realizan cambios con el fín de eliminarlo; por lo tanto, se cuenta con miles de personas que están constantemente depurando y perfeccionando una determinada aplicación; esto no ocurre con el software comercial, normalmente el soporte hay que pagarlo y las perosnas involucradas en el desarrollo no son tantas como las que tienen acceso al código fuente del software libre. Por esta razón empresas como PALM, han dejado de lado productos propietarios como el PALM OS para utilizar linux, SUN Microsystems, liberó el código fuente de su sistema operativo Solaris, ya que no era comparable con linux y está en el proceso de liberar uno de sus procesadores.

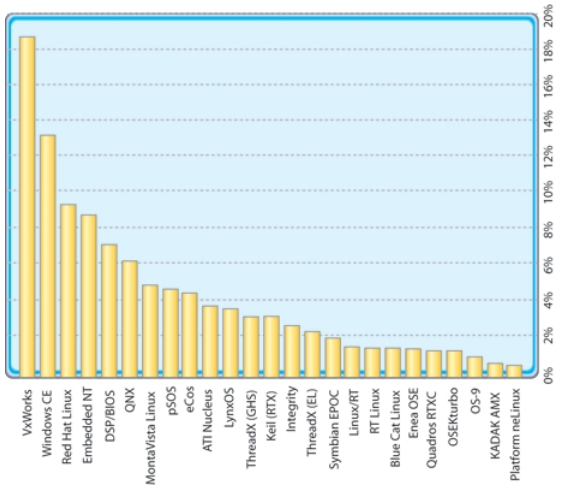

Figura 1.1: Utilización actual de OS para aplicaciones embebidas: Fuente [4].

De lo anterior se deduce que el mundo de los sistemas digitales ha cambiado en forma considerable, y que en la actualidad existen grandes facilidades para el desarrollo de productos de forma rápida y económica [1]. Desafortunadamente estas tendencias no se han aplicado aún en Colombia; existen varias razones para esto:

1. Desactualización de los programas acedémicos: En muchas de las Universidades de Colombia se utilizan tecnologías obsoletas que impiden la aplicación de metodologías de diseño modernas además de impedir el desarrollo de aplicaciones comerciales, un ejemplo de este tipo de tecnologías es la lógica TTL (74XX) y sus equivalentes en CMOS. Aunque vale la pena aclarar que este tipo de tecnología es útil a la hora de enseñar conceptos básicos, no puede ser utilizada como única herramienta de implementación. Un ejemplo de desactualización de metodologías de diseño lo encontramos en las herramientas de programación para microcontroladores, una gran cantidad de Universidades utilizan lenguaje ensamblador, lo cual impide la re-utilización de código y crea dependencias con el HW utilizado.

2. Falta de interés de la Industria: Un gran porcentaje de la industria Colombiana es consumidora de tecnología, es decir, no generan sus propias soluciones, no cuentan con departamentos de Investigación y Desarrollo; esto se debe a la falta de confianza en los productos nacionales y en algunos casos a la inexistencia de producción nacional. Por otro lado, la cooperación entre la Universidad y la industria es muy reducida, debido a falta de políticas en las Universidades que regulen esta actividad y a la poca inversión por parte de las empresas.

[1] En la actualidad cerca del 60 % de los ingenieros trabajan en compañías con menos de 10 desarrolladores de SW [5]

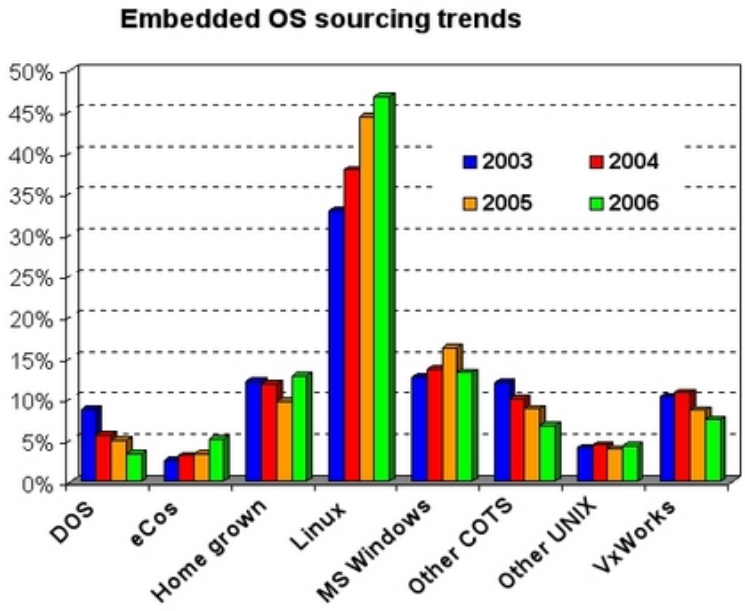

Figura 1.2: Utilización actual de OS para aplicaciones embebidas: Fuente http://www.linuxdevices.com/articles/AT7070519787.html.

3. Políticas del Estado: Casi la totalidad de estos nuevos dispositivos semiconductores deben ser importados, ya que en Colombia no existen distribuidores, por lo tanto, es necesario pagar una serie de impuestos que no desalientan su utilización, estos impuestos están por el orden del 26 % del valor del producto. Por otro lado la apertura económica permite que ingresen productos a bajo precio, con los que no pueden competir los pocos productos desarrollados en el pais, eso sumado a la falta de protección por parte de las políticas estatales condena a los desarrolladores de estos sistemas a la quiebra económica.

Este proyecto resume el trabajo realizado durante los últimos cuatro años en el área de la electrónica digital y más exactamente en el estudio de las metodologías de diseño modernas con aplicaciones comerciales y es presentado para cumplir parcialmente los requisitos de cambio de categoría de profesor asistente a asociado. El presente informe está dividido de la siguiente forma:

- En el capítulo 1 se realiza una breve descripción de los sistemas embebidos, se enumeran sus características, aplicaciones y se hace una descripción de las herramientas HW y SW necesarias para el diseño de los mismos. En los capítulos siguientes se desarrollan casos de estudio encaminados a la comprensión de estos sistemas:
- En el capítulo dos se trabaja con una plataforma comercial de bajo costo: El GameBoy de Nintendo, (gracias a los elevados volúmenes de producción el costo de este dispositivo es bajo alrededor de 40 USD) esta plataforma nos permite desarrollar conceptos básicos como la compilación cruzada, la interfaz HW-SW y los sistemas operativos.
- En el capítulo tres se muestra la implementación de la primera plataforma de desarrollo para sistemas embebidos diseñada en la Universidad Nacional, a pesar de ser muy sencilla permite dar un gran paso en el proceso de fabricación de este tipo de dispositivos.

- En el capítulo cuatro se realiza la implementción de una plataforma de desarrollo que utiliza el sistema operativo linux sobre un arreglo de compuertas programable en campo (FPGA) y se muestran los resultados de una aplicación en el área del Hardware Reconfigurable.
- El capítulo cinco es una guía de adaptación del sistema operativo linux para una arquitectura ARM.

Por último se muestra como algunos de los resultados de este estudio han sido introducidos a los cursos del área de los sistemas digitales en la Universidad Nacional de Colombia y como se ha realizado su difusión en otras Universidades de Colombia.

Capítulo 2

Conceptos Básicos de los Sistemas Embebidos

2.1. Definición

Un Sistema Embebidos es un sistema de propósito específico en el cual, el computador es encapsulado completamente por el dispositivo que el controla. A diferencia de los computadores de propósito general, un Sistema Embebido realiza tareas pre-definidas, lo cual permite su optimización, reduciendo el tamaño y costo del producto [2]

2.2. Características

- Los sistemas embebidos son diseñados para una aplicación específica, es decir, estos sistemas realizan un grupo de funciones previamente definidasm y una vez el sistema es diseñado, no se puede cambiar su funcionalidad. Por ejemplo, el control de un asensor siempre realizará las mismas acciones durante su vida útil.
- Debido a su interacción con el entorno los ES deben cumplir esctríctamente restricciones temporales. El término *Sistemas de Tiempo Real* es utilizado para enfatizar este aspecto.
- Los Sistemas Embebidos son heterogéneos, es decir, están compuestos por componentes Hardware y Software. Los componentes Hardware, como ASICs y Dispositivos Lógicos Programables (PLD) proporcionan la velocidad de ejecución y el cosumo de potencia necesarios en algunas aplicaciones.
- Los Sitemas Embebidos tienen grandes requerimientos en términos de confiabilidad. Errores en aplicaciones como la aviación y el automovilismo, pueden tener consecuencias desastrosas.

2.3. Arquitectura

Una arquitectura típica para un Sistema Embebido se muestra en la Figura 2.1; La cual integra un componente hardware, implementado ya sea en un PLD (CPLD, FPGA) o en un ASIC, conocido con el

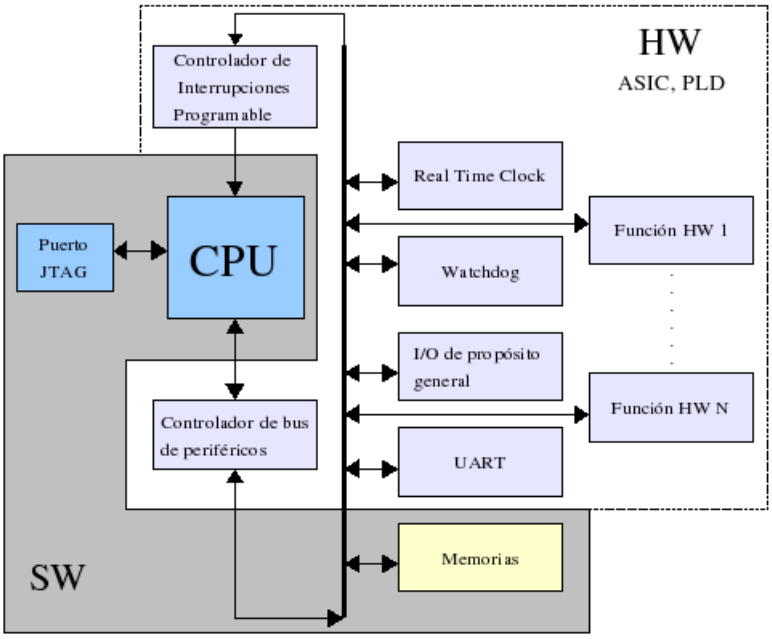

Figura 2.1: Arquitectura de un Sistema Embebido

nombre de periféricos y un componente software (procesador o DSP) capáz de ejecutar software, la parte del procesador está dividida en la CPU (En algunos casos posee una caché) y las unidades de Memoria.

Al momento de diseñar un Sistema Embebido encontramos las siguientes opciones:

- Componente HW y SW Integrado en un dispositivo semiconductor (SoC): En la actualidad existen muchas compañías que fabrican procesadores de 32 bits integrados a una gran variedad de periféricos, lo cual simplifica el diseño y reduce costos (menos componentes y menos área de circuito impreso) [1].
- Componente SW en un SoC y componente HW en una FPGA: Cuando no existen en el mercado SoC con la cantidad de periféricos requerida para una determinada aplicación, es necesario recurrir a la utilización de dispositivos comerciales que implementen dicha operación, en algunas ocaciones el periférico puede relizar funciones muy específicas de modo que no existe en el mercado, la solución es entonces implementar estos dispositivos en una FPGA, también se recomienda la utilización de FPGAs en sistemas que requieren una gran cantidad y variedad de periféricos ya que reduce la complejidad y costo del sistema.
- Componente SW y HW en una FPGA: Esta es tal vez la opción más económica y flexible, pero la de menor desempeño, ya que al utilizar los recursos lógicos de la FPGA para la implementación del procesador (softcore) la lngitud de los caminos de interconexión entre los bloques lógicos aumentan el retardo de las señales . Los procesadores *softcore* más populares en la actualidad son:
  - Microblaze de Xilinx[2]

[1] http://www.sharpsma.com, http://www.atmel.com, http://www.cirrus.com, http://www.samsung.com, http://www.freescale.com, etc

[2] http://www.xilinx.com

- Leon de Gaisler Research [3]
- LatticeMico32 de Lattice Semiconductors[4]
- OpenRisc [5]

2.4. Metodología de Diseño

La Figura 2.2, muestra un diagrama de flujo de diseño genérico para sistemas embebidos [**?**]

El proceso comienza con la *especificación del sistema*, en este punto se describe la funcionalidad y se definen las restricciones físicas, eléctricas y económicas. Esta especificación debe ser muy general y no deben existir dependencias (tecnológicas, metodológicas) de ningún tipo, se suele utilizar lenguajes de alto nivel, como UML, C++. La especificación puede ser verificada a través de una serie de pasos de análisis cuyo objetivo es determinar la validez de los algorítmos seleccionados, por ejemplo, determinar si el algoritmo siempre termina, los resultados satisfacen las especificaciones. Desde el punto de vista de la re-utilización, algunas partes del funcionamiento global deben tomarse de una librería de algorítmos existentes.

Una vez definidas las especificaciones del sistema se debe realizar un modelamiento que permita extraer de estas la funcionalidad. El modelamiento es crucial en el diseño ya que de él depende el paso existoso de la especificación a la implementación. Es importante definir que modelo matemático debe soportar el entorno de diseño. Los modelos más utilizados son: Máquinas de estados finitos, diagramas de flujos de datos, Sistemad de Eventos Discretos y Redes de Petri. Cada modelo posee propiedades matemáticas que pueden explotarse de forma eficiente para responder preguntas sobre la funcionalidad del sistema sin llevar a cabo dispendiosas tareas de verificación. Todo modelo obtenido debe ser verificado para comprobar que cumple con las restricciones del sistema.

Una vez se ha obtenido el modelo del sistema se procede a determinar su *arquitectura*, esto es, el número y tipo de componentes y su inter-conexión. Este paso no es más que una exploración del espacio de diseño en búsqueda de soluciones que permitan la implementación de una funcionalidad dada, y puede realizarse con varios criterios en mente: Costos, confiabilidad, viabilidad comercial.

Utilizando como base la arquitectura obtenida en el paso anterior las tareas del modelo del sistemas son mapeadas dentro de los componentes. Esto es, asignación de funciones a los componentes de la arquitectura. Existen dos opciones a la hora de implementar las tareas o procesos:

1. Implementación Software: La tarea se va a ejecutar en un procesador.
2. Implementación Hardware: La tarea se va a ejecutar en un sistema digital dedicado.

Para cumplir las especificaciones del sistema algunas tareas deben ser implementadas en Hardware, esto con el fín de no ocupar al procesador en tareas cíclicas, un ejemplo típico de estas tareas es la generación de bases de tiempos. La decisión de que tareas se implementan en SW y que tareas se implementan en HW recibe el nombre de *particionamiento*, esta selección es fuertemente dependiente de restricciones económicas y temporales.

Las tareas Software deben compartir los recursos que existan en el sistema (procesador y memoria), por lo tanto se deben hacer decisiones sobre el orden de ejecución y la prioridad de estas. Este proceso

[3] http://www.gaisler.com/
[4] http://www.latticesemi.com
[5] http://www.opencores.com

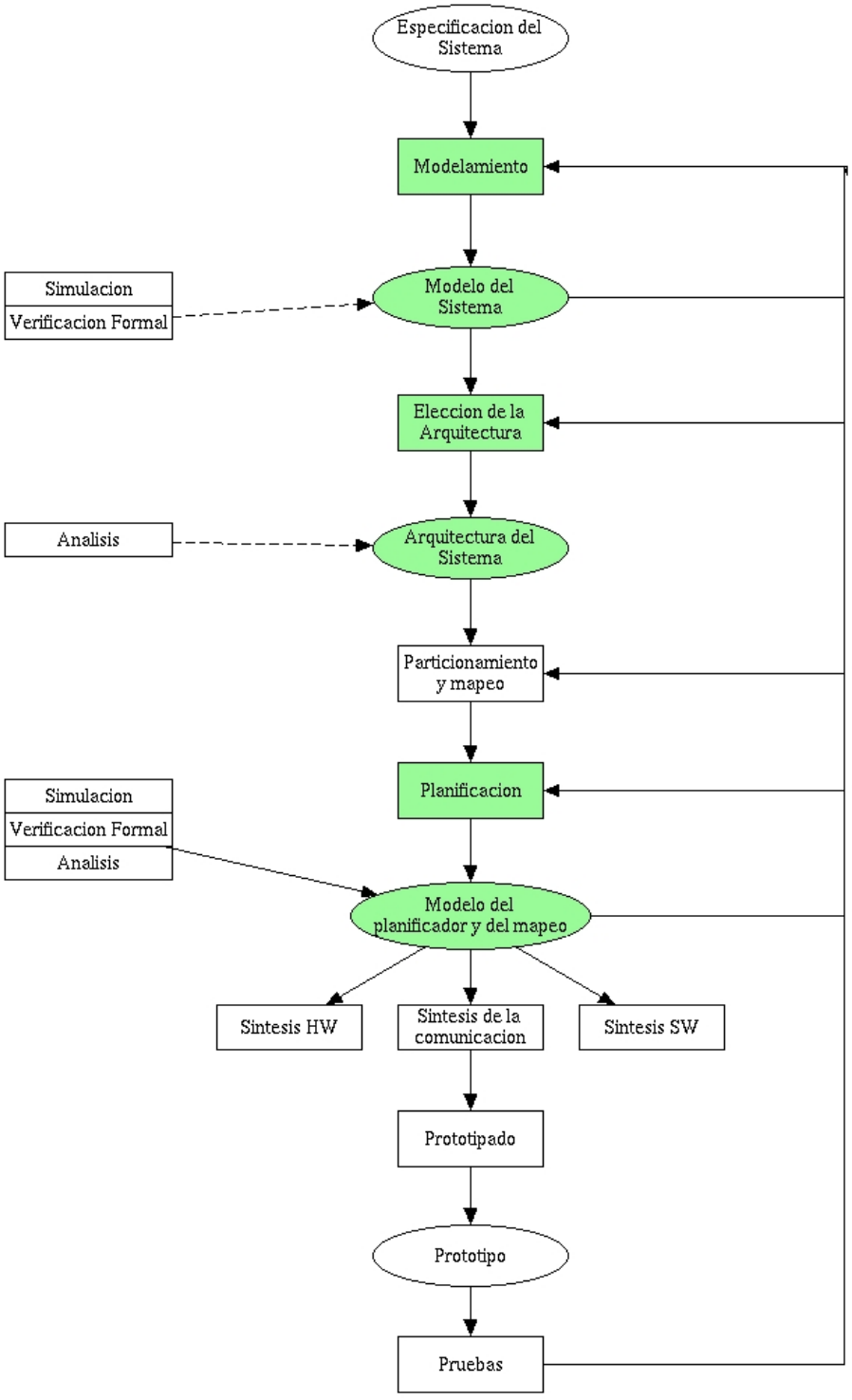

Figura 2.2: Flujo de Diseño de un Sistema Embebido

recibe el nombre de *planificación*. En este punto del diseño el modelo debe incluir información sobre el mapeo, el particionamiento y la planificación del sistema.

Las siguientes fases corresponden a la implementación del modelo, para esto las tareas hardware deben ser llevadas al dispositivo elegido (ASIC o FPGA) y se debe obtener el ″ejecutable″ de las tareas software, este proceso recibe el nombre de *síntesis* HW y SW respectivamente, así mismo se deben sintetizar los mecanismos de comunicación.

El proceso de prototipado consiste en la realización física del sistema, finalmente el sistema físico debe someterse a pruebas para verificar que se cumplen con las especificaciones iniciales.

Como puede verse en el flujo de diseño existen realimentaciones, estas realimentaciones permiten depurar el resultado de pasos anteriores en el caso de no cumplirse las especificaciones iniciales

2.4.1. Herramientas Software de libre distribución *GNU toolchain*

En el mercado existe una gran variedad de herramientas de desarrollo para Sistemas Embebidos, sin embargo, en este estudio nos centraremos en el uso de las herramientas de libre distribución; esta elección se debe a que la mayoría de los productos comerciales utilizan el toolchain de GNU[6] internamente y proporcionan un entorno gráfico para su fácil manejo. Otro factor considerado a la hora de realizar nuestra elección es el económico, ya que la mayoría de los productos comerciales son costosos y poseen soporte limitado. Por otro lado, el toolchain de GNU es utilizado ampliamente en el medio de los diseñadores de sistemas embebidos y se encuentra un gran soporte en múltiples foros de discusión (ver Figura 2.3).

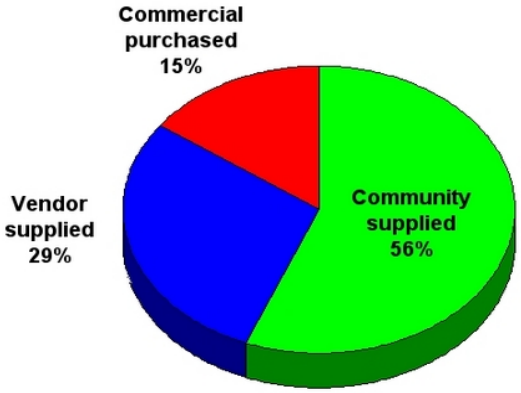

Figura 2.3: Tendencia de utilización de herramientas de desarrollo

2.4.2. Componentes del *GNU toolchain*

2.4.3. GNU binutils[1]

Son una colección de utilidades para archivos binarios y estan compuestas por:

[6] http://www.gnu.org

- **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 ejecutabe para determinar que nombre de atchivo y número de lpinea 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 miembros del archivo.
- **as** Utilidad que compila la salida del compilador de C (GCC).
- **c++filt** Este program 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 compilado 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 esccribibr 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 almenos 4 caracteres de longitud.
- **strip** Elimina todos los símbolos de un archivo tipo objeto.

2.4.4. GNU Compiler Collection[2]

El *GNU Compiler Collection* normalmente llamado GCC, es un grupo de compiladores de lenguajes de programación producido por el proyecto GNU. Es el compilador standard para el software libre de los sistemas operativos basados en Unix y algunos propietarios como Mac OS de Apple.

Lenguajes

GCC soporta los siguientes lenguajes:

- **ADA**
- **C**
- **C++**
- **Fortran**

- **Java**
- **Objective-C**
- **Objective-C++**

Arquitecturas

- **Alpha**
- **ARM**
- **Atmel AVR**
- **Blackfin**
- **H8/300**
- **System/370, System/390**
- **IA-32 (x86) and 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**
- **MorphoSys**

Como puede verse GCC soporta una gran cantidad de lenguajes de programación, sin embargo, en el presente estudio solo lo utilizaremos como herramienta de compilación para C y C++. Una característica de resaltar de GCC es la gran cantidad de plataformas que soporta, esto lo hace 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 HW[7], lo cual no ocurre al utilizar lenguaje ensamblador.

Por otro lado, el tiempo requerido para realizar aplicaciones utilizando C o C++ disminuye, ya que no es necesario aprender las instrucciones en assembler de una plataforma determinada; además, la

[7]Esto recibe el nombre de re-utilización de código

disponibilidad de librerías de múltiples propósitos reduce aún más los tiempos de desarrollo, permitiendo de esta forma tener bajos tiempos *time to market* y reducir de forma considerable el costo del desarrollo. Una consecuencia de esto se refleja en el número de desarrolladores en un grupo de trabajo, en la actualidad casi el 60 % de las empresas desarrolladoras de dispositivos embebidos tiene grupos con menos de 10 desarrolladores 2.4.

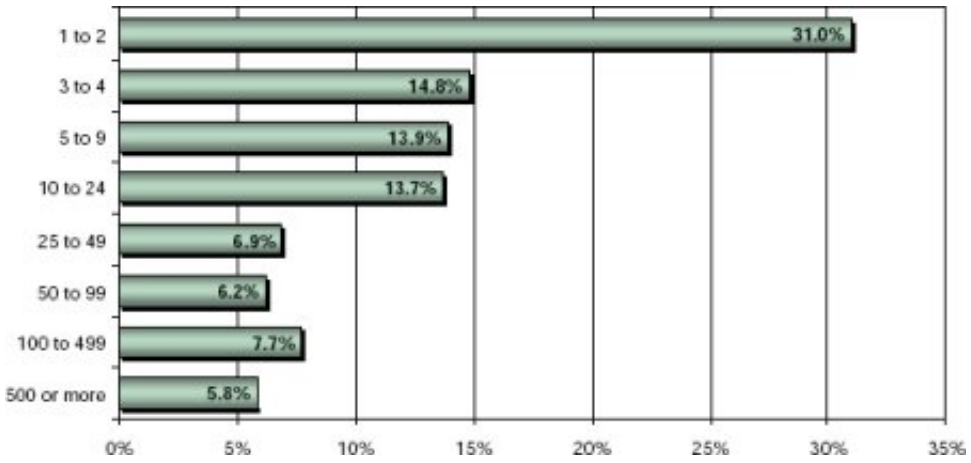

Figura 2.4: Número promedio de desarrolladores por compañía. Fuente Venture Development Corp

2.4.5. GNU Debugger

El depurador oficial de GNU (GDB), es un depurador que al igual que GCC tiene soporte para múltiples lenguajes y plataformas. GDB permite al usuario 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 no posee una interfaz gráfica, se han desarrollado varios front-ends como DDD o GDB/Insight. A continuación se muestra un ejemplo de una sesión con gdb.

```
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host libthread_db
library "/lib/libthread_db.so.1".

(gdb) run
Starting program: /home/sam/programming/crash
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xc11000
This program will demonstrate gdb

Program received signal SIGSEGV, Segmentation fault.
0x08048428 in function_2 (x=24) at crash.c:22
22          return *y;
(gdb) edit
(gdb) shell gcc crash.c -o crash -gstabs+
```

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
warning: cannot close "shared object read from target memory": File in wrong format
'/home/sam/programming/crash' has changed; re-reading symbols.
Starting program: /home/sam/programming/crash
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xa3e000
This program will demonstrate gdb
24
Program exited normally.
(gdb) quit
```

2.4.6. C Libraries

Adicionalmente es necesario contar con una librería que proporcione las librerías standard de C: stdio, stdlib, math; las más utilizadas en sistemas embebidos son:

- **glibc**[8] Es la librería C oficial del proyecto GNU. Uno de los inconvenientes 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**[9] Es una librería diseñada especialmente para sistemas embebidos, es mucho más pequeña que **glibc**.
- **newlib**[10] 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 [6].
- **diet libc**[11] 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.

2.5. Obtención y utilización del *GNU toolchain*

El primer paso en nuestro estudio consiste en tener una cadena de herramientas funcional que soporte la familia de procesadores a utilizar. La arquitectura sobre la cual realizaremos nuestra investigación es la ARM (Advanced Risc Machines), ya que un la más utilizada en la actualidad por los diseñadores de sistemas embebidos (ver figura 2.5) y se encuentran disponibles una gran variedad de herramientas para esta arquitectura. Existen dos formas de obtener la cadena de herramientas GNU:

1. Utilizar una distribución precompilada: Esta es la via más rápida, sin embargo, hay que tener cuidado al momento de instalarlas, ya que debe hacerse en un directorio con el mismo *path* con el que fueron creadas. por ejemplo */usr/local/gnutools*; si esto no se cumple, las herramientas no funcionarán de forma adecuada.
2. Utilizar un script de compilación: Existen disponibles en la red una serie de *scripts* que permiten descargar, configurar, compilar e instalar la cadena de herramientas, la ventaja de utilizar este método es que es posible elegir las versiones de las herramientas instaladas, al igual que el directorio de instalación. En este estudio utilizaremos los *scripts* creados por Dan Kegel [8].

2.5.1. Conceptos Previos

Antes de hablar sobre el uso de las herramientas GNU hablaremos sobre varios conceptos que deben quedar claros; estos son: El flujo de diseño software, y el formato ELF.

[8] http://www.gnu.org/software/libc/
[9] http://uclibc.org/
[10] http://sources.redhat.com/newlib/
[11] http://www.fefe.de/dietlibc/

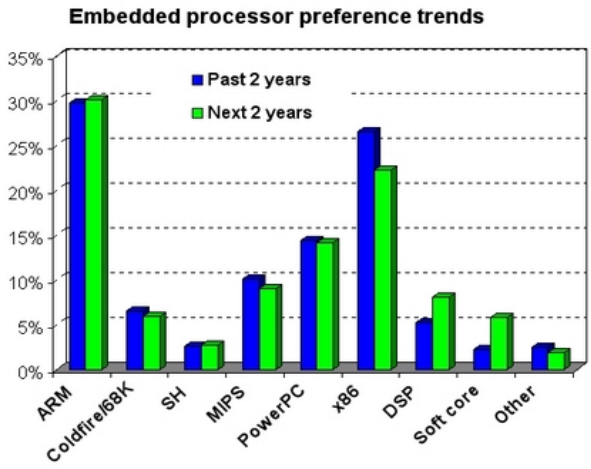

Figura 2.5: Tendencia del mercado de procesadores para sistemas embebidos. Fuente:[7]

El formato ELF

El formato ELF (*Executable and Linkable Format*) Es un stándard para objetos, librerías y ejecutables. Como puede verse en la figura 2.6 el formato ELF está compuesto por varias 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 [9].

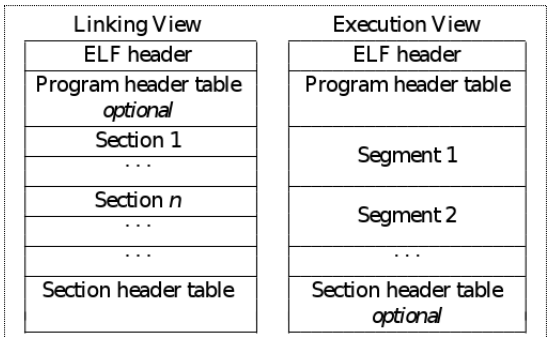

Figura 2.6: Tendencia del mercado de procesadores para sistemas embebidos. Fuente:[7]

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

- **.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 enlacedinámico

- **.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 .rodta1** Datos de solo-lectura (ROM)
- **.shstrtab** Nombres de secciones.
- **.symtab** Tabla de símbolos.
- **.text** Instrucciones ejecutables (ROM)

Para aclarar un poco este concepto consideremos el siguiente código:

```
#include <stdio.h>

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;
  }
  return 0;
}
```

En el ejemplo observamos que tenemos dos variables, una sin inicializar (*i*) y otra inicializada (*j*); estas variables estarán en las secciones *.bss* y *.data* respectivamente, así mismo los caracteres "Printing " Estarán incluidos en la sección *.rodata* ya que son datos que no cambian a lo largo de la ejecución del programa. Las instrucciones que forman el programa residen en la sección *.text*. A continuación se muestra la información de este archivo una vez compilado, utilizando la herramienta *objdump* de los utilitarios binarios *binutils* y más específicamente el comando:

objdump -h hello

```
hello:     file format elf32-littlearm

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000014  000080f4  000080f4  000000f4  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .hash         00000050  00008108  00008108  00000108  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .dynsym       000000f0  00008158  00008158  00000158  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .dynstr       0000008a  00008248  00008248  00000248  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .init         00000010  000082f4  000082f4  000002f4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  7 .text         0000017c  00008348  00008348  00000348  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  8 .fini         0000000c  000084c4  000084c4  000004c4  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  9 .rodata       00000010  000084d0  000084d0  000004d0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 .eh_frame     00000004  000084e0  000084e0  000004e0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 16 .data         0000000c  000105ac  000105ac  000005ac  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 17 .bss          00000004  000105b8  000105b8  000005b8  2**0
                  ALLOC
 18 .comment      00000094  00000000  00000000  000005b8  2**0
                  CONTENTS, READONLY
```

En el item 9, se observa la información correspondiente a la sección *.rodata*, la primera columna corresponde al tamaño de la sección, en este caso 16 bytes, las columnas 2 y 3 corresponden a la dirección de ejecución (VMA) y a la dirección de carga (LMA) respectivamente. La columna 4 indica la dirección dentro del ejecutable donde se encuentra almacenada esta información, en este caso la *0x000004d0*, utilizando la herramienta *hexdump* podemos ver el contenido de esa dirección en el archivo ejecutable:

hexdump -C hello — grep -i 000004d0

```
000004d0  50 72 69 6e 74 69 6e 67  20 25 64 0a 00 00 00 00  |Printing %d.....|
```

2.5.2. Flujo de diseño software

En la figura 2.7 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.

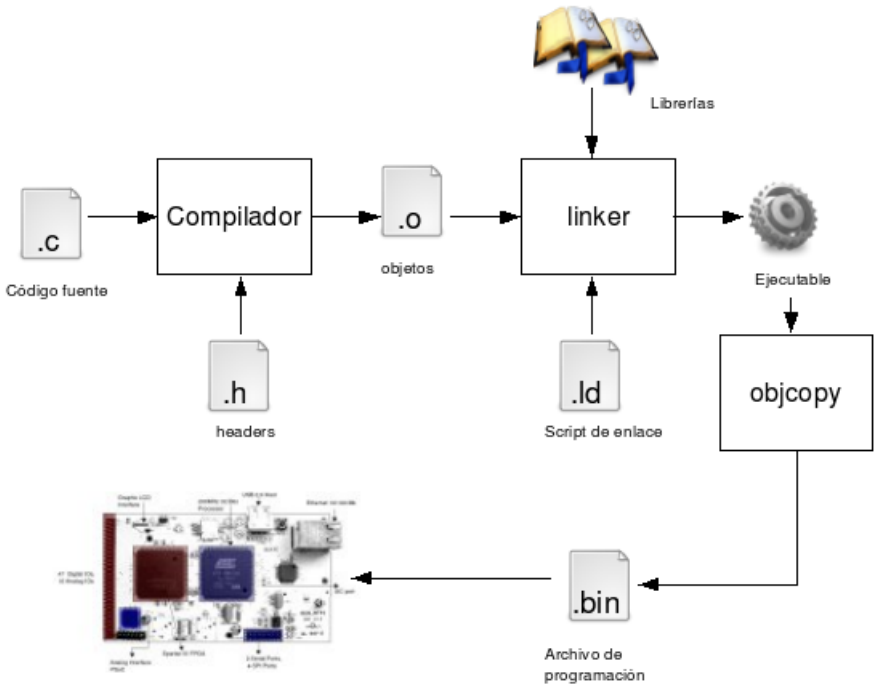

Figura 2.7: Tendencia del mercado de procesadores para sistemas embebidos. Fuente:[7]

A continuación se realiza una breve descripción de los pasos necesarios para generar un ejecutable para un sistema embebido:

1. **Escritura del código fuente:** Creación del código fuente en cualquier editor de archivos de texto.
2. **Compilación:** Utilizando el compilador gcc se compila el código fuente; vala la pena mencionar que en este punto el compilador solo busca en los encabezados (*headers*) de las librerías la definición de una determinada función, como por ejemplo el *printf* en el archivo *stdio.h*. Como resultado de este paso se obtiene un archivo tipo objeto.
3. **Enlazado:** En esta etapa se realizan dos tareas:
   a) Se enlazan los archivos tipo objeto del proyecto, junto con las librerías, si una determinada función no es edfinida por ninguna de las librerías pasadas como parámetro al linker, este generará un error y no se generará el ejecutable.
   b) Se define la posiciónes físicas de las secciones del ejecutable tipo ELF, esto se realiza a través de un link de enlazado el cual 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 herramiento *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.
5. **Descarga del programa a la plataforma**. Dependiendo de la plataforma existen varios métodos para descargar el archivo de programación a la memoria de la plataforma de desarrollo:
   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.
   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 determinada posición de memoria.
6. **Depuración** Una vez se descarga la aplicación a la plataforma es necesario someterla a una serie de pruebas con el fín de probar su correcto funcionamiento. Esto se puede realizar con el depurador GNU (GDB) y una interfaz de comunicación que puede ser un puerto serie o un adaptador de red.

2.6. Makefile

Como pudo verse en la sección 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 de comandos, es necesario esribir una serie de comandos cada vez que se realiza un cambio en el código fuente, lo cual resulta poco práctico. Para realizar este proceso de forma automática se creó la herramienta make, la cual recibe como entrada un archivo que normalmente recibe el nombre de *Makefile* o *makefile*. Un ejemplo de este tipo de archivo se muestra a continuación:

```
1  SHELL = /bin/sh
2
3  basetoolsdir = /home/at91/gcc-3.4.5-glibc-2.3.6/arm-softfloat-linux-gnu
4  bindir   = ${basetoolsdir}/bin
5  libdir   = ${basetoolsdir}/lib/gcc/arm-softfloat-linux-gnu/3.4.5
6
7  CC       = arm-softfloat-linux-gnu-gcc
8  AS       = arm-softfloat-linux-gnu-as
9  LD       = arm-softfloat-linux-gnu-ld
10 OBJCOPY = arm-softfloat-linux-gnu-objcopy
11
12 CFLAGS  =-mcpu=arm920t -I. -Wall
13 LDFLAGS =-L${libdir} -l gcc
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}/
24
25 all: hello_world
26
27 hello_world: ${OBJS} ${ASFILES} ${LIBS}
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 PREPROCESS.c = $(CC) $(CPPFLAGS) $(TARGET_ARCH) -E -Wp,-C,-dD,-dI
35
36 %.pp : %.c FORCE
```

```
37          $(PREPROCESS.c) $< > $@
```

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, específicamente los compiladores de C (CC), de assembler (AS), el linker (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 assembler que contiene el proyecto, para este caso *arm_init.o*. Las líneas 12 y 13 definen dos variables especiales que se pasan directamente al copmílador de C (CFLAGS) y al liniker (LDFLAGS)

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

make ejecutará el comando:

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

Observemos los comandos asociados a la etiqueta *hello_world:* En la misma línea aparecen *$OBJS $ASFILES $LIBS* esto le indica a la herramienta *make* que antes de ejecutar los comandos asociados a este label, debe realizar las acciones necesarias para generar *$OBJS $ASFILES $LIBS* o lo que es lo mmismo: *main.o, debug_io.o, at91rm9200_lowlevel.o, p_string.o, arm_init.o y libgcc.a*. *make* tiene predefinidas una serie de reglas para compilar los archivos .c la regla es de la forma:

```
.c.o:
        $(CC) $(CFLAGS) -c $<
.c:
        $(CC) $(CFLAGS) $@.c $(LDFLAGS) -o $@
```

Lo cual le indica a la herramienta make que para generar un archivo *.o* a partir de uno *.c* es necesario ejecutar *$(CC) $(CFLAGS) -c $¡*; de aqui la importancia de definir bien la variable de entorno *CC* cuando trabajamos con compiladores cruzados[12]. Hasta este punto al ejecutar el comando: *make hello_world*, *make* realizaría las siguientes operaciones:

```
arm-softfloat-linux-gnu-gcc  -mcpu=arm920t -I. -Wall   -c -o main.o main.c
arm-softfloat-linux-gnu-gcc  -mcpu=arm920t -I. -Wall   -c -o debug_io.o debug_io.c
arm-softfloat-linux-gnu-gcc  -mcpu=arm920t -I. -Wall   -c -o at91rm9200_lowlevel.o at91rm9200_lowlevel.c
arm-softfloat-linux-gnu-gcc  -mcpu=arm920t -I. -Wall   -c -o p_string.o p_string.c
arm-softfloat-linux-gnu-as    -o arm_init.o arm_init.s
```

En las líneas 28 se realiza el proceso de enlazado; al *linker* 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*
- **$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.

[12] Un compilador cruzado genera código para una plataforma diferente en la que se está ejecutando, por ejemplo, genera ejecutables para ARM pero se ejecuta en un x86

Bibliografía

[1] Aleph 1. Building the Toolchain. http://www.aleph1.co.uk/node/279.

[2] Wikipedia. Wikipedia, the free encyclopedia. http://en.wikipedia.org/.

[3] C. Lanfear, S. Balacco, and M. Volckmann. The 2005 Embedded Software Strategic Market Intelligence Program. *Venture Development Corporation http://www.vdc-corp.com*, August 2005.

[4] J. Turley. Embedded systems survey: Operating systems up for grabs. *Embedded Systems Programming*, May 2004.

[5] Venture Development Corp. Small teams dominate embedded software development. http://www.linuxdevices.com/articles/AT7019328497.html, 1 June 2006.

[6] Bill Gatliff. Porting and Using Newlib in Embedded Systems. http://venus.billgatliff.com/node/3.

[7] Linuxdevices. Embedded Linux market snapshot, 2005. *http://www.linuxdevices.com*, 4 May 2005.

[8] Dan Kegel. Building and Testing gcc/glibc cross toolchains. http://www.kegel.com/crosstool/, 20 February 2006.

[9] Michael L. Haungs. The Executable and Linking Format (ELF). http://www.cs.ucdavis.edu/ haungs/paper/node1.html, 21 September 1998.