

# *Sviluppo in ambiente Embedded*

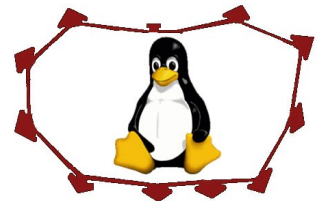
Tra Driver e Applicazioni.

Tecniche di sviluppo.

di

Rodolfo Giometti

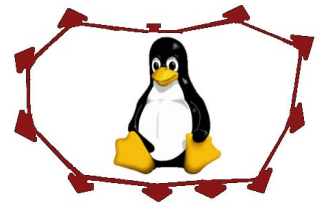
`<giometti@enneenne.com>`



# *Cosa e' un embedded*

Quando si parla di «sistema embebbed» bisogna entrare nell'ordine di idee che abbiamo a che fare con un sistema che molte volte ha risorse limitate.

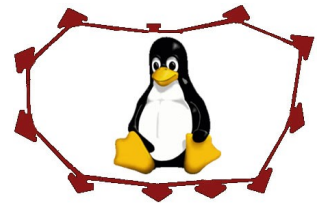
Lavorare con risorse limitate impone dei limiti allo sviluppo delle nostre applicazioni.



# *Quali problematiche*

Quando devo sviluppare una applicazione su di un sistema embedded devo tener presente che, a causa delle limitate risorse, non posso utilizzare le stesse tecniche di programmazione dei desktop.

Spesso la piattaforma di lavoro ha risorse molto limitate che rendono impossibile utilizzarla come macchina di sviluppo il che può portare ad avere la condizione che le due piattaforme sia diverse.

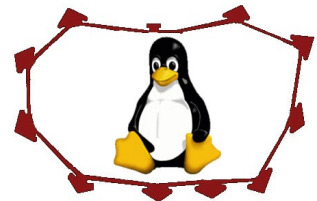


# *Cosa occorre*

In questi casi occorre quindi che sul sistema «host» ci sia installata una «toolchain» che permetta di compilare applicazioni per la macchina «target».

HOST = macchina di sviluppo

TARGET = macchina di lavoro

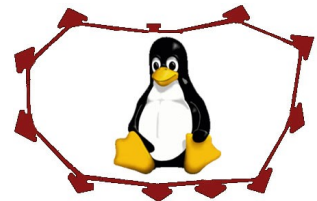


# La «*toolchain*»

La «*toolchain*» non è altro che una serie di applicazioni che, girando sull'host, permettono di compilare applicazioni per il target.

Generalmente fanno parte della *toolchain*: `gdb`, `ld`, `glibc` e le `binutils`.

Nel caso volessimo realizzare anche applicativi grafici o altre applicazioni che necessitano di librerie particolari, queste vanno aggiunte alla *toolchain*.

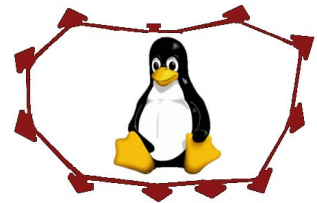


# // gcc

```
zaigor:$ make CFLAGS=-Wall test  
cc -Wall test.c -o test
```

```
zaigor:$ make CC=mipsel-linux-gcc \  
CFLAGS=-Wall test
```

```
mipsel-linux-gcc -Wall test.c -o test
```

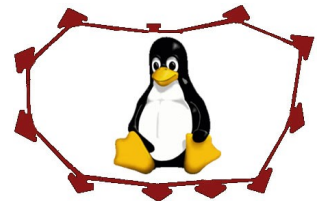


# *La distribuzione*

Proprio per le ridotte capacità di un sistema embedded è molto probabile che su di esso non possiamo utilizzare una distribuzione «classica».

In questi casi occorre quindi utilizzare una distribuzione embedded ad hoc.

Per ottenere una distribuzione ad hoc occorre che ci armiamo di una buona toolchain, di tanta pazienza, e ci mettiamo a ricompilare tutte le applicazioni!



# *Il «port» di una applicazione*

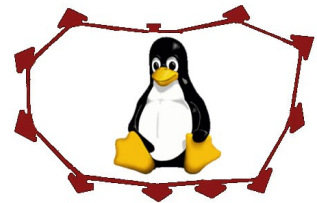
Effettuare il port di una applicazione non è difficile.  
Basta che la toolchain sia sufficientemente completa!

Generalmente si usa:

```
$ ./configure --host=arm-linux
```

o

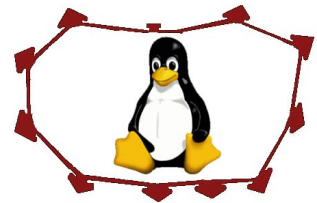
```
$ CC=arm-linux-gcc make
```



# *Il «port» del nucleo*

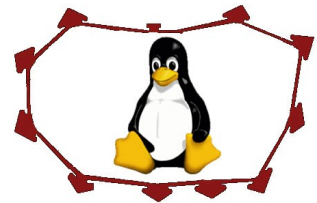
Ma avere una toolchain per ricompilarci le applicazioni per ottenere una distribuzione ad hoc non basta. Occorre anche fare in modo che Linux (il nucleo) si adatti al nostro hardware.

E qui iniziano i dolori...



# *Il «port» del nucleo - 2*

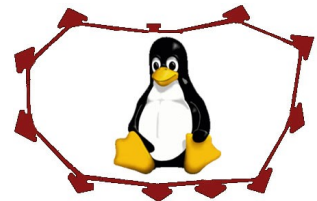
Quando dobbiamo realizzare il port di Linux per un sistema embedded quello che dobbiamo verificare è lo stato di supporto della architettura e quindi dell'hardware che abbiamo on-board.



# *Il supporto dell'architettura*

Il supporto dell'architettura ci serve per avere tutti quei meccanismi interni che poi ci permetteranno di gestire i driver per l'hardware (es. gestione delle interruzioni, dello schedulatore, della memoria, ecc.).

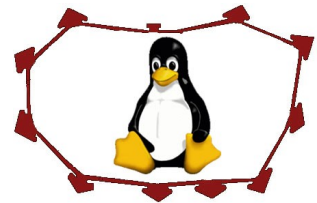
Senza questo supporto, effettuare il port di Linux per un certo sistema embedded diventa molto difficoltoso!



# *Il supporto delle periferiche*

Per quanto riguarda invece gestire l'hardware, occorre innanzi tutto verificare quali periferiche sono presenti sulla nostra scheda e che dobbiamo supportare.

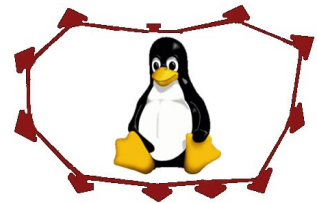
Molti sistemi embedded usano dei particolari processori denominati SOC (system on chip) i quali, oltre ad implementare la CPU, hanno anche tutta una serie di periferiche già pronte all'uso.



# *Stessa architettura – diverse periferiche*

Quando si realizza il supporto Linux per una architettura non i386 occorre tener presente che pur avendo la stessa architettura, due macchine, possono avere una configurazione hardware diversa (diversa memoria, chip flash, LCD, ecc.).

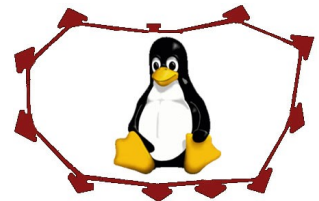
Nel mondo i386 le cose sono più semplici perché molto più «standardizzate». Ma come si cambia architettura questo non è più vero!



# *La fase di boot*

In questa fase generalmente non occorre fare quasi nulla poiché è direttamente implementata dal supporto dell'architettura.

Parte rilevante è comunque il *bootloader* che ha il compito di inizializzare il sistema in modo che Linux possa inicializzarsi (es. configurazione preliminare del memory controller per le RAM/Flash).

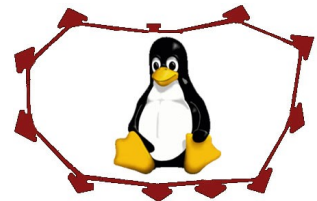


# *La fase di setup*

Questa fase è molto importante perché è una di quelle che dobbiamo definire noi che realizziamo il port.

Questa fase viene chiamata dalla fase di boot e serve per inizializzare le periferiche del sistema (GPIOs, timer di generazione dei clock interni, periferiche interne da abilitare).

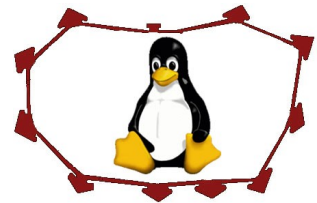
Generalmente qui si interpretano anche gli eventuali argomenti passati al nucleo dal bootloader.



# *La fase di setup - 2*

Generalmente tutti i supporti delle diverse architetture mettono a disposizione diverse funzioni o strutture dati di inizializzazione che devono poi essere scritte o riempite da chi realizza il port del sistema.

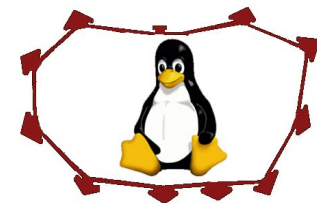
```
void __init board_setup(void);  
au1xxx_irq_map_t au1xxx_irq_map[];
```



# *Il supporto delle periferiche*

Come già accennato molti sistemi montano processori SOC. Per questi sistemi le cose sono più semplici perché l'hardware è sempre lo stesso. Per le altre periferiche invece bisogna vedere di volta in volta.

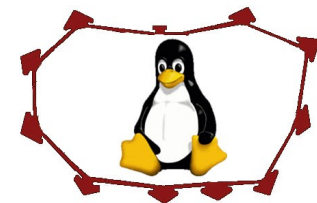
In generale, comunque, dobbiamo verificare lo stato del supporto software per ogni periferica del sistema, ed eventualmente scrivere tutto da zero o integrare quello che c'è.



# *Il supporto delle periferiche – fb*

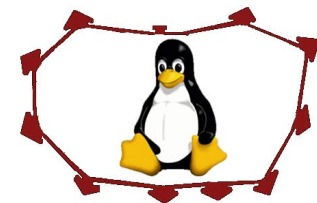
```
#if defined(CONFIG_FB_AU1100)
    argptr = prom_getcmdline();
    if (strstr(argptr, "video=") == NULL)
#if defined(CONFIG_MIPS_WWPC_ETH)
    strcat(argptr,
        " video=au1100fb:panel:Toppoly");
#else
    strcat(argptr,
        " video=au1100fb:panel:Hitachi");
#endif
#endif

au_writel(au_readl(SYS_PINFUNC) |
    SYS_PF_LCD, SYS_PINFUNC);
```



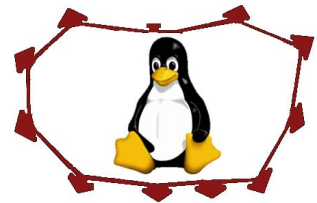
# *Il supporto delle periferiche – fb*

```
/* WWPC Toppoly 320x240 TFT panel */  
[7] = {  
    .name = "Toppoly_TD035STEB1",  
    .xres = 320,  
    .yres = 240,  
    .bpp = 16,  
    .control_base = 0x000480EA,  
    .horztiming = 0x00500CEF,  
    .verttiming = 0x000C013F,  
    .clkcontrol_base = 0x00028001,  
},
```



# *Il supporto delle periferiche – serial*

```
/* Enable ttyS1 for gps */  
au_writel(UART_MOD_CTRL_CLKENA,  
          UART1_ADDR+UART_MOD_CNTRL);  
au_writel(UART_MOD_CTRL_ENABLE |  
          UART_MOD_CTRL_CLKENA,  
          UART1_ADDR+UART_MOD_CNTRL);  
/* Enable ttyS2 for bluetooth */  
au_writel(UART_MOD_CTRL_CLKENA,  
          UART3_ADDR+UART_MOD_CNTRL);  
au_writel(UART_MOD_CTRL_ENABLE |  
          UART_MOD_CTRL_CLKENA,  
          UART3_ADDR+UART_MOD_CNTRL);
```

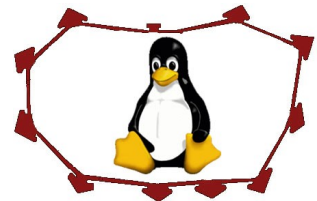


# *Il supporto delle periferiche – serial*

```
static int __init serial8250_init(void)
{
    int ret, i;
    printk("Serial: Au1x00 driver\n");

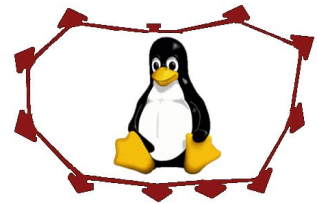
    uart_register_driver(&s_reg);
    serial8250_register_ports(&s_reg);

    return ret;
}
```



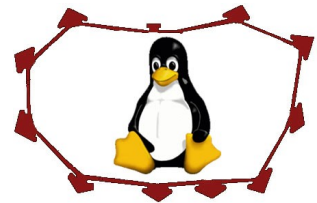
# *Il supporto delle periferiche – flash*

```
struct mtd_partition wwpc_part[] = {
    {
        .name = "rootfs",
        .offset = 0,
        .size = WWPC_ROOTFS_SIZE,
    }, {
        .name = "u-boot",
        .offset = MTDPART_OFS_APPEND,
        .size = WWPC_UBOOT_SIZE,
    }, {
        .name = "kernel",
        .offset = MTDPART_OFS_APPEND,
        .size = MTDPART_SIZ_FULL,
    }
};
```



# *Il supporto delle periferiche – keyboard*

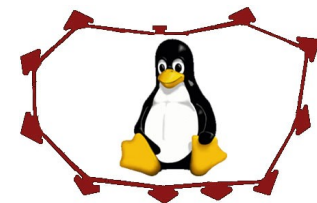
```
void wwpc_key_poll (...)
{
    ...
    switch (wk->type) {
    case TYPE_GPIO1 :
        t = au_readl(0xB1900110); break;
    case TYPE_GPIO2:
        t = au_readl(0xB170000C); break;
    }
    if (( (t & (1 << wk->bit)) > 0) ^ wk->negate)
        input_report_key(&dev, wk->key, 1);
    else
        input_report_key(&dev, wk->key, 0);
    input_sync(&dev);
    wwpc_key_set_timer();
}
```



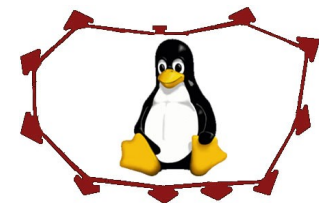
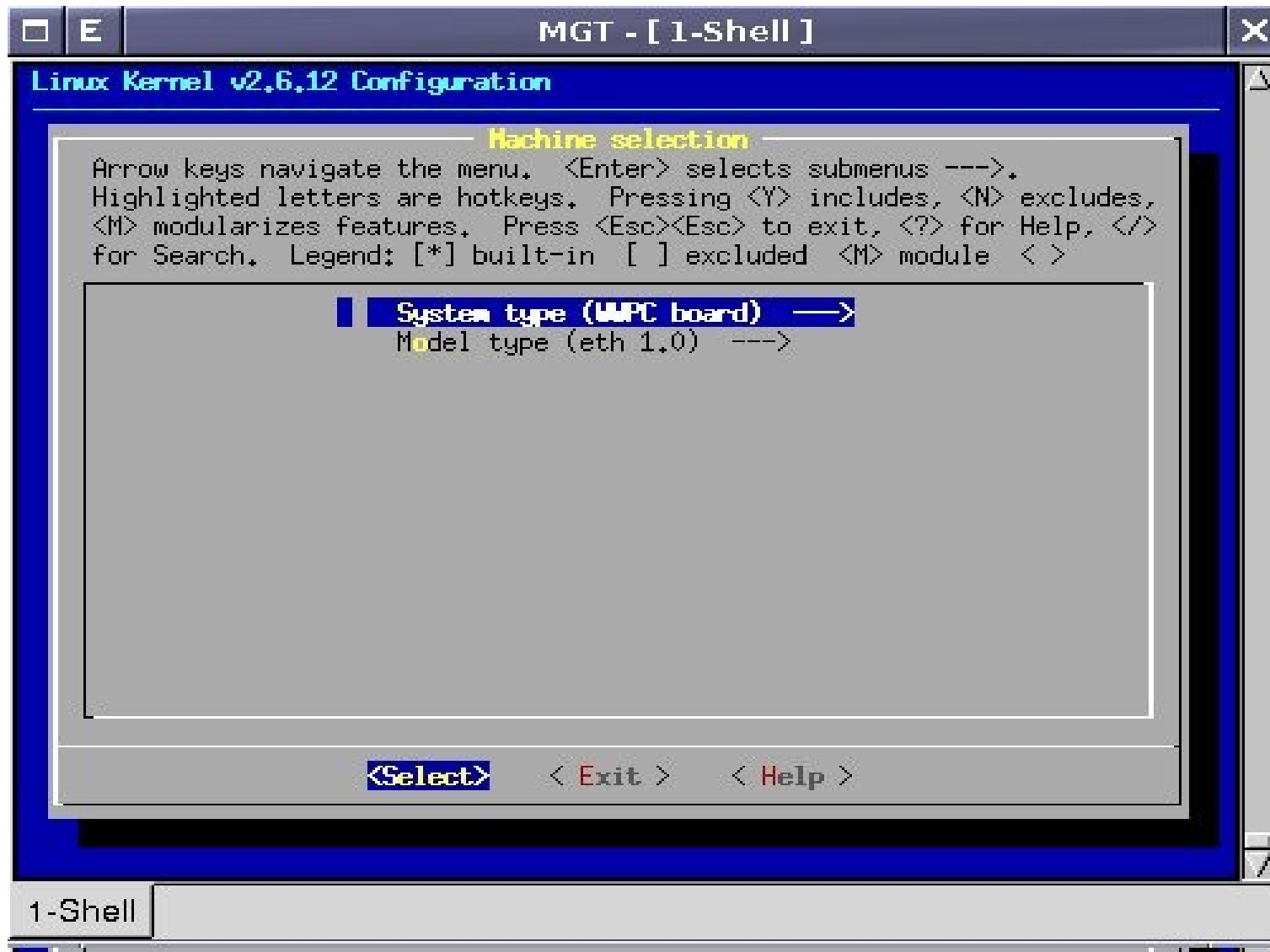
# *I file di configurazione*

Linux ha un sistema di configurazione molto semplice da utilizzare e/o modificare.

Quando realizziamo il port di un sistema embedded è molto importante fornire all'utilizzatore del sistema un tool di configurazione potente che si preoccupi lui di selezionare l'hardware on-board a seconda della scheda selezionata.



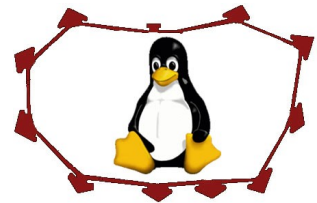
# I file di configurazione – machine sel.



# *I file di configurazione – misc dev*

```
obj-$(CONFIG_SYSCTL) += sysctl.o
obj-$(CONFIG_SND_AU1X00) += ext-ampl.o

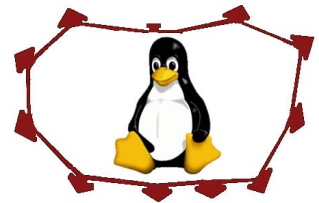
ifdef CONFIG_MIPS_WWPC_EXA
obj-$(CONFIG_APM_EMU) += apm-emu.o
endif
ifdef CONFIG_MIPS_WWPC_ETH
obj-m += ldisc.o
endif
```



# *Tecniche di sviluppo*

Come già detto lo sviluppo del software per il nostro sistema embedded viene fatto sul sistema host.

Una volta *cross-compilata* la nostra applicazione questa viene poi passata al target per essere eseguita in modo da verificarne il funzionamento.

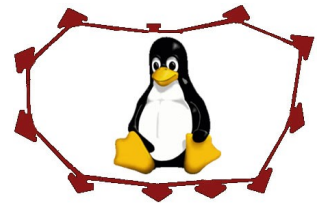


# *Tecniche di sviluppo - nucleo*

Quando sviluppiamo il nucleo, una volta effettuate le nostre modifiche, dobbiamo passarlo al target in modo che questo lo esegua per verificarne la bontà.

Per fare questo possiamo ottenere l'immagine del nucleo stesso e quindi trasferirla nella memoria di massa del target (di solito la memoria flash).

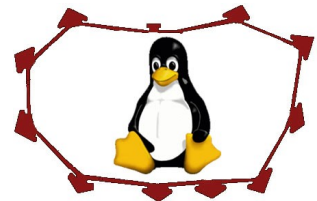
Questa tecnica, seppur sempre efficace, è molto «time consuming».



# *Tecniche di sviluppo – TFTP*

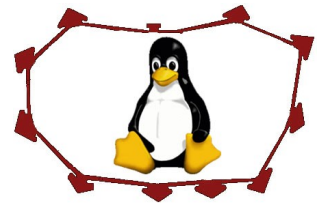
Quando il costruttore mi interpella prima (quasi mai) gli suggerisco sempre di aggiungere una scheda ethernet; o on-board o sulla scheda di sviluppo (*developing board*).

Questo perché permette di utilizzare il protocollo TFTP per caricare il nucleo in RAM riducendo drasticamente e tempi di test!



# *Esempio TFTP*

```
setenv ipaddr 192.168.32.24
setenv serverip 192.168.32.254
setenv netmask 255.255.255.0
setenv kernel_addr 81000000
setenv kernel_file wwpc/vmlinuz-2.6.12
tftp $(kernel_addr) $(kernel_file)
```

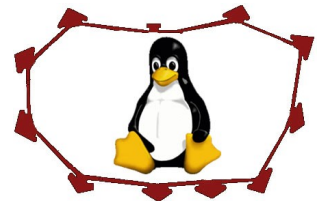


# *Tecniche di sviluppo – filesystem*

Una volta che abbiamo verificato che il nucleo è più o meno stabile, possiamo passare al test del filesystem e delle applicazioni *userland*.

Anche in questo caso possiamo creare un filesystem minimale da trasferire poi nella memoria di massa del target (flash).

Anche questa soluzione, seppur sempre efficace, è «time consuming».

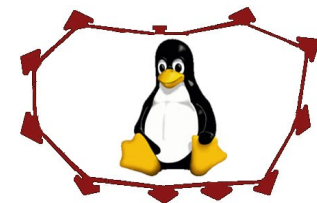


# *Tecniche di sviluppo – NFS*

Un metodo per ridurre il tempo di test è allora quello di utilizzare il protocollo NFS per permettere al target di utilizzare un filesystem che in realtà è sul sistema host.

Anche in questo caso è fondamentale la presenza della ethernet.

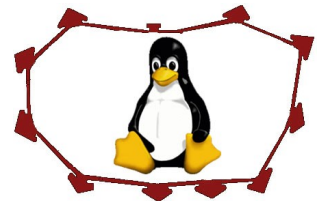
Questa tecnica permette addirittura di utilizzare un distro quale Debian su di un sistema embedded!



# Esempio NFS

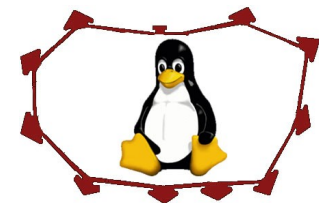
```
setenv rootpath /develop/mipsel/debian
setenv bootargs $(bootargs)
root=/dev/nfs rw
nfsroot=$(serverip):$(rootpath)
ip=$(ipaddr):$(serverip)::$(netmask):
$(hostname):$(netdev):off
```

```
/develop/embedded/mipsel/distro/debian
192.168.32.24 (rw,no_root_squash, sync)
```



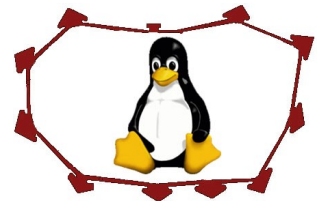
# WWPC!

Questo sistema è pensato per avere un'alta connettività e per lasciare il più possibile le mani dell'utilizzatore libere.



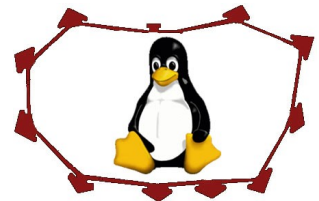
# *Le caratteristiche hardware principali*

- CPU AMD AU1100 (mips)
- 64 MB RAM – 32 MB Flash
- TFT 320x240 65K colori con touch screen
- Wi-Fi – Bluetooth – USB - IRDA
- Secure Digital (o Compact Flash)
- Audio stereo AC97 - Antenna GPS
- Accelerometro biassiale – sensore luminosità ambientale
- Doppia batteria (di cui una ricaricabile) da 1050mA/h



# *Le caratteristiche software principali*

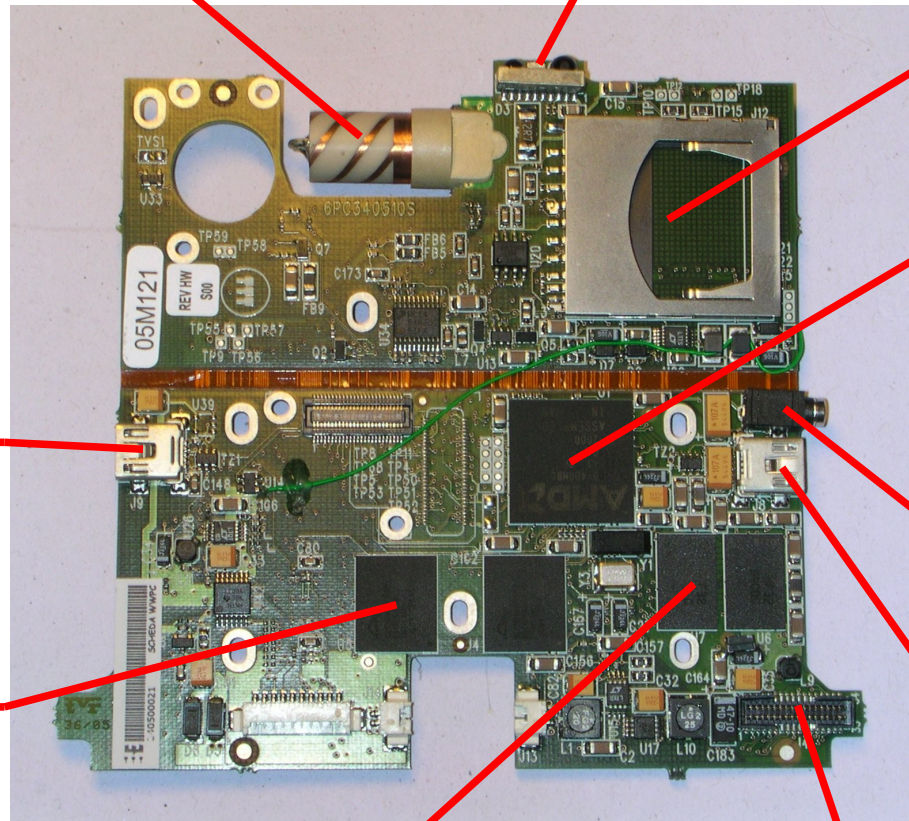
- Linux 2.6.13 (mips)
- Ambiente GNU basato su busybox + altri tool
- Ambiente grafico X-Window + GPE
- Toolchain di sviluppo basata su prodotti GNU (gcc, glibc, ecc.)



# *Come è fatto dentro*

Antenna GPS

Porta IRDA



Slot Secure Digital

CPU

Porta USB

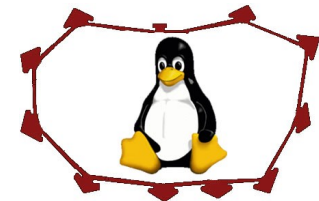
Presa cuffie

Flash

Porta USB

RAM

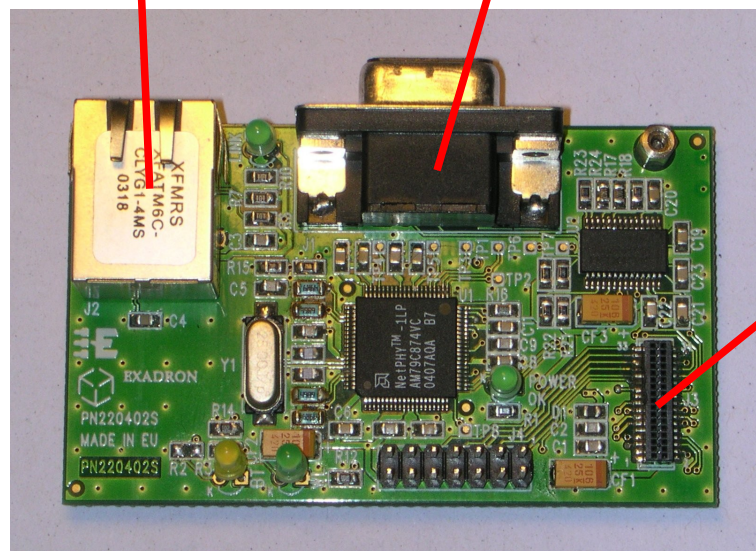
Porta Debug



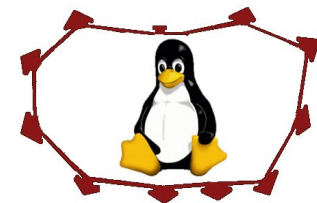
# La scheda di DEBUG

Porta ethernet

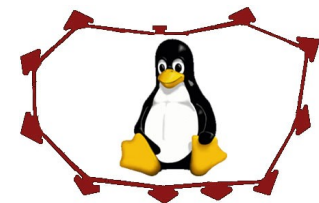
Porta seriale RS-232



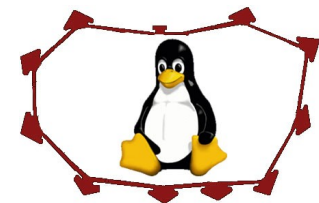
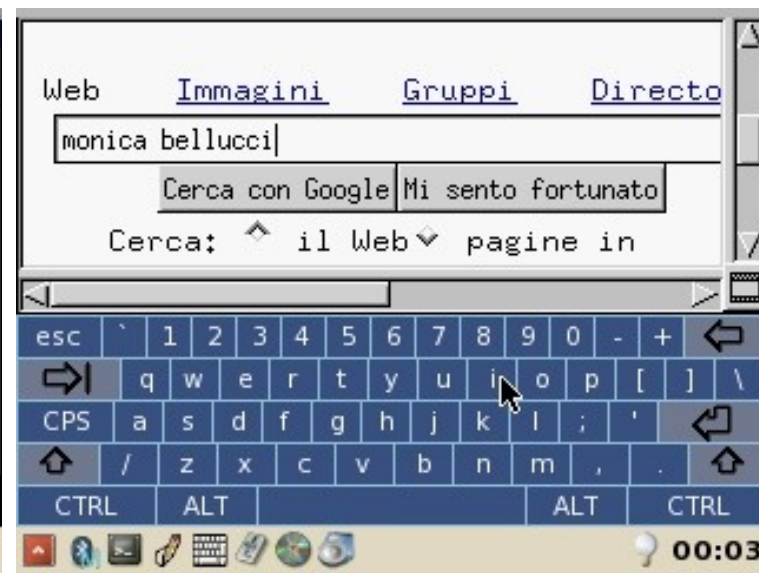
Porta di debug



# L'ambiente grafico - 1



# L'ambiente grafico - 2



# L'ambiente grafico - 3

