FOX Board BOOT

29 marzo 2008

Sommario

In questo articolo viene descritto il processo di boot di Linux sulla FOX Board 8+32. In altre parole si esamineranno i files contenuti nella directory `./arch/cris/boot` del kernel 2.6.15 fornito da AcmeSystem. Per capire esattamente la sequenza di boot verrà definita anche l'organizzazione della memoria, del partizionamento e conseguentemente della flash.

Indice

I ./arch/cris/boot **1**

1 Introduzione **2**

2 Indirizzamento, Memoria e Bootstrap **2**
2.1 Flash e RAM . . . 3
2.2 BootStrap . . . 4

3 BOOT della FOX board **4**
3.1 Resque System . . . 5
3.1.1 Analisi del flusso del codice . . . 9
3.2 Decompressione & Jump . . . 10
3.2.1 ../kernel/ . . . 13

II Partitioning **15**

4 Building `fimage` **15**

5 Verifica Partizioni **17**

III Appendici e Riferimenti **18**

A readmemory: verifica della flash **18**

B findinfile.c **21**

C LD: Sections And Relocation **22**
C.1 Sections . . . 22

Parte I
./arch/cris/boot

I file a cui si farà riferimento sono contenuti nella directory `./arch/cris/boot/` contenuta nel kernel-tree, ovvero nella directory `os/linux-2.6/` contenuta a sua volta nella root del software

della scheda FOX (tipicamente `devboard-R2_01/`). D'ora in avanti questa sarà, salvo esplicita dichiarazione, la directory di default da cui cercare i vari files.

1 Introduzione

Questo articolo articolo esamina l'architettura CRIS ossia "*Code Reduced Instruction Set*" progettata da AXIS Comunication (www.axis.com) per la realizzazione di processori RISC a 32 bit embedded orientati principalmente agli applicativi di rete e telecomunicazioni. La CPU in esame è la ETRAX100LX, dove LX indica proprio "Linux" in quanto è stata progettata pensando di dover eseguire questo sistema operativo; questo almeno a detta di AXIS.

Nel seguito si cercherà di documentare come avviene la fase di BOOT su schede di tipo FOX Board, che montano appunto i processori appena indicati. Il codice su cui ci si baserà è il kernel 2.6.15 distribuito da AcmeSystems ([1]) e la scheda in esame è una FOX8+32.

Nella directory *./asch/cris/* fornito da AcmeSystem sono in realtà presenti due architetture identificate dalle directory *arch-v10/* e *arch-v32/*. L'architettura che verrà qui esaminata sarà la v10 e ciò si traduce col fatto che alcune directory presenti sono in realtà del link simbolici:

arch → *arch-v10/*
boot → *arch-v10/boot/*
driver → *arch-v10/driver/*
lib → *arch-v10/lib/*

2 Indirizzamento, Memoria e Bootstrap

Tabella 1: Indirizzi di memoria (da [2])

Intervallo	*Size (Mb)*	*Descrizione*
00000000-03FFFFFF	64	EPROM/flash banco 0*
04000000-07FFFFFF	64	EPROM/flash banco 1*
08000000-0BFFFFFF	64	SDRAM banco 0*
0C000000-0FFFFFFF	64	SDRAM banco 1*
10000000-13FFFFFF	64	Peripheral chip-select 0*
14000000-17FFFFFF	64	Peripheral chip-select 1*†
18000000-1BFFFFFF	64	Peripheral chip-select 2*†
1C000000-1FFFFFFF	64	Peripheral chip-select 3*†
20000000-23FFFFFF	64	Peripheral chip-select 4*
24000000-27FFFFFF	64	Peripheral chip-select 5*†
28000000-2BFFFFFF	64	Peripheral chip-select 6*†
2C000000-2FFFFFFF	64	Peripheral chip-select 7*†
30000000-3FFFFFFF	256	Non usare‡
40000000-3FFFFFFF	1024	Interfaccia DRAM*
80000000-AFFFFFFF	768	Come 00000000-2FFFFFFF uncached
B0000000-B7FFFFFF	128	Registri interni
B8000000-BFFFFFFF	128	Indirizzo iniziale del codice
C0000000-FFFFFFFF	1024	Come 40000000-7FFFFFFF uncached

* : Aggiungere 80000000 per bypassare la cache.
† : Pheriperal chip-select 1-3 e 5-9 sono sono multiplexati con i pin 2-7 della porta PB e non sono disponibili se configurati come I/O generici.
‡ : Questa regione di memoria equivale ai Registri interni + Indirizzo iniziale del codice (B0000000-BFFFFFFF) con cache bypassata. Non usare per accedere ai registri.

Il microprocessore in esame non ha bisogno di un BIOS che esegua o fornisca le funzioni di basso livello e informi il micro in merito alla quantità di memoria del sistema. Questo processore ha già dei bus che gli permettono di collegarsi alla memoria montata sulla scheda.

Secondo quanto riportato da [2, cap. 5] il processore ha un bus dati a 32 bit che supporta anche memorie a 16 bit: l'organizzazione della memoria è *little-endian* ossia il LSB (*Least Significant Byte*) è posizionato all'indirizzo più basso. Vi è poi un bus di indirizzamento (esterno) a 26 bit per il controllo e configurazione di DRAM e SDRAM più 5 bit (leggasi pin) utilizzati come chip select, cioè usati per selezionare un particolare dispositivo, configurabili internamente. Il processore può poi essere collegato alla memoria esterna DRAM direttamente senza circuiti esterni di interfaccia con possibilità di settare la modalità sincrona o asincrona per la comunicazione.

In altre parole lo spazio di indirizzamento del processore è di 32 bit (ossia indirizza 4Gb), ma verso l'esterno il processore indirizza solo con i primi 0-25 pin ($2^{26} - 1$ =64Mb) un singolo banco di ram. Per indirizzare ulteriore ram si usano altri 5 bit che selezionano il banco di ram corrispondente; resta quindi un ultimo bit (31esimo) che indica se si sta bypassando la cache del processore oppure no, il che significa che se questo bit è a 0, il codice viene precaricato in cache. Lo spazio di indirizzamento è definito in tabella 1.

2.1 Flash e RAM

La flash contiene l'intero sistema, dal kernel ai dati, dai programmi allo spazio fisico in cui scrivere in modo persistente. Una prima idea di come è organizzata la flash si può avere semplicemente collegandosi alla FOX (tipicamente in telnet o ssh come descritto sul sito [1]):

```
[root@axis /root]246# mount
/dev/flash3 on / type cramfs (ro)
/dev/flash2 on /mnt/flash type jffs2 (rw,noatime)
proc on /proc type proc (rw,nodiratime)
tmpfs on /var type tmpfs (rw)
sysfs on /sys type sysfs (rw)
devpts on /dev/pts type devpts (rw)
none on /proc/bus/usb type usbfs (rw)
```

Da quanto si vede la FOX ha almeno due partizioni in flash. La prima è read-only e di tipo *cramfs*, mentre la seconda è in read/write ed è montata in */mnt/flash* ed è di tipo *jffs2*. Esaminando il sistema si vede che non è presente la directory boot o comunque non è presente un file immagine del kernel. Questo è legato al fatto che in fase di scrittura del file *fimage* (in *devboard-R2_01/* lanciando il comando *make*) viene compilato il codice necessario al caricamento del sistema. In particolare la parte iniziale del codice analizza la tabella delle partizioni; tale tabella indica come è organizzata e suddivisa la flash in cui vi sarà una partizione bootable tramite la quale sarà possibile avviare il kernel il quale è uno zImage scritto ad un indirizzo opportuno e noto in modo tale che venga decompresso e caricato in RAM.

Un volta eseguito il boot del sistema da flash (argomento che verrà approfondito in seguito), ovvero una volta che il kernel è caricato in DRAM e il sistema è visibile e accessibile, è possibile vedere le risorse a disposizione in termini di memoria tramite *cat /proc/meminfo*:

```
MemTotal:        30256 kB
MemFree:         22144 kB
Buffers:          1456 kB
Cached:           2696 kB
SwapCached:          0 kB
Active:           3544 kB
Inactive:         1816 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:        30256 kB
LowFree:         22144 kB
SwapTotal:           0 kB
```

```
SwapFree:            0 kB
Dirty:               0 kB
Writeback:           0 kB
Mapped:           2384 kB
Slab:             1480 kB
CommitLimit:     15128 kB
Committed_AS:     4712 kB
PageTables:        288 kB
VmallocTotal:   262144 kB
VmallocUsed:       448 kB
VmallocChunk:   261696 kB
```

Si ricorda inoltre che una volta che il sistema sarà partito, la memoria non sarà più indirizzata in modo fisico, ma utilizzando indirizzi virtuali tramite la MMU. Maggiori informazioni a riguardo si trovano in *../README.mm*.

2.2 BootStrap

Tabella 2: Modalità di boot-strap settando i pin bs1 e bs2.

bs2/bs1	*Metodo*	*Descrizione*
00	Normal	L'esecuzione comincia a 0x80000002 (flash con cache bypassata)
01	Serial	Boot via seriale 0 (9800bps, 8bit, no-parità, 1 bit di start e stop)
10	Network	Il Codice di boot-strap viene inviato via rete tramite un pacchetto SNI o MII
11	Parallel	Boot via parallela numero 0

La fase di bootstrap può avvenire in quattro modalità differenti. Indipendentemente dalla modalità scelta, la cache è sempre inizializzata. Mediante quattro pin (bs0, bs1, bs2, bs3) è possibile definire il valore nel registro R_BUS_STATUS, ma per definire esattamente quale metodo di boot-strap utilizzare, devono essere settati i soli bit (pin) bs1 e bs2. Le modalità di boot-strap, definite in tabella 2 (vedere anche [2, cap. 6]) sono:

00 - Normal Boot: L'esecuzione del codice parte all'indirizzo 0x80000002, ossia il secondo byte della flash caricato bypassando la cache. In altre parole viene eseguito il codice scritto in flash. Di ciò si parlerà in seguito in quanto riguarda il caricamento del kernel.

01 - Serial Boot: tramite la porta seriale 0 configurata come definito in tabella 2 verranno ricevuti 784 Byte che verranno copiati in cache all'indirizzo 0x380000F0, indirizzo da cui poi partirà l'esecuzione.

10 - Network Boot: Il codice di boot viene ricevuto attraverso un pacchetto ethernet SNI o MII. In [2] è definita la struttura del pacchetto dal quale viene ricevuto fino a un massimo di di 1484 byte caricati in cache. Il primo byte verrà scritto a 0x380000E6 e l'esecuzione comincerà a 0x380000F4.

11 - Parallel Boot: Tramite la porta parallela 0 vengono trasferiti 784 byte e scritti a 0x380000F0, indirizzo a cui comincerà anche l'esecuzione del codice ricevuto.

3 BOOT della FOX board

Il processo di boot della FOX Board avviene secondo quanto definito in precedenza nel caso di *Normal Boot*. Si noti che la scrittura nella flash avviene di default attraverso il *Network Boot* di cui comunque non si parlerà in questa sede. Analogamente a quanto definito nel caso dei sistemi x86 i files e directorys di interesse risiedono nella directory corrente:

- *boot*: Makefile
  - *compressed*: Makefile, README, decompress.ld, head.S, misc.c
  - *resque*: Makefile, head.S, kimagerescue.S, rescue.ld, testrescue.S
  - *tools*: build.c

La directory corrente definisce le istruzioni che devono essere eseguite in fase di boot per inizializzare la RAM, decomprimere, caricare ed eseguire il kernel. Si noti però che a differenza dei sistemi x86, i file sopra elencati non vengono compilati in fase di compilazione del kernel, ma in fase di *make* per generare l'intera immagine del sistema da scrivere nella flash.

3.1 Resque System

Contrariamente a quanto avviene nei sistemi x86, il codice subito eseguito dalla FOX non si trova nella directory *compressed/*, ma nella directory *resque/*. Il codice qui riportato (definito a partire da *head.S*) è contenuto nei primi 64k (primo settore) di flash ed analizza la tabella delle partizioni posta all'inizio del settore seguente. Il codice in questione (resque-code d'ora in poi) viene scritto all'inizio del primo banco di flash (indirizzo 0x00000000 oppure 0x80000000) in modo che il processore lo carichi subito e lo esegua in fase di boot.

Il compito del resque-code è controllare la tabella delle partizioni al primo settore subito dopo il resque-sector[1] (settore che contiene il resque-code). La tabella delle partizioni viene generata da uno script apposito che definisce offsets, lunghezze, tipi e checksum di ogni partizione che dovrà essere controllata.

Se uno qualsiasi dei checksum fallisce, si assume che la flash sia corrotta e quindi inutilizzabile per effettuare il boot. Viene allora configurata la porta seriale in modo da ricevere un flash-loader e una nuova immagine della flash. Via seriale viene quindi ottenuto il programma (flash-code) per effettuare il caricamento della flash che viene posto in cache ed eseguito subito dopo.

La tabella delle partizioni è progettata per essere trasparente all'esecuzione; contiene infatti delle piccole parti di codice nella parte iniziale che permette di saltare incondizionatamente in modo da evitare di "eseguire la tabella delle partizioni". In questa prima parte possono essere aggiunte (se necessario) altre istruzioni. Il formato della partizione è il seguente (offset - size):

- Area di "codice trsparente" (*Transparency Code*):

`00` - 2 Bytes	opcode `nop`
`02` - 2 Bytes	opcode `di`
`04` - 4 Bytes	opcode `ba <offset>` (versione 8 o 16 bit)
`08` - 2 Bytes	opcode `nop`

- Validazione (*Table Validation*):

`0A` - 2 Bytes	Numero magico e di versione per la tabella delle partizioni: 0xEF, 0xBE
`0C` - 2 Bytes	Lunghezza di ogni *Entry* più il marcatore finale (*End Marker*)
`0E` - 4 Bytes	Checksum delle *Entry* (e quindi della tabella delle partizioni)

- Formato delle *Entries*:

`12` - 4 Bytes	Offset in bytes della partizione a partire dall'inizio della flash
`16` - 4 Bytes	Lunghezza in bytes della partizione
`1A` - 4 Bytes	Checksum della partizione, semplice somma
`1E` - 2 Bytes	Tipo di partizione
`20` - 2 Bytes	Flags. Il bit 0 (unico usato) indica ro/rw (1/0)
`22` - 16 Bytes	Riservato per usi futuri

[1] Ogni settore è convenzionalmente definito di dimensione pari a 65536 Byte.

- Marcatore finale (*End Marker*):

  `32` - 4 Bytes [-1]
  `36` - 16 Bytes [0, padding]

Il bit 0 dei flags indica se la partizione è read-only o read-write. Tale bit permette di caricare la partizione in read-only, ma, se necessario, sarà possibile mascherare questo bit (o addirittura sovrascriverlo) in modo che il sistema possa poi usare la partizione anche in scrittura.

resque/resque.ld Prima di passare all'esame del codice vero e proprio è utile notare come il sistema procede ad assegnare gli indirizzi del codice. Questi indirizzi sono definiti nello script del linker (*resque.ld* appunto):

```
MEMORY
      {
      flash : ORIGIN = 0x00000000,
              LENGTH = 0x00100000
      }
SECTIONS
{
      .text :
      {
              stext = . ;
              *(.text) etext = . ;
      } > flash
      .data :
      {
              *(.data) edata = . ;
      } > flash
}
```

Questo script definisce dove le sezioni `.text` e `.data` del codice (vedere in seguito) devono essere scritte nella memoria `flash`. In particolare `flash` punta all'indirizzo 0x00000000 ossia al primo banco di flash montate sulla scheda, ossia al primo indirizzo a cui il microprocessore punterà in fase di *normal-boot*[2]. La lunghezza è 0x100000, ossia 64 MB; tale dimensione è solo ipotetica pari alla massima possibile in quanto non è detto che sia connessa una flash di questo tipo. Nella FOX in esame per esempio la flash è di 16 MB.

resque/head.S Contiene il codice vero e proprio contenuto a partire dal primo byte del primo banco di flash che procede al controllo delle partizioni. Per la descrizione dettagliata delle istruzioni assembler si rimanda a [3].

Tra le varie definizioni iniziali si osservi:

```
#define PTABLE_START CONFIG_ETRAX_PTABLE_SECTOR ; 65536
#define PTABLE_MAGIC 0xbeef     ; numero magico
#define CODE_START 0x40000000   ; indirizzo (DRAM) a cui verrà messo il codice
#define CODE_LENGTH 784         ; buffer del Serial-Boot
#define NOP_DI 0xf025050f       ; opcode delle istruzioni nop e di
#define RAM_INIT_MAGIC 0x56902387
```

Segue quindi la prima sezione di codice definita `.text`; questo è l'entry-point (punto di ingresso) del resque-code ed è pari a 0x80000000 se caricato dalla flash, come dovrebbe essere, bypassando la cache. Dalle specifiche del processore si legge che il boot viene eseguito all'indirizzo 0x80000002, quindi i primi due byte del codice vengono settati pari all'opcode di `nop` (no-operation) che occupa giusto 2 byte:

[2] In realtà il micro punta a 0x8000000 che equivale fisicamente allo stesso indirizzo di 0x00000000, ma senza l'uso della cache.

```
  nop    ; entry: qui siamo a 0x80000000
  di     ; qui a 0x80000002 - disabilita gli interrupt
  jump in_cache ; entra nell'area di cache ?!?
in_cache:
jtcd:
  move.d [jumptarget], $r0 ; 4 byte puntati da jumptarget in r0
  cmp.d 0xffffffff, $r0    ; r0 = -1 ?
  beq no_newjump           ; Se Sì, salta
  nop                      ; Se No, prosegui
  jump [$r0]               ; Salta all'indirizzo puntato da r0
jumptarget: .dword 0xffffffff ; sovrascrivibile in seguito
no_newjump:
#ifdef CONFIG_ETRAX_ETHERNET
  move.d 0x3, $r0    ; enable = on, phy = mii_clk
  move.d $r0, [R_NETWORK_GEN_CONFIG] ; registro per la configurazione di rete
#endif
#include "../../lib/dram_init.S" ; inizializzazione della DRAM
```

In questo codice l'unica cosa poco chiara è come si possa con un jump all'inizio saltare all'interno della cache, la quale dovrebbe essere bypassata visto che l'indirizzamento della flash a questo punto ha il 32-esimo bit a 1. Ad ogni modo in queste prime righe viene semplicemente verificato che `jumptarget` sia -1. Questo valore può essere sovrascritto in seguito per dirottare l'esecuzione del codice. Infatti se il valore non è -1 si assume essere questo valore un indirizzo valido di codice a cui saltare e si procede (`jump [$r0]`). In caso contrario se la configurazione del kernel lo prevede viene configurata l'interfaccia di rete e subito dopo obbligatoriamente la RAM.

Si procede con l'analisi della tabella delle partizioni e del relativo checksum:

```
  move.d PTABLE_START, $r3 ; carica l'indirizzo della partizione
  move.d [$r3], $r0   ; copia i primi 4 byte da 0x10000 in r0
  cmp.d NOP_DI, $r0   ; "nop" e "di" sono i primi valori ?
  bne do_rescue       ; No, allora salta
  nop                 ; Si, continua
  addq 10, $r3        ; salta il Transparency code sommando 10 byte
  move.w [$r3+], $r0  ; 2 byte da 0x1000A a r0, r3=0x1000C
  cmp.w PTABLE_MAGIC, $r0 ; r0=0xBEEF ?
  bne do_rescue       ; No, procedi al recupero di una immagine!
  nop                 ; Si, continua
  movu.w [$r3+], $r2  ; r2=lunghezza della Entry, r3=0x1000E
  move.d $r2, $r8     ; r8 = r2
  addq 28, $r8        ; r8+=28
  move.d [$r3+], $r4  ; r4=ptable checksum, r3=0x10012
  move.d $r3, $r1     ; r1=r3=puntatore alla Entry
  jsr checksum        ; jump-to-subroutine: r1, r2 input, r0 output
  cmp.d $r0, $r4      ; checksum calcolato = checksum letto ?
  bne do_rescue       ; No, salta
  nop                 ; Sì, tutte le Entry validate; continua
```

Si noti che a questo punto `r8` contiene la lunghezza del segmento che contiene le *Entries* più 28 ossia la lunghezza di *Transparency Code* più *End Marker*.

Se tutte le Entry sono validate si procede con l'analisi della singola Entry (e quindi della partizione relativa):

```
  moveq -1, $r7
ploop:
  move.d [$r3+], $r1 ; r1=offset dal PTABLE_START, r3+=4
  bne notfirst       ; se 0 è la prima partizione...
  nop                ;...e quindi continua
  move.d $r8, $r1    ; r1 = r8
  move.d [$r3+], $r2 ; r2 = lunghezza partizione, r3=0x1001A
  sub.d $r8, $r2     ; r2-=r8=offset della partizione di boot
  ba bosse           ; salta incondizionatamente
```

```
  nop
notfirst:
  cmp.d -1, $r1       ; fine della partition table?
  beq flash_ok        ; Sì, allora la flash è validata!
  move.d [$r3+], $r2 ; No, r2=lunghezza partizione, r3+=4
bosse:
  move.d [$r3+], $r5 ; r5=checksum della partizione, r3+=4
  move.d [$r3+], $r4 ; type e flags in r4, r3+=4=0x10022
  addq 16, $r3        ; salta i 16 byte riservati
  btstq 16, $r4       ; ro flag=0? o meglio: (16bit di r4)=0?
  bpl ploop           ; No, skip validation (salta a ploop)
  nop
  btstq 17, $r4       ; è una partizione bootable?
  bpl 1f              ; No, allora salta avanti a 1f
  nop
  move.d $r1, $r7     ; salva il boot partition offset in r7
1:
  add.d PTABLE_START, $r1 ; r1+=0x10000 (vero indirizzo part.)
  jsr checksum        ; checksum della partizione
  cmp.d $r0, $r5      ; checksum calcolato = salvato ?
  beq ploop           ; Sì, procedi alla prossima Entry
  nop                 ; No, procedi al resque di un'immagine
```

Ora, ammettendo che nel codice appena descritto non vi siano errori, se viene trovata una partizione valida da cui effettuare il boot, si salta a `flash_ok`:

```
flash_ok:
  cmp.d -1, $r7 ; r7 o contiene -1 o l'offset della partizione?
  bne 1f        ; No, salta
  nop           ; Sì, allora...
  move.d PTABLE_START, $r7   ;...r7 = 0x100000
1:
  move.d RAM_INIT_MAGIC, $r8 ; r8=0x56902387 servirà dopo
  jump $r7      ; boot!
```

Supponendo per esempio che in `r7` vi sia `PTABLE_START` (come normalmente dovrebbe essere sulla scheda FOX con la configurazione di default), il sistema effettuerà un jump all'indirizzo della partizione dove troverà le famose tre istruzioni `nop - di - ba <offset>`, ossia disabilita gli interrupt e salta incondizionatamente all'indirizzo `<offset>`. Già fin d'ora è presumibile che tale indirizzo sia, se la partizione è bootabile, il codice di decompressione e caricamento del kernel (*compressed/head.S*).

Altre funzioni quali `checksum` e `do_resque` non verranno analizzate nello specifico; basti però sapere che la prima esegue una semplice somma verificando a quale banco si sta puntando nella lettura della memoria flash. La seconda funzione invece è più complessa, ma non fa altro che configurare la seriale a 115.2 kbaud e si pone in attesa di 784 (`CODE_LENGTH`) bytes da caricare in cache. Successivamente si salterà all'indirizzo a cui il programma appena ricevuto è stato messo. Tale programma si pone poi in attesa di una immagine da scrivere nella RAM a 0x40000000 (CODE_START).

Nota: l'istruzione "`moveq -1, $r7`" serve fondamentale perchè se non dovesse essere chiaro quale sia l'offset della partizione bootable (per esempio se tale partizione non ci fosse), il sistema farebbe eseguire il codice a partire dalla tabella delle partizioni ossia da PTABLE_STRT (0x10000) come definito dal codice `flash_ok`.

resque/kimageresque.S Questo breve files ha del codice (`.text` quindi all'indirizzo 0x00000000) che permette di effettuare il processo di resque dell'immagine o comunque di codice eseguibile. Il codice riportato è pressochè identico a quanto definito in head.S per la funzione `do_resque`, con la differenza che il codice ricevuto viene copiato in RAM all'indirizzo `CODE_START` = 0x40004000.

resque/testrescue.S È semplicemente un programma di test da downloadare al microprocessore per testare la procedura di resque alternativa al boot. Il programma in questione accende azzera tutti i pin della porta A configurati come output (e quindi sulla FOX accende tutti i LED) entrando poi in un ciclo infinito:

```
#define ASSEMBLER_MACROS_ONLY
#include <asm/sv_addr_ag.h>
        .text
        nop
        nop
        moveq   -1, $r2
        move.b  $r2, [R_PORT_PA_DIR]
        moveq   0, $r2
        move.b  $r2, [R_PORT_PA_DATA]
endless:
        nop
        ba      endless
        nop
```

3.1.1 Analisi del flusso del codice

Per verificare se il codice definito in *resque/head.S* sia corretto e robusto per le varie possibilità di funzionamento, si supponga di avere due partizioni in flash: la prima è RO e bootabile, mentre la seconda è RW e non bootabile. Ne deriva l'ipotetica struttura della tabella delle partizioni seguente:

Trasparency Code	offset 0x00
Table Validation	offset 0x0A
Entry - partizione 1 - RO/bootable	offset 0x12
Entry - partizione 2 - RW/normal	offset 0x32
End Marker	offset 0x52

Ora pensando di essere appena prima del codice puntato da `ploop` si avrà `r3`=0x10012 e proseguendo da `ploop` in avanti dovrebbe accadere questo:

- si stà esaminando la prima Entry, quindi `r1`=offset da `PTABLE_START`=0; `r3` viene incrementato subito di 4 byte, `r3`=0x10016
- se è stato mosso un valore pari a 0 in `r1`, allora il flag `Z` è a 1, ossia il sistema riconosce questa come prima partizione (come è giusto che sia); il codice continua normalmente
- in `r2`=lunghezza partizione=x (il valore non interessa ora), `r3`+=4=0x1001A e si salta incondizionatamente a `bosse`
- a `bosse` vengono caricati checksum in `r4`, tipo di partizione e flag di boot in `r5` portando `r3`+=8=0x10022
- ora vengono aggiunti 16 (byte) all'indirizzo contenuto in `r3` portandolo a 0x10032 in modo da evitare l'area in cui non vi sono dati utili; questo indirizzo punta a End Marker (vedi sezione 3.1)
- vengono poi controllati se tutti i 16 bit di flag ro/rw (word bassa di `r5`) sono 0, ma il primo bit dovrebbe esse 1 per le ipotesi fatte in precedenza quindi si procede
- si controlla il flag di boot (primo bit della word alta di `r5`, o diciassettesimo bit della double-word `r5`) e per le ipotesi fatte questo flag è 1 quindi si procede copiando in `r7` il valore dell'offset della partizione contenuto in `r1`
- si procede salvando `r1` (offset) in `r7` e sommando a `r1` l'inizio della tabella delle partizioni, ma era 0 prima quindi ora `r1`=0x100000, alla chiamata del checksum e si ritorna a `ploop`

- ora in `r1` andrà il valore contenuto a 0x10032 (dword del'offset della seconda partizione), quindi il sistema si aspetta di esaminare una partizione diversa dalla prima (e infatti siamo alla seconda) e salta a `notfirst`
- a `notfirst` il sistema verifica nuovamente se `r1`=-1 (e non lo è): non salta a `flash_ok`, ma procede caricando in `r2` la lunghezza della seconda partizione e pone `r3`=0x1003A (ossia punta al checksum)
- quindi vengono caricati flag e tipo della partizione i quali daranno esito negativo e si salterà a ploop dove `r3` sarà pari a 0x10052 (ossia l'*End Marker*)
- a questo punto verrà copiato in `r1` il valore primo valore dall'*End Marker*; essendo `r1`=-1, si salta a `notfirst` dove, sempre perchè `r1`=-1, si assume la flash completamente validata e si salta a `flash_ok`
- qui, in `flash_ok`, essendo `r7` diverso da -1 si effettua il jump a questo indirizzo (oppure se pari a -1 allora salta a 0x10000)
- all'indirizzo `r7` (o 0x10000) vi sono le istruzioni `nop`, `di` e `ba` che permetteranno di eseguire il vero boot del sistema o comunque di saltare in un'area di codice valida

Il sistema sembra corretto, ma questa è la tipica condizione che di default è impostata nel kernel AXSIS. Per valutarne la robustezza si provi ad invertire i ruoli delle partizioni (ossia la seconda partizione sia RO e bootabile) si avrebbe il boot dalla partizione 2: si comincia con l'esame della prima partizione ed appena eseguito il controllo sui relativi flag che daranno esito negativo, si ritorna a `ploop` dove l'istruzione "`move.d [$r3+], $r1`" muove il primo valore contenuto nella Entry della partizione 2. A questo punto r1 è pari al nuovo offset (che sarà diverso da 0), quindi salta a `notfirst`, vede che `r1`!=-1 e si procede fino ad arrivare a soddisfare i flag di RO e Boot; questo implica che verrà salvato l'offset della partizione in `r7` ed eseguito il checksum della partizione, terminato il quale verrà nuovamente reimpostato `r1` pari a -1 (in quanto `[r3]` conterrà tale valore una volta tornati nuovamente all'indirizzo `ploop`). Questo stato dei registri farà si che il sistema, chiamando `flash_ok`, esegua il boot all'ultima partizione bootable trovata!

Si osservi poi che il checksum viene calcolato solo se la partizione è RO il che è sensato perchè la partizione RW cambia continuamente.

3.2 Decompressione & Jump

A questo punto occorre decomprimere il kernel ed eseguire il jump al codice effettivamente eseguibile.

Per i files assembler citati che ora verranno commentati va notata una cosa curiosa: il codice assembler non presenta caratteri particolari che definiscano che quanto scritto dopo l'opcode sia un registro, piuttosto che un indirizzo di memoria, ecc; per esempio il comando `move.d 0xA00,r0` copia il valore 0xA00 del registro general purpose `r0`, ma questo codice è grammaticalmente errato: se infatti si provasse a scriverlo in un programma (sia *user-space* che *kernel-space*) il compilatore restituirebbe un errore. La sintassi corretta è `move.d 0xA00,$r0` che è comunque diversa dai sistemi x86 che vogliono "`%`" davanti ai registri e "`$`" davanti ai valori costanti. Attualmente il perchè questo non dia un errore in compilazione è da chiarire.

compressed/decompress.ld Esaminando il codice di questa sotto directory appaiono nuovamente le sezioni `.text` e `.data`. È chiaro che non possono essere le stesse del caso precedente e infatti *decompress.ld* mostra come il codice in questa cartella deve essere composto:

```
OUTPUT_FORMAT(elf32-us-cris)
MEMORY
      {
      dram : ORIGIN = 0x40700000,
             LENGTH = 0x00100000
      }
```

```
SECTIONS
{
      .text :
      {
            _stext = . ;
            *(.text)
            *(.rodata)
            *(.rodata.*)
            _etext = . ;
      } > dram
      .data :
      {
            *(.data)
            _edata = . ;
      } > dram
      .bss :
      {
            *(.bss)
            _end = ALIGN( 0x10 ) ;
      } > dram
}
```

Come si legge, in questo caso si ha che la memoria indirizzata è la DRAM. L'indirizzo di inizio parte dai primi 7 MB e si estende per 1 MB.

compressed/head.S Teoricamente questo è il codice a cui corrisponde il jump dalla partizione bootable vista nella directory *resque/*. Il codice, da ciò che si evince da *compressed/decompress.ld*, è scritto per essere eseguito in RAM, ma inizialmente risiede in flash, quindi il registro di instruction pointer punta in flash:

```
.globl _input_data
.text
  nop  ; no operation
  di   ; disabilita gli interrupt
  cmp.d RAM_INIT_MAGIC, r8 ; già inizializzata la RAM?
  beq dram_init_finished   ; sì, allora salta
  nop                      ; no, allora inizializza la RAM
#include "../../lib/dram_init.S"
  dram_init_finished:
```

Si ricorda che il valore contenuto in `r8` viene settato dal codice `flash_ok` in *compressed/*. Si procede alla configurazione della DRAM confrontando RAM_INIT_MAGIC (0x56902387) con `r8` per vedere se la configurazione è già stata fatta, ma a questo punto `r8` dovrebbe essere 0. Vengono poi inizializzate le porte A e B; i valori delle macro sono ovviamente definiti dalla configurazione del kernel.

```
  move.b CONFIG_ETRAX_DEF_R_PORT_PA_DATA, r0
  move.b r0, [R_PORT_PA_DATA]
  move.b CONFIG_ETRAX_DEF_R_PORT_PA_DIR, r0
  move.b r0, [R_PORT_PA_DIR]
  move.b CONFIG_ETRAX_DEF_R_PORT_PB_DATA, r0
  move.b r0, [R_PORT_PB_DATA]
  move.b CONFIG_ETRAX_DEF_R_PORT_PB_DIR, r0
  move.b r0, [R_PORT_PB_DIR]
```

Segue il settaggio dello stack pointer ad un valore pari a 8 MB, considerato il minimo valore di RAM ammissibile per un kernel eLinux.

```
  move.d 0x40800000, sp
```

Ora si vuole spostare il codice del kernel compresso dalla flash alla DRAM. Per fare questo si rileva il valore del program counter, si toglie il bit 31 e si compensa l'istruzione sommando 2:

```
basse:
  move.d pc, r5     ; muove il program counter in r5
  and.d 0x7fffffff, r5 ; toglie il bit della cache
  subq 2, r5        ; aggiunge 2 a r5 per compensare move.d
  move.d r5, r0     ; indirizzo "basse" in flash
  add.d _edata, r5  ; r5 = "basse (flash)"+"_edata DRAM"
  sub.d basse, r5   ; r5 = r5 "basse DRAM"=indirizzo _edata in flash
```

Si osservi che a rigor di logica, anche se il codice viene caricato dalla flash, i vai indirizzi sono riconosciuti come indirizzi DRAM a causa del lavoro del linker: `basse` e `_edata` puntano a indirizzi compresi in 0x40700000÷0x407FFFFFF, mentre il program counter (`pc`) punta all'interno in flash. Questo implica che in `r5` vi sia l'indirizzo di `_edata` scritto in flash. Il codice viene quindi copiato in DRAM:

```
   move.d basse, r1  ; r1=basse=destinazione in DRAM
   move.d _edata, r2 ; r2=_edata= fine destinazione
1: move.w [r0+], r3  ; muove dalla flash una word, r0+=2
   move.w r3, [r1+]  ; muove la word in DRAM, r1+=2
   cmp.d r2, r1      ; indirizzo destinazione=_edata?
   bcs 1b            ; No, allora salta indietro
   nop
   move.d r5, [_input_data] ; for the decompressor
;; Clear the decompressors BSS (between _edata and _end)
   moveq 0, r0
   move.d _edata, r1
   move.d _end, r2
1: move.w r0, [r1+]
   cmp.d r2, r1
   bcs 1b
   nop
```

Il codice in questione viene copiato word per word in DRAM e successivamente viene azzerata l'area tra `_edata` e l'indirizzo `_end`. In questa area di memoria è l'area BSS la quale deve essere azzerata per definizione (vedere appendice 22). In `_input_data` viene posto il valore di `_edata` in flash. Seguirà quindi la chiamata a `_decompress_kernel` la quale ora dovrebbe risiedere in RAM e ritornerà in `_inptr` il valore del kernel compresso:

```
  jsr _decompress_kernel    ; definita in compressed/misc.c
  move.d [_input_data], r9 ; r9=indirizzo in flash del kernel compresso
  add.d [_inptr], r9        ; r9+=dimensione del kernel compresso
;; Restore command line magic and address.
  move.d _cmd_line_magic, $r10
  move.d [$r10], $r10
  move.d _cmd_line_addr, $r11
  move.d [$r11], $r11
;; Enter the decompressed kernel
  move.d RAM_INIT_MAGIC, r8 ; r8=0x56902387
  jump 0x40004000           ; salta nel kernel decompresso
.data
```

Il kernel decompresso si trova all'indirizzo 0x40004000 come definito in *compressed/misc.c* dalla macro `KERNEL_LOAD_ADDR`. A tale indirizzo dovrebbe trovarsi il codice di *../kernel/head.S*. Subito dopo l'istruzione di salto comincia l'area dati (`.data`) dove sono definite le due word `_cmd_line_magic` e `_cmd_line_addr`.

Nota: il registro `r9` contiene a questo punto l'indirizzo ai dati seguenti al kernel compresso, ossia punta a quello che più avanti verrà definito CRAMFS.

Nota: l'underscore ('_') prima di ogni label esterna (sia funzioni che variabili, per esempio `_decompress_kernel`) è necessario, a meno che non venga definito `.syntax register_prefix` nel file assembly oppure venga passata all'assemblatore o al compilatore l'opzione `--no-underscore`.

compressed/misc.c Qui sono implementate tutte le funzioni necessarie alla decompressione del kernel, in particolar modo `decompress_kernel()`. Come per i sistemi x86, in questa routine viene chiamata la funzione `gunzip()` implementata in `"../../../../../lib/inflate.c"`. `inptr` è l'indirizzo del singolo byte del buffer in ingresso alle funzioni di decompressione; alla fine del processo il suo valore contiene necessariamente la dimensione del buffer ossia del kernel compresso.

3.2.1 ../kernel/

Una volta eseguita la decompressione del kernel all'indirizzo 0x40004000 ed effettuato il salto a tale indirizzo, il codice a cui si punta è quello definito nella cartella *../kernel/* (*arch/cris/kernel* per l'esattezza). Il file principale è sempre *head.S*. Il codice a questo punto è in *supervisor-mode* (vedere [2, 3] per maggiori informazioni) in quanto l'unità di memory menagement (MMU) non è ancora stata configurata, cosa che viene fatta per prima da questo codice secondo la mappa descritta in *../README.mm*. Andando con ordine, prima di leggere il codice di *head.S* occorre tenere ben presente come la memoria viene organizzata in fase di linking. In quanto caso lo script del linker è al livello superiore della directory corrente (ossia di *../kernel/*) ed è *../vmlinux.lds.S*. Lo script è scritto tenendo conto della tabella MMU, infatti il codice parte dall'inidirizzo DRAM_VIRTUAL_BASE il quale in questo caso (*arch-v10*) è pari a 0xC0000000[3]; si noti che questo indirizzo è pari a quello fisico. Probabilmente è superfluo riportare questo codice, ma si tenga presente che tutti gli indirizzi definiti nel codice di *head.S* d'ora in avanti fanno riferimento (o sono definiti) in questo script.

Ora il sistema si trova ad un punto cruciale in cui deve individuare ed organizzare i vari filsystem presenti in flash. Come si è visto al paragrafo 2.1 il filesystem di root è di tipo CRAMFS, mentre la flash scrivibile (*/mnt/flash*) è in formato JFFS2. Rispettivamente queste due partizioni si appoggiano ai device driver */dev/flash3* e */dev/flash2* (vedere anche appendice A e parte II).

Quindi, dopo aver configurato la MMU, il sistema deve riorganizzare i vari segmenti del kernel posti in Falsh o in RAM a seconda di come è stato effettuato il loading. Si ricordi che in condizioni tipiche, la FOX a questo punto ha fatto il boot partendo sì dalla flash, ma caricando tutto il kernel in RAM. Quindi se tutto ciò è vero, ora l'esecuzione si trova in RAM. É importante sottolineare ciò perchè ora il codice si trova ad un punto in cui deve capire se l'esecuzione "di se stesso" proviene dalla RAM o dalla flash esaminando il program counter (pc). Ad ogni modo il sistema partirà in modalità un-cached per saltare successivamente alla modalità cached:

```
  move.d $pc,$r0
  and.d 0x7fffffff,$r0 ; get rid of the non-cache bit
  cmp.d 0x10000,$r0    ; just something above this code
  blo _inflash0
  nop
  jump _inram          ; enter cached ram
_inflash0:
  jump _inflash
  .section ".init.text", "ax"
_inflash:
```

Prima di proseguire ad esaminare i due casi si deve tenere presente che il kernel nel suo complesso è diviso in tre parti, ossia *kernel text*, *kernel data* e l'immagine del filesystem. Il linker

[3] Si noti che esaminando il file *Kconfig* relativo, sembra che l'indirizzo sia 0x60000000, ma se si osserva la linea di comando passata in fase di compilazione si osserva che in realtà l'indirizzo è 0xC0000000 in accordo con *README.mm* (vedere *../kernel/.vmlinux.lds.cmd*)

però a composto il kernel secondo la struttura finale di quando sarà caricato in memoria ossia *kernel text*, *kernel data* e *kernel BSS*, mentre il filesystem (ROM fs) risiede necessariamente e convenientemente in flash.

Si prosegue ora per completezza con l'analisi del primo caso; siccome questo non interessa il boot standard della FOX per evitare confusione si può saltare all'analisi del codice all'inidirizzo `_inram`. Se il sistema parte caricando il tutto dalla flash occorrerà copiare l'area text e data in DRAM; si procede con l'inizializzazione di ethernet, waitstates e, nuovamente, la RAM se necessario. A questo punto si è all'indirizzo `_dram_init_finished` ed ora arriva il vero processo di copia dei segmenti sopra citati:

```
   moveq   0, $r0             ; source
   move.d  text_start, $r1    ; destination
   move.d  __vmlinux_end, $r2 ; end destination
   move.d  $r2, $r4
   sub.d   $r1, $r4           ; r4=__vmlinux_end in flash
1: move.w  [$r0+], $r3
   move.w  $r3, [$r1+]
   cmp.d   $r2, $r1
   blo     1b
```

Il sistema in questo caso è in flash ed è già decompresso. Procedendo si verifica la presenza di un filesystem (CRAMFS) valido:

```
   moveq   0, $r0
   move.d  $r0, [romfs_length] ; 0 di default se manca cramfs
   move.d  [$r4], $r0          ; cramfs_super.magic
   cmp.d   CRAMFS_MAGIC, $r0   ; controllo il magic number
   bne     1f                  ; salta se non è cramfs valido
   nop
   move.d  [$r4 + 4], $r0      ; o leggi r0=cramfs_super.size
   move.d  $r0, [romfs_length] ; e salvala
#ifdef CONFIG_CRIS_LOW_MAP
   add.d   0x50000000, $r4 ; add flash start in virtual memory (cached)
#else
   add.d   0xf0000000, $r4 ; add flash start in virtual memory (cached)
#endif
   move.d  $r4, [romfs_start]  ; è anche l'indirizzo di start del filesystem
1: moveq   1, $r0
   move.d  $r0, [romfs_in_flash]
   jump    _start_it           ; salta all'inizializzazione (in cache)
```

Dal codice si capisce che ci si aspetta di trovare il filesystem subito dopo il kernel in quanto il registro `r4` punta inizialmente proprio all'indirizzo successivo a `__vmlinux_end` e viene poi adattato in funzione della tabella MMU. Infine si salta all'indirizzo `_start_it`.

Detto questo ora si esamina la tipica eventualità in cui il boot della FOX avviene, come descritto fino ad ora nelle precedenti sezioni, mediante decompressione e caricamento da pare del loader in RAM: allora l'esecuzione è posta all'indirizzo `_inram`. Il codice è strutturalmente simile al precedente con la differenza che ora è il registro `r9` che dovrebbe contenere l'indirizzo del filesystem CRAMFS. Tale valore era stato caricato dal loader (pag. 12):

```
  moveq   0, $r0
  move.d  $r0, [romfs_length]; 0 per default
  cmp.d   0x0ffffff8, $r9
  bhs     _no_romfs_in_flash ; r9 points outside the flash area
  nop
  move.d  [$r9], $r0         ; r9 valido: leggi cramfs_super.magic
  cmp.d   CRAMFS_MAGIC, $r0  ; cramfs valido?
  bne     _no_romfs_in_flash ; No, salta, altrimenti prosegui
  nop
  move.d  [$r9+4], $r0       ; carica cramfs_super.length
```

```
  move.d  $r0, [romfs_length]
#ifdef CONFIG_CRIS_LOW_MAP
  add.d   0x50000000, $r9 ; add flash start in virtual memory (cached)
#else
  add.d   0xf0000000, $r9 ; add flash start in virtual memory (cached)
#endif
  move.d  $r9, [romfs_start]
  moveq   1, $r0
  move.d  $r0, [romfs_in_flash]
  jump    _start_it
```

Viene quindi controllato il registro `r9` il quale, se non corretto, fa saltare l'esecuzione del programma a `_no_romfs_in_flash` che procederà alla ricerca di un filesystem valido; comunque normalmente l'esecuzione dovrebbe proseguire senza salti. A questo punto il sistema ha letto le due dword dopo il kernel compresso che indicano rispettivamente la sua lunghezza (`romfs_length`) e da dove comincia il filesystem ROM (`romfs_start`). Queste variabili sono globali e presenti nella sezione `.data` in fondo al file. Per quanto riguarda il codice relativo a `_no_romfs_in_flash` basti sapere che non fa altro che ricercare un CRAMFS valido andando a controllare direttamente l'indirizzo `_no_romfs_in_flash` definito dal linker-script dove dovrebbero essere contenuti i valori corretti che identificano tale filesystem. Infine si salta a `_start_it`.

All'indirizzo `_start_it` viene completata l'inizializzazione del sistema: per prima cosa si controlla se è presente una command line valida (e in caso contrario viene definita una "virtuale"), viene azzerata la regione BSS e si procede alla configurazione hardware di porte, DMA e device in genere in relazione alla configurazione del kernel. Finalmente si arriva all'istruzione `jump start_kernel` che effettua il salto alla omonima funzione C implementata in *init/main.c* nella root dei sorgenti del kernel. D'ora in avanti il codice è scritto in C.

Parte II
Partitioning

A questo punto è stato chiarito tutto il processo di boot del sistema, ma il sistema di partizionamento rimane ancora poco chiaro. Infatti non bisogna dimenticare che tutti i discorsi fatti vanno accompagnati sempre dall'organizzazione della flash che di per se trascende da quella che è la strategia di boot appena definita. In questa seconda parte si cercherà di definire più in dettaglio come è organizzata la flash, a cosa puntano i file di dispositivo che indicano le partizioni e come il tutto viene composto in fase di make. D'ora in avanti, se non diversamente specificato, la directory di riferimento sarà quella di default dei sorgenti dell'intero sistema (tipicamente `devboard-R2_01/`) e il kernel di riferimento sarà sempre il 2.6.15.

4 Building `fimage`

Come noto dalla documentazione reperibile da [1], una volata completata la fase di make nella directory root dei sorgenti, tra gli altri vengono generati i seguenti file:

```
devboard-R2_01] ls fimage flash?.* -al
-rw-r--r--  1 SkyNet root 8388632 2007-12-30 15:16 fimage
-rw-r--r--  1 SkyNet root 3473408 2007-12-30 15:16 flash1.img
-rw-r--r--  1 SkyNet root 4849664 2007-12-30 15:16 flash2.img
```

Il file `fimage` è il sistema completo da scrivere all'interno della flash. Come noto, questo file nasce dall'unione dei due file `flash1.img` e `flash2.img`, ma già fin d'ora si osserva che la somma dei soli due file immagine non è sufficiente la somma dei due. Inoltre Il file `fimage` è superiore

a 8Mb della flash di ben 24 byte (2^{23} = 8388608). Nel seguito quindi verrà esaminato come questo file viene generato.

La costruzione del file `fimage` avviene nel Makefile alla sezione *fimage* la quale richiede come parametri tre file. I parametri in questione sono i files relativi al resque code (in *os/linux-2.6/arch/cris/boot/rescue/*), l'immagine che contiene il kernel e il file system (fondamentale per un sistema UNIX anche embedded). Questi parametri vengono insieme ad altri all'inizio del Makefile nell'ordine in cui verranno assemblati:

```
PSIZE_rescue=0x010000       PSIZE_kernel=0x350000       PSIZE_jffs2_0=0x4A0000
PNAME_rescue=rescue         PNAME_kernel=flash1         PNAME_jffs2_0=flash2
PTYPE_rescue=rescue         PTYPE_kernel=kernel         PTYPE_jffs2_0=jffs2
PCSUM_rescue=yes            PCSUM_kernel=yes            PCSUM_jffs2_0=no
INAME_rescue=rescue.img     INAME_kernel=flash1.img     INAME_jffs2_0=flash2.img
```

Ciò che interessa maggiormente sono le variabili `INAME_*` e `PSIZE_*`. Il codice di resque ha una dimensione massima di un setore (64kbyte), flash1 (il kernel) 3.4Mbyte e il resto del sistema è settato a poco più di 4.7Mbyte. Per quanto riguarda il primo settore per il resque code, lo spazio è sovrabbondante visto che le dimensioni sono tipicamente di 664 byte. Secondo quanto detto nella prima parte dell'articolo questo codice deve risiedere in un settore e al settore successivo deve risiedere la tabelle delle partizioni. Presumibilmente la tabella delle partizioni risiederà in `flash1.img` insieme al kernel; vediamo quindi come vengono assemblate le varie sezioni.

Resque Code: `resque.img` Il resque code è definito nella sezione `$(INAME_rescue)` nella quale viene creato il file `resque.img` inizialmente compilando il contenuto di `resque.bin` e quindi, utilizzando una utility fornita con i tools della FOX (ossia `padflashimage` il quale richiede in ingresso la dimensione in esadecimale e il file da ridimenzionare) viene creato il file `resque.img` finale il quale conterrà nella parte successiva al codice di `resque.bin`, tutti byte di valore 0xFF.

Partition Table & kernel: `flash1.img`

```
$(INAME_kernel): vmlinuz $(INAME_cramfs_0) $(PTABLEFILE)
     [...]
     mkptable -a $(AXIS_TARGET_CPU) -v -f ptable.img $(PTABLEFILE)
     rm ptable_dummy.img
     cat ptable.img vmlinuz $(INAME_cramfs_0) > $@
     padflashimage $(PSIZE_kernel) $@
```

Quando `$(INAME_kernel)` viene eseguito richiede la presenza del file `vmlinuz`, `INAME_cramfs_0` (definito all'inizio del Makefile e pari a `rootfs.img`) e il file `ptablespec` che contiene fondamentalmente nomi e dimensioni dei tre files immagine prima citati: è una specie di file fstab. Tramite il comando *mkptable* viene generata la tabella delle partizioni con i relativi chechsum per ogni file di partizione definito in `ptablespec` (si noti che per 3 file la tabella è di 128 byte). Quindi viene generata una prima versione di `flash1.img` scrivendo in sequenza la tabella delle partizioni, il kernel e `rootfs.img`. Infine viene sistemata la dimensione del file tramite `padflashimage`. Si noti che il file `$(INAME_cramfs_0)` viene creato sempre nel *Makefile*. La sua generazione è una cosa abbastanza complessa, ma in definitiva non fa altro che creare un ambiente base UNIX-like dove trovare tutti i programmi e script di configurazione.

Filesystem scrivibile jffs2: `flash2.img` Questo viene generato semplicemente con il comando *mkfs.jffs2* (journaled flash filesystem 2) impostandone la dimensione a `PSIZE_jffs2_0`.

Immagine totale: `fimage` Infine si costruisce l'immagine `fimage`. Per poterlo fare occorre che le dipendenze precedentemente descritte siano soddisfatte:

```
fimage:  $(INAME_rescue) $(INAME_kernel) $(INAME_jffs2_0)
```

i tra files vengono composti, nell'ordine appena dichiarato, con il comando *cat*:

```
cat $^ > $@
```

ottenendo un primo file fimage di dimensione 0x800000 (8.388.608 byte). Questa dimensione è esattamente quella della flash. Si procede quindi con la generazione dell'hardware ID `HWID`.

```
HWID=$(HWID) ; \
if test "`echo -n $$HWID | wc -c`" -gt 8 ; then \
        echo "Hardware ID is too wide (max 8 bytes)!" ; \
        rm $@ ; \
        exit 1 ; \
fi ; \
echo "Adding hardware ID \"$(HWID)\" to $@" ; \
printf "%-8s" "$$HWID" >> $@ ; \
```

L'hardware ID è definito come 1.0 all'inizio del *Makefile*. Nonostante ciò viene controllato che non sia una stringa di più di 8 byte. Se tutto va bene viene aggiunto in fondo al file `fimage` una stringa di 8 caratteri (tramite la `printf`). Segue il calcolo della "somma" del file appena ottenuto e appeso nuovamente in fondo al file:

```
@( CSUM=`imgsum $@` ; \
   echo "Adding checksum \"`expr $$CSUM`\" to $@" ; \
   echo -n "$$CSUM" >> $@ ; )
```

Questo risultato è un valore decimale scritto in una stringa a 16 byte[4]: 16+8=24 ossia esattamente quanto occorre per avere la dimansione finale del file fimage a 8.388.632 byte. Quest'ultima parte serve "solo" per il boot via FTP o HTTP.

5 Verifica Partizioni

A questo punto si vuole capire dove e a cosa puntano i device */dev/flash* del kernel. A tal proposito sono stati sviluppati due programmi per effettuare alcuni test descritti nele appendici A e B. Il primo è un modulo il quale permette di andare a leggere determinate aree di memoria tramite accesso a proc del kernel della FOX. Il secondo programma invece permette di trovare un file incapsulato in un altro e di visualizzare a quale byte si trova.

Così facendo avremo l'immagine dei tre dispositivi. Ora volendo accedere in modo diretto e assoluto alla memoria della FOX, occorre collegarsi via *telnet* e, caricando il modulo `readmemory`, occorre passare l'indirizzo di memoria a cui si vuole accedere. Di default il modulo darà l'accesso solo ai primi 10k di memoria, ma è possibile far variare il parametro in fase di caricamento. Ma quale indirizzo passare? Chiaramente dipende da ciò che si vuole leggere, ma l'importante è non passare i valori fisici di memoria in quanto, in run time, l'unità MMU è configurata. Quindi in funzione di ciò che si vuole leggere occorre consultare il file *os/linux-2.6/arch/cris/arch-v10/README.mm*.

Inizialmente carichiamolo senza parametri con il comando `insmod readmemory`. Il modulo per default punto a 0xE0000000 e permetterà di leggere solo 64k. Da *README.mm* si osserva che la mappatura virtuale della memoria è tale per cui questo indirizzo punta sì all'inizio della chache, ma in modalità uncached. Per quanto detto nelle sezioni precedenti questo indirizzo è esattamente l'inizio a cui è contenuto il codice `resque.img` (pari al codice iniziale di `fimage`). Per verificarlo, occorre connettersi in FTP e scaricare il file */proc/readmemory*; questo file sarà da 65532 byte. A questo punto dovranno essere uguali i risultati di:

```
md5sum readmemory
dd if=fimage bs=65535 count=1 fimage | md5sum
```

[4]Verifica: `imgsum fimage | wc -c`

Nel paragrafo 4 si è detto che la parte iniziale di `flash1.img` è la tabella delle partizioni seguita dal codice di decompressione `decompress.bin`. Per verificarlo è possibile usare *findinfile* descritto nella sezione B il quale richiede in ingresso il "file contenuto" nel secondo file ("contenitore"). Verrà restituita la posizione a cui si trova il file. Lanciando:

```
findinfile /path/to/decompress.bin flash1.img     si otterrà 97
findinfile /path/to/decompress.bin fimage         si otterrà 65633
```

Chiaramente `ptable.img` occupa 96 byte. Tornando all'utilizzo del modulo, se volessimo leggere partendo dal primo byte la flash occorrerebbe caricare il modulo con:

```
insmod readmemory begin_segment=0xF0000000
```

Collegandosi nuovamente in FTP e scaricando */proc/readmemory*, questa volta il codice equivalente sarà pari ai primi 64k del file `fimage`. In particolare i primi byte (tipicamente i primi 664) sono il codice di `resque.bin`, verificabile con l'uguaglianza dell'output di *md5sum*:

```
md5sum /path/to/resque.bin
dd if=fimage bs=664 count=1 fimage | md5sum
```

Ora però rimane da chiedersi a cosa puntano i vari dispositivi `/dev/flash`*i*. Eseguendo operazioni simili a quelle appena indicate, ossia collegandosi in FTP, spostarsi in */dev/* e fare un *get* dei file `flash0`, `flash1` e `flash2` si verifica che la vera tabella delle partizioni è:

/dev/flash0 partizione da 64k nella quale vi è il codice `resque.bin`

/dev/flash1 dispositivo che indirizza `decompress.bin`, kernel e CRAMFS, cioè `flash1.img`

/dev/flash2 partizione JFFS2 per lo storage persistente dei dati, cioè `flash2.img`

/dev/flash3 partizione di root formattata CRAMFS

Per vedere esattamente come il chernel gestisce le partizioni gasta leggere il file */var/log/message* nel qale comparirà:

```
 Found a valid partition table at 0xf001000a-0xf0010056.
  /dev/flash1 at 0x00010000, size 0x00350000
  /dev/flash2 at 0x00360000, size 0x004a0000
 Adding readonly flash partition for romfs image:
  /dev/flash3 at 0x000e3628, size 0x00252000
Creating 4 MTD partitions on "cse0":
0x00000000-0x00010000 : "part0"
0x00010000-0x00360000 : "part1"
0x00360000-0x00800000 : "part2"
0x000e3628-0x00335628 : "romfs"
```

Parte III
Appendici e Riferimenti

A READMEMORY: verifica della flash

Nella sezione 2.1 si è visto come la flash sia in realtà divisa in almeno quattro dispositivi: in */dev* sono presenti addirittura sei dispositivi *flash{0,1,2,3,4,5}*. Per quanto detto nell'articolo è ragionevole pensare che il primo dispositivo faccia riferimento al codice di boot (resque-code) o comunque faccia riferimento alla prima parte della flash. Per verificare ciò si può leggere il codice sorgente, o più semplicemente si può usare il modulo readmemory seguente:

```
#include <linux/module.h>
#include <linux/proc_fs.h> //proc interface
#include <asm/uaccess.h>   //copy_from/to_user, access_ok, ecc
#include <asm/semaphore.h> //semaphore structure
MODULE_LICENSE("DUAL GPL/BSD");
MODULE_VERSION("0.1");
/** Parametri del modulo **/
static unsigned int begin_segment = 0xE0000000; // FLASH per MMU
static unsigned int segment_size  = 65536;      // byte da leggere
module_param(begin_segment, uint, S_IRUGO|S_IWUSR);
module_param(segment_size,  uint, S_IRUGO|S_IWUSR);
MODULE_PARM_DESC(begin_segment,"Indirizzo di inizio da cui leggere");
MODULE_PARM_DESC(segment_size, "Dimensione del segmento da leggere");
#define MAX_PHISICAL_FLASH_ONBOARD 0x00800000 // 8 Mb per FOX 8-32
#define MAX_PHISICAL_FLASH_INDEX   0x007FFFFF // 0x00800000 - 1
#define BEGIN_MMU_USEFUL_INDEX     0xC0000000 // see MMU map
#define MAX_SAFETY_DIMENSION       0x00800000 // 8 Mb
#define MAX_MEMORY_INDEX           0xFFFFFFFF
struct semaphore sem; //mutual exclusion semaphore
                      //evita le "Concurrency conditions"
//Strutture relative a /proc filesystem
struct proc_dir_entry *proc_memory_file = NULL;
#define PROC_MEMORY_FILE  "readmemory"
/** puntatore alla memoria **/
static void *readmemoryptr=NULL; //inizialmente 0 per precauzione
```

Funzione di lettura del file nel filesystema */proc*:

```
int readmemory_procread(char *buf, char **start, off_t offset,
                        int count, int *eof, void *data) {
   int len=offset,i;
   char *flashdata;

   if(readmemoryptr == NULL)
     return -ENOMEM;

   flashdata = readmemoryptr;

   if(down_interruptible(&sem))
     return -ERESTARTSYS;

   if(offset >= segment_size) {
        *eof = 1; // siamo alla fine del buffer
        goto exit_func;
     }

   if(count + offset >= segment_size)
     count = segment_size - offset;

   *start=buf; //necessario per leggere in più blocchi
   for(i=0; i<count; i++)
     *(*start+i) = flashdata[len++];

   exit_func:
   up(&sem);
   return count;
}
```

Cleanup e Initialization del modulo:

```
/** Funzione di scaricamento del modulo **/
void readmemory_cleanup_module(void) {
   readmemoryptr = NULL;
```

```
    if(proc_memory_file != NULL)
      remove_proc_entry(PROC_MEMORY_FILE, NULL); //file e parent dir

    printk(KERN_ALERT PROC_MEMORY_FILE" scaricato\n");
}
/** Funzione di inizializzazione del modulo **/
int readmemory_init_module(void) {
    //Inizializzazione dei semafori
    init_MUTEX(&sem);
    readmemoryptr =  (void*)begin_segment;
    if( begin_segment < BEGIN_MMU_USEFUL_INDEX )
      printk(KERN_ALERT "WORNING: indirizzo di partenza %X < %X\n",
             begin_segment, BEGIN_MMU_USEFUL_INDEX);
    else
      printk(KERN_ALERT "Accesso alla memoria a partire da %X\n",
             begin_segment);
    if( MAX_MEMORY_INDEX - segment_size < begin_segment )
      printk(KERN_ALERT "WORNING: inizio+dimensione > %X\n"
                        "  segment_size = %X",
             MAX_MEMORY_INDEX,
             segment_size = MAX_MEMORY_INDEX-begin_segment);
    else
      printk(KERN_ALERT "Dimensione del buffer: %X\n",
             segment_size);

    //Creazione del file virtuale
    proc_memory_file = create_proc_read_entry(PROC_MEMORY_FILE,
                                              0, //protection mask
                                              NULL,
                                              readmemory_procread,
                                              NULL);

    printk(KERN_ALERT "readmemory: /proc/"PROC_MEMORY_FILE" creato\n");
    return 0;
}
```

Inizizializzazione e unload del modulo:

```
module_init(readmemory_init_module);
module_exit(readmemory_cleanup_module);
```

Questo modulo permette di leggere una patrte (`segment_size` byte) della memoria a partira da un indirizzo prefissato (`begin_segment`). Questo indirizzo di default è pari a 0xE0000000 il quale corrisponde al primo banco di memoria della flash. Contrariamente a quanto si possa pensare, l'indirisso iniziale della flash non è 0x00000000 perchè a questo punto dell'inizializzazione, ossia quando il kernel è in caricato e in esecuzione, la mappatura della memoria non è più quella fisica, ma qualla virtuale tramite MMU (vedere *arch/cris/arch/README.mm*). Senza entrare nello specifico, basti sapere che la flash con chache bypassata è appunto l'indirizzo 0xE0000000.

Esaminando il modulo si vede che semplicemente crea il file virtuale `/proc/readmemory` accessibile in lettura. Per verificare (e ottenere) il primo segmento della flash occorre, una volta copiato il modulo nella FOX board e caricato, eseguire i comandi:

```
cat /dev/flash0 > flash1
cat /proc/readmemory > flash2
```

quindi i due file appena creati li si confronta per vedere se sono identici (per esempio con *md5sum*), e così dovrà essere.

Nota: questo modulo legge di default i primi 64k di flash, ma in realtà può leggere qualsiasi indirizzo di memoria (almeno in linea teorica perchè per esempio il segmento 0 non può essere letto) in accordo con la tabella della MMU.

B FINDINFILE.C

Questo programma è stato scritto per cercare un file contenuto in un altro. Lo scopo di questo programma è cercare il codice di particolari sezioni critiche per la fase di avvio, per la decompressione del kernel, ma non solo. findinfile richiede in ingresso due parametri che indicano percorso e nome di due file: il primo deve essere contenuto nel secondo.

Questo programma è stato scritto perchè al sottoscritto non è noto se esista un programma già fatto che soddisfi le stesse richieste.

```
#include <stdio.h>
#include <stdlib.h>
FILE *contenuto, *contenitore;
int main(int ac, char **vc)
{
   unsigned int  cp=0, posizione=0;
   unsigned char c1,c2;

   if(--ac<2) {
        printf("ERROR: parametri insufficienti\n"
               "\t%s [contenuto] [contenitore]\n",vc[0]);
        exit(1);
    }
   if(!(contenuto=fopen(vc[1],"r"))) {
        printf("ERROR: file %s non trovato",vc[1]);
        exit(2);
     }
   if(!(contenitore=fopen(vc[2],"r"))) {
        printf("ERROR: file %s non trovato",vc[2]);
        fclose(contenuto);
        exit(3);
     }

   printf("Contenitore:\t%s\nContenuto:\t%s\n",vc[2],vc[1]);

   while(!(feof(contenitore)))
     {
        fseek(contenitore,cp,SEEK_SET); //setta la posizione
        fscanf(contenitore, "%c", &c1);
        cp=ftell(contenitore);  //memorizza l'attuale posizione
        while(!(feof(contenuto)))
          {
             fscanf(contenuto, "%c", &c2);
             if(feof(contenuto))
               goto __EXIT;
             if(c1==c2)
             {
                if(posizione==0)
                  posizione=ftell(contenitore);
             }else
               {
                  rewind(contenuto); //setta a 0 la posizione
                  posizione=0;
                  break;
               }
             fscanf(contenitore, "%c", &c1);
          }
     }
__EXIT:
   fclose(contenuto);
   fclose(contenitore);

   if(posizione>0)
     printf("Trovata posizione di %s in %s a %u\n",
            vc[1],vc[2],posizione);
   return 0;
```

```
}
```

Il flusso del programma prevede di aprire il file "contenitore" leggere un carattere e memorizzare la posizione del puntatore in modo da risettarlo nel caso in cui si esca dal secondo `while`. Il secondo `while` infatti fa la stessa cosa con il file "contenuto", ne estrae un carattere e lo confronta con quello estratto da "contenitore". Se sono uguali si legge il un altro carattere da entrambi i file e così via fino a che tutto il file "contenuto non combacia" con la porzione del "contenitore" e quindi si procede all'uscita. Se i caratteri non coincidono, l'indice di "contenitore" viene riazzerato per poterlo rileggere da capo e si esce dal `while` più interno. A questo punto, come anticipato, l'indice di lettura del file "contenitore" viene settato al vecchio valore e così via. Se non viene rilevato nulla (`posizione==0`), nulla verrà stampato. Per usare il programma è sufficiente lanciarlo da shell.

C LD: Sections And Relocation

Da quanto esaminato nel codice iniziale di un kernel è ormai chiara l'importanza del linker script il quale definisce dove e come posizionare il codice e i dati. Nonostante anche il linker come il compilatore ha delle direttive architecture-dependent (vedere [5, cap5]), la struttura base di uno script per il linker GNU è normalmente comune a tutte le architetture. Quanto segue è fondamentalmente un estratto di quanto si può trovare nella documentazione Red Hat Linux Enterprise (www.redhat.com/doc/manual/) e su quella BSD in particolare quella relativa all'assemblatore e al linker ([4, 5, 6]).

C.1 Sections

Un linker script definisce come le varie aree dell'oggetto (o dell'eseguibile) devono essere organizzate. Queste aree sono dette *sezioni* e sono definite dal comando `SECTIONS`. Un semplice esempio (preso da [5, cap4]) può essere:

```
SECTIONS {
   . = 0x10000;
   .text : { *(.text) }
   . = 0x8000000;
   .data : { *(.data) }
   .bss : { *(.bss) }
}
```

In questo esempio si identificano subito alcune sezioni le quali cominciano con "`.`" (detto *location counter*) tra le quali vi sono `.text` e `.data`; il primo può anche essere chiamato `.code` ed identifica l'indirizzo di memoria a cui si troverà il codice eseguibile, mentre il secondo (`.data`) identifica dove si trova l'area dati. Nell'esempio appena definito si ha in particolare che il codice comincia all'indirizzo 0x10000. Non essendo definita nessuna lunghezza di tale segmento si ha che l'area di codice sarà tutta quella compresa tra 0x10000 e 0x8000000. Ci penserà il linker a posizionare i vari segmenti di codice in quest'area. Segue quindi `.data` che identifica come detto l'area dati, ma che a priori non è dato sapere quanto sia ampia. Ad ogni modo subito dopo essa viene posizionata la sezione `.bss`.

I tipi di sezioni posibili in un linker script sono quindi quattro (vedere [6]):

.text/.data contiene il programma vero e proprio. La sezione eseguibile è la `.text` e non viene normalmente alterata in quanto è spesso condivisa tra processi. `.data` è invece alterabile: in questa sezione sono rilocate per esempio le variabili in linguaggio C.

.bss questa sezione viene azzerata ogni qual volta un programma viene caricato. É normalmente usata per contenere le variabili non inizializzate o come area di storage. Non è necessario però che l'object file o il file di output contengano questa area già azzerata,

ma è necessario solo conoscerne la lunghezza. Infatti `.bss` è stato inventato proprio per eliminare questa area da dai file oggetto e questa area viene creata solo al momento del caricamento del programma.

.absolute quando ld mescola alcune parti di programma (codice e dati), gli indirizzi assoluti rimangono invariati.

.undefined tutte le aree non definite dalle precedenti

Prendiamo ora in considerazione uno script più complesso come per esempio *arch/cris/boot/-compressed/decompressed.ld* già riportato a pagina 10. In questo caso l'indirizzo della sezione `.text` è implicitamente definito nella dichiarazione `MEMORY` la quale definisce un'area di memoria in questo caso di RAM (definendo dram) indicandone l'indirizzo e la massima lunghezza. All'interno di `.text` si osservano ora delle sottosezioni come `_stext` e `_etext`. Queste non sono delle vere e proprie sezioni, ma sono delle etichette (label) che identificano degli indirizzi di memoria utilizzabili all'interno di un programma; un esempio è `_edata` il quale viene definito da *compressed/decompressed.ld* e usato in *compressed/head.S*. Queste etichette sono definite convenzionalmente in un linker script anche se spesso non vengono utilizzate; eccezion fatta per `_etext` che può essere utilizzata da altri compilatori come C/C++ (vedere [5, §4.5.2]). Queste due sottosezioni identificano semplicemente l'inizio e la fine del segmento `.text`. Segue quindi il comando "`> dram`" il quale indica cha tutto il contenuto di `.text` va messo in `dram`. Per come è definito lo script, in questo caso il codice partirà 0x40700000.

É importante notare una cosa: l'indirizzo di memoria definito per ogni sezione è da considerarsi un indirizzo virtuale. Ciò significa che in funzione di ciò che si sta scrivendo (codice di boot/firmware, programma, ecc) occorre sapere a priori se l'unità MMU è stata configurata oppure no; in altre parole occorre sapere se gli indirizzi fisici sono differenti da quelli virtuali.

Interessante è poi il comando `OUTPUT_FORMAT()` che indica al linker in quale formato ottenere il codice finale. Comandi di questo tipo sono moltissimi, tra cui per esempio spicca `ENTRY()` il quale definisce l'entry point di un programma. Per tutti gli altri comandi si rimanda a [5].

Info&Credits

Questo documento è redistribuibile secondo licenza GNU/GPL v2. Chiunque è libero riprodurlo, correggerlo e ampliarlo a patto che il risultato venga reso pubblico secondo quanto indicato dalla licenza. Il testo è quindi potenzialmente in continuo aggiornamento e si invita chiunque a migliorarlo segnalando errori e/o correggendo eventuali parti errate.

revisione 1 Marzo 2008 - Articolo redatto da Calzoni Pietro (aka *Calzo* - calzog @ gmail .com), scritto in LyX 1.4.3 sotto GNU/Linux Slackware 10.2. Si ringrazia la sezione di Azionamenti del dipartimento di Ingegneria Elettrica del Politecnico di Milano e MCM - Energy Lab per aver fornito l'hardware.

Il documento è distribuito per default in formato pdf. Altri formati (sorgenti LyX, LaTeX, DVI, ecc) sono disponibili su ruchiesta se non reperibili su internet. Il documento è scaricabile dal sito dell'associazione culturale LUGMan (Linux Users Group Mantova - info @ lugman .net) www.lugman.org nella sezione *Documentazione*.

Riferimenti bibliografici

[1] www.acmesystems.it - Sito ufficiale per reperire il materiale relativo alla FOX board (manuali, software, kenrel, ecc)

[2] www.axis.com - AXIS ETRAX 100LX Designer's Reference - 09/02/2006 - Manuale del microprocessore ETRAX100LX

[3] www.axis.com - AXIS ETRAX 100LX Programmer's Manual - Manuale di programmazione per ETRAX100LX

[4] http://www.redhat.com/docs/manuals/enterprise/RHEL-3-Manual/gnu-assembler/index.html (pdf version *rhel-as-en.pdf*) - Manuale sull'assemblatore GNU

[5] http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/gnu-linker/index.html (pdf version *rhel-ld-en.pdf*) - Manuale sul linker GNU

[6] http://docs.freebsd.org/info/as-all/as-all.info.bss.html (http://docs.freebsd.org/info/as-all/as-all.pdf) - Chiarissimo manuale sull'assemblatore e il linker GNU

Altri Link utili Il presente documento è estratto da un più grande documento (mai pubblicato) che presentava il processo di boot anche nei sistemi x86 di Linux. I link che seguono possono integrare quinti il discorso relativo al boot di Linux su sistemi differenti dalla FOX.

- www.linux.it/~rubini/docs/boot-it/boot.html - Il Boot di Linux - Alessandro Rubini - giugno 1997 - articolo comparso anche su Linux Journal
- http://tldp.org/HOWTO/Linux-i386-Boot-Code-HOWTO/index.html
- http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/pdf/Linux-i386-Boot-Code-HOWTO.pdf
- Documentation/i386/boot.txt - THE LINUX/I386 BOOT PROTOCOL - H. Peter Anvin - Documentazione fornita con i sorgenti Linux, tipicamente in */usr/src/linux/Documentation/i386/*
- www.acpi.info - ACPI Specification - ACPIspec20.pdf e superiore
- http://www.uruk.org/orig-grub/mem64mb.html - Definizione degli interrupt 15 con funzione E820H, E801H, 88H
- http://www.ctyme.com/rbrown.htm e http://www.ctyme.com/intr/int.htm - Ralf Brown's Interrupt List - Probabilmente le più vasta e completa descrizione relativa a tutti gli interrupt per x86