Simulación y test

Xavier Vilajosana Guillén

PID_00177264

Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

Uno de los aspectos más importantes y a la vez más olvidados en el desarrollo de aplicaciones es el que hace referencia al proceso de test y evaluación del software resultante. En los sistemas empotrados, el proceso de verificación y test aún adquiere mayor importancia dadas las características de estos sistemas, que hace difícil, en muchos casos, la interacción y el seguimiento del código en ejecución. La importancia de verificar el funcionamiento correcto de un programa es capital, puesto que en la mayoría de los casos los sistemas empotrados y de propósito específico están concebidos para funcionar de una manera autónoma, permanente y desatendida.

En este módulo, presentaremos las herramientas más utilizadas en el proceso de desarrollo de software para sistemas empotrados. De una manera específica, trataremos de las herramientas más utilizadas y detallaremos su funcionamiento y uso en la cadena de desarrollo. No obstante, en el módulo también presentaremos metodologías de evaluación que permiten de una manera sistemática hacer tests y evaluar software en general. Así, los conceptos aquí introducidos son del todo válidos para la evaluación del software en general.

El módulo nos presenta las principales herramientas o los dispositivos de hardware que, en general, son utilizados durante el desarrollo de software para sistemas empotrados. Entre otros, los conocidos osciloscopios o los analizadores de espectros, que nos permiten ver en el nivel de señal lo que sucede en nuestro dispositivo. A continuación, el módulo nos presenta las herramientas de software que conforman nuestro entorno de trabajo, entre las que se encuentra la herramienta principal para evaluar todo desarrollo de software: el depurador.

Siguiendo con la presentación de herramientas, se introducen los simuladores, otro tipo de herramienta hardware y software que resulta muy útil durante el proceso de desarrollo. Existen diferentes tipos de simuladores, desde placas de desarrollo de una arquitectura específica hasta programas que simulan el comportamiento de una arquitectura determinada.

Finalmente, el módulo nos presenta, desde un punto de vista más genérico, una metodología de evaluación del software. Esta metodología se emplea para verificar programas en cualquier ámbito y no específicamente para sistemas empotrados. No obstante, el uso de una metodología de verificación es imprescindible para garantizar la calidad de nuestros desarrollos.

Objetivos

El estudio de estos materiales os ayudará a lograr los objetivos siguientes:

1. Conocer las herramientas de desarrollo más habituales, y también su configuración o adaptación.

2. Conocer las técnicas más usadas para depurar sistemas empotrados.

3. Conocer las técnicas más relevantes de simulación en sistemas empotrados.

4. Conocer las técnicas de test más útiles.

1. Herramientas de desarrollo

Para trabajar en el desarrollo de software para sistemas empotrados, se hace del todo necesario tener una serie de herramientas que conformen el entorno de desarrollo[1]. Este va en muchos casos ligado a la plataforma de hardware y software elegido y, a menudo, es suministrado por el mismo fabricante del hardware. Tanto es así que, en muchos casos, cuando se selecciona un microcontrolador, esta selección no se hace tanto por las propiedades o por las características del chip como por las herramientas y por el software que facilita el fabricante.

(1)En inglés, *tool chain*.

No obstante, no todas las herramientas necesarias se encuentran en el entorno de desarrollo. Se necesitan, entre otras, herramientas de laboratorio que nos permitan elaborar un análisis más detallado de la vertiente electrónica de los dispositivos. En esta sección haremos un repaso de las herramientas principales.

1.1. Dispositivos usados para la programación y para el desarrollo

En este apartado presentamos los principales dispositivos que constituyen el entorno de desarrollo para sistemas empotrados o de propósito específico.

1.1.1. Osciloscopio

Un osciloscopio (podéis ver la fotografía que aparece en este subapartado) es un equipo de laboratorio utilizado para examinar cualquier señal eléctrica, tanto analógica como digital.

Los osciloscopios se utilizan en ocasiones para observar rápidamente la tensión en un pin particular o, en ausencia del analizador lógico, para hacer cualquier observación eléctrica más sofisticada. No obstante, el número de entradas es bajo (normalmente cuatro). Para la aplicación de depuración de código, pues, resulta una herramienta con un uso menos importante que un analizador lógico.

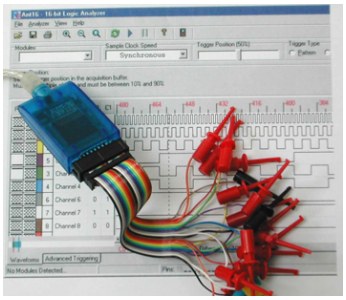
Osciloscopio

1.1.2. Analizador lógico

Un analizador lógico (podéis ver la ilustración de este subapartado) es un equipo de laboratorio diseñado especialmente para evaluar hardware digital. Puede tener docenas o, incluso, centenares de entradas, cada una capaz de detectar una sola cosa: cuando la señal eléctrica está en el nivel lógico 1 o 0.

Más concretamente, un analizador lógico es un instrumento de medida que captura los datos de un circuito digital y los muestra para analizarlos posteriormente, de manera similar a como lo hace un osciloscopio, pero, a diferencia de este, es capaz de visualizar las señales de múltiples canales. Además de permitir visualizar los datos y verificar el funcionamiento correcto del sistema digital, puede medir tiempo entre cambios de nivel, número de estados lógicos, etc. La captura de datos desde un analizador lógico se realiza a partir de la conexión de un cable a la salida lógica apropiada en el bus de datos que hay que medir.

Los analizadores son empleados principalmente para detectar errores y comprobar prototipos antes de su fabricación, analizando las entradas y posteriormente el comportamiento de sus salidas. Hoy en día, la mayoría de los analizadores lógicos disponen de una conexión al PC donde se pueden ver los datos mediante una plataforma software.



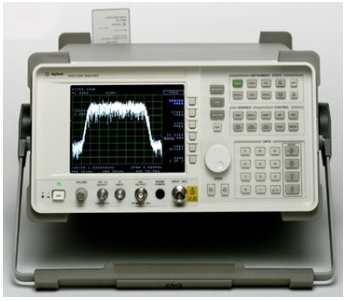
Analizador lógico USB con interfaz para el PC

1.1.3. Analizador de espectros

Un analizador de espectro (podéis ver la ilustración de este subapartado) es un equipo de medida electrónico que permite visualizar en una pantalla los componentes espectrales en un espectro de frecuencias de las señales presentes en la entrada, que pueden ser cualquier tipo de ondas eléctricas, acústicas u ópticas.

En el eje de ordenadas se suele presentar en una escala logarítmica el nivel en dBm del contenido espectral de la señal. En el eje de abscisas se representa la frecuencia, en una escala que es función de la separación temporal y el número de muestras capturadas. Se denomina **frecuencia central** del analizador la que se corresponde con la frecuencia del punto medio de la pantalla.

A menudo se mide con ellos el espectro de la potencia eléctrica. En la actualidad, está siendo sustituido por los analizadores de red.



Analizador de espectro Agilent

1.1.4. Analizador de red

Un analizador de redes es un instrumento capaz de analizar las propiedades de las redes eléctricas, especialmente las propiedades asociadas con la reflexión y la transmisión de señales eléctricas. Su uso, entre muchos otros, es el de estudiar las señales de radiofrecuencia que generan las comunicaciones inalámbricas de algunos dispositivos.

Hay dos tipos principales de analizadores de redes:

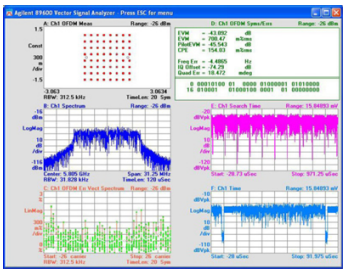
- **Analizador de redes escalar (SNA, *scalar network analyzer*)**[2]. El analizador de redes escalar tiene solo propiedades de amplitud.

[2]Un analizador del tipo SNA tiene un funcionamiento idéntico a un analizador de espectro combinado con un generador de escaneo.

- **Analizador de redes vectorial (VNA, *vector network analyzer*)**[3]. El analizador de redes vectoriales tiene propiedades de amplitud y fase.

[3]Un analizador del tipo VNA también puede denominarse *medidor de ganancia y fase* o *analizador de redes automático*.

En la actualidad, la mayoría de los analizadores de red presentan una interfaz para ser controlados desde un ordenador (podéis ver la imagen que ilustra este subapartado).



Panel de visualización de espectros de un analizador de red Agilent

1.1.5. Multímetro

Un multímetro, polímetro o *tester* (podéis ver la figura que acompaña este subapartado) es un instrumento de medida electrónico que incorpora varias funcionalidades. La mayoría lleva como mínimo un amperímetro, un voltímetro y un óhmetro. En el desarrollo de sistemas empotrados se utiliza durante el proceso de prototipado, puesto que permite:

- Un **comprobador de continuidad**, que pita cuando hay conducción eléctrica entre las dos sondas (en caso de que una pista tenga un corte o error, no pita).

- Lectura de valores en pistas en lugar de usar el osciloscopio, puesto que es más manejable.

- Medida de voltajes y corrientes pequeñas y resistencias elevadas.

Tester

1.1.6. JTAG

Join Test Action Group (JTAG) es un grupo formado por fabricantes de electrónica de todo el mundo que hacia el año 1985 decidieron encontrar una solución común al problema de depuración (*debugging*) de software. La solución se convirtió en el estándar 1149.1-1990 del IEEE Standard Test Access Port and Boundary-Scan Architecture. El IEEE Std 1149.1 permite que instrucciones de test y datos puedan ser cargadas en la memoria de los dispositivos y posteriormente recoger los resultados de los tests de la memoria del dispositivo.

JTAG de Atmel

Esta iniciativa tenía, entre otros, los objetivos siguientes:

- Mejorar el test tradicional (ad hoc para cada dispositivo).
- Mejorar la eficiencia del test.
- Reducir el coste del diseño de tests.
- Reducir el tiempo de producción reduciendo el tiempo de la fase de test.
- Permitir el test aislado de partes de dispositivos.
- Ofrecer un acceso más simple a los dispositivos.

La especificación de JTAG requiere los elementos siguientes:

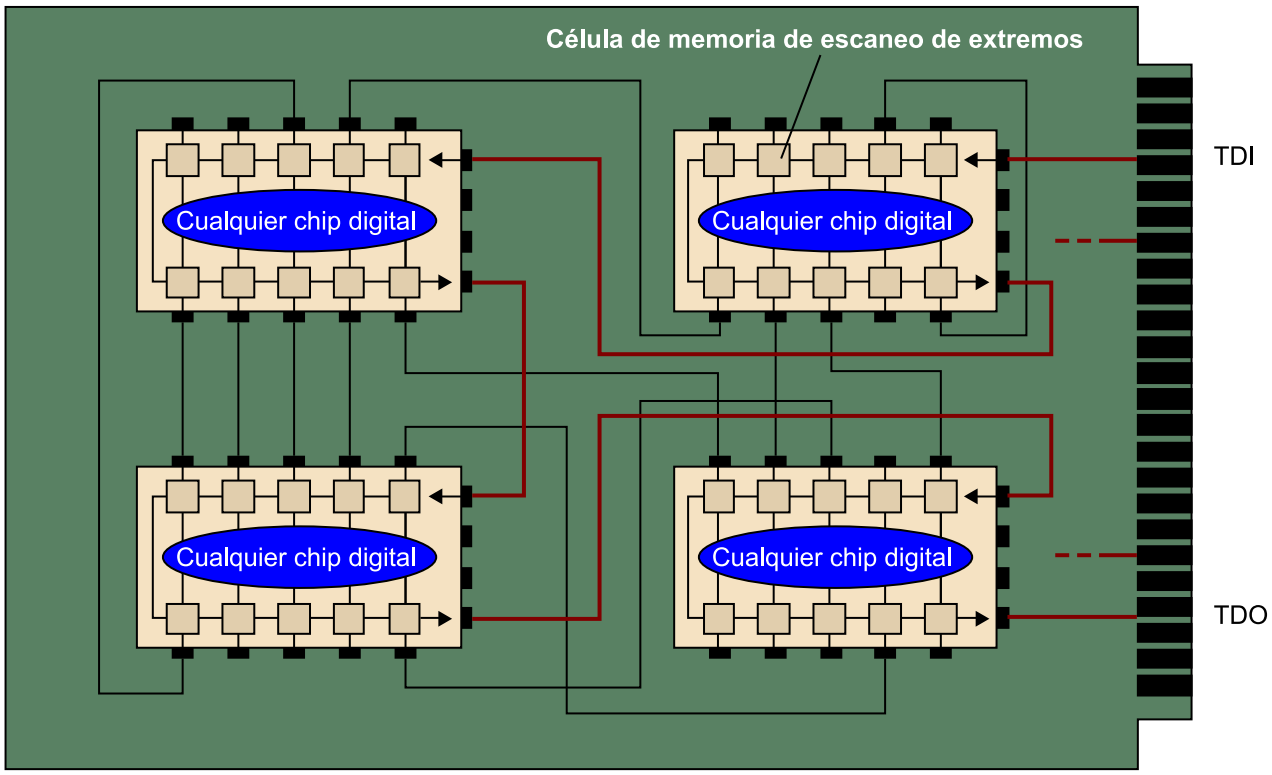
1) Un puerto denominado **test access port** (TAP).
2) Un controlador denominado **TAP controller**.
3) Un **registro de instrucciones** (IR).
4) Un conjunto de **registros de datos** (TDR).

El *TAP controller* genera señales de reloj y de control para el circuito del JTAG. El *TAP controller* se implementa como una máquina de estados finita que obedece a cambios de estados impuestos en el IR de una manera serial. El TAP tiene los pines siguientes:

- *Test data in* (TDI).
- *Test data out* (TDO).
- *Test clock* (TCK).
- *Test modo select* (TMS).
- *Test reset* (TRST), que es opcional.

El funcionamiento de la plataforma se basa en la técnica de escaneo de extremos (*boundary scan*) y que consiste básicamente en que observa y permite actuar sobre los extremos de las entradas y salidas de un chip. En particular, su funcionamiento se basa en el uso de celdas de memoria anexadas a cada pin, las celdas de memoria forman un registro de decalaje (*shift register*) denominado *boundary-scan register* (BSR). El BSR es parte de los TDR de la especificación JTAG. La primera celda es conectada al pin TDI y la última celda es conectada al pin TDO. El BSR es controlado por la señal de reloj TCK. En modo normal, las celdas simplemente retransmiten la señal que hay en el pin al que están anexadas. En el modo test, el valor de cada pin puede ser asignado haciéndole llegar un valor desde el pin TDI mediante decalajes (*shifts*) del BSR. Del mismo modo, se pueden leer directamente los valores de las salidas del chip haciendo llegar la información del BSR al TDO (mediante decalajes del BSR).

Arquitectura de escaneo de extremos utilizada por JTAG

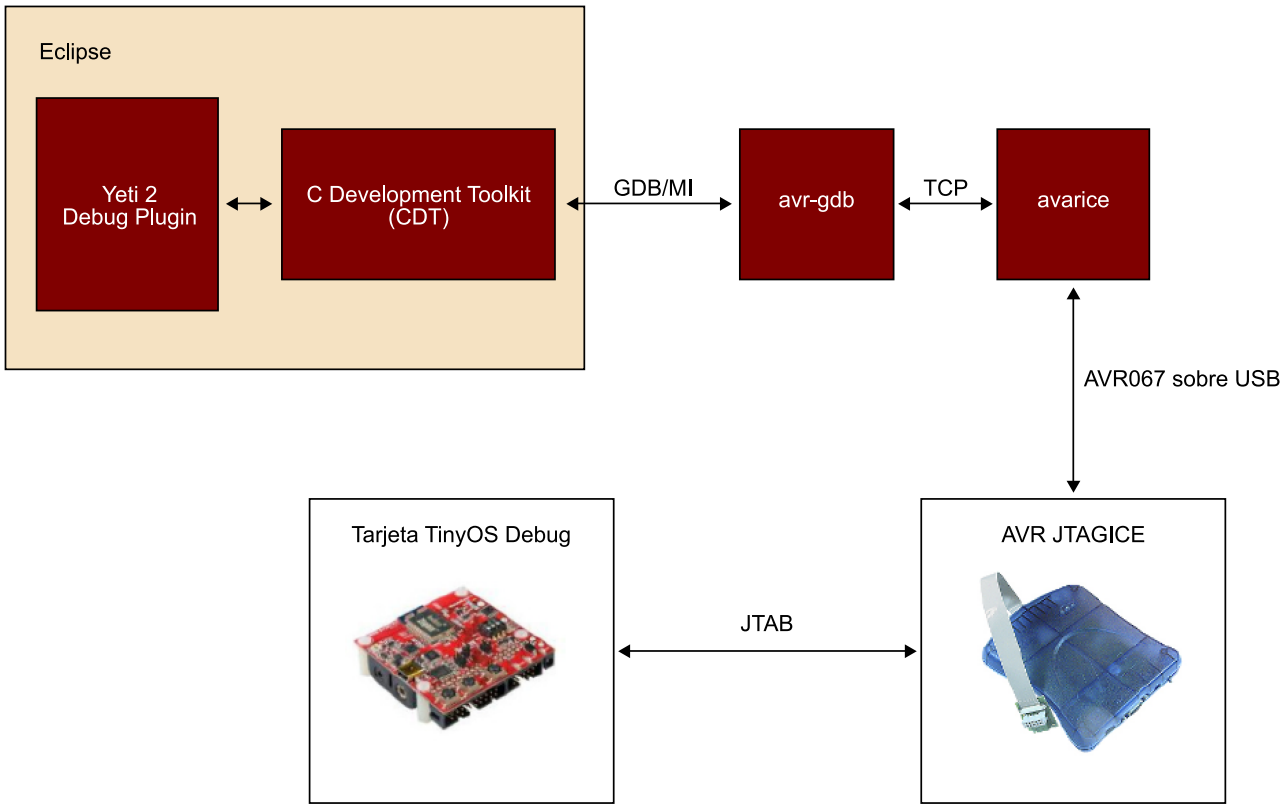


En la práctica, el JTAG es la especificación de un formato que es implementado en un dispositivo que se conecta a un puerto denominado *TAP*, que poseen la mayoría de los dispositivos electrónicos y que permite depurar en tiempo real y sobre la misma plataforma. En muchos casos, este dispositivo se conoce con el nombre de *JTAG*, *TAP controller* u *on-chip debugger* (OCD) porque permiten depurar en el mismo chip. Veremos más adelante que este tipo de dispositivos también se denominan *monitores*.

Hay diferentes fabricantes de OCD o JTAG y cada uno de ellos provee sus controladores (*drivers*) y software de depuración, que tienen que ser configurados de manera específica para cada plataforma. Encontramos conectores (*plugins*) para los entornos de desarrollo, como IAR o Eclipse, y controladores para las diferentes plataformas, como TI, AVR o ATMEL, etc.

La mayoría de las veces la herramienta de depuración utilizada es la desarrollada por GNU como depurador para Linux, denominada **gdb**. Los fabricantes de OCD desarrollan un software puente para que gdb pueda ser usado sobre los controladores del OCD específico. Por otro lado, también en muchos casos se desarrollan conectores para las plataformas, como eclipse+cdt o IAR, que facilitan la depuración y que integran funcionalidades específicas para la OCD. Estos conectores también mantienen una interfaz común con gdb.

Ejemplo de entorno de desarrollo Eclipse + CDT + Yeti2 plugin + AVR JTAG para depurar NesC y TinyOS



1.2. Herramientas de desarrollo y depuración

Hasta ahora hemos visto un conjunto de dispositivos que, combinados con software, nos permiten tener un control más cercano de nuestro desarrollo. En este apartado veremos con más detalle herramientas de desarrollo de software que conforman nuestro entorno de evaluación.

1.2.1. El entorno de desarrollo

Un entorno de desarrollo integrado[4] es un programa compuesto por un conjunto de herramientas de programación.

(4)En inglés, *integrated development environment* (IDE).

Se puede dedicar en exclusiva a un lenguaje de programación o se puede utilizar para varios lenguajes. Los IDE proveen un marco de trabajo amigable para la mayoría de los lenguajes de programación. Están compuestos, entre otros, por los elementos siguientes:

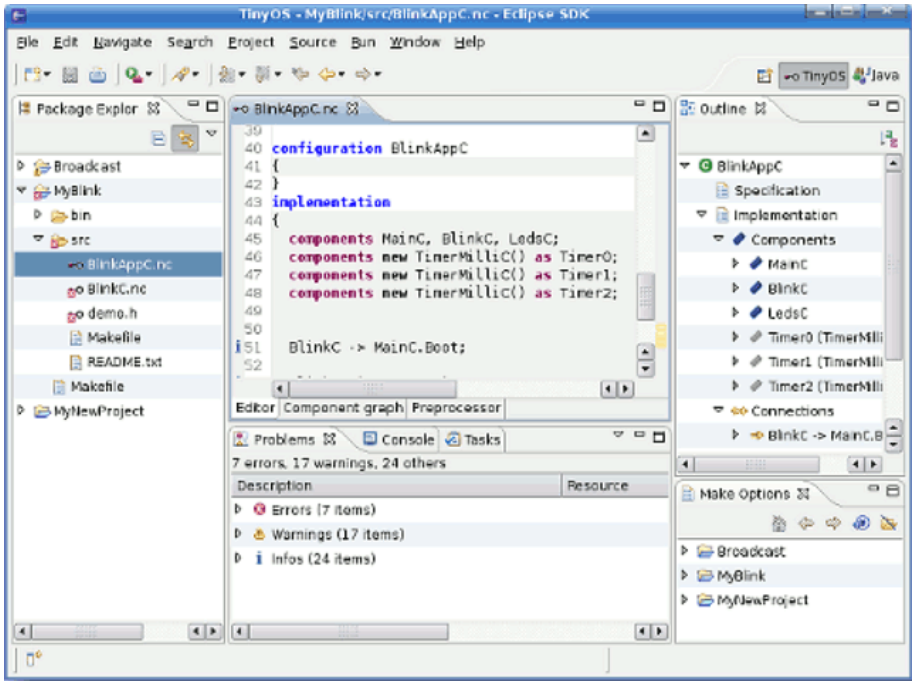
- Un editor de texto.
- Un compilador.
- Un intérprete.
- Un depurador.
- La posibilidad de ofrecer un sistema de control de versiones.
- Herramientas para ayudar a la construcción de interfaces gráficas de usuario.

Hay múltiples IDE para el desarrollo de aplicaciones para sistemas empotrados. (Aunque no todos son específicos, sí que ofrecen funcionalidades especializadas).

Nombre	Lenguaje	Licencia
Eclipse+CDT	C/C++ , NesC, otras (+ conectores)	Código abierto (*open source*)
IAR	C/C++ (msp430, etc.)	Propietario pero libre para programar con memoria < 4 kB
AVR studio	C/C++	Propietario
CodeVision	C/C++	Propietario pero libre para programar con memoria < 3 kB
CODE::BLOCKS	C/C++	Código abierto
CodeWarrior	C/C++ (*freescale*)	Propietario pero el *tool chain* es gratuito si se desarrolla sobre los microprocesadores de *freescale*
µVision IDE (Keil)	C/C++ (arm, cortex-m...)	Propietario
CCStudio	C/C++ (msp430...)	Propietario con versión gratuita con límite de 16 kB de código para MSP430

La elección de un IDE u otro viene condicionada, en muchos casos, por el compilador y el microcontrolador elegidos, puesto que cada plataforma tiene sus particularidades y en muchos casos los compiladores incluyen el IDE.

Eclipse con conectores CDT y Yeti2 para trabajar con TinyOS



1.2.2. Los depuradores

Un **depurador** (en inglés, *debugger*) es un programa que permite depurar o limpiar los errores de otro programa (programa objetivo). Hay muchos depuradores diferentes, especializados para arquitecturas diferentes o para lenguajes de programación específicos. Por el interés de este curso nos centraremos en uno de ellos.

GDB

GDB es el depurador de GNU. Se trata de un poderoso depurador que permite "ver" qué está sucediendo dentro de programas escritos en C, C++, entre otros. Entre las capacidades más notorias de este depurador se encuentran las siguientes:

- Es un depurador de programas complejos con múltiples archivos.
- Posee la capacidad para parar el programa o ejecutar una orden en un punto específico (puntos de ruptura, *breakpoints*), según una condición (*watchpoints*) o al recibir un *signal* (*catchpoints*).
- Tiene la capacidad para mostrar valores de expresiones cuando el programa se detiene automáticamente (*displays*).
- Es posible examinar la memoria o variables de varias formas y tipos, incluyendo estructuras, vectores y objetos.
- Es posible igualmente cambiar los valores de las variables para estudiar el comportamiento del programa sin necesidad de recompilar.
- Tiene la posibilidad de depurar programas en ejecución (procesos).
- Permite depurar programas que han finalizado.
- Dispone de múltiples modos de entrar al depurador.

1.2.3. Invocación del depurador

El depurador se puede ejecutar de una de las maneras siguientes:

```
$ gdb
```

```
$ gdb programa
```

para cargar el programa y entrar en el modo interactivo. El programa no empieza hasta que se indique con una orden

```
$ gdb programa core
```

para depurar un programa que ha finalizado con un núcleo (*core*). El depurador carga el programa y su entorno exactamente como acabó. Es útil para verificar por qué un programa acabó mal o para ver dónde un programa se "colgó" (usando CTRL–\ para cortar el programa y obtener un núcleo).

```
$ gdb programa pid
```

para depurar un programa en ejecución con el pid indicado. El proceso se detiene y el depurador lo controla en otro terminal. Resulta sumamente útil para depurar programas con interfaz desde otro terminal virtual. Una vez que se entra al modo interactivo, GDB acepta órdenes hasta que se le indique que se quiere salir con la orden quit.

Órdenes más habituales

Las órdenes usadas más frecuentemente son:

```
list [archivo:]función
list [archivo:]línea[,línea]
list
list &#8211;
```

para hacer una lista del código fuente a partir de una función o una línea. list solo continúa la lista previa. list – enumera las líneas anteriores.

```
break [archivo:]función
break [archivo:]línea
```

para colocar un punto de ruptura a comienzos de la función o a comienzos de la línea indicada.

```
run [argumentos]
```

para empezar la ejecución del programa desde el principio. Los argumentos son los pasados al ejecutable.

```
bt (backtrace)
```

para mostrar la *stack* del programa, indicando las funciones invocadas y en qué lugares fueron llamadas.

```
print exp.
```

para mostrar el valor de una expresión.

```
c
```

para continuar la ejecución del programa después de que haya sido detenido con un *signal* o un punto de ruptura.

```
next
```

ejecuta la próxima línea del programa sin entrar dentro de las funciones. Se puede aprovechar el hecho de que GDB repite la última orden con ENTER para ejecutar varias líneas seguidas.

`step`

ejecuta la próxima línea del programa entrante en funciones. Se puede aprovechar el hecho de que GDB repite la última orden con ENTER para ejecutar varias líneas seguidas.

`jump línea`

salta a las órdenes siguientes y empieza la ejecución a partir de *línea*. Resulta útil para continuar un programa que ha acabado con core, que arregla la situación y salta las líneas defectuosas.

`help [item]`

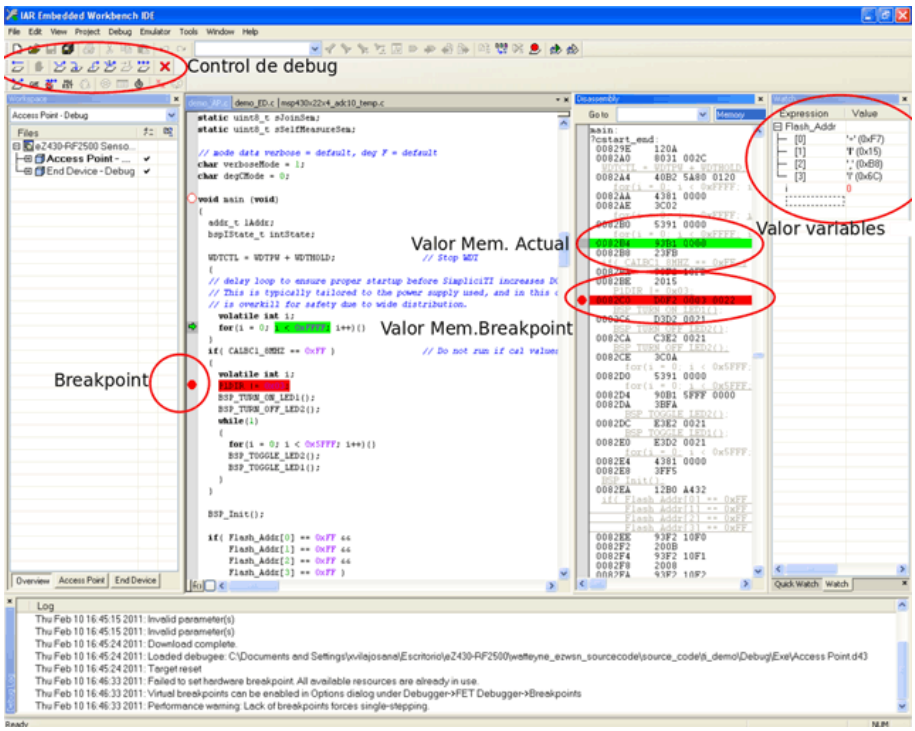
ayuda en línea.

`quit`

sale de GDB.

GDB, como hemos visto, ofrece una interfaz de órdenes que nos permite interactuar con el programa que queremos depurar. Sin embargo, hay múltiples interfaces de usuario que nos facilitan esta tarea. Muchos de los entornos de desarrollo presentados antes incorporan la interfaz gráfica para interactuar con un depurador. Por ejemplo, IAR y eclipse+CDT incorporan la interfaz para depurar el código usando GDB.

Entorno de desarrollo de IAR, en modo depurador



Como podemos ver en esta figura, el entorno de desarrollo nos facilita la tarea de depuración ofreciéndonos de manera visual las herramientas para depurar el código. En la parte superior izquierda de la imagen vemos los controles de depuración que nos permiten controlar la ejecución del programa.

De izquierda a derecha encontramos:

- **Reset**: reinicia la ejecución.

- **Step over**: salta a la instrucción siguiente. En el caso de ser una función, la trata de manera atómica y no analiza el código.

- **Step into**: salta a la instrucción siguiente. En el caso de ser una función, se posiciona a su primera instrucción.

- **Step out**: en el caso de haber entrado en una función, nos posiciona después de la última instrucción de esta función.

- **Next statement**: nos lleva siempre a la instrucción siguiente, tanto si es función como si no.

- **Run to cursor**: ejecuta hasta que llega a la posición del cursor o hasta que encuentra un punto de ruptura.

- **Run**: ejecuta hasta el final del programa o hasta que encuentra un punto de ruptura.

Es también muy útil la herramienta de inspección de valores de variable en tiempo de ejecución que podemos ver en la parte superior derecha de la figura.

A pesar de que en los últimos años las herramientas de depuración para sistemas empotrados han mejorado mucho, hay casos, ya sea por la disponibilidad de recursos, ya sea porque es un desarrollo muy especializado, en los que no podemos utilizar las herramientas introducidas hasta ahora.

1.2.4. Depuración con escritura a terminal

El uso de gdb o algún otro depurador nos permite ver paso a paso qué está sucediendo en nuestro código. Sin embargo, a veces nos interesa de manera rápida ver la traza de nuestro programa sin tener que ir paso a paso. Así pues, hay herramientas que nos permiten, mediante el puerto serie (o USB), que el dispositivo pueda escribir mensajes de traza en nuestro terminal. Esta forma de depuración, conocida popularmente con el nombre de **depuración con *printf***, no se recomienda como única vía de depuración de un programa, pero sí como herramienta complementaria a la depuración.

En muchos casos, esta herramienta se ofrece en formato de biblioteca por el lenguaje de programación del sistema empotrado, aunque no en todos los casos. Plataformas como las de Texas Instruments basadas en *simplicity* nos lo permiten, pero siempre mediante la escritura de *byte* por *byte* (o carácter por carácter) por medio de la UART. TinyOS, a pesar de que no de manera nativa, nos ofrece una biblioteca que nos permite hacer este tipo de depuración.

Enlace recomendado

Podéis encontrar más información sobre cómo se debe utilizar la biblioteca *printf* en TinyOS y NesC en: http://docs.tinyos.net/index.php/the_TinyOS_printf_Library_%28TOS_2.1.1%29

Ejemplo

En el lenguaje nesC para el sistema operativo TinyOS tendríamos el código siguiente:

```
event void Boot.booted() {
       printf("¡Iniciamos nuestro programa!\n");
       printf("El valor de la variable es: %u \n", var);
       printfflush();
 }
```

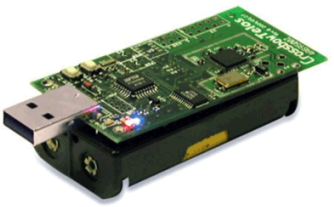
la salida sería:

¡Iniciamos nuestro programa!

El valor de la variable es: 0

1.2.5. Depuración vía LED

A pesar de que es una manera de depurar, el uso de los **LED** como mecanismo de comunicación entre dispositivo y programador no es recomendable si no es como complemento de alguna otra técnica. Básicamente, el programador establece una codificación ad hoc sobre las combinaciones posibles de LED encendidos y apagados, o la combinación de sus colores, si procede, que le permiten detectar ciertos estados del dispositivo o situaciones de error. Cabe señalar que, a pesar de que la depuración con LED no es demasiado ortodoxa, sí que lo es el uso de LED para señalizar estados del dispositivo una vez la aplicación ha sido del todo verificada. Normalmente es útil definir una codificación de LED para advertir al usuario de ciertos estados, como la inicialización, la carencia de batería, que está fuera del área de cobertura, la disfunción, el funcionamiento correcto, etc.



TelosB de Crossbow con dos LED

El uso de LED está extendido en la mayoría de los sistemas empotrados de baja capacidad, como son las redes de sensores inalámbricos, puesto que no disponen de interfaces más complejas para la visualización de los estados del dispositivo.

2. Simuladores

Cualquier diseño de un programa de propósito específico requiere una simulación, que consiste en intentar replicar las condiciones de aquel programa en un entorno controlado (de este modo, quizá no necesitemos ni siquiera el hardware de aquel entorno y, simplemente, necesitamos un entorno en software). Hay unas cuantas razones para recurrir a la simulación:

- Permiten a los equipos de trabajo de software y hardware trabajar con un cierto grado de independencia, encabalgando tareas que de otro modo habría que ejecutar secuencialmente.

- Ayudan a explorar la arquitectura, de manera que permiten comparar diferentes opciones de diseño sin comprometerse con implementaciones enteras.

- Permiten probar componentes independientes, antes de que otras partes del sistema estén disponibles.

- Permiten también conectar diferentes piezas del sistema para probarlas conjuntamente, para depurar la interacción, cuando solo una parte del sistema está disponible o cuando les queremos proporcionar un entorno de test controlado.

- Pueden ser usados para validar o diagnosticar las prestaciones en todas las etapas de diseño.

Actualmente hay una gran variedad de técnicas de simulación, tanto orientadas a software como hardware. La más flexible, pero también lenta, es la simulación "interpretada", que consiste en simular por software el comportamiento del sistema.

Como veremos en este subapartado, también hay simuladores más ligados al hardware del sistema de propósito específico, como pueden ser programas monitores y emuladores. Cada técnica de simulación tiene ventajas e inconvenientes frente a las otras.

2.1. Programas simuladores

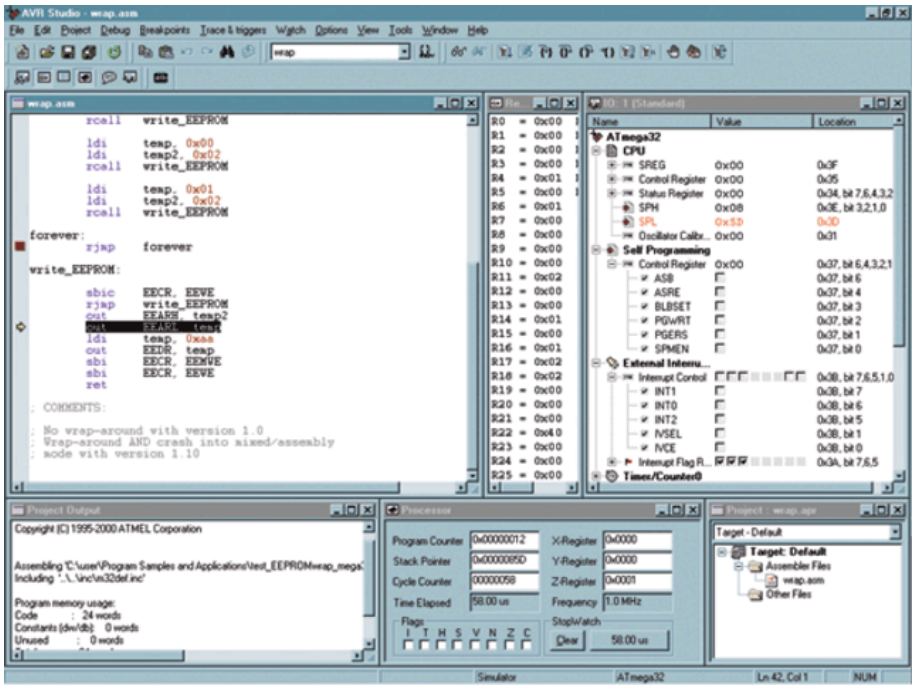
Un simulador ejecuta el programa del microcontrolador en una máquina huésped (como un PC).

Se puede inspeccionar el código mientras se está ejecutando paso a paso para ver exactamente qué está sucediendo mientras el programa corre. Los contenidos de los registros o las variables pueden ser alterados para cambiar el modo como funciona el programa.

Este tipo de herramienta elimina (o al menos retrasa) el ciclo necesario con una EPROM de borrado/grabación/programación común en el desarrollo de programas para un microcontrolador.

Un simulador no puede soportar interrupciones ni dispositivos reales y, normalmente, corre mucho más lentamente que el dispositivo de propósito específico real que quiere simular. A pesar de que los simuladores tienen algunas desventajas, son muy útiles en las primeras etapas de desarrollo de un proyecto de software cuando todavía no hay ningún hardware real con el que experimentar.

AVR studio con funcionalidad de simulador de un microprocesador de Atmel



La principal gran desventaja de un programa simulador es que, en principio, solo simula el microprocesador. Y, como sabemos, un sistema de propósito específico contiene unos cuantos periféricos importantes. Esto es así porque el fabricante del microprocesador es quien suele proporcionar el entorno de simulación y, por lo tanto, normalmente solo incluye el microprocesador.

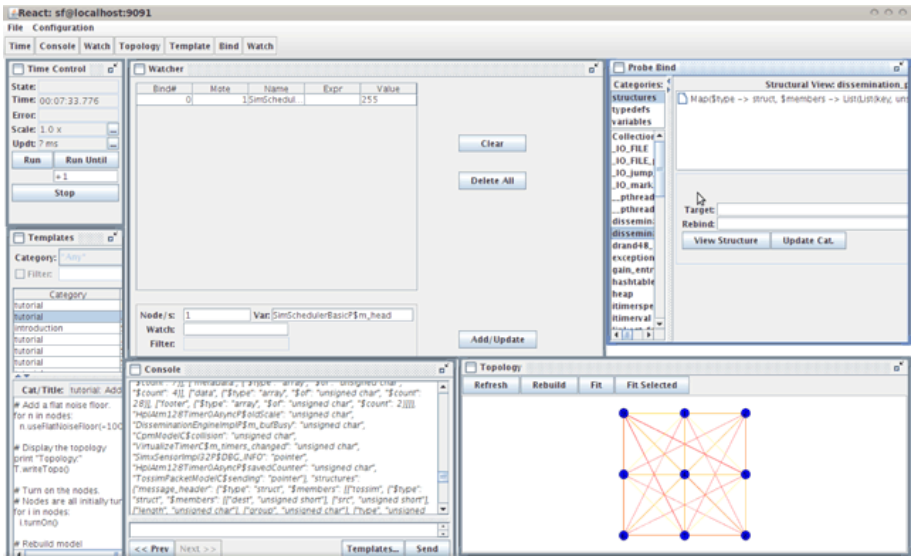
Hay simuladores para muchas plataformas; entre otros, TOSSIM o SIMX son simuladores para TinyOS que nos permiten evaluar el código antes de ejecutarlo en los dispositivos. El problema de los simuladores como TOSSIM es que resulta imprescindible que alguien haya desarrollado el código del simulador para la plataforma que queremos evaluar. En el caso de que seamos nosotros

Enlace recomendado

Para más información sobre TOSSIM, podéis consultar el enlace siguiente: http://docs.tinyos.net/index.php/tossim

los que desarrollemos el hardware y este no sea idéntico al de una plataforma ya existente, también deberemos desarrollar el módulo de TOSSIM específico para nuestra plataforma.

Simulador SimX



2.2. Programas monitores

A diferencia de un simulador, un programa monitor corre en el mismo microcontrolador y, a la vez, muestra su progreso en la máquina huésped (normalmente un PC). Esto se conoce como un *programa residente*.

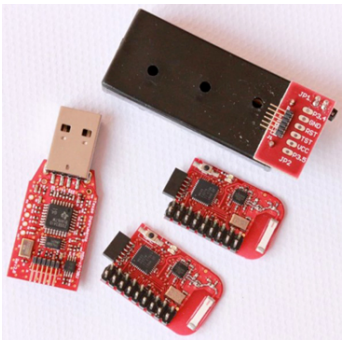
Tiene muchas de las ventajas de un simulador, con el beneficio adicional de ver cómo el programa corre en la máquina de destino real. El residente necesita ocupar algunos de los recursos del sistema de propósito específico (puesto que corre en él), que incluyen:

- Un puerto de comunicaciones para comunicarse con el huésped.
- Una interrupción para manejar la ejecución paso a paso.
- Cierta cantidad de memoria para la parte residente del monitor.

Un programa monitor permite al usuario examinar la memoria, los registros y los puertos de entrada/salida. Muchos programas monitores tienen algunas capacidades básicas en común, como son las habilidades para llevar a cabo las acciones siguientes:

- Examinar y alterar la memoria.
- Leer y escribir de los puertos de entrada/salida.
- Establecer puntos de interrupción en el código.
- Descargar código desde un PC huésped.
- Interrumpir la ejecución de código desde el teclado.
- Comunicarse vía un puerto serie rs-232 (normalmente).

Para utilizar el programa monitor, el usuario establece un punto de interrupción en algún lugar útil del código, corre el programa y entonces examina la memoria y los registros cuando se llega al punto de interrupción. Estos puntos de interrupción también se pueden introducir en las rutinas de error para ver si aquel error sucede.



ez430-rf2500 de Texas Instruments con conector para puerto USB y portapilas

El usuario descarga código a la RAM del sistema de destino y lo ejecuta desde allí. Esta comunicación se establece por un puerto serie, o quizá por una conexión de red. La interfaz con el usuario del PC tiene el aspecto de cualquier monitor utilizado para programar un PC (permite ver el código fuente, los registros, etc.), pero en nuestro caso una parte de este monitor es residente en el sistema de propósito específico.

El monitor, de hecho, consiste en dos piezas de software y un hardware que sigue la especificación JTAG vista anteriormente. La interfaz con el usuario corre en el PC, mientras que el monitor residente corre en el procesador del sistema de propósito específico. Hay algunos tipos de comunicación entre los dos.

Los monitores son una de las herramientas más comunes de descarga y test utilizadas durante el desarrollo de software de propósito específico. Esto se debe, principalmente, a su bajo coste.

Los programadores ya disponen de un PC, mientras que el precio de la interfaz (JTAG) no añade mucho coste al conjunto de herramientas de desarrollo (compilador, editor de código, etc.). Finalmente, los fabricantes de monitores residentes normalmente están dispuestos a distribuir gratuitamente el código fuente para sus monitores e incrementar así el número de usuarios.

2.3. Emuladores

Proporciona un control absoluto sobre el sistema de destino, pero no requiere ningún recurso de este. El emulador puede ser un dispositivo independiente (como en la figura que ilustra este subapartado), incluso con su propia pantalla, o puede tener una interfaz con el usuario mediante PC.

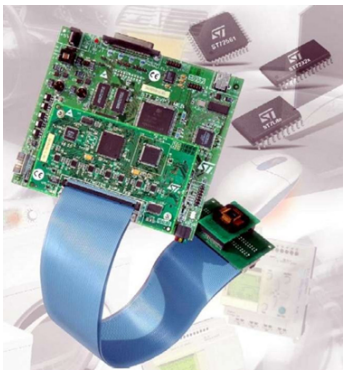


Imagen de circuito emulador de STMicroelectronics para su procesador ST72561

Los monitores son muy útiles para controlar el estado del software de propósito específico, pero solo un emulador permite examinar el estado del procesador en el que el programa está corriendo. De hecho, un emulador ocupa realmente el lugar del procesador.

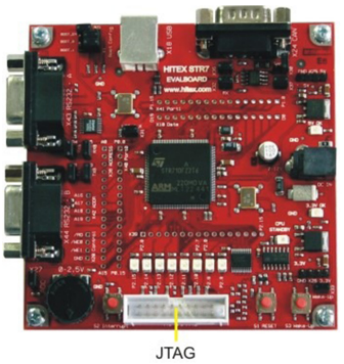
Un emulador[5] es un sistema de propósito específico en sí mismo, con su propia copia del procesador de destino, RAM, ROM y su propio software de propósito específico.

(5)Los emuladores son conocidos también con el término *placa de evaluación* o *evaluation board*.

Como consecuencia de estas características, un emulador resulta normalmente costoso, más incluso que el hardware examinado. Pero se trata de una herramienta muy potente y que puede significar una gran ayuda durante el desarrollo.

Tal como hemos visto que hace un monitor, un emulador suele utilizar un programa corriendo en un PC como interfaz con el usuario. A veces, incluso es posible utilizar la misma interfaz en los dos casos. Pero gracias al hecho de que el emulador contiene su copia del procesador que se quiere programar, es posible controlar el estado de este procesador en tiempo real.

Esto permite al emulador soportar características avanzadas de depuración, como puntos de interrupción por hardware y seguimiento en tiempo real, además de las ventajas de cualquier monitor.



Placa de evaluación de STR7 con el puerto para JTAG

2.3.1. Puntos de interrupción para hardware

Con un monitor se pueden establecer puntos de interrupción del programa. Ahora bien, estos puntos son *software*, en el sentido de que están restringidos a interrupciones por instrucción. Son el equivalente a "detiene la ejecución si esta instrucción está a punto de ser ejecutada".

Los emuladores, en cambio, también soportan puntos de interrupción por hardware. Estos puntos permiten parar la ejecución en respuesta a un amplio abanico de eventos. Estos eventos pueden ser tanto instrucciones como lecturas y escrituras de memoria, y también cualquier tipo de interrupciones del procesador.

2.3.2. Seguimiento en tiempo real

Otra característica útil de un emulador es el seguimiento en tiempo real. Típicamente, un emulador incorpora un gran bloque de RAM de propósito específico, dedicada a almacenar información sobre cada uno de los ciclos de procesador que se ejecutan. Esto nos permite ver exactamente en qué orden están pasando las cosas y, por lo tanto, nos ayuda a responder a muchas preguntas, como si una interrupción ocurrió antes o después de que una variable tomara un cierto valor.

Además, es posible restringir la información de seguimiento que se almacena o posprocesarla antes de inspeccionarla para reducir el volumen.

Una vez tenemos acceso al hardware, los analizadores lógicos y los osciloscopios pueden ser herramientas de depuración indispensables.

Resultan más útiles para depurar las interacciones entre el procesador y otros chips del sistema (de la placa electrónica). Como estos instrumentos solo pueden observar señales externas al procesador, no pueden controlar el flujo de

ejecución del programa, tal como hacen un monitor o un emulador. Esto provoca que estas herramientas sean menos valiosas por sí mismas. Pero combinadas con un monitor o un emulador, pueden tener una valía muy importante.

Cualquier subconjunto de entradas que queramos seleccionar se puede visualizar en pantalla con un eje de tiempo. Muchos analizadores lógicos también permiten empezar a capturar datos a partir de un patrón específico.

La mayoría de las herramientas de depuración que habéis estudiado en este subapartado serán utilizadas en un punto u otro de todos los proyectos de propósito específico. Los osciloscopios y analizadores lógicos se utilizan más frecuentemente para depurar errores de hardware; los simuladores, durante las primeras etapas del desarrollo del software, y los monitores y emuladores, durante la depuración del software real. Para ser más efectivos, deberíais entender la utilización de cada herramienta y cuándo y dónde la tenéis que aplicar para obtener los beneficios más elevados.

3. Metodologías de verificación y test

En esta sección se presentan las técnicas más habituales de verificación y evaluación de programas. Estas técnicas son presentadas de una manera genérica y en ningún caso se pueden considerar exclusivas para el entorno de los sistemas empotrados. En cambio, son metodologías utilizadas ampliamente en la verificación de todo tipo de software.

La evaluación del software es una parte esencial en el desarrollo. Si el software que desarrollamos no funciona, no será usado (y probablemente no nos pagarán). Sin embargo, antes de adentrarnos en el mundo de verificación de software es importante recordar a los usuarios de nuestro programa lo siguiente.

- Los **usuarios** ven el software desde fuera. No evalúan los algoritmos ni la estructura del código. El sistema para ellos es una **caja negra**. Solo les interesa la funcionalidad.

- Los **evaluadores** (*testers*) comprueban que los resultados que da el software sean los esperados. Les interesa la funcionalidad, pero también esperan que el software haga exactamente aquello que tiene que hacer. Mirarán que los datos son correctos, que los puertos sacan la información que tienen que sacar, que la memoria es liberada correctamente, sin entrar en detalles de los algoritmos o particularidades del código. Así, los evaluadores ven nuestro software como una **caja gris**.

- Los **desarrolladores** ven la arquitectura del programa, la estructura del código, los estados, los patrones. El código está abierto a ellos y ven el programa como una **caja blanca**. No obstante, en ocasiones una visión tan cercana les hace perder la perspectiva para ver funcionalidades mal diseñadas o con alguna carencia que, en cambio, los evaluadores o usuarios pueden detectar.

Así pues, una buena evaluación de software debe tener en cuenta las tres perspectivas que se exponen a continuación.

3.1. Evaluación de la caja negra

Los usuarios están fuera del sistema. Ellos solo ven las entradas y salidas del sistema. De este modo, cuando se evalúa la caja negra se deben buscar:

- **Funcionalidades**. Esta es la prueba más importante. ¿El sistema hace lo que tiene que hacer? No son importantes los detalles de cómo lo hace, o

cómo son las salidas: solo importa saber si la funcionalidad del sistema es la especificada.

- **Validación de las entradas**. Se ha de verificar que las entradas del software están protegidas a valores incorrectos o formatos no soportados. ¿Qué sucede si esperamos un valor positivo y le introducimos un valor negativo?

- **Validación de las salidas**. Se debe verificar el resultado manualmente, si procede. Se han de verificar todos los caminos posibles. Es muy útil hacer una tabla que especifique todas las entradas y todas las posibles salidas que se pueden dar. Con esta tabla se puede verificar el funcionamiento del programa.

- **Transiciones de estado**. Algunos sistemas tienen diferentes estados. Se debe verificar que el sistema pasa por los estados que tiene que pasar. Si hay diferentes caminos, se ha de verificar la corrección de todos ellos. Esto es particularmente importante en sistemas en tiempo real o en la implementación de protocolos de comunicación. En este proceso es muy útil construir un diagrama de estados y verificar que se cumplen todas las transiciones.

- **Casos extremos y casos no válidos pero en el umbral**. Se deben verificar los casos extremos. Tanto los valores que entran en los umbrales como los que están justo por encima.

Representación de la técnica de evaluación de caja negra. Solo vemos entradas y salidas

Una especificación simple de caja negra

Nota

Es común planificar la evaluación usando tablas en las que se especifican tanto las entradas como las salidas esperadas.

3.2. Evaluación de la caja gris

La evaluación con la caja negra en muchos casos es suficiente para validar un programa. Sin embargo, hay situaciones en las que se hace necesaria una verificación más profunda. En ocasiones, no se pueden obtener los resultados de un programa desde fuera, sino que o bien porque el programa es parte de un sistema más grande, o bien porque su funcionalidad no se expresa en una salida explícita, hay que mirar dentro del programa.

Esto es particularmente cierto en programas para sistemas empotrados que en muchos casos solo mueven datos de la memoria del dispositivo.

Cuando se utiliza la técnica de caja gris normalmente se lleva a cabo el mismo tipo de verificaciones, pero esta vez verificando algunos de los estados dentro del programa.

- **Verificación, auditoría y log**. Se hace uso de los logs del programa o trazas de memoria para ver qué ha sucedido.

- **Datos para otros programas**. Si el software que se está evaluando es un subsistema o biblioteca, se debe verificar que las salidas son las esperadas para los programas que lo usarán.

- **Información del sistema**. Cuando nuestro programa hace escrituras en disco, o usa relojes internos del sistema, se ha de verificar que las marcas horarias (*timestamps*), las sumas de verificación (*checksums*) y otros datos que genera el propio sistema son correctos y están en el formato deseado.

- **Revisión del estado de la memoria**. Aunque nuestro programa funcione, es posible que no hayamos hecho un uso adecuado de la memoria o que hayamos dejado variables sin usar o cadenas (*threads*) en estado zombi que posteriormente pueden afectar al funcionamiento de otras bibliotecas.

Proceso zombi

Un proceso zombi (en inglés, *zombie process* o *defunct process*) es un proceso que ha completado su ejecución pero que todavía tiene una entrada en la tabla de procesos. Esta entrada es todavía necesaria para permitir que el proceso que lo puso en marcha (proceso padre) obtenga el estado de finalización. En caso de que el proceso padre haya acabado, no se podrá eliminar esta entrada en la tabla de procesos y dejar la memoria ocupada. De hecho, el término *proceso zombi* deriva de la definición común de *zombi* (una persona no muerta). Siguiendo con la metáfora, el proceso hijo ha muerto, pero no ha sido enterrado.

3.3. Evaluación de la caja blanca

Esta evaluación es la más profunda. Este proceso de evaluación observará qué sucede exactamente dentro del código y se hará lo posible por lograr que el código falle. La evaluación de caja blanca puede ser un reto para el evaluador, puesto que su objetivo es hacer que el código falle. Para realizar este tipo de evaluación, hay que estar familiarizado con el código y conocer las precondiciones y poscondiciones impuestas a cada método.

Normalmente, se busca:

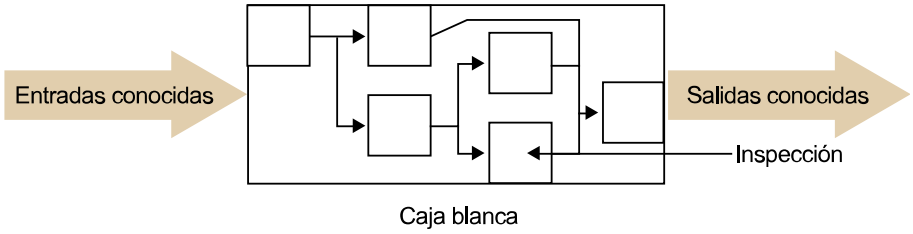
- **Verificar todos los caminos del código**. Para cada `if/else` o `switch/case` se tiene que verificar cada una de las ramas del código, verificar cuáles son los datos que nos permitirán ir por cada una de las ramas y ver si nuestro programa es capaz de generarlas.

- **Gestión de los errores**. Si en un método o función se genera un error, ¿este es gestionado? Se debe verificar que los errores no supongan estados inconsistentes ni que modifiquen de manera inconsistente los datos de memoria.

- **Gestión de la memoria**. En el caso de errores, ¿se liberan los recursos ocupados? En general, cuando acaba la ejecución de un método, ¿se libera la memoria que no ha de ser usada?

- **Gestión de recursos**. ¿Qué hace nuestro programa cuando se quiere acceder a un recurso que está ocupado?

Evaluación en caja blanca, hemos de estudiar el código por dentro



Caja blanca

Resumen

Este módulo nos ha presentado las principales herramientas que conforman el entorno de desarrollo para trabajar con sistemas empotrados. Nos ha mostrado las herramientas de hardware más usadas para evaluar y depurar nuestros programas. Hemos visto que los osciloscopios se emplean para ver las señales eléctricas que circulan por el software durante la ejecución de nuestro programa. A pesar de que son útiles, cuando se quieren estudiar varias entradas y salidas del hardware se utilizan los analizadores lógicos, que permiten observar un número más elevado de pines. Cuando lo que se quiere evaluar es el comportamiento de señales de radiofrecuencia generadas por el hardware, se utilizan los analizadores de espectro o analizadores de red, que nos permiten ver de manera frecuencial el comportamiento de nuestro programa. Hemos visto, también, que una herramienta muy útil es el monitor que usa la especificación del JTAG, una herramienta que nos permite depurar el código sobre la plataforma software real. En cuanto a software, se han introducido los entornos de desarrollo integrados (IDE) como herramientas que, de manera amigable, nos permiten desarrollar código, depurarlo y que simplifican mucho la tarea del programador. Sin embargo, hemos comprobado que la elección de la IDE está condicionada por la plataforma software y los compiladores elegidos y que en muchos casos están sujetos a licencias propietarias. Los depuradores son herramientas imprescindibles para asegurar el funcionamiento correcto de nuestras aplicaciones. Se han presentado las principales características y funcionalidades de los depuradores poniendo como ejemplo el depurador GDB de gnu, que está incluido en casi todas las plataformas *nix. También hemos visto que los IDE incluyen interfaces que hacen más amigable el proceso de depuración y que en muchos casos incluyen herramientas que nos permiten ver el estado de la memoria en tiempo real. Esto es fundamental en los sistemas empotrados, puesto que no disponemos de salidas explícitas que nos permitan evaluar el funcionamiento correcto del programa.

Seguidamente, el módulo nos ha presentado varias técnicas de simulación, muy ligadas a la depuración y que nos sirven para verificar el funcionamiento correcto de un sistema empotrado a medida que se va desarrollando. Tanto es así que en muchos casos los desarrollos no se hacen sobre el software definitivo, sino que se utilizan placas de evaluación o desarrollo que incluyen funcionalidades orientadas a la depuración. Una vez se ha desarrollado el programa y se está seguro de que funciona correctamente, se desarrolla el hardware recortando las funcionalidades que la placa de desarrollo ofrecía y que ya no son necesarias.

Finalmente, se nos ha presentado la parte más importante de este módulo, que es la metodología de depuración. Esta metodología es genérica y sirve para el desarrollo de software, en general, pero resulta completamente aplicable a los

sistemas empotrados. Hemos visto la depuración en caja negra, orientada a lo que verán los usuarios; en caja gris, que nos permite verificar la corrección de los datos durante el proceso de ejecución de nuestro programa, y, finalmente, la depuración en caja blanca, que evalúa los posibles estados de nuestro programa.

Bibliografía

Autores varios. "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications". En: *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*.

Autores varios. "TOSSIM: A Simulator for TinyOS Networks". *User's manual, in TinyOS documentation*.

Catsoulis, J. (2005). *Designing Embedded Hardware*. Sebastopol, California: O'Reilly and Associates.

IEEE Std 1149.1-1990 – Test Access Port y JTAG Architecture, y el Std 1149.1-1994b – complemento de IEEE Std 1149.1-1990, están disponibles en IEEE Inc., 345 East 47th Street, Nueva York. También se puede obtener una copia del estándar IEEE 1149.1 en línea: http://www.ieee.com/

Peckol, J. K. (2008). *Embedded Systems: A Contemporary Design Tool*. Hoboken, Nueva Jersey: Wiley.

Pilone, D.; Miles, R. (2008). *Head First Software Development*. Sebastopol, California: O'Reilly Media.

Stallman, R.; Shebs, R. P. S. (2009). *Debugging with gdb* (9.ª ed.). Boston, Massachusetts: Free Software Foundation.

TinyOS. Disponible en línea: http://www.tinyos.net/

Yaghmour, K. (2003). *Building Embedded Linux Systems*. Sebastopol, California: O'Reilly and Associates.

Anexo

1) Guía de uso de GDB

Con las órdenes mencionadas anteriormente, es posible depurar programas fácilmente y con buen control. Esta sección pretende mostrar brevemente cómo se puede empezar a usar gdb con solo estas órdenes. Para poder utilizar un depurador en UNIX, hay que compilar el programa con la opción –g. Por ejemplo:

```
cc -g prueba.c -o prueba
```

a. Ejecutar el programa desde el depurador. Método recomendado en las primeras fases de prueba. Se obtiene un control completo del programa y, si este falla, se visualiza inmediatamente dónde ha fallado. Es posible parar el programa en cualquier momento con CTRL + C y volver al depurador, lo cual permite verificar bucles infinitos, etc.

```
$ gdb prueba
(gdb) run [argumentos]
...
CTRL+C
(gdb)
```

Una vez detectadas funciones en las que puede haber problemas:

```
$ gdb prueba
(gdb) list función
...
(gdb) break línea
(gdb) run [argumentos]
...
break
(gdb) print expr
(gdb) next
...
(gdb) c
...
```

b. Determinar dónde un programa acaba con *core*.

```
$ gdb programa core
#0 main () at prueba.c:100
100     *(char *)0 = 10;
```

```
(gdb) bt
...
```

Una vez dentro de gdb, se pueden examinar otras variables y hacer un *backtrace* para verificar el camino que sigue el programa para producir la excepción. Es posible arreglar la situación y ejecutar un *jump* para continuar ignorando el error (podéis ver estas órdenes avanzadas):

```
(gdb) arreglar la situación
(gdb) jump línea                        ejecutar ignorando el error
```

c. Programa ejecutante. Es posible depurar un programa en ejecución. Para ello, hay que hacer:

```
$ ps                      para determinar el pid
$ gdb programa pid        para interceptar el programa
(gdb)                     en este momento el programa se detiene
...
```

2) Guía de uso de GDB

Los puntos de ruptura son puntos en los que el programa se detiene cuando pasa por ellos. *Watchpoints* son expresiones que detienen el programa cuando el valor cambia. *Catchpoints* son puntos de ruptura sobre *signals*. Las órdenes son como siguen:

```
break [archivo:]función
break [archivo:]línea
```

para colocar un punto de ruptura a comienzos de la función o a comienzos de la línea indicada.

```
tbreak [archivo:]línea
tbreak [archivo:]función
```

igual que *break*, pero el punto de ruptura es válido una sola vez. Útil para crear puntos de ruptura temporales.

```
watch exp
```

se habilita un *watchpoint* cuando la expresión `<expr>` cambia.

```
catch
```

se colocan puntos de ruptura en todos los gestores (*handlers*) de excepciones del contexto actual.

```
info break
info watch
```

muestra los *watchpoints* o puntos de ruptura habilitados.

```
info match
```

indica si se están interceptando las excepciones.

```
clear línea
clear función
```

para eliminar un punto de ruptura de la línea indicada. Podéis ver delete para eliminar puntos de ruptura por número.

```
delete numero
```

para eliminar un punto de ruptura por número. El número se puede ver con info break.

```
disable breakpoint
enable  breakpoint
```

para habilitar o deshabilitar temporalmente un punto de ruptura. A diferencia de delete, no se pierde la referencia de la línea en la que se encuentra, simplemente es ignorado.

```
condition breakpoint [expr]
```

para hacer que un punto de ruptura sea condicional. Es decir, solo se habilita si la expresión `<expr>` es cierta. Como `<breakpoint>` se debe pasar el número del punto de ruptura. Si `<expr>` no se especifica, se hace el punto de ruptura incondicional.

```
ignore breakpoint [count]
```

ignora `<count>`pasadas sobre el punto de ruptura `<breakpoint>`.

3) Examinando y cambiando los datos

GDB ofrece órdenes para manipular los datos, entre las cuales se encuentran:

```
whatis variable
```

indica de qué tipo es la variable.

```
ptype tipo
```

imprime la definición del tipo indicado.

```
print [/fmt] expr
```

imprime la expresión. /fmt es un indicador de formato. Para print el formato solo puede estar compuesto por una letra de cambio de tipo.

```
set variable=expresión
```

cambia el valor de una variable al resultado de la expresión.

```
display [/fmt] exp.
```

habilita un *display* continuo durante la depuración de la expresión indicada. Cada vez que el programa se para, se hace un print [/fmt] de la expresión. Se pueden tener varios *displays* a la vez. Con `delete` se pueden eliminar `displays` y con `enable` y `disable` se pueden habilitar y deshabilitar igual que si fueran puntos de ruptura.

```
undisplay numero
```

equivalente a delete de un *display*: destruye un *display*.

```
x [/fmt] address
```

examinar memoria. /fmt es un indicador opcional formado por '/', seguido de un número (contador), seguido de una letra de formato, seguido de una letra de tamaño. Las letras de formato son:

```
                o       octal                   f       float
                x       hexadecimal             a address
                d       decimal                 o unsigned       decimal
                t       binary                  s       string
                c       char
las letras de tamaño son:
                b       byte                    h       halfword
                w       word                    g       giant (8 bytes)
```

El contador indica cuántos elementos hay que imprimir, así:

```
                x /10xb vector
```

imprime los 10 bytes siguientes del vector en hexadecimal.

4) Manipulando la *stack*.

Las órdenes para manipular la *stack* permiten cambiar o examinar variables que se encuentran en otros contextos diferentes al local. Es posible verificar el contenido de todas las variables automáticas en la stack. Las órdenes son:

```
                        frame [N]
```

selecciona el *frame N* y lo imprime. Sin argumento la orden indica dónde está actualmente (por ejemplo, *frame* actual).

```
          bt [N]
```

para ver el contenido de la *stack*. Si se especifica un número positivo, se ven las *N* primeras entradas en la *stack* y si es un número negativo, se ven las últimas *N*.

```
          select-frame #
```

se selecciona el *frame number* indicado (el número lo da la orden bt). Al cambiar de *frame* permite ver o cambiar las variables sobre la *stack*. No desapila ni apila ningún nuevo *frame*.

```
          up
```

para ir al *frame* inmediatamente superior (es decir, la rutina que llama al actual).

```
          down
```

para ir al *frame* inmediatamente inferior.

```
   return
```

para forzar el desapilamiento del *frame* actual.

5) Órdenes para impresión de información de estado

Las órdenes siguientes imprimen información variada de estado del depurador y del programa depurado:

```
          info filas
```

muestra los archivos y procesos que se están depurando.

```
          info program
```

estado del programa en el momento.

```
info sources
```

muestra los archivos fuente en depuración.

```
info types
```

muestra todos los tipos definidos.

```
info variables
```

muestra todas las variables globales definidas.

```
info functions
```

muestra todas las funciones definidas.

```
info display
```

muestra todas las expresiones *display* en efecto.

```
info breakpoints
```

muestra todos los puntos de ruptura en efecto.

```
info watchpoints
```

muestra todos los *watchpoints* en efecto.

```
info args
```

muestra los argumentos del *frame* actual.

```
info locales
```

muestra las variables locales del *frame* actual.

6) Otras órdenes útiles

Otras órdenes de GDB que son útiles a la hora de depurar:

```
hile
```

para cargar un nuevo ejecutable y tabla de símbolos dentro del depurador.

```
cd
```

para cambiar de directorio.

```
pwd
```

para ver el directorio actual.

```
shell
```

para salir a un *subshell*.

```
search reg-expr
```

para buscar en el font una expresión regular a partir de la línea actual hacia abajo.

```
reverse-search reg-expr
```

para buscar en el font una expresión regular a partir de la línea actual hacia arriba.

```
make
```

para correr el programa make.