GNU/Linux embedded: concetti base

Alessandro Rubini <rubini@gnudd.com> http://ar.linux.it http://gnudd.com

- La distribuzione embedded
- Busybox
- Il boot loader

- La procedura di boot
- Il processore ARM
- Il ruolo del linker
- Assembler inline

La distribuzione embedded

Anatomia di una distribuzione embedded

La tipica distribuzione embedded include:

- Un boot loader (opzionale)
- Un kernel (opzionale)
- Una libreria (opzionale)
- Uno spazio utente

La personalizzazione in genere si limita allo spazio utente

Il vero embedded _non_ è quello fatto su x86

- Può essere difficile fare una distribuzione per PC

Molti embedded per PC sono realizzazioni domestiche

Busybox

Busybox è il protagonista di ogni distribuzione embedded

- È uno strumento nato per i dischi di installazione Debian

In busybox sono contenute molte applicazioni Unix

- L'occupazione su disco è limitata
- argv[0] viene usato per selezionare il comportamento
- Installazione come link simbolici o fisici

Per compilare

- make config && make

Per installare

- make install
- make install PREFIX=/path/to/installation/dir

```
http://ftp.busybox.net/downloads/busybox-1.9.1.tar.bz2
```

uC-libc

La micro-controller libc è una libreria di sistema ridotta

È una scelta utile solo in casi particolari

- Tutto il sistema va compilato con quella libreria
- Programmi non banali possono avere qualche limitazione
- Occorre un compilatore apposito oppure opzioni speciali
- Per x86 esiste un "cross-compilatore" a tal fine.
- Per arm e ppc esiste una versione apposita di ELDK

Nel caso di sistemi con MMU solitamente si usa glibc

I passi per realizzare la propria distribuzione

Creazione del proprio albero per un'applicazione embedded:

- Si sceglie una catena di compilazione
- Si prende una libc (normalmente quella del compilatore)
- Si sceglie un init
- Si compila busybox
- Si aggiungono le proprie applicazioni
- Si popola la directory /dev
- Si mette tutto sul filesystem prescelto
- Si documentano esattamente i passi compiuti

Come _non_ fare la propria distribuzione:

- Installare una distribuzione per PC
- Togliere i pacchetti non utilizzati
- Togliere il database rpm

Uso e configurazione di uclinux-dist

Caratteristiche di uclinux-dist (e snapgear):

- Kernel 2.0, 2.4, 2.6
- Glibc e uclibc
- Molte architetture supportate

Caratteristiche principali della distribuzione:

- Basata sul concetto di "vendor"
- Crea un'immagine del kernel e un filesystem in ./images/
- Il filesystem generato è visibile in ./romfs/
- La directory /dev non viene popolata realmente
- È possibile compilare senza essere superutente

```
for n in @*; do sudo mknod $(echo $n | tr '@,' '  '); done
```

Configurazione:

- make xconfig (come il kernel)
- output salvato in vari file chiamati .config
- per una vera personalizzazione occorre creare un albero in ./vendors/

Dall'accensione a /sbin/init

init: avvio del sistema

All'accensione del sistema:

- Il processore sceglie la periferica di avvio (pin-strap)
- Se previsto, viene scaricato codice dall'esterno ed eseguito
- Se previsto, l'esecuzione puo` avvenire da ROM interna al micro

Il boot loader

- Normalmente, il boot loader esegue dalla flash
- Il boot loader recupera un kernel e lo copia in ram
- L'esecuzione passa al kernel

Avvio da NAND

- Occore un piccolo IPL che carichi il boot loader
- Questo codice e` letto dalla ROM in una RAM interna

Talvolta si sceglie di saltare nel kernel senza copiarlo

- Il termine usato è XIP (execute in place)
- Il segmento testo viene usato dove sta
- Il segmento dati viene copiato in RAM
- Il bss viene allocato in RAM e azzerato

init: il codice del kernel

Il kernel inizia ad eseguire in arch/$ARCH/kernel/head.S

- Viene predisposto il segmento BSS
- Vengono fatte inizializzazioni di macchina
- Viene attivata la MMU
- Si salta al codice C

A questo livello non è disponibile printk

- Per la diagnostica si usano procedure in codice macchina
- Su alcune piattaforme esiste CONFIG_EARLY_PRINTK

Il codice C si occupa di tutto il resto

- start_kernel(), in init/main.c, lavora già con la MMU

Spesso il kernel si decomprime autonomamente

- Il codice in arch/$ARCH/boot/ gira senza MMU
- Anche a questo livello niente printk

init: i device driver

I device driver vengono inizializzati dal meccanismo __initcall

```
#include <linux/init.h>
static int __init local_init(void)
__initcall(local_init);
```

Le due macro __init e __initcall servono per istruire il linker

- Il codice di inizializzazione viene raggruppato insieme e poi liberato
- Tutte le funzioni definite come __initcall vengono elencate in una tabella
- L'implementazione di questi meccanismi sta in vmlinux.lds

```
. = ALIGN(4096);
__init_begin = .;
.text.init : { *(.text.init) }
.data.init : {
  *(.data.init);
  /* ... */
 }
 /* ... */
__initcall_start = .;
.initcall.init : { *(.initcall.init) }
__initcall_end = .;
. = ALIGN(4096);
__init_end = .;
```


init: invocazione di init

Il processo init viene eseguito direttamente dal kernel

```
/usr/src/linux-2.4/init/main.c:
       /* ... */
       if (execute_command)
           execve(execute_command, argv_init, envp_init);
       execve("/sbin/init", argv_init, envp_init);
       execve("/etc/init", argv_init, envp_init);
       execve("/bin/init", argv_init, envp_init);
       execve("/bin/sh", argv_init, envp_init);
       panic("No init found.  Try passing init= option to kernel.");

/usr/src/linux-2.6/init/main.c:
        if (sys_open("/dev/console", O_RDWR, 0) < 0)
                printk("Warning: unable to open an initial console.\n");
        (void) sys_dup(0);
        (void) sys_dup(0);

        if (execute_command) {
                run_init_process(execute_command);
                printk("Failed to execute %s.\n", execute_command);
        }
        run_init_process("/sbin/init");
        run_init_process("/etc/init");
        run_init_process("/bin/init");
        run_init_process("/bin/sh");
        panic("No init found.  Try passing init= option to kernel.");
```

init: la linea di comando del kernel

Il kernel riceve una linea di comando dal boot loader

- Talvolta la cmdline è compilata nel kernel stesso
- In ogni caso, viene recuperata da setup_arch(), chiamata da start_kernel()
  - Solo il codice di piattaforma sa dove recuperare i dati in RAM

Ogni argomento della linea di comando viene usato così:

- Se corrisponde ad una direttiva nota, viene usato
  - Alcune direttive sono sempre valide (mem=, quiet, ...)
  - Un driver può dichiarare le direttive di cui ha bisogno
- Se contiene il carattere "=", viene passato nell'environment del processo init
  - Si può quindi passare PATH=, LD_LIBRARY_PATH= o altro
- Gli argomenti rimanenti costituiscono argc/argv del processo init

Una volta invocato init, è tutto compito suo:

- Montare i dischi
- Attivare i servizi e le applicazioni
- Far partire e ripartire i processi
- Raccogliere i processi orfani
- Gestire lo shutdown del sistema

init: /etc/inittab

Il programma init può comportarsi in vari modi

- Si veda http://www.linux.it/kerneldocs/init/

Nella situazione usuale (SysVInit)

- Init legge /etc/inittab dove sono definite
  - I comandi da fare una volta sola
  - I comandi da far ripartire quando terminano
  - I comandi in risposta ad alcuni eventi particolari
- Ogni comando dipende dal "runlevel" in cui si è.
- /etc/rcN.d/ configura i runlevel
- Il comando "telinit" permette di comunicare con init:
  - Cambiare runlevel
  - Rileggere inittab

In un sistema embedded si usano approcci semplificati

- /etc/inittab in versione semplificata (es: busybox)
- /sbin/init come script per sh o un altro interprete

Il boot loader

Ruolo del boot loader

Il boot loader deve essere in grado di:

- Inizializzare i registri del controllore SDRAM
- Accedere alla memoria flash
- Leggere e scrivere su una porta seriale
- Saper caricare un file in memoria RAM

Un buon boot loader, è anche capace di:

- Scrivere in memoria flash
- Verificare l'integrità dei dati
- Fornire funzionalità di I/O basilare
- Trasferire dati su porta seriale con srecord/xmodem/uuencode
- Inizializzare la scheda di rete
- Ricevere e inviare pacchetti di rete
- Salvare una configurazione dell'utente
- Prevedere una gamma di configurazioni predefinite

Das U-boot

U-boot è un ottimo esempio di boot loader

- Gira su molte piattaforme
- È usato e sviluppato da molte persone
- Fa tutto quello che serve e anche di più

Struttura del codice di u-boot:

```
lib_$ARCH/
include/asm-$ARCH/
include/configs/$BOARD
board/$BOARD/
Makefile
MAKEALL
```

Per compilare

```
make ${BOARD}_config
make CROSS_COMPILE=$CROSS_COMPILE HOSTCC=$HOSTCC
```


Configurazione di u-boot

Il comando "make $DEVICE_config" chiama mkconfig

Lo script mkconfig riceve da make quattro parametri

- Il nome della configurazione (nome file in include/configs/)
- L'architettura (arm, ppc, ...)
- Il modello di processore
- Il nome della scheda

I file creati sono

- include/config.h (copiato da include/configs/$CFG.h)
- include/config.mk (generato in base agli argomenti
- link simbolici di piattaforma e processore

Lo script "MAKEALL" compila per tutte le macchine, salvando i messaggi di errore

Concetto di file immagine

U-boot carica ed esegue solo file correttamente incapsulati

- Questo vale sia via rete che da flash
- L'intestazione contiene:
  - tipo di file
  - tipo di compressione usata
  - sistema operativo di riferimento
  - nome simbolico
  - checksum

Lo strumento tools/mkimage è parte dei sorgenti di u-boot

La gestione dei file immagine è comoda:

- Si può sempre sapere cosa c'è sulla flash
- Ci si accorge subito degli errori
- Si può comprimere ciò che non è già oompresso di suo

La gestione dei file immagine è scomoda:

- Non si appoggia su un partizionamento della flash
- Occorre sempre avere mkimage a portata di mano
- Non si possono modificare al volo i file

Esempio: avvio a due stadi

In questo caso il primo u-boot ne carica un secondo, per permettere l'aggiornamento senza rischi

make bshdsl0_config

```
#define CONFIG_BOOTCOMMAND  "bootm 20000"
#define CFG_PROMPT          "0=> "
#define CFG_ENV_IS_NOWHERE
```

make bshdsl_config

```
#define CONFIG_BOOTCOMMAND  "bootm 60000 100000"
#define CONFIG_BOOTARGS     "console=ttyS0,115200 root=/dev/ram0"
#define CFG_PROMPT          "=> "
#define CFG_ENV_IS_IN_FLASH 1
#define CFG_ENV_ADDR        (PHYS_FLASH_1 + 0x40000)
#define CFG_ENV_SIZE        0x20000
```

Passaggio dei parametri al kernel

I parametri dal boot loader al kernel vengono passati sulla riga di comando

Su ARM, viene passata una tag-table

- Disposizione della memoria
- Dimensione di pagina
- Altro...

Su PowerPC, viene passata una struct "bdinfo"

- Frequenze di clock
- Altro...

Normalmente il kernel _non_ deve

- Inizializzare il controllore SDRAM
- Inizializzare i chip-select
- Cambiare la velocità della porta seriale

Il processore ARM

arm - Acorn Risc Machine

ARM e` un processore RISC con le seguenti caratteristiche:

- Architettura load-store
- 16 registri a 32 bit
- Istruzioni a tre operandi da 32 bit
- Esecuzione di una istruzione ogni ciclo macchina
- Bassi consumi di energia

Viene sviluppato da ARM Ltd. e licenziato a vari produttori

- ARM7: molto diffuso come microcontrollore, nommu
- StrongARM: DEC, poi Intel (ora fuori produzione)
- Xscale: Intel (es: 80200, PXA255, PXA270, IXP425)
- EP93xx: Cirrus Logic (es: 9302, 9315)
- iMX: Freescale (iMX1, iMX21, iMX31)
- AT91: Atmel
- Altri (molti altri)

http://en.wikipedia.org/wiki/ARM_architecture

arm - codice macchina

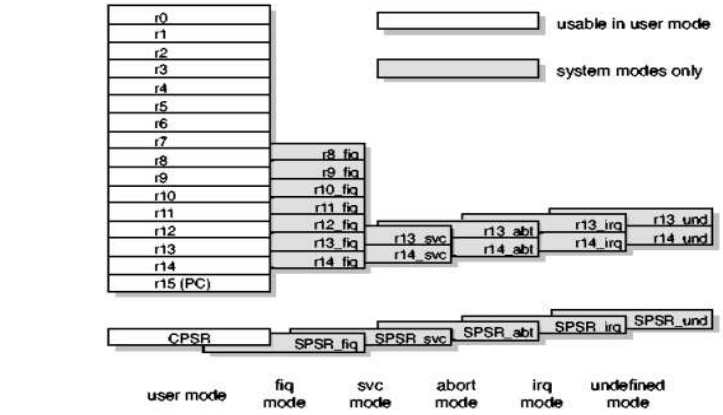
Caratteristiche importanti del codice macchina ARM

- L'accesso non allineato non e` consentito (RISC)
- Non esistono assegnamenti immediati a 32 bit (RISC)
- Non esiste stack pointer gestito via hardware (RISC)
- Istruzioni load-multiple e store-multiple
- Ogni istruzione e` ad esecuzione condizionale
- Assegnazione facoltativa dei bit di stato
- E` incluso uno shift register per uno degli operandi
- Tutti gli indirizzamenti sono relativi ad un registro

Caratteristiche peculiari di altri RISC, assenti in ARM

- Finestre di registri
- «Delay slot» per i salti
- Registro «zero»

arm - i registri



arm - la ABI standard, i coprocessori

r0	a1
r1	a2
r2	a3
r3	a4
r4	v1
r5	v2
r6	v3
r7	v4
r8	v5
r9	v6
r10	v7
r11	v8
r12	ip
r13	sp
r14	lr
r15	pc

Ruolo dei registri

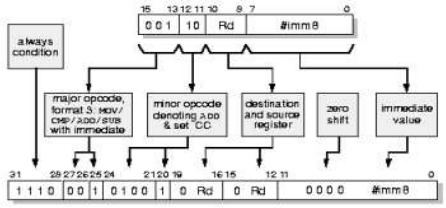
- I registri A sono gli argomenti a funzione
- I registri V vanno salvati prima di modificarli
- R12 e` "intra-procedure scratch"
- R13 e` lo stack pointer
- R14 e` il link register
- R15 e` lo stack pointer

I coprocessori

- L'architettura definisce 16 coprocessori
  - CP15 e` usato per cache e MMU (se presente)
  - CP0 e CP1 sono usate dalla FPU (se presente)
- Sono gia` definite le istruzioni:
  - Copia tra registri: MRC, MCR
  - Lettura e scrittura: LDC, STC
  - Operazione di coprocessore: CDP

arm - l'estensione thumb

Le istruzioni «thumb» sono istruzioni a 16 bit



- Compattezza del codice
- Accesso ad un sottoinsieme di registri
- Operazioni a due operandi

Le istruzioni BX e BLX servono a passare dall'uno all'altro

- Uso del bit 0 del program counter

L'avvio dei processori ARM

Avvio del core

Il vettore di reset (e avvio) di ARM è all'indirizzo 0

- A 0x0000.0000 si trovano gli 8 vettori ARM
  - reset, interruzioni, eccezioni, trap
- Non si tratta di vettori ma proprio di istruzioni
  - Il processore salta a 00 (04, 08, 0c, 10, 14, 18, 1c)
  - Normalmente l'istruzione (32 bit) è un salto relativo in avanti

Ogni implementazione sceglie cosa trovare a tale indirizzo

- Spesso è l'indirizzo della memoria flash parallela
  - In questo caso un piedino seleziona se accedere a 16 o 32 bit
- Per l'avvio da altre periferiche (NAND, USB, Ether) si usa una ROM
  - La ROM interna all'integrato gestisce il caricamento
  - Una volta caricato il codice lo esege in SRAM.
- Molti processori permettono di scambiare i banchi di indirizzi
  - All'avvio esegue sempre la ROM interna
  - In base a quanto trovato nel sistema la ROM commuta i blocchi

Gli otto vettori di avvio

I vettori del processore sono solo 8 e sono predefiniti

- Si trovano all'indirizzo 0 (0x00..0x1f)
- A posteriori possono essere spostati a 0xffff.f000

I vettori sono:

- reset
- undefined
- swirq
- prefetch abort
- data abort
- (reserved)
- irq
- fast irq

Esempio: u-boot/cpu/arm926ejs/start.S

```
_start:
        b       reset
        ldr     pc, _undefined_instruction
        ldr     pc, _software_interrupt
        ldr     pc, _pabt
        ldr     pc, _dabt
        nop
        ldr     pc, _irq
        ldr     pc, _fiq
_undefined_instruction:
        .word undefined_instruction
_software_interrupt:
        .word software_interrupt
_pabpt: .word prefetch_abort
_dabt:  .word data_abort
_irq:   .word irq
_fiq:   .word fiq
```


La ROM dell'AT91SAM92

Il codice in ROM svolge le seguenti funzioni

- Verifica se è presente ed è programmata la dataflash su SPI
- Se si, copia 4kB di codice in RAM interna e li esegue.
- Altrimenti, verifica se è presente ed è programmata la memoria NAND.
- Se si, copia 4kB di codice in RAM interna e li esegue.
- Altimenti, attende comandi dall'esterno
  - via porta seriale (debug unit)
  - via USB device

La verifica del contenuto della memoria seriale è brutale:

- Se le prime istruzioni sono salti relativi, allora è codice
- Se è codice, viene eseguito senza ulteriori controlli

Per ripristinare il sistema va disattivata la memoria seriale

Il codice del primo settore (AT91SAM92)

Il codice da eseguire dopo la ROM è disponibile qui:

`http://www.at91.com/repFichier/Project-209/at91bootstrap-2.3.tar.bz2`

Le funzioni da svolgere a questo punto sono:

- Inizializzare i PLL interni del processore
- Inizializzare i piedini di GPIO
- Inizializzare il controllore SDRAM
- (Ri)identificare la memoria esterna
- Copiare in RAM ulteriore codice (boot loader) ed eseguirlo

Lo spazio a disposizione non consente altro:

- Nessuna interazione utente
- Nessuna verifica della validità del boot loader

Avvio da NAND della famiglia iMX

Se l'avvio e` configurato da NAND (tramite pin-strap)
- La ROM interna (16k) carica 2kB nel buffer di trasferimento NAND
- Salta in tale buffer in modalita` ARM

Questo primo settore di NAND deve (come nell'AT91)
- Inizializzare i PLL interni del processore
- Inizializzare il controllore SDRAM
- Copiare in RAM ulteriore codice (boot loader) ed eseguirlo

Lo spazio a disposizione non consente altro:
- Nessuna interazione utente
- Nessuna verifica della validità del boot loader

Lo stadio successivo esegue in RAM (es: 0xc0f0.0000)

U-Boot per l'avvio da NAND

Dopo il primo settore di NAND (IPL), si esegue u-boot

Il codice deve essere compilato per eseguire da RAM
- #define CONFIG_SKIP_RELOCATE_UBOOT
- #define CONFIG_SKIP_LOWLEVEL_INIT
- TEXT_BASE = 0x80f00000

Il comando "nand" di u-boot permette di usare la flash
- nand read, nand write
- nand read.jffs2, nand write.jffs2
- nand erase, nand scrub

Per provare nuove istanze senza scriverle in flash
- Modificare TEXT_BASE in config.mk e include/configs/
- Caricare il nuovo u-boot ad un altro indirizzo (es: 0x80e0.0000)
- Usare il comando "go" di u-boot per entrare nel codice di prova

dev: Il ruolo del linker

Il linker (ld) deve solo appaiare simboli e indirizzi
- Un "simbolo" è solo un nome, senza alcun attributo
- Tutti i controlli su tipi e usi del simbolo sono già stati fatti
- Il linker, di conseguenza, non è specifico per il linguaggio C

Il comportamento del linker è definito da uno script
- La compilazione del kernel usa un ldscript specifico
- Il boot loader usa un ldscript specifico
- Il comportamento predefinito è in /usr/lib/ldscripts

Esempio: mimmo.c e mini.lds
Esempio: vmlinux.lds
Esempio: u-boot.lds

Inline assembly

Il gcc permette di scrivere codice assembly
Tale codice deve interagire con l'ottimizzatore
La sintassi è abbastanza impegnativa

```
asm("codice" : r-output : r-input : r-modificati);
```

La stringa "codice" non specifica i registri da usare
- Si usano nomi posizionali ("%0") o simbolici ("%[timeout]")

L'elenco dei registri (out e in) indica la corrispondenza C

L'elenco dei registri modificati comprende anche
- memory: la memoria esterna al processore è stata modificata
- cc: i "condition code" non sono preservati nella CPU

Esempi:

```
#define set_cr(x)                                       \
        __asm__ __volatile__(                           \
        "mcr    p15, 0, %0, c1, c0, 0   @ set CR"       \
        : : "r" (x) : "cc")
#define mb() __asm__ __volatile__ ("" : : : "memory")
```

La documentazione è nel manuale "info" di gcc