Simulació i test

Xavier Vilajosana Guillén

PID_00177254

Universitat Oberta de Catalunya
www.uoc.edu

Els textos i imatges publicats en aquesta obra estan subjectes –llevat que s'indiqui el contrari– a una llicència de Reconeixement-Compartir igual (BY-SA) v.3.0 Espanya de Creative Commons. Podeu modificar l'obra, reproduirla, distribuir-la o comunicar-la públicament sempre que en citeu l'autor i la font (FUOC. Fundació per a la Universitat Oberta de Catalunya), i sempre que l'obra derivada quedi subjecta a la mateixa llicència que el material original. La llicència completa es pot consultar a http://creativecommons.org/licenses/by-sa/3.0/es/legalcode.ca

Índex

Introducció

Un dels aspectes més importants i a la vegada més oblidats en el desenvolupament d'aplicacions és el que fa referència al procés de test i avaluació del programari resultant. En els sistemes encastats, el procés de verificació i test encara pren més importància, ateses les característiques d'aquests sistemes, que fa difícil, en molts casos, la interacció i el seguiment del codi en execució. La importància de verificar el funcionament correcte d'un programa és cabdal, ja que en la majoria dels casos els sistemes encastats i de propòsit específic estan concebuts per a funcionar d'una manera autònoma, permanent i desatesa.

En aquest mòdul, presentarem les eines més utilitzades en el procés de desenvolupament de programari per a sistemes encastats. D'una manera específica, presentarem les eines més utilitzades i en detallarem el funcionament i l'ús en la cadena de desenvolupament. No obstant això, en el mòdul també presentarem metodologies d'avaluació que permeten d'una manera sistemàtica fer tests i avaluar programari en general. Així, els conceptes aquí introduïts són del tot vàlids per a l'avaluació del programari en general.

El mòdul ens presenta les principals eines o els dispositius de maquinari que, en general, són utilitzats durant el desenvolupament de programari per a sistemes encastats. Entre d'altres, els coneguts oscil·loscopis o els analitzadors d'espectres que ens permeten veure a nivell de senyal el que passa en el nostre dispositiu. Tot seguit, el mòdul ens presenta les eines de programari que conformen el nostre entorn de treball, entre les quals hi ha l'eina principal per a avaluar tot desenvolupament de programari, el depurador.

Seguint amb la presentació d'eines, s'introdueixen els simuladors, un altre tipus d'eina maquinari i programari que és molt útil durant el procés de desenvolupament. Trobem diferents tipus de simuladors, des de plaques de desenvolupament d'una arquitectura específica fins a programes que simulen el comportament d'una arquitectura determinada.

Finalment, el mòdul ens presenta, des d'un punt de vista més genèric, una metodologia d'avaluació del programari. Aquesta metodologia és usada per a verificar programes en qualsevol àmbit i no específicament per a sistemes encastats. No obstant això, l'ús d'una metodologia de verificació és imprescindible per a garantir la qualitat dels nostres desenvolupaments.

Objectius

L'estudi d'aquests materials us ajudarà a assolir els objectius següents:

1. Conèixer les eines de desenvolupament més habituals, i també la seva configuració o adaptació.

2. Conèixer les tècniques més usades per a depurar sistemes encastats.

3. Conèixer les tècniques més rellevants de simulació en sistemes encastats.

4. Conèixer les tècniques de test més útils.

1. Eines de desenvolupament

Per tal de treballar en el desenvolupament de programari per a sistemes encastats, es fa del tot necessari tenir un seguit d'eines que conformen l'entorn de desenvolupament[1]. L'entorn de desenvolupament va en molts casos lligat a la plataforma de maquinari i programari escollida i, en molts casos, és proveït pel mateix fabricant del maquinari. Tant és així que, en molts casos, quan se selecciona un microcontrolador, aquesta selecció no es fa tant per les propietats o per les característiques del xip com per les eines i pel programari que facilita el fabricant.

(1)En anglès, *tool chain.*

No obstant això, no totes les eines necessàries es troben en l'entorn de desenvolupament. Calen, entre d'altres, eines de laboratori que ens permetin fer una anàlisi més profunda del vessant electrònic dels dispositius. En aquesta secció farem un repàs de les eines principals.

1.1. Dispositius usats per a la programació i per al desenvolupament

En aquest apartat, presentem els principals dispositius que constitueixen l'entorn de desenvolupament per a sistemes encastats o de propòsit específic.

1.1.1. Oscil·loscopi

Un oscil·loscopi (vegeu la fotografia que apareix en aquest subapartat) és un equip de laboratori utilitzat per a examinar qualsevol senyal elèctric, tant analògic com digital.

Els oscil·loscopis són utilitzats de vegades per a observar ràpidament la tensió en un piu particular o, en absència de l'analitzador lògic, per a fer qualsevol observació elèctrica més sofisticada. De tota manera, el nombre d'entrades és petit (normalment quatre). Per a l'aplicació de depuració de codi, doncs, és una eina amb un ús menys important que un analitzador lògic.

Oscil·loscopi

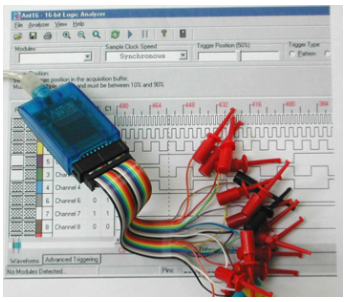
1.1.2. Analitzador lògic

Un analitzador lògic (vegeu la il·lustració d'aquest subapartat) és un equip de laboratori dissenyat especialment per a avaluar maquinari digital. Pot tenir dotzenes o, fins i tot, centenars d'entrades, cadascuna capaç de detectar una sola cosa: quan el senyal elèctric és al nivell lògic 1 o 0.

Més concretament, un analitzador lògic és un instrument de mesura que captura les dades d'un circuit digital i les mostra per a analitzar-les posteriorment, de manera similar a com ho fa un oscil·loscopi, però, a diferència d'aquest,

és capaç de visualitzar els senyals de múltiples canals. A més de permetre visualitzar les dades, verificar el funcionament correcte del sistema digital, pot mesurar temps entre canvis de nivell, nombre d'estats lògics, etc. La captura de dades des d'un analitzador lògic es fa a partir de la connexió d'un cable a la sortida lògica apropiada en el bus de dades que cal mesurar.

Els analitzadors són emprats principalment per a detectar errors i comprovar prototips abans de la seva fabricació, comprovant les entrades i analitzant posteriorment el comportament de les seves sortides. Avui en dia, la majoria dels analitzadors lògics disposen d'una connexió al PC on es poden veure les dades mitjançant una plataforma programari.

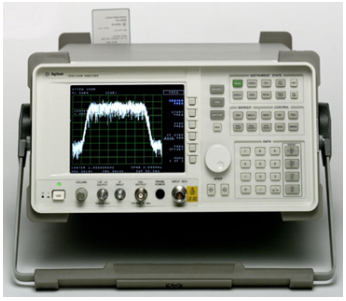


Analitzador lògic USB amb interfície per al PC

1.1.3. Analitzador d'espectres

Un analitzador d'espectre (vegeu la il·lustració d'aquest subapartat) és un equip de mesura electrònic que permet visualitzar en una pantalla les components espectrals en un espectre de freqüències dels senyals presents a l'entrada, que poden ser qualsevol tipus d'ones elèctriques, acústiques o òptiques.

En l'eix d'ordenades sol presentar en una escala logarítmica el nivell en dBm del contingut espectral del senyal. En l'eix d'abscisses es representa la freqüència, en una escala que és funció de la separació temporal i el nombre de mostres capturades. S'anomena **freqüència central** de l'analitzador la que es correspon amb la freqüència del punt mitjà de la pantalla.



Analitzador d'espectre Agilent

Sovint es mesura amb ells l'espectre de la potència elèctrica. En l'actualitat, està sent substituït pels analitzadors de xarxa.

1.1.4. Analitzador de xarxa

Un analitzador de xarxes és un instrument capaç d'analitzar les propietats de les xarxes elèctriques, especialment les propietats associades amb la reflexió i la transmissió de senyals elèctrics. Són, entre molts d'altres, usats per a estudiar els senyals de radiofreqüència que generen les comunicacions sense fil d'alguns dispositius.

Hi ha dos tipus principals d'analitzadors de xarxes:

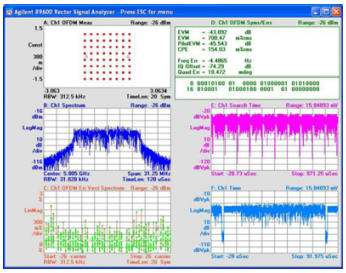
- **Analitzador de xarxes escalar (SNA, *scalar network analyzer*)**[2]. L'analitzador de xarxes escalar té només propietats d'amplitud.

- **Analitzador de xarxes vectorial (VNA, *vector network analyzer*)**[3]. L'analitzador de xarxes vectorials té propietats d'amplitud i fase.

(2)Un analitzador del tipus SNA té un funcionament idèntic a un analitzador d'espectre combinat amb un generador d'escaneig.

(3)Un analitzador del tipus VNA també pot ser anomenat *mesurador de guany i fase* o *analitzador de xarxes automàtic*.

En l'actualitat, la majoria d'analitzadors de xarxa presenten una interfície per a ser controlats des d'un ordinador (vegeu la imatge que il·lustra aquest subapartat).



Panell de visualització d'espectres d'un analitzador de xarxa Agilent

1.1.5. Multímetre

Un multímetre, polímetre o *tester* (vegeu la figura que acompanya aquest subapartat) és un instrument de mesura electrònic que incorpora diverses funcionalitats. La majoria porta com a mínim un amperímetre, un voltímetre i un òhmmetre. En el desenvolupament de sistemes encastats és usat durant el procés de prototipatge, ja que permet:

- Un **comprovador de continuïtat**, que xiula quan hi ha conducció elèctrica entre les dues sondes (en cas que una pista tingui un tall o error no xiula).
- Lectura de valors en pistes en lloc d'usar l'oscil·loscopi, ja que és més manejable.
- Mesura de voltatges i corrents petits i resistències elevades.

Tester

1.1.6. JTAG

Join Test Action Group (JTAG) és un grup format per fabricants d'electrònica d'arreu que cap a l'any 1985 van decidir trobar una solució comuna al problema de depuració (*debugging*) de programari. La solució va esdevenir l'estàndard 1149.1-1990 de l'IEEE Standard Test Access Port and Boundary-Scan Architecture. L'IEEE Std 1149.1 permet que instruccions de test i dades puguin ser carregades a la memòria dels dispositius i posteriorment recollir els resultats dels tests de la memòria del dispositiu.

JTAG d'Atmel

Aquesta iniciativa tenia, entre d'altres, els objectius següents:

- Millorar el test tradicional (*ad hoc* per cada dispositiu).
- Millorar l'eficiència del test.
- Reduir el cost del disseny de tests.
- Reduir el temps de producció reduint el temps de la fase de test.
- Permetre el test aïllat de parts de dispositius.
- Oferir un accés més simple als dispositius.

L'especificació de JTAG requereix els elements següents:

1) Un port anomenat **test access port** (TAP).
2) Un controlador anomenat **TAP controller**.
3) Un **registre d'instruccions** (IR).

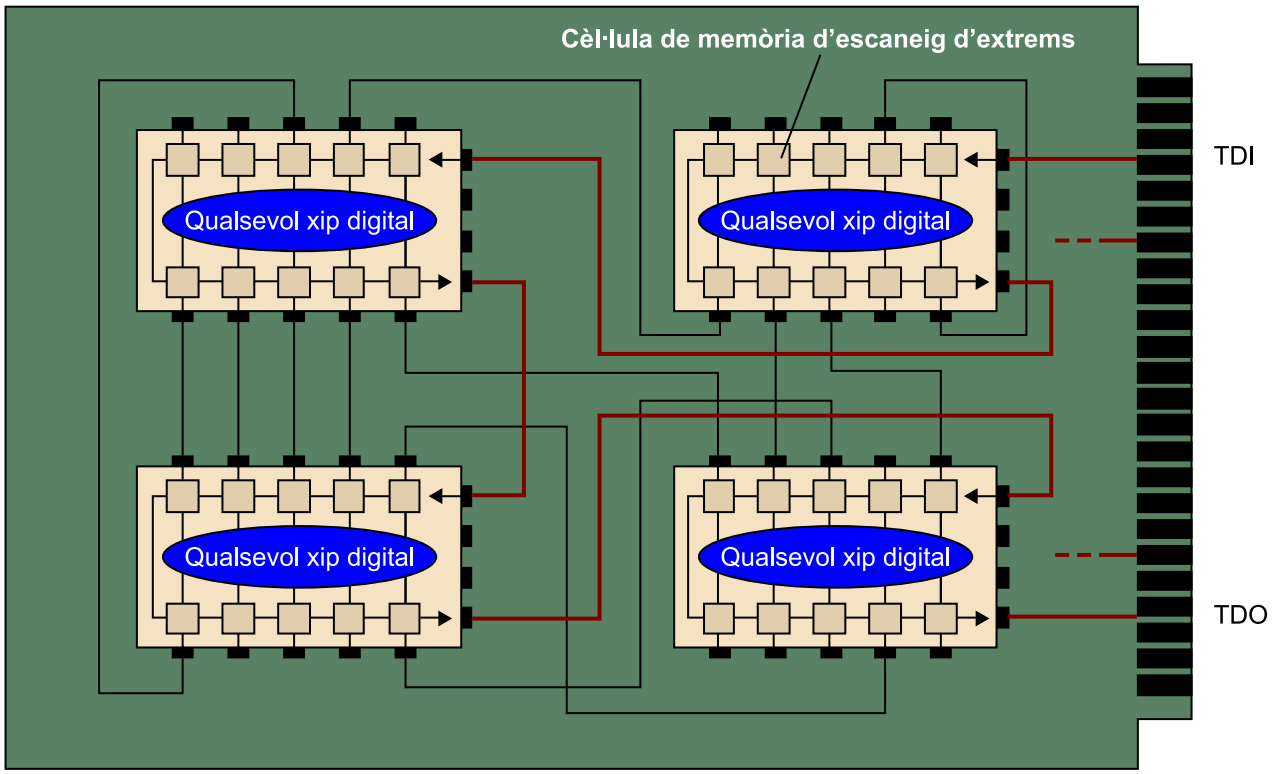
4) Un conjunt de **registres de dades** (TDR).

El TAP *controller* genera senyals de rellotge i de control per al circuit del JTAG. El TAP *controller* s'implementa com una màquina d'estats finita que obeeix a canvis d'estats imposats en l'IR d'una manera serial. El TAP té els pius següents:

- *Test data in* (TDI).
- *Test data out* (TDO).
- *Test clock* (TCK).
- *Test mode select* (TMS).
- *Test reset* (TRST) que és opcional.

El funcionament de la plataforma es basa en la tècnica d'escaneig d'extrems (*boundary scan*) i que bàsicament observa i permet actuar sobre els extrems de les entrades i sortides d'un xip. En particular, el seu funcionament es basa en l'ús de cel·les de memòria annexades a cada piu, les cel·les de memòria formen un registre de decalatge (*shift register*) anomenat *boundary-scan register* (BSR). El BSR és part dels TDR de l'especificació JTAG. La primera cel·la és connectada al piu TDI i la darrera cel·la és connectada al piu TDO. El BSR és controlat pel senyal de rellotge TCK. En mode normal, les cel·les simplement retransmeten el senyal que hi ha en el piu al qual estan annexades. En el mode test, el valor de cada piu pot ser assignat fent-li arribar un valor des del piu TDI mitjançant decalatges (*shifts*) del BSR. De la mateixa manera, es poden llegir directament els valors de les sortides del xip fent arribar la informació del BSR al TDO (mitjançant decalatges del BSR).

Arquitectura d'escaneig d'extrems utilitzada per JTAG

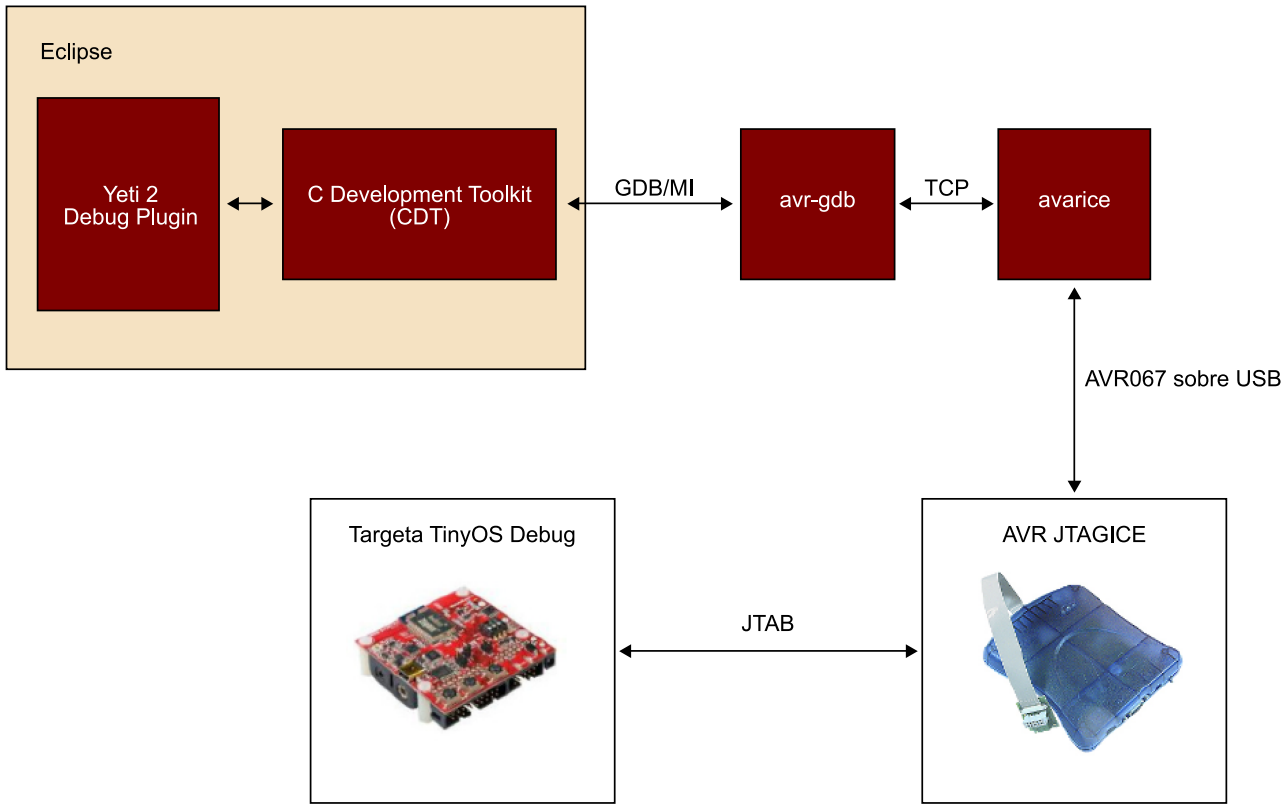


A la pràctica, el JTAG és l'especificació d'un format que és implementat en un dispositiu que es connecta a un port anomenat *TAP*, que porten la majoria de dispositius electrònics i que permet depurar en temps real i sobre la mateixa plataforma. En molts casos, aquest dispositiu es coneix amb el nom de *JTAG*, *TAP controller* o *on-chip debugger* (OCD) perquè permeten depurar en el mateix xip. Veurem més endavant que aquesta mena de dispositius també s'anomenen *monitors*.

Hi ha diferents fabricants d'OCD o JTAG i cadascun d'ells proveeix els seus controladors (*drivers*) i programari de depuració que han de ser configurats de manera específica per a cada plataforma. Trobem connectors (*plugins*) per als entorns de desenvolupament com IAR o Eclipse i controladors per a les diferents plataformes com TI, AVR o ATMEL, etc.

La majoria de vegades l'eina de depuració utilitzada és la desenvolupada per GNU com a depurador per a Linux, anomenada **gdb**. Els fabricants d'OCD desenvolupen un programari pont perquè gdb pugui ser usat sobre els controladors de l'OCD específic. D'altra banda, també en molts casos es desenvolupen connectors per a les plataformes com eclipse+cdt o IAR que faciliten la depuració i que integren funcionalitats específiques per a l'OCD. Aquests connectors també mantenen una interfície comuna amb gdb.

Exemple d'entorn de desenvolupament Eclipse + CDT + Yeti2 plugin + AVR JTAG per a depurar NesC i TinyOS



1.2. Eines de desenvolupament i depuració

Fins ara hem vist un conjunt de dispositius que, combinats amb programari, ens permeten tenir un control més proper del nostre desenvolupament. En aquest apartat aprofundirem en eines de desenvolupament de programari que conformen el nostre entorn d'avaluació.

1.2.1. L'entorn de desenvolupament

Un entorn de desenvolupament integrat[4] és un programa compost per un conjunt d'eines de programació.

[(4)]En anglès, *integrated development environment* (IDE).

Es pot dedicar en exclusiva a un llenguatge de programació, o bé es pot utilitzar per a diversos llenguatges. Els IDE proveeixen un marc de treball amigable per a la majoria dels llenguatges de programació. Són compostos, entre d'altres, pels elements següents:

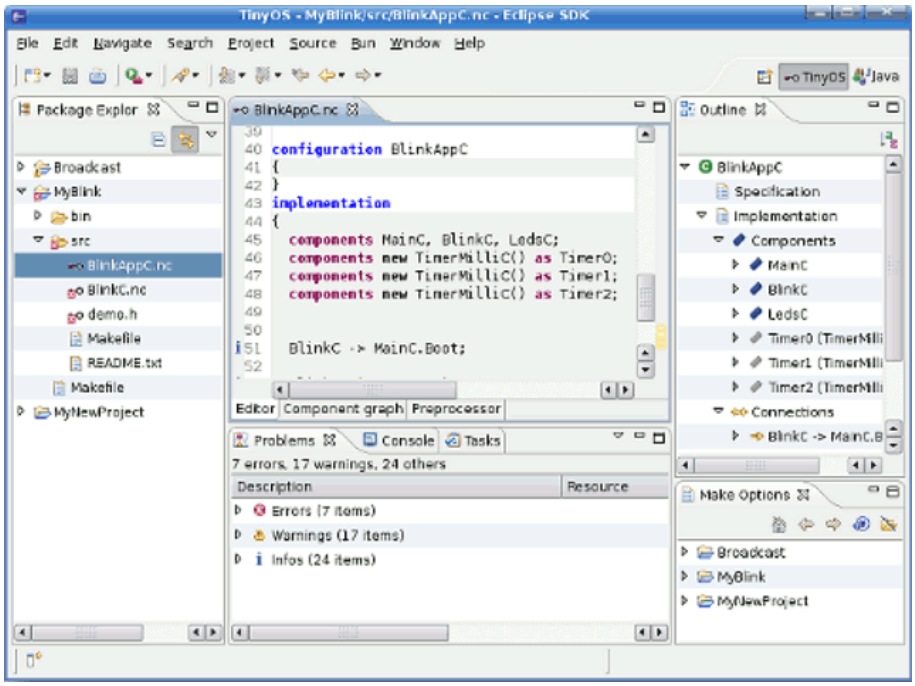
- Un editor de text.
- Un compilador.
- Un intèrpret.
- Un depurador.
- Possibilitat d'oferir un sistema de control de versions.
- Eines per a ajudar a la construcció d'interfícies gràfiques d'usuari.

Hi ha múltiples IDE per al desenvolupament d'aplicacions per a sistemes encastats. (Encara que no tots són específics, sí que ofereixen funcionalitats especialitzades.)

Nom	Llenguatge	Llicència
Eclipse+CDT	C/C++ , NesC, altres (+ connectors)	Codi obert (*open source*)
IAR	C/C++ (msp430, etc.)	Propietari però lliure per a programar amb memòria < 4 kB
AVR studio	C/C++	Propietari
CodeVision	C/C++	Propietari però lliure per a programar amb memòria < 3 kB
CODE::BLOCKS	C/C++	Codi obert
CodeWarrior	C/C++ (*freescale*)	Propietari però el *tool chain* és gratuït si es desenvolupa sobre els microprocessadors de *freescale*
µVision IDE (Keil)	C/C++ (arm, cortex-m...)	Propietari
CCStudio	C/C++ (msp430...)	Propietari amb versió gratuïta amb límit de 16 kB de codi per a MSP430

L'elecció d'un IDE o un altre és condicionada, en molts casos, pel compilador i el microcontrolador escollits, ja que cada plataforma té les seves particularitats i en molts casos els compiladors inclouen l'IDE.

Eclipse amb connectors CDT i Yeti2 per a treballar amb TinyOS



1.2.2. Els depuradors

Un **depurador** (en anglès, *debugger*) és un programa que permet depurar o netejar els errors d'un altre programa (programa objectiu). Hi ha molts depuradors diferents, especialitzats per a arquitectures diferents o per a llenguatges de programació específics. Per l'interès d'aquest curs ens centrarem en un d'ells.

GDB

GDB és el depurador de GNU. És un poderós depurador que permet "veure" què està succeint dins de programes escrits en C, C++, entre d'altres. Entre les capacitats més notòries d'aquest depurador són:

- Depurador de programes complexos amb múltiples arxius.

- Capacitat per a aturar el programa o executar una ordre en un punt específic (punts de ruptura, *breakpoints*), segons una condició (*watchpoints*) o en arribar un *signal* (*catchpoints*).

- Capacitat per a mostrar valors d'expressions quan el programa es deté automàticament (*displays*).

- És possible examinar la memòria o variables de diverses formes i tipus, incloent-hi estructures, vectors i objectes.

- És possible igualment canviar els valors de les variables per a estudiar el comportament del programa sense necessitat de recompilar.

- Possibilitat de depurar programes en execució (processos).

- Possibilitat de depurar programes que han finalitzat.

- Múltiples formes d'entrar al depurador.

1.2.3. Invocació del depurador

El depurador es pot executar d'una de les maneres següents:

```
$ gdb
```

```
$ gdb programa
```

per a carregar el programa i entrar en el mode interactiu. El programa no comença fins que sigui indicat amb una ordre

```
$ gdb programa core
```

per a depurar un programa que ha finalitzat amb un nucli (*core*). El depurador carrega el programa i el seu entorn exactament com va acabar. És útil per a verificar per què un programa va acabar malament o per a veure on un programa es va "penjar" (usant CTRL–\ per a tallar el programa i obtenir un nucli).

```
$ gdb programa pid
```

per a depurar un programa en execució amb el pid indicat. El procés es deté i el depurador el controla en un altre terminal. Summament útil per a depurar programes amb interfície des d'un altre terminal virtual. Una vegada que s'entra al mode interactiu, GDB accepta ordres fins que se li indiqui que es vol sortir amb l'ordre quit.

Ordres més habituals

Les ordres usades més freqüentment són:

```
list [arxiu:]funció
list [arxiu:]línia[,línia]
list
list &#8211;
```

per a fer una llista del codi font a partir d'una funció o una línia. list només continua la llista prèvia. list – enumera les línies anteriors.

```
break [arxiu:]funció
break [arxiu:]línia
```

per a col·locar un punt de ruptura al començament de la funció o al començament de la línia indicada.

```
run [arguments]
```

per a començar l'execució del programa des del principi. Els arguments són els passats a l'executable.

```
bt (backtrace)
```

per a mostrar la *stack* del programa, indicant les funcions invocades i en quins llocs van ser cridades.

```
print exp.
```

per a mostrar el valor d'una expressió.

```
c
```

per a continuar l'execució del programa després que hagi estat detingut amb un *signal* o un punt de ruptura.

```
next
```

executa la propera línia del programa sense entrar dins de les funcions. Es pot aprofitar el fet que GDB repeteix l'última ordre amb ENTER per a executar diverses línies seguides.

```
step
```

executa la pròxima línia del programa entrant en funcions. Es pot aprofitar el fet que GDB repeteix l'última ordre amb ENTER per a executar diverses línies seguides.

```
jump línia
```

salta a les ordres següents i comença l'execució a partir de *línia*. Útil per a continuar un programa que ha acabat amb core, que arregla la situació i salta les línies defectuoses.

```
help [item]
```

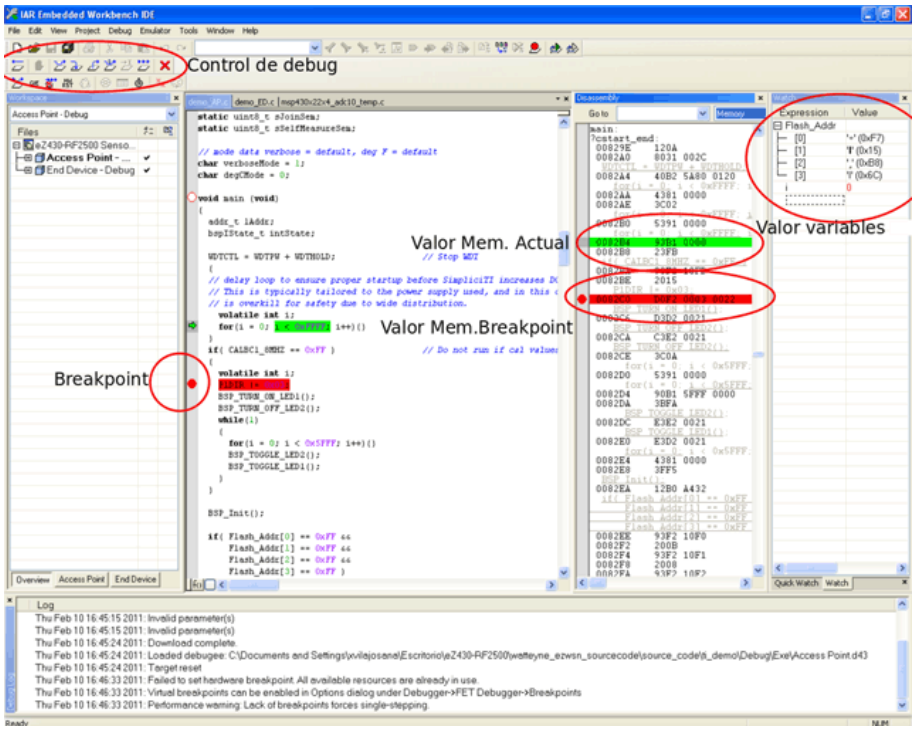
ajuda en línia.

```
quit
```

surt de GDB.

GDB, com hem vist, ofereix una interfície d'ordres que ens permet interactuar amb el programa que volem depurar. No obstant això, hi ha múltiples interfícies d'usuari que ens faciliten aquesta tasca. Molts dels entorns de desenvolupament presentats abans incorporen la interfície gràfica per a interactuar amb un depurador. Per exemple, IAR i eclipse+CDT incorporen la interfície per a depurar el codi usant GDB.

Entorn de desenvolupament d'IAR, en mode depurador



Com podem veure en aquesta figura, l'entorn de desenvolupament ens facilita la tasca de depuració oferint-nos de manera visual les eines per a depurar el codi. En la part superior esquerra de la imatge veiem els controls de depuració que ens permeten controlar l'execució del programa.

D'esquerra a dreta trobem:

- **Reset**: reinicia l'execució.
- **Step over**: salta a la instrucció següent. En el cas de ser una funció, la tracta de manera atòmica i no n'analitza el codi.
- **Step into**: salta a la instrucció següent. En el cas de ser una funció, es posiciona a la seva primera instrucció.
- **Step out**: en el cas d'haver entrat en una funció, ens posiciona després de la darrera instrucció d'aquesta funció.
- **Next statement**: ens porta sempre a la instrucció següent indiferentment de si és funció o no.
- **Run to cursor**: executa fins que arriba a la posició del cursor o fins que troba un punt de ruptura (*breakpoint*).
- **Run**: executa fins al final del programa o fins que troba un punt de ruptura.

És també molt útil l'eina d'inspecció de valors de variable en temps d'execució que podem veure en la part superior dreta de la figura.

Malgrat que en els darrers anys les eines de depuració per a sistemes encastats han millorat molt, hi ha casos, ja sigui per la disponibilitat de recursos, ja sigui perquè és un desenvolupament molt especialitzat, en què no podem utilitzar les eines introduïdes fins ara.

1.2.4. Depuració amb escriptura a terminal

L'ús de gdb o algun altre depurador ens permet veure pas a pas què està succeint en el nostre codi. No obstant això, de vegades ens interessa de manera ràpida veure la traça del nostre programa sense haver d'anar pas a pas. Així, doncs, hi ha eines que ens permeten, mitjançant el port sèrie (o USB), que el dispositiu pugui escriure missatges de traça al nostre terminal. Aquesta forma de depuració, coneguda popularment amb el nom de **depuració amb *printf***, no és recomanada com a única via de depuració d'un programa, però sí que ho és com a eina complementària a la depuració.

En molts casos, aquesta eina és oferta en format de biblioteca pel llenguatge de programació del sistema encastat, encara que no en tots els casos. Plataformes com les de Texas Instruments basades en *simplicity* ens ho permeten, però sempre mitjançant l'escriptura de *byte* per *byte* (o caràcter per caràcter) per mitjà de la UART. TinyOS, tot i que no de manera nativa, ens ofereix una biblioteca que ens permet fer aquesta mena de depuració.

Enllaç recomanat

Podeu trobar més informació sobre com s'ha d'utilitzar la biblioteca *printf* en TinyOS i NesC a: http://docs.tinyos.net/index.php/The_TinyOS_printf_Library_%28TOS_2.1.1%29

Exemple

En el llenguatge nesC per al sistema operatiu TinyOS tindríem el codi següent:

```
event void Boot.booted() {
       printf("Iniciem el nostre programa!\n");
       printf("El valor de la variable és: %u \n", var);
       printfflush();
 }
```

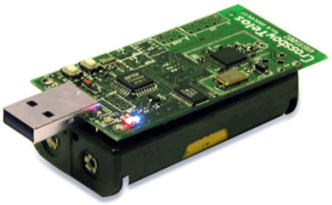
la sortida seria:

Iniciem el nostre programa!

El valor de la variable és: 0

1.2.5. Depuració via LED

Malgrat que és una manera de depurar, l'ús dels **LED** com a mecanisme de comunicació entre dispositiu i programador no és recomanable si no és com a complement d'alguna altra tècnica. Bàsicament, el programador estableix una codificació *ad hoc* sobre les combinacions possibles de LED encesos i apagats, o la combinació dels seus colors, si escau, que li permeten detectar certs estats del dispositiu o situacions d'error. Cal dir, però, que tot i que la depuració amb LED no és massa ortodoxa, sí que ho és l'ús de LED per a senyalitzar estats del dispositiu un cop l'aplicació ha estat del tot verificada. Normalment



TelosB de Crossbow amb dos LED

és útil definir una codificació de LED per a advertir a l'usuari de certs estats com la inicialització, la manca de bateria, que és fora de l'àrea de cobertura, la disfunció, el funcionament correcte, etc.

L'ús de LED està estès a la majoria de sistemes encastats de baixa capacitat com són les xarxes de sensors sense fil, ja que no disposen d'interfícies més complexes per a la visualització dels estats del dispositiu.

2. Simuladors

Qualsevol disseny d'un programa de propòsit específic requereix una simulació, que consisteix a intentar replicar les condicions d'aquell programa en un entorn controlat (d'aquesta manera potser no ens cal ni tan sols el maquinari d'aquell entorn i, simplement, necessitem un entorn en programari). Hi ha unes quantes raons per a recórrer a la simulació:

- Permeten als equips de treball de programari i maquinari treballar amb un cert grau d'independència, encavalcant tasques que d'una altra manera caldria executar seqüencialment.

- Ajuden a explorar l'arquitectura, de manera que permeten comparar diferents opcions de disseny sense comprometre's amb implementacions senceres.

- Permeten provar components independents, abans que altres parts del sistema estiguin disponibles.

- Permeten també connectar diferents peces del sistema per a provar-les conjuntament, per a depurar-ne la interacció, quan només una part del sistema està disponible o quan els volem proporcionar un entorn de test controlat.

- Poden ser usats per a validar o diagnosticar les prestacions en totes les etapes de disseny.

Actualment hi ha una gran varietat de tècniques de simulació, tant orientades a programari com a maquinari. La més flexible, però també lenta, és la simulació "interpretada". Consisteix a simular per programari el comportament del sistema.

Com veurem en aquest subapartat, també hi ha simuladors més lligats al maquinari del sistema de propòsit específic, com poden ser programes monitors i emuladors. Cada tècnica de simulació té avantatges i inconvenients davant les altres.

2.1. Programes simuladors

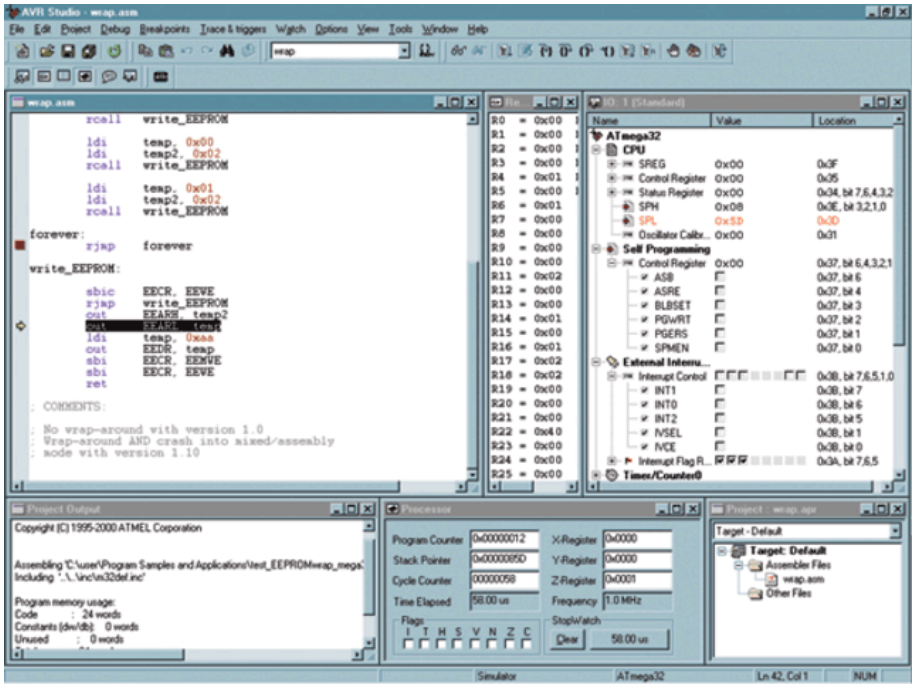
Un simulador executa el programa del microcontrolador en una màquina hoste (com un PC).

Es pot inspeccionar el codi mentre s'està executant pas a pas, per a veure exactament què està passant mentre el programa corre. Els continguts dels registres o variables poden ser alterats per a canviar la manera com funciona el programa.

Aquesta mena d'eina elimina (o almenys retarda) el cicle necessari amb una EPROM d'esborrament/gravació/programació comú en el desenvolupament de programes per a un microcontrolador.

Un simulador no pot suportar interrupcions ni dispositius reals i, normalment, corre molt més lentament que el dispositiu de propòsit específic real que vol simular. Tot i que els simuladors tenen alguns desavantatges, són molt útils en les primeres etapes de desenvolupament d'un projecte de programari quan encara no hi ha cap maquinari real amb el qual experimentar.

AVR studio amb funcionalitat de simulador d'un microprocessador d'Atmel



El principal gran desavantatge d'un programa simulador és que, en principi, només simula el microprocessador. I com sabem, un sistema de propòsit específic conté uns quants perifèrics importants. Això és així perquè el fabricant del microprocessador és qui normalment proporciona l'entorn de simulació i, per tant, normalment només inclou el microprocessador.

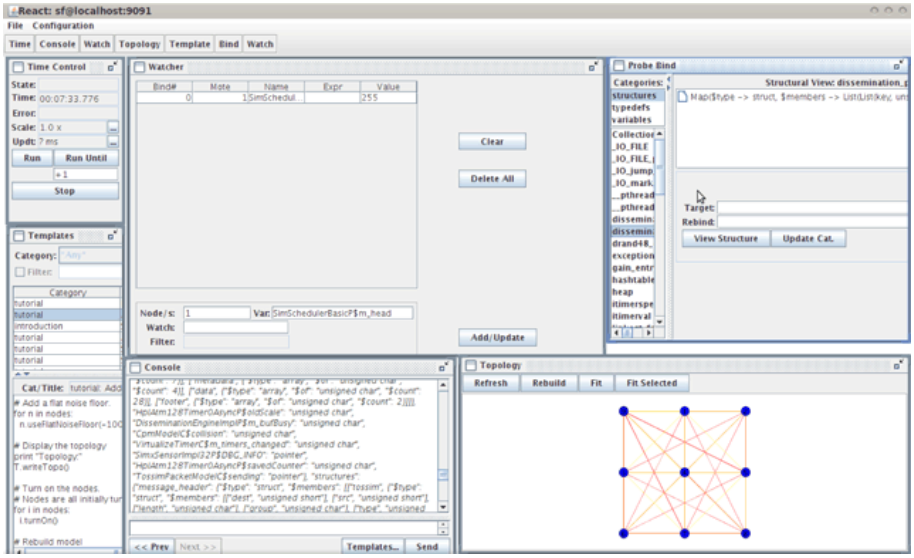
Hi ha simuladors per a moltes plataformes, entre d'altres, TOSSIM o SIMX són simuladors per a TinyOS que ens permeten avaluar el codi abans d'executar-lo en els dispositius. El problema dels simuladors com TOSSIM és que és del tot necessari que algú hagi desenvolupat el codi del simulador per a la plataforma que volem avaluar. En el moment en què siguem nosaltres els que ens

Enllaç recomanat

Per a més informació sobre TOSSIM, podeu consultar l'enllaç següent: http://docs.tinyos.net/index.php/TOSSIM

desenvolupem el maquinari i aquest no sigui idèntic al d'una plataforma ja existent, també haurem de desenvolupar el mòdul de TOSSIM específic per a la nostra plataforma.

Simulador SimX



2.2. Programes monitors

A diferència d'un simulador, un programa monitor corre en el mateix microcontrolador i, alhora, mostra el seu progrés en la màquina hoste (normalment un PC). Això es coneix com un *programa resident*.

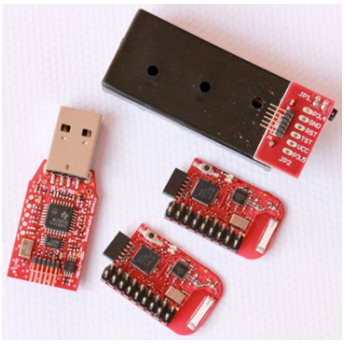
Té molts dels avantatges d'un simulador, amb el benefici addicional de veure com el programa corre en la màquina de destinació real. El resident necessita ocupar alguns dels recursos del sistema de propòsit específic (ja que hi corre), que inclouen:

- Un port de comunicacions per a comunicar-se amb l'hoste.
- Una interrupció per a manegar l'execució pas a pas.
- Certa quantitat de memòria per a la part resident del monitor.

Un programa monitor permet a l'usuari examinar la memòria, els registres i els ports d'entrada/sortida. Molts programes monitors tenen algunes capacitats bàsiques en comú, com són les habilitats per a dur a terme les accions següents:

- Examinar i alterar la memòria.
- Llegir i escriure dels ports d'entrada/sortida.
- Establir punts d'interrupció en el codi.
- Descarregar codi des d'un PC hoste.
- Interrompre l'execució de codi des del teclat.
- Comunicar-se via un port sèrie rs-232 (normalment).

Per a utilitzar el programa monitor, l'usuari estableix un punt d'interrupció en algun lloc útil del codi, corre el programa, i aleshores examina la memòria i els registres quan s'arriba al punt d'interrupció. Aquests punts d'interrupció també es poden introduir en les rutines d'error per a veure si aquell error succeeix.



ez430-rf2500 de Texas Instruments amb connector per a port USB i portapiles

L'usuari descarrega codi a la RAM del sistema de destinació i l'executa des d'allà. Aquesta comunicació s'estableix per un port sèrie, o potser per una connexió de xarxa. La interfície amb l'usuari del PC té l'aspecte de qualsevol monitor utilitzat per a programar un PC (permet veure el codi font, els registres, etc.), però en el nostre cas una part d'aquest monitor és resident al sistema de propòsit específic.

El monitor, de fet, consisteix en dues peces de programari i un maquinari que segueix l'especificació JTAG vista anteriorment. La interfície amb l'usuari corre en el PC, mentre que el monitor resident corre en el processador del sistema de propòsit específic. Hi ha alguns tipus de comunicació entre tots dos.

Els monitors són una de les eines més comunes de descàrrega i test utilitzades durant el desenvolupament de programari de propòsit específic. Això es deu, principalment, al seu baix cost.

Els programadors ja disposen d'un PC, mentre que el preu de la interfície (JTAG) no afegeix gaire cost al conjunt d'eines de desenvolupament (compilador, editor de codi, etc.). Finalment, els fabricants de monitors residents normalment estan disposats a distribuir gratuïtament el codi font per als seus monitors, per tal d'incrementar el nombre d'usuaris.

2.3. Emuladors

Proporciona un control absolut sobre el sistema de destinació, però no requereix cap recurs d'aquest sistema de destinació. L'emulador pot ser un dispositiu independent (com en la figura que il·lustra aquest subapartat), fins i tot amb la seva pròpia pantalla, o pot tenir una interfície amb l'usuari mitjançant PC.

Imatge de circuit emulador de STMicroelectronics per al seu processador ST72561

Els monitors són molt útils per a controlar l'estat del programari de propòsit específic, però només un emulador permet examinar l'estat del processador en què el programa està corrent. De fet, un emulador ocupa realment el lloc del processador.

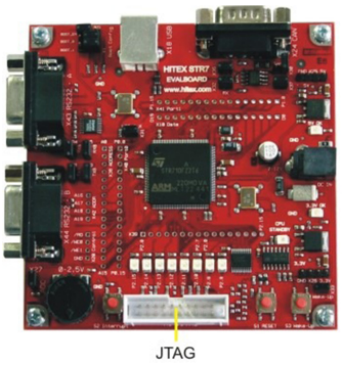
Un emulador[5] és un sistema de propòsit específic en ell mateix, amb la seva pròpia còpia del processador de destinació, RAM, ROM i el seu propi programari de propòsit específic.

Com a conseqüència d'aquestes característiques, un emulador és usualment costós, més fins i tot que el maquinari examinat. Però es tracta d'una eina molt potent, i que pot significar una gran ajuda durant el desenvolupament.

(5)Els emuladors són coneguts també amb el terme *placa d'avaluació* o *evaluation board*.

Tal com hem vist que fa un monitor, un emulador normalment utilitza un programa corrent en un PC com a interfície amb l'usuari. De vegades, fins i tot és possible utilitzar la mateixa interfície en tots dos casos. Però gràcies al fet que l'emulador conté la seva còpia del processador que es vol programar, és possible controlar l'estat d'aquest processador en temps real.

Això permet a l'emulador suportar característiques avançades de depuració, com punts d'interrupció per maquinari i seguiment en temps real, a més dels avantatges de qualsevol monitor.



Placa d'avaluació de STR7 amb el port per a JTAG

2.3.1. Punts d'interrupció per a maquinari

Amb un monitor es poden establir punts d'interrupció del programa. Ara bé, aquests punts són *programari*, en el sentit que estan restringits a interrupcions per instrucció. Són l'equivalent a "atura l'execució si aquesta instrucció està a punt de ser executada".

Els emuladors, en canvi, també suporten punts d'interrupció per maquinari. Aquests punts permeten parar l'execució en resposta a un ampli ventall d'esdeveniments. Aquests esdeveniments poden ser tant instruccions com lectures i escriptures de memòria, i també qualsevol mena d'interrupcions del processador.

2.3.2. Seguiment en temps real

Una altra característica útil d'un emulador és el seguiment en temps real. Típicament, un emulador incorpora un gran bloc de RAM de propòsit específic, dedicada a emmagatzemar informació sobre cadascun dels cicles de processador que s'executen. Això ens permet veure exactament en quin ordre estan passant les coses i, per tant, ens ajuda a respondre moltes preguntes, com si una interrupció va ocórrer abans o després que una variable prengués un cert valor.

A més, és possible o bé restringir la informació de seguiment que s'hi emmagatzema o postprocessar-la abans d'inspeccionar-la per tal de reduir-ne el volum.

Un cop tenim accés al maquinari, els analitzadors lògics i els oscil·loscopis poden ser eines de depuració indispensables.

Són més útils per a depurar les interaccions entre el processador i altres xips del sistema (de la placa electrònica). Com que aquests instruments només poden observar senyals externs al processador, no poden controlar el flux d'execució del programa, tal com fan un monitor o un emulador. Això fa que aquestes eines siguin menys valuoses per elles mateixes. Però combinades amb un monitor o un emulador, poden tenir una vàlua molt important.

Qualsevol subconjunt d'entrades que vulguem seleccionar es pot visualitzar en pantalla amb un eix de temps. Molts analitzadors lògics també permeten començar a capturar dades, a partir d'un patró específic.

La majoria d'eines de depuració que heu estudiat en aquest subapartat seran utilitzades en un punt o un altre de tots els projectes de propòsit específic. Els oscil·loscopis i analitzadors lògics són usats més freqüentment per a depurar errors de maquinari; els simuladors, durant les primeres etapes del desenvolupament del programari, i els monitors i emuladors, durant la depuració del programari real. Per a ser més efectius, hauríeu d'entendre la utilització de cada eina i quan i on l'heu d'aplicar per a obtenir els beneficis més elevats.

3. Metodologies de verificació i test

En aquesta secció, es presenten les tècniques més habituals de verificació i avaluació de programes. Aquestes tècniques són presentades d'una manera genèrica i en cap cas no es poden considerar exclusives per a l'entorn dels sistemes encastats. En canvi, són metodologies utilitzades àmpliament en la verificació de tota mena de programari.

L'avaluació del programari és una part essencial en el desenvolupament. Si el programari que desenvolupem no funciona, no serà usat (i probablement no ens pagaran). No obstant això, abans d'endinsar-nos en el món de verificació de programari és important recordar els usuaris del nostre programa.

- Els **usuaris** veuen el programari des de fora. No avaluen els algorismes ni l'estructura del codi. El teu sistema per a ells és una **caixa negra**. Només els interessa la funcionalitat.

- Els **avaluadors** (*testers*). Miren que els resultats que dóna el programari siguin els esperats. Els interessa la funcionalitat, però també esperen que el programari faci exactament allò que ha de fer. Miraran que les dades són correctes, que els ports treuen la informació que han de treure, que la memòria és alliberada correctament, sense entrar en els detalls dels algorismes o particularitats del codi. Així, doncs, els avaluadors veuen el nostre programari com una **caixa grisa**.

- Els **desenvolupadors**. Veuen l'arquitectura del programa, l'estructura del codi, els estats, els patrons. El codi és obert a ells i veuen el programa com una **caixa blanca**. De vegades, però, una visió tant propera els fa perdre la perspectiva per a veure funcionalitats mal dissenyades o amb alguna mancança que, en canvi, els avaluadors o usuaris poden copsar.

Així, doncs, una bona avaluació de programari ha de tenir en compte les tres perspectives que s'exposen tot seguit.

3.1. Avaluació de la caixa negra

Els usuaris són fora del sistema. Ells només veuen les entrades i sortides del sistema. D'aquesta manera, quan s'avalua la caixa negra s'hi han de cercar:

- **Funcionalitats**. Aquesta és la prova més important. Fa el sistema el que ha de fer? No són importants els detalls de com ho fa, o com són les sortides: només importa saber si la funcionalitat del sistema és l'especificada.

- **Validació de les entrades**. S'ha de verificar que les entrades del programari estan protegides a valors incorrectes o formats no suportats. Què passa si esperem un valor positiu i li introduïm un valor negatiu?

- **Validació de les sortides**. S'ha de verificar el resultat manualment, si escau. S'han de verificar tots els camins possibles. És molt útil fer una taula que especifiqui totes les entrades i totes les possibles sortides que es poden donar. Amb aquesta taula es pot verificar el funcionament del programa.

- **Transicions d'estat**. Alguns sistemes tenen diferents estats. S'ha de verificar que el sistema passa pels estats que ha de passar. Si hi ha diferents camins, s'ha de verificar la correcció de tots ells. Això és particularment important en sistemes en temps real o en la implementació de protocols de comunicació. En aquest procés és molt útil construir un diagrama d'estats i verificar que es compleixen totes les transicions.

- **Casos extrems i casos no vàlids però en el llindar**. S'han de verificar els casos extrems. Tant els valors que entren en els llindars com els que són just per sobre.

Representació de la tècnica d'avaluació de caixa negra. Només hi veiem entrades i sortides

Una especificació simple de caixa negra

Nota

És comú planificar l'avaluació fent servir taules en què s'especifiquen tant les entrades com les sortides esperades.

3.2. Avaluació de la caixa grisa

L'avaluació amb la caixa negra en molts casos és suficient per a validar un programa. No obstant això, hi ha situacions en què es fa necessària una verificació més profunda. De vegades, no es poden obtenir els resultats d'un programa des de fora, sinó que o bé perquè el programa és part d'un sistema més gran, o perquè la seva funcionalitat no s'expressa en una sortida explícita, cal mirar dins del programa.

Això és particularment cert en programes per a sistemes encastats que en molts casos només mouen dades de la memòria del dispositiu.

Quan es fa servir la tècnica de caixa grisa normalment es duen a terme la mateixa mena de verificacions, però aquest cop verificant alguns dels estats dins del programa.

- **Verificació, auditoria i log**. Es fa ús dels logs del programa o traces de memòria per a veure què ha passat.

- **Dades per a altres programes**. Si el programari que s'està avaluant és un subsistema o biblioteca s'han de verificar que les sortides són les esperades per als programes que el faran servir.

- **Informació del sistema**. Quan el nostre programa fa escriptures en disc, o fa servir rellotges interns del sistema, s'ha de verificar que les marques horàries (*timestamps*), les sumes de verificació (*checksums*) i altres dades que genera el mateix sistema són correctes i en el format volgut.

- **Revisió de l'estat de la memòria**. Encara que el nostre programa funcioni és possible que no hàgim fet un ús adequat de la memòria o que hàgim deixat variables sense usar o cadenes (*threads*) en estat zombi que posteriorment poden afectar el funcionament d'altres biblioteques.

Procés zombi

Un procés zombi (en anglès, *zombie process* o *defunct process*) és un procés que ha completat la seva execució però que encara té una entrada en la taula de processos. Aquesta entrada és encara necessària per a permetre que el procés que el va engegar (procés pare) n'obtingui l'estat d'acabament. En cas que el procés pare hagi acabat, no es podrà eliminar aquesta entrada en la taula de processos i deixar la memòria ocupada. De fet, el terme *procés zombi* deriva de la definició comú de *zombi* (una persona no morta). Seguint amb la metàfora, el procés fill ha mort, però no ha estat enterrat.

3.3. Avaluació de la caixa blanca

Aquesta avaluació és la mes profunda. Aquest procés d'avaluació mirarà què passa exactament dins del codi i es farà el possible per a fer que el codi falli. L'avaluació de caixa blanca pot esdevenir un repte per a l'avaluador, ja que el seu objectiu és fer que el codi falli. Per a fer aquesta mena d'avaluació, cal estar familiaritzat amb el codi i conèixer les precondicions i postcondicions imposades a cada mètode.

Normalment, es busca:

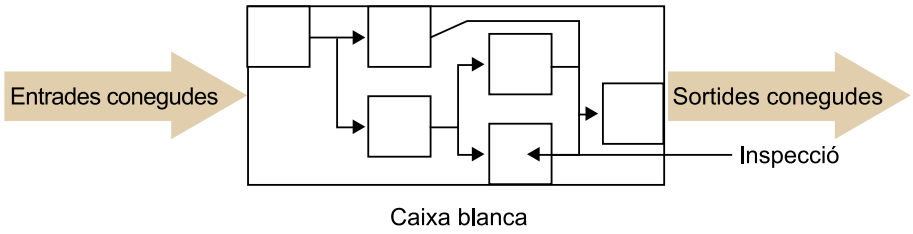
- **Verificar tots els camins del codi**. Per a cada `if/else` o `switch/case` s'ha de verificar cadascuna de les branques del codi. Verificar quines són les dades que ens permetran anar per cadascuna de les branques i veure si el nostre programa és capaç de generar-les.

- **Gestió dels errors**. Si en un mètode o funció es genera un error, aquest és gestionat? S'ha de verificar que els errors no comportin estats inconsistents ni que modifiquin de manera inconsistent les dades de memòria.

- **Gestió de la memòria**. En el cas d'errors, s'alliberen els recursos ocupats? En general, quan acaba l'execució d'un mètode s'allibera la memòria que no ha de ser usada?

- **Gestió de recursos**. Què fa el nostre programa quan es vol accedir a un recurs que està ocupat?

Avaluació en caixa blanca, hem d'estudiar el codi per dins



Resum

Aquest mòdul ens ha presentat les principals eines que conformen l'entorn de desenvolupament per a treballar amb sistemes encastats. Ens ha mostrat les eines de maquinari més usades per a avaluar i depurar els nostres programes. Hem vist que els oscil·loscopis són usats per a veure els senyals elèctrics que circulen pel programari durant l'execució del nostre programa. Tot i que són útils, quan es volen estudiar diverses entrades i sortides del maquinari s'utilitzen els analitzadors lògics que permeten observar un nombre més gran de pius. Quan el que es vol avaluar són el comportament de senyals de radiofreqüència generats pel maquinari s'utilitzen els analitzadors d'espectre o analitzadors de xarxa que ens permeten veure de manera freqüencial el comportament del nostre programa. Hem vist, també, que una eina molt útil és el monitor que fa servir l'especificació del JTAG, una eina que ens permet depurar el codi sobre la plataforma programari real. Quant a programari, s'han introduït els entorns de desenvolupament integrats (IDE) com a eines que, de manera amigable, ens permeten desenvolupar codi, depurar-lo i que simplifiquen molt la tasca del programador. No obstant això, hem vist que la tria de l'IDE és condicionada per la plataforma programari i compiladors escollits i que en molts casos estan subjecte a llicències propietàries. Els depuradors són eines imprescindibles per a assegurar el funcionament correcte de les nostres aplicacions. S'han presentat les principals característiques i funcionalitats dels depuradors posant com a exemple el depurador GDB de gnu que està inclòs en gairebé totes les plataformes *nix. També hem vist que els IDE inclouen interfícies que fan més amigable el procés de depuració i que en molts casos inclouen eines que ens permeten veure l'estat de la memòria en temps real. Això és del tot cabdal en els sistemes encastats, ja que no disposem de sortides explícites que ens permetin avaluar el funcionament correcte del programa.

Seguidament, el mòdul ens ha presentat diverses tècniques de simulació, molt lligades a la depuració i que ens serveixen per a verificar el funcionament correcte d'un sistema encastat a mesura que es va desenvolupant. Tant és així que en molts casos els desenvolupaments no es fan sobre el programari definitiu, sinó que s'utilitzen plaques d'avaluació o desenvolupament que inclouen funcionalitats orientades a la depuració. Un cop s'ha desenvolupat el programa i s'està segur que funciona correctament, es desenvolupa el maquinari retallant les funcionalitats que la placa de desenvolupament oferia i que ja no són necessàries.

Finalment, se'ns ha presentat la part més important d'aquest mòdul que és la metodologia de depuració. Aquesta metodologia és genèrica i serveix per al desenvolupament de programari, en general, però és del tot aplicable als sistemes encastats. Hem vist la depuració en caixa negra orientada al que veuran

els usuaris; en caixa grisa, que ens permet verificar la correcció de les dades durant el procés d'execució del nostre programa, i, finalment, la depuració en caixa blanca que avalua tots els possibles estats del nostre programa.

Bibliografia

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

Diversos autors. "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications". A: In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*.

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

IEEE Std 1149.1-1990 – Test Access Port i JTAG Architecture, i el Std 1149.1-1994b – complement de IEEE Std 1149.1-1990, estan disponibles a IEEE Inc., 345 East 47th Street, Nova York. També es pot obtenir una còpia de l'estàndard IEEE 1149.1 en línia: http://www.ieee.com/

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

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

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

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

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

Annex

1) Guia d'ús de GDB

Amb les ordres esmentades anteriorment, és possible depurar programes fàcilment i amb bon control. Aquesta secció pretén mostrar breument com es pot començar a usar gdb amb només aquestes ordres. Per a poder usar un depurador en UNIX cal compilar el programa amb l'opció –g. Per exemple:

```
cc -g prova.c -o prova
```

a. Executar el programa des del depurador. Mètode recomanat en les primeres fases de prova. S'obté un control complet del programa i, si aquest falla, es visualitza immediatament on ha fallat. És possible aturar el programa en qualsevol moment amb CTRL + C i tornar al depurador, la qual cosa permet verificar bucles infinits, etc.

```
$ gdb prova
(gdb) run [arguments]
...
CTRL+C
(gdb)
```

Una vegada detectades funcions en què hi pot haver problemes:

```
$ gdb prova
(gdb) list funció
...
(gdb) break línia
(gdb) run [arguments]
...
break
(gdb) print expr
(gdb) next
...
(gdb) c
...
```

b. Determinar on un programa acaba amb *core*.

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

```
...
```

Una vegada dins de gdb, es poden examinar unes altres variables i fer un *back-trace* per a verificar el camí que pren el programa per a produir l'excepció. És possible arreglar la situació i executar un *jump* per a continuar ignorant l'error (vegeu aquestes ordres avançades):

```
(gdb) arreglar la situació
(gdb) jump línia                        executar ignorant l'error
```

c. Programa executant. És possible depurar un programa en execució. Per a això fer:

```
$ ps                      per a determinar el pid
$ gdb programa pid        per a interceptar el programa
(gdb)                     en aquest moment el programa es deté
...
```

2) Guia d'ús de GDB

Els punts de ruptura són punts en què el programa es deté quan hi passa. *Watchpoints* són expressions que detenen el programa quan el valor canvia. *Catchpoints* són punts de ruptura sobre *signals*. Les ordres són com segueixen:

```
break [arxiu:]funció
break [arxiu:]línia
```

per a col·locar un punt de ruptura al començament de la funció o al començament de la línia indicada.

```
tbreak [arxiu:]línia
tbreak [arxiu:]funció
```

igual que *break*, però el punt de ruptura és vàlid una sola vegada. Útil per a crear punts de ruptura temporals.

```
watch exp
```

s'habilita un *watchpoint* quan l'expressió `<expr>` canvia.

```
catch
```

es col·loquen punts de ruptura en tots els gestors (*handlers*) d'excepcions del context actual.

```
info break
```

```
info watch
```

mostra els *watchpoints* o punts de ruptura habilitats.

```
info match
```

indica si s'estan interceptant les excepcions.

```
clear línia
clear funció
```

per a eliminar un punt de ruptura de la línia indicada. Vegeu delete per a eliminar punts de ruptura per nombre.

```
delete numero
```

per a eliminar un punt de ruptura per nombre. El nombre es pot veure amb info break.

```
disable breakpoint
enable  breakpoint
```

per a habilitar o deshabilitar temporalment un punt de ruptura. A diferència de delete no es perd la referència de la línia en què es troba, simplement és ignorat.

```
condition breakpoint [expr]
```

per a fer que un punt de ruptura sigui condicional. És a dir, només s'habilita si l'expressió `<expr>` és certa. Com `<breakpoint>` s'ha de passar el nombre del punt de ruptura. Si `<expr>` no s'especifica es fa el punt de ruptura incondicional.

```
ignore breakpoint [count]
```

ignora `<count>`passades sobre el punt de ruptura `<breakpoint>`.

3) Examinant i canviant les dades

GDB ofereix ordres per a manipular les dades, entre les quals hi ha:

```
whatis variable
```

indica quin tipus és la variable.

```
ptype tipus
```

imprimeix la definició del tipus indicat.

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

imprimeix l'expressió. /fmt és un indicador de format. Per a print el format només pot ser compost per una lletra de canvi de tipus.

```
set variable=expressió
```

canvia el valor d'una variable al resultat de l'expressió.

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

habilita un *display* continu durant la depuració de l'expressió indicada. Cada vegada que el programa s'atura es fa un print [/fmt] de l'expressió. Es poden tenir diversos *displays* alhora. Amb `delete` es poden eliminar `displays` i amb `enable` i `disable` es poden habilitar i deshabilitar igual que si fossin punts de ruptura.

```
undisplay numero
```

equivalent a delete d'un *display*: destrueix un *display*.

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

examinar memòria. /fmt és un indicador opcional format per '/', seguit d'un nombre (comptador), seguit d'una lletra de format, seguit d'una lletra de mida. Les lletres de format són:

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

El comptador indica quants elements cal imprimir, així:

```
                x /10xb vector
```

imprimeix els 10 bytes següents del vector en hexadecimal.

4) Manipulant la *stack*

Les ordres per a manipular la stack permeten canviar o examinar variables que es troben en altres contextos al local. És possible verificar el contingut de totes les variables automàtiques en la stack. Les ordres són:

```
                        frame [N]
```

selecciona el *frame N* i l'imprimeix. Sense argument l'ordre indica on és actualment (per exemple, *frame* actual).

```
          bt [N]
```

per a veure el contingut de la stack. Si s'especifica un nombre positiu es veuen les *N* primeres entrades en la stack i si és un nombre negatiu es veuen les últimes *N*.

```
          select-frame #
```

se selecciona el *frame number* indicat (el nombre el dóna l'ordre bt). En canviar de *frame* permet veure o canviar les variables sobre la stack. No desapila ni apila cap nou *frame*.

```
          up
```

per a anar al *frame* immediatament superior (és a dir, la rutina que crida l'actual).

```
          down
```

per a anar al *frame* immediatament inferior.

```
   return
```

per a forçar el desapilament del *frame* actual.

5) Ordres per a impressió d'informació d'estat

Les ordres següents imprimeixen informació variada d'estat del depurador i del programa depurat:

```
          info files
```

mostra els arxius i processos que s'estan depurant.

```
          info program
```

estat del programa al moment.

```
info sources
```

mostra els arxius font en depuració.

```
info types
```

mostra tots els tipus definits.

```
info variables
```

mostra totes les variables globals definides.

```
info functions
```

mostra totes les funcions definides.

```
info display
```

mostra totes les expressions *display* en efecte.

```
info breakpoints
```

mostra tots els punts de ruptura en efecte.

```
info watchpoints
```

mostra tots els *watchpoints* en efecte.

```
info args
```

mostra els arguments del *frame* actual.

```
info locals
```

mostra les variables locals del *frame* actual.

6) Altres ordres útils

Altres ordres de GDB que són útils a l'hora de depurar:

```
file
```

per a carregar un nou executable i taula de símbols dins del depurador.

```
cd
```

per a canviar de directori.

```
pwd
```

per a veure el directori actual.

```
shell
```

per a sortir a un *subshell*.

```
search reg-expr
```

per a buscar en el font una expressió regular a partir de la línia actual cap avall.

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

per a buscar en el font una expressió regular a partir de la línia actual cap amunt.

```
make
```

per a córrer el programa make.